OCTOBRE / NOVEMBRE / DÉCEMBRE 2011 N°4

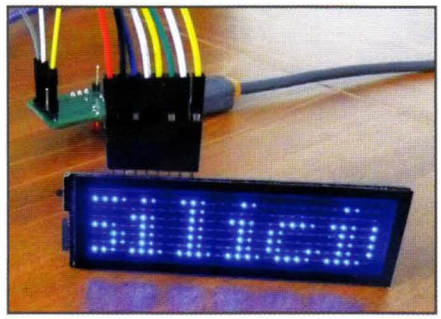
Open Silicium

MAGAZINE
INFORMATIQUE
OPEN SOURCE
EMBARQUÉ
ÉLECTRONIQUE

LE MAGAZINE DE L'OPEN SOURCE POUR L'ÉLECTRONIQUE & L'EMBARQUÉ

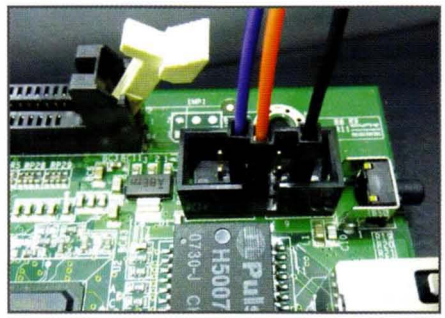
HACK / BADGE

Un firmware alternatif permet de transformer un gadget en afficheur série p.67



NAS / RAID

Renaissance d'un NAS LaCie RAID 4 disques sous Debian GNU/Linux p.88



AFFICHAGE / LED

Pilotez 8 afficheurs 7 segments et plus avec trois fils p.6

USB / NOTIFICATION

Apprenez à contrôler des périphériques USB sans pilotes avec la libUSB p.73

GESTION / CLAVIER

Gérez directement et indépendamment les entrées clavier/souris sous GNU/Linux p.47

LCD / DETECTION

Une méthode simple et efficace pour gérer la détection de vos afficheurs LCD p.11

DEBUG/BINAIRES

Utilisez ld_preload pour changer le comportement des binaires sans les modifier p.56

FPGA / ACTEL

Prototypage économique sur FPGA IGLOO d'Actel/ Microsemi à très faible consommation p.18

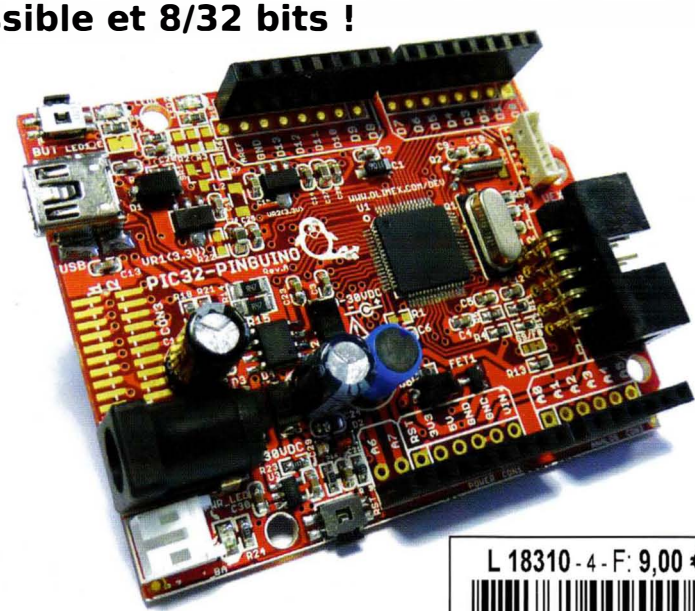
MICROCONTRÔLEUR

LES PIC AUSSI ONT LEUR ARDUINO !

Pinguino

Communautaire, économique, open source, accessible et 8/32 bits !

p.36



L 18310 - 4 - F: 9,00 € - RD

France Métro : 9 € / Belgique - Luxembourg : 9,50 € / Suisse : 14 CHF / DOM : 9,90 € / CAN : 15,50 $CA / N. CAL/S : 1200 CFP / POL/S : 1300 CFP

JAVA

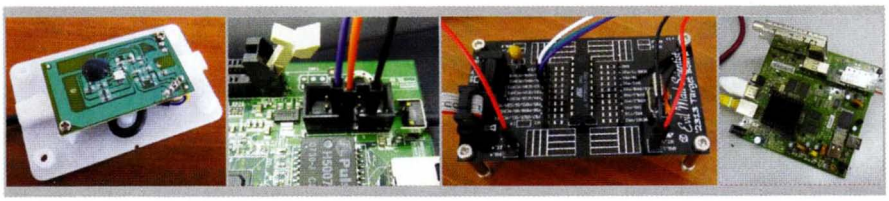
RENDEZ VOS DÉVELOPPEMENTS MULTIPLATEFORMES !

LMHS 56
Actuellement en kiosque !

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 28 OCTOBRE 2011 ET SUR :

www.ed-diamond.com

SOMMAIRE N°4



Open Silicium Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

B.P. 20142 – 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 – **Fax** : 03 67 10 00 21
E-mail : lecteurs@opensilicium.com
Service commercial : abo@opensilicium.com
Sites : www.opensilicium.com – www.ed-diamond.com

Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Secrétaire de rédaction : Véronique Wilhelm
Réalisation graphique : Kathrin Troeger

Responsable publicité : Valérie Fréchard
Tél. : 03 67 10 00 27 / v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : VPM Druck Rastatt / 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 : Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, N° ISSN : 2116-3324
Commission paritaire : K90 839

Périodicité : Trimestriel
Prix de vente : 9 €

ÉDITO

Mais pourquoi ?

Un petit nombre de lecteurs nous ont fait dernièrement part de leur surprise quant à la faible quantité d'articles traitant de Windows et d'applications Windows, pourtant en quantité raisonnable dans le milieu de l'électronique. C'est une bonne question et la
à une paire de mots (ou un, selon comment on l'écrit) : open source.

Les applications Windows, tout comme le système lui-même, ne sont ni ouvertes, ni en logiciel libre. Au-delà d'une certaine forme d'idéologie, cela pose un véritable problème culturel lorsqu'on navigue dans les eaux claires de l'open hardware et de l'open source. Bien sûr, avoir une machine Windows n'est pas inu
la situation, le matériel, la plateforme ou les applicatifs utilisés, nous n'avons tantôt pas le choix. Prenons le cas du notificateur de mails lumineux qui est traité dans les pages qui suivent. Sans une plateforme logicielle et un système adéquat il devient difficile, voire impossible, d'arriver à écrire le moindre code permettant de rendre ce matériel utilisable avec Mac OS X, un BSD ou GNU/Linux. Windows a été à un moment ou un autre indispensable à un développeur souhaitant faire de l'open source, car analyser un protocole ne relève pas de la divination (normalement).

Cela fait bien longtemps maintenant que l'utilisation quotidienne ou même répétitive d'un système Windows n'est plus qu'un lointain souvenir pour moi. De ce fait, étant plus naturellement à l'aise avec un système UNIX, il me devient difficile de comprendre pour quelles raisons certains utilisateurs ne *switchent* pas ou n'inversent pas leur répartition temps GNU/Linux vs temps Windows. Bien entendu, comme tout le monde (dans ma tranche d'âge), j'ai commencé sous MS/DOS, puis Windows 3.1/3.11 et enfin Windows 95/98. Je me souviens que, déjà à cette époque, ce sentiment de frustration face à un système si peu ouvert ...

suite page 4

ÉDITO

...et si peu techniquement documenté était très présent. On avait beau creuser et chercher, il arrivait toujours un moment où on se retrouvait dans une impasse. Et plus on avançait dans le temps, plus je regrettais MS/DOS (si, si). En ce temps, il arrivait inéluctablement un moment précis où le système vous rappelait qui étaient les développeurs, qui vendait la licence et qui devait se borner à utiliser le système ou l'application comme l'éditeur l'avait décidé. Plus que de la frustration, c'est un sentiment d'oppression qui m'envahissait.

Autant vous dire qu'aujourd'hui, je suis obligé de me retrouver face à un système propriétaire (ou semi-propriétaire) pour éprouver les mêmes sensations. Chose qui n'arrive plus, heureusement, que très rarement. La plupart du temps par obligation et parfois par hasard (cf. l'article sur le Toshiba Places dans ce numéro).

Ainsi, au fil du temps, on finit par s'habituer à une certaine richesse dans son environnement informatique. L'accoutumance au confort, sans doute. Les petits secrets du système ne sont plus vraiment inaccessibles, mais se troquent simplement contre du temps et de l'énergie. Aujourd'hui, l'ouverture des sources et les licences garantissant cette liberté de savoir et d'apprendre ont prouvé leur valeur et leurs avantages. Ainsi, cette méthode de développement, clairement plus efficace techniquement, s'étend petit-à-petit au code des pilotes et des microcontrôleurs mais aussi aux circuits, à la documentation et à bien d'autres choses dont la fermeture et l'opacité étaient fermement ancrées dans les habitudes. L'open hardware prend son envol doucement, convainquant des acteurs du milieu à force de démonstrations et de tentatives réussies.

C'est un milieu encore très frileux à l'idée de partager ouvertement quelque chose et les choses progressent sous la forme de petits essais bien mesurés et prudents. Bien sûr, certains misent plus gros que d'autres mais, globalement, les choses semblent aller dans la bonne direction.

C'est très satisfaisant pour une personne qui, comme moi, tire déjà pleinement avantage du logiciel libre aussi bien d'un point de vue personnel que professionnel. Open Silicium est né du terreau de l'open source et du logiciel libre avec, comme objectif, de vous faire découvrir que cette passion qui anime déjà une énorme quantité de développeurs est transposable aux matériels, aux plateformes de développement pour microcontrôleurs et au monde de l'embarqué en général. Et ça marche ! Comment pourrait-il en être autrement, puisque la frontière entre le logiciel et le matériel est de plus en plus mince ? L'époque où l'électronique se résumait à du modélisme ferroviaire à base de circuit logique, à du contrôle d'aquarium à grands coups de LDR et à la fabrication d'ampli en tous genres, est révolue. Il faut maintenant compter avec l'électronique numérique et des composants plus puissants que nos premiers PC, se programmant en C (langage de bas niveau aujourd'hui, mais de haut niveau il n'y a pas si longtemps) et coûtant moins qu'une laitue.

Seulement voilà, comment pourrions-nous légitimement démontrer que le partage des connaissances, l'exploration, le hack et l'open source sont des éléments bénéfiques pour vos créations et votre passion, si nous ne faisons rien pour vous éviter d'utiliser quelque chose de totalement fermé (j'allais dire « obscur ») ? En version plus épurée : comment voulez-vous correctement profiter de ces bienfaits si le système qui vous sert d'outil principal et de socle est développé dans une direction radicalement opposée ? Cela revient un peu à construire un château sur du sable...

Il y a là quelque chose qui serait contre-productif, sinon malsain, à détailler par exemple dans ce numéro, le développement d'un support Windows pour le notificateur de mails dont je vous parlais précédemment. D'une part parce qu'un support logiciel existe déjà et d'autre part pour deux raisons :

- Le code aura beau être en logiciel libre, tout comme le compilateur que vous utiliserez (bonne chance déjà sous Windows pour cela !) et même l'éditeur de texte, au final vous aurez quoi ? Un binaire qui ne fonctionne qu'avec un système propriétaire et donc avec des couches inférieures dignes des plus profonds abysses de par leur opacité. Peu importe jusqu'où vous descendrez dans le système, à un moment ou un autre, vous serez dans le noir. Chose qui n'arrive pas avec un système open source.
- C'est tout bonnement plus difficile. La difficulté peut être un jeu ou un défi, mais uniquement à condition qu'ils en vaillent le coup. Personnellement, je ne vois pas l'utilité de me compliquer la vie avec un système que je ne maîtrise pas et dont je ne pourrai jamais comprendre le fonctionnement interne. C'est une croisade sans espoir.

À ces deux points peut être opposé un argument en faveur du système propriétaire majoritaire, mais cela revient à remettre en cause l'objet et le moteur de la présente publication. Cet argument est la popularité de votre application ou de votre solution. Mais la motivation est alors bien différente. Il ne s'agira plus de comprendre, partager et utiliser, mais de proposer au plus grand nombre le fruit de votre travail. C'est là, généralement je pense, la principale raison de développer pour Windows. On vise la masse, on s'attaque à un marché ! Une telle manœuvre n'a plus rien à voir avec la seule passion, c'est tout autre chose.

ÉDITO

L'illustration peut en être faite en comparant cela à n'importe quelle activité passionnelle artisanale. Prenons par exemple l'artisan fromager. Il y a le passionné qui fait ses fromages dans le respect des traditions et éventuellement, essaie quelques variations basées sur l'expérience de ses prédécesseurs et son imagination. Voici la personne qui va produire de bons petits munsters, bien goûteux et bien odorants, mais pas nécessairement du goût de la majorité des consommateurs. A celui-ci, nous pouvons comparer la PMI qui va produire des produits laitiers *Babybel compliant*. Difficile de parler d'amour de l'art et de passion culinaire. Il en va de même entre l'ébéniste et IKEA, ou encore entre le restaurateur de voitures anciennes et le vendeur de kits de tuning vert fluo. On ne parle tout simplement pas de la même chose.

Pour en revenir à nos moutons, tout comme l'utilisateur et développeur de codes open source se dirigeant vers l'embarqué et l'électronique, l'amateur d'électronique gagnera à globaliser sa démarche. Si nous prenons le problème dans le sens inverse, je ne peux que plaindre la personne qui, à titre ludique/privé, utilise Windows mais se passionne pour Arduino, la robotique, la domotique, le hack, la réalité augmentée et j'en passe. Je ne vois là rien d'autre qu'une personne qui se handicape dans la course vers la satisfaction de sa passion.

Bien entendu, l'utilisation d'un système comme GNU/Linux n'est pas de tout repos. Inutile de dépeindre les choses sous leur plus beau jour. Parmi les difficultés nous avons, par exemple, le support quasi-constant de Windows sous la forme d'applications gratuites directement fonctionnelles. Un exemple simple est le système Plugwise permettant le contrôle et la mesure de consommation électrique sous la forme d'un réseau maillé sans fil de prises murales. Ce système est piloté par un ordinateur équipé d'une clef USB appelée Stick Plugwise (adaptateur ZigBee/s
par l'application Plugwise Source qui, contrairement à ce que son nom peut suggérer, est un logiciel propriétaire. Utiliser ce système avec une machine GNU/Linux ou n'importe quel système non Windows est un vrai sport. Les informations existent, les codes aussi, mais vous devrez prendre le temps de comprendre et d'apprendre. Nous ne sommes pas dans le mode « Installer Suivant Suivant Suivant Terminer Utiliser ». Il en va de même pour les IDE pour microcontrôleurs, les logiciels pour analyseurs logiques, les stations météo, etc.

Pour résumer, deux voies sont possibles :

- Avoir tout, tout de suite et facilement : utiliser Windows, installer les applications clef en main et obtenir un résultat en un instant. Mais, par la suite, se retrouver bloqué. Exemple, pour le logiciel Plugwise Source, quid si je veux remonter mes mesures de consommation sur le Web ou encore les afficher sur un petit écran tactile ? Exporter en XLS et composer un mini PC sous Windows ? Non, sérieusement, vous voilà coincé !

- Chercher, étudier, comprendre, creuser, tester, hacker, mériter et obtenir. Mais ensuite, plus rien n'est impossible. Le système Plugwise en place chez moi, qui n'est pas traité dans ce numéro, car je n'en ai pas fini avec lui, pour l'heure peut m'indiquer la consommation en temps réel et activer/désactiver chaque prise murale. Les graphes de consommation ne sont pas pour tout de suite et cela nécessitera du travail et du code. Mais je ne suis pas bloqué, je progresse et j'accumule des informations. Je ne suis limité que par mon entêtement à comprendre. Ceci avec le soutien de mon système d'exploitation, d'outils open source et mieux encore, d'autres utilisateurs curieux. Je ne travaille pas contre mon système mais avec.

La voie que j'ai choisie depuis longtemps et que d'autres ont choisie comme moi est également celle qui dicte la ligne rédactionnelle de ce magazine. Celle-ci n'est ponctuée d'éléments propriétaires que si cela constitue une nécessité absolue ou qu'il s'agit d'une concession raisonnable. Exemple : installer/initialiser/tester le système Plugwise sous Windows pour ensuite l'exploiter sous GNU/Linux. Dans le domaine des serveurs et des postes de travail, les concessions sont exceptionnelles grâce au travail de la communauté open source et la maturité des systèmes en logiciel libre. Dans le monde d'Open Silicium (embarqué, électronique numérique, hack, etc.), le ponctuel est plus présent mais les bénéfices, à terme, sont tout aussi importants que pour les serveurs et le desktop, voire supérieurs si l'on parle de bénéfices et de développements personnels.

Pour conclure, j'enfoncerai une dernière fois le clou avant de vous laisser lire toutes les bonnes choses dans les pages qui suivent. Les XP que vous gagnerez avec l'UNIX open source de votre PC seront valables pour votre système embarqué. Il en va de même concernant l'habitude de disposer de sources, de documentations et d'explications. Vous développerez les mêmes réflexes, les mêmes automatismes de valeur. « Use the source, Luke » est maintenant valable pour le logiciel comme pour le matériel. C'est cette idée qui concrétise Open Silicium, qui remplit ses pages et qui motive son édition.

Denis Bodor

PILOTER UN AFFICHEUR 8 CHIFFRES DFROBOT AVEC UN AVR

par Denis Bodor

Leds, afficheurs 7 segments, afficheurs alpha-numériques... toutes ces petites choses lumineuses possèdent incontestablement un petit quelque chose qui les rend tout bonnement fascinantes. Ces témoins et indicateurs sont un peu partout autour de nous, ce qui ne nous empêche pas de chercher à en ajouter, d'autant plus si on leur donne une véritable utilité au bout du compte. Bien entendu, ceci n'est vrai qu'en gardant le sens des proportions et en n'utilisant pas un « monstre » pour une application simpliste.

Je ne parle pas du nombre de leds ou de leur usage dans des proportions bibliques, mais de la logique les pilotant. Cette dernière précision est importante, car même si les microcontrôleurs actuels sont de plus en plus accessibles à toutes les bourses, utiliser, par exemple, un ATmega328 à 16Mhz disposant de 32Ko de flash et 1Ko de SRAM pour faire pulser trois leds constitue un gâchis inacceptable. La bonne philosophie consiste à trouver le juste équilibre entre les ressources à utiliser et l'objectif à atteindre. Gâcher 31Ko de flash est malvenu, mais le temps passé à trouver le moyen de grappiller quelques octets de code, voire à récrire finalement le programme en assembleur, représente également une ressource importante. Encore une fois, bien sûr, tout dépend de l'objectif. Si vous souhaitez améliorer vos connaissances en optimisation de code, rien de mieux que de relever le défi visant à faire tenir votre cahier des charges dans un microcontrôleur modeste arbitrairement choisi, mais préparez-vous à sauter quelques repas et perdre quelques heures de sommeil.

1 Le module

Le sujet qui nous intéresse ici est la mise en œuvre d'un afficheur 8 chiffres de DFRobot libellé « SPI LED Module » (réf. DFR0090). Celui-ci se présente sous la forme d'un module compact avec, en face avant, les huit afficheurs leds 7 segments et, à l'arrière, les 74hc595 (des registres à décalage). L'utilisation de ces derniers permet, à l'aide de seulement trois signaux, de piloter les 8 afficheurs, soit 64 leds (7 segments plus le point pour chaque afficheur). La connectique présente permet, de plus, de chaîner les modules. Leur nombre est ainsi une simple question de vitesse de propagation des signaux (temps d'affichage) et de courant qu'il est possible de fournir au module).

Le module dispose, en entrée, des signaux DATA, LATCH et CLOCK. Le premier permet de présenter les données à envoyer en série aux afficheurs (via les 74hc595) chaînés. Le second contrôle l'asservissement du module et donc le début et la fin de la transmission des données. Enfin, CLOCK, permet, comme un métronome, de cadencer l'envoi des données séries. Ainsi, avec l'alimentation du module en +5V et la masse, nous avons 5 broches utilisées sur le connecteur qui en compte 6 (ce dernier est relié à Vcc, la tension d'alimentation). Mais le module présente également un connecteur 6 broches en sortie, qui reprend exactement les mêmes signaux. Ceci permet de lier plusieurs modules entre eux. Ainsi connectés, deux afficheurs/modules, se comporteront comme un seul et même afficheur de 16 chiffres. Ceci est rendu possible par les caractéristiques même du 74hc595. En effet, au sein du module lui-même, ce sont 8 de ces composants qui sont reliés entre eux. Les données séries envoyées au premier 74hc595 vont littéralement déborder sur le second si nous envoyons plus de 8 bits correspondant aux 7 segments de l'afficheur led et le point.

Dans cet article, nous utiliserons un Atmel AVR ATtiny2313 placé sur une platine dédiée. L'AVR en question est relativement modeste avec ses 2Ko de

L'afficheur mis en œuvre repose sur l'utilisation de huit afficheurs leds 7 segments pilotés par des composants logiques chaînés, des 74hc595.

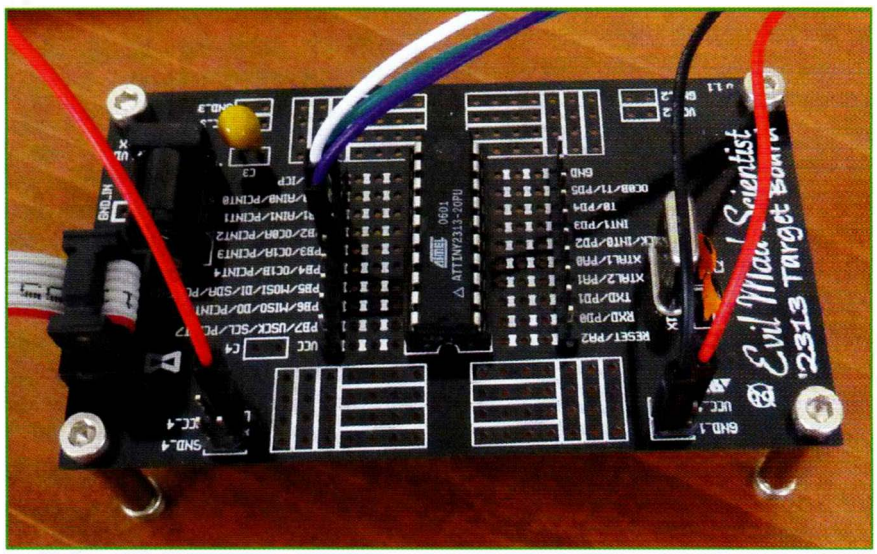
flash, 128 de SRAM et 128 d'EEPROM. C'est, pour peu que l'on fasse attention, largement suffisant pour transformer ce module en montage communiquant pouvant être piloté, par exemple, en mode série par un PC. Bien entendu, il ne s'agit là que d'un exemple, car l'ATtiny2313 dispose d'un grand nombre de périphériques internes dont un comparateur analogique. Notez qu'une version sensiblement plus « costaude » existe sous la forme de l'ATtiny4313, doublant simplement la quantité de flash, de SRAM et d'EEPROM. Nous avons ici délibérément choisi un petit microcontrôleur afin de mettre en évidence quelques points particuliers concernant les contraintes de programmation et, accessoirement, démontrer que les AVR des Arduino sont souvent démesurés par rapport à l'usage qui en est généralement fait.

A propos du circuit, nous utilisons ici une « ATtiny2313 Target Board » d'Evil Mad Scientist. Ce hacker propose en effet sur son site une platine pour ATtiny2313/4313 pour quelques 2 euros pièce ($3). La platine en question permet de facilement mettre en œuvre un tel AVR en fournissant des connectiques pour toutes les broches du microcontrôleur, un emplacement pour un quartz et ces condensateurs, un connecteur d'alimentation (avec condensateur), plusieurs broches Vcc/Gnd, un connecteur ISP et, enfin, quelques emplacements de prototypage pour ajouter des composants « maison ». Le tout, sous la forme d'un circuit 1/32 (0,8 mm) double face en fibre époxy sérigraphié, étamé, verni et avec des trous de fixation. Le circuit en lui-même (schéma et PCB) est également disponible au téléchargement sous licence CC-BY-SA 3.0, GPL et GFDL. De quoi largement faire plus propre qu'une platine Labdec (*breadboard* ou platine à essais).

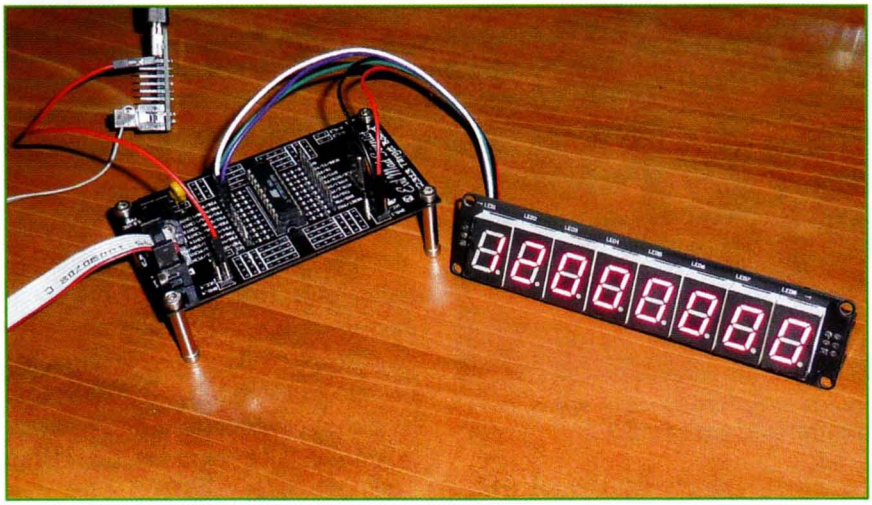
2 Le code

Vous l'avez compris, il ne s'agit pas ici de créer un « sketch » Arduino, mais un véritable programme en C bien normal. La différence n'est pas énorme puisque le soi-disant « langage Arduino » n'est rien d'autre qu'un ensemble de macros C utilisées dans un IDE simple (dont l'éditeur est des plus pénibles). Nous utiliserons donc ici, un éditeur de texte/code (Vim), AVR-GCC, AVR-libC, Avrdude (pour programmer l'AVR), Make et surtout, notre cerveau.

Comme nous faisons abstraction de la démesure d'Arduino pour un tel petit montage, nous ne pouvons pas utiliser une simple connexion USB pour programmer notre microcontrôleur. Il vous faudra user d'un programmeur pour charger le code via ISP (programmation In Situ AVR). Pour notre part, nous utilisons un programmeur Pololu disponible pour 15 euros et livré avec câble USB et connecteur ISP. Bien entendu, n'importe quel programmeur AVR ISP fera l'affaire pour peu qu'il soit reconnu par Avrdude (ce qui est généralement le cas). Ceci est également valable pour les montages utilisant un Arduino en guise de programmeur. Un article sur la programmation ISP avec Arduino dans le précédent numéro présentait une méthode pour programmer l'AVR d'un module Duemilanove sans autres accessoires que quatre petits câbles tous simples. Vous pouvez adapter ces explications facilement pour programmer n'importe quel AVR et non uniquement l'ATmega328 de l'Arduino.



La platine de prototypage pour ATtiny2313/4313 d'Evil Mad Scientist, bien plus pratique qu'une platine à essais et de très bonne facture.



Vue d'ensemble de la mise en œuvre du module d'affichage avec, en arrière-plan, le programmeur Pololu fournissant les signaux ISP, mais également un second port série TTL très pratique.

2.1 Principes en œuvre et stratégie de développement

Le module d'affichage dispose de trois signaux que nous connecterons à des sorties standards de l'AVR. Nous avons LATCH, correspondant au STCP du 74hc595, CLOCK pour SHCP et DATA pour DS. Sur le circuit du module le /OE (Output Enable) est relié à la masse et / MR (Master Reset) au Vcc (+5V). Q7s (la retenue) du premier 47hc595 est relié au DS du second et ainsi de suite jusqu'à arriver à la broche DATA du connecteur de sortie permettant de chaîner les modules afficheurs. On regrettera que l'ensemble des lignes /OE ne soient pas reliées à la sixième broche, inutilisée, du connecteur.

Le module n'accepte pas directement les valeurs numériques correspondant aux chiffres à afficher. Il ne met pas en œuvre de décodeurs BCD vers 7 segments comme le 4511B. Certains trouveraient cela dommage, car simplifiant grandement le code, mais d'un autre côté, cela permet une plus grande souplesse. Nous ne sommes pas ainsi tenus d'afficher uniquement des chiffres. Toute une palette de symboles ou d'effets graphiques sont utilisables avec des 74hc595. Le contre-coup consiste en la nécessité d'utiliser une table de symboles confectionnée par nos soins. Voici celle que nous utiliserons pour notre code :

```
0 : 00111111 : 0x3f
1 : 00000110 : 0x06
2 : 01011011 : 0x5b
3 : 01001111 : 0x4f
4 : 01100110 : 0x66
5 : 01101101 : 0x6d
6 : 01111101 : 0x7d
7 : 00000111 : 0x07
8 : 01111111 : 0x7f
9 : 01101111 : 0x6f

     g
   =========          76543210
   I       I          Dabcdefg
 d I       I e
   I   a   I
   =========
   I       I
 c I       I f
   I   b   I
   ========= @
             D
```

Nous avons ici les 7 segments d'un afficheur ainsi que le point décimal et la codification des symboles correspondant aux chiffres de 0 à 9. L'envoi de chaque symbole se fera en commençant par le bit de poids le plus faible. Il nous suffit, pour utiliser cet encodage, de stocker les 10 valeurs dans un tableau :

```
uint8_t chiffres[]={0x3f,0x06,0x5b,0x4f,0x66
,0x6d,0x7d,0x07,0x7f,0x6f};
```

La position dans le tableau donne directement le symbole du chiffre associé.

2.2 Envoi des données

La procédure pour envoyer des données est simple, même si la datasheet du 74hc595 est un peu déroutante. Il nous suffit de nous placer au niveau bas STCP, de présenter nos données sur DS et « clocker » à chaque bit sur SHCP. Pour envoyer huit bits, nous utiliserons une référence à notre tableau avec **`chiffres[digits[j]] & (1 << (7-i))`**. **`j`** donne la position du bit à envoyer et **`digits[]`** est un tableau contenant des chiffres à envoyer. Vous l'aurez compris, nous avons donc une double boucle :

```
volatile int8_t i,j;

SEGP &= ~(1 << STCP);  // STCP low
for(j=0;j<8;j++) {
  for (i=0;i<8; i++)  {
    if(chiffres[digits[j]] & (1 << (7-i)) ) {
      SEGP &= ~(1 << DS); // DS low
    } else {
      SEGP |= (1 << DS); // DS high
    }
    PINB = (1 << SHCP);  // SHCP high
    PINB = (1 << SHCP);  // SHCP low
  }
}
SEGP |= (1 << STCP);   // STCP high
```

`SEGP`, **`STCP`**, **`DS`** et **`SHCP`** sont définis par ailleurs en :

```
#define SEGP  PORTB
#define SEGPD DDRB
#define SHCP  PB1
#define DS    PB2
#define STCP  PB3
<file>
```

Nous n'oublierons pas, en début de la fonction **`main()`**, de placer ces E/S en sortie avec :

```
<file>
SEGPD  = (1 << SHCP) | (1 << DS) | (1 << STCP);
```

Cette base, se présentant sous la forme d'une double boucle **`for`**, parcourt un tableau de valeurs **`digits[]`** et envoie toutes les données bit par bit. Notez l'utilisation du registre **`PINB`** normalement destiné à lire les données sur un port configuré en entrée. Il s'agit là d'utiliser une fonctionnalité propre aux AVR et rarement utilisée. En effet, lorsqu'une ligne d'un port est configurée en sortie, le fait d'écrire un 1 à cet emplacement dans le registre d'entrée associé a pour

effet de basculer l'état en sortie. Nous n'avons ainsi pas besoin d'utiliser deux fois deux affectations pour « pulser » notre ligne d'horloge.

2.3 Factorisation et conversion

Nous placerons cette boucle dans une fonction permettant l'envoi d'une valeur numérique. C'est un choix totalement arbitraire et c'est pour cette raison que nous en faisons une section particulière de cet article. L'objectif est ici de disposer d'une fonction **`void outval(uint32_t val)`** prenant en argument une valeur entre 0 et 9 999 9999 et l'envoyant à l'unique module que nous mettons en œuvre. Un **`uint32_t`** (un **`unsigned long`** en réalité) est suffisant, permettant de stocker une valeur jusqu'à 4 294 967 296. C'est important, car si l'on décide de travailler avec des valeurs sur 64 bits pour une raison particulière, on touche alors à des fonctions qui sont présentes dans la **`libm.a`** de l'AVR-libC, la bibliothèque mathématique, et notre binaire s'en trouvera grossi au point de ne plus tenir dans les 2Ko de mémoire flash d'un ATtiny2313. Pour rappel, avec deux afficheurs, nous avons 16 chiffres et donc une valeur pouvant atteindre 9 999 999 999 999 999. Ceci ne tient pas dans un **`uint32_t`**. Il faudra utiliser une autre technique et éviter comme la peste, par exemple, de diviser un **`uint64_t`**.

En 32 bits en revanche, trois solutions s'offrent à nous. Les deux premières sont équivalentes, car l'une prend la forme d'une suite de divisions et l'autre d'une boucle. Voici tout d'abord la version « moche » :

```
volatile uint8_t digits[8];

digits[7] = val/10000000;
val -= digits[7]*10000000;

digits[6] = val/1000000;
val -= digits[6]*1000000;

digits[5] = val/100000;
val -= digits[5]*100000;

digits[4] = val/10000;
val -= digits[4]*10000;

digits[3] = val/1000;
val -= digits[3]*1000;

digits[2] = val/100;
val -= digits[2]*100;

digits[1] = val/10;
val -= digits[1]*10;

digits[0] = val;
```

La suite d'opérations est relativement explicite et c'est celle qui occupera, sans surprise, le plus de place en flash. Une version « boucle » pourrait être la suivante :

```
volatile int8_t i;
volatile int32_t j;
volatile uint8_t digits[8];

j=10000000;
for(i=7; i>=0; i--) {
  digits[i] = val/j;
  val -= digits[i]*j;
  j = j/10;
}*/
```

Nous partons, là aussi, des dizaines de millions et nous descendons dans les puissances de 10 jusqu'à arriver dans les unités. Notez que **`j`** est un **`int32_t`** alors que **`i`** se satisfera d'un **`int8_t`**, la SRAM est toute aussi rare que la flash sinon plus importante encore. Nous gagnons ici quelques précieuses dizaines d'octets d'espace pour le code.

Enfin, la dernière solution qui nous fera gagner une poignée d'octets, mais qui pourra être plus souple dans l'utilisation du code si nous comptons faire des choses originales en termes d'affichage, consiste à utiliser une conversion. Plutôt que d'extraire le nombre d'unités, de dizaines, de centaines, etc., nous allons transformer notre valeur numérique en chaîne de caractères avec **`ultoa()`** (*unsigned long to ASCII*). Nous n'aurons plus, ensuite, qu'à prendre individuellement chaque caractère de cette chaîne pour lui soustraire 48. Ceci est l'offset dans la table de caractères ASCII entre la valeur numérique d'un chiffre et son équivalent en termes de symbole. Le **`0`** est le caractère ASCII 48, le **`1`** est le **`49`** et ainsi de suite. Précisons ici que nous comptons obtenir un affichage sur huit positions préfixé de **`0`**, sous la forme **`00000123`**, par exemple.

Voici comment nous procéderons :

```
volatile int8_t i=0;
volatile int32_t j=0;
volatile uint8_t digits[8]={0,0,0,0,0,0,0,0};
volatile unsigned char buffer[9];

ultoa(val,buffer,10);

while(buffer[i]!='\0') i++;
i--;

while(i>=0) {
  digits[j]=buffer[i]-48;
  j++;
  i--;
}
```

Nous effectuons, dans un premier temps, la conversion en chaîne avec **`ultoa()`**. Notez que la taille de **`buffer[]`** est fixée à 9 caractères. Nous avons là nos 8 chiffres et le marqueur de fin de chaîne (**`\0`**). Attention, vous risquez le débordement de tampon si **`val`** atteint la centaine de millions ! En principe, nous devrions ajouter quelque chose comme **`if (val>99999999) val=0;`** en début de fonction pour nous protéger si la vérification n'est pas faite par ailleurs.

Nous explorons ensuite la chaîne obtenue à la recherche du marqueur de fin de chaîne et nous complétons enfin le tableau **`digits[]`** initialisé avec des **`0`**. Notez dans cette dernière boucle, un autre risque de dépassement de tampon. Si, par le plus grand des malheurs, vous utilisez un type numérique non signé pour **`i`**, c'est un cas d'école qui se présente : la boucle est infinie et des valeurs aléatoires sont inscrites un peu partout. Dans le cas d'un code dans l'espace utilisateur, le noyau vous rappelle immédiatement à l'ordre avec une erreur de segmentation (segfault). Dans le cas d'un microcontrôleur, c'est le plantage assuré et le résultat peut être catastrophique, jusqu'à rendre le microcontrôleur inutilisable et presque impossible à reprogrammer ou à effacer sans un matériel particulier (kit de développement STK500 par exemple, ou AVR Dragon). Les buffer overflows coûtent cher en développement sur microcontrôleur et sont difficiles à détecter. Il faut être prudent.

Conclusion allongée

En complétant notre **main()** d'une petite boucle de comptage pour test et quelques vérifications de paramètres, notre programme complet en binaire pèsera moins de 1Ko. Il est, bien entendu, possible de réduire sensiblement ce volume, mais n'oublions pas qu'il nous reste encore tout une autre moitié de la flash pour la logique du montage. De quoi largement implémenter la communication série via quelques fonctions et une routine d'interruption (voir Open Silicium 2) :

```
volatile unsigned char r_index;
unsigned char r_buffer[MAXSTR+1];
volatile unsigned char r_ready=0;

static void putchr(char c) {
  loop_until_bit_is_set(UCSRA, UDRE);
  UDR = c;
}

static void printstr_p(const char *s) {
  char c;

  for (c = pgm_read_byte(s); c; ++s, c = pgm_read_byte(s))  {
    if (c == '\n')  putchr('\r');
    putchr(c);
  }
}

ISR(USART_RXC_vect) {
  char c;
  c = UDR;
  if (bit_is_clear(UCSRA, FE)) {
    if (c != '\r') {
      putchr(c);
      if (r_index < MAXSTR) {
        r_buffer[r_index++]=c;
      }
    } else {
      putchr('\r');
      putchr('\n');
      r_buffer[r_index]=0x00;
      r_ready = 1;
      UCSRB &= ~_BV(RXCIE);
    }
  }
}

void gets_int(void) {
  r_ready=0;            // no waiting stuff
  r_index=0;            // flush index
  UCSRB |= _BV(RXCIE); // active RX int
}

void InitUART (unsigned char baudrate) {
  UBRRL = baudrate;
  UCSRB = (1 << RXEN) | (1 << TXEN);
  UCSRC = (1 << UCSZ1) | (1 << UCSZ0);
  UCSRB |= _BV(RXCIE);
}
```

Et dans **main()**, quelque chose comme :

```
volatile uint32_t i,k;
volatile uint8_t j;

InitUART (12); //4800@1Mhz (/8 fuse)
sei();
printstr_p(PSTR("boot...\n"));

[...]

while (1) {
  if(r_ready) {
    switch (r_buffer[0]) {
      case 'S' :
        printstr_p(PSTR("S ok\n"));
        if((r_index > 1) && (r_index < 10)) {
          k=1;
          i=0;
          for(j=r_index-1; j > 0; j--) {
            i += (r_buffer[j]-48)*k;
            k = k*10;
          }
          outval(i);
        } else {
          printstr_p(PSTR("Wrong arg !\n"));
        }
        break;
      case 'G' :
        printstr_p(PSTR("G ok\n"));
        outval(12345678);
        break;
      case 'Z' :
        printstr_p(PSTR("Z ok\n"));
        outval(0);
        break;
    }
    gets_int();
  }
}
```

Bien sûr, ce n'est là qu'un exemple rapide. Il n'est pas très intelligent de concevoir ce type de solution de cette manière puisque nous utilisons une chaîne de caractères obtenue via la liaison série, en **uint32_t** avant de la passer à **outval()**, qui procédera à l'opération inverse. L'interface de gestion série pourrait, par exemple, implémenter des ordres de mise en route de compteur, de remise à zéro, de relevé de mesure sur le comparateur ADC, ou de lecture des ports configurés en entrée. En conservant l'ATtiny2313 et ses 2Ko de Flash, il est possible de réaliser un afficheur série disposant d'un large panel d'effets, un peu comme celui en façade de certaines box Internet. Une autre option possible est de pousser plus loin que le simple prototypage par utilisation de modules. Intégrer un AVR dans le circuit et, avec quelques composants logiques comme le 74hc595, remplacer les afficheurs 7 segments numériques par leurs équivalents alphanumériques (14 ou 16 segments) peut constituer un exercice intéressant. Enfin, une autre option serait de conserver le module d'affichage mais de remplacer les afficheurs par des ULN2803 ou des transistors permettant alors de piloter des segments constitués de plusieurs leds (ou autres) et donc créer un afficheur 8 chiffres géant ! Et ce ne sont là que quelques idées... ■

DÉTECTION DE LA RÉSOLUTION DES MODULES LCD ALPHANUMÉRIQUES

par Yann Guidon

Grâce à une ou quelques petites résistances de rien du tout mais bien choisies, on peut aider un contrôleur embarqué à reconnaître le nombre de caractères affichés par un module LCD à base de HD44780. Dans cet article, nous examinons l'aspect électrique et électronique de cette méthode.

1 Introduction

Le contrôleur alphanumérique Hitachi HD44780 sert depuis des lustres à afficher des informations sur toutes sortes d'appareils nécessitant plus que quelques chiffres mais pas de sortie graphique. Les pompes à essence, les instruments de musique électroniques, les photocopieuses, les centrales d'alarme, les terminaux de carte bancaire dans les magasins et un nombre incalculable de montages Ohm made emploient ce circuit ou un des dérivés qui sont apparus depuis les années 80.

La prolifération de ces modules a baissé les prix et augmenté la diversité des modèles. Le HD44780 est relativement flexible et prévu pour travailler avec des résolutions de 1×16 et 2×8 caractères, l'ajout d'un circuit d'extension (ou plus) permet encore plus de configurations !

Après des années de récupération, d'enchères et de bricolages, j'ai constitué une collection de modules qui deviennent difficiles à gérer. Pas dans le sens difficiles à étiqueter ou classer, mais à programmer et utiliser. Quand on prototype beaucoup, il est bon d'avoir un vaste choix de modèles pour trouver celui qui convient le mieux, à condition de limiter la réécriture ou l'adaptation du logiciel. Ces modules ont une interface électrique et logicielle identique, mais des résolutions parfois incompatibles. Je dispose de modèles à 1×8, 2×8, 1×16, 1×20, 2×16, 2×20, 4×20 caractères et j'ai entendu parler d'autres résolutions : 1×40, 2×24, 2×32, 2×40, 4×16 et 4×40 !

Toutes ces versions se comportent de la même manière du point de vue du logiciel de contrôle, mais les effets des commandes changent avec les modules, qui peuvent être configurés différemment selon le constructeur, l'année de fabrication ou la méthode de matriçage choisie. La séquence d'instructions à envoyer pour initialiser le module,

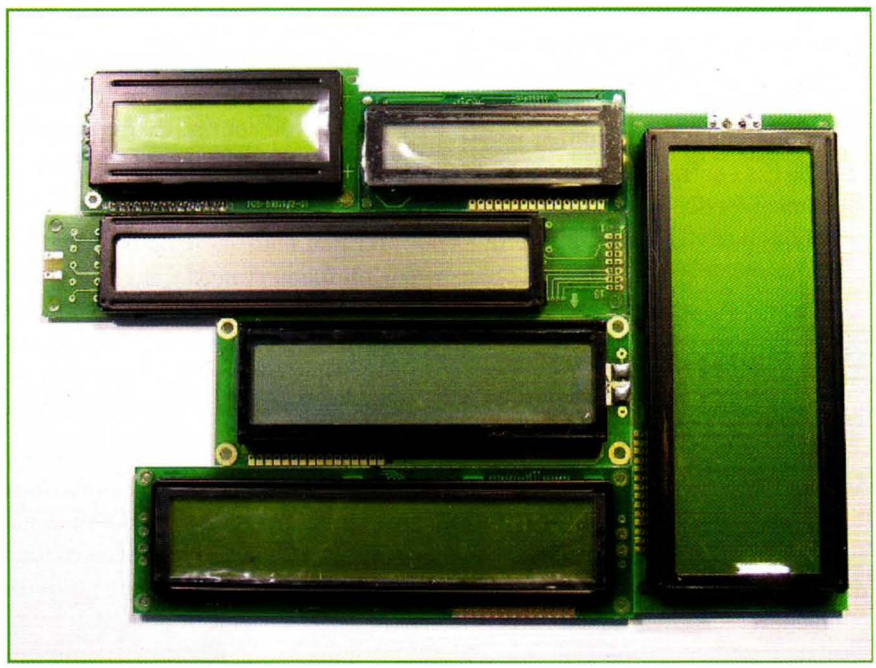


Figure 1 : Quelques afficheurs LCD alphanumériques de ma collection personnelle...

puis lui faire afficher des caractères aux bons endroits, demande parfois de la chance et de la persévérance. La documentation d'Hitachi est assez cryptique et les assembleurs font ce qu'ils veulent sur les circuits imprimés ou avec la matrice de cristal liquide. Des modules à l'aspect identique peuvent se révéler subtilement incompatibles... Une mauvaise configuration peut même changer la tension du bias et rendre l'affichage illisible...

Ces modules n'ont pas été prévus à l'origine pour être interchangeables et le standard de fait n'inclut aucune méthode pour reconnaître la configuration. Habituellement, un intégrateur achète quelques échantillons d'un module, conçoit un système autour, puis commande un conteneur entier pour la fabrication en masse, avec une référence identique ou suffisamment compatible. Mais les besoins évoluent, les microcontrôleurs sont partout et on aimerait bien avoir une petite bibliothèque de code portable qui fonctionne avec tous les types de modules. Pour cela, il faut trouver le moyen de détecter la configuration des caractères...

Pour terminer cette introduction, je tiens à préciser qu'il ne s'agit pas d'un tutorial sur le HD44780 et sa programmation. On trouve énormément de ressources à ce sujet sur Internet et une certaine familiarité est nécessaire pour bien comprendre la suite de cet article. Dans le cas contraire, cela devrait suffisamment éveiller votre curiosité pour que vous alliez voir les documentations par vous-même ;-)

2 Contexte et contraintes d'utilisation

Cet article a pour vocation d'inspirer un éventuel standard, ou au moins d'inciter d'autres amateurs et ingénieurs à adopter et améliorer le système. Pour favoriser cela, nous devons définir clairement le domaine d'application.

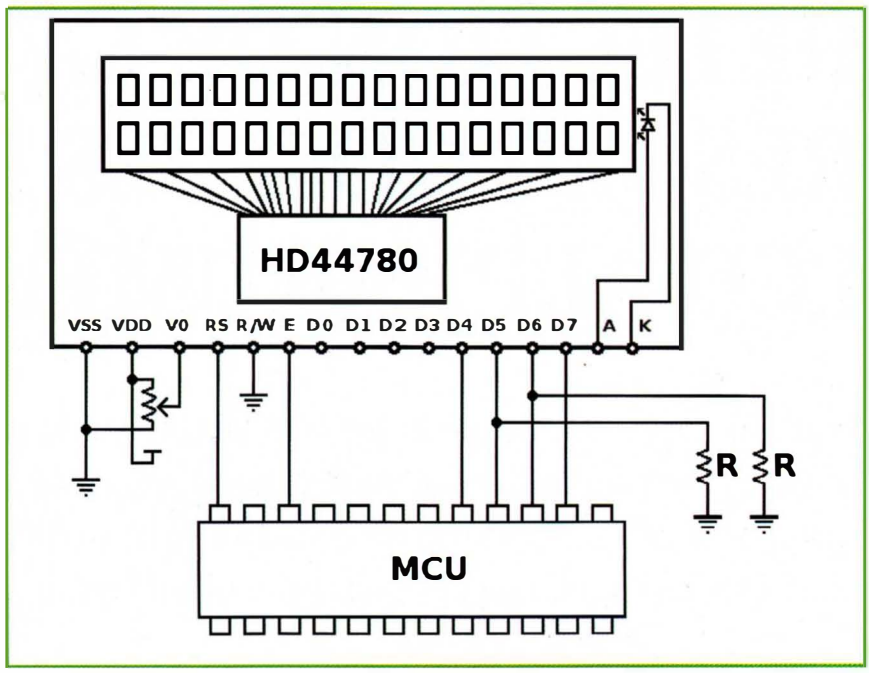


Figure 2 : Idée de départ, toute simple... Le microcontrôleur accède à l'afficheur en mode nibble et en écriture seule, mais peut aussi lire la configuration sur le bus de données (D4-D7).

D'abord, le système qui pilote l'afficheur est typiquement un microcontrôleur, un FPGA, ou tout circuit électronique capable de lire et d'écrire sur chacune de ses broches indépendamment. Nous devons utiliser des broches à 3 états (1, 0 et flottant) ce qui disqualifie d'office l'interface parallèle de l'imprimante (à cause des résistances de tirage au +5V) mais votre PIC, AVR, MSP ou MCU préféré devrait faire l'affaire. Pas besoin d'une bête de course, la fréquence de fonctionnement dépasse rarement le mégahertz.

Ensuite, nous allons nous concentrer sur un mode d'interfaçage sur 4 bits, ou mode nibble, qui économise 4 broches (D0 à D3) et utilise moins de ressources matérielles au prix d'un peu plus de code (pour découper les octets en deux moitiés).

Un grand nombre d'applications utilisent même le HD44780 en mode écriture seule pour économiser le câblage de la broche de validation de lecture. Au final, notre système de détection devra fonctionner avec les broches utilisées de la manière suivante :

- La broche R/W du module est fixée au Vss (0V) ce qui rend impossible la lecture des registres internes ou du flag Ready (qui indique lorsqu'une nouvelle opération peut être lancée). Le programme de contrôle doit générer les temporisations nécessaires.
- Les broches D4 à D7 sont connectées à notre électronique de pilotage (processeur hôte, microcontrôleur...) sur des broches bidirectionnelles, bien que la lecture des registres internes soit impossible. Mais il faut bien que les informations de configuration entrent par quelque part si on veut pouvoir les lire, non ?
- Les broches D0 à D3 du module sont laissées « en l'air », inutilisées. Elles sont maintenues à l'état haut par des résistances de rappel internes de la puce du HD44780. Elles peuvent tirer 125µA (typiquement), mais comme elles ne sont pas connectées, la consommation est nulle.
- Les broches R/S (sélection du registre) et E (impulsion de donnée valide) sont contrôlées par l'hôte en sortie uniquement.

Enfin, notre technique doit être simple, économique, ne doit pas interférer avec des systèmes existants ou rendre le module incompatible avec les spécifications du constructeur. L'hôte comme le HD44780 peuvent être alimentés en 3,3V ou en 5V, ou même avec des alimentations différentes et le système ne dépend pas du rétro-éclairage.

3 L'idée

Le décor est planté : nous tentons de lire une donnée câblée sur un bus de 4 bits en écriture seule. Le module ne va pas écrire sur le bus et il n'y aura pas de conflit électrique (Figure 2).

Durant le prototypage d'un circuit, on peut imaginer que la configuration de l'afficheur est indiquée par un ou des cavaliers sur le circuit imprimé, ou bien stocké en mémoire Flash. Dans le premier cas, cela fait de la soudure et de la surface supplémentaire et dans les deux cas, il existe le risque d'une incohérence entre la configuration de l'hôte et celle effective du module d'affichage. Il est bien connu que l'élément humain est le point faible des systèmes automatiques... La configuration doit donc être contenue dans le module.

En lisant les documentations des différentes révisions du circuit HD44780, on voit que les broches D4 à D7 ont des pull-ups de 125μA tout comme les broches D0 à D3. Lorsque le processeur hôte laisse flotter ses broches, il va lire l'état « 1 » partout. L'idée développée dans cet article consiste à mettre des résistances de pull-down sur certaines des 4 broches de données pour encoder une combinaison parmi 16.

- Matériellement, cela revient à souder une, deux ou trois résistances sur les broches adéquates.
- Logiciellement, il suffit de mettre le bus de données de 4 bits à l'état flottant et de lire le code, puis consulter une table (en logiciel) qui va donner les paramètres de l'afficheur : résolution, adresse des caractères, etc.

En pratique, c'est un peu plus subtil mais rien d'affolant, juste de l'affinage des paramètres. Il y a en tout cas de nombreux avantages à cette approche :

- Une résistance ne coûte presque rien (c'est de la poussière dans le jargon de l'industrie électronique), c'est même le niveau 0 de la technologie électronique, car il n'y a aucun composant actif à programmer.
- Les circuits imprimés existants peuvent être patchés facilement à la main, les résistances au format 1206, 0805 et 0603 sont plus petites que les broches standards au pas de 2,54mm. On peut même utiliser une résistance traversante de 1/4W ou 1/8W dont les longues pattes économisent la préparation d'un segment de fil...
- Pour la production en masse, la modification manuelle n'est pas envisageable, mais le circuit imprimé du module peut être facilement modifié en amont, en ajoutant des empreintes pour quelques résistances. Elles seront soudées (ou pas) durant les dernières étapes de la fabrication, selon les besoins.
- La méthode est compatible avec les protocoles et interfaces connus, avec des tensions mixtes, en mode 4 ou 8 bits, en écriture seule ou avec la lecture.

4 Optimisation et affinage des paramètres

Il faut reconnaître que malgré la complexité logicielle nécessaire, le HD44780 a un avantage important : il ne consomme presque rien (un tiers de milliampère environ). Il serait dommage de gâcher cette qualité en ajoutant des résistances inconsidérément...

Nous allons essayer d'optimiser la consommation électrique et nous avons trois approches complémentaires pour réduire l'intensité des courants de fuite que notre méthode implique.

4.1 Optimisation du nombre de résistances

Le courant de fuite dépend du nombre de résistances connectées entre les broches de données et les rails d'alimentation. Nous pouvons le réduire en allouant des codes avec peu de résistances aux cas les plus répandus. L'allocation intelligente nécessite de bien connaître les modules et leur utilisation...

- Le code sans résistance 1111 peut indiquer que le module n'a pas été transformé. On l'alloue donc à la configuration par défaut, 1×8 caractères. Qui peut le plus, peut le moins...
- Il existe quatre codes avec une résistance, que l'on alloue aux modules très répandus. Attention à l'ordre des broches :
  - 1110 (résistance sur D4) 1×16
  - 1101 (résistance sur D5) 2×16
  - 1011 (résistance sur D6) 2×20
  - 0111 (résistance sur D7) 4×20 (j'ai beaucoup de ces modules...)
- Les codes à deux résistances sont utilisées par des modèles moins courants, mais quand même possibles :
  - 0011 (résistances sur D6 et D7) 1×20
  - 0101 (résistances sur D5 et D7) 1×40

- 1001 (résistances sur D5 et D6) 2×8
- 0110 (résistances sur D4 et D7) 2×24
- 1010 (résistances sur D4 et D6) 2×32
- 1100 (résistances sur D4 et D5) 2×40

- Les quatre codes à trois résistances consomment le plus de courant et sont réservés aux cas improbables, aux modèles rares et aux extensions :
  - 0001 (résistances sur D5, D6 et D7) 4×16
  - 0010 (résistances sur D4, D6 et D7) 4×40 (Ce modèle n'est pas directement compatible, car il faut une deuxième broche E pour gérer les deux moitiés de l'afficheur, mais il est inclus par souci d'exhaustivité).
  - 0100 (résistances sur D4, D5 et D7) Réservé
  - 1000 (résistances sur D4, D5 et D6) Réservé (pour une configuration particulière et incompatible d'un afficheur plus courant)

- Quatre résistances consomment encore plus de courant et deux cas se présentent :
  - Soit le module n'est pas branché et les broches flottantes sont tirées vers le 0V.
  - Soit on considère qu'il y a un module et on convient de lire le code sur les autres broches (D0-D3).

Toutes ces combinaisons sont réunies dans le tableau suivant :

4.2 Optimisation de la valeur des résistances

On ne peut pas prendre une résistance au hasard. Notre méthode modifie les caractéristiques électriques (tensions et courants) sur un ou des fils du bus de données, qui a déjà deux sources de courant (le processeur hôte et le pull-up du contrôleur HD44780). Avec une résistance trop faible, la consommation du système augmentera inutilement (le courant venant de l'hôte sera absorbé vers la masse). Si elle est trop forte, le pull-up interne du HD44780 empêchera l'hôte de lire l'information.

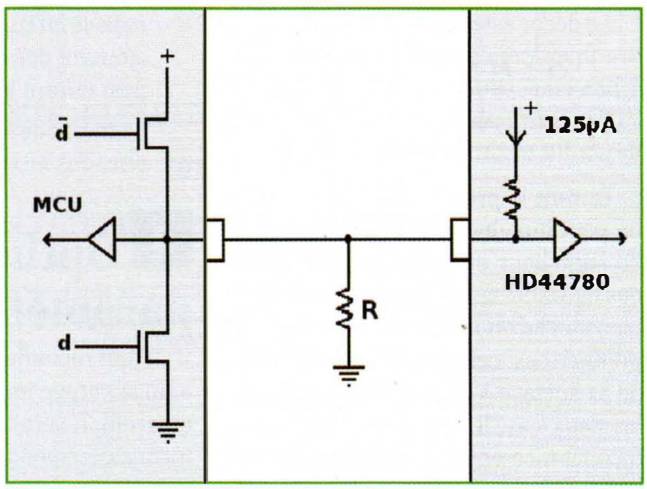


Figure 3 : La résistance sur le fil ajoute un troisième élément qui influence le courant et la tension.

Code lu sur D7-D4		Résistance sur				Résolution et commentaires
Déc.	Binaire	D7	D6	D5	D4	
0	0000	*	*	*	*	Réservé / Extension / Erreur
1	0001	*	*	*		4×16
2	0010	*	*		*	4×40 (ajouter la broche E2)
3	0011	*	*			1×20
4	0100	*		*	*	Réservé
5	0101	*		*		1×40
6	0110	*			*	2×24
7	0111	*				4×20
8	1000		*	*	*	Réservé, configuration alternative
9	1001		*	*		2×8
10	1010		*		*	2×32
11	1011		*			2×20
12	1100			*	*	2×40
13	1101			*		2×16
14	1110				*	1×16
15	1111					1×8 / défaut

- Le processeur hôte force facilement la tension sur le fil lorsqu'il écrit des données à envoyer sur le bus. Une broche normale peut fournir de 5 à 25mA, ce qui est suffisant pour avoir une tension proche de l'alimentation, donc une valeur claire.
- La tension sur le fil doit éviter de rester proche de la moitié de la tension d'alimentation. Dans ce cas, les transistors des broches d'entrées entreraient en conflit interne (ils seraient actifs simultanément) et se mettraient à consommer plus de courant.
- Les plages de tensions valides (représentant clairement un « 1 » ou un « 0 » logique) varient avec la tension d'alimentation. Elles sont plus étroites avec une tension plus basse.

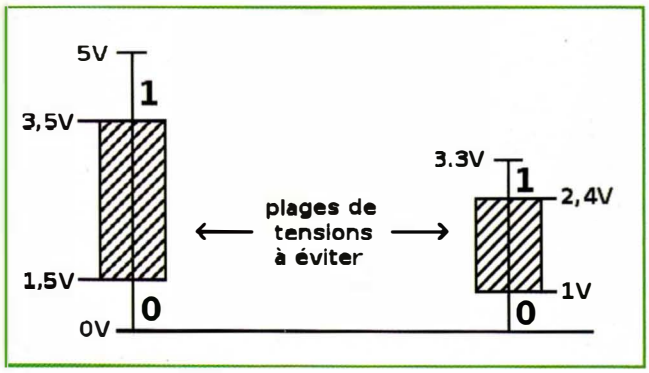


Figure 4 : La valeur de la résistance doit être choisie pour que les tensions évitent certaines zones qui augmentent la consommation interne du composant.

Pour que notre résistance fonctionne avec un système à alimentation mixte (par exemple, l'hôte sous 3,3V et le HD44780 sous 5V), on doit s'assurer que la résistance abaisse la tension du fil à moins de Vdd×0,3=3,3V×0,3=1V. Mais pas trop non plus, sinon le contrôleur devra fournir plus de courant pour écrire un « 1 » sur le fil.

Si on considère que le HD44780 est alimenté sous 5V, alors le datasheet indique que son pull-up tire 125µA (valeur typique donnée). La résistance de configuration devra être un peu plus faible que R=U/I=1V/0.000125A=8KΩ.

Avec cette première estimation de résistance, nous pouvons calculer le courant que les circuits devront fournir pour amener le fil à une tension de 5V par exemple : I=U/R=5V/8000Ω=625µA. C'est cinq fois le courant de rappel du HD44780. donc le processeur hôte devra fournir 0,5mA de son côté.

Il faut aussi savoir que le courant de pull-up augmente avec la tension d'alimentation. De plus, la valeur de 125µA est dite typique, c'est une valeur moyenne attendue mais pas garantie. Le datasheet indique qu'il peut atteindre 250µA et une résistance de 4,7KΩ serait plus sûre, augmentant encore ainsi le courant consommé.

Au final, une résistance consomme plus que l'afficheur en fonctionnement. Si quatre résistances de configuration sont utilisées, elles consommeront plus de 2mA ensemble ! Il faut encore réduire le courant moyen...

4.3 Optimisation de la période d'utilisation

La troisième technique d'économie d'énergie consiste à minimiser le temps pendant lequel les résistances pompent le courant. Cela implique que nos pull-downs sont commandés par un signal supplémentaire, mais il est hors de question d'en créer un uniquement pour cela.

En regardant les signaux disponibles, on voit qu'on n'a guère le choix. Les broches de données transmettent les données, la broche E est active à l'état haut (donc passe tout son temps à l'état bas, l'inverse de ce qu'on cherche) et il ne reste que la broche RS (sélection de registre) qui est active à l'état bas.

C'est donc sur la broche RS que nous allons connecter les résistances, au lieu du 0V. Leur consommation va donc dépendre de l'activité de l'afficheur, elle chutera considérablement durant les périodes où on n'envoie pas de commandes d'affichage.

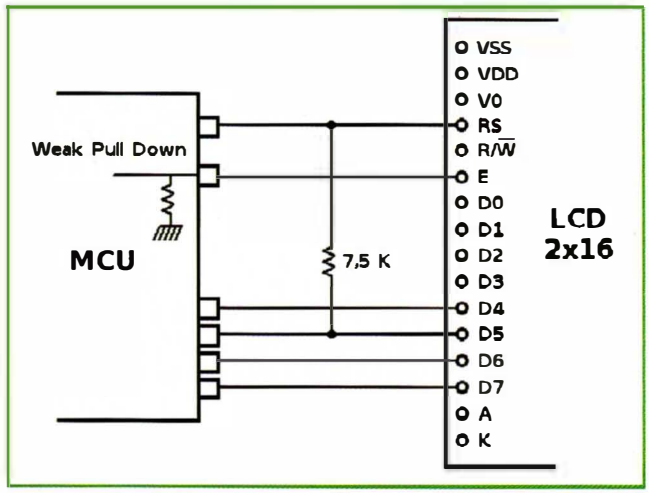


Figure 5 : Idée améliorée, quasiment toute aussi simple... Il ne faut pas oublier de fixer la tension de la broche E avec l'hôte. Elle est active à l'état haut et n'a pas de pull-up interne du côté du HD44780.

On peut même aller plus loin : laisser flotter les broches de l'afficheur dès qu'on a terminé l'envoi d'une commande. Cela réduit ainsi la consommation dans les pull-ups du bus de données et les broches de contrôle. La seule exception est la broche E qui n'a pas de résistance de rappel et ne doit pas flotter. Soit on code le logiciel pour maintenir constamment cette broche à l'état bas, soit on utilise une résistance de rappel au niveau de l'interface hôte (en interne si la broche est configurable, ou bien avec 100KΩ ou 1MΩ avec une résistance classique).

5 Mise en pratique

Afin de vérifier les calculs, j'ai testé le montage sur deux modules typiques de ma collection.

5.1 Module 2×16

Pour commencer, j'ai choisi un module acheté dans un magasin d'électronique il y a plusieurs années. C'est un modèle tout ce qu'il y a de plus standard, un clone économique mais de bonne facture. J'ai soudé une résistance 1/4W de 7,5KΩ en utilisant les pattes comme des fils. Le seul risque est qu'elles entrent en contact avec les broches du module, mais elles sont suffisamment rigides pour éviter les courts-circuits durant une manipulation.

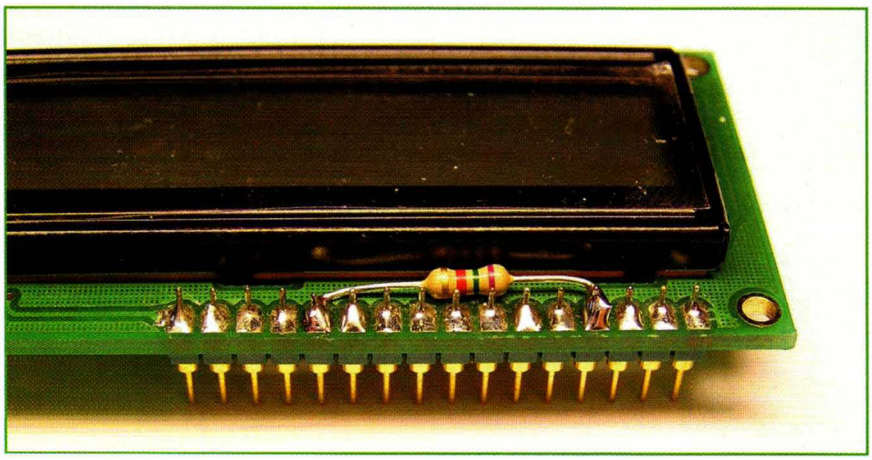


Figure 6 : Câblage d'une résistance traversante sur un module 2×16 caractères

Cet afficheur a une résolution de 2 lignes de 16 caractères et donc son code est « 1101 », on met juste une résistance entre le signal 4 (E) et le signal 12 (D5).

5.2 Module 1×16

Le deuxième module est un vieux circuit utilisant un HD44780A vintage en boîtier QFP. C'est assez remarquable, car la puce du contrôleur de la plupart des modules actuels est engluée sous une goutte de résine époxy et n'a pas de marquage évident.

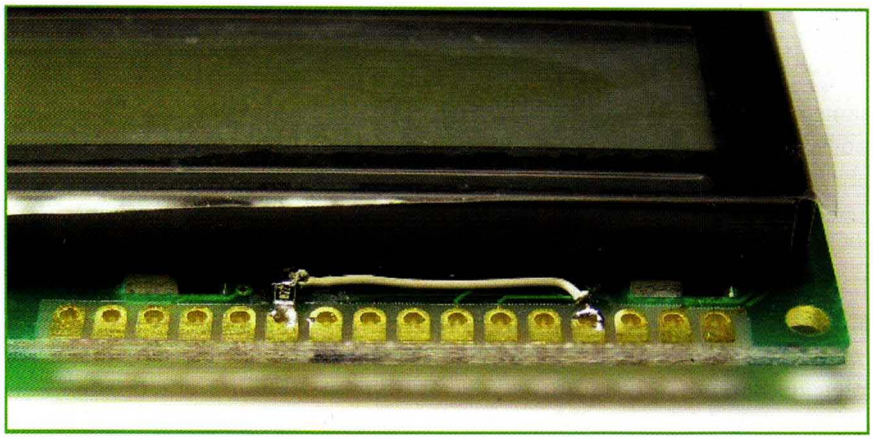


Figure 7 : Une résistance à montage en surface et un peu de fil font aussi l'affaire sur ce module 1×16 caractères

Comme la résolution est 16×1 caractères, le code est « 1110 », donc une seule résistance relie D4 (la broche 11) et RS (broche 4). J'avais la valeur de 8200Ω sous la main, j'aurais dû essayer avec 4,7kΩ.

Le module est neuf et son connecteur n'avait pas été soudé. En soudant une résistance de 8200Ω au format 0805, le plus important est de ne pas boucher le trou métallisé, ce qui empêcherait la soudure d'une barrette sécable.

5.3 Mesures

Ce module 16×1 étant équipé d'un contrôleur HD44780A, c'est-à-dire de première génération, il ne fonctionne que sous 5V et je n'ai pas testé sous 3,3V. Le contraste n'aurait pas été suffisant de toute façon.

D'abord, la consommation totale du module avec toutes les broches flottantes (donc tirées à 5V par les pull-ups, sauf E qui est connecté à Vss) est de 0,3mA, comme l'indique le datasheet.

Ensuite, avec RS et RW à 0V et une résistance de pull-down sur le bus de données, la consommation augmente à 0,7mA, c'est à dire 0,3mA plus le courant de 3 pull-ups (D4, RS et RW). Le courant qui circule dans la broche RS est 0,23mA, la somme des courants de pull-up de D4 et RS. La tension sur la broche D4 est alors de 0,91V (Figure 8, page ci-contre).

Lorsqu'on force D4 à 5V, on y mesure un courant de 0,6mA, alors que le courant de court-circuit dans le pull-up est de 0,11mA seulement. On n'est pas loin des 0,125mA du datasheet et les calculs donnent des valeurs très proches. Une résistance de 4,7KΩ serait plus sûre pour travailler avec des systèmes mixtes 5V/3,3V.

6 Pour aller plus loin...

Le système proposé plus haut permet 16 combinaisons, dont environ 10 utilisables. Si on a besoin de plus de

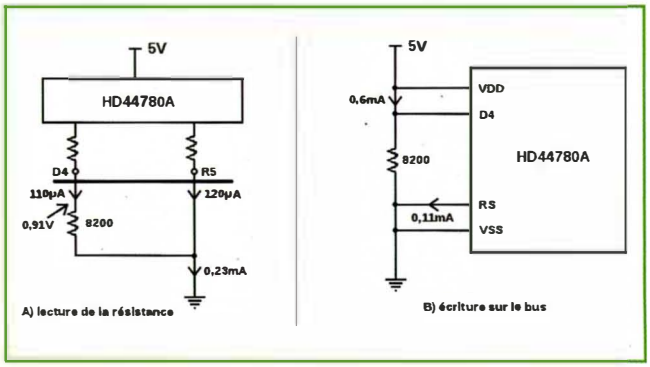


Figure 8 : Mesures à l'état haut et bas

combinaisons, on peut utiliser les autres broches de données (D0 à D3), mais on sort alors de notre domaine d'utilisation, car il faudrait câbler 4 autres fils.

Il existe une autre possibilité qui ne nécessite aucun câblage supplémentaire, mais plus de flexibilité sur le bus de données. En effet, nous avons considéré pour l'instant que le bus de données ne permettait pas de contrôler la direction individuelle des bits, nous plaçant dans la situation de la figure 9a. Pourtant, de nombreux microcontrôleurs peuvent définir une direction arbitraire sur la plupart de leurs broches.

Cela ouvre de nouvelles possibilités, comme le montre la figure 9b, où le rôle de la broche E est échangé tour à tour avec les 4 autres broches de données. Topologiquement, cela permet non plus 4 mais (5×4)/2=10 connexions réciproques et indépendantes, donc 2^10=1024 combinaisons !

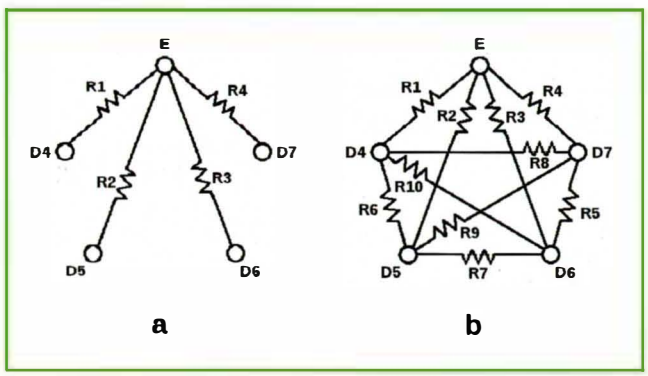


Figure 9 : Le principe de la lecture du code sur le bus de données (a) peut être étendu à plus de codes (b) si on contrôle la direction de chaque broche individuellement. Le pentagramme apparaît en effectuant quatre rotations du dessin original.

En plus, les 16 codes que nous avons déjà définis sont des sous-codes des 1024 possibles. C'est donc réellement une extension de notre système. Par exemple, ces codes peuvent indiquer si on doit configurer le contrôleur en mode 1 ligne ou 2 lignes, ou bien si les caractères ont 8 ou 10 pixels de haut (deux paramètres qui relèvent parfois plus de la devinette que de l'ingénierie).

En pratique, les résistances sont trop faibles pour que tous les codes fonctionnent. Par exemple, si on relie D4 et D5, ainsi que E et D4, avec des résistances, D4 et D5 apparaîtront à l'état bas si on met E à 0V. A moins d'effectuer la lecture en forçant alternativement D4 et D5 à l'état haut et bas, en plus de l'état flottant.

Une autre solution consiste à ajouter des diodes sur certains segments. Et ces diodes peuvent aussi servir à coder 4 combinaisons par lien au lieu de deux :

- Pas de résistance
- Résistance
- Résistance avec diode dans un sens
- Résistance avec diode dans l'autre sens

Chaque connexion entre deux broches peut ainsi coder deux bits d'information dans certains cas. Par contre, l'analyse de toutes les combinaisons valides devient un gros casse-tête. Tout cela rend le codage, l'algorithme de lecture et la réalisation matérielle trop complexes, nous en resterons pour l'instant au code simple à 16 combinaisons.

Conclusion

Une simple résistance (ou deux, ou éventuellement trois) permet à un module alphanumérique d'informer le circuit hôte sur sa configuration. On peut alors envisager des systèmes plus modulaires, interchangeables, extensibles. L'ajout de cette (ces) résistance(s) garde les modules compatibles avec les applications existantes qui ne détectent pas la résolution.

Les paramètres électriques ont été vérifiés sur un module, mais il faut toujours consulter les datasheets, mesurer les tensions et tester les systèmes dans leur ensemble, pour chaque nouveau type de module à utiliser. En cas de doute, la méthode de calcul de la résistance est assez simple : prendre la résistance équivalente du pull-up interne et la diviser par 10, ce qui permet d'obtenir environ un dixième de la tension d'alimentation à l'état bas.

Au passage, nous avons vu que le moyen le plus simple et le plus efficace de réduire la consommation de ce type d'afficheur est de laisser ses broches flotter.

On peut alors envisager qu'une seule bibliothèque de code suffise pour commander de nombreux types d'afficheurs. mais son écriture fera l'objet d'un autre article. ■

Référence

[1] HITACHI, « HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver) » (datasheet)

PROTOTYPAGE ÉCONOMIQUE SUR FPGA À TRÈS FAIBLE CONSOMMATION

par Yann Guidon

Alors que le code VHDL avance tranquillement et sûrement, il faut aussi songer à donner un support matériel au projet de la montre de Laura. Après plus d'un an de préparations, nous sommes excités de voir les premiers résultats concrets ! Nous allons commencer à réaliser un prototype en FPGA et explorer le monde de l'ultra-faible consommation en découvrant l'IGLOO d'Actel/Microsemi. Nous abordons dans ces pages les questions du choix, de la soudure, du brochage, du câblage, des alimentations et de l'initialisation.

1 Mon premier IGLOO

En raison de la complexité du projet, le premier prototype de montre est construit sur des plaques de prototypage. Nous ferons un circuit imprimé plus tard mais pour l'instant, nous avons besoin de vérifier que les connexions, tensions et courants sont tous corrects. Il est plus facile de développer le circuit avec des fils qu'on peut souder et dessouder à loisir, au lieu d'un circuit imprimé plus cher et complexe dont on n'est même pas sûr de la validité.

Dans cet article, nous nous penchons uniquement sur le module FPGA IGLOO dont la complexité est réduite au minimum. Les connecteurs, les interfaces et les alimentations sont reportées sur la carte d'accueil, sujet de l'article suivant.

Le matériel consistera donc en deux plaques économiques : le module IGLOO repose sur un adaptateur QFP100 universel (qui n'a pas été conçu pour notre FPGA) au pas de 0,5mm, alors que la carte d'accueil est juste un morceau de circuit prétroué à pastilles carrées. Les deux plaques seront reliées par des barrettes sécables mâles et femelles, chaque partie sera donc interchangeable et réutilisable pour d'autres projets !

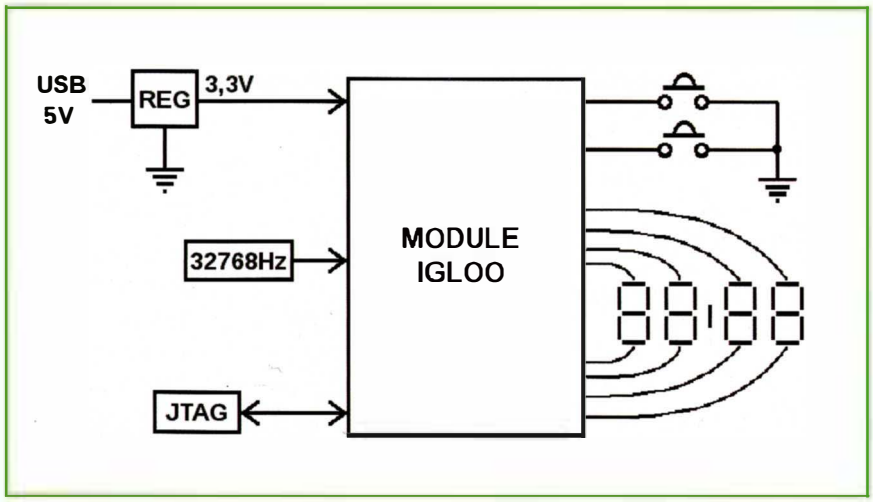


Figure 1 : Synoptique de la carte d'accueil qui utilisera le module conçu dans cet article.

Nous partons d'une conception existante que nous modifions pour l'adapter à nos contraintes et pour tenir compte des inconvénients des modules précédents. Ces modules sont déjà très simples, mais on va encore réduire les fonctionnalités. Par exemple, nous n'inclurons pas de connecteur JTAG ni d'oscillateur 25MHz et la PLL sera désactivée. La platine garde la génération du 1,5V mais le régulateur 3,3V, la programmation ainsi que l'horloge à quartz sont déportées sur la plaque à pastilles, avec les autres entrées-sorties, pour plus de flexibilité.

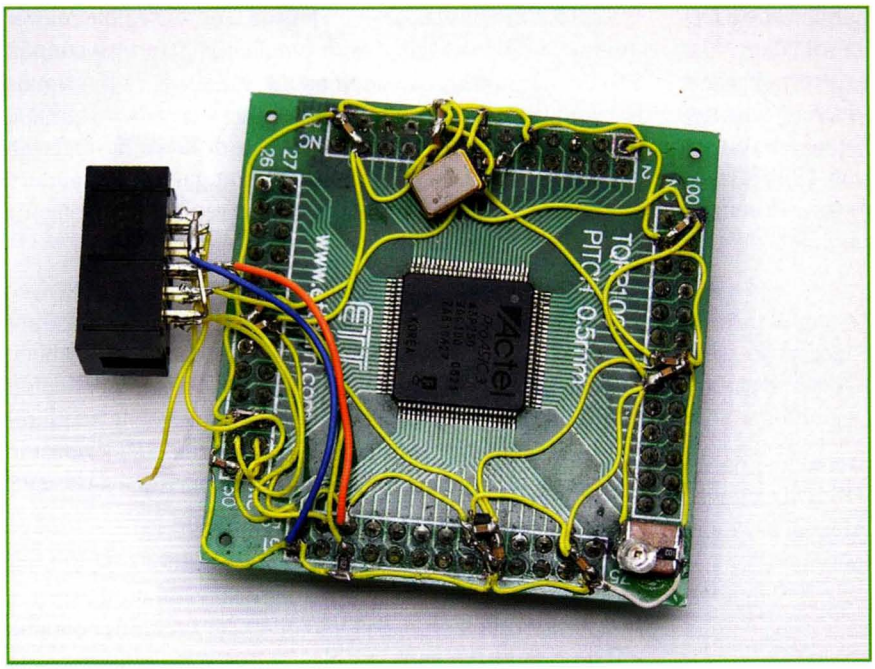


Figure 2 : Un ancien module de prototypage « maison » à base d'Actel ProAsic3. Il sert de base pour concevoir le module à base d'IGLOO.

2 L'IGLOO qu'il nous faut

La plus grosse différence avec les modules déjà conçus est le choix de la référence du FPGA. J'ai déjà l'habitude de la série ProASIC3 d'Actel, mais là, nous avons besoin de très faible consommation à basse fréquence. La série IGLOO est idéale pour cela et est dérivée des ProASIC3, donc facile à mettre en œuvre quand on connaît déjà les autres produits Actel(a).

Remarque

(a) La société Actel a été rachetée récemment par Microsemi. Nous utilisons le nom Actel par habitude.

Les IGLOO héritent de la plupart des caractéristiques des ProASIC3 (configuration mémorisée en Flash interne, très peu de composants externes nécessaires) et peuvent presque les remplacer, mais ils sont optimisés pour des fréquences, des tensions et courants encore plus faibles. A tel point qu'ils pourraient être alimentés par une petite pile bouton, par exemple, comme notre montre justement.

La structure des cellules logiques de l'IGLOO, de ses horloges et de ses mémoires est identique, mais les entrées/sorties ont plus d'options que les ProASIC3. Par exemple, les IGLOO disposent de « trigger de Schmitt » (entrées à hystérésis) très pratiques pour nos interfaces avec le monde analogique (les boutons par exemple).

La référence sélectionnée est **AGL125VQG100-V5**. Examinons cette séquence alphanumérique, qui décrit les caractéristiques de ce composant :

- **AGL** est le préfixe qui indique que le composant appartient à la famille IGLOO.
- **125** est une indication de la densité, ou capacité de la matrice FPGA. Il ne correspond à rien de physique ou d'objectif, mais il permet de comparer la taille par rapport à d'autres produits Actel (familles IGLOO, ProASIC3, Fusion, etc.). Contrairement à Altera (qui indique le nombre de cellules logiques) et comme avec Xilinx, c'est un nombre dont l'unité est arbitraire et commerciale : « system gates » ou « complexité équivalente en milliers de portes logiques ». Pour gonfler ce nombre, les constructeurs y incluent parfois la complexité des blocs de mémoire internes ou d'autres éventuels accélérateurs câblés directement dans la puce. Pour comprendre ce que cela implique, il faut bien lire la documentation d'Actel. On y voit que la densité « 125 » correspond à ces caractéristiques :
  - 3072 « tuiles » (portes logiques à 3 entrées ou mémoires 1 bit)
  - 16µW en mode « Flash*Freeze » (hibernation instantanée)
  - 8 blocs de SRAM double port de 512 octets
  - 128 octets de FlashROM
  - 1 PLL
  - 2 banques de broches d'entrées/sorties aux tensions indépendantes
- **VQG100** signifie que c'est un boîtier VQFP à 100 broches. VFQP à son tour veut dire *Very thin Flat Quad Package* ou boîtier carré très fin, de 14mm de côté. 71 broches sont disponibles pour les entrées/sorties et leur pas est de 0,5mm. Le **G** correspond à la variante Green, ou verte, garantie sans substances chimiques dangereuses (conformément à la directive RoHS).
- **-V5** est un suffixe qui spécifie que le composant fonctionne avec une tension interne de 1,5V. Sans ce suffixe, il fonctionne aussi à 1,2V, mais c'est plus cher et surtout, il faut tout de même alimenter en 1;5V lors de la programmation. De toute façon, notre montre fonctionne à des fréquences tellement faibles que cela ne fait pas de différence, je soupçonne que la matrice FPGA peut encore faire fonctionner la montre correctement sous 1V...

Cette référence de FPGA est un peu surdimensionnée pour une simple petite

montre, il y a largement plus de portes logiques et de mémoires que nécessaire et nous n'avons pas besoin d'une PLL ou d'autant de broches. Cependant, puisque cette montre est juste un premier projet et que les idées en attirent d'autres, il vaut mieux dépenser quelques euros de plus et voir un peu large. La référence choisie coûte environ 12EUR (sans taxes ni frais de port), donc ce n'est pas un investissement énorme. Les outils de programmation le sont, mais c'est un autre débat.

La taille de cet IGLOO est suffisante pour que plus tard, on ait un peu de place pour réaliser un tout petit microprocesseur ou microcontrôleur (au hasard, un YASEP ?), afin d'ajouter des fonctions intelligentes et des capteurs à la montre.

Enfin, le boîtier VQFP100 est le plus facile à souder parmi toutes les options de boîtier disponibles. Un IGLOO avec moins de broches existe seulement en BGA ou QFN, ce qui est insoudable sans équipements industriels...

3 Fabrication du module IGLOO

La première étape consiste à réunir les composants. Le FPGA est commandé chez un distributeur officiel (il faut se méfier des circuits parallèles comme les enchères pour de nombreuses raisons) et un circuit imprimé d'adaptation QFP100 a été trouvé pour quelques dollars sur un site web d'Asie du sud.

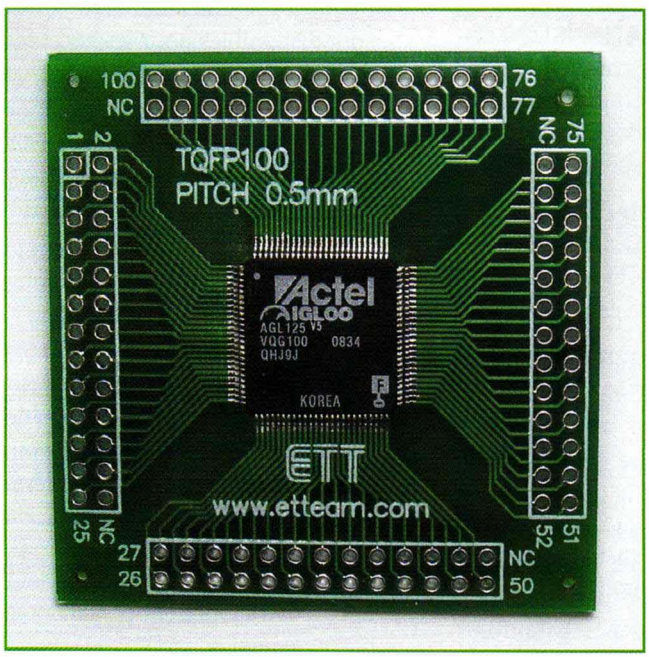


Figure 3 : La puce IGLOO en boîtier QFP100, soudée sur son support d'adaptation. Un peu plus lente que les ProAsic3, elle consomme aussi encore moins !

La soudure des broches au pas de 0,5mm semble relever du tour de force, mais c'est assez facile quand on connaît la méthode, qui utilise une des merveilles de la physique : la capillarité. La procédure de soudure n'est pas « patte à patte » mais tout d'un coup, avec le côté de la panne du fer. La tension de surface et le vernis du circuit gardent la soudure aux endroits désirés, tant qu'on arrive à réunir trois éléments ou facteurs importants :

- Placement : il faut aligner précisément les broches avec les pistes, qui font environ 230 microns de large. Tant que l'alignement n'est pas parfait, il ne faut pas continuer. On peut s'aider de ruban adhésif (en polyamide/Kapton pour supporter les hautes températures de la soudure) pour maintenir le circuit en place sur la plaque. C'est la partie la plus délicate, car l'alignement risque de bouger lors de la soudure...
- Flux : on mouille ensuite toutes les pattes avec une seringue ou une pipette de liquide de flux de soudure. Cela permettra à l'étain de « mouiller » et faire des contacts propres et des surfaces bien nettes et brillantes.
- Dosage de la soudure : le dernier secret c'est que la plaque est déjà étamée abondamment. Pas besoin de fil à souder ! Il y a juste ce qu'il faut pour avoir de jolis ménisques aux pattes (si on a tout fait correctement). Si un apport de soudure est nécessaire, il est vraiment minime et il faut s'armer de flux supplémentaire et de tresse à désouder pour éliminer tout ce qui dépasse...

On peut aussi aborder la question de la température et de l'alliage de brasure, deux sujets étroitement liés et qui ont une grosse importance sur la qualité de la soudure. Un fer thermorégulé permet de réaliser le meilleur travail : certains fers permettent de régler la puissance, mais cela ne contrôle pas directement la température, à garder entre 300 et 350° C. Les alliages sans plomb sont plus difficiles à utiliser et fondent à plus haute température, je préfère encore utiliser un alliage 60/40 (60% d'étain, 40% de plomb) qui donne facilement de bons résultats.

Lorsque le circuit est enfin positionné (ce qui est difficile, car les pattes sont en équilibre sur les bosses d'étain des pistes), on soude une broche dans un coin, puis une autre à l'opposé. On peut alors se passer de l'adhésif et ajuster plus finement le placement en chauffant la soudure dans un coin et en poussant délicatement le circuit pour aligner toutes les pistes.

Une fois que la position est parfaitement ajustée, on peut souder d'un coup toutes les broches d'un côté. Lentement, avec le côté de la panne du fer à souder, on chauffe les pattes et l'étain. Ce dernier, grâce au flux de soudure, va connecter les pattes et les pistes ensemble. Si le circuit est penché, l'éventuel excédent de soudure peut descendre et être enlevé facilement dans un coin. La capillarité ne laissera que la quantité nécessaire de soudure sur les pistes.

Une fois la soudure finie, on enlève le flux avec un liquide ou un spray spécifique, puis on examine les pattes avec un compte-fils, une loupe binoculaire ou un microscope de poche. Un grossissement de 20 permet de détecter la plupart des défauts.

3.1 Connecteurs

Nous allons bientôt souder des fils sur l'adaptateur afin de connecter les alimentations. Avant cela, et pour rendre cette opération plus pratique, nous soudons des connecteurs femelles au pas de 2,54mm. Ils permettront la fixation sur la plaque prétrouée au moyen de barrettes sécables mâles et rendent le module FPGA facilement réutilisable pour des projets futurs.

Une fois les connecteurs femelles soudés, n'oublions pas de créer un détrompeur. La carte carrée serait endommagée si elle était insérée du mauvais côté, alors on bouche un des connecteurs femelles en le fondant avec le fer à souder. La carte de l'adaptateur a quatre broches marquées « NC », « non connectées », et celle près de la broche n° 100 devient le détrompeur.

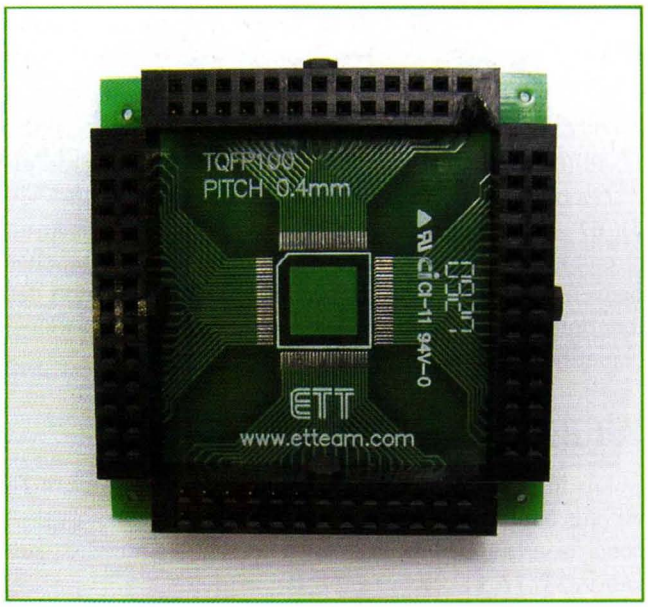


Figure 4 : L'adaptateur QFP100, côté verso, a maintenant des connecteurs femelles. Un détrompeur est aménagé au fer à souder en fondant le plastique près de la broche 100.

3.2 Câblage du module

Maintenant, les choses deviennent plus sérieuses et il faut passer beaucoup de temps à lire et relire le datasheet du FPGA [1] pour comprendre la fonction et l'utilisation de chaque broche. C'est là que la connaissance préalable des ProASIC3 est précieuse puisqu'on peut partir d'un module existant (figure 2) et chercher les différences.

La table d'assignation des broches est fournie sur la page 3-37 du PDF « IGLOO Low Power Flash FPGAs Datasheet ». Nous fournissons ici, en plus du nom, des commentaires sur la fonction et l'utilisation de chaque broche. Nous les organisons en quatre quadrants, un par côté du boîtier de la puce, pour faciliter le repérage.

Quadrant Ouest :

Pin	Nom	Tension	Fonction
1	GND	0V	
2	GAA2/IO67RSB1		E/S
3	IO68RSB1		E/S
4	GAB2/IO69RSB1		E/S
5	IO132RSB1		E/S
6	GAC2/IO131RSB1		E/S
7	IO130RSB1		E/S
8	IO129RSB1		E/S
9	GND	0V	
10	GFB1/IO124RSB1		E/S
11	GFB0/IO123RSB1		E/S
12	VCOMPLF	0V	
13	GFA0/IO122RSB1		E/S
14	VCCPLF	alimentation PLL (inutilisée: 0V sur carte fille)	
15	GFA1/IO121RSB1		E/S
16	GFA2/IO120RSB1		E/S
17	VCC	1,5V	
18	VCCIB1	3,3V	
19	GEC0/IO111RSB1		E/S
20	GEB1/IO110RSB1		E/S
21	GEB0/IO109RSB1		E/S
22	GEA1/IO108RSB1		E/S
23	GEA0/IO107RSB1		E/S
24	VMV1	3,3V	
25	GNDQ	0V	

Quadrant Sud :

Pin	Nom	Tension	Fonction
26	GEA2/IO106RSB1		E/S
27	FF/GEB2/IO105RSB1		E/S, Flash*Freeze
28	GEC2/IO104RSB1		E/S
29	IO102RSB1		E/S
30	IO100RSB1		E/S
31	IO99RSB1		E/S
32	IO97RSB1		E/S
33	IO96RSB1		E/S
34	IO95RSB1		E/S
35	IO94RSB1		E/S
36	IO93RSB1		E/S
37	VCC	1,5V	
38	GND	0V	
39	VCCIB1	3.3V	
40	IO87RSB1		E/S
41	IO84RSB1		E/S
42	IO81RSB1		E/S
43	IO75RSB1		E/S
44	GDC2/IO72RSB1		E/S
45	GDB2/IO71RSB1		E/S
46	GDA2/IO70RSB1	réservé pour compatibilité avec A3P250/AGL250	
47	TCK	1K->GND	JTAG (pull-down)
48	TDI		JTAG
49	TMS		JTAG
50	VMV1	3.3V	

```
Quadrant Est :
Pin   Nom              Tension     Fonction
 51     GND            0V
 52     VPUMP          3.3V        JTAG  (découpler avec 10nF et
330nF)
 53     NC                         Ne pas connecter
 54     TDO                        JTAG
 55     TRST           1K->GND     JTAG (pull-down)
 56     VJTAG          3.3V        JTAG
 57  GDA1/IO65RSB0                 E/S
 58  GDC0/IO62RSB0                 E/S
 59  GDC1/IO61RSB0                 E/S
 60  GCC2/IO59RSB0                 E/S
 61  GCB2/IO58RSB0                 E/S
 62  GCA0/IO56RSB0                 E/S
 63  GCA1/IO55RSB0                 E/S
 64  GCC0/IO52RSB0                 E/S
 65  GCC1/IO51RSB0                 E/S
 66     VCCIB0         3.3V
 67     GND            0V
 68     VCC            1,5V
 69      IO47RSB0                  E/S
 70  GBC2/IO45RSB0                 E/S
 71  GBB2/IO43RSB0                 E/S
 72      IO42RSB0                  E/S
 73  GBA2/IO41RSB0                 E/S
 74     VMV0           3.3V
 75     GNDQ           0V

Quadrant Nord :

Pin   Nom              Tension     Fonction
 76  GBA1/IO40RSB0                 E/S
 77  GBA0/IO39RSB0                 E/S
 78  GBB1/IO38RSB0                 E/S
 79  GBB0/IO37RSB0                 E/S
 80  GBC1/IO36RSB0                 E/S
 81  GBC0/IO35RSB0                 E/S
 82      IO32RSB0                  E/S
 83      IO28RSB0                  E/S
 84      IO25RSB0                  E/S
 85      IO22RSB0                  E/S
 86      IO19RSB0                  E/S
 87     VCCIB0         3.3V
 88     GND            0V
 89     VCC            1,5V
 90      IO15RSB0                  E/S
 91      IO13RSB0                  E/S
 92      IO11RSB0                  E/S
 93      IO09RSB0                  E/S
 94      IO07RSB0                  E/S
 95  GAC1/IO05RSB0                 E/S
 96  GAC0/IO04RSB0                 E/S
 97  GAB1/IO03RSB0                 E/S
 98  GAB0/IO02RSB0                 E/S
 99  GAA1/IO01RSB0  réservé pour compatibilité avec A3P250/AGL250
100  GAA0/IO00RSB0  réservé pour compatibilité avec A3P250/AGL250
```

La bonne surprise, c'est qu'il n'y a quasiment aucune différence avec le brochage du A3P125VQ100, qui a déjà été mis en œuvre pour d'autres projets. L'utilisation de l'IGLOO est donc accélérée par la réutilisation des notes de conception des modules précédents (qui sont distillées ici).

La seule différence visible avec notre IGLOO c'est la broche 27 qui, en plus des fonctions usuelles (broche d'entrée-sortie ou entrée de signal d'horloge), peut aussi recevoir un signal externe (appelé « Flash*Freeze ») qui sert à figer l'état interne du FPGA et à le placer en mode de consommation ultra-faible. Nous ne l'utilisons pas, car cette broche isole le cœur du FPGA des entrées-sorties alors que notre montre doit fonctionner en permanence (en particulier, recevoir le signal d'horloge à 32768Hz).

Le brochage de l'AGL125VQG100 est non seulement compatible directement avec l'A3P125VQG100, mais aussi avec l'A3P250VQG100 et l'AGL250VQG100. Ces derniers ont deux fois plus de tuiles logiques et utilisent aussi les broches 46, 99 et 100 comme broches d'alimentation. Afin de préserver une compatibilité ascendante et descendante entre des modules utilisant ces références, ces trois broches sont réservées. Cela nous laisse encore 68 broches pour nos circuits.

3.3 Broches au 0V

La première chose à faire est de trouver quelles broches sont connectées à Vss, ou le 0V, ou « ground » ou « la masse ». Ici, elles sont marquées « GND » et « GNDQ », mais certaines broches doivent être aussi connectées à ce potentiel commun et il faut patiemment lire les documentations et éplucher les schémas des platines de référence pour le comprendre. Par exemple « VCOMPLF » est une des broches alimentant la PLL, connectée à GND.

Les broches connectées au 0V sont donc **1 9 12 25 38 51 67 75 88**. Les broches **46** et **99** le seraient aussi sur un circuit imprimé compatible avec l'AGL250. D'autres broches peuvent aussi être mises à la masse selon l'utilisation des fonctions qu'elles alimentent ; on délègue cette connexion à la platine d'accueil.

3.4 Broches à 1,5V

Ensuite, il faut alimenter le cœur de la puce. Comme on l'a vu plus haut, il faut 1,5V avec cette référence. Les broches concernées s'appellent VCC, elles portent les numéros suivants : **17 37 68 89**.

La broche VCCPLF (n° **14**) alimente la PLL en 1,5V, mais nous n'en avons pas besoin (et cela consomme du courant), donc elle ne sera pas câblée sur la platine. La platine d'accueil la connectera au 1,5V ou à GND en fonction du projet (Figure 5).

3.5 Broches à 3,3V

Les broches d'entrées-sorties ont leurs propres alimentations, réparties en deux banques indépendantes qui peuvent chacune travailler entre 1,2V et 3,3V. Une banque inutilisée doit connecter son alimentation à GND pour économiser du courant.

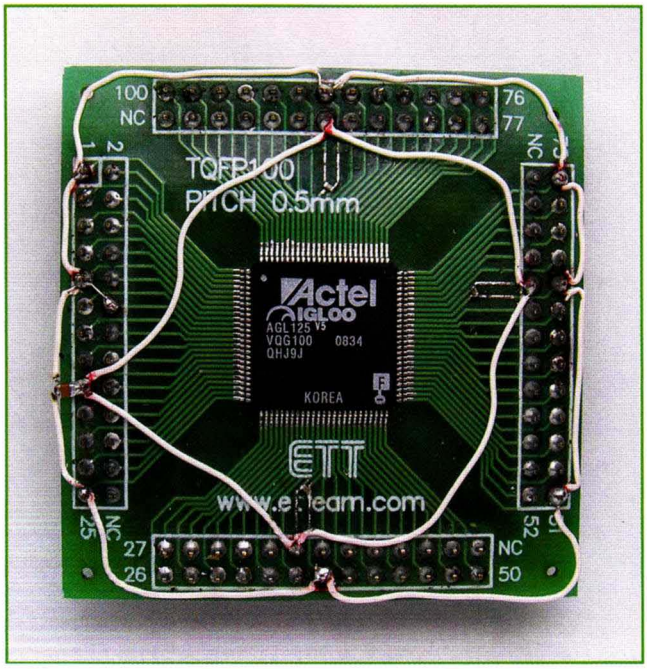


Figure 5 : Le 0V et le VCC sont connectés en anneaux concentriques et quatre condensateurs de découplage (format 0805 et 0402) sont soudés.

Sur cette platine, on sait qu'on doit alimenter au moins une banque en 3,3V. On choisit la banque 0 et on en profite pour y brancher aussi le régulateur 1,5V. Ainsi, la carte d'accueil n'a pas à gérer le VCC. De plus, si la banque 1 est inutilisée, elle peut être câblée à GND par la carte d'accueil.

Les broches d'alimentation des banques sont nommées VMVx et VCCIBx et portent les numéros **66 74 87** pour la banque **0**, et **18 24 39 50** pour la banque 1 (câblées séparément). La broche **100** est laissée inutilisée, par compatibilité avec les AGL/A3P250.

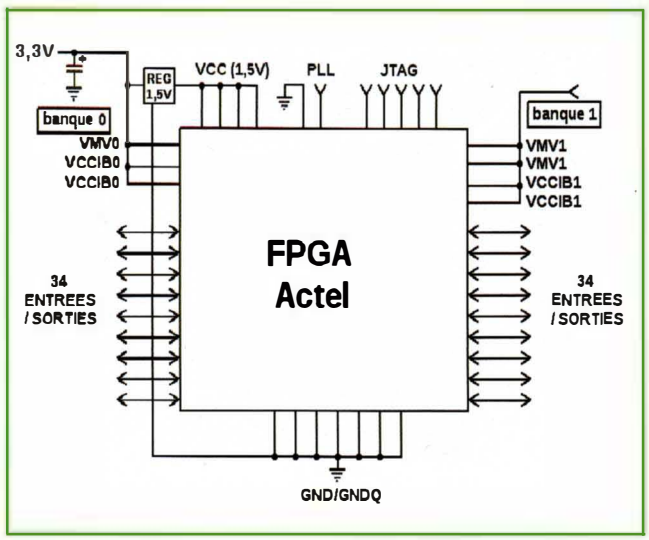


Figure 6 : Connexions essentielles du FPGA sur sa platine

3.6 Broches de l'interface JTAG

La programmation du FPGA (le flashage de la configuration interne) est réalisée au travers d'une interface standard JTAG. On retrouve les signaux usuels : TCK, RDI, TMS, TDO et TRST sur les broches **47 48 49 54 55**. Les broches **47** et **55** sont tirées vers le 0V par une résistance d'1KΩ. Toutes ces broches, ainsi que diverses alimentations, vont être connectées à une prise HE10 à 10 broches sur la plaque d'accueil, car les premiers modules utilisaient des fils volants qui n'étaient pas mécaniquement fiables. Par sécurité en revanche, les deux résistances sont implantées sur l'adaptateur QFP100 pour éviter un déclenchement intempestif du protocole JTAG (des broches non connectées peuvent capter des parasites et envoyer des signaux indésirés).

L'interface JTAG doit aussi être alimentée et Actel a inclut la broche de la pompe de charge interne (VPUMP) ainsi que l'alimentation du JTAG (Vjtag) dans le connecteur. Elles ne sont pas reliées directement au 3,3V, car elles consommeraient de l'énergie en dehors des opérations de programmation.

3.7 Découplage

Les broches d'alimentation doivent être « découplées » au moyen de condensateurs placés au plus près des pattes pour absorber les pics de courant qui se produisent durant le fonctionnement de tout circuit électronique.

La plaque d'adaptation QFP100 n'est pas prévue pour accueillir des condensateurs, mais nous allons ruser. En grattant le vernis au-dessus de certaines pistes, nous mettons à nu le cuivre aux endroits voulus et cela crée une surface soudable. On peut alors souder des composants ultraminiatures à montage en surface, des condensateurs céramiques au format 0603 (minuscule) ou 0402 (encore plus minuscule, pour les plus courageux).

Les valeurs sont typiquement de 10nF à 100nF et il ne faut pas hésiter à en mettre partout. Les broches de l'IGLOO sont conçues pour faciliter le découplage : les broches de 0V sont souvent au milieu des deux autres tensions d'alimentation et on peut regrouper deux condensateurs à plusieurs endroits.

Quelques condensateurs de plus forte valeur (au tantale par exemple) peuvent aussi compléter la plaque, par exemple plusieurs μF à chaque coin de la carte. Ils sont d'ailleurs recommandés pour stabiliser les régulateurs de tension.

Il faut aussi ajouter un découplage particulier à la broche VPUMP qui commute beaucoup de courant lors de la programmation. Cela sert à générer en interne la haute tension qui programme la mémoire Flash. Actel recommande un condensateur de 10nF en parallèle avec 330nF 16V. Pour faciliter les choses, la broche VPUMP (**52**) est juste à côté d'une broche GND (**51**).

3.8 Le régulateur 1,5V MIC5247-1.5

Le module est presque terminé, il ne reste plus qu'à ajouter le régulateur 1,5V pour VCC. On récupère le 3,3V sur le circuit d'alimentation de la banque 0.

Ce régulateur est le seul circuit actif (semi-conducteur) du module et là encore, il va falloir consulter les documentations. Le choix de la référence n'est pas critique, mais le soin apporté à la sélection sera répercuté sur la consommation du module.

Je dispose d'un modèle, le MIC5247-1.5 de Micrel. Des modèles équivalents ou similaires sont proposés par de nombreux autres fabricants. Celui-ci est spécifié pour fournir 150mA sous 1,5V (la tension de sortie est donnée pour être précise à 1%) tout en consommant 85µA.

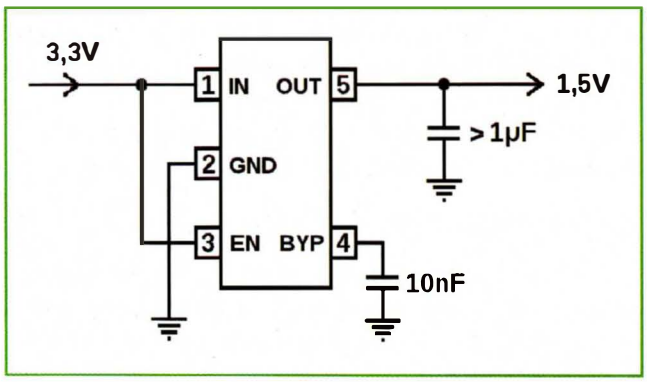


Figure 7 : Utilisation et brochage du régulateur de tension 1,5V

En raison de la très faible consommation du projet, il est peu probable que le cœur du FPGA (avec une très faible fréquence et activité) consomme plus de quelques milliampères, mais qui peut le plus peut le moins. Selon la documentation, le cœur nécessite environ 30µA lorsqu'il est inactif (selon la table 2-11, p2-9, du datasheet) et j'estime la consommation à moins de 100µA mais on devra mesurer pour être sûr. En tout cas, la consommation du régulateur sera plus élevée que celle du cœur du FPGA... La solution actuelle n'est donc pas optimale, mais on améliorera cette partie avant de concevoir le circuit imprimé. J'ai déjà des idées de pompes de charges en tête...

En plus des broches d'alimentation, le MIC5247 dispose de deux broches supplémentaires. L'une active ou désactive la sortie, on laisse cette broche à l'état haut (donc connectée à l'entrée 3,3V) pour que le régulateur soit constamment actif. L'autre broche sert à réduire le bruit intrinsèque de la tension du régulateur. On n'en a pas besoin et on peut laisser cette broche en l'air, mais 10nF ne font pas de mal (sauf aux yeux si la soudure est trop délicate, j'y ai mis un autre condensateur au format 0402).

Le boîtier du régulateur est au format SOT23-5, qui mesure moins de 3mm de côté. Ses 5 broches au pas de 0,95mm ne sont pas trop difficiles à souder individuellement. Le circuit

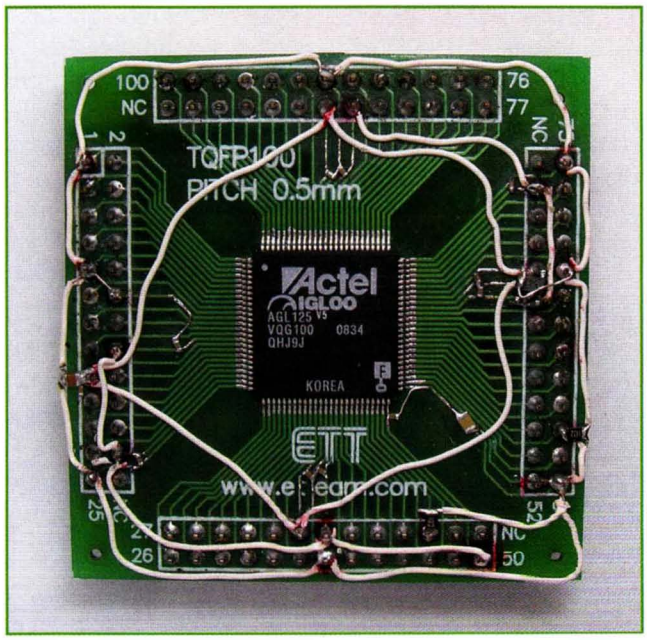


Figure 8 : La platine terminée, avec les deux circuits d'alimentation des banques d'entrées-sorties, les résistances de l'interface JTAG, les condensateurs de découplage et le régulateur 1,5V sur la droite.

imprimé ne dispose pas de pistes pour ce circuit et il faut encore ruser. Cette fois-ci, cela consiste à souder le circuit intégré en l'air en s'aidant des broches du connecteur femelle. Après quelques acrobaties et quelques broches délicatement tordues, le module est prêt !

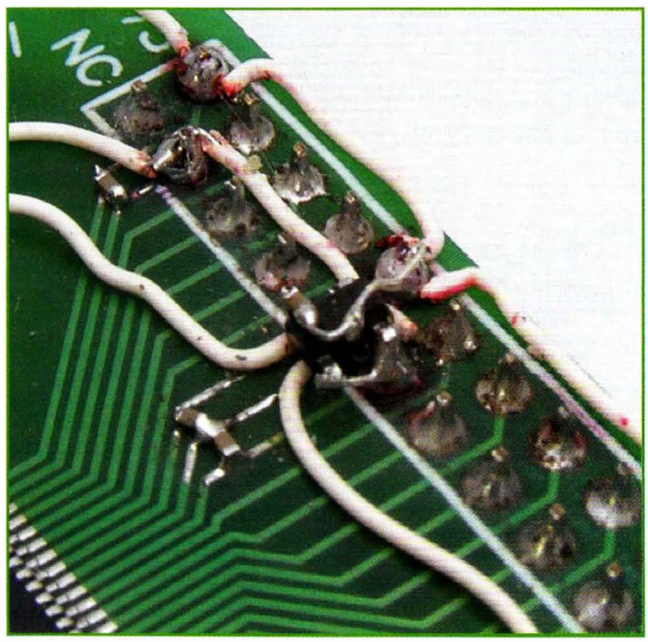


Figure 9 : Ce qui a l'air d'un blob de soudure est en fait le régulateur 1,5V en SOT23-5, retourné, soudé en l'air entre deux broches du connecteur femelle. C'est délicat mais ça marche (et c'est plus joli quand on a enlevé les résidus de flux de soudure).

4 Mise sous tension et remise à zéro

L'étape initiale de la conception et de la réalisation d'un circuit consiste à l'alimenter. Cela permet ensuite de le programmer et ainsi, en troisième lieu, de faire fonctionner ses entrées-sorties pour ajouter des périphériques au fur et à mesure. Avant d'en arriver là, on doit s'assurer que notre FPGA s'initialise correctement et cela demande de replonger dans la documentation...

4.1 Séquence d'allumage des alimentations

Comme la majorité des circuits intégrés modernes, l'IGLOO dispose de plusieurs rails d'alimentation, avec au moins deux tensions différentes : 1,5V pour le cœur et une tension pour chaque banque d'entrée-sortie. Une grande qualité de la famille IGLOO (et ses cousins ProASIC, Fusion, etc.) c'est que chaque rail peut être mis sous tension indépendamment, sans séquence particulière.

Certains autres circuits grilleraient si on ne respectait pas un certain ordre d'allumage (et d'extinction), mais l'IGLOO est conçu pour permettre d'alimenter chaque partie indépendamment. Il faut lire longtemps entre les lignes des manuels pour s'en rendre compte, car ce n'est pas explicitement expliqué, mais on y trouve ceci :

> IGLOO FPGA Fabric User's Guide, p343
>
> The I/O bank VMV pin must be tied to the VCCI pin within the same bank. Therefore, the supplies that need to be powered up/down during normal operation are VCC and VCCI. **These power supplies can be powered up/down in any sequence during normal operation** of IGLOO, IGLOO nano, IGLOO PLUS, ProASIC3L, ProASIC3, and ProASIC3 nano FPGAs. During power-up, I/Os in each bank will **remain tristated until the last supply (either VCCIBx or VCC) reaches its functional activation voltage**. Similarly, during power-down, I/Os of each bank are tristated once the first supply reaches its brownout deactivation voltage.

En langage clair, pour comprendre ce que tout ce charabia veut dire, on peut imaginer plusieurs cas :

- Si on alimente uniquement Vcc (au-delà de 0,85V environ), les entrées-sorties ne fonctionnent pas, mais le cœur (la matrice de portes logiques) est en état de marche et peut mémoriser des informations. Toutefois, les cellules logiques restent inactives, car lorsqu'une banque est désactivée (en raison de son Vccib inférieur à 0,8V), les broches d'entrée correspondantes fournissent la valeur 0 (c'est expliqué quelques pages plus loin). Sans aucune valeur d'entrée, le FPGA ne peut rien faire, mais il conserve les données en SRAM ou dans les registres. C'est utile soit pour le placer en mode d'hibernation, soit lors de la phase de programmation de la configuration (si on n'a pas envie d'alimenter toutes les broches).
- Si on alimente toutes les banques et le cœur avec les tensions nominales, tout se passe comme prévu. Mais on peut aussi n'alimenter que certaines banques d'entrées-sorties sans que cela influence les autres banques ou le cœur (il ne faut pas s'attendre à lire des données valides cependant). C'est très intéressant pour notre montre, car on pourrait désactiver la banque qui commande l'afficheur quand il n'est pas allumé.
- Si on alimente les banques d'E/S mais pas le cœur, aucune entrée-sortie n'est active. On lit, quelques pages plus loin, que les résistances de rappel ne sont pas actives non plus, ce qui force à en souder nous-mêmes si on s'attend à trouver des valeurs spécifiques sur les broches.

> IGLOO FPGA Fabric User's Guide, p350
> Internal Pull-Up and Pull-Down
>
> Low power flash device I/Os are equipped with internal weak pull-up/-down resistors that can be used by designers. If used, these internal pull-up/-down resistors will be activated during power-up, once both VCC and VCCI are above their functional activation level. Similarly, during power-down, these internal pull-up/-down resistors will turn off once the first supply voltage falls below its brownout deactivation level.

On peut retenir de tout cela que ce FPGA a été spécialement conçu pour réduire sa consommation par tous les moyens possibles. Il en résulte une grande flexibilité d'alimentation et cela ne va pas nous poser de souci pour notre montre. Au contraire, on pourrait même jouer avec différentes tensions sur différentes banques alimentées tour à tour pour grapiller encore quelques microampères !

4.2 Fais risette à la broche...

Contrairement à la majorité des FPGA, les IGLOO, comme toute leur famille de circuits programmables à cellules Flash, se retrouvent dans un état indéterminé (plus ou moins aléatoire) lorsqu'on branche leur alimentation. C'est en fait tout à fait normal si on examine la structure électronique de leurs portes logiques : elle réalisent les cellules de mémoire (les registres ou « bascule D » ou les « latches ») directement, comme un circuit électronique normal. À l'allumage, l'état doit être forcé par la broche RESET, car il n'y a pas d'autre moyen électronique d'assurer la cohérence des états du système. Des charges rémanentes aux grilles des MOSFET, des défauts de fabrication ou des résistances parasites peuvent influencer le contenu de chaque bit de mémoire, avant qu'on les force avec des valeurs données.

Les autres FPGA à base de cellules SRAM sont aussi dans un état indéterminé à la mise sous tension, mais cela est masqué par une fonctionnalité. En effet, on n'a pas toujours besoin d'activer de broche RESET pour initialiser l'état de tous les registres car la configuration, contenue dans une mémoire externe, contient aussi l'état initial du système. Altera, Lattice, Xilinx et d'autres fabricants permettent ainsi d'initialiser la valeur de démarrage des registres dans le fichier VHDL avec cette syntaxe :

```
signal mon_registre : std_ulogic_vector(7 downto 0) := "01101101";
```

Évidemment, l'électronique du FPGA à SRAM doit quand même déterminer la condition électrique pour démarrer le processus d'initialisation. Le signal de RESET a donc été déplacé et cela peut générer son lot de complexité, qui varie d'un fabricant à l'autre.

En électronique standard, et donc avec les FPGA Actel, l'initialisation automatique des registres n'est pas possible, on doit donc spécifier leur état initial comme une réaction à un signal explicite. En VHDL cela s'écrit en utilisant cette structure canonique :

```
signal mon_registre : std_ulogic_vector(7 downto 0);

...

process (clk, reset)
begin
  if reset='0' then  -- RESET asynchrone
    mon_registre <= "01101101";
  else
    if clk'event and clk='1' then
      mon_registre <= data;
    end if;
  end if;
end process;
```

Qu'est-ce que cela implique pour l'électronique de notre petit IGLOO ? C'est simple : nous devons utiliser une broche d'entrée-sortie standard pour initialiser l'état, au lieu de passer par une broche dédiée comme avec les autres types de FPGA. De ce point de vue, les FPGA d'Actel sont bien les plus proches des technologies ASIC, ils permettent de passer du prototype à l'ASIC avec le moins de modifications et de surprises.

4.3 Resynchronisation du RESET

Puisque l'on parle du VHDL, donc de la circuiterie interne, il faut aussi évoquer la question de la resynchronisation du signal RESET avec l'horloge. Cela peut sembler compliqué, mais la solution est simple et tient en une seule bascule D :

```
process(clk32khz)
begin
  if clk32khz'event and clk32khz='1' then
    reset_interne <= reset_externe;
  end if;
end process
```

Dans ce code, **reset_externe** est le signal provenant de la broche de remise à zéro. **reset_interne**, ou juste **reset** dans les autres entités VHDL de la montre, est le signal de réinitialisation asynchrone des registres.

Une fois entré dans la puce, le signal RESET se propage sur toute la surface pour initialiser tous les registres qui le nécessitent (les états, heures, minutes, secondes, etc.). Cette propagation prend « un certain temps » et il peut arriver que l'horloge change juste à ce moment-là. C'est d'autant plus probable que la fréquence est élevée et la propagation longue, cela peut causer des instabilités ou des aberrations.

La resynchronisation oblige le signal **reset_interne** à changer d'état uniquement après un front montant de l'horloge, pour l'ensemble du FPGA. Il ne va être actif que durant une période d'horloge, l'état du système entier sera correctement défini sans chevauchement temporel.

Il n'est pas rare de trouver des resynchronisations du RESET à 2 ou 3 niveaux sur des circuits complexes à très haute performance, pour s'assurer que la remise à zéro s'effectue sans influence de l'horloge. Dans notre cas, à très faible fréquence, cette resynchronisation est juste une précaution que l'on peut compléter par d'autres techniques, comme un détecteur d'états interdits ou un watchdog.

4.4 Circuits de RESET

La nouvelle question est : quel circuit adopter pour générer le signal de RESET ? Cela dépend de nos exigences, semblables à la plupart des circuits classiques : peu de composants, simple, fiable, avec des composants faciles à trouver... Mais surtout : réduire la consommation au maximum ! Si elle pouvait être nulle, ce serait même idéal.

Historiquement, à l'époque des circuits TTL, NMOS et CMOS en boîtier DIP, les techniques étaient très simples. On partait du principe que l'alimentation devait passer de 0V à 5V en un temps donné, une milliseconde par exemple, donc la broche de RESET devait passer à l'état haut un peu plus tard. Une simple cellule RC (une résistance chargeant un condensateur) convient et le délai est déterminé au moyen de la formule t=RxC.

Le temps de déclenchement est plus complexe que cela car la tension d'activation du signal dépend de la tension d'alimentation et d'autres paramètres (voir l'article sur les afficheurs LCD page 21). Mais pour faire simple, une résistance

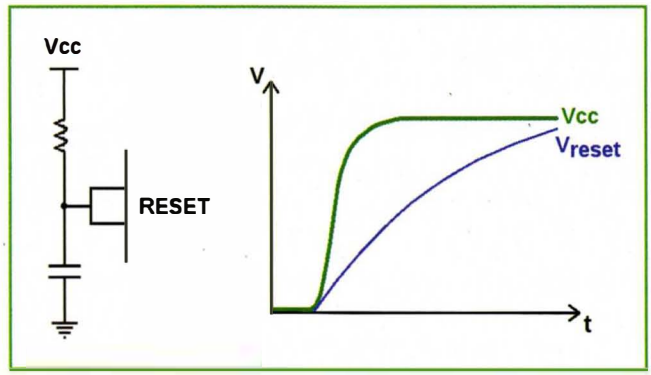


Figure 10 : Une simple cellule RC contrôle la broche de RESET et retarde l'arrivée de la tension d'alimentation.

1MΩ en série avec une capacité de 100nF convient pour retarder la tension d'alimentation d'un dixième de seconde environ.

Évidemment, dans la vie de tous les jours, c'est plus compliqué. Par exemple, que se passe-t-il lorsque l'alimentation a des fluctuations, descend en-dessous de la tension de fonctionnement et rend le système instable ? Notre circuit RC n'a pas le temps de réagir et est inutile. De plus, il ne permet pas de détecter une tension absolue, que se passe-t-il si la tension d'alimentation descend ou monte très lentement ? Rien de bon.

Pour résoudre cela, on a besoin d'une référence de tension et d'un comparateur. Ce sont des fonctions analogiques actives de base, qui sont intégrées de nos jours dans des circuits intégrés spécialisés tels le LM809, MAX809 ou TPS3809 (selon le fabricant). Cette famille de circuits, en plus d'une faible consommation et d'un encombrement réduit, contient aussi un retard électronique pour s'assurer que la remise à zéro est bien effective avant de laisser le système redémarrer.

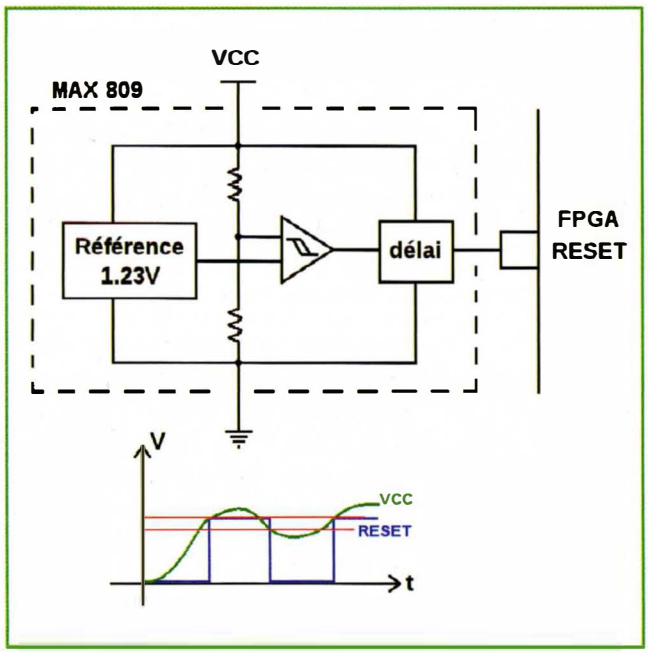


Figure 11 : Un circuit spécialisé de type LM809, MAX809 ou TPS3809 permet de mettre en marche un système proprement et le protège des fluctuations d'alimentation.

Cette sophistication n'est pas gratuite. La consommation de ce type de circuit de RESET peut varier de 10µA à 100µA, ce qui est insignifiant dans la plupart des systèmes mais pas pour nous, puisque c'est l'ordre de grandeur de la consommation de notre FPGA et on voudrait bien s'en passer.

Pour notre montre, nous allons choisir la simplicité, accepter quelques imperfections (c'est juste une montre) et exploiter les fonctionnalités et propriétés de notre FPGA. Celui-ci peut fonctionner à une tension bien plus faible que la tension d'alimentation de 3,3V : si l'alimentation descend à 2V par exemple, il y a encore assez de marge pour que le cœur fonctionne et conserve les données, même si les ports d'entrées-sorties sont désactivés.

Les broches recèlent aussi de fonctions que nous pouvons utiliser d'une manière différente de la fonction originale. Par exemple, l'IGLOO dispose de résistances configurables (pull-up ou pull-down), pour empêcher les entrées de flotter et les amener à un état déterminé. Nous pouvons les utiliser comme résistance de limitation de courant pour la charge d'un condensateur, ce qui économise un composant discret et de la soudure.

La majorité des circuits électroniques ont aussi des diodes de protection sur leurs entrées, destinées à empêcher la détérioration des transistors par d'éventuelles décharges d'électricité statique. L'une de ces diodes fait circuler un courant de la broche vers le rail d'alimentation si la tension de la broche est supérieure à celle du rail(b).

Remarque

(b) Le principe présenté ici est valable pour les IGLOO et ProASIC3 classiques, mais des versions plus récentes fonctionnent différemment afin de supporter des tensions plus élevées à leurs broches. Lisez bien les documentations !

Ce qui est particulièrement intéressant, c'est que cette diode est en parallèle avec la résistance de pull-up. Cela signifie que le courant allant de l'alimentation vers la broche (pour charger le condensateur) est freiné (et le condensateur se charge lentement). Par contre, lorsque l'alimentation est coupée, la résistance est court-circuitée par la diode (on trouve seulement une chute de tension d'environ 0,3V aux bornes de la diode) et le condensateur se décharge rapidement.

Ainsi, le signal RESET va bien repasser à zéro quasiment immédiatement en cas de coupure d'alimentation, pour retarder la remise en route.

Aussi, la famille IGLOO dispose de triggers de Schmitt qui apportent une hystérésis et évitent un déclenchement intempestif, par exemple à cause de bruits parasites lors de changements très lents de la tension.

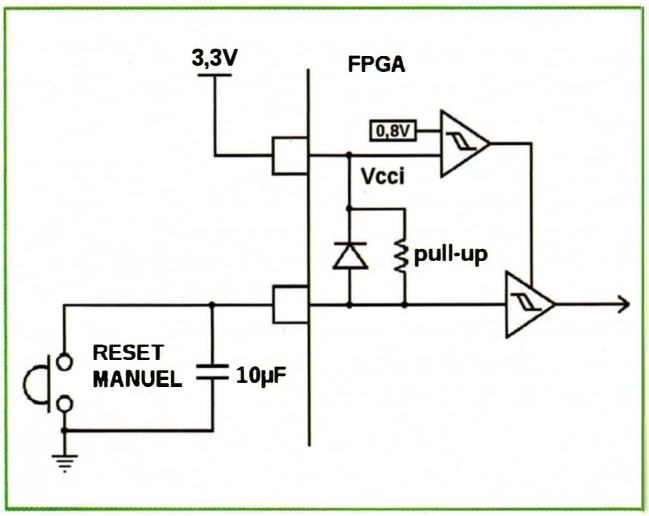


Figure 12 : Notre IGLOO nécessite seulement un condensateur pour sa remise à zéro, en utilisant astucieusement les circuits internes.

Enfin, l'IGLOO a des détecteurs de tension qui surveillent l'alimentation de chaque banque d'entrées-sorties. Les tensions de seuil ne sont pas aussi précises qu'avec un LM809, mais cela remplace presque ce circuit. Nous avons vu plus haut que les cellules logiques lisent la valeur zéro lorsque l'alimentation de la banque d'une broche descend en dessous de 0,8V. nous sommes donc assurés de la réinitialisation du cœur lorsque les tensions sont faibles.

4.5 Et notre cellule RC ?

Quelle doit être la valeur de la capacité de la cellule RC ? On se dit bien sûr que plus le condensateur est petit, mieux c'est. Cela réduit la taille et le coût du circuit. Cependant, les documentations indiquent que la résistance de pull-up est un peu faible :

```
IGLOO Low Power Flash FPGAs DataSheet, p2-37, table Table 2-40 :
I/O Weak Pull-Up/Pull-Down Resistances
Minimum and Maximum Weak Pull-Up/Pull-Down Resistance Values

Vcci : 3,3V
R(weak pull down min) 10KΩ
R(weak pull down max) 45KΩ
R(weak pull up min)   10KΩ
R(weak pull up max)   45KΩ
```

Si on suit la formule t=R×C en prenant le pire cas (10KΩ) il faudra un condensateur de 10µF pour obtenir un délai d'un dixième de seconde.

C'est une valeur bien élevée pour une si petite fonction, mais si on regarde un peu autour du condensateur, on voit la diode de protection de la broche. Si l'alimentation est coupée, le condensateur se décharge, mais dans quoi ?

La diode est connectée à Vcci, qui dans notre cas est connecté au 3,3V, alimentant aussi le régulateur 1,5V. Si le 3,3V est interrompu, le condensateur du RESET va fournir une alimentation de secours de courte durée qui va maintenir le cœur du FPGA sous tension. Cela augmente donc l'immunité du système aux pics de tensions, et sous ce nouvel angle, on peut envisager d'augmenter la valeur à 100µF. Il faudra juste être un peu patient lors de la mise sous tension ou lors des remises à zéro manuelles.

Conclusion

Le module IGLOO est maintenant prêt ! Le rôle et l'utilisation des broches les plus importantes ne devraient plus être un mystère maintenant. Les conditions indispensables pour le fonctionnement (en particulier les alimentations et l'initialisation) sont réunies et le prochain article parlera de la mise en œuvre et des périphériques sur la carte d'accueil. On peut noter que la majorité des informations fournies ici s'appliquent aux autres produits Flash Actel (ProASIC3, Fusion et leurs dérivés), même s'il faut toujours vérifier que chaque détail s'applique à votre version de puce. Un IGLOO n'est pas un IGLOO nano ou un IGLOO PLUS, il y a de subtiles différences.

Les FPGA sont parfois très intimidants et j'ai essayé de montrer que l'IGLOO est moins magique qu'il n'en donne l'air. Armé des bonnes informations (telles que cet article et un bon angle de lecture des documentations officielles), on peut réaliser facilement un système évolué sans perdre trop de temps, d'argent ou de santé mentale.

Travailler en comptant chaque microampère peut sembler rébarbatif, mais cela nous ouvre les yeux sur ce qu'il se passe vraiment dans nos circuits. Où passe le courant de la pile et qu'est-ce qu'on peut éliminer ou modifier pour étendre encore l'autonomie ? Il existe de nombreux trucs et astuces pour y arriver mais pour les comprendre, il faut aussi comprendre les techniques de base de l'électronique. Cela va nous occuper pendant encore quelques articles !

En attendant la suite et si vous êtes intéressé, n'hésitez pas à consulter les documentations de ce FPGA sur le site du fabricant : http://www.actel.com/products/igloo/docs.aspx. Et encore merci à Laura pour les illustrations ! ■

POLYCAPROLACTONE / PCL :
UNE MATIÈRE POUR LES NULS EN MÉCANIQUE

On a beau faire des efforts, dans certains domaines on est moins doué que dans d'autres, sans doute par manque de réel intérêt pour l'activité en question. Pour ma part, la mécanique fait partie de ces domaines. Dès qu'il s'agit de modeler ou réaliser des éléments mécaniques, cela devient du grand n'importe quoi, pas solide pour deux sous et généralement peu fiable, fragile et laid. Heureusement, il existe des matériaux magiques pour compenser un peu (sauf pour la laideur).

La polycaprolactone également appelée PCL, un polyester biodégradable (non toxique), possède une faible température de fusion (60° C) qui permet de modeler des formes et des supports à la main très facilement. La PCL est plus connue sous les noms et marques comme ShapeLock, InstaMorph, Friendly Plastic ou encore Polymorph Thermoplastic. Elle se trouve dans les boutiques de loisirs créatifs ou, tout simplement, sur eBay.

La PCL est vendue sous la forme de granules blancs translucides. Ce polyester est réutilisable, mais bien plus facile d'usage sous cette forme.

Pour modeler la PCL, il faut la chauffer à 60°C. Le plus simple et le moins risqué est alors d'utiliser de l'eau à cette température pour obtenir un réchauffement homogène.

Les grains de PCL vont alors devenir transparents et ramollir à mesure qu'ils approchent de la température optimale.

Après quelques dizaines de secondes seulement, la PCL est prête à être utilisée. On peut constater la différence de transparence avec les grains à température ambiante.

Une fois retirés de l'eau chaude, les grains ramollis se collent les uns aux autres et il devient possible de travailler la pâte pour en extraire l'eau et l'air.

On modèlera la forme souhaitée très facilement. Si la température baisse, il suffit de replonger l'ensemble dans l'eau à 60°C (et non, ceci n'est pas une patte de poulet).

Ainsi, la webcam endommagée se voit équipée d'un support amovible solide. Notez la partie gauche, encore translucide car non entièrement refroidie.

Les résultats ne sont pas forcément très beaux, mais ils sont plus faciles à obtenir qu'un assemblage/découpage de pièces.

Une fois froide, la PCL peut être percée (attention à l'échauffement) et sciée/découpée. La gamme de réalisations possibles est presque infinie. Qu'il s'agisse de pièces mécaniques, de supports ou de fixations, on a que l'embarras du choix. Pour le reste, imaginez simplement ce qu'il devient possible de faire en mélangeant PCL et servo ou encore PCL et leds ! ■

HP PRE3 : ANDROID M'A TUER ?

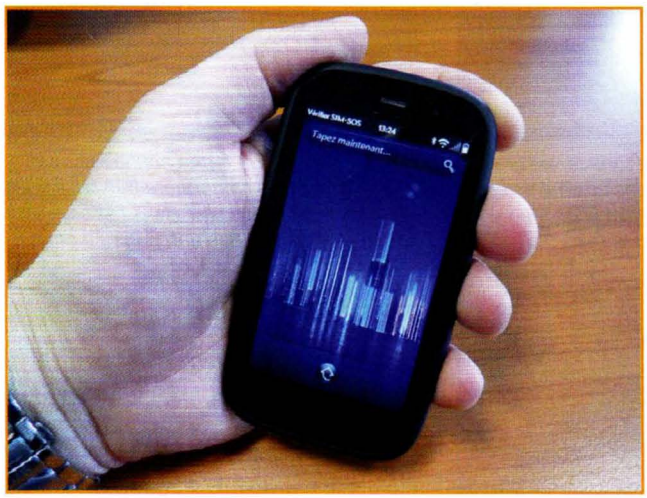
Connaissez-vous le smartphone Pre3 de HP ? Moi, je ne le connaissais pas, ou pas vraiment. Puis j'ai entendu l'annonce de l'arrêt de la production des périphériques WebOS par HP et, je crois bien le même jour, je craquais pour un Pre3 sur Expansys car il était en promotion. Disons le clairement : je m'en veux ! Non de l'avoir acheté, mais plutôt de ne pas avoir porté plus tôt mon attention sur cette famille de périphériques...

J'ai eu un PalmPilot 1000, un Palm IIIxe et enfin un Treo 600. Cela remonte à plus de temps qu'il n'est raisonnable de compter. L'abandon du Treo coïncida avec la fin de l'ère des PDA. Seuls quelques courtiers en assurance ou agents immobiliers exhibaient encore ces assistants numériques en marchant d'un pas décidé de leur immaculée Range Rover à la terrasse d'un restaurant. N'exerçant ni l'une ni l'autre de ces activités, c'est une collection de Nokia en tous genres qui se succédèrent et les fonctionnalités plus avancées était délivrées par des lappys (laptops) IBM/Lenovo de plus en plus accessibles/transportables.

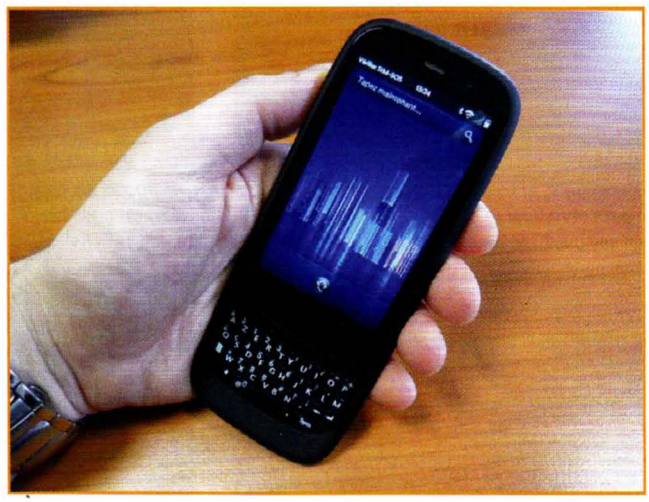
Plus récemment est arrivé mon premier smartphone à écran tactile sous Android, puis le second, puis la tablette Android, etc., etc., etc. Attendez une seconde ! Les relations avec ces « jouets », c'est un peu comme les bons amis, avec qui on partage du temps et dont on s'éloigne jusqu'au jour où on se demande : « Mais ? Que sont-ils devenus ? ».

Palm Computing a réussi à survivre sans moi, qui en aurait douté ? D'autres Treo ont succédé au 600 dont certains sous Windows Mobile et ce jusqu'à l'arrivée du Palm Pre (prononcé à l'anglaise, « pri »), sous le nom de code Castle, en juin 2009.

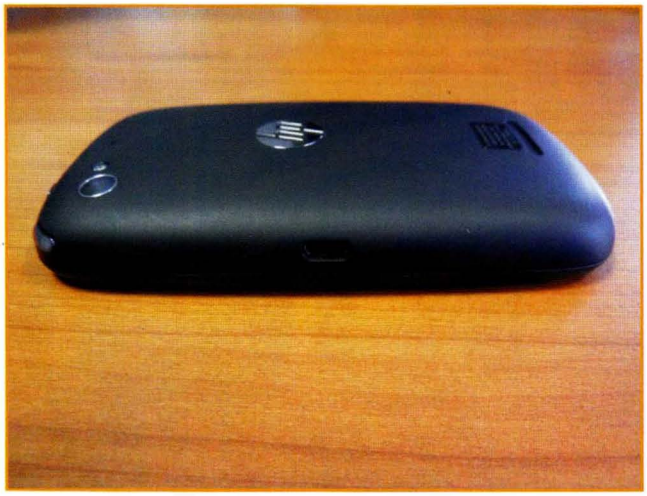
Mais il était sans doute déjà trop tard pour affronter l'Android de Google annoncé en avril de la même année. Le Pre, tout comme les smartphones « prophétisés » par Google, intégrait des fonctions standards comme un appareil photo, un lecteur multimédia, un GPS, et une connectivité Internet (SMS, mail, Web) via WiFi et les services de données



Le HP Pre3, version « grand frère » du Veer. Un smartphone aux caractéristiques classiques mais utilisant un système d'exploitation basé sur GNU/Linux : WebOS 2.2. Ne le cherchez pas, l'arrêt de la fabrication des périphériques WebOS par HP a été annoncé quelques jours après la mise en vente de cet appareil.



Le Pre3 dispose d'un écran coulissant découvrant un clavier mécanique. Ici, la version AZERTY n'est pas sans rappeler les claviers des Palm Treo dans leur organisation et par la présence de la fameuse touche « Sym ». Le Pre3 conserve avec lui un peu de l'esprit du Palm même s'il est très différent physiquement et logiciellement.



La coque arrière du Pre3 est couverte d'une couche de silicone qui apporte un touché agréable. Le boîtier tout en rondeur ajoute à l'impression général de bioformes. Deux défauts cependant : il est épais et relativement lourd.

Le Pre3 dispose d'un clavier mécanique. Troublant au premier abord avec ses touches molles et minuscules, il s'avère finalement relativement agréable et précis.

de l'opérateur. Tout cela était proposé sous la forme d'un système GNU/Linux auquel, tout comme Android, s'ajoutaient un environnement graphique, un SDK permettant le développement d'applications et un service de téléchargement d'applications gratuites ou payantes.

L'historique du créateur du Palm Pilot est plein de rebondissements. C'est en 1992 que Jeff Hawkings crée Palm Computing. La société sera ensuite rachetée par US Robotics et c'est en 1996 que le Palm Pilot est lancé sur le marché. Un an après, 3Com rachète US Robotics puis la société française Smartcode Technologies qui devient Palm Computing Europe. 2000 est l'année où Palm est scindé et arrive sur le marché boursier en tant que tel (Nasdaq). Fin 2003, la division système d'exploitation, après être devenue PalmSource en 2001, est introduite en bourse. Tout va pour le mieux, si bien que Palm rachète son concurrent Handspring la même année. 2007, un fond d'investissements californien prend 25% du capital de Palm et en 2009 est annoncé le Pre sous WebOS 1. 2010 marque le rachat par HP (Hewlett-Packard) pour quelques 1,2 milliards de dollars et Palm devient une filiale du géant HP. Et, sans doute ne voyant pas de moyen de se faire une place entre Android et iOS, c'est au cours du premier semestre de cette année (2011) que HP annonce la fin de la production des périphériques sous WebOS.

Trois (ou quatre) périphériques se sont succédés peu avant la fin de l'aventure : le Pre en 2009, le Pre Plus mi-2010, le Pre2 fin 2010 et enfin, après le rachat par HP, le HP Pre3 annoncé début 2011 arrive finalement en août de la même année (et peu de temps après dans ma main). Cette quatrième génération de périphériques utilise HP WebOS 2.2 et accompagne un autre périphérique d'HP, le TouchPad sous WebOS 3.0.

Le Pre avait tout pour réussir et concrétisait le fantasme de beaucoup d'utilisateurs GNU/Linux : un vrai mobile utilisant comme base un système embarqué open source. De plus, WebOS proposait, dans les premières versions, une compatibilité ascendante avec Palm OS, permettant ainsi de faire fonctionner les milliers d'applications de cette plateforme. Palm, avant tout le monde avait compris que ce qui valorisait un PDA n'était pas tant ses caractéristiques techniques et sa puissance que le catalogue d'applications disponibles. C'est maintenant un principe admis, mais Palm en avait fait depuis longtemps, son argument de vente. Une autre caractéristique notable, que certains n'ont implémenté (ou découvert, allez savoir) que bien plus tard, est la gestion effective du multitâche sur smartphone. Dans WebOS, un ingénieux système de « cartes », dans l'interface graphique, présente les écrans des applications actives sous la forme de petites cartes qu'il suffisait de glisser du doigt.

Le SDK étant disponible, sans inscription et gratuitement, aussi bien pour Windows, Mac OS X et GNU/Linux, une communauté de développeurs s'est rapidement formée sous la bannière de WebOSInternals. Ainsi ont été créées de nombreuses applications *homebrew* et portés des jeux comme Doom, Quake et Duke Nukem 3D.

WebOS n'est pas mort, mais HP ne compte plus fabriquer de périphériques de ce type, ce qui met à mal la plateforme quoi qu'on en dise. Et pourtant, à mon goût, bien plus qu'Android, WebOS est le système GNU/Linux pour mobiles que les développeurs et les hackers attendaient. L'annonce de l'arrêt de la fabrication de périphériques WebOS par HP, le lendemain de la commercialisation du Pre3 (17/08), précéda une autre annonce visant à écouler le stock de TouchPad, bradés à 99 euros. Bien entendu, rapidement, le périphérique ne fut plus disponible après une telle vente flash. On retrouve à présent les TouchPad sur eBay, par exemple, à quelques

300 euros et quelques Pre3 à plus de 250 euros. Après la ruée, c'est la pénurie et certains acheteurs ayant sauté sur l'occasion inespérée, n'hésitent pas à en jouer. Chanceux, j'ai donc l'un des rares Pre3 AZERTY existants, mais je ne désespère pas de trouver un TouchPad à un tarif raisonnable...

1 Tour du propriétaire

Avant toutes choses, voici les spécifications techniques de la bête :

- Réseaux : Quad band GSM (850/900/1800/1900MHz), Tri band 3G/UMTS networks (900/1900/2100MHz)
- Processeur : 1.4 GHz Qualcomm Snapdragon MSM8x55
- Processeur graphique : Adreno 205
- Mémoire : 512Mo RAM
- Stockage : 8Go ou 16Go interne
- Périphérique d'entrée : Touchscreen, clavier (sous l'écran à glissière) et zone tactile
- Affichage : écran 3,58 pouces multitouch (capacitif) 24bits, 480×800 pixels
- Caméra : Arrière, 5 mégapixels, auto-focus, flash LED avec enregistrement vidéo HD
- Caractéristiques physiques : dimensions de 111 mm x 64 mm x 16 mm, poids de 155g
- Connectivité : Wifi 802.11 b/g/n, Bluetooth 2.1, HSPA+, EvDo

Comme vous pouvez le constater, le smartphone est tout à fait dans les spécifications courantes du marché. C'est là un matériel tout à fait respectable malgré une épaisseur un peu dérangeante.

Mais c'est à l'usage qu'on se rend compte des qualités du Pre3. Après une mise en service initiale assez pénible (nécessité d'utiliser une carte SIM et procédure d'enregistrement longue et complète), on découvre une interface bien pensée et surtout réellement multitâche. Le tout est entièrement piloté par une petite zone tactile située au bas de l'écran. Une gestuelle simple permet de contrôler l'interface. Un mouvement de bas en haut permet d'afficher la liste des applications et si vous vous trouvez dans l'une d'elles, le même mouvement la réduit légèrement, la présentant sous la forme d'une carte. Ainsi, vous pouvez à nouveau accéder à la liste des applications et en lancer une nouvelle.

L'écran bascule ainsi du plein écran à la gestion de « cartes » vous permettant de switcher horizontalement de l'une à l'autre. Pour quitter une application ? C'est tout simple, il suffit de glisser la carte vers le haut de l'écran et sa fermeture est ponctuée d'une notification sonore. On utilise donc l'interface dans les deux dimensions. C'est une logique que l'on retrouve également dans la liste des applications. Chaque carte propose un type d'application dont la liste peut défiler verticalement. Un défilement horizontal permet

L'ouverture de l'appareil nécessite un certain sang-froid. La coque arrière est, en effet, attachée par quelques clips. Les boutons au-dessus de l'appareil traversent la coque et ceux sur le côté sont mobiles. Bref, on a vraiment l'impression de tout casser en tentant l'opération et on souffle une fois l'appareil ouvert en espérant ne plus jamais avoir à recommencer.

de passer d'une catégorie à une autre. L'ensemble est fluide et très agréable. Difficile de dire qu'on n'est pas bluffé et on s'étonne de ne pas avoir rencontré cela dans d'autres OS pour mobile et en particulier dans Android (notez cependant que je n'ai pas pratiqué tous les OS du marché et que mon jugement est donc très imparfait). Cette présentation est telle qu'on se surprend, une fois revenu à son Samsung Galaxy, à essayer les fameuses *gestures* pour quitter une application ou basculer de l'une à l'autre.

Enfin, sur l'écran d'accueil, une barre horizontale dans la partie inférieure permet de glisser jusqu'à 5 applications les plus utilisées. Pour cela, on revient au classique toucher-maintenir permettant de déplacer les icônes.

2 Dis ?! Tu me montres ton Linux ?

On ne se refait pas. A peine l'interface découverte vient l'envie irrésistible de gratter cette belle interface pour découvrir ce qui se cache en-dessous. La première étape consiste donc à activer le mode développeur. Pour cela, inutile de jouer du bootloader ou d'utiliser une procédure hasardeuse, tout ceci est prévu d'origine. Il nous suffit ainsi d'utiliser le champ de recherche placé en haut de l'écran principal et de saisir « webos20090606 ». Magiquement, un bouton « Developer Mode Enabler » nous permet d'accéder à un écran caché de la configuration permettant de basculer un switch « Developer Mode » à « OUI ». Ceci fait et après une pression sur le bouton « Reset the Device » le mobile redémarre avec le mode développeur activé. C'est tout.

Il vous faudra ensuite télécharger NovaCom (et par la même occasion, le SDK) sur le site officiel de HP (developer.palm.com). Plusieurs systèmes sont supportés dont Ubuntu et par conséquent le système dont celui-ci découle : Debian GNU/Linux (mon mien à moi que j'utilise). L'installation affiche un message d'avertissement concernant le démarrage de **/opt/Palm/novacom/novacomd**. Compréhensible, l'erreur vient du fait que les scripts de lancement sont faits pour UpStart, le remplaçant d'init SysV chez Ubuntu (l'une des quelques innovations par rapport à Debian). Trop pressé d'explorer WebOS, on remettra à plus tard l'écriture d'un vrai script d'init utilisant **start-stop-daemon** et on lance sauvagement le démon via **sudo /opt/Palm/novacom/novacomd** après connexion du Pre3 au PC en USB.

L'accès à la ligne de commandes de Web OS se fait un peu à la manière de ce que propose Android avec **adb shell**. Ici, cependant, un démon se charge de la communication, via USB et un autre binaire vous fournira l'accès au shell **root** (oui **root**, vous lisez bien). Sur l'écran d'activation du mode développeur, en y retournant de la même manière, vous avez l'opportunité de spécifier un mot de passe pour protéger cet accès dangereux. Faites-le, c'est plus prudent, ne serait-ce que pour vous protéger de vos propres erreurs de manipulation.

Au lancement du démon, celui-ci va chercher un périphérique compatible et négocier la communication :

```
% sudo /opt/Palm/novacom/novacomd
[2011/9/7 20:51:40] novacom_usb_findandattach_thread:565: usb_handle 0x00000000,
bus=002 dev=042
[2011/9/7 20:51:40] novacom_go_online:ea5256930f82e07532f5abc18d092a0623b12c1a
[2011/9/7 20:51:40] parse_devdata:658: id::mantaray-linux
[2011/9/7 20:51:40] novacom_go_online:702: id:mantaray-linux/()
[2011/9/7 20:51:40] parse_devdata:658: sn::6eef7460c5
[2011/9/7 20:51:40] novacom_go_online:710: sn:6eef7460c5
[2011/9/7 20:51:40] dev 'ea5256930f82e07532f5abc18d092a0623b12c1a' via usb type
mantaray-linux
[2011/9/7 20:51:40] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=80
[2011/9/7 20:51:40] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=60
[2011/9/7 20:51:40] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=60
[2011/9/7 20:51:40] Got RESET packet, process it?, rxid=0x0e6b7867,
sessionid=0x0e6b7867
[2011/9/7 20:51:40] Got RESET packet, restart...
[2011/9/7 20:51:40] novacom_go_offline:744: going offline
[2011/9/7 20:51:40] removing id ea5256930f82e07532f5abc18d092a0623b12c1a
[2011/9/7 20:51:40] novacom_usbll_prepare_tx_packet:410: novacom device was deleted
while waiting for tx
[2011/9/7 20:51:40] novacom_deviceinit_thread:1140: hit create channel/device going
away race
[2011/9/7 20:51:40] novacom_deviceinit_thread:1142: socketchan::created: 0x89bfc38,
socket 0, novacom -1, device 0x89b47c0
[2011/9/7 20:51:41] novacom_go_online:ea5256930f82e07532f5abc18d092a0623b12c1a
[2011/9/7 20:51:41] parse_devdata:658: id::mantaray-linux
[2011/9/7 20:51:41] novacom_go_online:702: id:mantaray-linux/()
[2011/9/7 20:51:41] parse_devdata:658: sn::6eef7460c5
[2011/9/7 20:51:41] novacom_go_online:710: sn:6eef7460c5
[2011/9/7 20:51:41] dev 'ea5256930f82e07532f5abc18d092a0623b12c1a' via usb type
mantaray-linux
[2011/9/7 20:51:41] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=80
[2011/9/7 20:51:41] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=60
[2011/9/7 20:51:41] novacom_deviceinit_thread:1142: socketchan::created: 0x89b4a78,
socket 0, novacom 80, device 0x89b47c0
[2011/9/7 20:51:41] tokenstorage_path:99: path /root/.nova/
[2011/9/7 20:51:41] tokenstorage_read:183: path /root/.nova/
ea5256930f82e07532f5abc18d092a0623b12c1a
[2011/9/7 20:51:41] tokenstorage_read:188: unable to open '/root/.nova/
ea5256930f82e07532f5abc18d092a0623b12c1a' file
[2011/9/7 20:51:42] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=61
[2011/9/7 20:51:43] novacom_usb_tx_thread:370: usbll(0002002a) wrote tx packet len=61
```

Dans un autre terminal, on utilisera :

```
% novaterm -c login -r mot_de_passe
% novaterm
root@DenisbodorHPPre3:/# uname -a
Linux DenisbodorHPPre3 2.6.32.9-palm-rib #1 PREEMPT 155 armv7l GNU/Linux
```

Notez l'authentification préalable avec les options **-c login -r mot_de_passe**. La première permet de spécifier le service concerné, **login** ou **logout** pour la connexion et **add** et **remove**. L'utilisation de la commande **novaterm** seule permet d'obtenir un shell sur le mobile.

Comme vous pouvez le voir, un noyau Linux 2.6.32 anime le système et après quelques commandes d'usage on découvre avec enthousiasme un système GNU/Linux presque standard :

- Système de fichiers en ext3
- Utilisation du device mapper

```
# ls -l /dev/mapper/
crw-rw---- 1 root  root   10, 25 Jan  6  2010 control
brw------- 1 root  root   254,  8 Aug 31 11:13 store-cryptodb
brw------- 1 root  root   254,  9 Aug 31 11:13 store-cryptofilecache
brw------- 1 root  root   254,  5 Jan  6  2010 store-filecache
brw------- 1 root  root   254,  3 Jan  6  2010 store-log
brw------- 1 root  root   254,  6 Jan  6  2010 store-media
brw------- 1 root  root   254,  4 Jan  6  2010 store-mojodb
brw------- 1 root  root   254,  0 Jan  6  2010 store-root
brw------- 1 root  root   254,  7 Aug 31 11:13 store-swap
brw------- 1 root  root   254,  2 Jan  6  2010 store-update
brw------- 1 root  root   254,  1 Jan  6  2010 store-var
```

- Utilisation d'UpStart (??)
- Getty (??)
- BusyBox v1.17.3 avec la plupart des commandes utiles
- WPA supplicant

```
# iwconfig eth0
eth0      AR6K 802.11nag  Nickname:""
          NWID:off/any  Mode:Managed  Bit Rate=1 Mb/s   Tx-Power=0 dBm
          Sensitivity=0/3
          Retry:on   RTS thr=0 B   Fragment thr=0 B
          Encryption key:off
          Power Management:on
          Link Quality:8/94  Signal level:-87 dBm  Noise level:-96 dBm
          Rx invalid nwid:0  Rx invalid crypt:14  Rx invalid frag:0
          Tx excessive retries:358  Invalid misc:0   Missed beacon:944
```

- Une prise en charge des leds via **/sys/class/leds/** (mon dieu, c'est bô !)
- Utilisation de PulseAudio (??)

```
# pulseaudio --dump-conf
### Read from configuration file: /etc/pulse/daemon.conf ###
daemonize = no
fail = yes
high-priority = yes
```

```
nice-level = -4
realtime-scheduling = yes
realtime-priority = 5
allow-module-loading = no
allow-exit = no
use-pid-file = yes
system-instance = yes
cpu-limit = no
[...]
```

- SQLite 3

- Gestion de paquets via IPKG

```
# ipkg list_installed
[...]
wpasupplicant - 0.6.10-31 -
xz-embedded - 1.0-1 -
yajl - 1.0.7-3 -
zeroconfigd - 1.0.1-15 -
zlib - 1.2.3-r2 -
```

- Utilisation de LVM2

```
# lvscan
ACTIVE '/dev/store/root' [568.00 MB] inherit
ACTIVE '/dev/store/var' [64.00 MB] inherit
ACTIVE '/dev/store/update' [16.00 MB] inherit
ACTIVE '/dev/store/log' [24.00 MB] inherit
ACTIVE '/dev/store/mojodb' [144.00 MB] inherit
ACTIVE '/dev/store/filecache' [136.00 MB] inherit
ACTIVE '/dev/store/media' [5.41 GB] inherit
ACTIVE '/dev/store/swap' [504.00 MB] inherit
```

Même si vous ne trouvez pas de périphériques WebOS (Pre ou TouchPad) à un prix abordable, l'émulateur inclus dans le SDK officiel vous permettra de développer vos applications en attendant. A noter que celui-ci est bien plus réactif que l'émulateur Android pour un même système hôte et pour une résolution équivalente.

Beaucoup d'informations intéressantes données sur *WebOS Internals*, à la rubrique *hardware*, ne se vérifient pas avec le Pre3. Les capteurs (accéléromètre, IR, etc.), par exemple, sont décrits comme étant accessibles via **/dev/input/*** (cf. article sur le sujet dans le présent numéro), ce qui aurait été des plus pratiques. Malheureusement, ce n'est pas le cas. Il semblerait que les informations données concernent le Pre et le Pre2 et non cette toute dernière (dans tous les sens du terme) génération.

Quoi qu'il en soit, nous constatons ici que nous avons bien affaire à un système Linux, bien moins « adapté » que ne l'est Android par Google, ce qui ouvre des perspectives intéressantes en termes de développement système.

3 La couche du dessus : SDK, PDK et outils

WebOS s'accompagne d'un SDK complet et très intéressant. Décidément, plus on découvre la plateforme, moins on comprend pourquoi elle ne s'est pas faite au moins une petite place sur le marché. **novaterm** et le démon **novacomd** ne sont qu'une petite partie de l'environnement de développement et d'émulation. De plus, tout comme avec **adb** d'Android, le démon est en mesure de gérer plusieurs périphériques simultanément, dont des instances de l'émulateur.

Pour émuler WebOS et un périphérique Pre, vous devrez avoir installé VirtualBox. Un certain nombre d'images systèmes sont installées avec le SDK, mais il est possible d'en télécharger d'autres directement sur le site officiel pour développeurs d'HP (page de téléchargement du SDK, tout en bas). On installera ces images, fournies en archive Zip, sans désarchiver, avec **palm-emulator --install ~/OS/nova-cust-image-sdk2210.vmdk.zip** par exemple.

La simple invocation de **palm-emulator** seul vous permettra, ensuite, de choisir l'image à utiliser. Attention, si vous êtes utilisateur de QEMU/KVM, assurez-vous que le module noyau d'accélération (**kvm-amd** ou **kvm-intel**) ne soit pas chargé. Le *HP WebOS Emulator* repose entièrement sur VirtualBox et ne présente que peu de différences avec un périphérique physique. Bien entendu, le clavier n'est qu'approximativement similaire et la zone de gesture n'est pas présente. La correspondance suivante s'applique donc :

- [Alt] : touche Option
- [Fin] : gesture vers le haut
- [Echap] : gesture de droite à gauche (*swipe back*)
- [Début] : réduit/agrandit une carte (interface)
- Flèche gauche/droite : bascule d'une application à l'autre en vue cartes
- [Ctrl] : touche Sym

Lorsque plusieurs périphériques ou émulateurs sont accessibles, vous pouvez obtenir la liste avec :

```
% novaterm -l
56911 b58f87537c782a7350ec0dcead2bf64836b8celf tcp emulator
51271 ea5256930f82e07532f5abc18d092a0623b12c1a usb mantaray-linux 0a92eb2ebb
```

Vous choisirez ensuite le périphérique via l'option **-d** suivi de son identifiant ou de l'un des alias, et ce avec n'importe quel outil du SDK. Exemple, **novaterm -d usb** vous connectera au shell **root** d'un périphérique physique et **novaterm -d emulator** à la machine virtuelle.

L'installation d'applications et la gestion d'un grand nombre de fonctionnalités logicielles peuvent être effectuées via un jeu complet de commandes **palm-*** installées par le SDK. Installer une application au format IPKG, par exemple, se fera via :

```
% palm-install -d tcp com.palmdts.enyo.helloworld_1.0.0_all.ipk
installing package com.palmdts.enyo.helloworld_1.0.0_all.ipk
on device "emulator" {b58f87537c782a7350ec0dcead2bf64836b8ce1f}
tcp 56911
```

Nous pouvons, avec **palm-install**, lister les applications en présence :

```
% palm-launch -d tcp  -l
listing applications on device "emulator"
{b58f87537c782a7350ec0dcead2bf64836b8ce1f}
tcp 56911
[...]
com.palm.app.vpn 2.2.0 SDK "VPN"
com.palm.app.wifi 2.2.0 SDK "Wi-Fi"
com.palm.app.youtube 2.2.0 SDK "YouTube"
com.palmdts.enyo.helloworld 1.0.0 "Enyo HelloWorld"
```

Lancer celle de notre choix :

```
% palm-launch -d tcp com.palmdts.enyo.helloworld
launching application com.palmdts.enyo.helloworld on
device "emulator" {b58f87537c782a7350ec0dcead2bf64836b8ce1f}
tcp 56911
```

Et, enfin, la supprimer :

```
% palm-install -d tcp -r com.palmdts.enyo.helloworld
removing package com.palmdts.enyo.helloworld on device
"emulator" {b58f87537c782a7350ec0dcead2bf64836b8ce1f}
tcp 56911
```

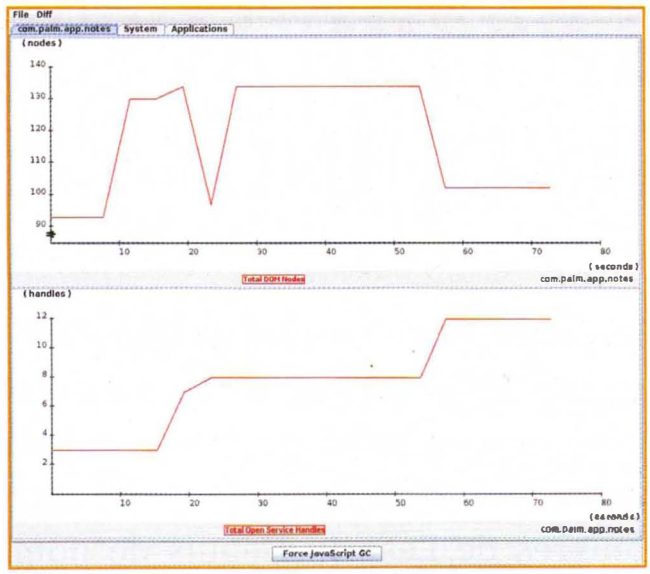
L'impression générale à l'utilisation du SDK est une facilité et une ouverture très agréable. On sent réellement que le fabricant fait confiance au développeur et à l'utilisateur. C'est difficile à expliquer et cela tient en de petites choses comme l'accès **root**, la simplicité des outils, l'utilisation modérée de Java ou encore des outils en ligne de commandes dont l'option **--help** est réellement utile et accompagnée d'exemple.

Nous reviendrons sur WebOS et ce périphérique dans un prochain numéro où nous découvrirons l'environnement de développement et les frameworks Mojo et surtout le tout nouveau Enyo.

Conclusion

Que dire pour terminer cet article sinon exprimer un profond sentiment de tristesse. Nous avons là un système bien plus « standard » qu'Android, une interface graphique très sympathique, un SDK agréable et des outils de développement qui combleraient de bonheur la majorité des utilisateurs UNIX, Windows et Mac OS X (ces deux derniers peut-être dans une moindre mesure).

Bon nombre d'ingrédients semblaient donc réunis pour un succès même si la concurrence est rude sur le marché. Pour expliquer cet échec, la seule réponse tient en peu de choses : problème de timing et de marketing. Je me considère comme le client typique pour ce produit. Ayant déjà possédé



Le SDK fournit de quoi écrire ses programmes aussi bien en JavaScript qu'en C (PDK), les packager, les débugger et les profiler. Il s'agit d'un SDK bien conçu et disponible pour Mac OS X, Windows et GNU/Linux. Ici, l'analyseur de performance palm-worm.

un Palm je connaissais la réputation du fabricant et je suis un grand amateur de ce genre de « jouets ».

Le Pre n'a toujours été qu'une rumeur pour moi ou un essai, un prototype, sinon un vaporware. « Pre », quel drôle de nom pour un produit terminé. Quelqu'un n'a pas fait son travail au marketing ce jour-là. Mais je pense que Palm a surtout été victime de son succès et que l'image de la vieille société qui a déjà eu son heure de gloire n'est pas sans rapport avec l'échec du matériel WebOS. Difficile avec ce type d'aura de convaincre les utilisateurs avancés, leaders d'opinion, de ne pas céder aux sirènes de cette société de Mountain View qui ose tout et propose une plateforme totalement nouvelle. L'histoire de l'informatique est pleine de bons concepts qui n'ont pas eu le succès mérité, il n'y a là rien de nouveau.

Je dirai, en conclusion, que le Pre aurait mérité plus d'attention et qu'il aurait pu, alors, avoir une chance d'évoluer. WebOS, quant à lui, en particulier du fait de l'utilisation de JavaScript et du C, est en mesure d'être une alternative à Android, Windows Mobile, iOS et autres... à condition qu'il y ait un matériel pour le faire fonctionner (quelqu'un aurait-il dit « Meego » ?). C'est la dure loi des systèmes embarqués pour la téléphonie mobile et les tablettes. Pour que ma mayonnaise prenne il faut : une communauté (de développeurs ou d'utilisateurs), du matériel d'un ou plusieurs constructeurs, une belle interface, un système solide, et un catalogue d'applications conséquent. Si l'une de ces pièces est manquante, la plateforme commence la course avec un handicap. Si deux font défaut, c'est presque perdu d'avance. WebOS n'avait pas une logithèque importante et aujourd'hui il n'a plus de plateforme matérielle. Tirez vous-même les conclusions qui s'imposent... ■

LES PIC AUSSI ONT LEUR ARDUINO ! C'EST PINGUINO !

par J.P. Mandon & Regis Blanchot

Le projet Pinguino a été créé en 2008 pour répondre à la demande d'étudiants en arts de l'école internationale d'art d'Aix-en-Provence. Le besoin d'intégrer à des œuvres artistiques des systèmes programmables est né du croisement entre l'art et la technologie où interactivité et algorithmique côtoient les œuvres de l'esprit. Depuis de nombreuses années, cette école associe à l'enseignement artistique tous les enseignements techniques nécessaires à l'apprentissage des « nouveaux médias ».

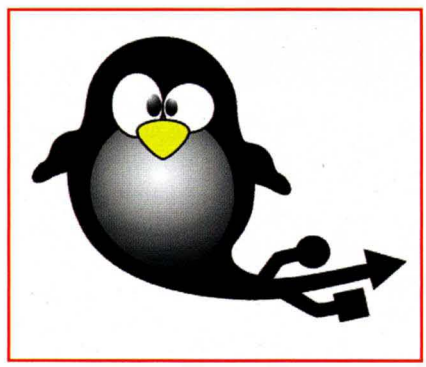
1 L'aventure Pinguino

Il était devenu nécessaire pour répondre à cette demande de créer un système facilement reproductible et utilisable sur tous les systèmes d'exploitation. Bien qu'une offre existe déjà, Arduino ou autres cartes compatibles, aucune ne satisfaisait le cahier des charges défini pour Pinguino :

- Open hardware et open software,
- Construction à la portée d'un électronicien débutant.

Il est né de cette réflexion une carte que nous avons voulue aisément reproductible. Aucun composant de surface n'est utilisé et un environnement de développement écrit en langage Python permet de se familiariser avec le langage C, pour ensuite évoluer vers des environnements de développement plus puissants comme Eclipse ou Code::Blocks.

Les étudiants utilisent désormais Pinguino dans des domaines aussi divers que la 2D, la 3D ou la vidéo, comme un outil de l'interactivité.



1.1 Arduino/ATMEL vs Pinguino/MICROCHIP

Le choix d'utiliser un microcontrôleur PIC plutôt qu'un AVR a été fait dès le départ, principalement parce que celui-ci disposait d'un module USB intégré évitant ainsi l'emploi d'un convertisseur USB/RS232 et donc, utilisable directement sur la plupart des ordinateurs actuels.

Bien que n'ayant aucun lien avec Microchip, Pinguino est devenu au fil du temps et de sa popularité une brique dans le monde du PIC. Aujourd'hui, bien que toujours farouchement indépendant, ce projet bénéficie de l'aide de Microchip.

1.2 Un projet communautaire

Plutôt que concurrent d'Arduino, ce projet s'est voulu compatible mais présentant ses propres spécificités. L'une de celles-ci est d'intégrer les besoins de la communauté de ses utilisateurs au fil des versions. Le module de traduction par exemple, a été réalisé par la communauté vénézuelienne des utilisateurs de Pinguino. La version bêta 8 a permis l'intégration de librairies utilisateurs dans le code, aussitôt suivie de l'intégration des librairies de commandes de servomoteurs et d'affichage LCD réalisées par des utilisateurs de la communauté. Les prochains développements sur PIC32 ont vu de nombreux développeurs rejoindre le groupe créé à l'origine.

Cette communauté s'est enrichie peu à peu de contributeurs plus « professionnels ». Début 2011, la société OLIMEX a rejoint le projet Pinguino pour apporter son expertise dans la construction d'une carte 32 bits. La carte PINGUINO 32 se décline en deux versions. PIC32-PINGUINO est la carte d'entrée de gamme. PIC32-PINGUINO-OTG se voit munie en plus de l'USB OTG (protocole On-The-Go,

qui permet au Pinguino de jouer le rôle du maître (host) ou de l'esclave (device) sur le bus) d'un lecteur de cartes micro-SD intégré et câblé sur l'un des bus SPI du processeur.

Chacune de ces cartes a été conçue pour fonctionner en milieu industriel :

- Processeur PIC32-MX446F256H avec 256k de flash et 32k de RAM,
- 1xSPI, 2xI2C, 2xUART, 45 entrées/sorties,14 entrées analogiques 10 bits,
- Alimentation faible bruit utilisant des convertisseurs CC/CC plutôt que des régulateurs traditionnels, cette particularité autorise des tensions d'alimentations jusqu'à 30V et de faibles pertes (30µA en mode sleep du processeur),
- Chargeur Li-Po intégré sur tous les modèles avec commutation automatique sur batterie si l'alimentation principale est déconnectée,
- Horloge temps réel intégrée sur tous les modèles,
- Régulateur de tension séparé pour la partie analogique et la partie digitale limitant les phénomènes de bruit sur les entrées analogiques,
- Connecteur UEXT pour la connexion de modules OLIMEX (zigbee, wifi, GPS, etc.).

L'apparition de cartes 32 bits n'a pas pour autant stoppé le développement 8 bits. La feuille de route du projet maintient le développement actif d'extensions pour les cartes à base de PIC 18F. Dans les prochains mois par exemple, une carte d'extension permettra de gérer l'USB Host sur les cartes 8 bits et de pouvoir ainsi connecter une carte type 2550 à un téléphone Android.

1.3 Les développements futurs

La communauté des utilisateurs tient clairement aujourd'hui à se démarquer et à développer des fonctionnalités propres à Pinguino. Les futures versions intégreront donc une nouvelle interface utilisateur plus proche de celle d'un environnement de développement.

1.3.1 Personnalisation

La personnalisation sera également développée et le design « processing » peu intuitif mais bien connu des utilisateurs Arduino voisinera avec des thèmes plus complets, voire des thèmes définis par l'utilisateur.

1.3.2 32 bits

Avec le développement sur PIC 32 bits commencé début 2011, la version 10 de Pinguino permettra d'exploiter facilement toute la puissance d'un microcontrôleur 80 MIPS avec 80 entrées/sorties. Les outils sont d'ores et déjà disponibles en téléchargement sur le site dans une version de test mise à jour très régulièrement.

1.3.3 Android

Avec l'avènement du système d'exploitation Android, les téléphones se sont transformés en véritables ordinateurs. Une équipe de développeurs se constitue pour adapter l'environnement de développement à cet OS. Dans un avenir proche, il sera possible de programmer une carte Pinguino directement depuis son smartphone.

1.3.4 pBasic

Toujours dans l'optique de simplifier au maximum l'accès à des systèmes électroniques complexes, Pinguino se verra prochainement doté d'un langage BASIC actuellement en cours de développement. Les cartes Pinguino pourront donc être utilisées et programmées de manière simple et en totale autonomie (sans ordinateur). Ce projet intéressera particulièrement le milieu éducatif et les plus jeunes. L'objectif est de proposer un petit ordinateur capable d'interagir avec le monde physique pour quelques dizaines d'euros.

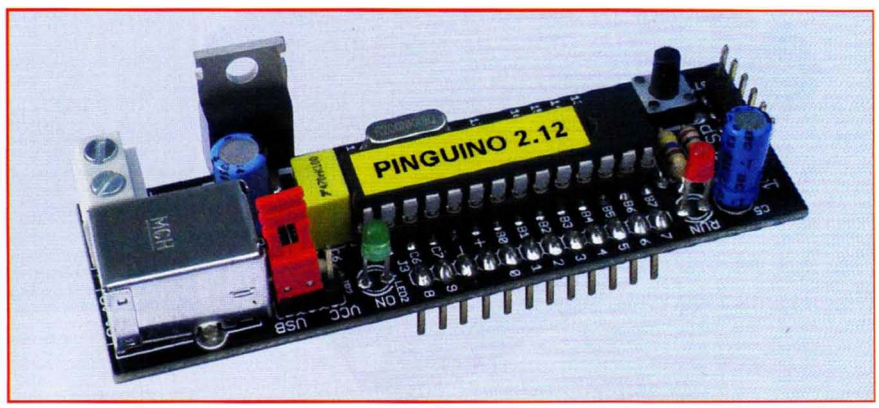
1.3.5 ARM7

Enfin, la volonté des développeurs est désormais de porter Pinguino sur d'autres cibles. La première d'entre elles devrait être une carte à base de processeurs ARM7 de STM32 autorisant ainsi l'utilisation de programmes écrits pour Pinguino32 Microchip sur une autre plateforme.

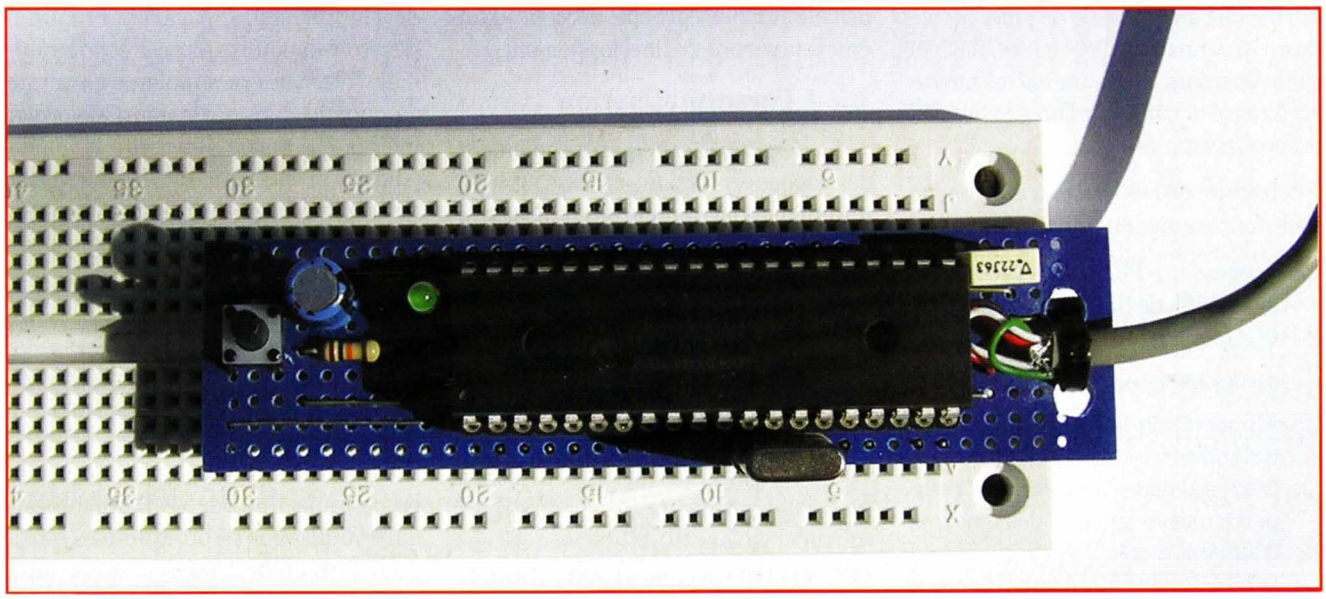
2 Le hardware

2.1 Cartes 8 bits

Ces cartes à base de PIC 18F2550 et 18F4550 sont à l'origine du projet. Avec seulement quelques composants externes, elles permettent d'exploiter toute la puissance d'un processeur 12MIPS avec USB intégré. Il existe de nombreuses déclinaisons de cette carte, sur plaque d'essai, sur circuit imprimé, « fait-maison » ou de fabrication industrielle. Ce type de carte de développement peut être câblée en une heure sur une plaque d'essai et permet de découvrir la facilité avec laquelle on peut programmer une application sans connaissances préalables.



Pinguino Modèle ARDE (Association de Robotique Espagnole)



Pinguino 4550 sur plaque d'essai

Avec le succès, de nombreuses versions commerciales ont vu le jour parmi lesquelles on peut citer :

- les cartes EMPEROR de EDTP Electronics aux États-Unis,
- les cartes KIDULES et Picstar de DIDEL S.A en Suisse,
- les cartes PICuno (au format Arduino) de Cytron Technologies en Malaisie.

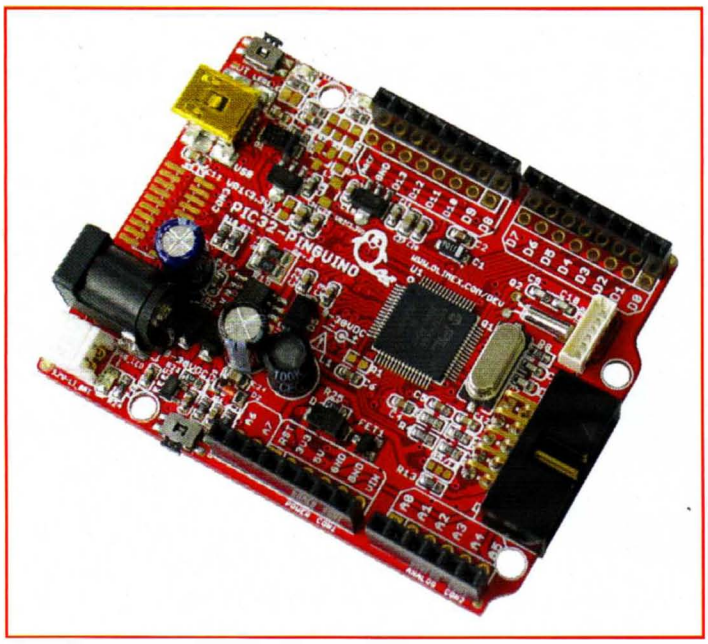
2.2 Cartes 32 bits

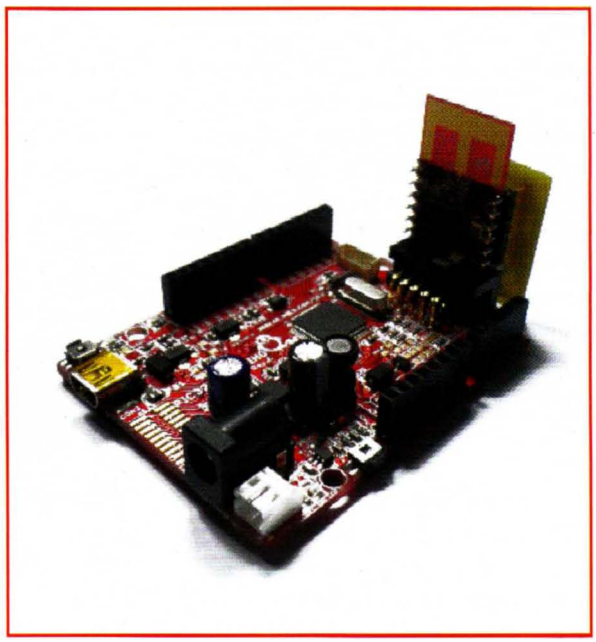
Ces cartes sont à la base du développement 32 bits de Pinguino. Bien qu'il existe quelques versions « maison », le support du développement est réalisé sur les cartes PIC32-PINGUINO de OLIMEX.

On retrouve le design Arduino assurant la compatibilité avec les *shields* (extensions) déjà existants. La présence du connecteur UEXT (sur la droite en bas) permet l'adjonction d'extensions supplémentaires sans prendre de place sur les connecteurs d'entrée/sortie. Le module ZIGBEE par exemple, peut être connecté sur ce module d'extension.

2.3 Open Hardware

Conformément à son engagement « open hardware », les dossiers de construction des différentes cartes





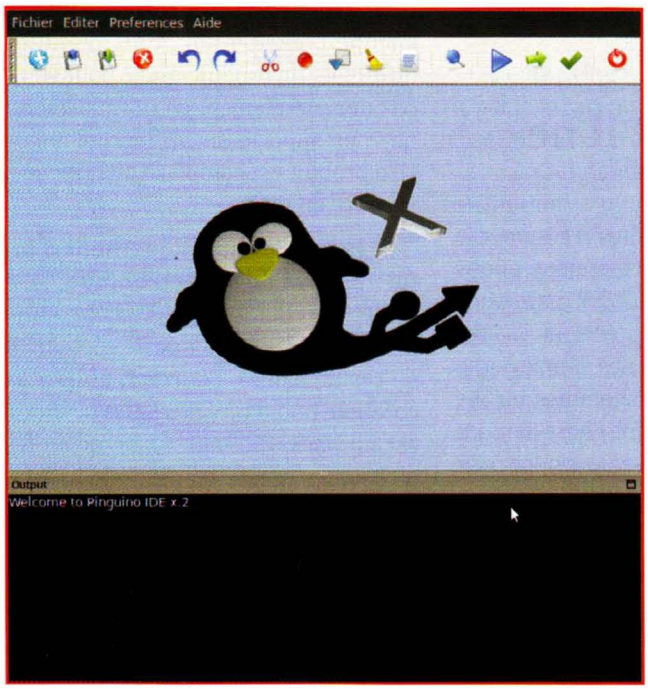
Pinguino sont disponibles en téléchargement sur le site. L'esprit « garage » auquel les développeurs sont particulièrement attachés est préservé et il est donc possible de fabriquer sa propre carte Pinguino, qu'elle soit à base d'un microcontrôleur 8 bits ou 32 bits, y compris PIC32-PINGUINO. Le coût de fabrication à l'unité de cette dernière et la complexité du montage (composants de surface) la réservent cependant à quelques initiés.

3 L'environnement de développement

L'IDE de Pinguino a été développé en Python avec une interface graphique wxPython. On retrouve un design proche de l'environnement Arduino avec la possibilité de choisir différents thèmes (arduino, regino, mini-regino et d'autres téléchargeables sur le site).

L'IDE propose la coloration syntaxique, l'auto-complétion et un système de renvoi vers les pages de documentation des différentes fonctions sur le wiki du site.

Côté compilation, le développement sur processeur 8 bits est assuré par le compilateur SDCC et la suite GPUTILS. Pour le 32 bits, une version compilée à partir des sources de GCC assure la compilation MIPS. La librairie C est basée sur Newlib.



Pinguino IDE avec thème « mini-regino »

Les programmes sont téléchargés sur les cartes Pinguino via l'IDE et les bootloaders USB installés sur les microcontrôleurs.

Enfin, l'IDE est compatible GNU/Linux, Windows et Mac OS-X (la partie 32 bits de l'environnement est actuellement en cours de développement pour ce dernier).

Il est primordial de noter que l'ensemble de l'environnement est open source. Les compilateurs utilisés n'ont aucun lien avec les versions *Lite* du compilateur Microchip. L'ensemble des librairies de bas-niveau ont été réécrites par les développeurs du projet.

4 La diffusion de Pinguino

Utilisé dans un premier temps dans un but pédagogique, les spécificités de cet environnement de développement ont un peu dépassé les limites du cadre fixé. La présence d'un module USB natif sur tous les processeurs utilisés, associée à la simplicité de fabrication de la version 8 bits, ont permis de donner à ce projet un rayonnement plus large que celui initialement prévu. Une large communauté a vu le jour en Amérique du sud (Vénézuela, Colombie, Brésil) permettant non seulement un développement plus rapide, mais également la traduction de l'environnement en plusieurs langues (via le module Locale de python).

La version 32 bits étend encore la popularité du projet. En France, l'université de CAEN utilise Pinguino pour l'enseignement de l'électronique digitale à l'IUT de mesures physiques. Aux États-Unis, Pinguino32 fait son entrée cette année à l'école d'Arts de Chicago pour devenir une alternative à Arduino. Les derniers développements sur cette version sont porteurs de nombreux projets. L'interfaçage avec Android par exemple ne nécessite que 4 instructions pour obtenir une liaison de communication avec un téléphone Android via ADB. En Espagne, l'association ARDE et Campus Party ont intégré Pinguino dans un cycle de présentations et d'ateliers pour promouvoir son utilisation dans la robotique.

5 Sur le Web

- Le site Pinguino : www.pinguino.cc

  Ce site regroupe le blog, le wiki, le forum, les outils en téléchargement, la boutique, ainsi que toutes les informations sur les dernières évolutions du projet.

- Le Pinguino Google Group : http://groups.google.com/group/pinguinocard

  Espace plus spécialisé dans le développement et le côté technique du projet.

- http://code.google.com/p/pinguino32/

  Les sources de l'environnement de développement. ■

HACKERS, HACKS ET HACKING : MISE AU POINT

Vous tenez entre vos mains le quatrième numéro d'Open Silicum, marquant bientôt une première année de publication. Après ce premier tour du calendrier, un doute nous a subitement effleuré : serait-il possible que la définition historiquement admise de certains termes que nous utilisons puisse être mal interprétée ? Dans le doute, prenons ici le temps de préciser clairement ces termes et ainsi d'éviter des dérives déjà constatées dans les médias grand public et les productions cinématographiques en tous genres.

Pour peu que l'on mette de côté une irascibilité pourtant légitime, la manière dont le sens des mots change à mesure qu'ils gagnent en popularité est presque amusante. Je dis bien « presque »... « geek », « nerd », « développeur », « gamer » sont autant d'exemples. En effet, aujourd'hui on est un « geek » quand on relève ses mails depuis son iPhone ou qu'on regarde de temps en temps VDM ou BashFR sur le quai d'une gare (trô lol (sic)). Hier, cela décrivait des personnes qui investissaient tout leur temps dans leur passion pour l'informatique, au détriment de leur vie sociale. Initialement péjoratif, « geek » est depuis devenu « tendance » et est majoritairement utilisé pour (se) prouver qu'on est bien en phase avec son époque, empreinte de dépendance à la technologie.

Mais le terme dont le détournement affiche le plus de contraste avec sa définition historique est sans conteste « hacker ». C'est également l'un des premiers termes issus du monde des technologies à avoir fait surface dans les médias. Ainsi, depuis les années 90, on a commencé à entendre ce mot, généralement associé à des délits ou des actes pénalement répréhensibles. Depuis lors, l'usage répété du terme et de ses dérivés a installé un sens erroné dans l'esprit du public, qui n'y voit maintenant qu'un aspect négatif et destructeur. L'abus de langage tend à devenir la seule définition généralement utilisée en dépit de l'étymologie et certains préfèrent ainsi utiliser « acteurs du hack » plutôt que hackers, afin d'éviter la confusion.

1 Ce dont il ne s'agit pas

Plutôt que de s'échiner à résumer le sens et l'esprit du terme, nous allons détailler ce qu'il ne décrit pas. Mais avant toutes choses, précisons l'usage des mots en une sentence simple digne d'ignobles petits bonshommes bleus : un hacker hacke quelque chose dans un acte de hacking et utilise/développe pour cela un ou plusieurs hack(s). Passons maintenant à la « déconstruction » de la définition erronée.

Il ne s'agit pas d'utiliser les objets de la manière initialement prévue. Utiliser une machine à café pour faire du café n'est pas un hack. Pas d'avantage que d'utiliser un media center pour simplement lire des films ou encore de suivre la notice d'utilisation d'un quelconque gadget électronique. Tous les objets qui nous entourent ont une ou plusieurs fonction(s) ou utilité(s) qui ont motivé et guidé leur création. En faisant usage de ces objets en suivant les prescriptions de leurs créateurs, vous ne faites preuve d'aucune créativité, vous n'apportez aucune valeur ajoutée, aucune contribution. Vous ne faites qu'utiliser un produit étudié pour vous, ce qui généralement (et normalement) ne vous apporte aucun mérite, même si le produit est complexe ou difficile à prendre en main.

Il ne s'agit pas uniquement d'informatique ou d'électronique. C'est une erreur courante que de penser qu'en parlant de hacks, on parle forcément de ces domaines. Pourtant, utiliser sa cafetière pour cuire des pâtes ou du riz est un hack, tout aussi bien qu'un fer à repasser et une feuille d'aluminium pour dorer des croque-monsieurs, ou que d'utiliser un bas en guise de courroie d'alternateur. Le hacking n'est pas une pratique récente, c'est ainsi que se font les découvertes. C'est la capacité des hackers à imaginer et concrétiser des idées, indifféremment de ce qui est conventionnellement supposé possible, qui constitue l'art du hacking. Il s'agit d'imagination et d'adaptation, mais

crédit photo Senjaykattimani

Ne disposant pas des ressources nécessaires, certains habitants d'Inde fabriquent leur propre véhicule motorisé avec des pièces de récupération, les Jugaads.

aussi et surtout d'une certaine forme d'abstraction vis-à-vis des conventions et d'une vision plus pragmatique et critique d'une situation. Des sentences comme « Ils ne savaient pas que c'était impossible, alors ils l'ont fait », « C'est impossible seulement si l'on croit que ça l'est » ou encore « Pourquoi ? Parce que c'est possible », donnent quelques pistes sur cette manière exceptionnelle de voir les choses.

Il ne s'agit pas de reproduire une manipulation. Un hack est, par définition, une innovation et l'implémentation de quelque chose d'inédit. Installer CyanogenMod 7 sur son Samsung Galaxy S, par exemple, n'est en aucun cas un hack, pas plus que d'utiliser une astuce quelconque pour obtenir un accès root sur un mobile Android. Ce qui est initialement un hack, s'il est reproduit, documenté ou testé/confirmé par d'autres n'en est plus un. Ainsi, pour rester dans l'exemple du shell root sur mobile, l'implémentation d'un code exploitant une faille ou une erreur peut être initialement un hack et le fait d'un hacker. Mais la personne qui suivra les indications et utilisera le code ainsi produit, ne fera finalement que suivre une série d'instructions pour obtenir un résultat attendu (ou espéré). Il utilisera donc un « produit » en suivant une documentation, ce qui va radicalement à l'opposé de la notion de hacking.

Le hacking n'est pas du piratage. C'est là le raccourci le plus souvent utilisé. Même si on en comprend la raison, la manœuvre est bien maladroite. Par définition, détourner de son usage primaire un matériel ou un système est du hacking. Ainsi, si l'innovation et le talent d'improvisation sont bel et bien mis en œuvre, l'intrusion dans un système et son éventuel détournement n'est pas le hack, mais une conséquence possible du hack (un effet de bord). Paradoxalement, les personnes disposant d'aptitudes propres aux hackers ne sont pas celles qui sont les plus nuisibles, pour la simple et bonne raison que leurs motivations sont bien différentes de celles d'autres individus, réutilisant des hacks connus. Le hacker, le vrai, agit pour l'art et pour un accomplissement personnel... Pour le défi. Ce qui fait généralement les gros titres des médias est, le plus souvent, l'œuvre de personnes, certes douées, mais ne partageant pas cette volonté de dépassement de soi et ce besoin de prouver que les limites ne sont souvent que le fruit de conventions. Le hacking est synonyme de création et non de destruction. Ainsi, la découverte et la divulgation responsable de failles de sécurité permet l'amélioration d'un système, si tant est que les responsables du projet ou du système concerné sachent être attentifs (bon exemple : Google. Mauvais exemple : Sony). Associer piratage et hacking est aussi stupide que de considérer toutes les personnes qui ont la main verte comme des cultivateurs de plantes aux propriétés psychotropes. Le hack n'implique pas des actes illégaux et les actes illégaux ne sont souvent pas des hacks mais des exploitations de failles connues.

Ce n'est pas tant le résultat qui compte, que la façon d'y arriver. Le hack est généralement valorisé en fonction de son élégance. Il y a ainsi moins de mérite à obtenir un résultat de manière brutale qu'à le faire de manière subtile. Si nous reprenons le domaine de la téléphonie sous Android, reflasher son mobile avec un système personnalisé proposant un accès root est bien moins élégant que de trouver et exploiter le système existant pour obtenir un résultat identique. La qualité du hack est proportionnelle à la difficulté ou, plus exactement, à l'ingéniosité à mettre en œuvre.

Le hack n'est pas rentable. Ce n'est pas son objectif, qui se résume, en quelque sorte, à la passion du métier bien fait. Utiliser une *appliance* quelconque et détourner son usage peut paraître tantôt plus économique que l'acquisition d'une autre appliance, plus chère mais plus adaptée. Prenez l'exemple simple du hack du badge DX présenté dans ce numéro. Au premier abord, on peut conclure que c'est une façon économique de disposer d'un affichage annexe pour un système embarqué (je parle ici du hack en lui-même et non de l'utilisation du code résultant). Cependant, il faut prendre deux choses en considération.

crédit photo High Contrast - CC BY

La firme Tata Motors produit la Tata Nano, la voiture la moins chère au monde en mettant en œuvre le concept de gandhian engineering : obtenir plus à partir de moins pour le plus grand nombre.

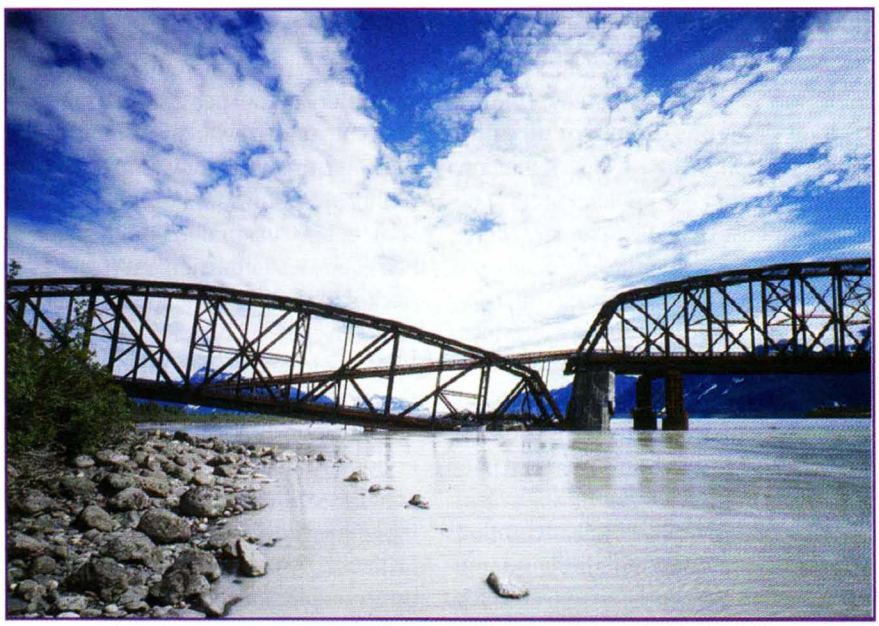
La première est d'ordre commercial et industriel. Personne ne vous prendra au sérieux si vous présentez cela dans un environnement de production. Le hack s'intègre généralement mal dans une démarche de production. Et, si tant est que cela devienne faisable, la généralisation et la sérialisation du processus supprime la nature initiale du hack. Intégrer un produit grand public dans un autre n'est plus un hack, mais l'utilisation des connaissances qui en découlent.

Le second élément à prendre en considération est plus important encore : c'est le temps. Le temps passé à hacker un tel périphérique inclut l'étude, l'analyse, les essais, l'implémentation finale et éventuellement la documentation décrivant le hack. En prenant tout cela en considération, pensez-vous que le hack soit effectivement plus économique que l'acquisition d'un système d'affichage dédié ? Un autre exemple est celui des NAS. Ceux-ci sont potentiellement tous des serveurs miniatures. Sans base documentaire aucune, prendre le temps de construire un système complet utilisant pleinement les fonctionnalités matérielles disponibles est-il plus rentable que d'assembler un mini PC ? Le plus souvent la réponse est « non ». Un hack ne prend pas en compte le temps de réalisation car, et c'est le point précédent, le voyage compte autant, sinon plus, que la destination.

Bien entendu, des petits hacks peuvent intervenir dans la vie professionnelle, le plus souvent en R&D (recherche et développement). Adapter rapidement une sonde JTAG d'un kit d'évaluation à un autre en jouant du fer à souder et via quelques lignes de C est un hack ponctuel plus efficace que d'attendre un mois l'arrivée d'une nouvelle sonde adaptée. Ceci reste cependant une exception et ne concerne pas la production d'un produit. Au contraire, lorsqu'on découvre des choses comme un ou deux câble(s) soudé(s) sur un circuit en ouvrant un périphérique, on y voit davantage une horrible rustine corrigeant une incompétence quelconque, qu'un hack élégant et respectable.

2 Esprit, motivation et exploration

Nous l'avons dit précédemment, le hack n'est pas une pratique récente. Ce n'est finalement qu'un mot utilisé pour définir une façon d'envisager un problème et d'y trouver une solution.



crédit photo Historic American Buildings Survey

Un hack n'est pas spécifique à l'informatique, il peut s'appliquer à de nombreuses choses. Ici, le pont de Miles Glacier ayant subi une réparation de fortune suite à un tremblement de terre. En anglais, on parlera alors de « kludge » en lieu et place de « hack ».

crédit photo Pauri Vaar

Régulièrement, les hackers de toutes sortes se réunissent pour partager leur savoir et échanger des connaissances. Ce genre de manifestations couvre bien des domaines et est généralement un lieu de rencontre où se mélangent bidouilleurs, pirates et forces de l'ordre.

On peut ainsi regarder l'histoire avec un œil différent et en particulier les innovations techniques de l'espèce humaine.

Lorsqu'on comprend que le hacking n'est finalement rien d'autre que le fait de dépasser les limites qui semblent imposées et ainsi apporter une solution novatrice à un problème ponctuel ou chronique, on se rend compte que notre histoire est finalement ponctuée de hacks. Il devient même légitime de se demander si le hack n'est pas la seule façon d'innover et de faire progresser nos connaissances.

Aujourd'hui, alors que de plus en plus de produits existent pour régler les petits problèmes de tous les jours, les consommateurs perdent l'habitude de regarder au-delà de ce qu'on leur propose et de ce qu'on leur vend. Ils développent ainsi une dépendance à la technologie bien plus forte et dangereuse que celle des passionnés, qui eux, ont une relation plus interactive avec leur domaine de prédilection.

C'est là une habitude qui s'installe et qui découle directement d'une certaine accoutumance au confort. Nous avons dans nos vies des objets pour chaque usage : un four micro-ondes pour réchauffer les aliments, des plaques vitrocéramiques pour cuire et cuisiner, des machines à café pour nos breuvages énergisants, des réfrigérateurs pour conserver les denrées, etc. Pourquoi diable devrions-nous détourner l'un de ces appareils de son usage initial ? Pourquoi donc tenter de le faire réparer (ou le réparer soi-même) si l'opération est plus coûteuse que l'achat d'un nouveau produit, au même prix, qu'on nous jure plus efficace, mais pas forcément plus robuste ? D'autres utilisateurs, disposant de moins de ressources que nous, ne se posent pas ces questions. Il ne peuvent pas se le permettre...

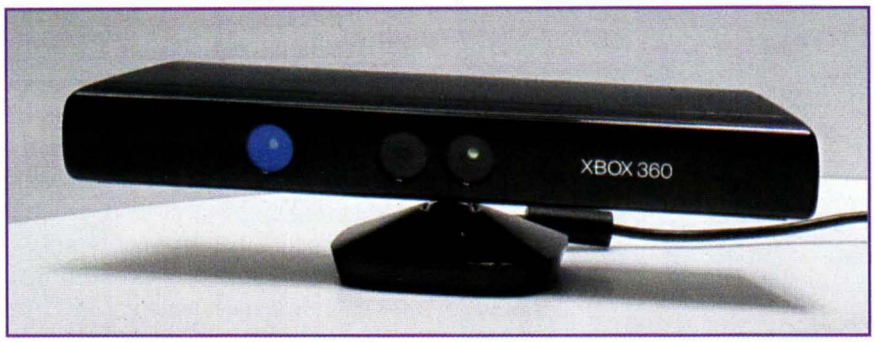
3 Jugaad

Dans les pays industrialisés, le hacking est une activité intellectuelle, pratiquée par une minorité de la population. Ailleurs, cette capacité à trouver des solutions « avec ce qu'on a », n'est pas un sport cérébral ou une forme d'artisanat ludique, mais une affaire de survie.

Le terme « jugaad » nous vient d'Inde, où il désigne plusieurs choses. Les différences linguistiques et culturelles entre nous et l'Inde rendent difficile la simple traduction, à l'instar de celle de « hack » en français (« bidouille » est péjoratif, « bitouille » simplement ignoble, etc.). Le terme peut ainsi désigner principalement deux choses. La première est un véhicule généralement construit intégralement autour d'un moteur diesel initialement utilisé pour les pompes à eau des installations agricoles. Il s'agit de moyens de transport économiques, partagés entre plusieurs voyageurs et qui sont destinés au transport d'une vingtaine de personnes ou plus. Se déplaçant généralement à une vitesse pouvant aller jusqu'à 60km/h, ils sont dépourvus de tous systèmes de sécurité modernes et ne respectent aucune norme en vigueur. Ils sont également réputés pour freiner assez mal, via des dispositifs tout aussi artisanaux que le reste du véhicule.

Devenus très courants, en particulier dans les petits villages d'Inde, ils échappent à toute tentative de réglementation et peuvent être tout aussi bien utilisés pour les transports scolaires que pour celui de marchandises ou de matières premières. Ces véhicules improvisés, fabriqués avec les moyens du bord, sont une réponse à un besoin d'une population disposant de peu de ressources,

Mais jugaad désigne plus que cela et englobe également toutes les réalisations de même nature. Jugaad en tant que désignation d'un moyen de transport n'est pas généralisé à l'ensemble de l'Inde, mais le sens pratique qui y est associé est lui connu mondialement. En finance, par exemple, on parle parfois de jugaad comme la « voie indienne », désignant ainsi la capacité d'innover et de créer ce dont on a besoin avec ce qu'on a sous la main. De plus en plus, cette philosophie trouve sa place pour réduire les coûts de R&D et de production dans les pays semi-industrialisés, qui inspirent ainsi le reste du monde. Juggad, dans ce contexte, est associé à d'autres pratiques comme le *frugal engineering*



crédit photo James Pfaff

Le périphérique Kinect de Microsoft, initialement destiné à la console Xbox 360, a fait l'objet de nombreux hacks. Suite au lancement d'un concours dans des circonstances qui restent étranges (intervention d'un membre de l'équipe de développement du produit), ce périphérique s'est vu doté d'un pilote open source pour Linux en moins d'une semaine grâce au hacker Hector Martin.

(également appelé *gandhian engineering*) visant à réduire un processus de fabrication, ou un produit complexe, en petits éléments simples pour ensuite tout ré-assembler de la façon la plus économique possible. L'idée, telle que mise en œuvre par Tata Motors pour sa Nano, la voiture la moins chère du monde, est d'aller à l'essentiel et le plus simplement possible de manière à créer plus à partir de moins et ce, pour le plus grand nombre.

La principale différence entre le jugaad et le hack réside dans la motivation. Construire un véhicule de toutes pièces avec des restes, combiner les éléments de deux réfrigérateurs différents pour obtenir un appareil fonctionnel ou encore adapter de force une pièce conçue pour un autre modèle mécanique, sont autant d'exemples découlant de la nécessité, du besoin et de l'absence de choix et de ressources.

La fracture entre les pays comme l'Inde et l'Europe, par exemple, est énorme car chez nous, rien ne nous oblige réellement à comprendre comment fonctionnent les objets qui nous entourent et la majorité des consommateurs se contentent de céder à des habitudes insufflées par les lois de la consommation. Les habitants des pays industrialisés, comme le nôtre, sont incités à jeter et remplacer, car c'est l'obsolescence programmée qui maintient, en grande partie, l'équilibre du système.

4 Conclusion (temporaire)

Temporaire c'est le mot, car force est aujourd'hui de constater que des habitudes sont prises par les consommateurs, incités à remplacer plutôt qu'à réparer. La réparation d'un produit revient à comprendre son fonctionnement et donc à étudier la logique mise en œuvre par les concepteurs du produit. C'est la première étape d'un hack. Parfois, la réparation n'est pas prévue par le constructeur, ni souhaitée d'ailleurs. Il faut alors franchir une nouvelle étape du hack consistant à adapter la pièce de rechange à l'appareil réparé. Le plus souvent, avec un baladeur MP3 par exemple, il est possible de trouver la pièce correspondante et ce ne sera finalement pas un hack, même si l'effort est bien là (essayez d'ouvrir un iPod G3, vous verrez). Dans d'autres cas, il faudra un travail plus long. Si vous insistez, en dépit du temps que cela vous en coûtera, et que vous réussissez, votre réparation sera peut-être un hack. Si vous tentez l'opération alors que le matériel fonctionne correctement, pour adapter une batterie de plus forte capacité, ce sera certainement un hack.

Le hack n'est pas une affaire de génie mais d'astuce, d'ingéniosité et de curiosité. Malheureusement, dans bien des domaines, le hack nécessite de plus en plus de connaissances, à mesure que la technologie progresse, bien sûr, mais également à mesure que les produits intègrent de plus en plus de mécanismes visant à limiter les possibilités de réparation et donc de hack. N'avez-vous pas remarqué ? Certains laptops ne permettent plus le remplacement de la batterie qui pourtant est un élément qui se dégrade bien plus vite que le reste de l'appareil. C'est également vrai pour les baladeurs et les téléphones, mais pas seulement. Dans un tout autre domaine, demandez à votre garagiste, par exemple. Révolue semble bel et bien être l'époque où tout un chacun pouvait réparer et entretenir presque intégralement son véhicule à la maison, seul ou aidé d'une connaissance.

Alors qu'on nous parle de développement durable, de limiter les rejets et d'aller vers plus de recyclage, on nous empêche d'appréhender pleinement le fonctionnement des outils et des technologies qui nous entourent et nous aident. Pourtant, rien ne pourrait correspondre davantage au recyclage que la réparation de nos biens.

Et étrangement, ce sont dans les pays où finissent le plus souvent nos produits partiellement défectueux, que la population, n'ayant pas le choix, met en œuvre un art et un savoir-faire qui peu à peu nous échappe ou ne vit plus que dans de rares groupes de personnes. Ceux-là mêmes qui sont relégués au rang d'originaux ou de bidouilleurs fous sinon de pirates et de nuisibles, du simple fait d'un raccourci linguistique stupide et motivé par la fainéantise de faire une quelconque distinction.

Cette conclusion est donc bel et bien temporaire, car j'aime à croire que nos ressources n'étant finalement pas illimitées, le moment viendra où le savoir acquis par le hacking et l'exploration de ce qui nous entoure nous sera bien utile et où le fait de concevoir des produits plus facilement hackables s'imposera naturellement. D'ici là, nous devrons continuer à défaire ce que d'autres font avant de pouvoir améliorer les technologies qui remplissent nos vies. Si le défi est de plus en plus difficile à relever, le prix n'en sera sans doute que plus grand. A vaincre sans péril, on triomphe sans gloire, paraît-il... ■

Abonnez-vous !

Profitez de nos offres d'abonnement spéciales disponibles au verso !

Téléphonez au 03 67 10 00 20 ou commandez par le Web

Économisez plus de 20%*

* Sur le prix de vente unitaire France Métropolitaine

Numéros de 4 Open Silicium

Les 3 bonnes raisons de vous abonner :

- Ne manquez aucun numéro.
- Recevez Open Silicium Magazine tous les 3 mois chez vous ou dans votre entreprise.
- Économisez 9,00 €/an ! (soit plus de 2 magazines offerts !)

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur www.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

par ABONNEMENT : 27€*

au lieu de 36,00* en kiosque

Économie : 9,00 €*

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE
Pour les tarifs hors France Métropolitaine, consultez notre site : www.ed-diamond.com

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Open Silicium

Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **www.ed-diamond.com/cgv** et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

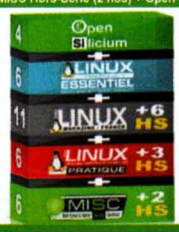
PROFITEZ DE NOS OFFRES D'ABONNEMENT SPÉCIALES POUR LIRE PLUS ET FAIRE DES ÉCONOMIES !

➡ Voici une sélection des offres de couplage avec Open Silicium

offre 13 — Open Silicium Magazine (4 nos)

par ABO : **27€*** — au lieu de 36,00€** en kiosque — Economie : 9,00 €

offre 14 — Linux Pratique Essentiel (6 nos) + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos) + Misc (6 nos) + MISC Hors-Série (2 nos) + Open Silicium (4 nos)



par ABO : **224€*** — au lieu de 304,70€** en kiosque — Economie : 80,70 €

offre 16 — Open Silicium + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos)

par ABO : **108€*** — au lieu de 146,50€** en kiosque — Economie : 38,50 €

offre 17 — Open Silicium + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Misc (6 nos)

par ABO : **141€*** — au lieu de 194,50€** en kiosque — Economie : 53,50 €

Vous pouvez également vous abonner sur : www.ed-diamond.com ou par Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

➡ Voici une sélection de nos autres offres de couplage (Toutes les offres sur : www.ed-diamond.com)

offre 10 — MISC (6 nos) + MISC Hors-Série (2 nos)

par ABO : **44€*** — au lieu de 64,00€** en kiosque — Economie : 20,00 €

offre 3 — GNU/Linux Magazine (11 nos) + Linux Pratique (6 nos)

par ABO : **78€*** — au lieu de 107,20€** en kiosque — Economie : 29,20 €

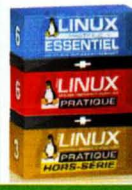
offre 4 — GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos)

par ABO : **83€*** — au lieu de 110,50€** en kiosque — Economie : 27,50 €

offre 5 — + GNU/Linux Magazine (11 nos) + Misc (6 nos)

par ABO : **84€*** — au lieu de 119,50€** en kiosque — Economie : 35,50 €

offre 15 — Linux Pratique Essentiel (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos)



par ABO : **72€*** — au lieu de 94,20€** en kiosque — Economie : 22,20 €

offre 6 — + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos)

par ABO : **110€*** — au lieu de 146,20€** en kiosque — Economie : 36,20 €

offre 7 — + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Misc (6 nos)

par ABO : **116€*** — au lieu de 158,50€** en kiosque — Economie : 42,50 €

offre 8 — + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Misc (6 nos)

par ABO : **143€*** — au lieu de 194,20€** en kiosque — Economie : 51,20 €

➡ Nos Tarifs s'entendent TTC et en euros

	F France Métro	D DOM	T TOM	E1 Europe 1	E2 Europe 2	EUC Etats-Unis Canada	A Afrique	RM Reste du Monde
13 Abonnement Open Silicium	27 €	29 €	31 €	32 €	31 €	33 €	32 €	36 €
3 GLMF + LP	78 €	85 €	96 €	99 €	95 €	101 €	98 €	111 €
4 GLMF + GLMF HS	83 €	89 €	101 €	104 €	100 €	105 €	103 €	116 €
5 GLMF + MISC	84 €	90 €	102 €	105 €	101 €	107 €	104 €	117 €
6 GLMF + GLMF HS + Linux Pratique	110 €	119 €	134 €	138€	133 €	140 €	137 €	154 €
7 GLMF + GLMF HS + MISC	116 €	124 €	140 €	144 €	139 €	146 €	143 €	160 €
8 GLMF + GLMF HS + MISC + LP	143 €	154 €	173 €	178 €	172 €	181 €	177 €	198 €
10 MISC + MISC HS	44 €	47 €	53 €	55 €	52 €	56 €	54 €	60 €
13 Open Silicium Magazine	27 €	29 €	31 €	32 €	31 €	33 €	32 €	36 €
14 GLMF + GLMF HS + MISC + MISC HS + LP + LP HS + LPE + Open Silicium	224 €	241 €	272 €	280 €	268 €	285 €	277 €	313 €
15 LPE + LP + LP HS	72 €	78 €	88 €	91 €	87 €	93 €	90 €	101 €
16 Open Silicium + LM + LMHS	108 €	116 €	130 €	134 €	129 €	136 €	133 €	150 €
17 Open Silicium + LM + LMHS + MISC	141 €	151 €	169 €	174 €	168 €	177 €	173 €	194 €

- Europe 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède
- Europe 2 : Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande
- Zone Reste du Monde : Autre Amérique, Asie, Océanie
- Zone Afrique : Europe de l'Est, Proche et Moyen-Orient

* Toutes les offres d'abonnement : en exemple, les tarifs ci-dessus correspondant à la zone France Métro (F) ** Base tarifs kiosque zone France Métro (F)

Mes choix :

Mon 1er choix	Je sélectionne le N° (1 à 17) de l'offre choisie :	
Mon 2ème choix	Je sélectionne le N° (1 à 17) de l'offre choisie :	
Mon 3ème choix	Je sélectionne le N° (1 à 17) de l'offre choisie :	
	Je sélectionne ma zone géographique (F à RM) :	
	J'indique la somme due : (Total)	€

Exemple : je souhaite m'abonner à l'offre GNU/Linux Magazine + GNU/Linux Magazine Hors-série + MISC (offre 7) et je vis en Belgique (E1), ma référence est donc 7E1 et le montant de l'abonnement est de 144 euros.

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° ⌊_⌊_⌊_⌊_ ⌊_⌊_⌊_⌊_ ⌊_⌊_⌊_⌊_ ⌊_⌊_⌊_⌊_

Expire le : ⌊_⌊_⌊_⌊_

Cryptogramme visuel : ⌊_⌊_⌊_

Date et signature obligatoire

LIRE DIRECTEMENT LE CLAVIER VIA LE SOUS-SYSTÈME D'ENTRÉE DE LINUX

Vous en conviendrez avec moi, l'évolution des systèmes GNU/Linux tend vers l'automatisation à outrance. Alors qu'il y a quelques années l'utilisateur devait connaître son matériel et le détailler dans la configuration système, aujourd'hui, il se contente d'installer une distribution et de connecter les périphériques. C'est une avancée majeure face à un système propriétaire majoritaire qui nécessite toujours l'installation de pilotes externes pour la plupart des périphériques. Malheureusement, dans certains cas, ce qui simplifie la vie des utilisateurs lambda n'est pas souhaitable pour ceux qui visent un contrôle plus fin du système.

Voilà un fait qui semble souvent « normal » pour les nouveaux utilisateurs de GNU/Linux. On installe le système et tout, ou presque, est automatiquement supporté : réseau, carte audio, adaptateur graphique, périphériques USB, etc. Les technologies utilisées dans les distributions modernes, reposant sur l'utilisation de D-Bus et udev, permettent non seulement le chargement automatique de pilotes, mais également la configuration à la volée des périphériques amovibles. Ainsi, connectez un clavier ou une souris USB sur un poste de travail GNU/Linux et ceux-ci seront immédiatement utilisables sans le moindre problème tout comme un graveur USB ou la plupart des périphériques de ce type. Il en va de même pour un changement de contrôleur graphique, l'ajout d'une carte Ethernet, ou encore la connexion d'un adaptateur audio. Ce même utilisateur lambda, certes ravi, aura d'ailleurs tendance à trouver cela parfaitement légitime alors que son précédent système d'exploitation propriétaire nécessitait, quant à lui, l'installation d'une collection de pilotes généralement sur CD-ROM... Mais ceci, tout comme le manque de respect de certains envers les développeurs open source, par le fait de trouver cela légitime et exigible, est un tout autre débat.

Ce qui est bon pour la majorité n'est pas nécessairement bon pour tous. L'utilisateur d'un laptop connectant un pad numérique aura, par exemple, vite fait de constater, quelque soit le système, qu'en activant [Verr num], une partie de son clavier intégré sera remappé en conséquence (selon le modèle de pad, bien entendu). Dans une déclinaison plus spécifique, ce problème peut également apparaître dans le cas où vous souhaiteriez utiliser le pad en question comme console d'accès ou d'authentification couplé à un PC standard. Il n'est pas souhaitable que la saisie de valeurs numériques sur le pad soit considérée comme une entrée clavier destinée à la console, au terminal ou à l'application en cours d'utilisation. Dès lors qu'il est question d'avoir deux entrées différentes et indépendantes, un problème se pose avec la configuration automatique.

La résolution de ce problème tient en deux parties distinctes :

- Il faut, d'une part, empêcher que tous les périphériques ne soient considérés comme des entrées standards utilisables,
- et, d'autre part, il faut être en mesure de lire directement les données envoyées par un périphérique de saisie arbitrairement choisi.

1 Empêchons le système de choisir pour nous

Deux cas de figure doivent être envisagés ici : X et la console. La console pause un problème épineux, car le support des périphériques USB HID est intégré dans le noyau (même sous forme de module). C'est d'ailleurs ce support qui nous permettra, par la suite, d'accéder directement aux données émises par le périphérique de notre choix. Hors de question, donc, de supprimer ce support. Il faut, de plus, ajouter à cela que la majorité des BIOS de PC sont configurés par défaut de manière à émuler un clavier standard avec un périphérique USB. On retrouve généralement cette fonctionnalité dans la configuration du BIOS, sous le nom « Legacy support ». Ceci permet aux bootloaders comme LILO ou GRUB de fournir à l'utilisateur une possibilité de manipuler un menu ou d'interagir avec une interface de configuration à un moment où aucun support USB n'est encore en place.

Linux, au démarrage, désactive cette fonctionnalité en chargeant le support USB HID. La solution la plus simple et la plus radicale consiste donc, à supprimer les terminaux virtuels (lignes **getty** dans **/etc/inittab**) et donc à empêcher toute interaction entre le périphérique et l'ensemble du système. Bien entendu, il faudra également, dans le même fichier, veiller à désactiver la possibilité de provoquer un reboot avec la combinaison de touches [Ctrl]+[Alt]+[Suppr] si celles-ci sont accessibles. Il serait dommage que l'utilisateur, aux prises avec une unique application, puisse redémarrer le système. Ne riez pas, certains automates, à l'époque sous MS/DOS, ont présenté ce défaut, et il ne s'agissait pas d'un assemblage d'amateur...

Une autre solution pourrait consister à remapper le clavier entièrement à l'aide d'un fichier de mapping spécifique qu'on chargerait au démarrage avec **loadkeys**.

Finalement, après avoir longuement cherché, nous n'avons pas trouvé de méthode véritablement « propre » permettant d'exclure un périphérique de saisie spécifique du système sans désactiver les fonctionnalités dont nous avons besoin. Ce support semble trop intégré au système et c'est donc à la couche supérieure qu'il nous faut intervenir. Pour la console, la suppression du **getty** semble une bonne solution.

Côté X et X.org en particulier, la configuration est plus accessible. Par défaut, le serveur X, lors de son démarrage, va vérifier les périphériques disponibles et utiliser ceux qui lui conviennent. Il fait de même aujourd'hui pour l'adaptateur graphique. Voilà pourquoi, en l'absence d'un éventuel **/etc/X11/xorg.conf**, l'utilisateur ne se rendra compte de rien. Sur la distribution dérivée de Debian, Ubuntu GNU/Linux 10.10, par exemple, le fichier de configuration n'existe tout simplement pas. Pourtant dans le journal **/var/log/Xorg.0.log** on retrouve tous nos éléments :

```
(II) LoadModule: "evdev"
(II) Loading /usr/lib/xorg/modules/input/evdev_drv.so
(II) Module evdev: vendor="X.Org Foundation"
    compiled for 1.9.0, module version = 2.3.2
    Module class: X.Org XInput Driver
    ABI class: X.Org XInput driver, version 11.0
[...]
(II) config/udev: Adding input device Power Button (/dev/input/event0)
(**) Power Button: Applying InputClass "evdev keyboard catchall"
(**) Power Button: always reports core events
(**) Power Button: Device: "/dev/input/event0"
(II) Power Button: Found keys
(II) Power Button: Configuring as keyboard
(II) config/udev: Adding input device LITEON
     Technology USB Multimedia Keyboard (/dev/input/event2)
(**) LITEON Technology USB Multimedia Keyboard:
     Applying InputClass "evdev keyboard catchall"
(**) LITEON Technology USB Multimedia Keyboard:
       always reports core events
(**) LITEON Technology USB Multimedia Keyboard:
       Device: "/dev/input/event2"
(II) LITEON Technology USB Multimedia Keyboard:
       Found keys
(II) LITEON Technology USB Multimedia Keyboard:
      Configuring as keyboard
(II) XINPUT: Adding extended input device
     "LITEON Technology USB Multimedia Keyboard"
     (type: KEYBOARD)
(**) Option "xkb_rules" "evdev"
(**) Option "xkb_model" "evdev"
(**) Option "xkb_layout" "fr"
(**) Option "xkb_variant" "oss"
(II) config/udev: Adding input device KEYBOARD (/dev/input/event4)
(**) KEYBOARD: Applying InputClass "evdev keyboard catchall"
(**) KEYBOARD: always reports core events
(**) KEYBOARD: Device: "/dev/input/event4"
(II) KEYBOARD: Found keys
(II) KEYBOARD: Configuring as keyboard
(II) XINPUT: Adding extended input device "KEYBOARD" (type: KEYBOARD)
[...]
```

La tendance s'est donc clairement inversée. Alors qu'il fallait précédemment configurer un matériel pour l'utiliser, vous devez maintenant demander au système de ne pas le configurer afin de l'ignorer. Il vous faudra donc créer un fichier **xorg.conf** contenant votre configuration et en particulier les paramètres suivants :

```
Section "ServerFlags"
        Option "AutoAddDevices" "False"
        Option "AllowEmptyInput" "True"
EndSection
```

La première directive, **AutoAddDevices**, empêche le serveur d'ajouter lui-même les périphériques. La seconde, **AllowEmptyInput**, permet au serveur d'être lancé même si les périphériques d'entrée (comprendre la souris et le clavier), ne sont pas présents. Ceci est indispensable pour les tests, le temps d'obtenir un fichier de configuration fonctionnel. Prévoyez également une éventuelle seconde machine vous permettant de vous connecter, via SSH, sur celle que vous configurez afin de pouvoir stopper le serveur X ou le gestionnaire de connexion (GDM) et reprendre la main en cas de problème.

Remarque **Bien configurer son X**

Désactiver l'ajout automatique de périphériques dans X.org est également un bon moyen de gérer pleinement et précisément sa configuration. Bien qu'actuellement la tendance soit à l'absence complète de fichier **/etc/X11/xorg.conf**, confectionner son fichier de configuration est le seul moyen d'avoir la maîtrise complète de l'environnement et de gérer ses périphériques. Un exemple simple est justement la configuration des périphériques d'entrée comme les claviers. Imaginez simplement la situation suivante : vous disposez de deux claviers, le premier est connecté sur le port PS/2 de la machine car, bien qu'il supporte les deux modes de connexion (USB et PS/2), ceci vous permet d'économiser un port USB au bénéfice du connecteur mini-DIN trop souvent délaissé (le pauvre).

Ce clavier utilise une organisation, un *layout* ou *mapping*, AZERTY standard pour la France. Le second clavier, quant à lui, est un QWERTY US, mieux adapté à la programmation car les symboles utiles sont plus facilement accessibles au détriment des caractères accentués. Le but d'une telle configuration est de permettre une migration en douceur d'un layout à un autre, ou de faciliter l'utilisation de la machine par deux personnes aux motivations différentes (programmeur vs utilisateur).

Dans ce cas, si l'on ne souhaite pas reposer sur une infrastructure de configuration lourde comme celle de GNOME, la solution consiste à tout simplement désactiver la gestion automatique et spécifier les caractéristiques des claviers manuellement. Un exemple de configuration peut être ceci :

```
Section "ServerLayout"
        Identifier     "X.org Configured"
        Screen          0  "Screen0" 0 0
        InputDevice    "Mouse0" "CorePointer"
        InputDevice    "Keyboard0" "CoreKeyboard"
        InputDevice     "Keyboard1" "SendCoreEvents"
EndSection

[...]

Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "evdev"
        Option      "Device" "/dev/input/event0"
        Option      "CoreKeyboard"
        Option      "XkbRules"      "xorg"
        Option      "XkbModel"      "pc105"
        Option      "XkbLayout"     "fr"
        Option      "XkbVariant"    "latin9"
EndSection

Section "InputDevice"
        Identifier  "Keyboard1"
        Driver      "evdev"
        Option      "Device"    "/dev/input/by-id/usb-046a_0023-event-kbd"
        Option      "XkbRules"      "xorg"
        Option      "XkbModel"      "pc105"
        Option      "XkbLayout"     "us"
EndSection
```

Les périphériques sont identifiés par leur entrée dans **/dev/input/** et comme vous le voyez, le premier est configuré en **"XkbLayout" "fr"** alors que le second utilise **"us"**. Dès le prochain démarrage de X, nous avons ainsi deux claviers utilisables de manière indépendante. La connexion d'un nouveau périphérique de saisie, comme un pavé numérique pour portable, ne sera pas pris en charge par X et devra faire l'objet, si nécessaire, d'une modification du fichier **xorg.conf**. Notez qu'il ne s'agit pas de la seule manière de procéder, il est également possible de reposer sur les **InputClass** en précisant, par exemple, le chemin dans **/dev/** ou encore le nom du périphérique, respectivement avec les options **MatchDevicePath** et **MatchProduct**.

Puisque plus rien ne fonctionne automatiquement, vous voilà obligé de spécifier votre configuration manuellement. Comme au bon vieux temps. Vous pouvez tenter d'utiliser le système d'auto-configuration depuis la console avec **X -configure**. Cette commande vous générera un fichier de configuration « modèle » prenant en compte votre matériel. Il vous faudra ensuite éditer ce fichier et apporter des modifications pour prendre en charge vos périphériques. En ce qui concerne les périphériques d'entrée, ce que vous devriez configurer ressemble par exemple à ceci :

```
Section "InputDevice"
Identifier "Generic Keyboard"
Driver     "evdev"
Option     "Device" "/dev/input/by-path/platform-i8042-serio-0-event-kbd"
Option     "CoreKeyboard"
Option     "XkbRules"      "xorg"
Option     "XkbModel"      "pc105"
Option     "XkbLayout"     "fr"
Option     "XkbVariant"    "latin9"
EndSection

Section "InputDevice"
Identifier "Configured Mouse"
Driver     "evdev"
Option     "CorePointer"
Option     "Device" "/dev/input/by-path/platform-i8042-serio-1-event-mouse"
Option     "Protocol"           "ExplorerPS/2"
Option     "Emulate3Buttons"    "on"
Option     "Emulate3TimeOut"    "50"
Option     "EmulateWheel"       "on"
Option     "EmulateWheelButton" "2"
Option     "YAxisMapping"       "4 5"
Option     "XAxisMapping"       "6 7"
Option     "ZAxisMapping"       "4 5"
EndSection
```

Plusieurs points sont remarquables et nous entraînent à la découverte du support des périphériques de saisie sous GNU/Linux. Nous avons tout d'abord le pilote X utilisé : **evdev**. Il s'agit du support pour les périphériques « événementiels » rapportés par le noyau, evdev signifiant *EVent DEVice*. Le sous-système de gestion des entrées (*input subsystem*) du noyau fournit une couche d'abstraction pour les périphériques comme les claviers, les souris ou les joysticks, mettant ainsi à la disposition des programmes de l'espace utilisateur une interface unifiée de gestion : c'est evdev. Dans l'extrait de log donné précédemment, vous remarquerez qu'une partie des évènements provenant de l'ACPI sont également « remontés » sous cette forme. Voilà pourquoi le bouton de mise sous tension est, en quelque sorte, un clavier à une touche.

Lorsqu'on utilise le support evdev de X, on désigne le périphérique par son point d'entrée dans **/dev**. C'est udev qui créera les liens symboliques nous évitant de devoir retrouver seul les correspondances entre **/dev/input/event*** et les périphériques :

```
% ls -l /dev/input/by-path
platform-i8042-serio-0-event-kbd -> ../event0
platform-i8042-serio-1-event-mouse -> ../event6
platform-i8042-serio-1-mouse -> ../mouse0
platform-pcspkr-event-spkr -> ../event4
```

Vous pourrez retrouver des informations concernant ces périphériques via le pseudo système de fichiers **/sys** :

```
% cat /sys/class/input/event0/device/name
AT Translated Set 2 keyboard

% cat /sys/class/input/event6/device/name
PS/2 Generic Mouse
```

Une entrée dans **/proc** vous permettra d'en faire autant : **/proc/bus/input/devices**. A ce stade et une fois votre configuration pleinement complétée et fonctionnelle, vous disposerez d'une architecture dont vous êtes, à nouveau, le seul maître. C'est certes une étape délicate et le sevrage peut être pénible selon le matériel utilisé. Cependant, à terme, vous devriez pouvoir obtenir exactement le même résultat que si X avait tout fait à votre place. La seule différence étant que la connexion d'un nouveau périphérique ne sera pas prise en charge par X. Ainsi, entrer une série de valeurs via un nouveau clavier ainsi connecté ne devrait pas provoquer d'insertion dans l'application que vous êtes en train d'utiliser. C'est exactement ce dont nous avons besoin.

2 Lire /dev/input

Ce nouveau périphérique pourra être lu depuis un utilitaire que nous allons concocter. Mais encore faut-il avoir le droit d'y accéder, ce qui n'est généralement pas le cas. Voici une règle **udev** permettant de régler le problème des permissions sur les entrées dans **/dev/input/event*** (à placer, par exemple, dans **/etc/udev/rules.d/zinput.rules**) :

```
KERNEL=="event*", NAME="input/%k", MODE="0666"
```

Il serait possible de régler plus finement ces permissions en ajoutant par exemple un groupe spécifique comme propriétaire du pseudo-fichier, mais ceci sera suffisant pour les besoins de la démonstration et pour une première phase de tests sur une machine de développement.

Remarque **Pourquoi limiter les permissions ?**

Il est maintenant temps de préciser clairement les choses. Les évènements liés aux périphérique de saisie sont accessibles via les entrées evdev. Comprenez bien qu'il s'agit de TOUS les évènements ! Le sous-système de gestion n'a que faire que la saisie effectuée soit une phrase destinée à un éditeur de texte, une adresse mail dans un formulaire HTML ou un mot de passe pour une connexion FTP ou SSH. En récupérant les évènements sur une entrée dans **/dev/input/**, vous récupérez les données envoyées par le périphérique. avant même que cette saisie soit dissimulée ou ne fasse l'objet d'un écho à l'écran.

Pour être plus clair encore, un utilisateur qui peut lire une entrée **/dev/input/** est à même d'espionner tout ce qui est tapé sur un clavier. Les informations qui vont suivre permettent d'écrire un programme capable de décoder ces informations et donc d'écrire un *keylogger* espionnant les saisies de l'utilisateur. Ce n'est, bien entendu, pas l'objet de cet article, mais les différences, dans les faits, sont minces. Et si vous vous demandez si les sécurités proposées par certains sites bancaires, consistant à afficher un pad numérique cliquable agencé aléatoirement, sont sûres, dites-vous que les mouvements du pointeur de souris fournissent quelques informations intéressantes et qu'en quantité suffisante, il serait possible, à un moment ou un autre, de recomposer le code ou mot de passe ainsi entré.

Si votre machine est susceptible d'être utilisée par plus d'un utilisateur, veillez à limiter au maximum les permissions sur les pseudo-fichiers dans **/dev/input/** et vérifiez régulièrement quels processus ont ouvert ces fichiers (avec **sudo lsof | grep "/input/event"** par exemple).

Pour obtenir des informations plus précises afin d'écrire votre ou vos règle(s) udev, vous pouvez utiliser la commande **udevadm** (remplaçant l'ancienne commande **udevinfo**) :

```
% sudo udevadm info -q all -n /dev/input/event10
P: /devices/pci0000:00/0000:00:1d.1/usb6/6-2/6-2:1.0/input/input12/event10
N: input/event10
S: input/by-id/usb-05d5_KEYBOARD-event-kbd
S: input/by-path/pci-0000:00:1d.1-usb-0:2:1.0-event-kbd
E: UDEV_LOG=3
E: DEVPATH=/devices/pci0000:00/0000:00:1d.1/usb6/6-2/6-2:1.0/input/input12/
event10
E: MAJOR=13
E: MINOR=74
E: DEVNAME=/dev/input/event10
E: SUBSYSTEM=input
E: ID_INPUT=1
E: ID_INPUT_KEY=1
E: ID_INPUT_KEYBOARD=1
E: ID_VENDOR=05d5
E: ID_VENDOR_ENC=05d5
E: ID_VENDOR_ID=05d5
E: ID_MODEL=KEYBOARD
E: ID_MODEL_ENC=KEYBOARD
E: ID_MODEL_ID=0689
E: ID_REVISION=0101
E: ID_SERIAL=05d5_KEYBOARD
E: ID_TYPE=hid
E: ID_BUS=usb
E: ID_USB_INTERFACES=:030101:030102:
E: ID_USB_INTERFACE_NUM=00
E: ID_USB_DRIVER=usbhid
E: ID_PATH=pci-0000:00:1d.1-usb-0:2:1.0
E: XKBMODEL=pc105
E: XKBLAYOUT=fr
E: XKBVARIANT=latin9
E: XKBOPTIONS=lv3:ralt_switch
E: DEVLINKS=/dev/input/by-id/usb-05d5_KEYBOARD-event-kbd /dev/input/by-path/
pci-0000:00:1d.1-usb-0:2:1.0-event-kbd
```

Vous pouvez ensuite écrire une règle plus précise comme (sur une ligne) :

```
SUBSYSTEM=="input",ATTRS{idVendor}=="05d5",
  ATTRS{idProduct}=="0689", MODE="0666", GROUP="dialout"
```

dialout est utilisé ici, car mon nom d'utilisateur fait partie de ce groupe en raison de l'accès fréquent aux ports séries. Sachez cependant qu'il est également possible de spécifier un nom de propriétaire en lieu et place du groupe pour restreindre davantage l'accès. Une fois la modification apportée et après avoir redémarré le support udev, nous pouvons afficher les premières informations en provenance du périphérique :

```
% cat /dev/input/by-id/usb-05d5_KEYBOARD-event-kbd | hexdump
0000000 ee94 4e48 6191 0000 0004 0004 0053 0007
0000010 ee94 4e48 619c 0000 0001 0045 0001 0000
0000020 ee94 4e48 619e 0000 0000 0000 0000 0000
0000030 ee94 4e48 fdcf 0000 0004 0004 0053 0007
0000040 ee94 4e48 fdd6 0000 0001 0045 0000 0000
0000050 ee94 4e48 fdd8 0000 0000 0000 0000 0000
0000060 ee94 4e48 1d0f 0001 0004 0004 005d 0007
0000070 ee94 4e48 1d18 0001 0001 004c 0001 0000
0000080 ee94 4e48 1d1a 0001 0000 0000 0000 0000
0000090 ee94 4e48 3c4f 0001 0004 0004 005d 0007
00000a0 ee94 4e48 3c56 0001 0001 004c 0000 0000
00000b0 ee94 4e48 3c58 0001 0000 0000 0000 0000
00000c0 ee94 4e48 7ad0 0001 0004 0004 0053 0007
00000d0 ee94 4e48 7ad8 0001 0001 0045 0001 0000
00000e0 ee94 4e48 7add 0001 0000 0000 0000 0000
00000f0 ee94 4e48 170f 0002 0004 0004 0053 0007
0000100 ee94 4e48 1716 0002 0001 0045 0000 0000
0000110 ee94 4e48 1718 0002 0000 0000 0000 0000
```

L'opération est simple puisqu'il s'agit d'afficher le contenu du pseudo-fichier dans **/dev/input/by-id/** (qui est un lien symbolique vers l'entrée correspondante dans **/dev/input/event***). Les données étant binaires, elles sont redirigées vers **hexdump** qui les affichera sous forme de valeurs hexadécimales. On reconnaît là clairement, dans la sortie provoquée par la pression sur une touche du périphérique, un motif et donc des données formatées et encodées. Il s'agit ici d'un pavé numérique pour ordinateur portable. En désactivant NumLock (le verrouillage numérique) activé par défaut à la connexion, on constate un changement :

```
% cat /dev/input/by-id/usb-05d5_KEYBOARD-event-kbd | hexdump
0000000 bc41 4e4c f615 0000 0004 0004 005d 0007
0000010 bc41 4e4c f61b 0000 0001 004c 0001 0000
0000020 bc41 4e4c f61c 0000 0000 0000 0000 0000
0000030 bc41 4e4c d0f5 0001 0004 0004 005d 0007
0000040 bc41 4e4c d0fa 0001 0001 004c 0000 0000
0000050 bc41 4e4c d0fb 0001 0000 0000 0000 0000
```

En y regardant de plus près, il s'agit quasiment de deux séries de trois lignes identiques. En réalité, cela correspond à deux événements : l'appui sur une touche et le relâchement.

3 Décodons, que diable !

Le principe de fonctionnement du sous-système de gestion d'entrées, concernant les claviers, est d'identifier et rapporter dans l'espace utilisateur les codes des touches utilisées, ainsi que les évènements en rapport (pression, relâchement, répétition). Comprenez bien, qu'à ce niveau et à celui auquel nous travaillons aujourd'hui, les symboles sérigraphiés sur les touches importent peu. Que nous lisions le **/dev/input/event*** d'un clavier AZERTY-FR ou d'un QWERTY-US revient au même et les mêmes codes et évènements seront renvoyés par le sous-système.

Mais avant de nous attaquer à la lecture des informations et au décodage des données, nous devons nous familiariser avec l'API et plus exactement les IOCTL qui nous seront utiles. Nous n'allons pas utiliser ici de langage de haut niveau mais ce bon vieux C, afin d'augmenter la douceur et le lustre de notre pilosité naturelle. De plus, le fichier **linux/input.h** contient tout ce qu'il nous faut pour confectionner un utilitaire sympathique sans avoir à dépendre de bibliothèques ou d'interpréteurs lourds.

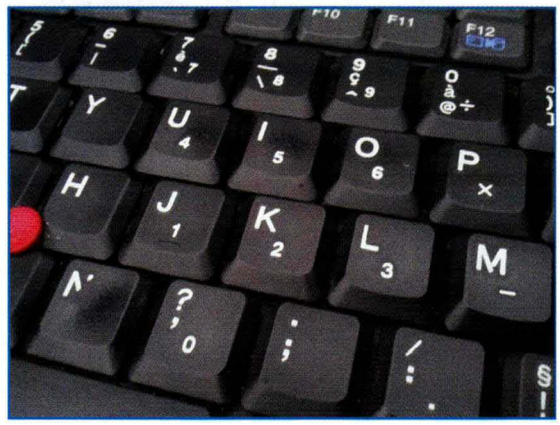
Si nous attaquons simplement la lecture des données, ceci revient à lire l'entrée **/dev/input** de notre choix avec **read()** et faire correspondre ces éléments lus avec une ou plusieurs occurrences d'une structure **input_event** :

Remarque **Clavier pour portable ?**

C'est en jouant avec différents périphériques d'entrée que nous avons constaté qu'il existait une différence importante entre un pavé numérique standard et un pavé numérique pour laptop. Normalement, les leds de notification sur le clavier sont contrôlées par le système, tout comme le caractère numérique du pavé en cas d'activation de NumLock. Une touche enfoncée sur cette partie d'un clavier standard est toujours la même. C'est le système, en fonction de NumLock, qui décide s'il s'agit d'une touche de direction ou d'un nombre qui aura été saisi.

Les pavés numériques pour ordinateurs portables sont différents. Il faut savoir, en effet, que les claviers des portables sont généralement des claviers parfaitement standards disposant d'une fonction NumLock. A l'activation de cette dernière, une partie du clavier se comportera comme un pavé numérique. Si l'on connecte un clavier USB sur la machine, le fait d'activer NumLock va également activer ce « remappage » sur le clavier intégré. Il ne sera plus possible d'utiliser normalement le clavier en question sous peine de saisir des chiffres en lieu et place de quelques lettres.

Un pavé numérique USB pour portable, plutôt que d'activer NumLock, utilise une stratégie différente. Une fois la touche NumLock utilisée, il va faire précéder chaque saisie d'un appui et d'un relâchement de NumLock (touche 69) et faire de même après le relâchement de la touche numérique utilisée. A chaque saisie donc, ce ne sont non pas deux évènements qui sont envoyés, mais 6. Le pavé numérique active pour vous NumLock à chaque touche.



Certains modèles disposent également d'une touche [000] facilitant la saisie de valeurs numériques importantes. Cela fonctionne de la même manière en simulant trois saisies de la touche [0]. Il n'est pas certain que TOUS les pavés numériques USB fonctionnent de la sorte et absolument rien ne vous permet de le savoir au moment de l'achat. Dans le cadre de cet article, la solution consiste simplement à ignorer les évènements liés à la touche 69. Pour une utilisation « normale », en revanche, il est clair qu'un pavé fonctionnant simplement comme un « morceau de clavier USB complet » est beaucoup moins intéressant que ce type de périphériques bien mieux conçus.

```
struct input_event {
        struct timeval time;
        __u16 type;
        __u16 code;
        __s32 value;
};
```

On y trouve ainsi un type d'évènement, un code et une valeur. Cependant, cette structure est utilisée pour tous les types de périphériques et ne nous permet donc pas de déterminer si, effectivement, nous avons affaire à un clavier. On pourrait supposer qu'un chemin vers le pseudo-fichier, récupéré sur la ligne de commande ou dans un fichier de configuration, est un élément déterminant en soit. Procéder à quelques vérifications avant de lire les évènements ne nous coûtera toutefois pas grand-chose si ce n'est quelques IOCTL bien choisis.

Après ouverture classique du pseudo-fichier, nous pouvons, par exemple, récupérer un nom de périphérique :

```
int fd = -1;
char name[256]= "Unknown";

if ((fd = open(argv[1], O_RDONLY)) < 0) {
  perror("evdev open");
  exit(EXIT_FAILURE);
}

if(ioctl(fd, EVIOCGNAME(sizeof(name)), name) < 0) {
  perror("evdev ioctl EVIOCGNAME");
  exit(EXIT_FAILURE);
}
```

Mais nous pouvons également obtenir des informations plus intéressantes et utiles :

```
struct input_id device_info;
u_int8_t evtype_b[(EV_MAX + 7) / 8];

if(ioctl(fd, EVIOCGID, &device_info)) {
  perror("evdev ioctl EVIOCGID");
  exit(EXIT_FAILURE);
}

if (ioctl(fd, EVIOCGBIT(0, EV_MAX), evtype_b) < 0) {
  perror("evdev ioctl EVIOCGBIT type");
  exit(EXIT_FAILURE);
}
```

Les informations récupérées par **EVIOCGID** et placées dans une structure **input_id** nous apportent une poignée d'éléments :

```
struct input_id {
        __u16 bustype;
        __u16 vendor;
        __u16 product;
        __u16 version;
};
```

Il faudra cependant jouer du **switch**/**case** pour décoder tout cela via les déclarations présentes dans **input.h** et en particulier pour **bustype** allant de l'USB (**BUS_USB**) au Bluetooth (**BUS_BLUETOOTH**) en passant par le i8042 (**BUS_I8042**), le contrôleur clavier lié au port PS/2, ou encore des choses plus exotiques inexistantes sur les machines de type PC. Je vous fais grâce ici du morceau de code relativement pénible.

L'IOCTL **EVIOCGBIT** nous apporte plus d'éléments intéressants concernant les caractéristiques du périphérique lui-même et en particulier les évènements qu'il est en mesure de nous rapporter. C'est un masque de bits qui est utilisé et nous profitons de la macro définie dans **input.h**. **EV_MAX** est également une valeur définie dans le fichier d'en-tête et celle-ci est susceptible de changer dans le futur.

evtype_b est déclarée comme suit :

```
u_int8_t evtype_b[(EV_MAX + 7) / 8];
```

Cela peut paraître un peu tordu, mais comme **EV_MAX** nous donne une valeur désignant des bits, il nous faut définir une variable d'une taille exprimée dans la plus petite unité possible : l'octet. Ajouter **7** et diviser par 8 nous permet, ainsi, de toujours prévoir suffisamment d'espace pour notre masque. Nous en profitons pour écrire une petite macro nous simplifiant le test du masque de bits :

```
#define test_bit(bit, array) (array [bit / 8] & (1 << (bit % 8)))
```

Nous n'aurons alors qu'à utiliser **test_bit()** avec en argument le bit à tester et la valeur renvoyée par le périphérique. Nous écrirons donc quelque chose comme :

```
for (i = 0; i < EV_MAX; i++) {
  if (test_bit(i, evtype_b)) {
    printf("  Event type 0x%02x ", i);
    switch (i) {
      case EV_SYN :
        printf(" (Synch Events)\n");
        break;
      case EV_KEY :
        printf(" (Keys or Buttons)\n");
        break;
[...]
```

Nous bouclons sur les valeurs de zéro au numéro maximum possible pour un évènement. Nous utilisons ensuite notre macro pour, tout simplement, vérifier que le périphérique que nous manipulons supporte ce type d'évènements. Si c'est le cas, nous informons l'utilisateur. Notez que nous aurions pu utiliser un tableau de chaînes de caractères pour obtenir quelque chose de similaire. Cependant, je pense que l'utilisation du **switch**/**case**, bien qu'initialement plus lourde est également plus facile à maintenir. Un nouveau type d'évènement reviendrait à ajouter un test et peu nous importeront alors les valeurs associées. En effet, il ne s'agit pas d'une suite continue (dans **input.h**) :

```
#define EV_SYN          0x00
#define EV_KEY          0x01
#define EV_REL          0x02
#define EV_ABS          0x03
#define EV_MSC          0x04
#define EV_SW           0x05
#define EV_LED          0x11
#define EV_SND          0x12
#define EV_REP          0x14
#define EV_FF           0x15
#define EV_PWR          0x16
#define EV_FF_STATUS    0x17
#define EV_MAX          0x1f
```

A ce stade, nous pouvons déjà plus ou moins déterminer si nous avons affaire à un clavier ou non, en nous basant sur les évènements qu'il prend en charge :

```
% ./evdevinfo /dev/input/by-id/usb-05d5_KEYBOARD-event-kbd
The device on /dev/input/by-id/usb-05d5_KEYBOARD-event-kbd
says its name is KEYBOARD
evdev driver version is 1.0.0
vendor:0x05d5, product:0x0689, version:0x0110 is on a Universal Serial Bus
Supported event types:
  Event type 0x00  (Synch Events)
  Event type 0x01  (Keys or Buttons)
  Event type 0x04  (Miscellaneous)
  Event type 0x11  (LEDs)
  Event type 0x14  (Repeat)
```

Or, pour une souris, nous avons :

```
Supported event types:
  Event type 0x00  (Synch Events)
  Event type 0x01  (Keys or Buttons)
  Event type 0x02  (Relative Axes)
  Event type 0x04  (Miscellaneous)
```

Le type concernant des touches ou des boutons (**0x01**), également valable pour une souris, associé à la présence de leds (**0x11**) et des fonctionnalités de répétition (**0x14**) peut être vu comme une combinaison propre aux claviers. Cependant, ce n'est pas suffisant ; pour en avoir le cœur net, nous pouvons procéder à une autre vérification :

```
u_int8_t key_bitmask[(KEY_MAX + 7) / 8];

if((ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(key_bitmask)), key_bitmask)) < 0) {
  fprintf(stderr,"EVIOCGBIT Error : %s (%d)\n", strerror(errno),errno);
} else {
  if (test_bit(KEY_Q, key_bitmask)) {
    printf("Got KEY_Q. This may be a real keyboard\n");
  } else {
    fprintf( stderr, "No KEY_Q in key_bitmask. Not a keyboard\n");
    close(fd);
    exit(EXIT_FAILURE);
  }
}
```

C'est encore une fois l'IOCTL **EVIOCGBIT** que nous utilisons, mais cette fois, en spécifiant le paramètre **EV_KEY** et un masque binaire **key_bitmask** comme nous l'avons fait avec les types d'évènements supportés. Il nous suffit ensuite de choisir arbitrairement une ou plusieurs touche(s) à tester. Nous pouvons ainsi raisonnablement penser, que si le périphérique dispose d'une touche **KEY_Q**, il s'agit bien d'un clavier. Notez deux choses intéressantes à ce propos : la première est un simple rappel et vous précisera clairement que la touche **KEY_Q** est la touche se trouvant à l'emplacement **KEY_Q**, mais que celle-ci peut être sérigraphiée « A » comme c'est le cas pour un clavier AZERTY. La seconde chose concerne la présence physique de la ou des touches testée(s). Ici, notre pavé numérique pour portable ne dispose pas de la touche en question, pourtant elle apparaît dans le masque binaire. Nous avons procédé à plusieurs tests sur différents claviers USB et tous retournent des touches qui n'existent pas physiquement. Certains font de même pour les leds, ce qui semble indiqué que c'est en réalité le composant intelligent dans le périphérique qui répond et ce, indépendamment des parties mécaniques. Quoi qu'il en soit, nous ne cherchons pas une liste des touches disponibles, mais simplement à vérifier qu'il s'agit bien d'un clavier. Il ne s'agit que d'une vérification supplémentaire utile uniquement si vous souhaitez garder un aspect générique à votre programme (dans le cas contraire, VID et PID peuvent suffire).

Une fois toutes les vérifications faites sur les fonctionnalités et les caractéristiques du périphérique, nous pouvons lire l'entrée **/dev/** avec un simple **read()** :

```
while(1) {
  rb=read(fd,ev,sizeof(struct input_event)*64);

  if (rb < (int) sizeof(struct input_event)) {
    perror("evtest: short read");
    exit(EXIT_FAILURE);
  }

  for (i=0; i<(int)(rb / sizeof(struct input_event));i++) {
    if (ev[i].type == EV_KEY && ev[i].value == 1) {
      printf("struct[%u] : ", i);
      printf("type %u code %u value %u\n",
          ev[i].type, ev[i].code, ev[i].value);
    }
  }
}
```

Le principe consiste simplement à lire les données du fichier et « caster » sous la forme d'autant de **struct input_event** que nécessaire avec un maximum de 64. En temps normal, nous sommes censés lire au minimum une occurrence d'une telle structure **input_event**, mais ce nombre est généralement plus important. En effet, nous lisons tous les évènements du périphérique, mais seuls certains nous intéressent. Voilà pourquoi nous limitons ensuite le traitement aux évènements de type **EV_KEY** (en rapport avec les touches) et en particulier ceux ayant une valeur de **1** correspondant à une pression sur une touche (**0** étant le relâchement et **2** la répétition si la touche reste enfoncée). Nous obtenons ainsi la liste des évènements qui nous importent :

```
Waiting for EV_KEY event...
struct[1] : type 1 code 76 value 1
struct[1] : type 1 code 75 value 1
struct[1] : type 1 code 77 value 1
struct[1] : type 1 code 72 value 1
struct[1] : type 1 code 75 value 1
struct[3] : type 1 code 76 value 1
struct[5] : type 1 code 77 value 1
struct[1] : type 1 code 75 value 1
```

Si nous voulons utiliser ces données et en particulier le code de chaque touche dans une application, deux choix s'offrent à nous :

- Jouer avec des **switch/case** et les définitions provenant de **input.h**. Dans le cas d'un pavé numérique utilisé en guise de digicode ou autre, cela reste parfaitement faisable puisqu'il s'agit, tout au plus, d'une quinzaine de touches.
- Créer un tableau de caractères d'une taille **KEY_MAX** et utiliser les valeurs de **code** comme index. Attention, personne ne vous garantit ici une quelconque constance de version en version du sous-système et du noyau. De plus, **EV_MAX** étant, pour l'instant, défini à **0x1ff**, cela représente un grand tableau de 512 octets qui n'est rien d'autre qu'un layout de clavier tel qu'il en existe déjà dans le système. Bref, vous réinventez la roue et ré-implémentez la prise en charge d'un clavier complet, là où d'autres solutions seront sans doute plus adaptées.

Terminons cette partie en précisant que les évènements se propagent généralement dans tout le système. C'est ce qui explique que notre programme de démonstration, utilisé sur un clavier déjà pris en charge par X, affichera bien les codes, mais que la sortie sur le terminal sera parasitée par les caractères traduits par X et son module **evdev**. Il est possible de changer ce comportement en capturant les évènements de manière définitive. Un simple IOCTL suffit :

```
if (ioctl(fd, EVIOCGRAB, 1)) {
  perror(strerror(errno));
}
```

Attention cependant à ne pas utiliser ces quelques lignes si vous ne disposez que d'un seul clavier pour vos tests ! En effet, il ne vous sera plus possible, dès lors, de contrôler le fonctionnement du programme, car X ne recevra aucun évènement du périphérique. Il vous faudra alors utiliser une autre machine et une connexion distante (SSH) pour tuer votre programme (oui, c'est du honteux vécu et on comprend généralement une fraction de seconde après validation).

Remarque **Lire certes, mais écrire aussi !**

Les périphériques supportés par le sous-système d'entrée ne sont pas en lecture seule. J'arrêterai là, de suite, les rêveurs et rêveuses : n'espérez pas déplacer la souris en écrivant dans **/dev/input/*** ;)

Écrire dans le pseudo-fichier correspondant à un périphérique permet d'en changer le comportement, le configurer et, éventuellement, lui faire afficher des données. Dans le cas d'un clavier, ce sont, par exemple les leds NumLock, CapsLock et ScrollLock qui sont visées.

Le simple code suivant vous permettra, par exemple, d'allumer NumLock :

```
struct input_event myev;

myev.type = EV_LED;
myev.code = LED_NUML;
myev.value = 1;

if(write(fd, &myev, sizeof(myev)) < 0) {
  fprintf(stderr, "writre error (%s)\n",
       strerror(errno));
}
```

myev.type est le type d'évènement tel que spécifié dans la liste donnée précédemment, **myev.code** désigne la led accédée et **myev.value** à **1** active la led, et à **0** l'éteint.

Prenez simplement garde à bien ouvrir le fichier **/dev/input/** en **O_RDWR** et non **O_RDONLY** comme nous l'avons fait précédemment. Il faut également prendre en compte le fait qu'il peut y avoir interférence avec le support du clavier dans X ou dans la console, si vous n'avez pas pris les mesures nécessaires. La led Numlock par exemple, n'est qu'un témoin lumineux et ne contrôle absolument pas la configuration du pavé numérique. La touche « num » (code **69**) informe le système de la volonté de l'utilisateur de passer en numérique ; en conséquence, ce dernier remappe les touches et informe l'utilisateur du changement via la led **LED_NUML**.

Notez également qu'il vous est possible de récupérer via **ioctl (fd, EVIOCGBIT (EV_LED, sizeof (led_bitmask)), led_bitmask)**, la liste des leds disponibles sur le clavier et, via l'IOCTL **EVIOCGLED**, les leds présentement actives.

Conclusion

L'utilisation du sous-système de gestion d'entrées de Linux présente un avantage certain pour les systèmes embarqués, mais également pour de simples installations à base de machines standards. Une identification doublée d'une authentification par code numérique peut ainsi facilement être mise en œuvre. Pour peu que l'on ajoute un simple lecteur RFID USB/série et un système d'affichage (LCD alphanumérique, écran standard ou afficheur à leds), il est possible de créer quelque chose de polyvalent, d'évolutif et s'intégrant facilement dans un système existant.

Il est tout aussi clair que les éléments donnés ici permettent de créer un redoutable keylogger. Tout ce qui manque c'est une simple boucle pour parcourir **/dev/input/*** à la recherche du premier périphérique qui ressemble à un clavier. J'espère avoir titillé en vous un peu de la paranoïa propre aux personnes qui comprennent les machines. Regarder de temps en temps quels sont les processus qui ouvrent les fichiers de **/dev/input/** n'est pas une perte de temps, pas plus que de se pencher sur le fonctionnement des règles udev. ■

En étudiant les périphériques via le système de gestion d'entrées de Linux, nous avons découvert en quoi un pavé numérique pour portable était différent d'un pavé numérique standard : il ne maintient pas NumLock activé en permanence, mais l'active avant chaque pression sur les touches et le désactive juste après le relâchement. Ceci évite de placer le clavier du laptop en mode numérique et il conserve alors son agencement (layout) normal.

LE MONDE MERVEILLEUX DE LD_PRELOAD

par Étienne Dublé
[Ingénieur de Recherche LIG / CNRS.
Utilisateur GNU/Linux. Auteur de l'outil IPv6 CARE.]

Il existe un mécanisme permettant de modifier le comportement d'un processus en modifiant certains appels de fonction qu'il utilise. Je l'ai mis en œuvre pour la première fois il y a quelques années et depuis, je pourrais presque dire qu'il a rendu ma vie d'informaticien plus agréable. ;) Je vous propose donc de le découvrir.

1 Introduction

Vous savez sans doute que la plupart des programmes Linux utilisent des librairies partagées. La commande **ldd** permet de les lister :

```
$ ldd /bin/grep
  linux-gate.so.1 =>  (0x0033f000)
  libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0x00d74000)
  libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x00110000)
  /lib/ld-linux.so.2 (0x00b8e000)
$ ldd /bin/cat
  linux-gate.so.1 =>  (0x00b59000)
  libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x0043b000)
  /lib/ld-linux.so.2 (0x00e65000)
$
```

L'intérêt des librairies partagées apparaît évident dans cet exemple : on a factorisé une grande partie du code binaire commun à ces deux exécutables. Cela permet de réduire la taille disque (et mémoire) nécessaire et de faciliter la maintenance (si on corrige un bug dans la **libc**, on a juste à mettre à jour cette librairie, pas besoin de recompiler tous les programmes qui l'utilisent).

Quand on lance un exécutable de ce type, une des premières tâches du nouveau processus est de charger ces librairies partagées et de refaire une édition des liens (en particulier pour que les appels de fonction dans le code de l'exécutable pointent vers la fonction correspondante dans la librairie partagée). Cette tâche est codée dans la librairie **/lib/ld-linux.so.x**, que l'on appelle le « *dynamic loader / linker* » **[DYNLDR]**.

Remarque **La compilation en statique**

Dans certains cas, assez rares, on peut juger que l'utilisation des librairies partagées n'est pas adaptée à ce qu'on veut faire (au hasard un live CD avec juste un mini-Linux et XBMC, XBMC étant compilé avec une myriade de librairies qu'il sera le seul à utiliser...). Dans ce cas on peut utiliser l'option **-static** de **gcc** lors de la compilation du programme.

Le mécanisme que je vais vous présenter permet de rajouter la librairie partagée de votre choix dans la liste. On peut de cette manière modifier à volonté le fonctionnement d'un exécutable.

2 Faites-vous respecter par votre machine !

Par ce premier exemple, nous allons explorer ce que permet **LD_PRELOAD**.

2.1 Le problème

Parfois, je trouve que les messages d'erreur renvoyés par ma machine sont un peu agressifs.

```
$ ls xxx.jpg
ls: impossible d'accéder à xxx.jpg: Aucun fichier ou dossier de ce
type
$
```

Pourquoi tant de haine ?? Je suis déjà désolé de ne pas retrouver mon fichier, j'apprécierais donc que cette machine fasse preuve d'un minimum de compassion à mon égard, plutôt que de me renvoyer un message si abrupt ! Je crois même y discerner une vague insinuation que je ne tape pas les bonnes commandes ! Non mais c'est qui le chef ??

LD_PRELOAD va nous permettre de lui apprendre la politesse.

2.2 Premier essai

La première étape est de trouver quelle fonction affiche ces messages insultants. Pour cela, on peut utiliser **ltrace**, qui affiche les appels de fonction effectués par le processus :

```
$ ltrace -o $(tty) ls xxx.jpg 2>&1 | sed -e 's/.*/# ICI LE COUPABLE
! #/g'
[...]
__errno_location()                              = 0xb7797694
# ICI LE COUPABLE ! #
error(0, 2, 0xb758fbd4, 0x8060de0, 0xb758fbd4)  = 0
exit(2 <unfinished ...>
[...]
$
```

Normalement, c'est-à-dire sans l'option **-o <fichier>**, **ltrace** affiche la trace des fonctions détectées sur la sortie d'erreur. Or, **ls** va envoyer son message d'erreur sur cette même sortie d'erreur, ce qui peut rendre les choses difficiles à lire. J'avoue que dans ce cas simple, la difficulté aurait été toute relative, mais il m'a été difficile de résister à chercher la petite commande qui va bien. ;)

C'est donc apparemment la fonction **error()** qui est responsable de ma mauvaise humeur.

Après avoir regardé un peu sa page de manuel, on peut redéfinir cette fonction pour imiter son comportement en l'adaptant à notre besoin.

Dans cette première version, pour simplifier, on ne va pas gérer les deux premiers paramètres de cette fonction (**status** et **errnum**) et on sortira dans tous les cas à la fin (en théorie cela dépend de **status**).

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

#define PREFIXE_POLI    "Monsieur, vous m'en voyez desolee, " \
        "mais je crois que la commande a renvoye une erreur... " \
        "je vous l'indique ci-apres.\n" \
        "Je suis a votre disposition pour toute autre requete ; " \
        "je suis entierement devouee a vous satisfaire."

void error(int status, int errnum, const char *format, ...)
{
        va_list ap;
        char *chaine_formatee;

        // formatons la chaine de l'erreur
        va_start(ap, format);
        vasprintf(&chaine_formatee, format, ap);
        va_end(ap);

        // affichons cette chaine avec un
        // prefixe qui rend les choses plus agreables
        fprintf(stderr, "%s\n%s\n", PREFIXE_POLI, chaine_formatee);

        // liberons la memoire
        free(chaine_formatee);

        // on quitte
        exit(EXIT_FAILURE);
}
```

La seule chose un peu moins évidente dans cette fonction, si vous n'y êtes pas habitué, est la gestion des arguments variables (avec les macros **va_start()**, **va_end()** et la fonction **vasprintf()**, pour lesquelles vous pourrez au besoin consulter les pages de manuel correspondantes).

Compilons ce code (**my_error.c**) sous la forme d'une librairie partagée **libmy_error.so** :

```
$ gcc -Wall -fPIC -c -o my_error.o my_error.c
$ gcc -shared -fPIC -Wl,-soname -Wl,libmy_error.so -o libmy_error.so
my_error.o
```

Ce petit *makefile* nous permettra d'automatiser la chose :

```
CC=gcc
LD=gcc

CFLAGS_OYNLIB=$(CFLAGS) -Wall -fPIC
LOFLAGS_OYNLIB=$(LOFLAGS) -fPIC -shared

OBJECTS=my_error.o

all: libmy_error.so

libmy_error.so: $(OBJECTS)
  $(LD) $(LOFLAGS_OYNLIB) -Wl,-soname -Wl,$@ -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS_OYNLIB) -c -o $*.o $*.c
```

On va ensuite charger cette librairie partagée au démarrage de notre processus **ls xxx.jpg**, en plus des librairies normales dont a besoin l'exécutable **ls**. Il suffit pour cela de définir la variable d'environnement **LD_PRELOAD** avec le chemin vers notre librairie :

```
$ export LD_PRELOAD=$PWD/libmy_error.so
```

Au lancement de cette commande **ls xxx.jpg**, la fonction **error()** sera donc définie avec exactement le même prototype dans 2 librairies partagées différentes (la nôtre et la **libc**).

Mais, comme on parle de *PRE*loading, notre librairie sera chargée la *PRE*mière, et c'est donc elle qui recevra les appels à cette fonction.

Testons :

```
$ ls xxx.jpg
Monsieur, vous m'en voyez desolee, mais je crois que la commande a
renvoye une erreur... je vous l'indique ci-apres.
Je suis a votre disposition pour toute autre requete ; je suis
entierement devouee a vous satisfaire.
impossible d'accéder à xxx.jpg
$
```

Aaaah c'est quand même plus agréable. :)

2.3 dlsym(RTLD_NEXT, ...), le chaînon manquant

La fonction **error()** de notre exemple était simple, donc facile à imiter. Mais si on veut implémenter toutes les caractéristiques de cette fonction (gestion de **status** et **errnum**, écriture vers la sortie d'erreur et non pas la sortie standard, ajout du message correspondant à **strerror(errno)**, etc.), on va finalement compliquer notre code de manière assez importante. Ce qu'on voudrait faire, c'est juste afficher notre message sympathique, puis passer la main à la fonction **error()** originelle, celle de la **libc**.

Le souci, c'est qu'en écrivant une fonction avec le même prototype, on a, en quelque sorte, caché cette fonction originelle : si on tente de l'appeler directement, c'est vers notre fonction que l'on sera redirigé.

On peut néanmoins obtenir un pointeur vers cette fonction **error()** originelle de la manière suivante :

```
typeof(error) *libc_error;
[...]
libc_error = dlsym(RTLD_NEXT, "error");
```

Quelques explications s'imposent. La fonction **dlsym()** fait partie de la librairie **libdl**. Cette librairie permet d'implémenter un système de *plugins* dans une application, le plus souvent en utilisant uniquement les 3 fonctions suivantes :

- **dlopen()** pour charger une librairie ;
- **dlsym()** pour obtenir un pointeur vers un des symboles de cette librairie (en général, on lui passe un nom de fonction et on obtient un pointeur vers cette fonction) ;
- **dlclose()** une fois qu'on n'a plus besoin de cette librairie.

Notre technique basée sur **LD_PRELOAD** est un cas un peu particulier. Premièrement, les librairies en question, et plus particulièrement la **libc**, ont déjà été chargées au lancement du processus. Donc, on ne va pas utiliser **dlopen()** ni **dlclose()**, et **dlsym()** va devoir chercher le symbole dans ces librairies déjà chargées. Deuxièmement, on veut pointer sur la fonction **error()** de la **libc** et non celle de **libmy_error.so**. On doit donc indiquer à **dlsym()** qu'il faut commencer la recherche après la librairie courante, à savoir **libmy_error.so**. En indiquant **RTLD_NEXT** comme premier argument, **dlsym()** va satisfaire ces deux exigences.

On sait maintenant tout ce qu'il faut pour écrire notre deuxième version :

```
#define _GNU_SOURCE
#include <dlfcn.h> // pour dlsym()
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

#define PREFIXE_POLI    "Monsieur, vous m'en voyez desolee, " \
        "mais je crois que la commande a renvoye une erreur... " \
        "je vous l'indique ci-apres.\n" \
        "Je suis a votre disposition pour toute autre requete ; " \
        "je suis entierement devouee a vous satisfaire."

void error(int status, int errnum, const char *format, ...)
{
        typeof(error) *libc_error;
        va_list ap;
        char *chaine_formatee;

        // affichons le message sympathique
        fprintf(stderr, "%s\n", PREFIXE_POLI);

        // formatons la chaine de l'erreur
        va_start(ap, format);
        vasprintf(&chaine_formatee, format, ap);
        va_end(ap);

        // recuperons un pointeur vers la "vraie"
        // fonction error() de la libc
        libc_error = dlsym(RTLD_NEXT, "error");

        // appelons cette fonction
        (*libc_error)(status, errnum, "%s", chaine_formatee);

        // liberons la memoire
        free(chaine_formatee);
}
```

On peut remarquer que ce code n'est guère plus complexe que le précédent, alors que mine de rien on a mis en place toutes les fonctionnalités de la fonction **error()** originelle...

Testons :

```
$ make
gcc  -Wall -fPIC -c -o my_error.o my_error.c
gcc  -fPIC -shared  -Wl,-soname -Wl,libmy_error.so -o libmy_error.so
my_error.o
$ LD_PRELOAD=$PWD/libmy_error.so ls xxx.jpg
Monsieur, vous m'en voyez desolee, mais je crois que la commande a
renvoye une erreur... je vous l'indique ci-apres.
```

```
Je suis a votre disposition pour toute autre
requete ; je suis entierement devouee a vous
satisfaire.
ls: impossible d'accéder à xxx.jpg: Aucun
fichier ou dossier de ce type
$
```

3 Quelques infos en plus

A ce stade, vous avez les bases pour utiliser **LD_PRELOAD**. Mais ce monde merveilleux a beau être merveilleux, il l'est encore plus quand on connaît les quelques informations qui suivent.

3.1 Le fichier /etc/ld.so.preload

Le fait que **LD_PRELOAD** soit une variable d'environnement apporte une certaine souplesse. Vous pouvez par exemple la définir en haut d'un script qui va lancer l'exécutable que vous voulez patcher ou, pour une configuration plus permanente, dans un fichier chargé au démarrage de votre session (**~/.bashrc** au autre).

Il existe également une alternative. Plutôt que d'indiquer le chemin de votre librairie dans la variable **LD_PRELOAD**, vous pouvez l'indiquer dans le fichier **/etc/ld.so.preload**. Dans ce cas, *tous les processus qui seront lancés par la suite* (à quelques rares exceptions près, voir plus loin), par n'importe quel utilisateur, y compris si vous relancez la machine (!!), *voudront charger votre librairie*. Ceci est utile dans le cas où vous avez trouvé l'optimisation qui tue et que vous voulez patcher tous les programmes (ou toute une catégorie de programmes) du système ! Contrairement à ce que vous pouvez penser, cela arrive.

Quoi qu'il en soit, il est parfois utile de déterminer quel programme on est en train de patcher, par exemple pour appliquer sélectivement votre patch à certains programmes. En ce qui me concerne, j'ai utilisé deux astuces dans ce but, que je vous laisse expérimenter :

- **getenv("_")** permet d'obtenir la ligne de commandes que **bash** est en train d'exécuter ;
- on peut également lire le fichier **/proc/self/cmdline**.

3.2 Les limites du LD_PRELOAD-ing

Il existe une petite minorité d'exécutables que vous ne pourrez pas patcher avec **LD_PRELOAD**. Les chances que vous tombiez sur ce genre de limitations sont finalement réduites, sauf si vous commencez à en mettre un peu partout comme moi... Et dans ce cas, ce paragraphe vous permettra de gagner pas mal de temps. Les anglophones pourront avoir plus de détails sur ces limitations en lisant le paragraphe correspondant dans le guide utilisateur d'IPv6 CARE, un outil dont je reparlerai plus loin (voir **[IC_USERGUIDE]**).

La première catégorie d'exécutables qui entre en ligne de comman^W^W de compte est bien sûr celle des binaires compilés en statique, mais ils sont peu nombreux.

L'utilisation de **LD_PRELOAD** est également limitée pour les exécutables dont le bit SUID ou SGID est activé (voir encadré).

Remarque **Rappels sur les bits SUID et GUID**

Voici à quoi sert le bit SUID : imaginons que **user1** crée un script exécutable **do_it.sh**, qu'il donne les droits d'exécution à tout le monde et qu'il mette à 1 le bit SUID (**chmod +s do_it.sh**) ; dans ce cas, si **user2** lance ce script, celui-ci sera exécuté en tant que **user1**, car **user1** est le propriétaire de **do_it.sh**. Cela permet par exemple à **user2** d'effectuer des actions pour lesquelles il n'a pas suffisamment de privilèges, contrairement à **user1**. Le bit SGID fonctionne de la même façon, mais pour le groupe.

En particulier, le mécanisme **LD_PRELOAD** est désactivé si vous tentez de lancer un exécutable dont vous n'êtes pas propriétaire et qui a le bit SUID (ou GUID) activé, car cela engendrerait une faille de sécurité évidente : il suffirait de repérer une fonction que cet exécutable utilise pour exécuter un code arbitraire avec les droits d'un autre utilisateur (ou groupe). En revanche, si c'est le propriétaire qui lance cet exécutable, pas de souci, le mécanisme **LD_PRELOAD** est bien pris en compte. Les commandes suivantes résument ce comportement :

```
$ export LD_PRELOAD="$PWD/libmy_error.so"
$ ls xxx.jpg  # verification que ça marche
Monsieur, [...]
ls: impossible d'accéder à xxx.jpg: Aucun fichier ou dossier de ce type
$ sudo chmod +s /bin/ls    # on met le bit SUID
$ ls xxx.jpg # verification que ça ne marche plus
ls: impossible d'accéder à xxx.jpg: Aucun fichier ou dossier de ce type
$ sudo su          # passons root, root etant proprietaire de /bin/ls
# export LD_PRELOAD="$PWD/libmy_error.so"
# ls xxx.jpg # verification que ça re-marche
Monsieur, [...]
ls: impossible d'accéder à xxx.jpg: Aucun fichier ou dossier de ce type
#
```

Les mécanismes de contrôle d'accès du type SELinux ou AppArmor peuvent également empêcher un exécutable donné d'être **LD_PRELOAD**-able. Dans ce cas, il est nécessaire de modifier leur configuration. Il faut en particulier donner à ces exécutables le droit d'accéder à votre librairie. En consultant **[IC_USERGUIDE]** vous aurez des indications sur les modifications à effectuer.

Un autre point de difficulté est tout simplement le passage de la variable d'environnement **LD_PRELOAD** vers les sous-processus. En effet, certains exécutables peuvent choisir de ne pas transmettre l'intégralité de leurs variables d'environnement à leurs sous-processus.

Dans le doute, si on veut vérifier que l'on n'est pas tombé dans un des cas énoncés ci-dessus, et que notre librairie sera donc bien préloadée au lancement d'un exécutable donné, on peut tout bêtement utiliser **ldd**. Voici un exemple :

```
$ export LD_PRELOAD=$PWD/libmy_error.so
$ ldd /bin/ls
  linux-gate.so.1 =>  (0x00579000)
  /home/etienne/libmy_error.so (0x0059c000)
[...]
  libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x005a0000)
[...]
$
```

On voit ici que notre librairie sera bien préloadée.

Enfin, on peut (en cherchant bien !) rencontrer également quelques difficultés au niveau des fonctions que l'on veut patcher.

Par exemple, si un programme appelle une fonction **a()** de la **libc** et que la fonction **a()** appelle la fonction **b()**, également dans la **libc**, alors on ne pourra pas patcher cet appel à la fonction **b()** (car cet appel a été lié de manière statique au moment de la compilation de la **libc**).

D'autre part, si on veut patcher des fonctions de très bas niveau et les rappeler dans notre code avec **dlsym()**, il faut vérifier qu'elles ne sont pas utilisées dans le code de la fonction **dlsym()** sans quoi on va faire une jolie boucle... Eh oui, j'ai déjà eu besoin de préloader la fonction **close()** ! Dans ce cas, on peut s'en sortir si on s'autorise à bidouiller un peu, en recherchant une autre fonction de la **libc** qui fasse le même travail (la fonction **__close()** dans ce cas) et en l'appelant directement. En effet, pour des raisons d'optimisation, dans la **libc** il y a souvent plusieurs fonctions qui font quasiment la même chose, à une vérification de paramètres près par exemple.

3.3 La fonction dlvsym()

Il peut arriver que certaines fonctions de la **libc** aient changé de prototype au cours des versions (c'est rare mais cela existe, c'est le cas par exemple de **gethostbyname_r()**). Pour des raisons de compatibilité ascendante, la **libc** fournit dans ce cas une fonction pour chacun des différents prototypes connus, taggée avec un numéro de version. Dans ce cas, la fonction **dlvsym()** permet de spécifier ce tag et donc de sélectionner le bon prototype, contrairement à **dlsym()**.

3.4 L'aspect parallélisme

Il faut garder à l'esprit qu'une fois que votre variable **LD_PRELOAD** est définie (et à plus forte raison si vous utilisez le fichier **ld.so.preload**), tous les processus qu'on lance dans cet environnement chargent la librairie... Ils peuvent être nombreux ! Surtout si on parle de la phase de démarrage du système par exemple. Tout cela signifie que votre code va être potentiellement exécuté de multiples fois par des processus en parallèle, et même parfois par plusieurs *threads* d'un même processus. Il faut donc, d'une part, que votre code supporte ce parallélisme et d'autre part, éviter de calculer 30000 décimales de pi à cet endroit-là.

Prenons l'exemple où vous avez écrit un programme qui calcule la météo du lendemain. Vous en êtes super fier parce que les prévisions générées sont fiables à 100%. Seul problème, on les obtient au bout de 3 jours, même avec votre version *multi-threads*. Vous pensez que la fonction **f()** de la **libc** est celle qui prend trop de temps. Avec **LD_PRELOAD** il est facile d'implémenter un petit outil de *profiling* pour calculer le temps cumulé passé dans cette fonction **f()** :

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <time.h>

void f()
{
        static typeof(f) *libc_f = NULL;
        static time_t temps_cumule = 0;
        time_t date_de_l_appel;

        // recuperation du pointeur vers
        // la fonction f() dans la libc
        if (libc_f == NULL)
        {
                libc_f = dlsym(RTLD_NEXT, "f");
        }

        // sauvegardons le moment de l'appel
        date_de_l_appel = time(NULL);

        // appelons cette fonction
        (*libc_f)();

        // ajoutons au temps cumule
        temps_cumule += time(NULL) - date_de_l_appel;

        // affichons
        printf("%ld\n", temps_cumule);
}
```

La précision à la seconde près de la fonction **time()** n'est sans doute pas très adéquate (il existe **gettimeofday()**), mais c'est juste un exemple.

Dans ce code, on peut remarquer que j'ai utilisé 2 variables statiques : la première, **libc_f**, me permet de ne faire l'appel à **dlsym()** qu'une seule fois pendant toute la durée du processus, ce qui permet un gain évident en matière de performances ; la deuxième, **temps_cumule**, me permet de conserver l'information entre deux appels, ce qui est nécessaire pour notre exemple.

Le problème quand on utilise ces variables statiques, c'est que notre code n'est sans doute plus *thread-safe*. En effet, à la ligne **temps_cumule += [...]**, le thread doit lire la valeur existante, puis stocker la nouvelle valeur ; donc, avec notre programme météo multi-threads, on a le risque que 2 threads arrivent en même temps et qu'ils lisent la même valeur existante, ce qui fausserait le calcul.

Ces problèmes de multi-threading peuvent paraître difficiles à résoudre, car dans notre contexte de **LD_PRELOAD**-ing, c'est le programme qui crée et qui gère les threads, pas notre librairie. Il existe pourtant une solution simple et élégante : le *Thread-Local Storage* (voir **[TLS]**). Cette fonctionnalité nous permet d'exploiter un espace de stockage propre à chaque thread, ce qui évite tout problème de concurrence. Pour cela (avec **gcc** en tout cas) on déclare simplement notre variable avec le mot-clé **__thread** :

```
static __thread time_t temps_cumule = 0;
```

En faisant cela, notre petit outil de profiling va espionner chaque thread indépendamment les uns des autres. Il ne restera plus qu'à trouver un moyen d'identifier les résultats de chaque thread par rapport aux autres... Peut-être avec quelque chose comme :

```
static __thread long thread_id = 0;
[...]
if (thread_id == 0)
{
        thread_id = get_random_number();
}
[...]
printf("Thread %ld: %ld\n", thread_id, temps_cumule);
```

3.5 L'aspect performances

Cet aspect performances est fortement lié à l'aspect parallélisme que nous venons de voir.

En ce qui concerne le système de **LD_PRELOAD**-ing en lui-même, on n'a pas de perte de performance par rapport au programme exécuté normalement (sauf bien sûr si votre code est beaucoup plus lent^W^Wévolué que le code originel que vous patchez). En effet, comme on l'a vu, la modification est mise en place au moment de l'édition des liens, où les appels de fonctions qui nous intéressent sont liés à notre librairie au lieu d'une autre. Ensuite, durant toute la vie du processus, la chose est en place et le système n'a aucune sur-couche à gérer.

Il existe d'autres procédés qui permettent de modifier le comportement des processus en cours d'exécution (voir encadré). Mais les performances avec ce genre de techniques sont largement dégradées, ce qui les cantonne en général à des programmes de *debugging*. Les performances natives qu'offre **LD_PRELOAD** sont ainsi un avantage remarquable de cette technique.

Pour ce qui est de l'appel à **dlsym()** par contre, il s'agit d'une opération que l'on fait en plus par rapport au processus normal. Donc, si on met en place un patch pour une fonction appelée de manière récurrente, il est important de n'effectuer cet appel à **dlsym()** qu'une seule fois (voir l'exemple du paragraphe précédent).

3.6 L'aspect maintenance

Imaginez que vous changiez votre processeur et que votre serveur X ne démarre plus. Après un peu de **strace** / **ltrace**, vous diagnostiquez le problème : le serveur X appelle une fonction bas niveau **f()** de la **libc** censée parler à la carte graphique pour l'initialiser, mais celle-ci est un peu vieille et ne répond pas assez vite. Du coup, il y a un timeout. Avec votre ancien processeur, qui était de la même génération que votre carte graphique, ça marchait sans souci... Qu'à cela ne tienne, en 45 minutes vous avez téléchargé les sources de ce serveur X, rajouté une temporisation au moment de l'appel à cette fonction **f()**, recompilé, relancé, ça marche, vous

Remarque **Modification de processus en cours d'exécution**

Parmi les autres systèmes qui permettent de modifier un processus en cours d'exécution, le plus utilisé est sans doute celui basé sur la fonction **ptrace()** (voir **man ptrace**). C'est la méthode qu'utilise **gdb**, par exemple. Dans ce cas, on a un processus qui en contrôle un autre. Avec cette méthode, on peut faire bien plus que du *debugging* pas-à-pas : on peut par exemple modifier la mémoire du processus esclave... Pour plus de liberté encore, certains outils émulent carrément un processeur, c'est ce que fait **valgrind** par exemple.

envoyez un mail à probleme-de-fonction-f@x.org et vous oubliez cette mésaventure.

Un an plus tard, vous mettez à jour votre système avec une nouvelle version. Le serveur X lui aussi est mis à jour, tac, on relance, ah mince ça ne marche plus ! Ça vous rappelle quelque chose... Un nouveau petit coup de **strace** / **ltrace**, ah oui c'est vrai ! Bon, cette fois vous décidez de faire un patch basé sur **LD_PRELOAD**, parce qu'en fait, ce sera plus simple, il n'y aura pas besoin des sources et de tout recompiler... En 10 minutes vous avez votre patch. Vous avez défini votre variable **LD_PRELOAD** sur la ligne qui lance le serveur X et vous oubliez cette mésaventure... Cette fois pour de bon ! Car le serveur X aura beau être mis à jour, votre patch est indépendant et ne sera donc pas écrasé. Et puis ce n'est pas un simple patch *applicable aux sources de xserver-xorg version x.y.z*, mais bien un patch *applicable à n'importe quel programme qui utilise la fonction f()*... autant dire qu'il est pérenne. C'est l'autre avantage de **LD_PRELOAD**.

3.7 Les langages de programmation

Dans cet article, j'utilise le langage C car c'est le plus naturel pour écrire des librairies partagées. Mais rien ne vous empêche par exemple de les écrire en C++ et de préfixer les fonctions avec **extern "C"**. Ou même de les interfacer avec quelque chose d'autre, tout est permis tant que les performances sont suffisantes...

Pour ce qui est de l'application de votre patch, il faut garder à l'esprit que celui-ci aura uniquement un effet sur les codes compilés. Par exemple, si vous exécutez une classe Java compilée en bytecode, votre patch va agir sur la JVM (la machine virtuelle Java, c'est-à-dire la commande **java**). La conclusion serait la même avec des langages interprétés comme Python ou Perl : ce que vous patchez, c'est l'interpréteur.

Le champ d'application de **LD_PRELOAD** dans ce cas est donc plus limité. En particulier, les techniques de debugging (par exemple la technique de profilage implémentée précédemment) seront difficilement applicables, car le lien entre le code source à corriger et le comportement de l'interpréteur n'est pas direct.

Par contre, **LD_PRELOAD** peut servir pour d'autres usages que le debugging... Dans le cas de bytecode Java, si on n'a pas d'accès aux sources, l'utilité de **LD_PRELOAD** est assez évidente. Pour un langage interprété, il est en général plus simple de modifier les sources directement. (Mais, comme on l'a vu, un patch basé sur **LD_PRELOAD** peut être plus avantageux sur le plan de la maintenance.)

Pour conclure sur ce sujet, on va voir plus loin plusieurs exemples où on modifie le comportement de toute une catégorie de programmes qui tournent sur une machine, quel que soit le langage de programmation utilisé. A ce titre **LD_PRELOAD** peut rendre de grands services.

3.8 Les autres UNIX

J'ai pu tester avec succès le mécanisme **LD_PRELOAD** + **dlsym()** avec FreeBSD et OpenSolaris, en plus de Linux bien sûr. Il semble que cela puisse également marcher sous Mac OS X (en utilisant la variable **DYLD_INSERT_LIBRARIES** comme équivalent de **LD_PRELOAD**).

4 LD_PRELOAD c'est simple et efficace...

Afin que vous puissiez entrevoir l'espace de possibilités qu'ouvre cette technique, cette partie donne quelques exemples additionnels d'applications possibles.

4.1 ... pour tester la cohérence d'un code

Imaginez que vous travaillez dans l'industrie sur un logiciel vieux de 15 ans et écrit en C. Il doit tourner 24h/24, 7j/7 pour superviser la production, dans des usines qui fabriquent des bouteilles en verre. Mais ça c'est juste la théorie : vous avez noté que dans certaines usines le serveur se relance toutes les 3 semaines environ, à cause d'un manque de mémoire. Il doit y avoir une fuite quelque part.

Il vous faut donc trouver un moyen efficace pour cibler un problème sur une base de code plutôt imposante. Outre les solutions existantes (**valgrind** par exemple), vous tombez sur un tutoriel concernant **LD_PRELOAD**, où les fonctions **malloc()** et **free()** sont redéfinies pour rechercher une incohérence. Intéressant...

En surchargeant ces mêmes fonctions, on peut également obtenir des statistiques des allocations effectuées par le programme... Et une fois qu'on a ces statistiques, pourquoi ne pas en tirer parti et implémenter un gestionnaire de mémoire plus efficace pour ce programme ? Ou même, un *garbage collector* ?

Entre-temps, la fuite mémoire a été découverte dans un ajout de code récent. Mais vous gardez ces réflexions dans un coin de votre tête...

4.2 ... pour gérer un effet de bord complètement imprévisible

Un ou deux ans plus tard, vous travaillez sur un projet de grille de calcul (en gros un réseau de centres de calcul) qui recueille (entre autres) les données générées par l'accélérateur

de particules « LHC ». S'agissant d'une grille de calcul mondiale reposant sur un *middleware* gourmand en adresses publiques, la problématique du passage à IPv6 est primordiale et c'est sur ce thème que vous travaillez. Vous avez donc milité comme il faut pour que tous les composants logiciels deviennent *IPv6-compliant*. Tout cela avance bien, sans souci majeur, jusqu'au jour où vous recevez un e-mail qui vient du CERN (qui gère le LHC) et qui dit en gros « depuis la migration IPv6 de tel composant, notre *load-balancing* DNS ne fonctionne plus ; donc on va revenir à la version précédente ». Un tel effet de bord vous paraît assez improbable, surtout que les machines en question n'ont qu'une adresse IPv4. Mais le CERN étant le contributeur le plus important de la grille, un tel retour en arrière pourrait presque compromettre la migration, donc vous tentez aussitôt de diagnostiquer le souci. Et là, vous vous rendez compte qu'effectivement, dans leur environnement réseau bien précis, la fonction **`getaddrinfo()`** (introduite lors de la migration) trie les adresses retournées par le DNS et présente toujours la même en première position ; donc les clients interrogent tous le même serveur. Qu'à cela ne tienne vous avez plusieurs cordes à votre arc. Vous commencez donc à réfléchir à la technique que vous allez employer pour réordonner ces adresses aléatoirement en sortie de cette fonction...

Sauf si vous êtes chez le dentiste et que vous feuilletez ce magazine de manière vraiment distraite, vous devriez deviner la fin de l'histoire... **[RDMZGAI]**

4.3 ... pour gérer une technologie que le programmeur n'a pas prévue ou ne maîtrisait pas

En fait, ce n'était pas tout à fait la fin de l'histoire.

Alors, où en est-on avec cette migration de middleware ? Eh bien on a maintenant quasiment 100% du code qui est migré. Par contre, un tiers des composants n'ont pas l'air de fonctionner dans un environnement IPv6. Ah bon, comment ça ??? Eh bien a priori, certaines des librairies que le middleware utilise ne sont pas compatibles. Et comme on ne gère pas le code de ces librairies externes au projet, il n'est pas toujours possible de savoir si un jour elles seront migrées et si oui, quand.

Résumons la situation : vous aimeriez donc modifier le comportement de certains morceaux de logiciel (pour les rendre compatibles IPv6), sans modifier leur code... Cela ne vous rappelle pas quelque chose ? (Ça commence par LD et ça finit par PRELOAD...).

Vous vous mettez donc au travail. Bien entendu, là il ne s'agit plus d'un simple patch pour un programme isolé : vous voulez rendre compatibles IPv6 tous les processus tournant sur une machine. Bon OK, vous vous autorisez quelques exceptions (voir paragraphe sur les limites), mais votre but final est que tous les processus du middleware soient capables de parler IPv6, qu'ils soient codés en shell, en C/C++, en Java, en Python ou en Perl !! (Entre nous, y a-t-il vraiment eu une concertation avant de commencer à coder ce middleware ????)

Vous décidez alors de partir de votre projet existant, IPv6 CARE, qui permet déjà aux développeurs de détecter les appels de fonction réseau effectués par leur programme et d'en déduire un diagnostic de compatibilité IPv6 (« check » mode, voir **[IPV6CARE]**). Avec ce nouveau « patch » mode, il ne faudra pas se contenter de détecter ces appels, mais carrément les corriger. Par exemple, si l'on s'aperçoit qu'un serveur effectue un appel à la fonction **`accept()`** avec en paramètre un socket IPv4, il est évident qu'il ne sera pas capable de recevoir des clients IPv6. Donc, à la place du **`accept()`**, on devra dans ce cas créer un socket IPv6, faire un **`select()`** pour attendre une connexion sur le socket IPv4 ou le socket IPv6 et enfin, réaliser l'**`accept()`** sur le socket qui a réveillé le **`select()`**. Trois adaptations de ce genre plus tard, avec tout ce qu'il faut autour et pas mal de tests, vous avez atteint votre objectif (voir **[IC_PATCH]**).

4.4 ... pour adapter un programme complètement borné à des contraintes imprévues

Bon, je suis désolé d'être aussi directif au sujet de votre destin fictif, mais maintenant que ça tourne au niveau d'IPv6, vous allez passer à autre chose, en l'occurrence à du monitoring réseau, car la grille en a besoin. Vous travaillez donc dorénavant sur un système qui comprend un serveur central et des sondes à installer dans chaque site. Ces sondes sont des systèmes Linux avec différents outils de diagnostic réseau. L'un de ces outils permet de mesurer la bande passante entre 2 sondes (donc le plus souvent entre 2 sites).

Pour vos tests, vous avez installé 2 sondes virtualisées sur un serveur Xen. Pas de bol, l'utilitaire pour tester la bande passante refuse de faire le test. En effet, il vérifie que la synchronisation NTP entre vos sondes fonctionne, or celle-ci n'est pas opérationnelle.

Voici l'explication. Dans sa configuration par défaut, le serveur Xen (*dom0* dans le jargon) et les machines virtuelles qui y sont hébergées (ici les 2 sondes) partagent la même horloge. On peut modifier ce comportement (voir encadré), mais à l'époque vous ne le saviez pas. On peut se dire que si les sondes ont la même horloge, alors évidemment il est inutile de les synchroniser. Mais ce programme ne teste pas les horloges, il fait des appels aux démons NTP sur chaque sonde pour savoir si la synchronisation NTP est OK.

Or, dans cet environnement, une machine virtuelle ne peut pas modifier son horloge (vu que celle-ci est la même pour tous...), donc le démon NTP retourne une erreur de synchronisation.

Remarque **Désynchroniser les horloges sous Xen**

Si la configuration par défaut est effectivement de partager la même horloge entre dom0 et les machines virtuelles hébergées, on peut choisir de les désynchroniser via le sysctl **xen.independent_wallclock**.

Vous cherchez bien, mais ce programme borné n'a pas l'option « ignorer la synchro NTP », donc il commence rapidement à vous donner des boutons avec ses messages répétés affirmant (à tort !) que les sondes ne sont pas synchronisées.

Finalement, vous vous rendez compte que l'interrogation du démon NTP des sondes se fait avec une fonction de l'API NTP, donc il est tout bête de patcher cette fonction pour qu'elle renvoie toujours « synchro OK ». C'est fait en 5 minutes et ça laisse quand même la satisfaction d'avoir plié à vos exigences un programme qui faisait vraiment preuve de mauvaise volonté.

4.5 ... pour mettre de la couleur dans votre vie d'informaticien

Suite à ces diverses expériences où **LD_PRELOAD** vous a bien rendu service, vous décidez d'écrire un article sur ce sujet pour GLMF/Open Silicium.

Pour introduire cet article, vous cherchez un premier exemple accrocheur. Vous vous dites qu'il serait intéressant de patcher les programmes pour qu'ils affichent les messages d'erreur en rouge. Dans un script un peu verbeux par exemple, cela peut être très utile. Vous commencez donc à investiguer ce sujet.

En fait, il suffit de patcher la fonction **error()**... Et d'utiliser des constantes avec des codes d'échappement comme :

```
#define ROUGE  "\e[31m"
#define COULEUR_DEFAUT  "\e[0m"
```

Il faut quand même prendre soin de désactiver ce changement de couleur si la sortie standard n'est pas un terminal (sinon, si on redirige la sortie de commande vers un fichier par exemple, on y retrouve ces séquences d'échappement...). La fonction **isatty()** permet de gérer ce petit souci.

Et s'il y a des lecteurs grincheux qui trouvent que les erreurs ne sont toujours pas suffisamment repérables, on pourra leur répondre qu'il existe une séquence pour écrire en inverse vidéo, ou même pour faire clignoter le texte ! Bien sûr, en contrepartie, on s'éloigne des standards et on peut donc rencontrer des émulateurs de terminaux qui n'implémentent pas ces fonctionnalités. Il faut les tester au préalable avec des commandes du type :

```
$ echo -e "\e[31;7mblablabla\e[0m"
```

Et vérifier que **blablabla** est bien affiché en inverse vidéo sur fond rouge.

Mouais, c'est pas mal comme exemple. Par contre... Pour représenter ces messages en couleur, pas facile de respecter les conventions d'écriture des articles GLMF/Open Silicium ...

Bon, dans ce cas, on ne va pas les mettre en couleur, mais juste les rendre plus agréables...

Conclusion

J'espère que vous avez apprécié cette petite promenade dans le monde merveilleux de **LD_PRELOAD**. A priori, à la lecture de cet article, vous avez acquis suffisamment de points d'expérience pour vous y lancer sans risque, si cela vous dit de jouer le jeu... ■

Remerciements

Je voudrais remercier Bernard Rappachi pour sa relecture de la dernière fois et parce que j'ai apprécié la période où il était mon chef ; Florian Marcos pour nos séances de *brainstorming* sympas (les bonnes idées apparaissent souvent autour d'une bière) ; et enfin, l'ensemble des personnes qui participent à GNU/Linux Magazine et Open Silicium, parce qu'il en résulte un excellent outil de veille technologique.

Liens

[DYNLDR] Il y a tout sur le *dynamic linker/loader* dans sa page de manuel : **man ld.so**

[IC_USERGUIDE] http://sourceforge.net/projects/ipv6-care/files/User%20Guide/

[TLS] http://fr.wikipedia.org/wiki/Thread_Local_Storage

[RDMZGAI] https://twiki.cern.ch/twiki/bin/view/EGEE/RandomizeGetAddrInfo

[IPV6CARE] http://sourceforge.net/projects/ipv6-care/

[IC_PATCH] https://twiki.cern.ch/twiki/bin/view/EGEE/IPv6CARE

Complétez votre collection des anciens numéros de...

Les 4 façons de commander !

Par courrier
En nous renvoyant ce bon de commande.

Par le Web
Sur notre site : www.ed-diamond.com.

Par téléphone
Entre 9h-12h & 14h-18h au 03 67 10 00 20 (paiement C.B.)

Par fax
Au 03 67 10 00 21 (C.B. et/ou bon de commande administratif)

Open Silicium

Bon de commande Open Silicium

Réf.	Désignation	Prix / N°s
OS N°1	L'EMBARQUÉ DEVIENT ACCESSIBLE À TOUS ! Prise en main et évaluation de la carte Mini2440 de FriendlyARM	9,00 €
OS N°2	ARDUINO - Apprenez à exploiter votre module Arduino et à tirer le meilleur du micro-contrôleur AVR d'Atmel	9,00 €
OS N°3	ARM + FPGA = ROBOTIQUE ! Couplez ARM et FPGA pour contrôler efficacement jusqu'à 32 servomoteurs sur plateforme Armadeus/Linux	9,00 €
Tous LMHS	GNU/Linux Magazine Hors-Série	6,50 €

Déjà parus dans le domaine de l'électronique & de l'embarqué, à redécouvrir sur www.ed-diamond.com

LMHS 51 : 6,50 € | LMHS 47 : 6,50 € | LMHS 43 : 6,50 € | Promo LMHS 39 : 2€ | LMHS 38 : 6,50 € | LMHS 27 : 6,50 € | LMHS 25 : 6,50 € | LMHS 24 : 6,50 € | LMHS 23 : 6,50 €

Bon de commande à remplir (ou photocopier) et à retourner aux Éditions Diamond - Open Silicium - BP 20142 - 67603 Sélestat Cedex

	Référence	Prix / N°	Qté	Total
EXEMPLE :	~~OS N°1~~	~~9,00 €~~	~~1~~	~~9,00 €~~
EXEMPLE :	~~LMHS N°47~~	~~6,50 €~~	~~1~~	~~6,50 €~~
		TOTAL :		
		FRAIS DE PORT FRANCE MÉTRO. :		+3,9 €
		FRAIS DE PORT HORS FRANCE MÉTRO. :		+6 €
		TOTAL :		

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Téléphone :

e-mail :

Je choisis de régler par :

❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond

❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

❑ Je souhaite recevoir des infos des Éditions Diamond

❑ Je souhaite recevoir des infos des partenaires des Éditions Diamond

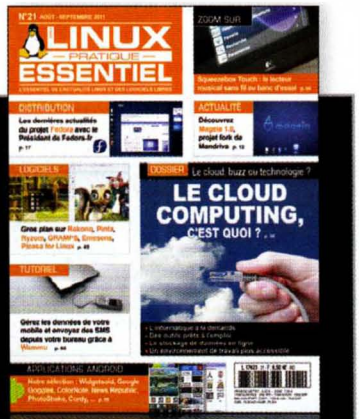
Retrouvez les sommaires et commandez tous nos magazines sur notre site :
http://www.ed-diamond.com

Open Silicium

LES ÉDITIONS DIAMOND CE SONT...

5 MAGAZINES COMPLÉMENTAIRES À LA MESURE DE VOS ENVIES !

BESOIN D'ACTUS CONCERNANT LINUX ?



L'essentiel de l'actualité Linux et des logiciels libres

www.linux-essentiel.com

Tous les 2 mois en kiosque

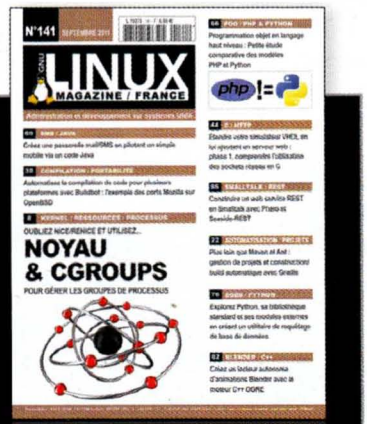
VOUS UTILISEZ LINUX RÉGULIÈREMENT ?

Découvrir, comprendre et utiliser Linux

www.linux-pratique.com

Tous les 2 mois en kiosque

VOUS CODEZ ET ADMINISTREZ DES MACHINES SOUS LINUX ?



Administration et développement sur systèmes UNIX

www.gnulinuxmag.com

Tous les mois en kiosque

VOUS OPTIMISEZ LA SÉCURITÉ DE VOS SYSTÈMES ?

100 % sécurité informatique

www.miscmag.com

Tous les 2 mois en kiosque

ATTENTION NOUVEAUTÉ !

Open Silicium

Vous souhaitez suivre le développement de l'open source dans le secteur de l'embarqué ?

L'open source pour l'électronique & l'embarqué

www.opensilicium.com

Tous les 3 mois en kiosque et toujours disponible sur : www.ed-diamond.com

Abonnez-vous et retrouvez tous les anciens numéros sur : **www.ed-diamond.com**

PRÉSENTATION DU HACK SÉRIE DU BADGE DEAL EXTREME

Parfois, il arrive qu'on achète des gadgets idiots sans pour autant penser qu'on les utilisera sérieusement. Motivé par la simple curiosité et le produit déjà mentalement démonté avant d'être arrivé dans la boîte à lettres en provenance de Chine, il arrive même qu'on ait des bonnes surprises, voire de très très bonnes surprises. C'est le cas pour ce badge acheté chez Deal Extreme et s'avérant être un fantastique matériel grâce à Charles Rincheval et son site digitalspirit.org.

Le produit acheté quelques 9 euros est un *Programmable Scrolling LED Name/Message/Advertising Tag Card Badge*. Un nom imposant pour un gadget tout simple constitué d'une matrice de leds en charliplexing de 29 colonnes et 7 lignes. Disponible en version bleue ou rouge, l'objet est alimenté par une pile CR2032 et présente, à l'arrière, quelques boutons permettant de configurer un message à *scoller*. Personnellement, je n'ai jamais vu personne arborer un tel accessoire, mais sur la base de quelques suppositions, il semblait offrir quelques perspectives intéressantes. A défaut de *reverse* réussi, il aurait toujours été possible d'interfacer les boutons avec un microcontrôleur, voire de tout simplement réutiliser la matrice de leds pour un autre usage.

Mais voilà qui était sans compter l'ingéniosité et la persévérance de Charles Rincheval. C'est par hasard que je suis tombé sur un billet comprenant un lien vers une page du Wiki de digitalspirit.org. Quelle ne fut pas ma surprise en constatant que, non seulement le produit dont je disposais y était décrit en détails, mais de plus, qu'un firmware alternatif et open source était disponible. Firmware permettant, et c'est là une coïncidence troublante, de piloter le badge directement depuis une liaison série TTL !

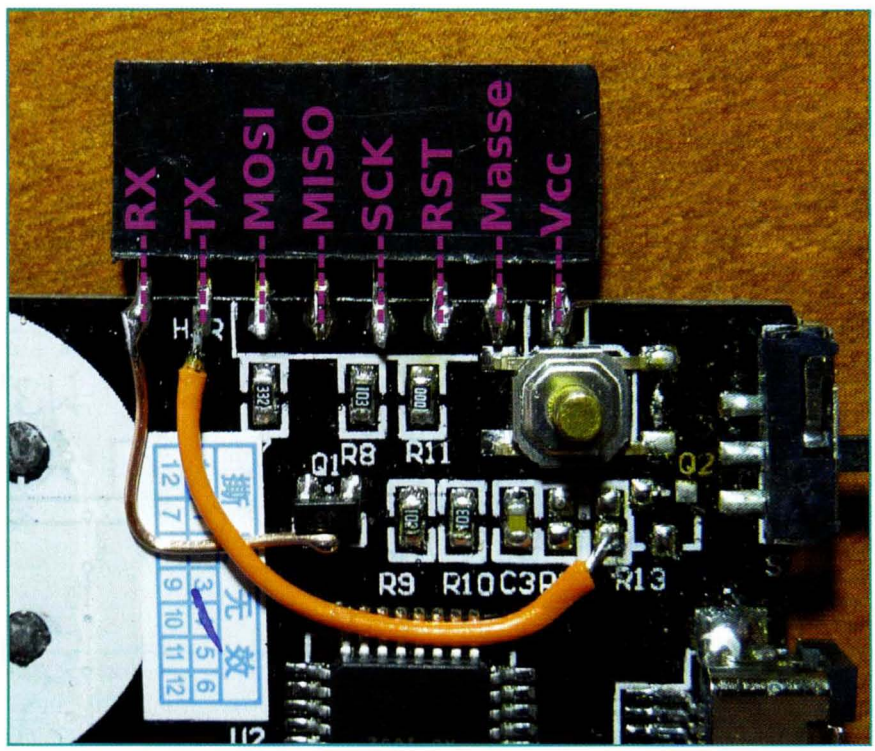


Figure 1

Le badge, une fois démonté révèle un microcontrôleur Atmel AVR ATmega88PA et une EEPROM. Les boutons poussoirs sont reliés à l'entrée ADC de l'AVR via une série de résistances. Le firwmare original utilise donc une valeur de tension mesurée pour déterminer le bouton enfoncé. Mais ce qui nous intéresse ici n'est pas tant le code d'origine mais plutôt celui de Charles. En effet, celui-ci étend les fonctionnalités du badge et le transforme en afficheur asservi par une liaison série.

Afin de reprogrammer l'AVR intégré, il faut accéder aux lignes ISP. Heureusement, le circuit du badge dispose d'une série d'emplacements référencés « HJR » à proximité du bouton ESC/Menu/Font proposant respectivement MOSI, MISO, SCK, RST, masse et Vcc. Après retrait du connecteur pour la pile, on en profite donc pour souder un connecteur 8 broches au pas de 2.54mm. Les deux broches supplémentaires seront destinées aux signaux RX et TX. Ceux-ci sont disponibles respectivement à la base du transistor Q1 et au bas de l'emplacement de la résistance R13 (non présente). Nous arrivons donc à quelque chose comme ceci : Figure 1.

L'ensemble reste relativement propre, même s'il est possible de mieux faire si l'on prévoit, par exemple, une intégration dans le boîtier d'un PC : Figure 2.

Nous utilisons ici un programmeur ISP Stange DX-ISP acheté sur eBay pour quelques euros. Celui-ci permet de choisir, via deux micro-switchs, si le programmeur fournit ou non Vcc et avec une tension d'alimentation de +5V ou +3,3V. Ne prenons pas de risque, certaines lignes de leds étant connectées au même port que ceux utilisés pour la programmation ISP, nous choisissons d'utiliser 3,3V, au plus proche des +3V délivrés par la pile bouton CR2032 d'origine. Autant avec des leds classiques rouges il est possible de prendre quelques largesses, autant avec des composants comme des leds CMS/SMD, mieux vaut éviter de stresser les composants.

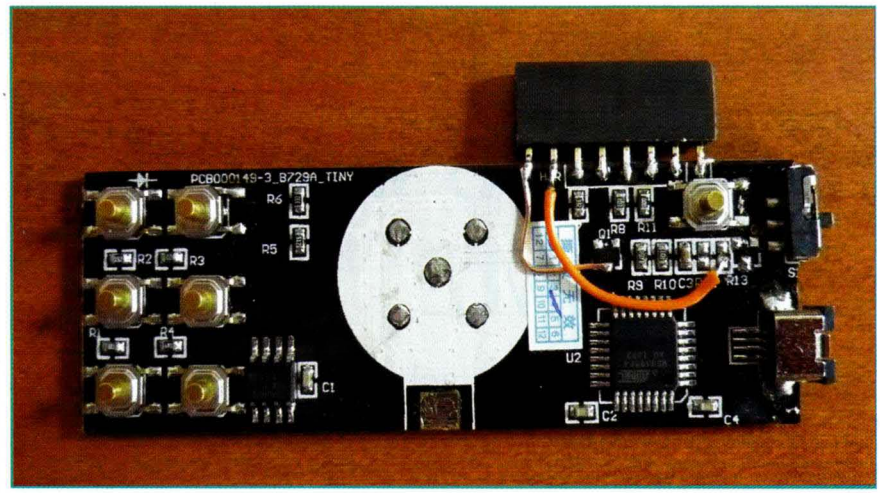


Figure 2

Une fois la connexion établie avec le programmeur, un problème étonnant survient : la version d'Avrdude, la 5.10 installée sous forme de paquet par la Debian Stable, ne reconnaît pas l'ATmega88PA mais uniquement l'ATmega88. Heureusement, la configuration d'Avrdude est adaptable à souhait, aussi bien pour la configuration des programmeurs que pour la reconnaissance des microcontrôleurs. Il nous suffit donc, sans perturber le système de paquets Debian, de créer un **`.avrduderc`** dans notre répertoire personnel et d'y inclure les informations concernant ce composant. Pour cela, nous copions simplement la section dédiée à l'ATmega88 de notre **`/etc/avrdude.conf`** dans le nouveau fichier avant d'y faire quelques modifications minimes :

```
part
    id                  = "m88pa";
    desc                = "ATMEGA88PA";
     has_debugwire = yes;
     flash_instr   = 0xB6, 0x01, 0x11;
     eeprom_instr  = 0xBD, 0xF2, 0xBD, 0xE1, 0xBB, 0xCF, 0xB4, 0x00,
                     0xBE, 0x01, 0xB6, 0x01, 0xBC, 0x00, 0xBB, 0xBF,
                     0x99, 0xF9, 0xBB, 0xAF;
    stk500_devcode   = 0x73;
    signature        = 0x1e 0x93 0x0f;
    pagel            = 0xd7;
    bs2              = 0xc2;
    chip_erase_delay = 9000;
[...]
```

Les champs modifiés sont **`id`** et **`desc`** respectivement pour l'identifiant du composant et sa description, ainsi que **`signature`** en passant le **`0x0a`** d'origine à **`0x0f`** faisant ainsi correspondre la signature à celle de l'ATmega88PA. Il nous suffira alors de spécifier **`-p m88pa`** sur la ligne de commandes de **`avrdude`** (dans le **`Makefile`**) pour que tout rentre dans l'ordre. L'ATmega88PA est une version optimisée et améliorée de l'ATmega88(P). Les différences se résument principalement à une réduction de la consommation de courant (voir *application note* Atmel AVR528).

Ceci fait, il ne nous reste plus qu'à récupérer les sources du firmware de Charles sur GitHub (https://github.com/hugokernel/203LedMatrix) et les compiler avant de les charger dans l'AVR. La première tentative de connexion via un adaptateur série TTL USB, à 9600 bps, est un échec. Après consultation du datasheet de

l'ATmega88PA, on décide alors de définir manuellement la vitesse du port série dans **usart.h** via :

```
#define BAUD_PRESCALE 12
```

Cette valeur correspond, page 200 du datasheet, à une communication en 4800 bps pour un AVR cadencé à 1Mhz (oscillateur interne à 8Mhz avec diviseur de 8 activé par défaut dans les *fuses*), avec un pourcentage d'erreur de 0,2 tout à fait acceptable. En jetant un œil plus en détail dans les sources, la valeur calculée semble être 51, soit effectivement 9600 bps mais avec une horloge à 8Mhz et donc sans diviseur. Ceci suppose donc que les fuses de l'AVR aient été modifiés, et en particulier le bit 7 de lfuse (CKDIV8) passant ainsi de **0x62** à **0xe2**. Personnellement, dans un premier temps, je préfère ne pas trop tripoter les fuses. Une erreur idiote et c'est le retrait de l'AVR du circuit et de pénibles manipulations avec le programmeur AVR Dragon d'Atmel.

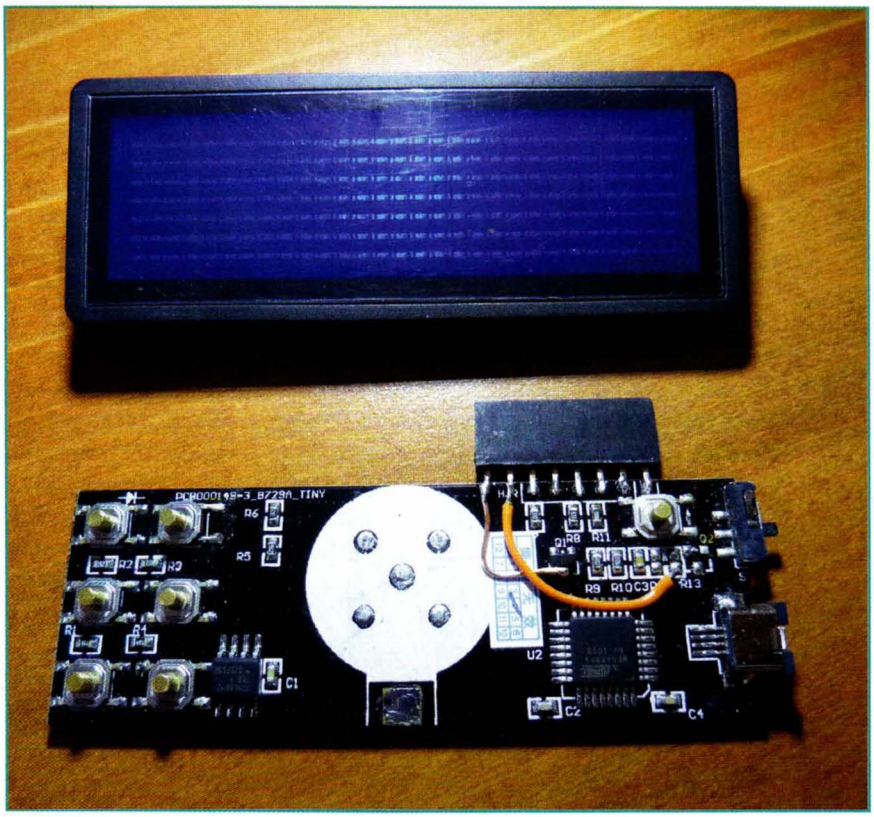


Figure 3 : Vue d'ensemble du badge, en haut la version d'origine et en bas le circuit modifié

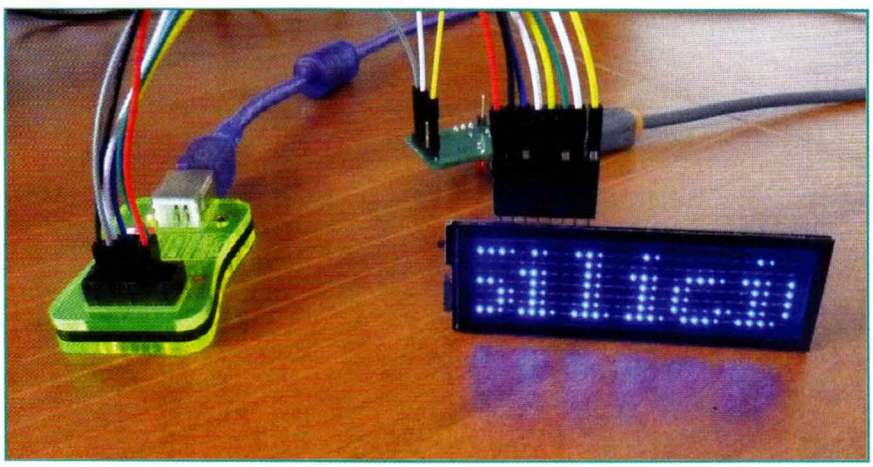


Figure 4 : L'assemblage complet du hack avec à gauche le programmeur ISP, à droite le badge affichant un texte envoyé via le port série et à l'arrière l'adaptateur série TTF vers USB.

Cette correction dans les sources apportée, on peut reprogrammer l'AVR et enfin accéder au menu via une console série :

```
M:[1], S:[9], D:[1], L:[0], C:[0]
Action :
[0...3] Select msg
[t] Edit msg
[c] Clear
[s] Set scroll speed...
[l] Letter spacing...
[d] Reverse direction
[a] Chain mode
[w] Save conf to EEPROM
[b] Read button...

Your choice :
```

Nous voici à même de contrôler l'afficheur matriciel et... ça marche ! Voici un beau hack comme on aimerait en voir plus souvent. Nous ne nous étendrons pas davantage sur le sujet. Charles documente largement sa création ainsi que le fonctionnement de certaines parties du badge. On remarquera toutefois un certain nombre d'avertissements à la compilation qui nécessitent quelques modifications des sources (qualificateurs, déclarations implicites, *header* obsolète), mais ce n'est là rien de grave en comparaison du travail effectué.

Concluons ce court article en saluant la performance du travail de Charles Rincheval et en le remerciant chaleureusement tant il est agréable de pouvoir reposer sur le travail d'autrui pour ses propres créations et sur la lecture d'un code source pour la compréhension du fonctionnement d'un matériel. Dommage que le fabricant du badge en question n'ait pas compris cela... ■

TOSHIBA PLACES STB1F :

PAS CHER, VERROUILLÉ, HONTEUX ET ANTI-OPENSOURCE !

par Denis Bodor

Certains constructeurs ont tout compris : l'intérêt de l'open source, la puissance de développement d'une communauté de développeurs, la sécurité par design et non par dissimulation, etc. D'autres en revanche n'ont simplement rien compris et ne méritent pas le moindre égard. Sous le feu des projecteurs et dans une grande démonstration de ce qu'il ne faut pas faire, voici le STB1F.

Pour la petite histoire, c'est avec une joie immense que je suis rentré chez moi, un appareil qui avait tout pour plaire sous le bras. L'adaptateur Internet TV multimédia Toshiba intègre en effet :

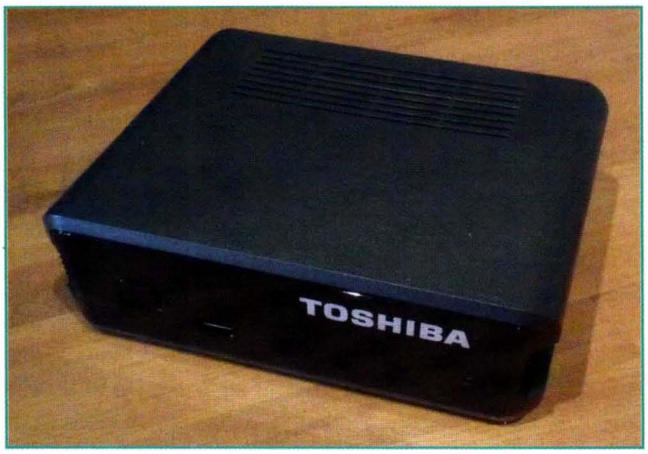
- Sortie HDMI
- Ethernet
- Wifi
- Tuner TNT
- Port USB
- Sortie audio optique
- Télécommande à fréquence (mais capteur IR en façade du boîtier)

Les conjectures furent vite posées, il ne pouvait s'agir que d'un matériel « costaud » fonctionnant sous GNU/Linux et pouvant devenir une myriade de choses entre mes mains. C'est dans un Leclerc que j'ai trouvé la bête, en haut d'une pile de ses congénères arborant un 49,90 € barré et remplacé par un 19,90 €. Le passage en caisse fut rapide et une nouvelle surprise, le périphérique fut facturé 14,90 €. La tête pleine de certitudes, je pensais déjà à mon article à venir, vantant les mérites d'un système embarqué à moins de 15 €, idéal pour s'initier et planifiant déjà d'en acquérir d'autres tant l'occasion était rare.

Le fantasme prit de l'ampleur lors du déballage de l'appareil. Bloc d'alimentation, câble HDMI fourni. Très bien. Quelques secondes plus tard, c'est le tournevis en main que je perçais l'étiquette de garantie dissimulant la première vis. Après retrait des trois autres, placées sous les pieds souples, je découvrais les entrailles de la créature. Trois ports USB internes entouraient le processeur caché par un radiateur passif et quelques puces de RAM et de flash. Deux de ces ports USB standards étaient équipés respectivement d'une clef USB de 4Go format ext3 et d'un adaptateur Wigi Ralink rt2800usb. Sous le circuit un connecteur de 4 broches.

Après quelques mesures au testeur, je trouvais la masse et le +3.3V. Il ne pouvait s'agir que d'une console série. Via un adaptateur USB série/TTL, indispensable compagnon, et une connexion rapide à 115200, je constatais :

```
39idxfs60edeb0081c111289cd2e209935deb171780c272S

#xos2P55-100 (sfla 128kbytes. subid 0x99/99) [serial#22e02226c13528f679dae176c90ecdc3]
#stepxmb 0xac
#DRAM0 Window  :    0x# (17)
#DRAM1 Window  :    0x# (17)
#step6 @0x0*** zxenv has been customized compared to build ***
xloadsize=30676
xload rc=6
#step22
#ei
Uncompressing Linux... Update caches... Ok, booting the kernel.
```



Et plus rien ! Intéressant. Il est rare de voir ce genre de choses, en particulier lorsque le connecteur est disponible et qu'il ne s'agit pas d'emplacements à souder. Et c'est là le point précis du début de mes déceptions.

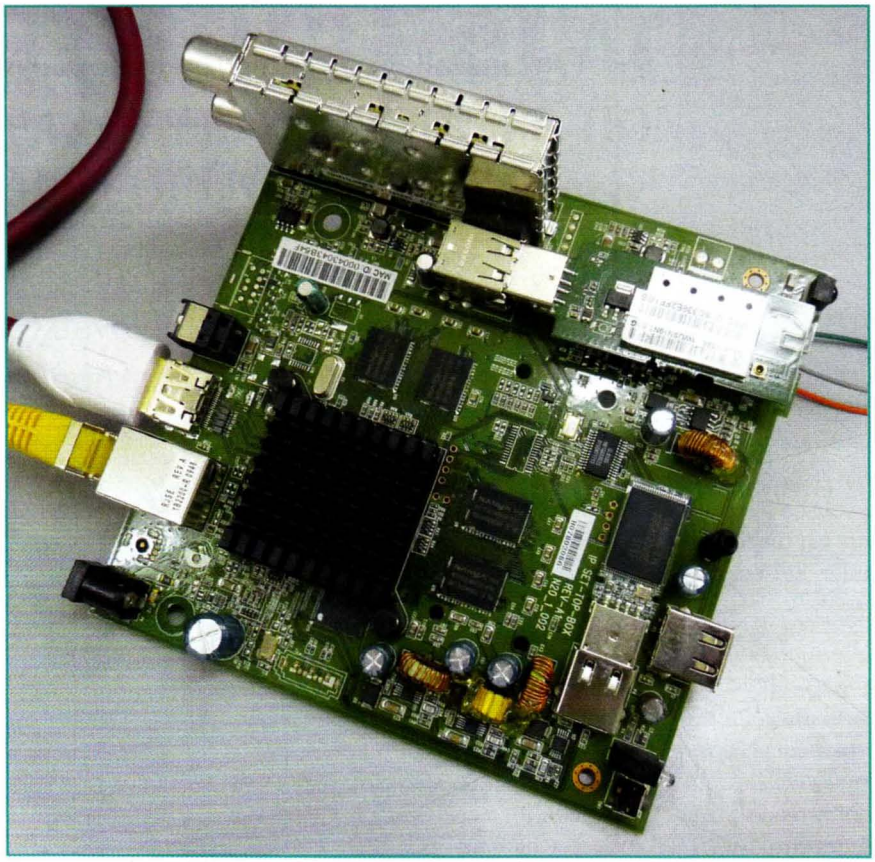
Après une recherche, il s'avéra que le matériel en question n'est pas le fruit du seul travail de Toshiba. Il s'agit en réalité d'une *appliance* de la société Netgem à laquelle Toshiba a ajouté ses composants. Netgem propose ainsi une plateforme pour plusieurs marques et on apprend sur le site officiel (http://www.netgem.com/about-netgem.php) que de grands noms sont au rendez-vous : SFR, N9UF Telecom, Telstra, Toshiba... Par ailleurs (http://www.netgem.com/about-us.php), on peut également lire « An open platform architecture allowing fast deployment of convergent applications » à la ligne des bénéfices du « NetGem Inside ».

Voyons si nous pouvons trouver un autre point d'accès pour bénéficier de cette « open platform architecture ». Après une connexion Ethernet, un scan de port présente un service SSH ouvert mais :

```
% ssh root@192.168.0.102
Permission denied (publickey).
```

SSH demande une authentification SSH2 par clé. Pas de couple login/pass utilisable donc. Une documentation « How to connect to the box using SSH protocol » signé Netgem le confirme « You should be provided RSA public/private keys. They may be used with monitor or partner login ». Deux noms d'utilisateurs sont donc utilisables, mais nous ne disposons pas des clefs associées. C'est une voie sans issue.

Il semble nécessaire d'être un partenaire officiel. Soit. Une autre recherche sur le net nous apprend l'existence de developer.netgem.com présentant le « N5000: An Internet TV Adaptor » comme l'une des plateformes de développement. Un matériel semble-t-il matériellement parfaitement identique au nôtre. Mais point de SDK téléchargeable, il faut signer le « Netgem Developer Program agreement ». Le développement ne porte que sur des modules complémentaires utilisant JavaScript et un langage markup. Il n'existe pas de SDK pour



le développement natif : « the low level tools and libraries are for internal use by Netgem only ». Ah ? Dites messieurs de Netgem, et si les éléments actuellement sous GPL avaient été « for internal use only » du projet GNU, pourriez-vous proposer cette plateforme ?

A propos de code sous GPL, le message de boot est relativement clair, tout comme les PDF publicitaires, il s'agit d'un Linux. Où sont les sources ? Simple, c'est par ici : http://www.netgem.com/support-gpl-linux.php. On trouvera là les fichiers **linux-netboxHD-5.2.16.tar.bz2** et **utils-netgem-netboxHD-5.2.16.tar.bz2**.

Le premier contient les sources du noyau Linux sans autre forme d'éléments. Pas de documentation. Pas de **README**. Pas de configuration. Rien. Il s'agit des bonnes vieilles sources de Linux comme fournies du temps où les fabricants d'appliances se moquaient parfaitement de la notion d'ouverture : « Vous voulez des sources, en voici. Mais n'en attendez pas plus ». Le second fichier n'est pas plus engageant. Trois répertoires **libs**, **netutils** et **sysutils** contiennent les sources de choses comme FFmpeg, LibExif, un serveur DHCP, les Wireless Tools, un pilote HD44780, Busybox... Le tout dans le même état que les sources du noyau. La dernière fois que j'ai vu quelque chose de similaire c'était au temps où Synology proposait des NAS sous Linux sans vraiment appliquer la notion d'ouverture. Fort heureusement, ils ont, à présent, changé cette politique à la limite de la plaisanterie, pour faire de l'open source, du vrai. Les sources GPL de Netgem ne nous apportent donc absolument aucune aide. Il s'agit simplement d'un énorme tas de code, pas même un puzzle pour développeur curieux.

Que faire à partir de là ? Rien. Ne cherchez pas. Bon nombre d'utilisateurs s'y sont cassé les dents et cherchent toujours un moyen d'accéder pleinement au matériel qu'ils ont acheté, parfois bien plus cher que moi. Une page web de Netgem annonce « Currently there are 3 000 000+ Netgem set top boxes in use worldwide » qu'on pourrait extrapoler en « il y a 3 millions de personnes dans le monde qui sont potentiellement frustrées ».

A ceux qui se demandaient s'il était toujours possible d'adapter une appliance, je peux répondre « non ». Oh bien sûr, je pourrais chercher les versions des différents logiciels en place, trouver une faille, manipuler des transactions réseau et peut-être même arriver à utiliser le connecteur JTAG de la carte mère. Mais à quel prix ? Devant tant de barrières et une telle détermination à bloquer toute forme d'ouverture, le jeu en vaut-il la chandelle ? Sans compter l'incompréhension face à de tels comportements. Les personnes souhaitant connaître le fonctionnement du produit ne veulent pas « pirater » ou voler des films. Il existe des méthodes plus simples pour cela. Elle ne veulent pas non plus prendre le contrôle de l'infrastructure DRM et de la gestion des comptes clients. Celles qui souhaitent le faire n'ont pas besoin que les firmwares soient ouverts, demandez à Sony ! Les personnes comme moi, comme vous lecteurs, veulent utiliser pleinement leur produit, le faire vivre, le faire évoluer, tout simplement.

Me voici donc avec un matériel entre les mains qui aurait pu être une source de connaissances et d'expériences importante mais qui, à présent, ne me servira plus qu'à une seule chose : me souvenir que, lorsque j'achèterai un nouveau produit, pour une utilisation standard ou par curiosité, je devrai bien m'assurer qu'il ne s'agit pas d'un produit Netgem. Quant à Toshiba, inutile de préciser que j'aurai de sérieux doutes quant à opter pour un produit de cette marque.

Pour 14,90 € j'ai finalement acheté un adaptateur Wifi Ralink, une clef USB de 4Go et un adaptateur secteur fournissant 12V à 1,5A. Je ne m'en sors pas si mal, avec un goût amer dans la bouche et avec une bonne leçon à retenir : il existe encore des gens qui prennent le milieu de l'open source pour un verger où il suffit de se servir sans vergogne. ■

NOTIFICATEUR DE MAILS USB DREAM CHEEKY SOUS GNU/LINUX

par Denis Bodor

Alors que les périphériques USB génériques sont de plus en plus courants, les grands constructeurs ont tendance à unifier les pilotes. De ce fait, le support pour GNU/Linux et d'autres OS open source s'en trouve facilité. C'est le cas, par exemple, des webcams utilisant presque toutes maintenant le pilote UVC. Cependant, lorsqu'on s'écarte des sentiers battus et qu'on touche à des périphériques s'apparentant plus à des gadgets, le support fait cruellement défaut, tout comme l'éventualité d'une documentation technique détaillant les spécifications précises. Le cas qui nous intéresse aujourd'hui est le notificateur de mails USB de Dream Cheeky.

Vous l'avez peut-être déjà vu, voire vous en possédez un vous-même. Ce notificateur se présente sous la forme d'un petit boîtier translucide en forme d'enveloppe, équipé d'un simple connecteur USB. On le retrouve un peu partout dans les boutiques en ligne de vente de gadgets USB. Son objet est relativement simple : fournir une notification lumineuse multicolore renseignant sur l'état de votre ou vos boîtes mails. Ce périphérique est livré avec une petite application Windows chargée de faire le lien entre votre boîte mails et un signal lumineux. En y regardant de plus près, on constate que, tout naturellement, c'est l'application en question qui fait tout le travail. Le périphérique en lui-même peut être vu comme un simple témoin lumineux RVB asservi par USB.

La connexion à une machine GNU/Linux provoque la détection automatique du périphérique qui est alors vu comme un produit HID (*Human Interface Device*) :

```
usb 4-1: new low speed USB device using uhci_hcd and address 26
usb 4-1: New USB device found, idVendor=1d34, idProduct=0004
usb 4-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 4-1: Product: DL100B Dream Cheeky Generic Controller
usb 4-1: Manufacturer: Dream Link
usb 4-1: configuration #1 chosen from 1 choice
generic-usb 0003:1D34:0004.0006: hiddev96,hidraw0: USB HID v1.10
  Device [Dream Link DL100B Dream Cheeky Generic Controller]
  on usb-0000:00:1a.1-1/input0
```

Bien que le périphérique soit effectivement supporté par le pilote générique de la classe USB HID, normalement destinée aux périphériques comme les claviers, souris, joysticks et

Remarque

Précisons ici que le périphérique en question, bien que se présentant comme le modèle original semble sensiblement différent. En effet, le modèle en notre possession, provenant de Materiel.net ne semble pas intégrer de notification audio comme spécifié sur la page officielle de Dream Cheeky. Plusieurs modèles différents, utilisant le même contrôleur, semblent donc exister. A moins bien entendu que la notification audio soit purement logicielle et donc dépendante du système d'exploitation. Notre modèle possède bien la mention du constructeur moulée dans le plastique, mais le talent des usines chinoises n'en sera pas pour autant mis en doute. On a déjà vu pire pour ne baser notre avis que sur ce genre de choses. Quoi qu'il en soit, l'intérieur du périphérique se résume à une puce noyée dans la résine, un quartz, un condensateur et la connexion USB. Pas de quoi émettre le moindre bruit, étrange...

tantôt les systèmes d'affichage, ce n'est pas pour autant que nous saurons comment l'initialiser et communiquer avec lui. Le standard USB HID décrit le protocole de communication pour ce type de périphériques et non la correspondance entre les données envoyées et les actions à provoquer.

La consultation des données du périphérique USB ne nous donne pas beaucoup d'informations complémentaires :

```
Bus 004 Device 026: ID 1d34:0004
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               1.10
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x1d34
  idProduct          0x0004
  bcdDevice            0.02
  iManufacturer           1 Dream Link
  iProduct                2 DL100B Dream Cheeky Generic Controller
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength               9
    bDescriptorType       2
    wTotalLength         34
    bNumInterfaces        1
    bConfigurationValue   1
    iConfiguration        0
    bmAttributes       0x80
      (Bus Powered)
    MaxPower            500mA
    Interface Descriptor:
      bLength             9
      bDescriptorType     4
      bInterfaceNumber    0
      bAlternateSetting   0
      bNumEndpoints       1
      bInterfaceClass     3 Human Interface Device
      bInterfaceSubClass  0 No Subclass
      bInterfaceProtocol  0 None
      iInterface          0
        HID Device Descriptor:
          bLength             9
          bDescriptorType    33
          bcdHID           1.10
          bCountryCode        0 Not supported
          bNumDescriptors     1
          bDescriptorType    34 Report
          wDescriptorLength  37
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength             7
        bDescriptorType     5
        bEndpointAddress  0x81  EP 1 IN
        bmAttributes        3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize    0x0008  1x 8 bytes
        bInterval           10
Device Status:     0x0000
  (Bus Powered)
```

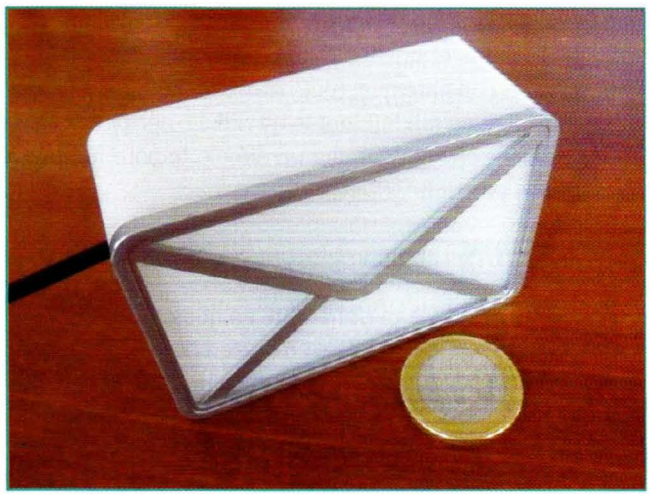
Nous avons là un seul profil de configuration avec une seule interface et un seul *endpoint*.

1 Impasse ? Non.

Ce n'est qu'après une recherche sur le Web via les numéros VendorId et ProductId, qu'on constate que les informations sur ce périphérique sont rares, mais que d'autres personnes s'y sont déjà cassé les dents. C'est généralement l'analyse du support logiciel sous Windows qui permet de déboucher sur quelques données utilisables. En effet, l'analyse du bus USB en lui-même, en raison de sa vitesse de transmission, nécessite un équipement relativement coûteux comme un analyseur logique haut de gamme. Les analyseurs de bus USB pouvant enregistrer les données circulant à quelques 480Mbits/s se trouvent à partir de tarifs de l'ordre de 1000 euros. Bien entendu, pour un analyseur USB 1.0 (1.5Mbits/s et 12Mbits/s) le tarif descendra à quelques 300 euros, mais cela reste un matériel relativement coûteux qu'il n'est pas raisonnablement (et économiquement) possible de réaliser par soi-même.

Une autre solution pour inspecter ce qui circule sur un bus USB, est d'attaquer à la source logicielle et donc au niveau du support USB hôte du système d'exploitation supporté par le logiciel livré avec le périphérique. Il s'agit donc d'utiliser Windows et de collecter les données à ce niveau via un ajout dans la pile de pilote. C'est ce qui semble avoir été fait par les utilisateurs ayant implémenté une solution logicielle pour le notificateur Dream Cheeky.

Deux voies peuvent ainsi être empruntées. La première est utilisée dans un billet sur le blog de Circuits@Home, tiré des expérimentations d'un certain Krulkip. Il détaille ainsi la mise en œuvre du notificateur avec le shield Arduino USB Host Shield 2.0 de Circuits@Home. Ce shield permet d'ajouter un support USB hôte à un module Arduino et repose sur le très classique Maxim MAX3421E utilisé sur la totalité des



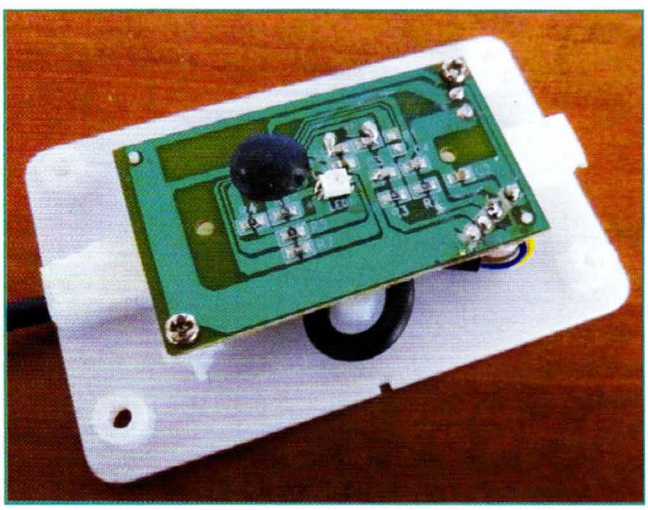
Le notificateur de mails et de messagerie instantanée Dream Cheeky se présente sous la forme d'une boîtier translucide équipé d'un connecteur USB.

cartes permettant d'implémenter un périphérique ADK pour Google Android. Cette implémentation Arduino utilise les fonctionnalités HID du périphérique.

Une autre solution est de se passer du support HID et de dialoguer directement avec le périphérique via des requêtes de contrôle USB. C'est la solution choisie par Pierre Ficheux pour la réalisation d'un pilote Linux (noyau donc) pour le notificateur. Servant plus de base d'apprentissage pour le développement de pilote Linux, son implémentation nécessite la déconnexion préalable du périphérique du support HID du noyau puis l'insertion du module. On obtient alors une entrée **/sys/bus/usb/drivers/uwn/*/color** qu'il suffit d'utiliser à l'aide d'un simple **echo -n "r v b" >** où **r**, **v** et **b** sont respectivement les valeurs de rouge, vert et bleu entre 0 et 255.

2 Ré-implémentons

Pourquoi ? Maintenir un pilote Linux, même aussi simple que celui-ci, est parfois pénible. De plus, il existe des solutions depuis l'espace utilisateur, permettant de contrôler un périphérique comme ce notificateur. C'est la libUSB. Un programme utilisateur dialoguant directement avec le périphérique repose sur une couche d'abstraction constituée de la libUSB et sera donc indépendant de toutes mises à jour du noyau. En effet, même si la présence d'une entrée **/sys** offre des possibilités d'utilisation intéressantes, une intégration dans le système passera obligatoirement par l'utilisation d'un mécanisme de mise à jour piloté par la distribution utilisée. Alors qu'un simple programme utilisateur n'aura besoin que d'une dépendance vers la libUSB, et la création d'un paquet relativement classique pour une distribution donnée, un pilote noyau demande bien plus de travail et de maintenance à long terme. Enfin, une implémentation reposant uniquement sur la libUSB permettra un portage simplifié vers d'autres systèmes disposant de leur version de la bibliothèque (FreeBSD, NetBSD, OpenBSD, OpenSolaris (+forks), Darwin, Mac OS X et même Windows).



L'intérieur du matériel est relativement simple et comprend un circuit avec quelques composants et une IC noyée dans la résine.

Le protocole utilisé dans le pilote de Pierre est relativement simple, puisqu'il se résume à l'envoi de trois suites d'initialisation de 8 octets, puis à des messages reprenant les valeurs de rouge, de vert et de bleu.

L'implémentation sera donc fort simple grâce aux facilités fournies par la libUSB 1.0.

```
int main(int argc, char **argv) {
  libusb_context *ctx = NULL;
  struct libusb_device_handle *devhaccess;
  int i;

  unsigned char buf[8];
```

ctx est notre contexte de manipulation de la libUSB qui est utilisé pour toutes les opérations génériques sur le bus. **devhaccess** est un *handler* permettant un accès direct au périphérique USB à l'instar des opérations sur les fichiers. **i** est une simple variable d'incrémentation utilisée par la suite et **buf[8]** est notre *buffer* de commandes à destination du notificateur. Pour accéder au bus, nous commençons par initialiser le contexte. Ceci ne nécessite pas de permissions particulières, c'est l'accès au périphérique qui est autorisé ou non en fonction de la configuration udev et des permissions de l'utilisateur.

```
  if(libusb_init(&ctx) != 0) {
    fprintf(stderr,"libusb_init Error : %s (%d)\n",
        strerror(errno),errno);
    exit(EXIT_FAILURE);
  }

  libusb_set_debug(ctx, 3);
```

Cette dernière ligne nous permet de préciser le niveau de verbosité des messages de la libUSB. Ici **3** sera une valeur normale mais, en phase de développement de l'utilitaire, nous préférerons 4. A présent, le contexte est installé et nous pouvons utiliser notre première fonction de la libUSB pour obtenir un accès au périphérique de notre choix :

```
  if ((devhaccess = libusb_open_device_with_vid_pid( ctx, VID, PID)) == 0) {
    fprintf(stderr,"libusb_open_device_with_vid_pid Error : %s (%d)\n",
        strerror(errno),errno);
    printf("no device. Giving up !\n");
    libusb_exit(ctx);
    exit(EXIT_FAILURE);
  }
```

Comme vous pouvez le constater, nous utilisons les identifiants de constructeur et de produit définis par ailleurs :

```
#define VID 0x1d34
#define PID 0x0004
```

libusb_open_device_with_vid_pid facilite grandement les choses et nous évite de scanner le bus à la recherche de notre périphérique. Notez cependant que seul le premier matériel correspondant sera utilisé et si vous avez connecté plusieurs modèles identiques à votre machine, vous devrez alors parcourir tout le bus (arbre) pour trouver celui qui vous intéresse. Ceci, dans le cas du notificateur, semble problématique car le numéro de série est ici **0**. Il est donc fort probable que rien ne permette de différencier deux périphériques de ce type. Nous n'avons malheureusement pas pu tester cette affirmation, ne disposant que d'un seul exemplaire.

Nous avons un accès au périphérique via **devhaccess** mais ne pouvons pas encore l'utiliser. En effet, le protocole USB ne permet pas des accès simultanés au même périphérique. Il nous faut en réclamer l'utilisation, bloquant ainsi toutes tentatives concurrentes. Cependant, il est possible que notre périphérique soit déjà utilisé, en particulier par le noyau. Nous devons alors tester si un pilote prend actuellement en charge le matériel et, au besoin, demander au noyau de détacher le support logiciel en question :

```
if(libusb_kernel_driver_active(devhaccess, 0) == 1) {
  printf("kernel driver is active on interface... trying to detach\n");
  if(libusb_detach_kernel_driver(devhaccess, 0)) {
    fprintf(stderr,"Unable to detach kernel driver... giving up !\n");
    giveup(devhaccess,ctx);
  }
  printf("Kernel driver detached. Ready to claim device\n");
}
```

Attention, ceci n'est possible que pour un support noyau. Si un autre programme de l'espace utilisateur a accès au périphérique (l'a déjà réclamé), ces quelques lignes de codes ne régleront pas le problème. Ceci fait, nous pourrons réclamer nous-même l'accès au matériel :

```
printf("Claiming interface 0\n");
if(libusb_claim_interface(devhaccess, 0) != 0) {
  fprintf(stderr,"libusb_claim_interface  Error : %s (%d)\n",
      strerror(errno),errno);
  giveup(devhaccess,ctx);
}
printf("All the interface are belong to me\n");
```

Comme vous pouvez le constater, nous testons la valeur retournée par la fonction **libusb_claim_interface()** car l'opération peut échouer (programme concurrent). Pour simplifier la gestion d'erreurs, si le programme prend de l'ampleur par la suite, nous utilisons une fonction fermant le handler et quittant le contexte libUSB avant de terminer le programme :

```
void giveup(libusb_device_handle *devh, libusb_context *ctx) {
  libusb_close(devh);
  libusb_exit(ctx);
  fprintf(stderr,"Fatal error. Sorry.\n");
  exit(EXIT_FAILURE);
}
```

Après l'appel sans erreur de la dernière fonction, nous avons la main mise sur le périphérique et nous pouvons alors nous pencher sur le protocole lui étant spécifique. D'après les données contenues dans le pilote de Pierre, nous devons envoyer :

```
memset(buf, 0, sizeof(buf));
buf[0] = 0x1f;
buf[1] = 0x02;
buf[3] = 0x2e;
buf[6] = 0x2b;
buf[7] = 0x03;
printf("Sending init1\n");
if(libusb_control_transfer(
      devhaccess,
      LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_RECIPIENT_INTERFACE,
      0x09, 0x200, 0, buf, 8, 1000 ) < 0 )
  fprintf(stderr,"usb_control_msg Error : %s (%d)\n",
      strerror(errno),errno);
```

La fonction **libusb_control_transfer**, comme son nom l'indique, permet d'envoyer une requête de contrôle. Les paramètres sont, dans l'ordre, le handler du périphérique, le type de requête, le champ de requête à utiliser, la valeur pour ce champ, un index, les données à envoyer, la taille des données et un temps limite. Ces valeurs sont celles directement issues du pilote de Pierre et adaptées aux arguments des fonctions de la libUSB.

La première chaîne d'initialisation est **{0x1f,0x02,0x00,0x2e,0x00,0x00,0x2b,0x03}** et les suivantes sont **{0x00,0x02,0x00,0x2e,0x00,0x00,0x2b,0x04}** et **{0x00,0x00,0x00,0x00,0x00,0x00,0x2b,0x05}**. A noter que cette dernière correspond à une extinction des trois leds (rouge, vert et bleu). Pour ces deux dernières chaînes d'initialisation, nous réutilisons tout simplement un code identique au précédent en jonglant avec **memset()** et des initialisations. Ceci fait et si nous n'avons pas détecté d'erreur, nous pouvons à loisir changer les valeurs pour chaque couleur. Pour ce faire, nous utilisons une fonction dédiée :

```
int sendRGB(libusb_device_handle *devhaccess,
    unsigned char r, unsigned char g, unsigned char b) {
  unsigned char buf[8];
  memset(buf, 0, sizeof(buf));
  buf[6] = 0x2b;
  buf[7] = 0x05;

  buf[0] = r;
  buf[1] = g;
  buf[2] = b;

  if(libusb_control_transfer(
        devhaccess,
        LIBUSB_REQUEST_TYPE_CLASS | LIBUSB_RECIPIENT_INTERFACE,
        0x09, 0x200, 0, buf, 8, 1000 ) < 0 ) {
    fprintf(stderr,"usb_control_msg Error : %s (%d)\
n",strerror(errno),errno);
    return 1;
  } else {
    return 0;
  }
}
```

Comme le montre l'initialisation de **buf[8]**, les trois premières positions correspondent aux valeurs RVB (entre 0 et 63), les positions 6 et 7 sont systématiquement **0x2b** et **0x05** pour une commande de la luminosité des leds. Jouer avec les couleurs devient très facile, exemple :

```
for(i=63;i>0;i--) {
  sendRGB(devhaccess, i, 0, 0); usleep(2500);
}
printf("0\n"); sendRGB(devhaccess, 0, 0, 0);
```

Et voici un petit fondu au noir depuis la valeur maximum de rouge. Notez que l'utilisation d'une valeur plus importante que 63 donnera exactement le même résultat que cette dernière. Après nous être amusé avec toutes les couleurs dans tous les sens, il ne nous reste plus qu'à terminer le programme correctement :

```
  printf("Release interface...\n");
  if(libusb_release_interface(devhaccess,0) !=0)
    fprintf(stderr,"libusb_release_interface  Error : %s (%d)\n",
        strerror(errno),errno);

  printf("Closing interface...\n");
  libusb_close(devhaccess);

  printf("LibUSB exit...\n");
  libusb_exit(ctx);

  return(EXIT_SUCCESS);
}
```

On rend la main sur le périphérique et on quitte le contexte libUSB avant de terminer. Pour compiler notre petit outil, un **Makefile** fera l'affaire :

```
TARGET  := main
WARN    := -Wall
CFLAGS  := -O2 ${WARN} `pkg-config libusb-1.0 --cflags`
LDFLAGS := `pkg-config libusb-1.0 --libs`
CC      := gcc

all: ${TARGET}

${TARGET}.o: ${TARGET}.c
${TARGET}: ${TARGET}.o

clean:
        rm -rf ${TARGET}.o ${TARGET}
```

Notez l'utilisation de **pkg-config** pour obtenir les arguments adéquats pour le compilateur et pour l'éditeur de liens. Nous enchaînons directement sur la compilation avec un simple :

```
% make
gcc -O2 -Wall  `pkg-config libusb-1.0 --cflags`   -c -o main.o main.c
gcc `pkg-config libusb-1.0 --libs`  main.o   -o main
```

L'exécution de notre code aura pour effet d'allumer le notificateur en rouge et de baisser progressivement mais rapidement l'intensité jusqu'à l'éteindre complètement.

3 Version HID

Alors que l'accès au périphérique sans support HID via la libUSB est relativement aisé, il peut être nécessaire parfois de ne pas s'encombrer de bibliothèques supplémentaires. Dans cet ordre d'idées, la libUSB cède la place au support de périphériques HID intégré au noyau Linux. Nous l'avons vu, lorsque nous connectons le notificateur à la machine, celui-ci est directement pris en charge. Une entrée est alors ajoutée dans **/dev**, plus exactement nous y trouvons **/dev/usb/hiddev0**. Qui dit entrée dans **/dev** dit forcément méthode d'accès depuis l'espace utilisateur. Il existe des bibliothèques permettant la communication avec les périphériques HID mais :

- Nous avons dit « pas de bibliothèque ».
- Aucune ne semble exister sous la forme d'un paquet pour la distribution Debian stable utilisée ici.
- Toutes les implémentations semblent utiliser la libUSB en version 0.1, alors qu'il est fortement déconseillé de préférer cette version à la 1.0, bien que largement répandue.

Ce serait fort dommage de gâcher notre plaisir, puisque le noyau met à notre disposition un certain nombre d'ioctl nous permettant de communiquer relativement facilement avec les périphériques HID simplement avec la libc. Mais entrons dans le vif du sujet sans plus attendre. Après avoir inclus les fichiers d'en-tête courants nous ajoutons :

```
#include <linux/hiddev.h>

#define VID 0x1d34
#define PID 0x0004
```

Nous ouvrons ensuite l'entrée **/dev** tout à fait normalement :

```
int fd;

if ((fd=open("/dev/usb/hiddev0", O_RDONLY)) < 0) {
  fprintf(stderr,"Open Error : %s (%d)\n",
      strerror(errno),errno);
  exit(EXIT_FAILURE);
}
```

Le handle **fd** ainsi obtenu sur le fichier nous permet de tenter notre premier ioctl, **HIDIOCGDEVINFO**, permettant de remplir une structure **hiddev_devinfo** qui contiendra, comme son nom l'indique, diverses informations sur le périphérique :

```
struct hiddev_devinfo dinfo;

if (ioctl(fd, HIDIOCGDEVINFO, &dinfo) < 0) {
  fprintf(stderr,"Get device info Error : %s (%d)\n",
      strerror(errno),errno);
  close(fd);
  exit(EXIT_FAILURE);
}
```

dinfo contient maintenant tout ce qu'il nous faut pour vérifier que l'entrée **/dev** choisie correspond bien au périphérique que nous souhaitons utiliser. Nous faisons cela en vérifiant les identifiants de vendeur et de produit qui sont remontés dans la structure **hiddev_devinfo** :

```
if (dinfo.vendor != (short)VID || dinfo.product != (short)PID) {
  fprintf(stderr,"Sorry, not my device\n");
  close(fd);
  exit(EXIT_FAILURE);
}
printf("Got my device\n");
```

Dès lors, si le programme n'a pas terminé son exécution, nous pouvons vraisemblablement supposer que c'est bien là notre notificateur ou un clone. Pour nous en assurer sommairement, nous récupérons sa chaîne d'identification avec un autre ioctl :

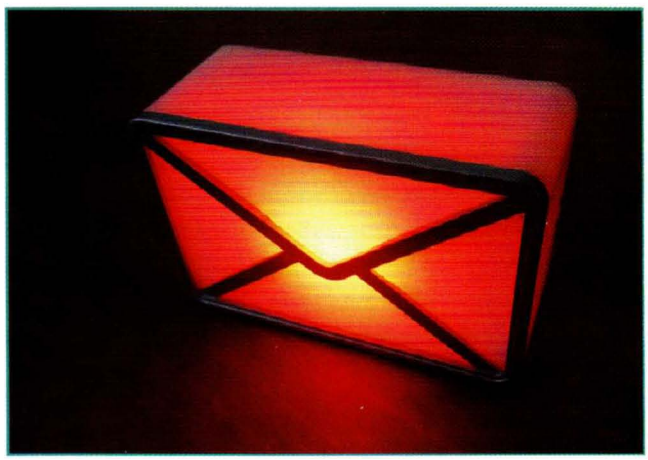
```
unsigned char mystring[256];

memset(mystring, 0, sizeof(mystring));

if (ioctl(fd, HIDIOCGNAME(256), mystring) < 0) {
  fprintf(stderr,"Get device info Error : %s (%d)\n",
      strerror(errno),errno);
  close(fd);
  exit(EXIT_FAILURE);
}
printf("Device string : %s\n", mystring);
```

256 caractères devraient être suffisants pour stocker ces informations et les afficher. Dans le cas présent, nous obtenons effectivement la chaîne : **"Dream Link DL100B Dream Cheeky Generic Controller"**. A ce stade, nous pouvons commencer les réjouissances en communiquant avec le périphérique HID. Ceci se fait par l'intermédiaire de rapports. Le protocole HID est relativement complet et complexe, car il prend en charge une vaste gamme de périphériques très différents comme des claviers, des souris, mais également des lecteurs d'empreintes digitales et des casques/micro (headset). Il existe ainsi tout une notion de classes de périphériques et de descripteurs permettant de connaître le format dans lequel communiquer avec un matériel HID. Nous ne rentrerons pas dans le détail ici, car ce n'est pas l'objet de l'article. Sachez cependant que si vous comptez vous frotter aux périphériques HID, il va vous falloir assimiler une importante quantité d'informations sur l'architecture du protocole. Un bon point de départ est la page de Supelec Rennes sur le sujet : http://www.rennes.supelec.fr/ren/fi/elec/docs/usb/hid.html. Ensuite, vous passerez par les incontournables spécifications officielles, « Device Class Definition for Human Interface Devices (HID) Firmware Specification », une centaine de pages seulement, mais relativement denses (et indigestes).

En ce qui nous concerne, tout ce que nous avons à faire est de remplir un rapport et de l'envoyer au périphérique pour qu'il réponde à nos commandes. Pour ce faire, nous composons une fonction utilitaire se chargeant de faire tout le travail :

```
void send_report(int fd, int id, unsigned char *buf, int n) {
  struct hiddev_usage_ref_multi uref;
  struct hiddev_report_info rinfo;
  int i;

  uref.uref.report_type = HID_REPORT_TYPE_OUTPUT;
  uref.uref.report_id = id;
  uref.uref.field_index = 0;
  uref.uref.usage_index = 0;
  uref.num_values = n;

  for (i = 0; i < n; i++)
    uref.values[i] = buf[i];

  if (ioctl(fd, HIDIOCSUSAGES, &uref) < 0) {
    fprintf(stderr,"HIDIOCSUSAGE Error : %s (%d)\n",
        strerror(errno),errno);
    close(fd);
    exit(EXIT_FAILURE);
  }

  rinfo.report_type = HID_REPORT_TYPE_OUTPUT;
  rinfo.report_id = id;
  rinfo.num_fields = 1;

  if (ioctl(fd, HIDIOCSREPORT, &rinfo) < 0) {
    fprintf(stderr,"HIDIOCSREPORT Error : %s (%d)\n",
        strerror(errno),errno);
    close(fd);
    exit(EXIT_FAILURE);
  }
}
```

L'opération se passe en deux parties comme précisé dans le **Documentation/usb/hiddev.txt** livré avec les sources du noyau Linux. Premièrement, nous remplissons une structure **hiddev_usage_ref_multi**. Les points importants sont **num_values** devant être renseigné avec la taille de la chaîne envoyée et, bien entendu, le tableau **values[]** correspondant à la chaîne elle-même. Nous validons ce premier mouvement avec l'ioctl **HIDIOCSUSAGES**. Nous pouvons ensuite passer au rapport avec **HIDIOCSREPORT** qui suit le remplissage d'une



Moment d'intense satisfaction à l'allumage du notificateur déclenché par une application utilisateur faite-maison sous GNU/Linux.

structure **hiddev_report_info** contenant les informations sur le rapport en question et décrivant ce que nous envoyons. Notez la correspondance entre les deux structures et en particulier l'utilisation de **id**.

Notre fonction prendra donc en argument le handle sur le fichier **/dev**, l'ID du rapport, la chaîne à envoyer et sa taille. On retrouve presque les mêmes arguments que pour la libUSB utilisée précédemment. Il ne nous reste plus qu'à l'utiliser pour envoyer, tout d'abord, les deux chaînes d'initialisation au périphérique :

```
unsigned char init1[]  = { 0x1F, 0x01, 0x25, 0x00, 0xC4, 0x00, 0x25, 0x03 };
unsigned char init2[]  = { 0x00, 0x01, 0x25, 0x00, 0xC4, 0x00, 0x25, 0x04 };

send_report(fd, 0x00, init1, 8);
send_report(fd, 0x00, init2, 8);
```

Celles-ci proviennent du blog de Circuits@Home intitulé « Driving the Cheeky Mail Notifier from Arduino ». Ensuite, nous n'avons qu'à envoyer les couleurs exactement comme dans la méthode précédente :

```
unsigned char msg1[]  = { 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x25, 0x05 };
unsigned char msg2[]  = { 0x00, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x25, 0x05 };
unsigned char msg3[]  = { 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x25, 0x05 };
unsigned char msg4[]  = { 0x00, 0x00, 0xFF, 0x00, 0x00, 0x00, 0x25, 0x05 };
unsigned char msg5[]  = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x25, 0x05 };

/* couleur */
send_report(fd, 0x00, msg1, 8);
sleep(1);
send_report(fd, 0x00, msg2, 8);
sleep(1);
send_report(fd, 0x00, msg3, 8);
sleep(1);
send_report(fd, 0x00, msg4, 8);
sleep(1);
send_report(fd, 0x00, msg5, 8);

close(fd);
return(EXIT_SUCCESS);
}
```

Vous voici à présent en mesure de piloter le notificateur sans la moindre bibliothèque. Les deux méthodes ont leurs avantages et leurs inconvénients qu'il faudra bien évaluer et pondérer en fonction de l'objectif et de la plateforme que vous visez. Mais, quelque soit la façon dont vous accéderez au matériel, voyons comment tirer un peu plus que quelques changements de couleur de cette petite boîte translucide...

4 Plage de couleurs

Pouvoir contrôler les valeurs des trois composantes de couleur est une chose, mais vous conviendrez avec moi que ce n'est pas ce qu'il y a de plus pratique. Notez que ce qui va suivre n'est en rien spécifique à l'objet de cet article et peut être adapté facilement à un code pour microcontrôleur. Il faudra simplement ajuster la plage d'intensités, 0-63 étant relativement rare. Dans un premier temps, nous pouvons composer une palette de couleurs personnalisée qu'il suffira d'utiliser. Pour cela, nous cherchons en tâtonnant quelques couleurs classiques de base. Via l'utilisation des arguments en ligne de commandes pour notre outil, ainsi que les valeurs de rouge, vert et de bleu présentées, par exemple, par GIMP, nous pouvons obtenir ceci :

- Rouge = 63, 0, 0
- Orange = 63,15,0
- Jaune = 63,40,0
- Vert = 0,63,0
- Cyan = 0,63.15
- Mauve = 40,0,63
- Violet = 63,0,63
- Blanc = 63,63,55

Une implémentation possible et sans doute la plus simpliste consiste à stocker ces 24 valeurs dans un tableau auquel nous accéderons via un multiple de 3 (0, 3, 6, 9, 12, 15, 18 et 21). Nous avons ainsi accès à la valeur pour rouge et, implicitement à vert et bleu, à un et deux pas plus loin. Nous créons donc :

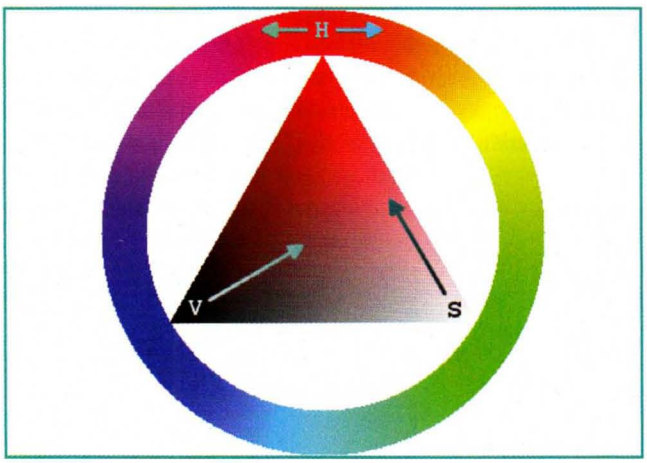
```
int palette[24]={63,0,0,
                 63,15,0,
                 63,40,0,
                 0,63,0,
                 0,63,15,
                 40,0,63,
                 63,0,63,
                 63,63,55};
```

Pour expérimenter notre implémentation, il nous suffit d'une petite boucle et d'une sélection aléatoire de 25 changements de couleur qui se suivront espacés d'un 25ème de seconde :

```
#define HIGH 7
#define LOW  0

  srand ((unsigned int)time(NULL));

  for(j=0;j<25;j++) {
    i = (rand() % (HIGH - LOW + 1) + LOW)*3;
    printf("Sending : %u\n", i);
    printf("Sending : %u, %u, %u\n", palette[i],
palette[i+1],  palette[i+2]);
    sendRGB(devhaccess,  palette[i], palette[i+1],
palette[i+2]);
    usleep(1000000/25);
  }
```

Afin d'obtenir notre choix aléatoire entre 0 et 7, nous utilisons une solution classique utilisant un modulo et les fonctions **srand()** et **rand()**. Ceci fait, nous multiplions par 3 pour sauter de rouge en rouge et ensuite obtenir les deux autres valeurs pour la couleur. Notre notificateur va ainsi clignoter de manière multicolore. Nous en faisons déjà plus que l'outil propriétaire initialement livré avec le produit.

Une palette fixe de huit couleurs n'est qu'un premier pas et ne nous offre pas beaucoup de liberté d'action. Même en multipliant les couleurs dans notre palette, nous sommes limité. Il existe une autre solution nous fournissant un accès à une infinité de teintes, ou presque. Il s'agit d'utiliser l'espace de couleurs TSV (HSV en anglais) pour Teinte, Saturation et Valeur. Respectivement nous définissons une teinte dans un cercle chromatique de 360 degrés, une saturation entre le blanc et la teinte saturée qui définit l'intensité, et une valeur qui détermine la brillance de la couleur.



Roue de couleurs HSV telle qu'on peut la trouver dans les applications de traitement et de retouche graphiques. L'une des méthodes les plus simples pour permettre à un utilisateur de choisir une couleur.

Le principal avantage de cette solution est pour une saturation et une valeur donnée (au maximum), de pouvoir choisir une teinte avec une valeur unique entre 0 et 359 degrés. Nous arrivons ainsi, dans la version la plus réduite, à une palette de 360 couleurs que nous pouvons ensuite décliner en intensité et en luminosité. Mieux encore, nous pouvons passer d'une teinte à une autre de manière douce bien plus simplement. Il nous faut cependant disposer d'une fonction de conversion permettant le passage de TSV à RVB. Après quelques recherches, on découvre qu'une telle fonction est largement documentée et implémentée dans plusieurs langages. Pour notre part nous utiliserons :

```
void HSVtoRGB(int  *r, int *g,int *b, int h, int s, int v) {
  int c;
  long l, m, n;

  // saturation zéro, pas de couleur
  if(s == 0) {
    *r = *g = *b = v;
    return;
  }

  // chroma
  c = ((h%60)*255)/60;
  h /= 60;

  // intermédiaire
  l = (v*(256-s))/256;
  m = (v*(256-(s*c)/256))/256;
  n = (v*(256-(s*(256-c))/256))/256;

  // choix dominante
  switch(h) {
    case 0:
      *r = v;
      *g = n;
      *b = l;
      break;
    case 1:
      *r = m;
      *g = v;
      *b = l;
      break;
    case 2:
      *r = l;
      *g = v;
      *b = n;
      break;
    case 3:
      *r = l;
      *g = m;
      *b = v;
      break;
    case 4:
      *r = n;
      *g = l;
      *b = v;
      break;
    default:
      *r = v;
      *g = l;
      *b = m;
      break;
  }
}
```

Notez que toutes les couleurs RVB se retrouvent dans l'espace de couleurs TSV, mais l'inverse n'est pas vrai. De plus, dans le cas qui nous intéresse aujourd'hui, le choix d'intensité lumineuse de chaque led est relativement réduit. Enfin, il faut également prendre en considération les caractéristiques des leds. Vous l'aurez remarqué dans notre palette précédente, le blanc correspond à un mélange RVB de 63, 63 et 55. De manière générale, les caractéristiques techniques des leds bleues, même dans une led RVB, indiquent une intensité plus importante. En utilisant 63, 63 et 63 nous obtenons un blanc bleuté très froid. La conversion TSV vers RVB ne sera donc pas parfaite tout en restant largement utilisable pour un simple notificateur.

Voici un petit exemple d'utilisation :

```
for(i=0;i<256;i++) {
  HSVtoRGB( &r, &g, &b, 0, 255, i );
  sendRGB(devhaccess,  r/4,  g/4,  b/4);
  usleep(1000000/250);
}

for(j=0;j<3;j++) {
  for(i=0;i<360;i++) {
    HSVtoRGB( &r, &g, &b, i, 255, 255 );
    sendRGB(devhaccess,  r/4,  g/4,  b/4);
    usleep(1000000/250);
  }
}
```

```
for(i=255;i>=0;i--) {
  HSVtoRGB( &r, &g, &b, 0, 255, i );
  sendRGB(devhaccess,  r/4,  g/4,  b/4);
  usleep(1000000/250);
}
```

Nous commençons par définir une teinte rouge et augmentons l'intensité (valeur). Nous tournons ensuite 3 fois dans le cercle de teintes puis, de retour sur le rouge, réduisons l'intensité pour revenir à un état éteint identique à la situation de départ.

Je vous laisse le soin de créer la fonction de transition permettant de passer, en douceur, d'une couleur aléatoire à une autre, soit la combinaison des deux fonctions précédentes.

5 Fonctions spéciales ?

Nous avons, pour communiquer avec le périphérique, une chaîne de huit octets. Les interrogations sont nombreuses en l'absence d'une documentation officielle. Il est, en effet, très probable qu'il y ait plus « d'intelligence » qu'on le pense dans la puce intégrée dans le périphérique. Peut-être est-il possible de spécifier un clignotement spécifique, de définir une fréquence pour cela ou encore d'utiliser des fonctions plus avancées ou de tests. Difficile de dire ce que sait vraiment faire ce notificateur, d'autant qu'il semble que le même composant central soit utilisé pour d'autres périphériques du constructeur.

Cependant, ce qui est actuellement accessible depuis un système open source ouvre des perspectives intéressantes. En effet, il n'est pas déraisonnable d'étendre les fonctionnalités du périphérique tout en gardant l'élément de base. Modifier le notificateur et l'utiliser en tant qu'élément central dans un système d'éclairage d'ambiance est l'affaire d'une simple modification : retirer la led RVB et replacer chaque ligne de contrôle par la base d'un transistor. Nous pourrions alors piloter un ensemble de leds plus important. Bien entendu, entre une telle modification et la création d'un système complet il n'y a que peu de différences. Les choses peuvent rapidement s'emballer en termes de fonctionnalités : support Ethernet, Wifi, Bluetooth, intégration d'effet pré-enregistré, réaction aux événements audio, etc.

Enfin, pour conclure, rappelons qu'il y a peu de différences entre la libUSB et les implémentations USB hôte pour les différentes plateformes de microcontrôleur. Je pense, par exemple, aux shields Arduino USB, mais également à certains modèles de PIC, d'AVR, de Cortex/ARM ou de STM32. La présence d'un port USB hôte permet de connecter un périphérique comme celui qui fait l'objet du présent article pour peu que l'on ait quelques informations sur la manière de le piloter. Bien entendu, il s'agit ici d'un simple exercice de style, il y a des méthodes plus simples pour piloter une led RVB avec un microcontrôleur. ■

WARBIKING

OU COMMENT MÉLANGER SPORT, WI-FI ET SYSTÈMES AUTONOMES ?

Je n'aime pas les activités qui n'apportent rien à mon esprit et ne créent pas grand-chose sinon rien du tout. En haut de ma liste noire d'activités futiles : le footing. Personnellement, je cours derrière quelque chose (le train, le temps, etc.) ou parce que quelque chose me court après (une bête, généralement). Le vélo en revanche est plus intéressant s'il y a une destination ou une raison (les fraises des bois ou les mûres sauvages en sont d'excellentes). Si l'on ajoute la possibilité de cataloguer les réseaux Wi-Fi au fil du trajet, voici une activité idéale pour garder la forme tout en s'amusant et avec, tantôt, une récompense sous la forme de fruits rouges...

Tout d'abord, précisons clairement les choses, à commencer par la définition que nous utilisons ici du *wardriving* et activités similaires. Contrairement à ce que précise Wikipédia sur la page française expliquant le terme, le wardriving n'est PAS le fait de tenter une intrusion dans un réseau ou une machine via un point d'accès Wi-Fi !

La page anglophone est plus pertinente : *Wardriving is the act of searching for Wi-Fi wireless networks by a person in a moving vehicle, using a portable computer or PDA*. En d'autres termes, ceci ne consiste qu'en l'activité de détecter des réseaux Wi-Fi. Pour être davantage précis, il n'y a pas, le plus souvent, de communication bidirectionnelle avec le ou les point(s) d'accès, car qui dit « communication » dit normalement « échange de données ». Le wardriving est donc une activité passive consistant simplement à « observer » ce qui nous est montré.

En terme de légalité, sans qu'on puisse se baser sur une jurisprudence quelconque, on ne peut pas dire que le wardriving, répondant à cette définition précise, soit illégal. En réalité, tout ceci est plus complexe qu'il n'y paraît. On peut légitimement estimer que lire des noms (SSID) de points d'accès et les recenser est assimilable à une activité consistant à noter sur un bout de papier les numéros dans une rue et éventuellement les noms des personnes y habitant. Ajoutons à cela que le « Code des postes et des communications électroniques » réglemente clairement la mise en place de réseaux sur le domaine public. Ainsi, une subtile distinction est faite entre un réseau établi sur la voie publique nécessitant une déclaration auprès de l'ACERP, l'Autorité de Régulation des Communications Électroniques et des Postes (anciennement ART, Autorité de Régulation des Télécommunications) et les réseaux « internes » ouverts au public. Il y a toujours une différence entre la loi et l'esprit de la loi. Au final, c'est toujours un juge qui décidera en fonction de son interprétation, mais nombreuses sont les documentations qui précisent : « L'utilisation à l'extérieur des bâtiments sur le domaine public n'est pas autorisée ». Nous sommes donc en présence d'un flou tout à fait caractéristique lorsqu'il est question de mélange droit/technologie. En conséquence, le fait de simplement « écouter » sans émettre ou communiquer ne semble aucunement répréhensible. Le fait de diffuser ces informations place ensuite le débat dans un tout autre domaine qui est celui du respect de la vie privée. Personnellement, je ne voudrais pas être le juge devant prendre une décision si un utilisateur obtient et diffuse les données privées d'une personne ayant installé un point d'accès ouvert (non protégé) sur le domaine public.

Le terme wardriving, littéralement la « guerre en conduisant », découle du *wardialing* qui consistait, il y a quelques années, à composer des numéros de téléphone aléatoirement ou en série dans le but de trouver des machines connectées à l'aide de modems (c'était bien avant la popularisation d'Internet auprès du grand public). Dans sa version moderne, la technique consiste tout bonnement à utiliser un adaptateur Wi-Fi et à se déplacer à bord d'un véhicule afin d'enregistrer les informations émises par les points d'accès environnants (SSID, BSSID, protection utilisée).

Voici un exemple typique de récepteur GPS USB tel qu'on en trouve un peu partout dans les boutiques en ligne. Généralement équipé d'un aimant permettant la fixation sur un toit de voiture, par exemple, il se résume à un récepteur couplé à un convertisseur USB/série communiquant à 4800 bps.

Les motivations incitant au wardriving sont multiples. La curiosité en est une, la collecte d'informations statistiques en est une autre, ou encore, tout simplement, la volonté d'en apprendre un peu plus sur le fonctionnement de cette technologie, éventuellement couplée à celle du positionnement GPS. D'autres vous diront simplement « parce que c'est possible » ou que cela passe le temps, mais personnellement, c'est un petit plus pour motiver quelques ballades bien salutaires en VTT et une excuse pour un petit article dans Open Silicium.

A propos du VTT/cyclisme justement, le *warbiking* est l'une des déclinaisons du wardriving troquant la voiture contre le vélo. Dans le même genre de variations, on trouve le *warwalking* ou *warjogging* pour la marche, le *warrailing* pour le train, le métro ou tramway, mais aussi, plus original, le *warballooning* avec des systèmes autonomes dérivant aléatoirement dans le ciel. En réalité, toutes les déclinaisons sont possibles et permettent d'établir des cahiers des charges et de poser des problématiques aussi intéressantes les unes que les autres. Finalement, le recensement des points d'accès n'est pas tant une fin en soi qu'une excuse pour apprendre et expérimenter.

Bien entendu, il doit exister des individus qui procèdent à ce genre de démarches dans le seul but de faire de la reconnaissance, pour ensuite cibler et attaquer les points d'accès utilisant une protection faible (WEP) ou inexistante. Cela reste je pense une minorité, car le ratio résultat/travail est de loin inférieur à d'autres techniques permettant de prendre le contrôle de machines de pauvres utilisateurs Windows sans défenses.

1 Problématique et matériel

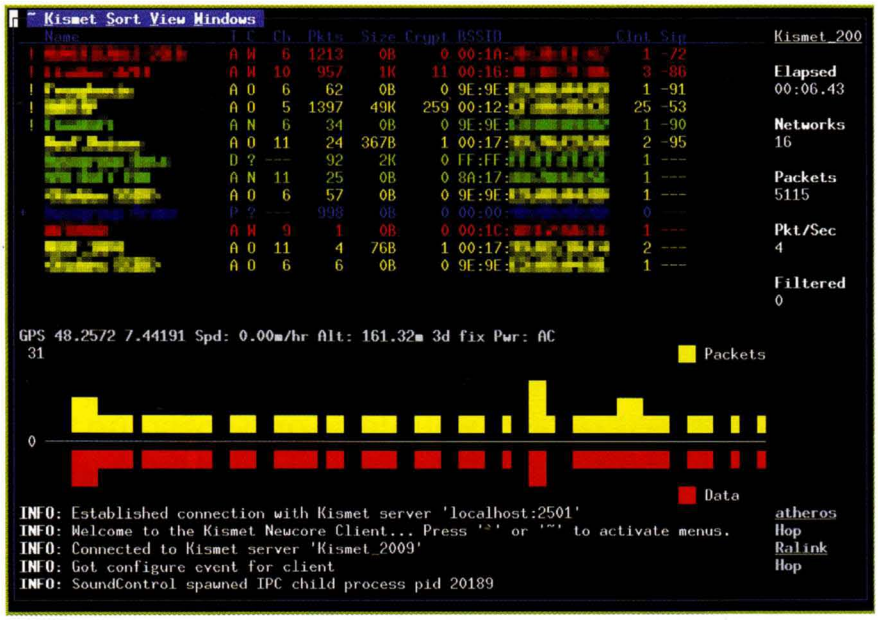
Dans les grandes lignes, le warbiking est légèrement plus contraignant que le wardriving. En effet, l'autonomie est un point capital. Alors qu'un véhicule motorisé, comme une voiture, est en mesure de fournir une alimentation puissante et constante, un vélo est généralement mû par la seule force musculaire de son propriétaire. Dans le même ordre d'idée, alors que le poids de l'équipement dans une voiture importe peu, il n'en va pas de même sur un vélo. Autre point intéressant, la vitesse de déplacement d'un vélo est bien plus faible que celle d'une voiture. Ceci, en revanche, est plutôt un point positif à l'inverse des deux précédents.

En toute logique donc, le système doit répondre aux critères suivants :

- Il doit être portable et donc relativement léger.
- Il doit fonctionner à partir d'une source d'alimentation autonome et éventuellement disposer de fonctionnalités d'économie d'énergie.
- Il doit être facilement utilisable dans des conditions où clavier et souris perdent toute ergonomie.
- Il doit être en mesure de faire fonctionner des applicatifs ouverts, open source et stables (nous n'allons pas ré-implémenter un applicatif complet pour quelques tours de pédales).
- Il doit utiliser du matériel robuste et économique, l'un étant compensé par l'autre, puisqu'à défaut de résister à une chute, il ne doit pas provoquer une déprime de 72h chez son utilisateur (potentiellement déjà blessé dans ladite chute).

La conclusion s'impose d'elle-même puisqu'à quelques nuances près, ce que nous venons de décrire s'appelle un ordinateur portable et plus exactement un netbook. Autonomie, mobilité, robustesse (selon modèle) et faible encombrement sont parfaitement adaptés. De plus, qui dit « netbook » dit nécessairement « Wi-Fi » et « plateforme compatible GNU/Linux » (selon modèle) et par là même « grand choix d'outils matures et stables ».

Inutile donc de se creuser la tête pour choisir une plateforme embarquée à base d'ARM ou de MIPS nécessitant une personnalisation extrême du système, des chaînes de compilation et de l'IHM. Attention, ceci peut être justifié avec un



Le client Kismet en fonctionnement présente la liste des réseaux détectés, diverses informations sur le travail en cours, un graphique des quantités de données détectées et la position GPS actuelle. Tout ceci est, bien entendu, pleinement configurable directement dans l'interface en mode texte (couleurs, informations, notifications, etc). La configuration est ensuite stockée dans des fichiers placés dans ~/.kismet/ et rechargée lors du prochain démarrage du client.

cahier des charges différent pour, par exemple. du warballooning ou même du warwalking, où le poids aura, par exemple, une toute autre importance.

Deux éléments annexes complètent cette base qui sera, dans le cas présent, un netbook Samsung NB30 (Atom N450 & 1Go de RAM) :

- Un adaptateur Wi-Fi WUSB54CG de LinkSys/Cisco connecté en USB et utilisant le pilote Linux rt73usb. Bien que le netbook dispose d'un adaptateur Wi-Fi PCI interne, un périphérique USB sera utilisé. Le fait de conserver l'intégrité du netbook tout en s'offrant des ouvertures quant à l'ajout futur d'une antenne omnidirectionnelle externe s'en trouve grandement facilité via l'usage d'une clef USB. Une erreur de manipulation pouvant survenir lors des modifications, mieux vaut avoir à changer une clef USB qu'un netbook. Autre point intéressant, l'utilitaire dont nous allons nous servir (Kismet) est en mesure de gérer plusieurs adaptateurs Wi-Fi afin de couvrir simultanément plusieurs canaux Wi-Fi. Bien entendu, la multiplication des adaptateurs USB, équipés ou non d'antennes de quelques 30cm, impactera directement la consommation électrique et donc l'autonomie (même en utilisant un hub USB actif alimenté individuellement). La saturation du bus USB est également à prendre en considération, même si l'utilisation de 13 adaptateurs ne devrait poser aucun problème.
- Un récepteur GPS USB. Puisque l'idée n'est pas de créer une stupide liste de points d'accès, mais bel et bien d'avoir une excuse pour entretenir (ou obtenir) de belles cuisses, il nous faut conserver le chemin parcouru. La géolocalisation GPS est une solution efficace et économique. Ainsi, un simple récepteur GPS acheté quelques euros sur eBay fera l'office. Ces périphériques utilisent de simples convertisseurs USB/séries et envoient des trames NMEA intelligibles par n'importe quelle application compatible. Aucun pilote n'est donc nécessaire si ce ne sont ceux déjà intégrés dans le support du noyau Linux.

Le choix de l'adaptateur Wi-Fi n'est pas très important pour peu qu'il puisse être utilisé sous Linux en mode moniteur. La seule fonctionnalité nécessaire tient en la possibilité de changer le canal de réception pour procéder à ce qu'on appelle du *channel hoping*. Kismet, dont nous parlerons plus tard, prend en charge un grand nombre d'adaptateurs. En effet, bien qu'il existe des fonctionnalités communes pour ces adaptateurs, certaines spécificités en fonction des pilotes, doivent être prises en compte. Ici, en plus de la clef WUSB54CG, nous pouvons utiliser l'adaptateur miniPCI Atheros AR9285 intégré au netbook.

Netbook, clef Wi-Fi et récepteur GPS forment la base du système qui sera complété, dans le futur, par un afficheur 7 segments à leds permettant d'afficher un compte de points d'accès détectés sans avoir à relever l'écran du netbook. Une autre solution pourrait consister à développer un petit programme simple en Python, C, ou shell, affichant cette information en grand à l'écran. Cependant, l'ouverture de l'écran présente un risque plus important pour le matériel en cas de chute. De plus, le rétro-éclairage réduira l'autonomie du système déjà passablement limité du fait de l'activité permanente du disque dur, des périphériques USB, etc. Notez au passage que la configuration de la gestion des évènements ACPI est indispensable puisque le fait de rabattre l'écran ne doit pas passer en veille l'ordinateur, mais uniquement éteindre l'écran (ou du moins son système de rétro-éclairage).

La principale difficulté technique réside donc, non pas dans le choix et la connexion des composants, mais dans l'agencement et la fixation de tout ce petit monde sur un VTT. Dans le cas présent, la récupération d'un support de guidon pour un panier et quelques fixations annexes ont fait l'affaire. Tout ceci est, bien entendu, totalement dépendant du netbook choisi.

Avec un système d'affichage déporté éventuellement complété par des contrôles manuels, l'ensemble de l'électronique « lourd » pourra être placé à l'arrière du vélo, plus facilement aménageable pour des antennes supplémentaires et éventuellement une perche pour les fixer. Ceci fait partie des améliorations qui seront apportées à l'ensemble après les premières expérimentations sur le terrain.

2 Logiciels

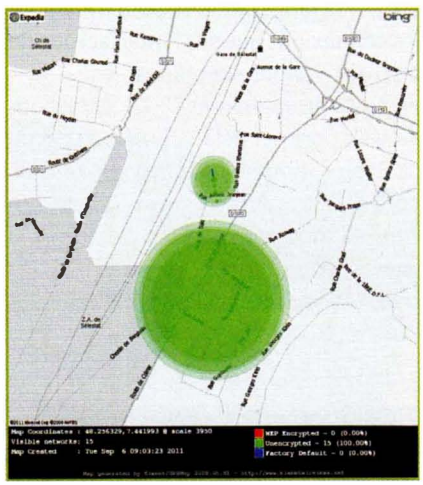
Comme de coutume, le système d'exploitation utilisé sur le netbook sera un GNU/Linux et en particulier une distribution Debian. La modularité de cette distribution fait que le netbook ne sera pas obligatoirement dédié à l'activité cycliste. Le choix de GNU/Linux en lui-même se passe presque d'explication tant il est évident que les outils disponibles avec ce système sont bien plus nombreux, stables et personnalisables à souhait.

Kismet représente le centre de notre modeste assemblage. Kismet est un logiciel libre de détection de réseaux (entre autres choses) pour réseaux sans fil 802.11. Il est disponible en version GNU/Linux, *BSD et Mac OS X. Lorsqu'on parle de Kismet, on fait généralement référence à l'architecture logicielle Kismet dans son ensemble. En réalité, Kismet se compose de trois éléments :

- Le serveur Kismet, qui est la partie active de recherche et de catalogage de réseaux Wi-Fi. Celle-ci fonctionne sans interface particulière et peut être lancée en tâche de fond.
- Le client Kismet, qui est une interface se connectant au serveur et capable de mettre en forme les informations collectées afin que l'utilisateur puisse voir la progression du travail.
- Le drone Kismet, qui n'est pas obligatoire et qui constitue une sorte de sonde logicielle chargée de collecter les informations et de les envoyer à un serveur.

L'infrastructure Kismet complète utilise ainsi un serveur qui conserve les données collectées par ses soins, ainsi que celles de zéro, une ou plusieurs sonde(s) située(s) sur des ordinateurs distants. Enfin, le client présente tout cela à l'utilisateur. L'interface par défaut est en mode texte et lancée dans une console. Les clients Mac et Windows sont un peu plus graphiques, mais présentant des données textuelles, ils n'apportent rien de particulier en termes d'ergonomie.

On pourrait donc facilement se passer, sur un système UNIX, de l'interface graphique X. Le gain en termes d'autonomie ne serait cependant pas énorme et surtout, ce serait se passer d'une des particularités du NB30 : un écran tactile résistif. Une fois correctement configuré, celui-ci permettra de facilement contrôler le client Kismet durant les phases de test. Attention, lorsque je parle d'interface X, comprenez bien qu'il ne s'agit pas de saturer mémoire et processeur avec un environnement graphique surchargé et tape-à-l'œil. Des horreurs lourdes comme GNOME ou KDE seront donc à proscrire au bénéfice d'environnements légers comme e17 ou LXDE. Moins vous aurez de processus en mémoire, moins le CPU travaillera et plus longtemps durera votre ballade de santé.



Exemple de carte produite par gpsmap. Cet utilitaire est présent dans les sources de Kismet, mais absent du paquet mis à disposition sur le site officiel. Ceci n'est pas un problème, car il est peu probable que vous produisiez les cartes sur le netbook ayant servi à détecter les réseau Wi-Fi. Un Kismet récent sur le netbook donc, et un paquet officiel Debian/Ubuntu sur la machine de traitement.

Le drone Kismet ne nous sera pas utile et nous nous limiterons au serveur et à son interface. En revanche, la gestion du GPS passera par le démon GPSd. Il est possible de laisser le serveur Kismet accéder directement au port série et donc aux données GPS/NMEA. Cependant, nous avons pu constater que le format des trames NMEA n'est pas toujours respecté avec des récepteurs économiques d'entrée de gamme. Le démon GPSd s'en sort, quant à lui, beaucoup mieux et nous « brûlerons » quelques cycles CPU pour utiliser une architecture client/serveur. Accessoirement, comme beaucoup d'applications sont compatibles avec le serveur GPSd, nous pourrons les utiliser en même temps que Kismet si nécessaire.

La configuration de Kismet passe par le fichier **/usr/etc/kismet.conf** où sera décrite l'utilisation des différents composants, à commencer par l'interface Wi-Fi :

```
ncsource=wlan2:name=Ralink
```

La **ncsource** désigne l'interface Wi-Fi à utiliser. Présentement, **wlan0** est l'adaptateur interne Atheros et **wlan1** un autre exemplaire du même modèle de cette clef LinkSys/Cisco. Udev, dans le système, attribue un nom d'interface unique à chaque adaptateur indépendamment de l'ordre de connexion. Ainsi, votre **wlan2** restera toujours **wlan2**. Si vous avez un doute, un petit **cat /proc/net/dev** vous donnera la liste des interfaces et **iwconfig** vous apportera le reste des informations.

Notez que le serveur Kismet utilise un algorithme avancé pour détecter et enregistrer les informations sur les points d'accès qu'il rencontre. En Wi-Fi, il y a jusqu'à 13 (voire 14) canaux utilisables et un adaptateur, tout comme un point d'accès, n'utilise qu'un seul canal. En conséquence, lors de la phase de

Remarque **A propos des versions**

Précisons ici que le paquet Debian, et celui d'Ubuntu par voie de conséquence, est relativement ancien (mai 2008). Les responsables du projet Kismet s'en sont rendu compte et référencent un paquet non-officiel (par rapport à Debian) à jour. C'est cette version que nous utilisons ici, ce qui explique l'emplacement peu courant du fichier de configuration.

On trouve beaucoup de documentations sur le Web concernant des versions plus anciennes de Kismet, or la syntaxe du fichier de configuration a sensiblement changé. Personnellement, je préfère utiliser une version récente d'un utilitaire et lire la documentation officielle de A à Z que d'installer une version vieille de plus de 2 ans, mais documentée abondamment par ailleurs.

recherche (de *scan*), votre adaptateur client (dans la structure *managed*) va donc parcourir tous les canaux qu'il peut utiliser (vérifiez avec **iwlist [interface] channel**) pour trouver un point d'accès et vous l'afficher, pour vous proposer de vous y connecter.

Lorsqu'on cherche à cartographier et répertorier des points d'accès tout en se déplaçant, un problème se pose. Imaginons que nous scannons du canal 1 au 13 linéairement et qu'à l'instant où nous commençons, nous passons devant quelques points d'accès respectivement sur les canaux 8, 12 et 10... Nous allons tout simplement les rater. Il n'y a pas de solution miracle et plus nous nous déplaçons rapidement, plus nous raterons de réseaux. Ce que Kismet suppose, c'est que certains canaux sont plus utilisés par défaut que d'autres et que peu d'utilisateurs changent la configuration par défaut. Il met donc en œuvre une suite savamment composée, surveillant davantage certains canaux que d'autres. De plus, en présence de plusieurs adaptateurs, il est à même de faire en sorte qu'à aucun moment deux adaptateurs ne cherchent des réseaux sur le même canal. Il est également possible de configurer manuellement un canal ou une liste de canaux par adaptateur. Si vous disposez de 13 clefs Wi-Fi, vous pouvez alors configurer l'ensemble de manière à scanner la totalité des canaux simultanément (au prix d'une consommation élevée de courant).

Toutes ces subtilités se configurent sur la ligne donnée précédemment. Je vous invite à consulter le **README** intégré à la documentation de l'utilitaire pour en savoir plus. Concernant le récepteur GPS, voici ce qu'il nous faut ajouter :

```
gps=true
gpstype=gpsd
gpshost=localhost:2947
gpsreconnect=true
```

La première ligne active l'utilisation du récepteur. Le type est donné sur la seconde ligne où nous précisons l'utilisation du démon GPSd. Comme ce dernier est, par défaut, configuré pour n'attendre les connexions que sur l'interface locale **127.0.0.1**, nous spécifions **localhost** en guise de serveur et **2947** correspondant au port par défaut. Enfin, en cas de problème, nous fixons **gpsreconnect** à VRAI pour permettre la reconnexion automatique en cas de perte. Ainsi, si nécessaire, nous pourrons arrêter et relancer le démon GPSd sans relancer celui de Kismet.

Dans la configuration actuelle, nous n'avons pas besoin d'utiliser l'aspect pleinement client/serveur étant donné que les deux éléments fonctionneront sur la même machine. Sachez cependant qu'il vous est possible de configurer le serveur de manière à attendre les connexions de clients réseaux grâce à :

```
listen=tcp://127.0.0.1:2501
allowedhosts=127.0.0.1
maxclients=5
```

listen précise l'adresse de l'interface et le port sur lesquels attendre les connexions. **allowedhosts** nous permet de spécifier une liste ou une plage d'adresses de clients autorisés et enfin, comme son nom l'indique **maxclients** indique le nombre maximum de clients à supporter.

A présent, tout est prêt pour le premier démarrage. Après connexion du GPS nous relançons le serveur avec **/etc/init.d/gpsd restart**. Nous attendons quelques instants, le temps que le récepteur trouve suffisamment de satellites pour une géolocalisation précise. Nous pouvons nous aider de **xgps**, un client graphique livré avec GPSd pour nous assurer de la bonne marche des choses. Une fois que nous sommes prêts sur ce point, nous lançons soit **kismet**, qui lancera à la fois le serveur et le client, soit **kismet_server** dans une fenêtre **xterm**, puis **kismet_client** dans une autre. Bien entendu, si vous n'avez pas besoin d'une vision en temps réel de la situation, mieux vaut se passer du client pour économiser des ressources et donc gagner en autonomie.

Il ne nous reste plus qu'à nous promener au gré des rues pour faire notre petit marché et collecter des informations sur le taux d'informatisation de notre voisinage...

3 Résultats, conclusion et améliorations

Après quelques essais, l'ensemble du système répond aux attentes, même si un problème assez classique fait surface : le soleil. Ce n'est pas nouveau, la luminosité d'un écran en plein soleil, en particulier celui d'un netbook tactile, est presque insuffisante pour une lecture correcte. Ceci n'est pas excessivement grave étant donné les risques découlant du fait de fixer l'écran tout en se déplaçant (on se fait peur parfois). Kismet offre une solution pour se tenir informé de l'état d'avancement de son travail. Pour cela,

il vous suffit d'installer le paquet **sox** contenant le player **play** permettant de jouer un certain nombre d'avertissements sonores en fonction des évènements. Côté client, vous pouvez configurer la notification audio directement dans l'interface texte. Côté serveur, cela se fera dans le fichier **kismet.conf** :

```
enablesound=true
soundbin=play
sound=newnet,true
sound=newcryptnet,true
sound=packet,true
sound=gpslock,true
sound=gpslost,true
sound=alert,true
```

Notez que ces lignes configurent un son émis par le serveur Kismet et n'a rien à voir avec ceux du client. Les fichiers WAV intégrés au paquet Kismet sont relativement médiocres, mais vous trouverez sans doute dans le répertoire **/usr/share/sounds** de quoi vous satisfaire. Un son discret et bref comme un simple « bip » fera très bien l'affaire. Il est également possible de configurer serveur et client pour utiliser un système de synthèse vocale.

Une autre amélioration possible concerne la robustesse de l'ensemble. Échanger le disque dur classique intégré au netbook avec un disque SSD devrait réduire la consommation, limiter les problèmes liés aux vibrations et réduire le dégagement de chaleur (et donc la consommation du ventilateur intégré). C'est là quelque chose qui sera fait dans les prochains temps, avec le système décrit ici.

L'utilisation des deux adaptateurs, bien que celui intégré ne permette pas autant de possibilités en termes de hack, apporte un gain en efficacité tout à fait intéressant. En effet, même à la vitesse de déplacement d'un vélo, le signal d'un réseau Wi-Fi peut être « manqué ». Utiliser plusieurs clefs via un hub disposant de sa propre source d'alimentation est dans la *TODO list*, tout comme l'utilisation d'antennes externes plus performantes.

A propos d'antennes, il est important de noter ici que la cartographie des données repose sur la position où vous vous trouvez au moment où un nouveau réseau est détecté. Par conséquent, les cartes que vous allez créer, avec **gpsmap** par exemple, n'affichent pas la position des points d'accès trouvés, mais la vôtre. On remarquera également qu'il est relativement difficile d'utiliser les données GPS issues de Kismet en dehors de **gpsmap**.

Le netbook Samsung NB30 qui, même s'il n'a pas été acquis pour l'usage décrit ici, présente plusieurs caractéristiques intéressantes dont un écran tactile, une coque solide, un encombrement réduit et trois ports USB.

Quelques scripts de conversion sont disponibles sur le Web afin de pouvoir utiliser ces données avec Google Earth ou Google Map, par exemple. Mais vous n'êtes pas limité à cette seule représentation. Pourquoi ne pas tenter de mesurer vos performances physiques non pas en fonction du nombre de kilomètres parcourus, mais du nombre de points d'accès trouvés ? Il pourrait en résulter de beaux graphes si vous pédalez avec assiduité.

Parmi les extensions possibles, nous avons déjà abordé l'éventualité d'un afficheur 7 segments à leds. Un montage relativement simple consisterait à utiliser un microcontrôleur quelconque pouvant communiquer en mode série ou USB. Celui-ci recevrait ainsi une valeur numérique qu'il lui faudrait afficher. Le calcul de cette information est dépendante de la manière dont on la récupérera. Kismet ne semble pas disposer d'une fonction permettant de lancer, par exemple, un script lors de la découverte d'un nouveau réseau. En revanche, il y a la notification sonore. Nous pouvons donc très facilement remplacer **play** par un script shell ou un code en Python permettant d'envoyer un « tic » à notre montage. Une autre solution, sans doute moins élégante, peut consister à parcourir le fichier produit par Kismet et compter les SSID avant d'envoyer l'information à l'afficheur. Les ressources CPU et disque utilisées par cette option en font cependant une solution peu adaptée au warbiking.

Arrêtons là ce qui n'est qu'un article de présentation d'une solution largement perfectible. Nous vous conseillons de parcourir le fichier de configuration de Kismet, largement commenté, ainsi que la documentation. Cet outil s'est tellement bonifié au fil du temps qu'il offre des fonctionnalités de personnalisation inespérées. Ce qui est présenté ici n'est qu'une base et, qui sait, votre projet final se rapprochera davantage d'un *StreetView Bike* (avec asservissement d'un APN) que d'une simple collecte de points d'accès... Ah oui, c'est vrai, les voitures de Google StreetView ne collectent plus ces informations, c'est votre téléphone Android qui le fait à présent, paraît-il ;) ■

MISE À JOUR D'UN NAS LACIE ETHERNET RAID

par Denis Bodor

Les NAS sont des terrains de jeu fantastiques. Ainsi, en apprendre de plus en plus sur ces périphériques, très souvent sous GNU/Linux, est non seulement très satisfaisant, mais également très utile. Une connaissance approfondie des mécanismes en œuvre vous permet ainsi de faire évoluer vos périphériques de stockage réseau, diagnostiquer les pannes et améliorer leur fonctionnement général. Ceci, sans compter, bien sûr, le gain d'expérience personnelle concernant le système GNU/Linux. Le cas qui nous intéresse aujourd'hui est celui d'un NAS ayant déjà quelques années, construit par LaCie et utilisant 4 disques SATA en RAID fournissant 1 To de stockage.

La raison pour laquelle je me suis penché sur ce périphérique est sa défaillance soudaine. Utilisé initialement comme un matériel en production, l'un de ses disques a montré des signes de faiblesse m'obligeant à basculer la production sur son jumeau et ajouter un nouveau NAS à l'infrastructure. C'est le jeu habituel des chaises musicales, un NAS défaille, celui en *spare* prend sa place et un nouveau remplace le spare. Je ne voulais pas, cependant, laisser mourir la bête ayant bien servi la cause 24h/24 et 7j/7 pendant plus de 4 ans et qui finalement, serait sans doute réparable.

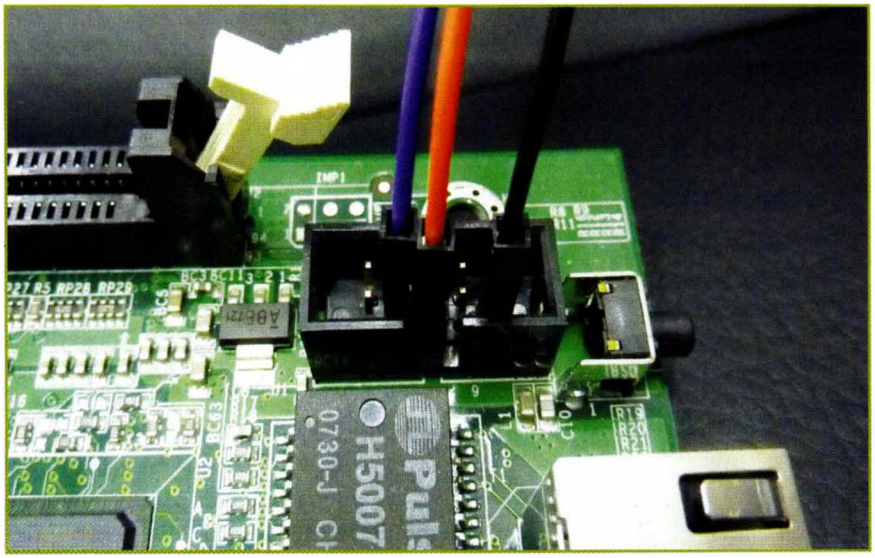
1 Tout commence par une console série

Le NAS Ethernet RAID de LaCie semble peu utilisé ou du moins peu documenté côté « bidouille » au bénéfice de NAS plus petits et, à l'époque, moins coûteux. Avec surprise, ce n'est pas dans les discussions consacrées aux matériels LaCie qu'on trouve la réponse à la question du branchement d'une console série, mais du côté Intel. Le produit est en effet basé sur le Intel Entry Storage System SS4000-E dont il partage la même base. C'est sur tomjudge.com que j'ai trouvé la description de la connexion sur le connecteur COMB1 de la carte principale après démontage presque complet du NAS en ma possession (davantage par curiosité que par nécessité). Le connecteur DL-10 ressemble à ceux présents sur les cartes mères et les cartes PCI fournissant un port RS232, mais il n'utilise pas un brochage identique. La connexion se fera ainsi :

```
          9 7 5 3 1
(bouton   o o o o o
 reset)   . o o o o
            8 6 4 2   (emplacement RAM)

COMB1    DB9 mâle
  3 ------ 2
  5 ------ 3
  9 ------ 5 (masse)
```

Ensuite, un simple câble null-modem fera l'affaire. Si vous n'en possédez pas, inversez/croisez tout simplement 2 et 3 et reliez le port au port série du PC avec un câble femelle/femelle série (sans doute moins évident à trouver qu'un null-modem). Notez bien qu'il s'agit là de RS232 (+/- 12V) et non de signaux TTL. N'utilisez donc surtout par un adaptateur série TTL sur USB (type FTDI FT232R) au risque de le détruire. La connexion se fera en 115200 8N1 à l'aide d'un émulateur de terminal série comme Minicom, `cu` ou GNU screen (`screen /dev/ttyXXX 115200`).

Puisque nous en sommes à parler de matériel, on remarquera quelques caractéristiques intéressantes dans ce NAS. Tout d'abord, les connecteurs sur la carte principale sont tous soudés et au pas de 2,54 mm. Ensuite, l'alimentation se fait via un connecteur ATX. C'est remarquable, car remplacer l'alimentation ne sera pas bien difficile comparé à d'autres périphériques du même type, plus récents. L'alimentation et les mesures du système de ventilation sont également standards et facilement remplaçables. On peut donc y voir là un matériel fait pour durer et aisé à réparer.



Connexion de la console série sur le connecteur DL-10 interne. Attention, il s'agit de RS232 et non de tensions TTL. La connexion se fera donc sur un véritable port série, éventuellement au travers d'un adaptateur USB/RS232.

Dès le connecteur branché et la connexion établie, on peut faire un tour du propriétaire. Après démarrage du NAS via RedBoot, un bootloader standard et bien connu, on arrive sur une invite de login. Les getty sont rares lorsqu'on se connecte en console série. Beaucoup de constructeurs estiment que l'accès à cette console est sans doute suffisamment difficile pour ne pas demander de mot de passe. Ici, ce n'est pas le cas. Le login sera naturellement **root** et le mot de passe celui défini pour l'administration du système dans l'interface web ou, par défaut **storage**.

Nous pouvons alors en apprendre un peu plus sur le matériel :

```
lacieRAID login: root
Password:
Welcome to

                Powered by FalconStor Software, Inc.
                Copyright 2001-2006 by FalconStor.
                All Rights Reserved.
                http://www.falconstor.com
#

# uname -a
Linux lacieRAID 2.6.10-iop1-9 #1 Tue Jan 2 14:52:23 CST 2007 armv5tel unknown

# cat /proc/cpuinfo
Processor       : XScale-IOP8032x Family rev 0 (v5l)
BogoMIPS        : 395.67
cpu MHz         : 395.673
Features        : swp half thumb fastmult edsp
CPU implementer : 0x69
CPU architecture: 5TE
CPU variant     : 0x0
CPU part        : 0x2e2
CPU revision    : 0
Cache type      : undefined 5
Cache clean     : undefined 5
Cache lockdown  : undefined 5
Cache format    : Harvard
I size          : 32768
I assoc         : 32
I line length   : 32
I sets          : 32
D size          : 32768
D assoc         : 32
D line length   : 32
D sets          : 32

Hardware        : Intel IQ31244
Revision        : 0000
Serial          : 0000000000000000

# cat /proc/meminfo
MemTotal:       256916 kB
MemFree:         54520 kB
Buffers:          5300 kB
Cached:          76192 kB
[...]

# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00040000 00020000 "RedBoot"
mtd1: 00020000 00020000 "log"
mtd2: 00040000 00020000 "rammode"
mtd3: 00020000 00020000 "naskey"
mtd4: 00200000 00020000 "zImage"
mtd5: 00400000 00020000 "ramdisk.gz"
mtd6: 01880000 00020000 "vendor"
mtd7: 00080000 00020000 "wmdata"
mtd8: 00001000 00020000 "RedBoot config"
mtd9: 00020000 00020000 "FIS directory"
```

Un processeur Intel XScale ARMv5 est au cœur du système avec 256 Mo de mémoire (sous forme d'une barrette standard et donc upgradable. Une flash de 32 Mo permet le stockage du firmware initial comprenant le bootloader. le noyau Linux, le système de fichiers en RAM, etc. Mais le NAS. comme souvent. utilise en réalité une partie du premier disque pour le système de fichiers racine et le firmware utilisé en fonctionnement normal. La flash ne sert qu'à démarrer le NAS et à fournir un système en cas de réinitialisation ou de changement de disques.

```
# cat /proc/mdstat
Personalities : [linear] [raid0] [raid1]
[raid5] [raid10]
md1 : active raid10 sdc3[4] sda3[0] sdd3[3]
sdb3[1]
```

```
      975983744 blocks 64K chunks 2 near-copies [4/3] [UU_U]
      [=>.................]  recovery =  7.8% (38501120/487991872) finish=17049.6min
speed=436K/sec
md0 : active raid1 sdc1[2] sda1[0] sdd1[3] sdb1[1]
      262976 blocks [4/4] [UUUU]

unused devices: <none>

# mdadm --detail --scan
ARRAY /dev/md1 level=raid10 num-devices=4 spares=1 UUID=135b605c:effb8d25:eb06c658:694f6ec7
   devices=/dev/sda3,/dev/sdb3,/dev/sdd3,/dev/sdc3
ARRAY /dev/md0 level=raid1 num-devices=4 UUID=a3157ae5:c785e1ae:4d90c3ee:2b7f1348
   devices=/dev/sda1,/dev/sdb1,/dev/sdc1,/dev/sdd1

# mount
/dev/md0 on / type ext3 (rw)
none on /proc type proc (rw,nodiratime)
none on /sys type sysfs (rw)
none on /proc/bus/usb type usbfs (rw)
/dev/vbdi2 on /nas/NASDisk-00002 type xfs (rw,nosuid,usrquota)
/dev/vbdi3 on /nas/NASDisk-00003 type xfs (rw,nosuid,usrquota)
/dev/vbdi7 on /nas/NASDisk-00007 type xfs (rw,nosuid,usrquota)
```

Les disques sont organisés en RAID 1 ou 10 selon qu'il s'agisse du système de fichiers racine sur **md0** ou de stockage **md1** (dépendant de la configuration initiale du NAS bien sûr). Les systèmes de fichiers de stockage sont en XFS et la racine en ext3. Enfin, côté noyau, on constate, qu'en plus des modules habituels, on trouve des éléments spécifiques et parfois propriétaires :

```
# lsmod
Module                  Size  Used by    Tainted: P
nfsd 91876 8 - Live 0xbf236000
exportfs 4320 1 nfsd, Live 0xbf233000
lockd 59768 2 nfsd, Live 0xbf223000
sunrpc 128164 2 nfsd,lockd, Live 0xbf202000
kvbdi 40728 3 - Live 0xbf1f7000
fsiscsi 177124 0 - Live 0xbf1ca000
kipstor 758232 5 kvbdi,fsiscsi, Live 0xbf10f000
kfsnalias 91708 0 - Live 0xbf0f7000
kfsnevt 53336 2 kipstor,kfsnalias, Live 0xbf0e8000
krudp 205900 3 fsiscsi,kipstor,kfsnevt, Live 0xbf0b4000
kfsnbase 225892 8 kvbdi,fsiscsi,kipstor,kfsnalias,kfsnevt,krudp, Live 0xbf028000
fsnlzo 276340 1 kipstor, Live 0xbf06f000
usb_storage 27264 0 - Live 0xbf289000
vfat 11008 0 - Live 0xbf285000
fat 33596 1 vfat, Live 0xbf27b000
nls_iso8859_1 3648 0 - Live 0xbf279000
nls_cp437 5280 0 - Live 0xbf276000
nls_base 5924 4 vfat,fat,nls_iso8859_1,nls_cp437, Live 0xbf273000
ohci_hcd 14952 0 - Live 0xbf26e000
ehci_hcd 23972 0 - Live 0xbf267000
usbcore 97388 4 usb_storage,ohci_hcd,ehci_hcd, Live 0xbf24e000
bksc_mod 53248 1 - Live 0xbf061000
e1000 79284 0 - Live 0xbf013000
gd31244 51448 12 - Live 0xbf005000
eeprom 6016 0 - Live 0xbf002000
gpio 4040 0 - Live 0xbf000000
```

Le firmware, côté applicatif serveur, est principalement constitué de la solution IPstore de la société FalconStor. On retrouve d'ailleurs ce nom dans le message post-login et un peu partout dans le système. Il semblerait que, entre autres choses, le contenu de **/usr/local/ipstor** constitue la base de ce système et nous n'avons trouvé aucune trace des sources sur le net. Ceci inclut le module **kvbdi.ko** et d'autres, placés dans **/usr/local/ipstor/lib/modules/2.6.10-iop1-9/**.

Une partie du firmware est donc clairement propriétaire, comme cela nous sera confirmé par la suite.

2 Migration vers un firmware Intel

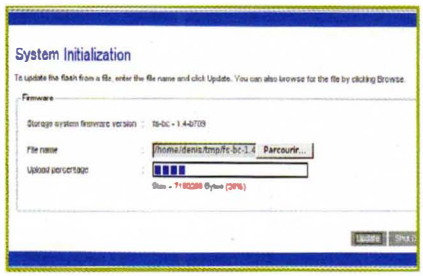
Je n'ai absolument rien contre LaCie, au contraire (cf article dans Open Silicium 2 sur le BigDisk Network), et même si je considère que la présence d'éléments propriétaires dans un NAS est une très mauvaise chose, ce n'est pas la raison primaire d'une migration vers un firmware proposé par Intel. En réalité, le fait est que la majorité des informations sur ce NAS proviennent du SS4000-E et que le *Download Center* Intel (downloadcenter.intel.com) met à disposition un firmware compilé et une archive de sources accompagnée de documentations. Je suis, comme n'importe quel représentant de mon espèce, feignant et, entre chercher des heures sur les sites de support LaCie et tenter une migration, le plus simple est de prendre le risque. Il faut préciser que le site de support LaCie est très beau et ergonomique mais que, dans un souci de simplicité je suppose, les matériels en fin de vie sont souvent enterrés au fin fond d'un FTP. Retrouver la trace du NAS en ma possession s'est vite avéré pénible et je ne suis pas patient. Fort heureusement, la manipulation m'a donné raison, les firmwares LaCie et Intel semblent être tout bonnement identiques, voire le fruit d'un développement conjugué (avec FalconStor et Lanner Electronic pour certains éléments).

J'ai donc suivi les recommandations de la documentation de mise à jour d'Intel :

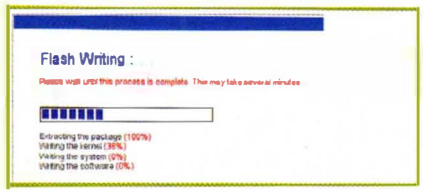
- Récupérer le firmware binaire **fs-bc-1.4-b710.pkg** du centre de téléchargement Intel.
- Récupérer toutes les données du NAS puisque la manipulation est destructive.
- Arrêter le NAS complétement par n'importe quel moyen. Ceci n'a aucune importance. En l'occurrence,

le NAS était en pleine reconstruction RAID lorsque celui-ci a été coupé par déconnexion de l'alimentation (c'est barbare, je sais).

- Sortir tous les disques du NAS.
- Mettre en route le NAS afin qu'il boote sur le firmware en flash uniquement.
- Une fois le périphérique démarré, celui-ci est accessible via l'adresse 192.168.1.101 (précédemment via 192.168.100.100, merci la console série et **ifconfig**) et on pointe son navigateur sur l'adresse en question.
- Une page d'initialisation est affichée par défaut, attendant l'insertion des disques. Il ne faut pas insérer les disques à ce moment.
- Pointer le navigateur sur http://192.168.1.101/system_init_vendorF.cgi.
- Dans le formulaire de mise à jour, choisir le fichier **fs-bc-1.4-b710.pkg** et valider le formulaire.

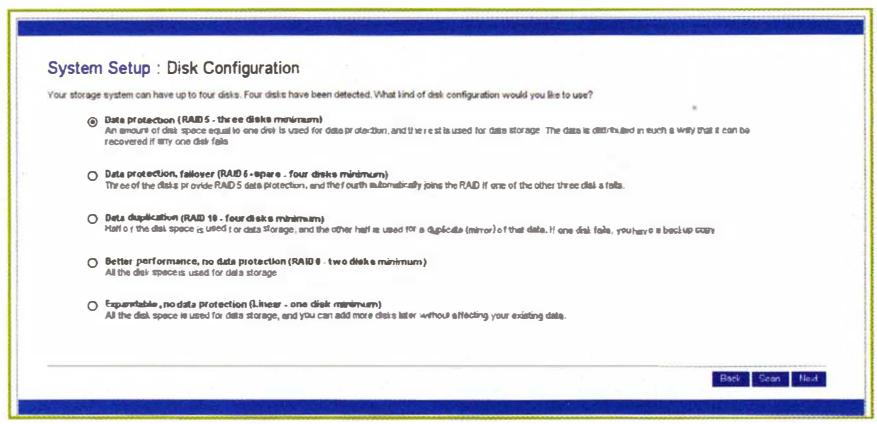


- Le firmware est téléchargé et la flash est réinscrite étape par étape : noyau, système, puis applicatifs.



- Au terme de la mise à jour, le NAS redémarre et on pointe à nouveau le navigateur sur son IP (inchangée).
- Sans éteindre le périphérique, on réinsère alors les disques un par un. Il ne faut surtout pas redémarrer avec les disques en place car, dans ce cas, le premier array RAID est utilisé pour booter.
- Après un scan des disques et avoir vérifié leur bonne prise en charge, on peut poursuivre l'installation. Là, les disques sont partitionnés, initialisés et les arrays RAID sont construits. Le firmware est ensuite installé depuis la flash sur **md0**.

- Le système redémarre une dernière fois et présente, enfin, l'interface web de configuration et d'initialisation comme après un reset d'usine.



Vous pourrez alors entièrement configurer le NAS, de ses adresses IP aux comptes utilisateurs en passant par le type de RAID utilisé et les partages à créer. L'avantage de cette mise à jour est de permettre la gestion des disques de grande capacité, l'amélioration des performances et surtout, en ce qui nous concerne, la corrélation entre les sources en GPL fournies par Intel et le firmware en partie propriétaire. Après cette mise à jour, on se rend compte que le seul intérêt dans l'utilisation d'une image Intel tient dans le fait que celle-ci est plus facile à trouver que celle de LaCie. Les logos apparaissant sur les différentes pages web de l'interface laissent clairement penser qu'il n'existe qu'un seul firmware pour ce périphérique, qu'il s'appelle LaCie Ethernet RAID ou SS4000-E. Mais il fallait l'installer pour le savoir...

3 Exploration plus en profondeur

Les scripts de boot et d'init sont relativement conséquents et regroupent énormément de choses, de la configuration des disques au lancement de services. La majorité du travail est fait dans **/usr/local/ipstor**. A titre d'exemple, le sous-répertoire **bin/** comprend quelques 13000 lignes de scripts shell stockées dans quelques 80 scripts. A cela, il faut ajouter les binaires tantôt open source (**smbd** et autres), tantôt propriétaire (**ipstor***). Fouiller dans cette masse relève du

masochisme, mais on est relativement heureux de constater que les choses sont bien séparées par ailleurs. Les « produits toxiques » (propriétaires) semblent confinés dans **/usr/local**.

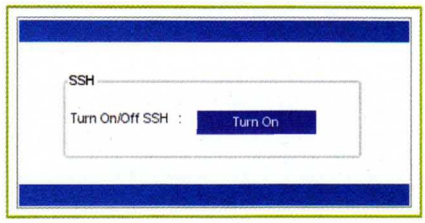
C'est en regardant les sources GPL fournies par Intel qu'on en apprend un peu plus. C'est le cas, par exemple, du binaire **/fs/hwctrl** qui nous permet d'accéder depuis l'espace utilisateur à un ensemble de ressources matérielles :

```
# /fs/hwctrl -fan read
FAN speed = 1748 rpm

# /fs/hwctrl -temp 1
Temperature 1 (IOP) = 51 deg.C
# /fs/hwctrl -temp 2
Temperature 2 (SATA disk) = 31 deg.C
# /fs/hwctrl -temp 3
Temperature 3 (SATA disk) = 34 deg.C

# /fs/hwctrl -getrtc
07171311201l.23
```

Nous avons donc accès aux informations sur le ventilateur, la température relevée par plusieurs sondes, le contrôle de la RTC mais également des leds en façade ou encore la possibilité de changer les adresses MAC des deux interfaces Ethernet e1000 Intel.

En fouillant davantage, on se rend rapidement compte qu'effectivement, il y a une séparation claire entre le résultat du travail d'Intel et celui de FalconStor et que, de ce fait, nous sommes très limités dans nos options. On notera d'ailleurs que le fichier **readme.txt** livré avec les sources apporte une information complémentaire pleine de sous-entendus : *This archive does not include certain source code from software vendor FalconStor, which Intel was unable to obtain from FalconStor for this posting. If interested please contact FalconStor - www.falconstor.com for more information.* Ce qui pourrait se traduire et être interprété comme « Voici ce qu'on peut vous donner. On aurait bien aimé faire plus et on a demandé à FalconStor, mais ils ne veulent pas. Débrouillez-vous avec eux ! ».

A ce stade, il est clair qu'obtenir un firmware débarrassé des éléments propriétaires risque de devenir une toute autre aventure. Ceci dit, tout dépend de ce que vous souhaitez tirer du produit et ce à quoi vous le destinez. Si la question est d'avoir un simple accès distant pour apporter vos modifications et utiliser le NAS comme ce pour quoi il a été conçu, il existe une astuce toute simple. En fait, ce n'est pas même une faille mais une fonctionnalité cachée : pointez votre navigateur sur https://IP_DU_NAS/ssh_controlF.cgi.



Après activation, un simple **ssh root@IP_DU_NAS** en utilisant le mot de passe administrateur, tout comme sur la console série, et vous voilà connecté ! **scp** manque à l'appel et il faudra jongler avec **wget** et un serveur HTTP, par exemple, pour échanger des données, mais c'est un point d'entrée intéressant. Notez que cette page d'activation n'est pas spécifique à la version 1.4 du firmware mais existe, semble-t-il, depuis les premières versions.

Si vous souhaitez allez plus loin avec votre NAS, il faudra vous tourner vers un système GNU/Linux plus « pur » et cela tombe bien, il existe...

4 Debian ?

Eh oui, il est parfaitement possible d'installer une distribution Debian Armel sur ce type de NAS. Le principe de base pour l'installation consiste à utiliser les fonctionnalités du bootloader pour charger un noyau Linux et une image RAM (initrd/initramfs) contenant l'installateur Debian. Le tout via la console série. Notez qu'il existe un **.pkg** disponible mais que, semble-t-il, celui-ci ne fonctionne pas ou plus en raison d'un problème de build automatique. Quoi qu'il en soit, il est toujours possible de se passer de l'installation via l'interface web mais cela suppose que vous ayez une console série. Comme les développeurs Debian, nous partons du principe que, si vous en êtes à ce stade (installation manuelle d'une distribution), vous sortez clairement de l'utilisation classique du NAS en question. De ce fait, il vous est impératif d'avoir un accès console et vous devez savoir comment l'obtenir.

Pour démarrer l'installation, il vous faudra un serveur HTTP. C'est la méthode la plus simple mais pas la seule. Il est, en effet, possible d'uploader dans la mémoire du NAS les images noyau et initrd via YMODEM. Ceci étant relativement pénible et fortement dépendant du client que vous utilisez en guise d'émulateur de console série, nous préférons ici la méthode HTTP. Tout se joue dans RedBoot, le bootloader pré-installé.

Au démarrage du NAS, guettez le message de RedBoot et utilisez CTRL+C. Vous n'avez qu'une seconde, il faut être vif :

```
+No network interfaces found

EM-7210 ver.T04 2005-12-12 (For ver.AA)
== Executing boot script in 1.000 seconds -
enter ^C to abort
^C
RedBoot>
```

Sur la ligne de commandes de RedBoot, utilisez les instructions suivantes pour basculer le bootloader en mode RAM :

```
RedBoot> fis load rammode
RedBoot> g
```

Après validation de la commande **g** (*go*), le NAS semble redémarrer, mais ce n'est en réalité que le bootloader qui se recharge. Suite à cela, il faut à nouveau en stopper l'exécution avec CTRL+C :

```
+Ethernet eth0: MAC address 00:0e:0c:ea:1c:62
IP: 10.9.9.1/255.255.255.0, Gateway: 10.9.9.1
Default server: 10.9.9.10, DNS server IP: 0.0.0.0

EM-7210 (RAM mode) 2005-12-22
== Executing boot script in 1.000 seconds - enter
^C to abort
^C
RedBoot>
```

Notez la subtile différence dans l'affichage, la mention **RAM mode** ainsi

que l'activation de l'interface Ethernet. A ce moment, vous pouvez configurer le réseau :

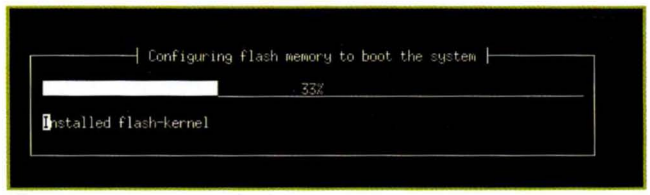
```
RedBoot> ip_address -l 192.168.0.200 -h 192.168.0.2
IP: 192.168.0.200/255.255.255.0, Gateway: 10.9.9.1
Default server: 192.168.0.2, DNS server IP: 0.0.0.0
```

Nous avons défini ici à 192.168.0.200 l'adresse du NAS et à 192.168.0.2 l'adresse du serveur par défaut qui fait fonctionner le serveur HTTP. Ce dernier accueille les fichiers **initrd.gz** et **zImage** récupérés depuis http://http.us.debian.org/debian/dists/squeeze/main/installer-armel/current/images/iop32x/netboot/ss4000e/ et placés à la racine du serveur web. Il ne nous reste plus qu'à charger les deux fichiers en mémoire et à passer la main au noyeau :

```
RedBoot> load -v -r -b 0x01800000 -m http /initrd.gz
----------------------------------------
----------------------------------------
----------------------
Raw file loaded 0x01800000-0x01abeb73, assumed entry at 0x01800000

RedBoot> load -v -r -b 0x01008000 -m http /zImage
----------------------------------------
----------------------
Raw file loaded 0x01008000-0x0113a8a7, assumed entry at 0x01008000

RedBoot> exec -c "console=ttyS0,115200 rw root=/dev/ram
mem=256M@0xa0000000" -r 0x01800000
```

Notez que l'affichage risque d'être perturbé par la barre de progression du téléchargement, mais ceci ne pose pas de problèmes autres qu'esthétiques. Une fois la commande **exec** validée, le système démarre et la procédure d'installation (super laide et lente en raison de la console série) débute. Nous disposons de quatre disques dans le NAS, ce qui nous laisse une grande liberté de configuration. Notez toutefois qu'en raison de la présence d'array RAID, il convient de faire le ménage avant partitionnement. En effet, l'installateur Debian prend en charge la présence de RAID sur un système en cours d'installation et il n'est pas possible de reformater ou modifier une partition si celle-ci est prise en charge par un RAID logiciel. Désassemblez alors la configuration RAID du firmware original dans la configuration des partitions de l'installeur Debian avant de vous attacher à reconfigurer le système. C'est là la seule difficulté de la procédure. Le reste n'est qu'une installation presque standard de distribution Debian. Il vous faudra cependant être un peu patient. En effet, à force d'utiliser chaque jour des machines avec 2, 4 ou 8 cœurs et autant de gigaoctets de mémoire, on en oublie qu'il n'y a pas si longtemps, 400 Mhz et 256 Mo de RAM était une configuration respectable...

Le processus d'installation passe par toutes les étapes habituelles, de la création des comptes au choix des paquets, en passant par le partitionnement et en finissant par l'installation des éléments nécessaires au démarrage. Pas de Grub ou de LILO ici, l'image du noyau et l'initramfs sont placés comme il se doit en flash. Malheureusement, une première



Le système d'installation Debian se charge, en fin de processus, de placer le noyau et l'image initramfs en mémoire flash.

tentative de démarrage se soldera par le chargement et la décompression du noyau et puis... plus rien. Nous n'avons pas configuré RedBoot pour l'installation Debian et bien que les images en flash soient les bonnes, ce n'est pas suffisant. L'installeur Debian ne l'a pas fait à notre place. Nous redémarrons donc le NAS et guettons le moment d'utiliser CTRL+C pour obtenir une invite du bootloader. Nous pouvons alors reconfigurer la séquence de démarrage ainsi :

```
RedBoot> fconfig
Run script at boot: true
Boot script:
.. fis load ramdisk.gz
.. fis load zImage
.. exec
Enter script, terminate with empty line
>> fis load ramdisk.gz
>> fis load zImage
>> exec -c "console=ttyS0,115200 rw root=/dev/ram
mem=256M@0xa0000000" -r 0x01800000
>>
Boot script timeout (1000ms resolution): 3
Use BOOTP for network configuration: false
Gateway IP address: 192.168.0.1
Local IP address: 192.168.0.200
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.0.2
Console baud rate: 115200
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from 0xf1fc0000-0xf1fc1000: .
... Erase from 0xf1fc0000-0xf1fc1000: .
... Program from 0x0ffd2000-0x0ffd3000 at 0xf1fc0000: .
... Lock from 0xf1fc0000-0xf1fc1000: .
```

Nous ne changeons pas grand chose si ce n'est la ligne **exec** en **exec -c "console=ttyS0,115200 rw root=/dev/ram mem=256M@0xa0000000" -r 0x01800000**, identique à celle utilisée pour démarrer l'installation. Nous en profitons également pour allonger le délai permettant d'interrompre la séquence de démarrage, de une seconde à trois. La configuration est stockée en flash dans une zone spécifique. Les lecteurs ayant lu les articles concernant la Fonera dans GLMF ne seront pas dépaysés puisqu'il s'agit du même bootloader dans une configuration presque équivalente. Un reset et nous voilà en train de contempler le démarrage de notre NAS sous Debian se soldant par l'invite de login.

Rien de bien particulier ici, il s'agit d'une classique Debian. Après une petite mise à jour de la liste des paquets avec **apt-get update** et l'installation des éléments indispensables comme **vim** ou **screen**, nous avons tout loisir de personnaliser cette installation. Par la même occasion, nous en profitons pour vérifier l'état des disques avec **smartctl** du paquet **smartmontools**. Ici le **/dev/sdc** d'origine nous annonce tristement un **Reallocated_Sector_Ct** de 2. Ce n'est pas bon du tout et un signe avant-coureur d'une perte de données. Le disque sera donc tout simplement éliminé de l'installation ou remplacé (plus tard).

Nous disposons de deux interfaces réseaux parfaitement prises en charge, de quatre emplacements de disques SATA et d'un ensemble de paquets intéressants. De quoi faire de notre NAS, une passerelle/firewall Internet (bruyante), une machine d'archivage de logs ou... un NAS aux « petits oignons » (NFS4, FTP, rsync, Unison, etc). Bien plus que ne pourra jamais faire le firmware d'un constructeur. Accessoirement, nous avons un plein accès aux deux ports USB 2.0 avec tous les drivers supportés par notre linux 2.6.32. Ceci signifie l'ajout d'interfaces réseaux USB, de disques, de ports séries, etc. Que du bonheur en perspective. Mais...

5 M'sieur ? Qu'est ce qui marche pas ?

La réponse est simple, en dehors des éléments gérés par le code propriétaire initial, nous n'avons absolument rien pour contrôler le matériel. Les leds en façade, par exemple, ne disposent pas de points d'entrées, pas plus que le contrôle de la température et du ventilateur. Mais ce n'est pas là quelque chose d'inaccessible. En regardant de plus près ce qui est mis à disposition dans les sources fournies par Intel, nous trouvons un répertoire **Firmware v1.4/vendors/hwctrl** contenant une archive. En plus des sources « classiques » GPL du système, nous avons ainsi accès non seulement aux sources de RedBoot pour ce NAS, mais aussi et surtout à des morceaux de code bien plus intéressants.

L'archive contient, entre autres choses, le fichier **hwctrl.c**. Celui-ci comporte un copyright de Lanner Electronics, mais aucune mention de licence. Comme il s'agit de sources déclarées open source par Intel et que d'autres fichiers, comme un module noyau d'accès aux GPIOs du même développeur, font mention explicitement de la licence GPL, on peut raisonnablement supposer qu'il s'agit d'un code libre. Un petit coup d'œil au **readme.txt** nous indique « Hardware control interface between Falconstor software and Linux kernel » suivi d'une liste d'options utilisables avec la commande **hwctrl** :

- Contrôle de ventilateur,
- Information sur les températures,
- Information sur les tensions,
- Contrôle des leds,
- Changement des adresses MAC des interfaces Ethernet,
- Contrôle de l'arrêt du système,
- Lecture/écriture de la RTC,
- Changement de l'Intel ID (??),
- et quelque options, semble-t-il, à ne pas utiliser.

Le NAS dispose d'un processeur raisonnablement cadencé et d'une mémoire vive suffisante. Nous prenons donc le parti de ne pas nous amuser à installer et configurer un système de compilation croisée sur un PC x86 à destination de l'ARM du NAS. Nous optons, tout simplement, pour l'installation d'un environnement de compilation complet sur le NAS directement via les paquets adéquats de la Debian (**build-essential** et quelques éléments supplémentaires).

Après copie et désarchivage de **hwctrl-intel-20060505.tgz** sur le NAS, on tente une première compilation qui affiche une liste complète d'avertissements à l'utilisation de l'option **-Wall** de **gcc**. Absence de prototypes d'une partie des fonctions, absence de fichiers headers et déclarations implicites ou encore fonctions qui « oublient » de retourner une valeur... c'est la foire au nightly-code. Le binaire est toutefois produit, mais son premier lancement avec **-fan read** échoue lamentablement, fichier manquant. Cette première difficulté est contournée facilement, c'est dans **hwctrl.h** que se trouve le problème en définissant **I2C_DEV** sur le mauvais pseudo-fichier **/dev**. Fort heureusement, le noyau de Debian a correctement détecté l'interface i2c intégrée, mais celle-ci est accessible via **/dev/i2c-0**. On en profite d'ailleurs pour estimer les difficultés à venir car le programme semble également avoir besoin de **/dev/gpiodrv** et **/dev/eepromdrv**.

Après correction du **.h** et recompilation, une partie des fonctionnalités sont présentes :

```
#./hwctrl -fan read
FAN speed = 2518 rpm

# ./hwctrl -temp 1
Temperature 1 (IOP) = 45 deg.C

# ./hwctrl -temp 2
Temperature 2 (SATA disk) = 28 deg.C

# ./hwctrl -temp 3
Temperature 3 (SATA disk) = 30 deg.C

# ./hwctrl -temp
Segmentation fault
```

Segfault ? Un coup d'œil sérieux aux sources est nécessaire. Et là, c'est la surprise ! La gestion des options est une véritable horreur. A croire que les développeurs de chez Lanner Electronics ne connaissent pas **getopt**. Un nettoyage sera

nécessaire pour ne pas laisser ceci dans notre beau NAS Debian. A titre d'exemple, histoire que vous soyez prévenu et évitiez l'arrêt cardiaque à l'ouverture du fichier, voici deux lignes qui se suivent dans la fonction **convert()** :

```
        char buf1[5];
        buf1[5]= 0;
```

Et je passe sur toute la moitié inférieure du source débutant par **/***. Je n'en dirai pas plus, constatez par vous-même. En ce qui concerne le contrôle des leds...

```
# ./hwctrl -ledon 1
(ERROR) open device /dev/gpiodrv failed
```

Le contrôle des leds utilise **/dev/gpiodrv** qui n'existe pas avec le noyau Debian. Heureusement pour nous, le pilote est inclus dans les sources fournies par Lanner Electronics, il s'agit des fichiers **gpio.c** et **gpio.mod.c** (ce dernier est généré automatiquement). On reconnaît là les sources d'un module additionnel parfaitement classique. Il nous suffit alors de copier **gpio.c** dans un répertoire et d'y adjoindre un **Makefile** adéquat. Le lecteur curieux aura sans doute remarqué qu'un **Makefile** est livré avec les sources et regroupe les instructions pour la compilation de l'ensemble. Cependant, il s'agit d'une configuration pour une compilation croisée (modules + utilitaires) et nous préférons, vu le peu de travail que cela demande, séparer clairement les éléments.

Voici notre **Makefile**, qui n'a rien d'exceptionnel :

```
KDIR= /lib/modules/$(shell uname -r)/build

obj-m   := gpio.o

all:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

install:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules_
install
        depmod -a

clean:
        rm -f *~
         $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

Les sources du noyau ne sont pas nécessaires, seul le paquet **linux-headers-2.6-iop32x** devra être installé pour fournir les fichiers présents dans **/lib/modules/2.6.32-5-iop32x/build/include**. On notera aussi, dans **gpio.c** l'inclusion de **hwctrl.h**. Mais un certain nombre de définitions sont absentes, comme **IOP321_GPOE** ou **IOP321_GPID**. Il s'agit des adresses des registres permettant le contrôle des E/S. Une petite recherche nous montre que ces définitions ne sont pas dans les sources vanilla de Linux 2.6.32. Pourtant, dans les sources du noyau 2.6.10 fournies par Intel, on trouve effectivement un fichier **include/asm-arm/arch-iop3xx/iop321.h** contenant tout ce qu'il nous faut, tout comme dans le 2.6.10 officiel. Nous pourrions donc copier les fichiers du 2.6.10 et ajouter simplement un **#include "iop321.h"** dans notre module. En y regardant de plus près, on constate que sur les 330 lignes du fichier, seules quelques-unes nous intéressent :

```
/* IOP321 chipset registers */
#define IOP321_VIRT_MEM_BASE 0xfeffe000  /* chip virtual mem address*/

#define IOP321_REG_ADDR(reg) (IOP321_VIRT_MEM_BASE | (reg))

/* General Purpose I/O Registers */
#define IOP321_GPOE       (volatile u32 *)IOP321_REG_ADDR(0x000007C4)
#define IOP321_GPID       (volatile u32 *)IOP321_REG_ADDR(0x000007C8)
#define IOP321_GPOD       (volatile u32 *)IOP321_REG_ADDR(0x000007CC)
```

Remarque

On notera un changement majeur entre les sources utilisées par Lanner/Intel/LaCie et celles du 2.6.32 de Debian. La définition des adresses est totalement différente. Bien entendu, nous utilisons ici des données du 2.6.10 pour éviter de trop toucher les sources du module **gpio.ko**. On remarquera cependant qu'il serait bon de réécrire ce pilote pour le faire correspondre aux dénominations/définitions actuelles.

Dans **/arch/arm/include/asm/hardware/iop3xx.h** :

```
#define IOP3XX_PERIPHERAL_VIRT_BASE    0xfeffe000
[...]
/* General Purpose I/O  */
#define IOP3XX_GPOE             (volatile u32 *)IOP3XX_GPIO_REG(0x0000)
#define IOP3XX_GPID             (volatile u32 *)IOP3XX_GPIO_REG(0x0004)
#define IOP3XX_GPOD             (volatile u32 *)IOP3XX_GPIO_REG(0x0008)
```

Et dans **arch/arm/mach-iop32x/include/mach/iop32x.h** :

```
#define IOP3XX_GPIO_REG(reg)   (IOP3XX_PERIPHERAL_VIRT_BASE + 0x07c4 + (reg))
```

Au final, les adresses sont les mêmes, **0xfeffe7c4**, **0xfeffe7c8**, **0xfeffe7cc** correspondant respectivement à GPOE (*Out Enable*), GPID ((*In Data*) et GPOD (*Out Data*).

Nous plaçons donc ce code dans un **myiop321.h** que nous incluons juste après **hwctrl.h**. Nous compilons gentiment le module avec :

```
% make
make -C /lib/modules/2.6.32-5-iop32x/build SUBDIRS=/home/denis/SRC/gpio8 modules
make[1]: entrant dans le répertoire " /usr/src/linux-headers-2.6.32-5-iop32x "
  CC [M]  /home/denis/SRC/gpio8/gpio.o
  Building modules, stage 2.
```

```
  MODPOST 1 modules
  CC      /home/denis/SRC/gpioB/gpio.mod.o
  LD [M]  /home/denis/SRC/gpioB/gpio.ko
make[1]: quittant le répertoire " /usr/src/linux-headers-2.6.32-5-
iop32x "
```

Et nous enchaînons sur un test de chargement :

```
% sudo insmod ./gpio.ko

% dmesg | tail
[...]
[151269.340000] gpiodrv: load, major number 211

% cat /proc/gpiobuttons
power_button:0
reset_button:0
```

Tout ceci semble être très satisfaisant, mais il nous faut encore créer l'entrée dans **/dev** avec ce bon vieux **mknod** :

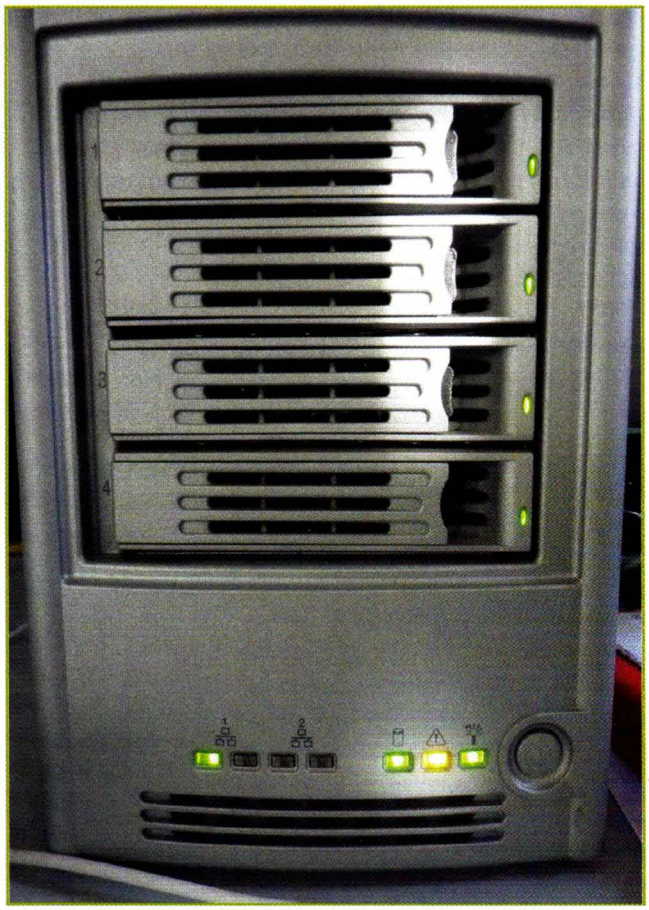
```
# mknod /dev/gpiodrv c 211 0
```

Notez que ce n'est pas vraiment nécessaire, étant donné que cela est fait par **hwctrl**. On va dire que je tire sur l'ambulance mais, quelque soit la personne responsable chez Intel, LaCie ou Lanner, elle est prudente, puisque la création des entrées **/dev** est également faite depuis **/fs/hwtool** appelé depuis le **/etc/init.d/rcS** du firmware original.

Enfin, nous essayons quelques options en rapport avec les leds en façade :

```
# ./hwctrl -gpioinit
SGPIO10...
SGPIO14...
SGPIO20...
SGPIO24...

# ./hwctrl -ledon 1 green
turn on led...
SGPIO10...

# ./hwctrl -ledon 2 amber
SGPIO14...

# ./hwctrl -ledon 3 amber blinkon
SGPIO20...
```

Et... *it works* ! Les leds des disques SATA sont numérotées de 1 à 4 et les leds 5 et 6 correspondent respectivement au témoin disque global et au voyant d'état du système. Nous avons donc, sous notre contrôle, 6 leds vertes et 6 leds oranges (ambre). Les états disponibles sont allumé, éteint ou clignotant avec une fréquence de 1 Hz. Attention, toutes les leds ne permettent pas tous les modes. On regrettera que le choix des leds de notification ne se soit pas porté sur le vert et le rouge, qui combinés, permettent d'obtenir trois couleurs (vert, rouge, orange/jaune). Mais, après tout, ce n'est l'affaire que de quelques minutes d'utilisation d'un bon fer à souder :)



Le NAS installé avec un système Debian offre un accès direct aux témoins à leds en façade moyennant quelques efforts.

Notez que l'option **-gpioinit** n'est pas décrite par le **readme.txt** mais présente dans le script **hwtool** dans la fonction **StartDriver()**, juste après la création des **/dev/***. Dernier point et non des moindres, l'activation des leds de chaque disque, couplée à l'utilisation de **hwctrl -hd n link** (où **n** est le numéro du disque) permet d'avoir un comportement standard témoignant de l'activité des disques (en mode inversé toutefois). C'est, je pense, le principal intérêt de toutes ces manipulations, bien qu'un système de notification complet soit également très bénéfique.

6 Module noyau Debian's style

. Compiler un module et le charger « à la hache » c'est bien pour vérifier que tout fonctionne. En revanche, si l'on considère que l'intégrité d'une distribution est importante (et c'est le cas), il est impératif de procéder à une intégration digne de ce nom et donc à la création et l'installation de l'élément

sous la forme d'un paquet. Encore une fois, nous avons de la chance de travailler avec des composants de qualité et des développeurs d'une distribution qui aiment les choses « propres ». Il nous suffit donc d'installer les paquets **dkms** et **debhelper**, puis de copier les sources du module dans un sous-répertoire de **/usr/src**.

Nous y ajoutons un fichier **dkms.conf** contenant :

```
PACKAGE_NAME="ss4400gpio"
PACKAGE_VERSION="1.0.0"
CLEAN="make clean"
BUILT_MODULE_NAME[0]="gpio"
DEST_MODULE_NAME[0]="gpio"
DEST_MODULE_LOCATION[0]="/updates"
AUTOINSTALL="yes"
```

Après quoi nous pouvons utiliser :

```
# dkms add -m ss4000gpio -v 1.0.0
Creating symlink /var/lib/dkms/ss4000gpio/1.0.0/source ->
                 /usr/src/ss4000gpio-1.0.0
DKMS: add Completed.
```

Ceci aura pour effet d'ajouter nos sources au système DKMS initialement développé par Dell pour faciliter l'intégration de modules annexes au développement principal du noyau. Nous nous empressons de vérifier la bonne marche des choses avec :

```
# dkms build -m ss4000gpio -v 1.0.0

Kernel preparation unnecessary for this kernel.  Skipping...

Building module:
cleaning build area......
make KERNELRELEASE=2.6.32-5-iop32x -C
  /lib/modules/2.6.32-5-iop32x/build
  M=/var/lib/dkms/ss4000gpio/1.0.0/build.........
cleaning build area......

DKMS: build Completed.
```

La construction avec DKMS se passe sans problème et nous pouvons enchaîner sur la production du fichier **.dsc** indispensable à la fabrication d'un paquet source Debian :

```
% dkms mkdsc -m ss4000gpio -v 1.0.0 --source-only
Using /etc/dkms/template-dkms-mkdsc
copying template...
modifying debian/changelog...
modifying debian/compat...
modifying debian/control...
modifying debian/copyright...
modifying debian/dirs...
modifying debian/postinst...
modifying debian/prerm...
modifying debian/README.Debian...
modifying debian/rules...
copying legacy postinstall template...
Copying source tree...
```

```
Building source package... dpkg-source --before-build ss4000gpio-dkms-1.0.0
debian/rules clean

dpkg-source -b ss4000gpio-dkms-1.0.0
dpkg-source: avertissement: aucun format source indiqué dans debian/source/format, voir dpkg-
source(1)
dpkg-genchanges -S >../ss4000gpio-dkms_1.0.0_source.changes
dpkg-genchanges: inclusion du code source original dans l'envoi (" upload ")
dpkg-source --after-build ss4000gpio-dkms-1.0.0

DKMS: mkdsc Completed.
Moving built files to /var/lib/dkms/ss4000gpio/1.0.0/dsc...
Cleaning up temporary files...
```

Nous réitérons la commande en remplaçant **mkdsc** par **mkdeb** pour produire un paquet (presque) binaire. Son installation avec **dpkg** provoquera la compilation adaptée pour le noyau en cours d'utilisation.

```
# dpkg -i ss4000gpio-dkms_1.0.0_all.deb
Sélection du paquet ss4000gpio-dkms précédemment désélectionné.
(Lecture de la base de données... 31139 fichiers et répertoires déjà installés.)
Dépaquetage de ss4000gpio-dkms (à partir de ss4000gpio-dkms_1.0.0_all.deb) ...
Paramétrage de ss4000gpio-dkms (1.0.0) ...
Loading new ss4000gpio-1.0.0 DKMS files...

Loading tarball for module: ss4000gpio / version: 1.0.0

Creating /var/lib/dkms/ss4000gpio/1.0.0/source
Copying dkms.conf to /var/lib/dkms/ss4000gpio/1.0.0/source...
Loading /var/lib/dkms/ss4000gpio/1.0.0/2.6.32-5-iop32x/armv5tel...

DKMS: ldtarball Completed.
First Installation: checking all kernels...
Building only for 2.6.32-5-iop32x
Building for architecture armel
Building initial module for 2.6.32-5-iop32x
Done.
gpio.ko:
Running module version sanity check.
- Original module
   - No original module exists within this kernel
- Installatio   - Installing to /lib/modules/2.6.32-5-iop32x/updates/dkms/

depmod.....
DKMS: install Completed.
```

Notre cher module **gpio.ko** est produit et ajouté dans **/lib/modules/2.6.32-5-iop32x/updates/dkms**. Le **depmod** permettant de régénérer le **modules.dep** ainsi que les fichiers *maps* termine la procédure. Si nous devions installer un nouveau noyau, la construction du module s'en trouvera grandement automatisée et simplifiée.

Réflexions et conclusion

Avec un peu de temps et d'huile de coude, il n'est pas impossible d'utiliser une *appliance* pour la transformer en un périphérique plus polyvalent tout en conservant la quasi-totalité des fonctionnalités matérielles initialement à disposition. Ceci, bien entendu, à condition que les informations soient disponibles. Ici, il semblerait que la base électronique du système soit celle de nombreux autres modèles de NAS, mais également de kits de développement. C'est là, sans doute, la raison de la disponibilité des sources qui nous auraient gravement fait défaut dans le cas contraire.

Mais cette petite aventure témoigne également de ce qui peut être l'une des raisons pour lesquelles une société ne souhaite pas diffuser son travail en open source. Ceux qui ont déjà eu affaire à des sources propriétaires savent de quoi je parle. Tout comme ceux, les *community managers*, responsables de la bonne marche vers une politique d'ouverture au sein de ces mêmes sociétés. En cachant à l'utilisateur averti les sources, celui-ci ne voit que les effets du syndrome « bah, ça marche, non ? », trop souvent présent dans un développement en circuit fermé. N'avez-vous jamais entendu quelqu'un dire que ses sources ne sont pas suffisamment « présentables » pour les ouvrir ? Moi si !

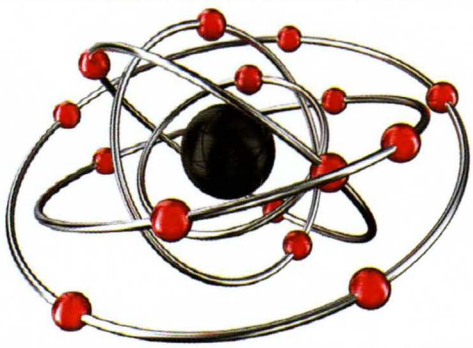
Pourtant, en dehors de l'image qui s'en trouve ainsi mise à mal, le jeu en vaut la chandelle. Car, même si **hwctrl.c** n'est vraiment pas beau à voir, il y a plein de développeurs de talent qui, en peu de temps, seraient ravis d'en faire un « beau morceau de code ». Pourquoi alors ne pas prendre les devants et, non pas simplement diffuser cela dans une archive brute, mais opter pour la création d'un Wiki et d'un Git par exemple ? Diffuser une première version des sources ainsi, au même moment que la mise en vente du produit et gérer une communauté de développeurs serait un gage de qualité pour une version suivante du firmware. Tout comme fournir des exemplaires « de développement » du produit à qui souhaite proposer une contribution. Ce sont là des leviers permettant non seulement une amélioration notable de la qualité et de la robustesse d'une appliance, mais également une excellente méthode de recrutement. Certains l'ont d'ores et déjà compris, il suffit de suivre l'exemple et maximiser les efforts dans ce sens. Tout le monde sera gagnant, des constructeurs aux développeurs, en passant par le marketing et surtout, surtout... par les clients. ■

N°141 SEPTEMBRE 2011

L 19275 - 141 - F: 6,50 €

GNU LINUX MAGAZINE / FRANCE

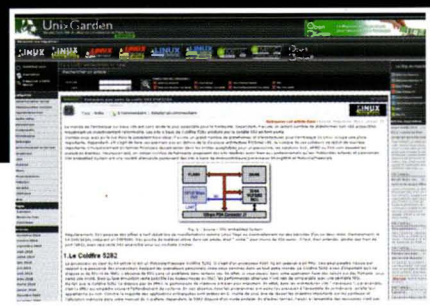
Administration et développement sur systèmes UNIX



France Métro : 6,50 € / DOM : 7 € / TOM Surface : 950 XPF / POL. A : 1400 XPF / CH : 13,80 CHF / BEL.PORT.CONT : 7,50 € / CAN : 13 $CAD / TUNISIE : 8,80 TND / MAR : 75 MAD

www.unixgarden.com

Récoltez l'actu UNIX et cultivez vos connaissances de l'Open Source !



Administration système

Utilitaires

Comprendre

Embarqué

Graphisme

Environnement de bureau

Bureautique

Audio-vidéo

Administration réseau

News

Distribution

Programmation

Sécurité

Agenda-Interview

Matériel

Web

Jeux

Réfléchir

UnixGarden