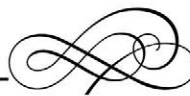
N°17 | MARS - AVRIL 2017



HACKABLE MAGAZINE

DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

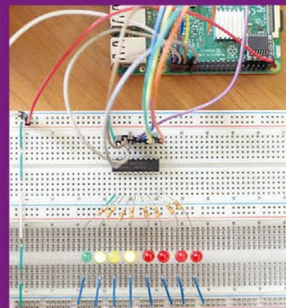
France MÉTRO. : 7,90 € – CH : 13 CHF – BEL/LUX/PORT.CONT : 8,90 € – DOM/TOM : 8,50 € – CAN : 14 $ CAD

PI / BLUETOOTH

Configurez simplement un clavier Bluetooth pour votre Raspberry Pi en ligne de commandes p. 60

PI / LEDS

Prenez le contrôle des leds de votre Pi et ajoutez-en autant qu'il vous chante p. 66



REPÈRES / SÉCURITÉ

Utilisez les codes tournants ou rolling codes pour sécuriser vos projets télécommandés p. 88

HACK / USB

Bidouillez l'Arduino UNO pour utiliser ses deux microcontrôleurs et profiter des fonctions USB p. 76

3 projets Arduino pour Créer vos périphériques USB ! p. 24

1 · Créez des boutons pour faire vos copier/coller

2 · Contrôlez le volume audio sous Windows/Linux/Mac/Android

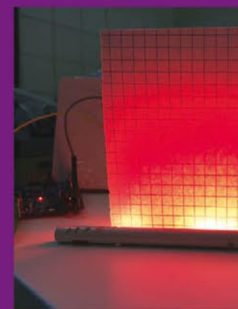
3 · Ajoutez l'USB à un vieux clavier de 30 ans

PI / EEPROM

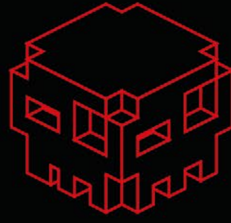
Manipulez tous types de mémoires avec votre Pi avec l'économique programmeur TL866cs p. 04

LED / ANIMATION

Créez un effet de flammes animées avec une matrice de leds WS2812b et une carte Arduino p. 14



* WARGAME * CONFERENCES * CHALLENGES * WORKSHOPS *



NUIT DU HACK

XV

24->25 Juin 2017
New York Hôtel Convention Center
Disneyland Paris
www.nuitduhack.com
@hackerzvoice

Ce document est la propriété exclusive de Alex Arnaud

ÉDITO

Personnaliser ça n'est pas important... c'est capital !

Peut-être qu'une telle affirmation vous parait évidente, mais ce n'est généralement pas le cas pour tout le monde. Ce que la plupart des gens ne réalisent souvent jamais durant toute leur vie est que la plupart des objets qui les entourent ne leur sont pas réellement adaptés ou destinés. Il ne s'agit que d'objets dont l'utilité, l'aspect ou les fonctionnalités sont génériques, adaptés non pas aux individus qui les possèdent, mais à une moyenne des préférences d'un ensemble, d'un groupe.

Cette illusion qui consiste à disposer d'objets, d'accessoires et d'outils qui semblent nous convenir n'est rien d'autre qu'une adaptation de nos personnes aux objets en question, alors que le principe naturel est précisément strictement opposé. Lorsqu'on y réfléchit un peu, on se rend compte que même lorsqu'on change, par exemple, sa vieille bouilloire pour une autre, on fait un choix qui est forcément biaisé. On choisit dans une gamme de produits qui finalement ne sont faits pour personne en particulier, et certainement pas pour nous-mêmes. En cela, la diversité du choix qui vous est proposé ne change absolument rien, il s'agit toujours de produits dont le but est de satisfaire un ensemble de personnes, mais, en même temps, qui sont voués par définition à ne jamais satisfaire personne totalement.

Pourtant, tout autour de nous, dans nos vies, nous avons ces objets que nous côtoyons régulièrement, quotidiennement. Chaque matin, j'utilise « ma » bouilloire pour mon petit-déjeuner (café, mails, café, GitHub, café, Twitter... Oui je sais « faut » manger le matin, mais je suis un rebelle), mais ce n'est pas réellement **ma** bouilloire, c'est un clone de celle de bien d'autres personnes. Du moins c'était le cas avant que je ne décide de changer les leds vertes qui l'éclairent pour des leds bleues. Là et uniquement là, c'est devenu MA bouilloire. La seule où j'ai changé ce petit élément et qui, même si d'autres personnes peuvent avoir eu la même idée, la seule où **moi** j'ai fait ce changement. Elle est maintenant unique... unique et vraiment mienne.

Peu importe la façon dont vous vous appropriez ces objets, qu'il s'agisse d'y faire des modifications complexes ou d'y coller simplement un autocollant, l'élément clé consiste à avoir une connexion avec ce qui vous entoure et ne pas finir par vivre dans un catalogue Ikea, dans un environnement qui finalement ne vous ressemble pas, n'est pas vraiment le vôtre, mais un ensemble de pièces génériques.

Le résultat de cette approche est non seulement de vivre dans un environnement qui vous est réellement familier, mais également de développer de plus en plus de capacités à adapter, transformer et créer des objets. Que cela commence par un coup de peinture, la couture d'un bout de tissu ou le changement de leds dans une bouilloire, importe peu, cela réveille votre créativité, votre capacité à changer les choses et à les adapter à vos besoins. Peut-être que le résultat laissera stoïque votre voisin ou vos amis, mais ces objets contrairement aux leurs, seront adaptés à vos préférences. Vous y aurez mis votre marque.

Sans doute suis-je en train de prêcher des convertis, mais vous êtes aussi les personnes les mieux placées pour rappeler autour de vous que cette capacité à ne pas se satisfaire de produits qui ne sont créés pour personne et qu'adapter, modifier et personnaliser est un don présent en chacun de nous, indubitablement propre à l'espèce humaine, et qui devrait s'exprimer en chaque instant, quel que soit le moyen, électronique ou non...

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

10, Place de la Cathédrale - 68000 Colmar
Tél. : 03 67 10 00 20 – Fax : 03 67 10 00 21
E-mail : lecteurs@hackable.fr
Service commercial : cial@ed-diamond.com
Sites : http://www.hackable.fr/
http://www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27 v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Landau, Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes : Abomarque : 09 53 15 21 77
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution,
N° ISSN : 2427-4631
Commission paritaire : K92470
Périodicité : bimestriel
Prix de vente : 7,90 €

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

Suivez-nous sur Twitter

À PROPOS DE HACKABLE...

HACKS, HACKERS & HACKABLE

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

SOMMAIRE

ÉQUIPEMENT

04
Utilisez un programmeur d'EPROM avec votre Raspberry Pi

ARDU'N'CO

14
Créez un effet « feu » avec une matrice de leds

EN COUVERTURE

24
Créez des boutons à copier/coller

30
Créez un contrôleur de volume pour votre ordinateur

44
Transformez un vieux matériel de 30 ans en clavier USB

EMBARQUÉ & INFORMATIQUE

60
Configurez un clavier Bluetooth pour votre Pi

66
Changez la configuration des leds de votre Raspberry Pi

DÉMONTAGE, HACKS & RÉCUP

76
Utilisez votre Arduino UNO comme périphérique USB

REPÈRE & SCIENCE

88
Les codes tournants ou comment ne pas envoyer le même message deux fois

ABONNEMENT

51/52
Abonnements multi-supports

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

UTILISEZ UN PROGRAMMEUR D'EPROM AVEC VOTRE RASPBERRY PI

Denis Bodor

Le titre de cet article ne fait pas honneur au matériel et au logiciel que je vais décrire ici. Bien au-delà de la lecture et l'écriture de simples mémoires comme les EPROM, UVPROM ou EEPROM, le périphérique que vous êtes sur le point de découvrir sait également gérer la mémoire des microcontrôleurs Atmel AVR, Microchip PIC ou encore de la famille 80C51/87C51/80C52/87C52 de plusieurs constructeurs, mais également tester les circuits logiques, manipuler les mémoires i2c, les mémoires SRAM, les GAL, etc.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Le Minipro TL866CS est de construction solide et se présente sous la forme d'un « pavé » de 65x100x25 millimètres. Il se connecte à un PC, un Mac ou une Raspberry Pi à l'aide d'un simple câble USB.

Le matériel/logiciel dont nous allons parler est en mesure d'utiliser plus de 11000 composants, en lecture, en écriture ou en test. C'est un outil polyvalent qui pourra vous être d'une grande aide dans bien des domaines même si c'est avant tout une solution pour lire et écrire des mémoires de divers types. La raison initiale de cette acquisition, en ce qui me concerne, est un projet de construction d'un ordinateur 8 bits à base de microprocesseur Zilog Z80 et de MOS 6502 par la suite (ou l'inverse, je ne suis pas décidé). Ce genre de projets implique généralement l'écriture de programmes en code machine (assemblé à la main SVP) qui sont alors stockés dans une mémoire EPROM ou EEPROM directement accédée par le processeur.

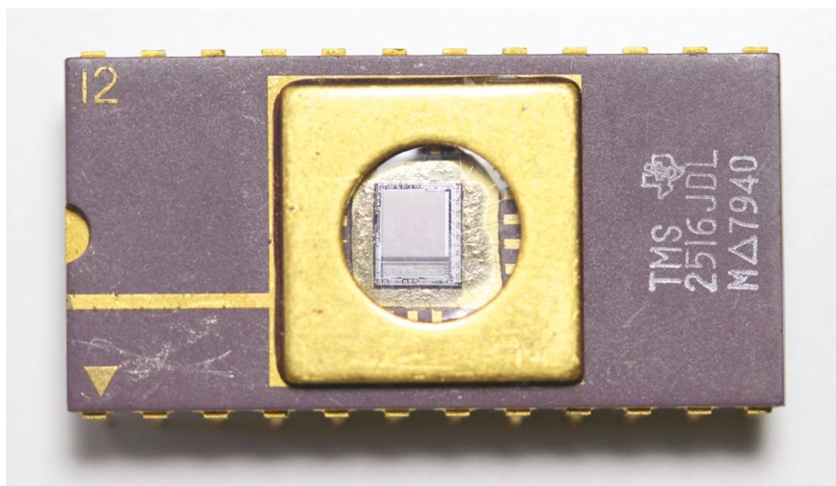
Ceci sort totalement du cadre du présent article, mais j'en reparlerai dans les pages du magazine dès lors que cela aura pris une tournure « présentable ». Quoi qu'il en soit, il me fallait un moyen de lire et écrire ce type de mémoire et je n'avais aucunement l'intention de jongler entre GNU/Linux (édition et développement) et Windows (enregistrement) pour ce genre de choses. Il me fallait donc un programmeur utilisable sous Debian GNU/Linux et par conséquent forcément compatible Raspbian. Finalement, j'ai trouvé bien plus que cela et, je pense, des choses qui pourront également vous servir, vous faire faire des économies et surtout, vous sortir du pétrin dans certains cas.

En effet, le produit dont je vais vous parler n'est pas un simple programmeur de mémoires, mais il peut également lire et programmer des microcontrôleurs et en particulier les Atmel AVR qui sont à la base de la plupart des cartes Arduino. Il est ainsi possible, tout comme avec un programmeur AVR Dragon d'Atmel ou un montage à base d'Arduino (voir *Hackable n°8*), de sauvegarder le croquis binaire d'une carte Arduino, d'en faire des copies d'un AVR à un autre (du même modèle bien sûr) ou de tout simplement programmer des AVR sans carte Arduino. Ceci est également valable pour un autre type de microcontrôleur très populaire : le PIC de Microchip (compagnie qui a récemment racheté Atmel).

1. LE MATÉRIEL : LE PROGRAMMEUR TL866CS

Le matériel qui nous intéresse ici est le Minipro de XG Autoelectric, modèle TL866CS. Celui-ci est très facile à trouver auprès de revendeurs partout dans le monde, sur Amazon et bien entendu sur eBay. Il vous coûtera quelques 50€, plus ou moins quelques euros en fonction des accessoires optionnels fournis et des frais de port. J'ai acquis le mien sur eBay pour 45,96€ (+ 10€ de port) auprès d'un vendeur allemand appelé *womenfashion4evernew* (????).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Voici la plus vieille mémoire EPROM en ma possession : une TMS2616 de 2Ko fabriquée par Texas Instruments la 40ème semaine de 1979. Ce composant a plus de 35 ans et... il fonctionne toujours ! J'ai pu en lire le contenu sans le moindre problème avec le Minipro (l'écriture cependant n'est pas prise en charge).

Le TL866CS se présente sous la forme d'un boîtier plastique blanc relativement robuste équipé d'un support ZIF (*Zero Insertion Force*) 40 broches permettant d'enficher un composant et de le bloquer à l'aide d'un petit levier. Le matériel est livré avec son câble USB et un jeu d'adaptateurs permettant d'utiliser d'autres formats de composants que DIP : PLCC44, PLCC32, MSOP8/SSOP8 et SOIC8. Un CD d'installation ainsi qu'une pince d'extraction PLCC sont également fournis.

Le logiciel livré sur CD n'est pas forcément intéressant dans le cadre de cet article puisque l'objectif est avant tout une utilisation sur Raspberry Pi et/ou sur un PC sous GNU/Linux. De toute façon, même en cas d'utilisation sous Windows, il sera plus intéressant de récupérer directement la dernière version du logiciel sur le site officiel (**http://www.autoelectric.cn/en/TL866_main.html**). C'est à se demander pourquoi certains fabricants fournissent encore des CD...

La connexion du produit à la Pi ou un PC se résume au branchement du câble USB fournissant à la fois l'alimentation et le lien de communication pour piloter le périphérique.

Notez que le Minipro se décline en deux modèles : le TL866CS et le TL866A, plus cher et plus difficile à trouver. Le TL866CS est plus récent, mais moins complet. En effet, le modèle TL866A se distingue par la possibilité de programmer des composants via une interface ICSP en plus de l'interface parallèle via le support ZIF. L'ICSP est une connexion permettant la programmation *in situ* ou en d'autres termes celle de composants « en place » dans un circuit. Les cartes Arduino, par exemple, possèdent un connecteur 6 broches, destiné à la connexion d'un programmeur AVR pour accéder au microcontrôleur sans avoir à retirer le composant de son support.

Le modèle TL866A possède non seulement un connecteur supplémentaire, mais utilise également un firmware différent. Bien que possible, la transformation d'un TL866CS en TL866A, est une opération délicate et, bien entendu, illégale en principe. La programmation ICSP pourra être nécessaire pour les cartes Arduino Micro ou Leonardo par exemple, puisqu'il n'est pas possible de retirer le microcontrôleur de la carte. De la même façon, les microcontrôleurs PIC de plus de 40 broches ne pourront pas être placés sur le support ZIF et nécessiteront également l'utilisation de l'ICSP pour leur programmation. Néanmoins, étant donné le prix du modèle TL866A, qui est presque le double du modèle TL866CS, il sera plus judicieux et économique de simplement acheter un programmeur AVR (AVR ISP) ou PIC (PICkit) supplémentaire, ou tout simplement utiliser une carte Arduino en tant que programmeur ICSP comme nous l'avons fait dans *Hackable n°8*.

Note : Avant de traiter de l'utilisation du TL866CS avec Raspbian, je tiens à préciser qu'il est cependant de bon ton de procéder à une première installation, sous Windows. Il suffit pour cela de télécharger le logiciel sur le

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

site du fabricant et l'installer **avant de brancher le périphérique**, car l'outil installe également le pilote adéquat. Une fois cette étape franchie, il vous suffira de connecter le TL866CS et laisser Windows mettre en œuvre le pilote. Lors du premier lancement de l'application, celle-ci vous demandera alors de mettre à jour le firmware du périphérique et c'est précisément ce que nous cherchons à faire. Il n'existe pas, à ma connaissance, d'autre solution simple pour faire cette mise à jour autrement que via Windows et le logiciel officiel.

2. PRISE EN CHARGE DU TL866CS PAR LA PI

Lors du branchement du TL866CS à votre Raspberry Pi, celle-ci détectera effectivement la connexion du périphérique, mais aucun pilote ne sera utilisé. Ce produit n'apparaît ni comme un périphérique HID ni comme un port série, mais simplement comme un matériel USB non supporté par le noyau Linux. L'utilisation de la commande `dmesg` vous confirmera cependant la détection :

```
new full-speed USB device number 6 using dwc_otg
New USB device found, idVendor=04d8, idProduct=e11c
New USB device strings: Mfr=1, Product=2, SerialNumber=0
Product: MiniPro TL-866 Programmer
Manufacturer: www.autoelectric.com
```

Le logiciel Minipro officiel n'existe qu'en version Windows, il ne peut donc être utilisé sur la Pi. Mais un développeur du nom de Valentin Dudouyt, a décidé début 2014 de créer son propre outil pour prendre en charge ce matériel. Depuis, le logiciel a évolué et s'étoffe peu à peu de fonctionnalités. Celui-ci n'existe pas sous forme de paquet pour Raspbian, mais Valentin a très tôt ajouté les éléments nécessaires pour produire facilement un paquet Debian/Ubuntu. Comme vous le savez sans doute, Raspbian n'est rien d'autre d'une version de Debian GNU/Linux pour notre chère framboise, nous pouvons donc utiliser ce système de construction pour produire facilement un paquet.

Avant toutes choses, nous devons installer les éléments nous permettant de récupérer et compiler le logiciel de Valentin, appelé judicieusement `minipro` !

```
$ sudo apt-get install git build-essential \
debhelper libusb-1.0-0-dev
```

Ceci devrait suffire sur une installation « fraîche » de Raspbian, mais restez attentif aux messages d'erreurs dans la suite des opérations. Nous allons récupérer les sources du logiciel directement sur GitHub via la commande `git` :

```
$ mkdir DEB
$ cd DEB
$ git clone https://github.com/vdudouyt/minipro.git
Clonage dans 'minipro'... remote:
Counting objects: 513, done.
remote: Total 513 (delta 0), reused 0 (delta 0), pack-reused 513
Réception d'objets: 100% (513/513), 451.46 KiB | 0 bytes/s, fait.
Résolution des deltas: 100% (279/279), fait.
 Vérification de la connectivité... fait.
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Ceci devrait avoir pour effet de vous créer un répertoire **minipro**, clone du dépôt GitHub contenant les sources. Il nous suffit alors de nous placer dans ce répertoire et d'utiliser la commande **dpkg-buildpackage** permettant la création d'un paquet :

```
$ cd minipro
$ dpkg-buildpackage -b
dpkg-buildpackage: paquet source minipro
dpkg-buildpackage: version source 0.1-1
dpkg-buildpackage: distribution source unstable
[...]
dpkg-source --after-build minipro
dpkg-buildpackage: envoi d'un binaire seulement
  (aucune inclusion de code source)
```

À ce stade, **dpkg-buildpackage** va vérifier que tous les paquets nécessaires à la compilation sont présents. Normalement nos manipulations précédentes ont réglé ce point, mais il n'est pas impossible qu'en fonction de l'évolution de la distribution Raspbian, ceci ne soit plus le cas d'ici quelques versions. **dpkg-buildpackage** ne manquera cependant pas de vous signaler son mécontentement en précisant les paquets manquants, et il vous suffira de les installer avec **apt-get** avant de lancer à nouveau la construction.

Une fois la tâche de **dpkg-buildpackage** accomplie, il ne vous restera plus qu'à remonter d'un cran dans l'arborescence et de procéder à l'installation du paquet qui s'y trouve :

```
$ cd ..
$ sudo dpkg -i minipro_0.1-1_armhf.deb
```

Les commandes **minipro** et **miniprohex** sont désormais installées dans votre système, mais nous n'en avons pas fini avec la configuration. Le TL866CS ne dispose pas de pilote et l'outil que nous venons d'installer accède donc directement au périphérique. Ceci demande des permissions particulières, car seul le super-utilisateur **root** peut faire ce genre de choses.

Nous allons donc modifier la configuration du système de façon à préciser que les utilisateurs du groupe **adm**, dont l'utilisateur **pi** fait partie, peuvent accéder à ce nouveau périphérique en lecture comme en écriture. Pour cela, nous ajoutons un fichier **90-minipro.rules** dans le répertoire **/etc/udev/rules.d**, contenant (sur une ligne) :

```
SUBSYSTEM=="usb",ATTRS{idVendor}=="04d8",ATTRS{idProduct}=="e11c",
MODE="0660", GROUP="adm"
```

Ceci précise que, lors de sa connexion, le périphérique identifié par **04d8:0xe11c**, appartiendra au groupe **adm** et que les permissions seront ajustées pour que le propriétaire (**root**) et le groupe puisse y lire et écrire des données (**0660**). Il suffira alors de relancer le service **udev** avec la commande **sudo service udev restart** puis de débrancher/rebrancher le TL866CS.

Enfin, il peut être très intéressant d'installer un autre paquet en compagnie de **minipro** : **srecord**. Cet outil permet de manipuler des fichiers contenant des valeurs hexadécimales dans différents formats et, en compagnie de **miniprohex** (qui est en réalité un script shell), permettra de travailler avec des fichiers *Motorola S-Record* (alias srec) ou *Intel Hex* (alias ihex).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

3. UTILISONS NOTRE TL866CS

L'utilisation de **minipro** est relativement simple puisqu'il ne s'agit que d'un outil de lecture/écriture sur différents composants. En fonction du modèle placé sur le support ZIF, le périphérique se chargera d'automatiquement ajuster à la fois l'utilisation des broches et les tensions à mettre en œuvre. En effet, certains composants comme les EPROM effaçables par exposition aux UV nécessitent une tension particulière (12,7 V généralement) pour pouvoir procéder à l'écriture. Vous n'avez pas à vous soucier de ce genre de choses ici, puisqu'il vous suffit de spécifier le bon composant.

La liste des modèles supportés est conséquente, très très conséquente... Vous pouvez l'afficher en utilisant **minipro -l** mais, il faut l'avouer, défiler quelques 11551 lignes n'est pas très pratique. Heureusement, les outils en ligne de commandes sont là pour nous aider, et en particulier **grep**. Il suffit alors de rediriger la sortie de **minipro -l** vers **grep** avec un **|** (AltGr+6) et spécifier tout ou partie de la chaîne de caractères souhaitée. Exemple avec une mémoire flash Atmel AT29C010A de 1 mégabit :

```
$ minipro -l | grep AT29C01
AT29C010A
AT29C010A @PLCC32
AT29C010A @TSOP32
```

Le Minipro TL866CS est livré avec un certain nombre d'adaptateurs permettant de placer, sur le support ZIF, des composants qui ne sont pas au format DIP. La pince d'extraction PLCC sur la droite est de piètre qualité, mais fait le travail pour peu que l'on soit délicat. Un câble USB, non présent ici, est également livré avec le produit.

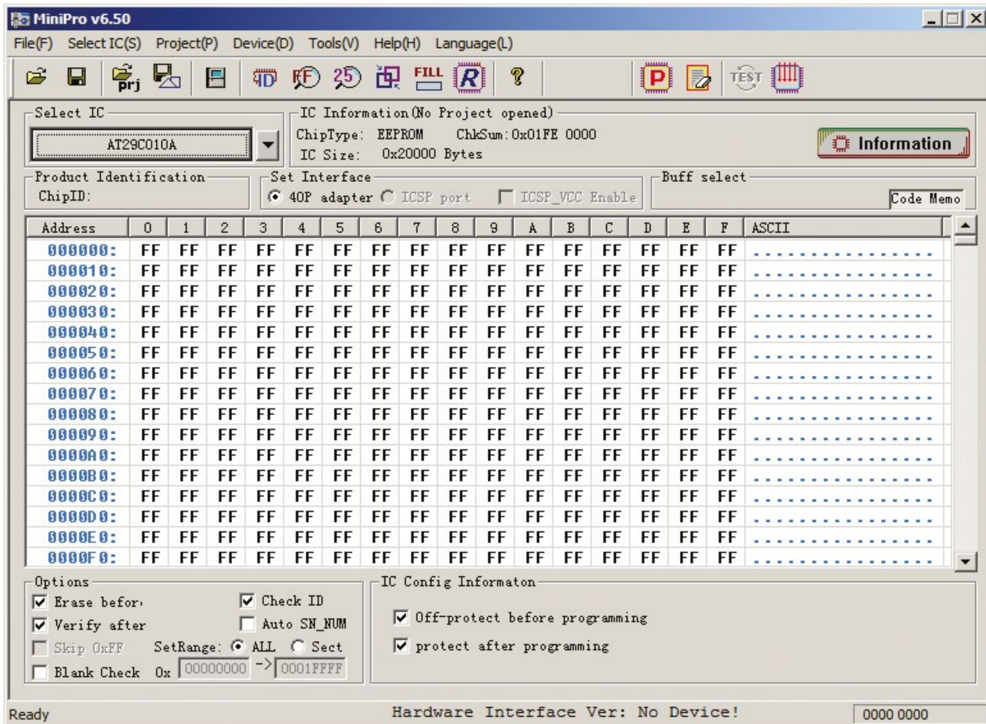
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Nous trouvons dans la liste, trois modèles utilisables, le premier au format DIP (avec des petites pattes adaptées aux platines à essais), un au format PLCC (carré avec les connecteurs en bordure) et un au format TSOP (pour un montage en surface avec des pattes au pas de 0,5 mm).

Pour lire le contenu de cette mémoire flash, nous utilisons alors `minopro` ainsi :

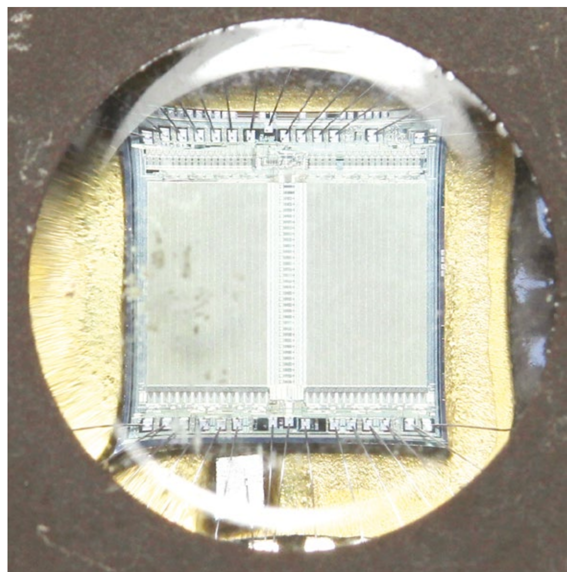
```
$ minipro -p "AT29C010A" -r flash.bin
Found Minipro TL866CS v03.2.72
Chip ID OK: 0x1fd5
Reading Code... OK
```

L'option `-p` permet de préciser le modèle de composant à utiliser. Notez la présence de guillemets qui est généralement une bonne habitude à prendre, car certains modèles possèdent un espace dans leur nom et, par exemple, `-p AT29C010A @PLCC32` ne fonctionnera donc pas alors que `-p "AT29C010A @PLCC32"` oui. L'option `-r` provoque la lecture du composant et elle est suivie d'un nom de fichier où seront alors placées les données obtenues. Notez la ligne `Chip ID OK`. Dans la mesure du possible, le TL866CS et `minipro` tenteront de vérifier que le composant que vous avez spécifié est bien celui se trouvant dans le support ZIF. La plupart des composants possèdent un identifiant qui peut être lu et comparé à celui présent dans `minipro` et, en cas de différence, un message d'erreur sera affiché. Vous pouvez cependant passer outre en forçant l'opération avec l'option `-y`, qui est, bien entendu, à n'utiliser que si vous savez très bien ce que vous faites.



Le logiciel officiel disponible au téléchargement sur le site du fabricant offre une interface graphique et un certain nombre de fonctionnalités supplémentaires par rapport à la version en ligne de commandes utilisable sous Linux. C'est aussi et surtout le seul moyen de mettre à jour le firmware dans le TL866CS...

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Gros plan sur la puce d'une mémoire EPROM exposée sur un composant AMD AM27256. On distingue les grandes zones de stockage, ainsi que les circuits de décodage d'adresse en bordure et les connexions reliant le circuit intégré aux pattes. Même sans en avoir l'usage, ces composants sont de magnifiques objets tout à fait captivants...

Certains composants, comme les mémoires EEPROM i2c par exemple, n'ont pas d'identifiant et aucune vérification ne sera effectuée :

```
$ minipro -p "AT24C256" -r eeprom.bin
Found Minipro TL866CS v03.2.72
Reading Code... OK
```

L'écriture se fera tout aussi facilement, en remplaçant l'option **-r** par **-w** et en spécifiant le fichier contenant les données à écrire :

```
$ minipro -p "AT24C256" -w eeprom.bin
Found Minipro TL866CS v03.2.72
Writing Code... OK
Reading Code... OK
Verification OK
```

Notez que l'écriture est suivie automatiquement d'une lecture afin de s'assurer que les données enregistrées soient effectivement celles présentes dans le fichier.

Maintenant, si nous prenons notre fichier **eeprom.bin** de 32768 octets et tentons d'en écrire les données sur la mémoire flash, nous obtenons une erreur :

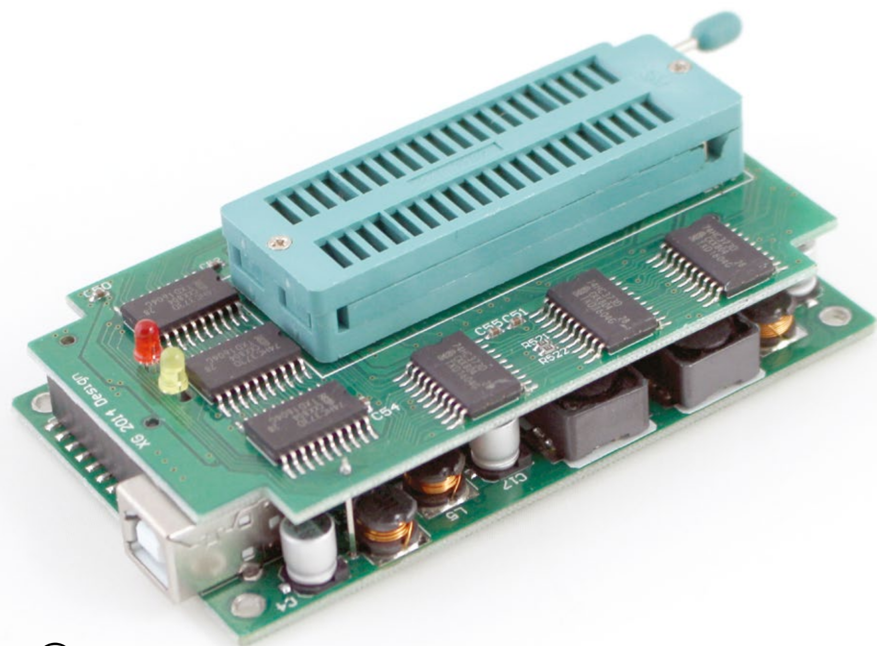
```
$ minipro -p "AT29C010A" -w eeprom.bin
Found Minipro TL866CS v03.2.72
Chip ID OK: 0x1fd5
Incorrect file size: 32768 (needed 131072)
```

En effet, **minipro** s'attend à avoir un fichier correspondant précisément à l'ensemble des données à écrire dans le composant. Ici, nous tentons de placer seulement 32768 octets dans une mémoire AT29C010A disposant d'un espace de stockage de 131072 octets. L'option **-s** nous permet de considérer cette erreur comme un simple avertissement. Un message apparaîtra effectivement pour signaler le problème, mais **minipro** s'exécutera malgré cela en n'écrivant que les données à sa disposition dans le composant :

```
$ minipro -p "AT29C010A" -w eeprom.bin -s
Found Minipro TL866CS v03.2.72
Chip ID OK: 0x1fd5
Warning: Incorrect file size: 32768 (needed 131072)
Writing Code... OK
Reading Code... OK
Verification OK
```

minipro utilise des fichiers au format binaire, c'est-à-dire que les données lues et enregistrées sont stockées « telle que » dans des fichiers. En utilisant cependant **miniprohex** en lieu et place, vous pourrez préciser un autre format en spécifiant simplement l'extension de fichier à utiliser : **.bin** pour binaire, **.hex** pour Intel Hex ou **.srec** pour le format Motorola S-record. **miniprohex** est en réalité un script qui lance la commande **srec_cat** pour convertir les données avant d'exécuter **minipro**.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



À la rédaction de Hackable, il y a des coutumes et des rites... Systématiquement démonter le matériel testé en fait partie et on découvre ici qu'on en a vraiment pour notre argent, la construction est propre et de qualité. Remarquez les nombreuses bobines sur la platine inférieure, faisant partie du circuit destiné à produire les tensions élevées pour la programmation de certains composants.

4. LE CAS DES MICROCONTRÔLEURS AVR

En plaçant un microcontrôleur Atmel AVR comme un ATmega328P d'une carte Arduino Uno dans le TL866CS, il est également possible d'en lire le contenu, exactement de la même manière qu'une mémoire de type EPROM ou flash :

```
$ miniprohex -p ATMEGA328P -r atmega.hex
Found Minipro TL866CS v03.2.72
Chip ID OK: 0x1e950f
Reading Code... OK
Reading Data... OK
Reading fuses... OK
```

Vous remarquez que la sortie affichée par `minipro` est sensiblement différente. Nous n'avons pas ici un simple fichier, mais trois : `atmega.hex` qui est le code contenu en flash, `eeprom.bin` qui contient les données dans l'EEPROM intégré au microcontrôleur et `fuses.conf` :

```
$ cat fuses.conf
fuses_lo = 0x0062
fuses_hi = 0x00df
fuses_ext = 0x00f9
lock_byte = 0x00ff
```

Les « fusibles » d'un AVR déterminent sa configuration : s'il doit utiliser un bootloader comme celui d'Arduino, sa source d'horloge, la protection de la mémoire, la capacité de reset en cas de chute de tension, etc. Tous ces éléments sont généralement gérés pour vous avec Arduino mais là, on parle de programmation de microcontrôleur AVR. `minipro` vous permet de récupérer tout ce qui se trouve dans la mémoire de l'AVR, de façon à en faire une sauvegarde ou permettre la copie sur un autre AVR (même si vous avez perdu le croquis original par exemple).

Les dernières versions de l'environnement Arduino permettent également d'obtenir une version compilée d'un croquis sous la forme d'un fichier Intel Hex, via le menu ***Croquis*** puis ***Exporter les binaires compilés***. Vous retrouverez alors, dans le répertoire du croquis en question, les fichiers dont les noms se terminent avec `.ino.standard.hex` et `.ino.with_bootloader.standard.hex`. Respectivement, une version avec juste votre croquis et une version avec le bootloader en plus, placé en début du fichier. Ce dernier pourra être utilisé avec `miniprohex` ainsi :

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
$ miniprohex -p ATMEGA328P \
  -w ~/croquis.ino.with_bootloader.standard.hex
Found Minipro TL866CS v03.2.72
Chip ID OK: 0x1e950f
Writing Code... OK
Reading Code... OK
Verification OK
```

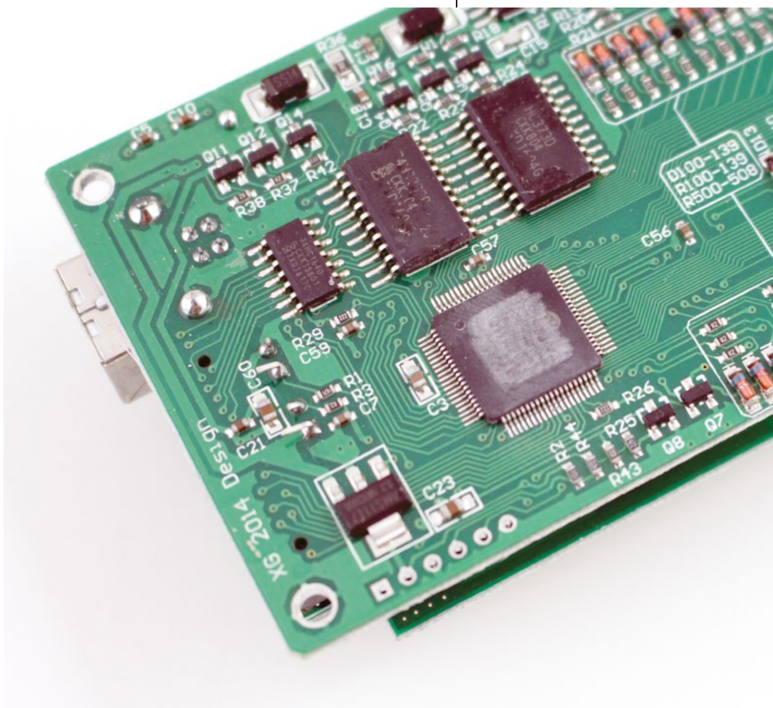
Le TL866CS pourra donc vous servir de programmeur pour vos microcontrôleurs (AVR ou autre) mais, il faut l'avouer, un matériel et un logiciel dédié sont généralement plus adaptés (**avrdude** par exemple). Ceci pourra cependant vous sortir du pétrin dans certaines situations délicates. En ce qui me concerne, je vois davantage dans le TL866CS une solution pour manipuler les mémoires que les microcontrôleurs, même si cela pourrait m'être effectivement utile un jour.

Chose assez classique dans ce genre de produits, la sérigraphie du composant « intelligent » dans le matériel a été effacée. D'après certains utilisateurs ayant expérimenté plus avant, il s'agirait d'un PIC18F87J50 de Microchip. On trouve sur le Web différentes techniques pour transformer un TL866Cs en TL866A en reflashant ce microcontrôleur. Personnellement, je n'en vois pas l'intérêt étant donné le bas coût des programmeurs ICSP...

5. POUR FINIR

Tout le monde n'a pas forcément l'usage d'un tel matériel, mais je pense qu'il constitue un élément important d'une boîte à outils électronique qui se veut complète.

Les mémoires EPROM et EEPROM parallèles sont de moins en moins utilisées au bénéfice des mémoires flash et EEPROM SPI ou i2c. Il y a cependant des domaines où ce type de matériel est tout simplement indispensable comme la construction d'un ordinateur 8 bits (mon cas), les expérimentations avec des BIOS de PC alternatifs (Coreboot par exemple), l'extraction de ROM de vieilles consoles de jeux (Atari 2600, etc.) ou de bornes d'arcade, ou encore tout simplement pour le plaisir de jouer avec ces adorables EPROM dont on voit la puce par la petite lucarne permettant l'exposition aux UV...



Je n'ai pas parlé ici de l'environnement Mac, mais sachez que **minipro** peut également être compilé pour Mac OS X (ou macOS comme on dit depuis la version Sierra fin 2016). C'est d'ailleurs la seule solution permettant de nativement utiliser un TL866CS ou TL866A sur ce système. L'alternative consiste sinon à démarrer le Mac sous Windows ou à utiliser une machine virtuelle avec un système Windows. **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

CRÉEZ UN EFFET « FEU » AVEC UNE MATRICE DE LEDS

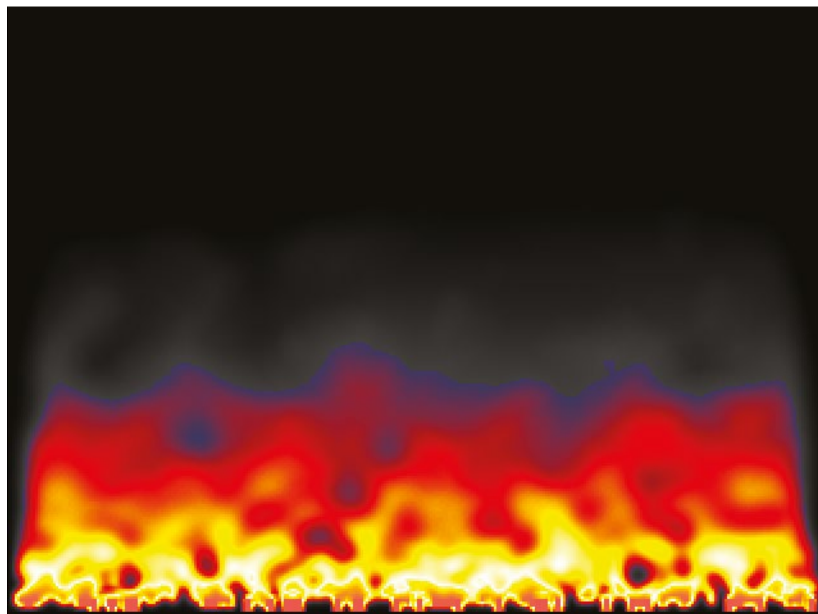
Denis Bodor

Les leds « intelligentes » de type WS2812b également appelées Neopixels permettent toutes sortes de réalisations. Plus il y en a, plus on peut obtenir des résultats attrayants et des effets attirant l'œil. Nous allons ici mettre en œuvre 64 de ces adorables composants, agencés en un carré de 8 par 8 qui fera office d'écran à très basse résolution. Notre objectif : simuler et animer un feu.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Les leds WS2812b ou similaires sont des petites merveilles. Elles se pilotent très simplement depuis une carte de type Arduino et peuvent prendre n'importe quelle couleur, mélange de rouge de vert et de bleu. Mieux encore, une seule broche, en plus de l'alimentation 5V et de la masse, suffit à en contrôler une seule ou des dizaines, connectées les unes aux autres en file indienne, la broche OUT de l'une reliée à la broche IN de la suivante et ainsi de suite. Ces composants sont disponibles à l'unité ou montés sur des circuits imprimés ou des rubans souples. Lignes, barres, rubans, anneaux, rectangle, carré... vous n'avez que l'embarras du choix.

Nous allons utiliser ici une matrice de ces leds ou plus exactement 64 d'entre elles, agencées en un carré de 8 par 8. Ce type d'organisation peut être facilement réalisé en utilisant des leds achetées individuellement ou encore avec des segments de rubans de 8 leds, mais il est encore plus facile de simplement acheter cela en module préfabriqué. Si le module est de qualité, vous évitez ainsi toute la phase de soudage des leds et les problèmes d'alignement qui l'accompagne. J'ai acheté mes modules sur eBay auprès d'un vendeur appelé « trekdu » pour environ 13€. Il est possible d'en trouver pour moins de 10€, mais le fait de choisir un vendeur français (sur Rennes) assure une livraison bien plus rapide que depuis Shenzhen (et j'étais pressé).



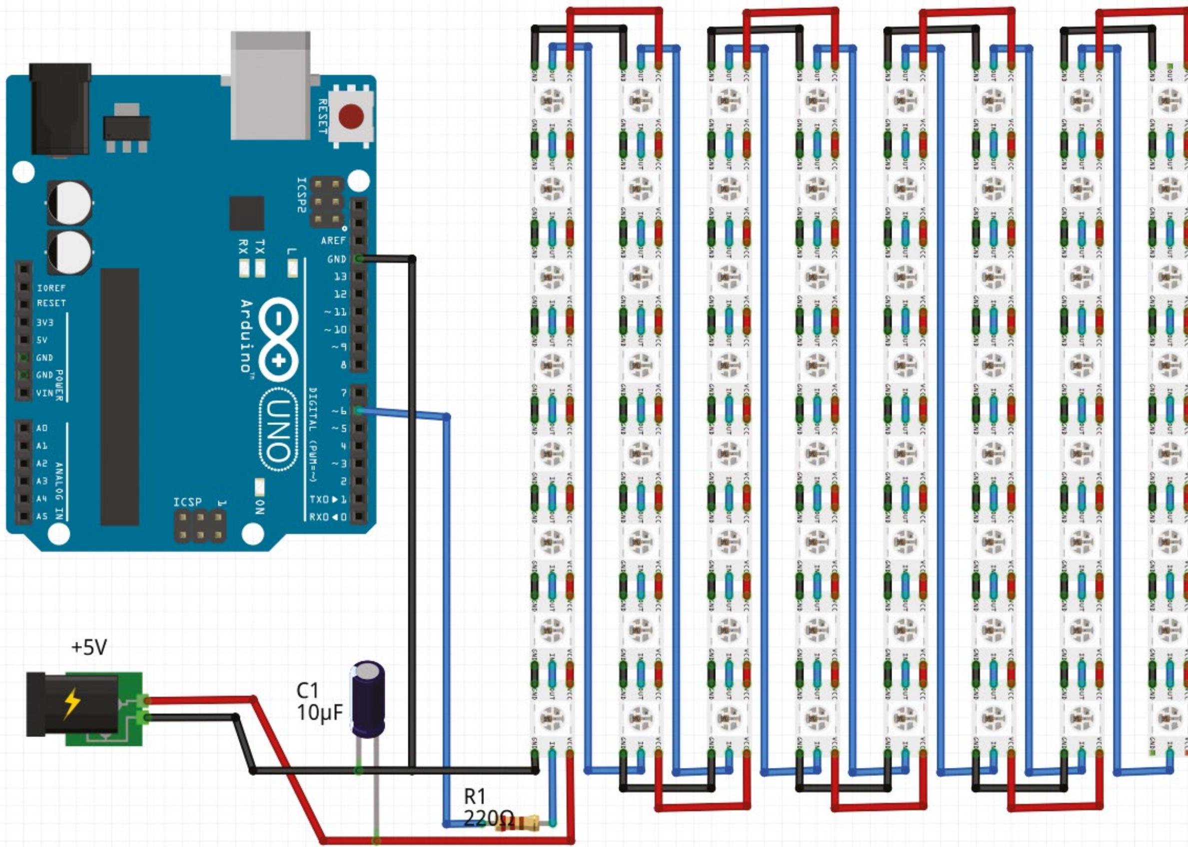
Un effet de feu ou de flamme tel que nous allons l'utiliser n'a rien de nouveau et est un exercice assez classique en termes de programmation graphique. Ici le résultat obtenu en 320×240 avec un programme en C reposant sur la bibliothèque SDL tel qu'on peut en trouver un peu partout sur le Web. Nous cherchons à obtenir un effet similaire, mais sur un carré de seulement 8 par 8 pixels/leds.

Avec une telle matrice, le nombre de réalisations possibles est presque illimité : affichage de texte, réalisation de jeux, VU-mètre, notifications diverses et variées, barres de progression, histogramme, icône, simulateur de TV (voir *Hackable n°9*), éclairage animé, veilleuse, simulateur d'aube, horloge... Mais en achetant ce module, j'avais une idée bien précise en tête : créer un effet de flamme ou de feu réaliste en utilisant un algorithme très classique et utilisé de longue date.

Sans doute avez-vous déjà vu cet effet, c'est un classique du genre, largement utilisé dans la scène démo où des programmeurs de talent réalisent des démonstrations (démos), généralement en utilisant des astuces techniques de pointe pour répondre à certaines contraintes imposées ou non. Cet effet est relativement simple à obtenir et il existe un grand nombre d'exemples dans différents langages de programmation, mais, le plus souvent, qui sont destinés à s'afficher dans une fenêtre sur un écran d'ordinateur. Ici, nous avons l'équivalent d'un écran, mais nous ne disposons que d'une résolution de 8 pixels par 8 pixels.

C'est donc une parfaite occasion de découvrir cet algorithme et de jouer avec. Nous en profiterons pour découvrir et utiliser une bibliothèque différente de celle que nous avons précédemment mise en œuvre dans les pages du magazine afin de piloter nos WS2812b.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Le montage ci-contre présente 64 WS2812b tels que généralement connectés sur un module 8×8. On prendra soin d'ajouter un condensateur entre 100µF et 1mF entre la tension d'alimentation et la masse au plus proche des leds, ainsi qu'une résistance de protection de 220 à 470 Ohms entre la carte Arduino et la broche DIN de la première led.

1. LE MONTAGE ET L'ALIMENTATION

Contrôler une ou plusieurs WS2812b avec un montage à base d'Arduino, ou toute autre carte de ce type, n'est pas très difficile. Une seule sortie de l'Arduino est utilisée, celle permettant d'envoyer les données d'affichage aux leds. Si, comme moi, vous optez pour un module tout fait, tout ce que vous avez à faire est de relier la broche de réception des données (In, DIN ou Data In) à une sortie de la carte Arduino, et le tour est joué.

Mais un point important est également à prendre en considération : l'alimentation des leds. Une WS2812b ou compatible demande une alimentation 5V stable et propre (entre 3,5 et 5,3 V pour être précis). Chaque composant contient un circuit intégré et trois leds haute luminosité (rouge, vert et bleu). En fonction de l'intensité des leds, le courant nécessaire pour chaque composant peut grimper jusqu'à 60 mA.

Ainsi, il peut être envisageable d'alimenter une poignée de WS2812b avec la broche 5V d'une carte Arduino (200 mA), mais certainement pas 64. Un rapide calcul nous indique que si toutes les leds sont allumées, notre alimentation devra être capable de fournir 64 x 0,06A, soit un peu moins de 4 ampères. Dans les faits cependant, et selon l'origine du bloc d'alimentation utilisé, mieux vaudra prévoir une marge de manœuvre. Une alimentation 4A pourra parfaitement faire l'affaire, mais personnellement j'ai préféré utiliser une 10A

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

(achetée il y a quelque temps déjà sur eBay auprès d'un vendeur de Guangzhou appelé « chinly2012 » à 12,80€ port gratuit). Rappelez-vous, contrairement à la tension, le courant délivré par une alimentation est un maximum qui peut être fourni, le montage n'utilisera que le courant dont il a besoin.

2. EXIT ADAFRUIT_NEOPIXEL ET VIVE FASTLED !

Dans les articles précédents traitant des leds WS2812b nous avons systématiquement utilisé la bibliothèque NeoPixel développée par Adafruit Industries pour ses propres produits et modules. Cette bibliothèque est simple à utiliser, mais comporte quelques problèmes parmi lesquels le manque de compatibilité en dehors des plateformes Arduino, un code très peu optimisé ou encore une gamme réduite de composants utilisables.

En effet, le NeoPixel d'Adafruit n'est autre qu'une simple désignation commerciale pour des leds WS2812b, mais il existe bien d'autres composants fournissant le même type de fonctionnalités : APA102, lpd8806, P9813, tm1809, UCS1903, SM16716, SK6822... Certains fonctionnent comme le WS2812b et d'autres utilisent un signal d'horloge supplémentaire (ce qui évite la mise en œuvre d'interruptions et donc permet d'utiliser la communication série, les servomoteurs ou encore les récepteurs IR sans problème, en même temps que les leds).

FastLED est une autre bibliothèque permettant d'utiliser les WS2812b ainsi que tous les modèles de leds « intelligentes » cités précédemment. Initialement appelée *FastSPI_LED*, la bibliothèque FastLED est l'œuvre de Daniel Garcia et Mark Kriegsman. Vous pouvez l'installer en version 3.1.0 directement depuis le gestionnaire de bibliothèques de l'environnement Arduino.

Pour un peu plus d'une dizaine d'euros on trouve des ensembles de leds WS2812b déjà montés sous diverses formes avec comme ici un carré de 64 leds. Notez la présence de condensateurs de filtrage à côté de chaque led, signe de qualité pour ce type de module.

FastLED est non seulement très optimisé, mais supporte énormément de modèles différents de composants. De plus, cette bibliothèque est disponible pour toutes les cartes Arduino (Due incluse), mais également pour les cartes à base d'ATTiny (Trinket par exemple), les Teensy 2 et 3, RFDuino, SparkCore, Photon et ESP8266/NodeMCU.

Ce sont là des caractéristiques intéressantes, mais ce n'est pas tout. FastLED intègre également des fonctionnalités très pratiques qui vont au-delà du simple pilotage des composants :

- fonctions mathématiques optimisées jusqu'à 10 fois plus rapides que leurs équivalents Arduino ;
- gestion de couleurs TSV (Teinte/Saturation/Valeur) et RVB (Rouge/Vert/Bleu), vous n'avez plus besoin d'écrire votre fonction de conversion ;
- gestion de l'intensité non destructive, les couleurs souhaitées ne sont pas impactées par l'intensité globale des leds ;
- correction de couleurs, vous pouvez obtenir un vrai blanc en spécifiant des valeurs RVB maximum ;
- manipulation de segments de leds ;
- gestion de palettes de couleurs ;
- etc.

Pour vous donner une petite idée de ce qu'est capable de faire FastLED, voici un petit exemple :

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
#include <FastLED.h>

CRGBArray<64> leds;

uint8_t teinte;

void setup() {
  FastLED.addLeds<WS2812B, 6, GRB>(leds,64).setCorrection(TypicalLEDStrip);
  FastLED.setBrightness(100);
}

void loop() {
  leds.fill_rainbow(teinte++);
  FastLED.delay(10);
}
```

Ces quelques lignes permettent d'obtenir un dégradé arc-en-ciel se déplaçant doucement le long des 64 leds connectées à la broche 6 de la carte Arduino. Le tout avec une luminosité réduite et une correction de couleurs adaptée... en moins de 20 lignes de code ! Notez cependant que ce croquis n'est pas typique de l'utilisation de FastLED et repose sur des fonctionnalités en développement (donc susceptibles de changer dans un avenir plus ou moins proche).

3. L'ALGORITHME DE SIMULATION DE FEU

Précisons-le de suite, il n'y a pas une seule et unique façon d'obtenir l'effet recherché, mais une pléthore de solutions et d'algorithmes. De plus, celui que nous allons mettre en œuvre est suffisamment simple pour être rapidement utilisé, tout en offrant l'opportunité d'y faire des modifications qui, selon le résultat recherché pourront être très intéressantes. Le principe cependant est toujours le même : placer une série de pixels au hasard puis, sur cette base, propager les changements du bas vers le haut.

C'est la façon de propager les changements qui crée le rendu que nous souhaitons obtenir. Pour mettre en place cette logique, nous travaillerons avec un tableau de 8 par 8 cellules correspondant aux valeurs que peut prendre chaque pixel. Il ne s'agit pas ici de couleurs, mais d'une valeur sur 8 bits (`uint8_t`), entre 0 et 255, indiquant la « température » du pixel (0 pour froid, 255 pour brûlant). Ce tableau représentera notre grille de pixels qui sera ensuite traduite en couleurs à afficher sur le module.

L'algorithme en lui-même n'a rien de bien complexe puisqu'après avoir placé quelques pixels chauds au bas du tableau nous procèderons comme suit :

- pour chaque pixel de la ligne juste au-dessus de la dernière, additionner la valeur des trois pixels immédiatement inférieurs et diviser par 4 ;
- s'il s'agit du premier pixel et qu'il n'y a donc pas de valeur inférieure gauche, utiliser le dernier pixel de la ligne inférieure (le 8ème) ;
- s'il s'agit du dernier pixel, il n'y a pas de valeur inférieure droite, utiliser le premier pixel de la ligne inférieure ;

Une masse, une alimentation 5V et une broche pour envoyer les données, il n'en faut pas plus pour piloter des dizaines de leds WS2812b ou similaires... à condition, bien sûr, de disposer d'une alimentation capable de fournir le courant nécessaire.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

- rafraîchir l'affichage sur la base de ces nouvelles valeurs du tableau ;
- passer à la ligne supérieure et recommencer l'opération pour chaque pixel, et ainsi de suite ;
- une fois arrivé à la ligne la plus en haut, recommencer tout le processus après avoir ajouté des pixels aléatoires au bas du tableau.

L'opération consiste donc à démarrer avec un jeu de pixels de valeurs aléatoirement choisies. Ceci peut être vu comme les braises de notre feu. Le fait de faire une moyenne des valeurs des pixels inférieurs pour calculer celle du pixel actuel a pour effet de mélanger les valeurs et de créer des transitions douces. Comme il ne s'agit pas d'une vraie moyenne, mais que nous divisons par 4 la somme de 3 pixels, plus nous montons dans le tableau, plus l'intensité est réduite. Mais le dégradé reste dépendant des valeurs initiales de la ligne la plus en bas et des valeurs des pixels alentour.

Voici pour la base. Tout ce que nous avons à faire est finalement de parcourir toutes les cellules du tableau et de faire un calcul relativement simple. Nous obtenons alors une grille complète de valeurs correspondant à la « température » de chaque pixel. Nous devrons ensuite traduire cela en différentes couleurs pour procéder à l'affichage. Pour cette étape, nous reposerons sur l'une des fonctionnalités de FastLED : la gestion de palettes.

Nous pourrions, en effet, nous contenter d'utiliser une seule composante, le rouge par exemple, et transposer directement les valeurs du tableau en intensité de rouge de 0 à 255. Cependant, pour que l'effet soit réaliste nous devons utiliser des couleurs typiques d'une flamme, qui s'étalent généralement en un dégradé du blanc au jaune, puis au rouge et enfin au noir.

Notez que cette base d'algorithme peut être utilisée telle quelle, mais également être ajustée en fonction des besoins. Énormément de paramètres entrent en ligne de compte et permettent d'ajuster le rendu final :

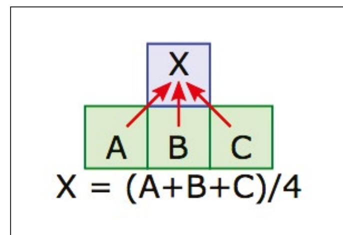
- Combien de pixels chauds doivent être aléatoirement allumés en bas du tableau ?
- Avec quelle probabilité ceci doit arriver ?
- Les nouveaux pixels doivent-ils être dans une plage de valeurs spécifiques ou non ?
- Faut-il prendre en compte ou non la valeur actuelle du pixel dans la moyenne ?
- Faut-il prendre en compte les pixels adjacents de droite et de gauche dans le calcul de la valeur ?
- Faut-il accélérer le refroidissement en divisant par une plus grande valeur ?
- etc.

Je vais faire ici un certain nombre de ces choix dans le croquis qui va suivre, mais je vous invite, comme toujours à explorer par vous-même et adapter cela à vos attentes.

4. NOTRE CROQUIS ARDUINO

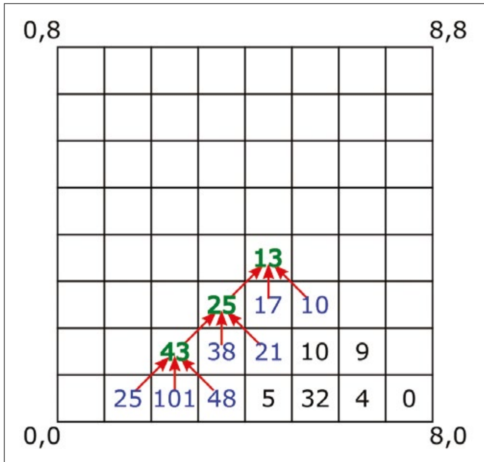
Il est temps d'entrer dans le vif du sujet et de mettre toutes ces théories en pratique. Après avoir installé FastLED via le gestionnaire de bibliothèque, nous pouvons débuter la rédaction du croquis en commençant par les classiques déclarations :

```
#include <FastLED.h>

// nombre de leds
#define NBRPIX 64

// matrice pour calculer l'image
uint8_t firematrice[8][8];

// tableau de leds
CRGB leds[NBRPIX];
// palette
CRGBPalette16 gPal;
```



La base d'un algorithme de simulation de feu est très simple. Il suffit, pour chaque pixel, de faire un simple calcul basé sur les valeurs des trois pixels immédiatement inférieurs.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Exemple de propagation de valeurs dans notre tableau. On démarre par des valeurs choisies aléatoirement au bas de l'affichage et on remonte ligne par ligne en calculant chaque pixel. À chaque ligne on rafraîchit l'affichage et, arrivé en haut, on repart pour un tour...

Après avoir inclus la bibliothèque, nous définissons une macro **NBRPIX** correspondant au nombre total de leds WS2812b utilisées. S'en suit alors la déclaration de notre fameux tableau de 8 par 8 **uint8_t**. Comme il s'agit d'une variable globale et de C++, celle-ci sera automatiquement initialisée à 0.

Nous enchaînons ensuite sur la déclaration d'une variable **leds**, étant un tableau de **CRGB** d'une taille correspondant au nombre de leds. Contrairement à la bibliothèque Adafruit NeoPixel, FastLED expose la variable correspondant à la valeur des couleurs des leds. En effet, ce tableau **leds** pourra directement être manipulé sans utiliser de méthode spécifique. Pour donner une couleur à la première led par exemple, nous n'utiliserons pas quelque chose comme :

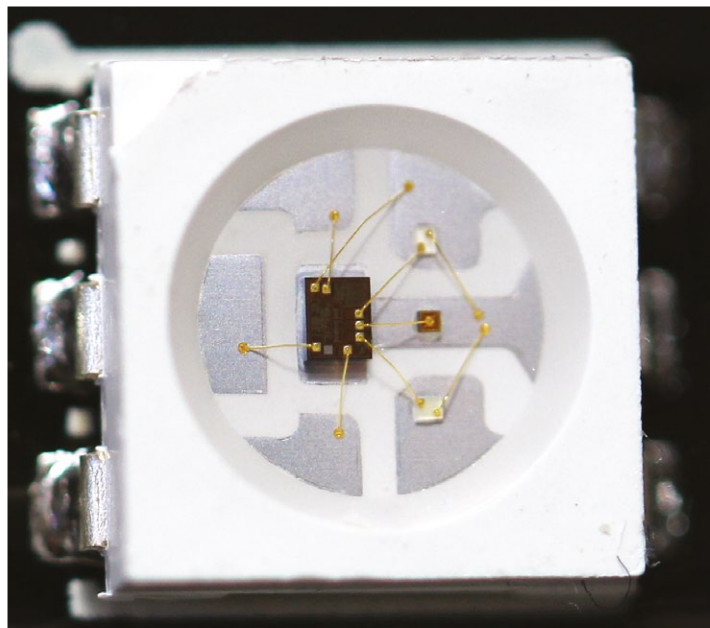
```
pixels.setPixelColor(0, pixels.Color(255,0,0));
<file>
mais directement :
<file>
leds[0].setRGB( 255, 0, 0);
// ou
leds[0] = 0xFF0000;
// ou encore
leds[0].r = 255;
leds[0].g = 0;
leds[0].b = 0;
// ou tout simplement
leds[0] = CRGB::Red;
```

Enfin, nous déclarons également une palette de couleurs **gPal** que nous initialiserons puis utiliserons par après. Tout est à présent en place et nous pouvons passer à notre fonction **setup()** :

```
void setup() {
  // initialisation du PRNG
  randomSeed(analogRead(0));

  // configuration des leds
  FastLED.addLeds<WS2812B, 6, GRB>(leds, NBRPIX);
  // ajustement luminosité
  FastLED.setBrightness(155);
  // initialisation palette personnalisée
  gPal = CRGBPalette16(CRGB::Black, CRGB::Red, CRGB::Red, CRGB::Yellow,
                       CRGB::Yellow, CRGB::Yellow, CRGB::Yellow, CRGB::White,
                       CRGB::White, CRGB::White, CRGB::White, CRGB::White,
                       CRGB::White, CRGB::White, CRGB::White, CRGB::White);
}
```

Là encore, l'utilisation de FastLED se démarque de celle d'Adafruit NeoPixel avec une syntaxe un peu particulière (utilisation de patron ou de *template* C++). Nous ajoutons les leds en appelant la méthode de classe **FastLED.addLeds** et en spécifiant le type de composant (ici des WS2812b), la broche Arduino utilisée et l'ordre des données de couleur.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Les WS2812b ne sont pas de simples leds. On distingue clairement sur ce gros plan le circuit intégré et les trois leds reliées avec des fils en or (Canon EOS 700D avec objectif macro EF-S 60mm f/2.8 USM, pour les amateurs de photo).

Dans le cas des WS2812b, il ne s'agit pas de rouge/vert/bleu, mais de vert/rouge/bleu. On précise également, en argument, la variable **leds** que nous avons déclaré précédemment ainsi que le nombre de leds présentes.

Nous réglons également l'intensité lumineuse globale à 155 (sur 255 max), puis définissons notre palette de 16 couleurs. Les valeurs utilisées ici, avec un dégradé non linéaire et une grosse part de blanc par rapport aux autres couleurs, ont été choisies après tâtonnement et de nombreux essais. Le choix de la palette est dépendant de la façon dont nous calculons les valeurs de température de chaque pixel et donc de l'effet souhaité (flamme de bougie, feu de camp miniature, feu vif et attisé, etc.).

La façon dont je suis arrivé à cette solution est relativement simple : tester l'algorithme et ses différents paramètres avec une seule couleur (rouge) et donc sans palette, puis une fois une animation satisfaisante obtenue, utiliser la palette et l'ajuster pour aligner chaque teinte.

Il est temps maintenant de passer à la fonction **loop()** effectuant tout le travail d'animation. On commence donc par placer un pixel chaud au bas du tableau :

```
void loop() {
  // On ajoute peut-être un pixel chaud en bas
  // 2 chances sur 3
  if(random(0,3)) {
    firematrice[0][random(0,8)] = random(160,256);
  }
```

Les arguments passés à **random()** sont respectivement la valeur minimum et maximum (exclue) qui peut être choisie aléatoirement. Comme nous utilisons ceci dans une condition **if()**, les instructions dans la portée ne sont exécutées que si la fonction retourne une valeur non nulle. Nous avons donc ici deux chances sur trois d'exécuter le code. Celui-ci est une simple affectation de valeur où la position verticale est fixe (ligne 0) et celle horizontale est choisie aléatoirement. Enfin, la valeur affectée est également aléatoire, mais bornée entre 160 et 255 (inclus). Nous avons donc une chance sur trois de placer un pixel d'une valeur relativement élevée à n'importe quel endroit de la ligne du bas.

Dans l'état, l'algorithme fonctionne déjà relativement bien, mais les écarts de couleurs entre les pixels allumés et éteints sont trop importants. Nous voulons donc lisser tout cela. Pour ce faire, nous faisons, pour chaque pixel la moyenne « bricolée » des pixels de droite et de gauche :

```
  // On lisse la dernière ligne en faisant la moyenne
  for (int x=0; x<8; x++) {
    firematrice[0][x]=
      (firematrice[0][(x-1>0) ? x-1 : 7]+
      firematrice[0][x]+
      firematrice[0][(x+1<7) ? x+1 : 0])/4;
  }
```

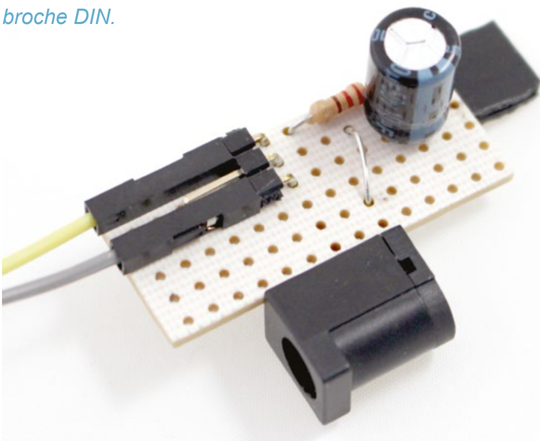
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Notez l'utilisation de l'opérateur ternaire nous permettant de choisir la valeur du dernier pixel de la ligne en lieu et place de celui de droite si nous sommes à la position 0 (si `x-1>0` alors on utilise `x-1`, sinon 7), et le premier pixel de la ligne si nous sommes à la dernière position. L'afficheur est donc « enroulé sur lui-même ».

Nous pouvons ensuite parcourir le tableau en commençant par le premier pixel de la seconde ligne en partant du bas. On procèdera au calcul que nous avons détaillé précédemment en faisant la somme des trois pixels inférieurs et en divisant pas 4 :

```
  // Pour chaque pixel on fait une moyenne des pixels plus bas
  // et on divise pour "refroidir" le tout
  for (int y=1; y<8; y++) {
    for(int x=0; x<8; x++) {
      firematrice[y][x]=
        (firematrice[y-1][(x-1>0) ? x-1 : 7]+
        firematrice[y-1][x]+
        firematrice[y-1][(x+1<7) ? x+1 : 0])/4;
    }

    // Après chaque ligne traitée, on rafraîchit l'affichage
    for(int i=0; i<8; i++) {
      for(int j=0; j<8; j++) {
        // on applique la palette aux leds
        CRGB color = ColorFromPalette(gPal, firematrice[i][j]);
        leds[(j*8)+i]=color;
      }
    }
    // affichage
    FastLED.show();
    delay(10);
  }
}
```

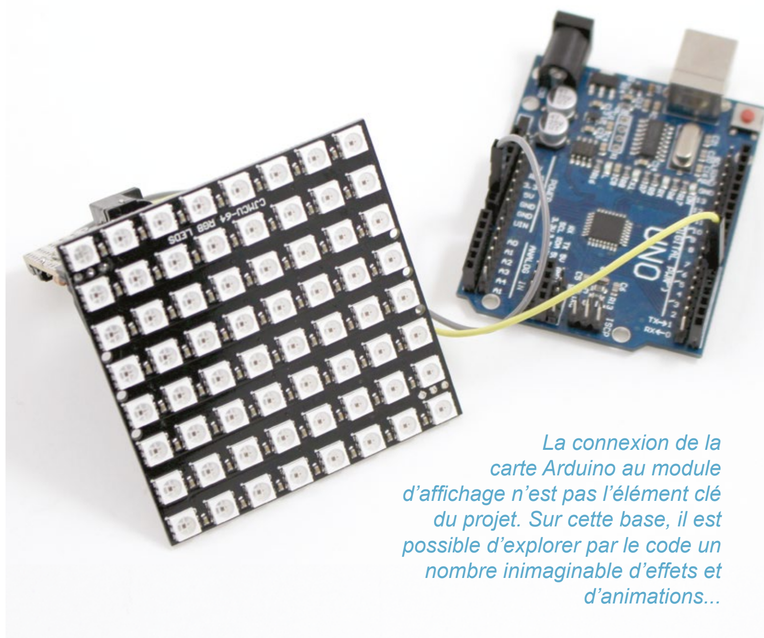
L'affichage des pixels dans les bonnes couleurs se fait en parcourant le tableau entièrement pour collecter les valeurs qu'il contient et en les transposant en composantes de rouge, de vert et de bleu qu'on place dans `leds[]` avant d'appeler `FastLED.show()` pour communiquer les données aux WS2812b. Cette transposition se fait très simplement en utilisant la fonction `ColorFromPalette()` et en lui passant en argument la palette à utiliser et une valeur entre 0 et 255 (8 bits).



Pour assurer un bon fonctionnement et une protection des WS2812b, il est recommandé d'ajouter un condensateur entre la masse et la ligne d'alimentation, au plus près des leds, ainsi qu'une résistance sur la broche DIN.

Notez que FastLED utilise ici une astuce. En effet, une vraie palette de couleurs contient généralement un index de 256 valeurs, mais ceci occuperait quelques 768 octets de mémoire vive, denrée rare sur un microcontrôleur comme celui d'un Arduino. Il est possible d'utiliser une telle palette traditionnelle (type `CRGBPalette256`), mais il est généralement préférable d'opter pour un `CRGBPalette16` qui ne contient qu'un index de 16 entrées, mais se comportera comme une palette de 256 couleurs. FastLED se chargera, en coulisse, de calculer les valeurs intermédiaires. Voilà pourquoi nous n'avons défini que 16 couleurs en début de croquis, mais pouvons utiliser une valeur d'index entre 0 et 255.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



La connexion de la carte Arduino au module d'affichage n'est pas l'élément clé du projet. Sur cette base, il est possible d'explorer par le code un nombre inimaginable d'effets et d'animations...

5. AMÉLIORATIONS, AJUSTEMENTS ET RÉGLAGES

L'effet final obtenu sur les 64 leds que compte notre afficheur est impacté par énormément de choses, disséminées dans le croquis :

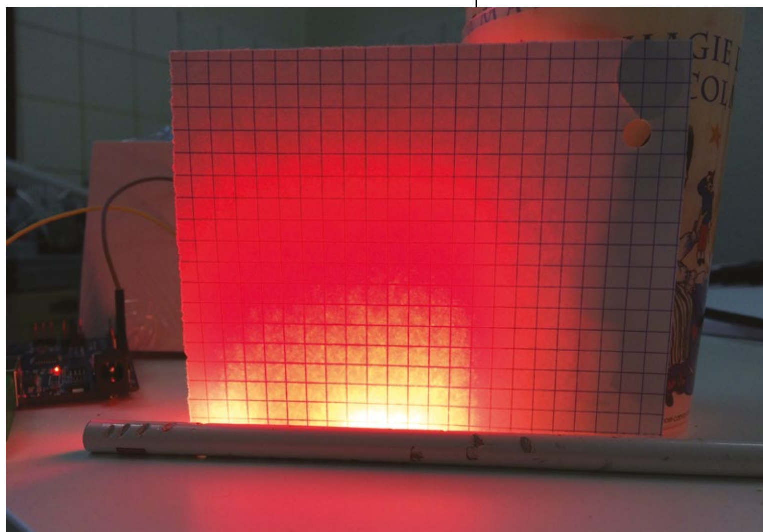
- la probabilité d'allumer un pixel chaud (`if(random(0,3))`),
- le nombre de pixels allumés (nombre de lignes dans la portée du `if`),
- le lissage utilisé sur la première ligne (ici division par 4),
- la division opérée pour « refroidir » les pixels dans le calcul du tableau complet (premier parcours du tableau),
- la palette de couleurs utilisée lors de l'affichage.

Tous ces paramètres peuvent être ajustés pour avoir un feu plus « calme », concentré en un point particulier, avec des couleurs différentes ou encore davantage proches d'une animation de vieux jeux vidéos. Il est également possible d'utiliser deux tableaux, un contenant l'état actuel des pixels et un autre pour le résultat des opérations arithmétiques, de façon à ce que les calculs effectués ne s'impactent pas mutuellement. On peut également travailler par phase en procédant à plusieurs passages successifs : refroidissement, placement des nouveaux pixels, propagation des valeurs...

Enfin, nous avons l'utilisation pratique de l'effet. Nous pouvons facilement jouer sur les paramètres et les valeurs utilisées dans les calculs pour conditionner le rendu. Ceci peut être un simple potentiomètre sur une entrée analogique, mais également une liaison série ou Bluetooth permettant de faire un système de notification amusant : nombre de mails non lus, charge système, température, etc.

Comme toujours, je ne peux que vous conseiller de jouer avec ce croquis et tester toutes les variations possibles afin de vous approprier ce projet et le rendre unique. Ce sera là également l'occasion de faire davantage connaissance avec FastLED qui, sur bien des points, surpasse admirablement les autres solutions de contrôle de leds intelligentes de type WS2812b. **DB**

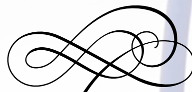
Cette photo ne rend absolument pas hommage au résultat effectivement obtenu, mais après une longue discussion avec nos infographistes nous avons convenu qu'il nous serait vraiment difficile de convaincre l'imprimeur de coucher une vidéo sur du papier...



Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

CRÉEZ DES BOUTONS À COPIER/COLLER

Denis Bodor



Inutile de le cacher, ce projet est une simple excuse pour découvrir quelque chose de bien plus vaste et intéressant que le fait de se faciliter la vie en faisant des copier/coller à l'aide de boutons dédiés. Ce qui se cache derrière ce simple exemple n'est rien de moins que d'automatiser n'importe quelle séquence de touches, peu importe leur nombre, pouvant être déclenchée par une simple pression sur un bouton.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Les cartes Arduino permettent de faire énormément de choses, mais certaines d'entre elles possèdent des caractéristiques très spécifiques. Je qualifierai les modèles UNO et MEGA 2560 de génériques, car permettant d'utiliser des fonctionnalités de base comme le contrôle des entrées/sorties, la communication série, SPI et i2c et offrant, de façon générale, l'occasion de se familiariser avec la richesse (déjà importante) de la plateforme.

Et puis nous avons les modèles plus « spécifiques » comme l'Esplora, la Due, la Zero ou encore la Yùn, proposant des caractéristiques uniques venant compléter ce qu'offrent déjà les modèles les plus populaires : puissance de calcul, architecture différente, processeur 32 bits, audio, connectivité sans fil ou filaire, etc.

À mi-chemin entre ces deux parties de la gamme Arduino/Genuino, je placerai deux modèles précis de cartes : l'Arduino Leonardo et l'Arduino Micro. Celles-ci, comme la UNO par exemple, comprennent un microcontrôleur Atmel de la famille AVR, mais celui-ci, l'ATmega32U4, dispose d'une fonctionnalité captivante : il peut se présenter comme un périphérique USB.

Bien sûr, lorsque vous branchez votre carte Arduino, quel que soit son modèle, vous le faites en USB, mais ce n'est pas de cela qu'il s'agit. Avec une UNO par exemple, l'USB n'est qu'un pont permettant de faire communiquer le microcontrôleur avec votre ordinateur pour programmer la carte ou la faire dialoguer avec le moniteur série de l'environnement de programmation (IDE). Les Leonardo et Micro vont plus loin et vous permettent, à l'aide d'un croquis rédigé par vos soins, d'avoir la mainmise sur ces fonctionnalités USB et donc de créer un périphérique USB personnalisé.

1. L'USB : OMNIPRÉSENT, MAIS RELATIVEMENT COMPLEXE

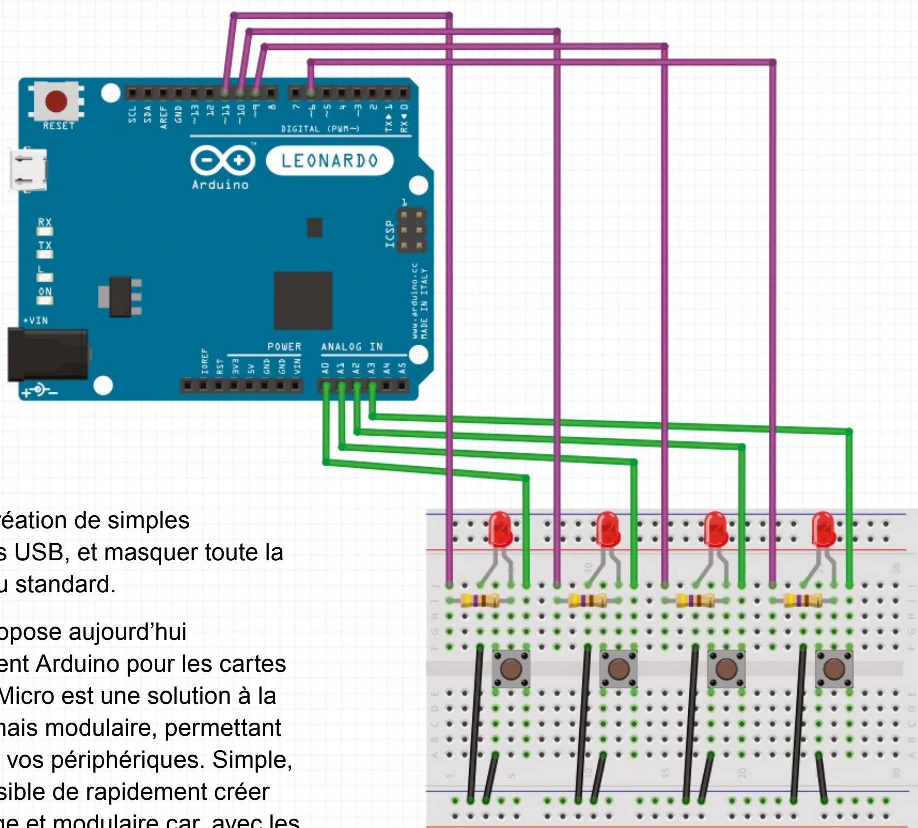
L'interface USB, depuis une quinzaine d'années, a détrôné presque toutes les autres formes de connexions permettant à un ordinateur de dialoguer avec un périphérique : port PS/2, port série, port parallèle, interface SCSI, cartes spécifiques pour scanner, etc. Désormais, la quasi-totalité des appareils que vous pouvez brancher à un PC ou un Mac sont en USB. Et c'est précisément là le but initial de ce standard dont le nom lui-même a été choisi en ce sens : *Universal Serial Bus* ou bus série universel en bon français (ou « *Umfassend Systematische Bindung* » si vous êtes un scientifique nazi œuvrant sur la face cachée de la lune).

Fut un temps, ceci a été une véritable catastrophe pour les bidouilleurs de tout poil. Alors qu'il était très facile de brancher des montages sur un port parallèle, ou à défaut sur un port série (merci le bon vieux MAX232), l'arrivée de l'USB a posé un énorme problème en nécessitant la prise en charge d'un protocole difficile à implémenter tant au niveau matériel (signaux) que logiciel (protocole, pilotes, etc.).

La polyvalence de l'USB et sa capacité à uniformiser tous types de connexions induisent une grande complexité. Bien entendu, avec la démocratisation de ce type de connexion, des solutions accessibles aux électroniciens amateurs ont commencé à voir le jour, d'un simple pont série 5V vers RS232 puis à l'USB, aux solutions entièrement logicielles comme AVR-USB (devenu V-USB depuis) permettant aux microcontrôleurs Atmel AVR de « parler » USB à l'aide de seulement deux petites résistances de 68 ohms (mais en USB 1.1 uniquement).

Il aura cependant fallu attendre l'Arduino Leonardo et son ATmega32U4, accompagné des bibliothèques adéquates pour véritablement rendre le plus accessible

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Matériellement, le montage se résume à la mise en œuvre de fonctionnalités simples de la carte Arduino : 4 leds et 4 boutons. Pour la réalisation pratique, j'ai utilisé des boutons poussoirs intégrant des leds rouges, mais il n'existe pas de connexion directe entre l'une et l'autre fonction du produit. Un résultat similaire peut donc être obtenu en séparant les composants.

possible la création de simples périphériques USB, et masquer toute la complexité du standard.

Ce que propose aujourd'hui l'environnement Arduino pour les cartes Leonardo et Micro est une solution à la fois simple, mais modulaire, permettant de construire vos périphériques. Simple, car il est possible de rapidement créer un tel montage et modulaire car, avec les dernières versions de l'environnement et des bibliothèques, il est possible de pousser vraiment très loin le contrôle de l'USB.

Trois approches sont possibles de cette manière :

- la plus simple consistant à utiliser les bibliothèques **`Keyboard`** et **`Mouse`** permettant d'émuler un clavier ou une souris ;
- utiliser la bibliothèque **`HID`** servant de base aux deux précédentes et permettant de créer n'importe quel périphérique USB de type (ou classe) HID (*Human Interface Device*) propre aux périphériques ayant une interaction physique avec l'utilisateur au sens général du terme ;
- jouer, au plus bas niveau, avec **`PluggableUSB`** pour avoir un accès complet (et complexe) aux fonctionnalités USB en développant un module *PluggableUSB* sous forme d'une bibliothèque, à la façon dont **`HID`** a été créé.

Dans les pages qui vont suivre, nous aurons l'occasion de découvrir différentes approches, sans pour autant arriver à **`PluggableUSB`** (peut-être une prochaine fois) nécessitant l'assimilation d'une grande partie des spécifications USB. Mais avant d'en arriver à des réalisations plus avancées, voyons tout d'abord ce qu'il est possible de faire avec les bibliothèques installées en standard dans l'environnement.

2. DÉMARRER AVEC UN PETIT PROJET SIMPLE, MAIS UTILE

Notre objectif sera ici relativement simple puisque nous allons chercher à étendre les fonctionnalités d'une installation informatique classique. Si, comme moi, le fait de voir des personnes utiliser la souris ou les menus pour faire du copier/coller plutôt que les touches de raccourcis Ctrl+C et Ctrl+V, a tendance à bien vous horripiler, voici une solution (pour vos nerfs) : donner à cette pauvre personne des boutons pour faire de même plus rapidement.

Concrètement, cette excuse nous permettra de prendre en

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

main la fonction d'émulation (qui n'en est pas vraiment une) de clavier et d'associer la pression sur un bouton poussoir lu par la carte Arduino avec un message, envoyé en USB, simulant l'utilisation d'un raccourci.

Attention : deux choses sont ici importantes à comprendre. Premièrement, ceci ne fonctionnera qu'avec les cartes Leonardo et Micro, construites autour du microcontrôleur ATmega32U4. Un hack est possible sur les cartes comme la UNO à condition qu'elles intègrent un microcontrôleur ATmega16U2 en guise de passerelle série/USB (cf. un peu plus loin dans le magazine), mais cela reste une bidouille et ne fonctionne pas avec les clones Arduino.

L'autre point important concerne le fonctionnement même de ce genre de croquis et fonctionnalités : la carte Arduino va se faire passer pour un clavier USB et envoyer des séquences de touches. Ceci signifie que, une fois le croquis chargé, votre ordinateur verra un nouveau clavier (en plus d'une carte Arduino) et le prendra en charge. Ainsi, si votre croquis envoie du texte saisi, celui-ci a toutes les chances de finir dans la fenêtre active à ce moment, et donc, en toute logique dans l'éditeur de code. La solution consiste donc, quoi que vous programmiez, à toujours conditionner l'envoi de texte et de raccourcis à une action de votre part (pression sur un bouton, connexion d'une broche, etc.) ou au minimum, à une temporisation vous laissant le temps de prendre les devants.

Voici notre croquis de démonstration :

```
#include <Keyboard.h>

int bouton[] = {A0,A1,A2,A3};
int led[]    = {11,10,9,6};

void raccourcis(int val) {
  // pour Win & Linux
  //Keyboard.press(KEY_LEFT_CTRL);
  // pour Mac
  Keyboard.press(KEY_LEFT_GUI);
  switch(val) {
    case 0:
      // "Annuler"
      // attention QWERTY w=z !
      Keyboard.press('w');
      break;
```


Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

DISPONIBLE DÈS LE 10 MARS !

HACKABLE HORS-SÉRIE n°2

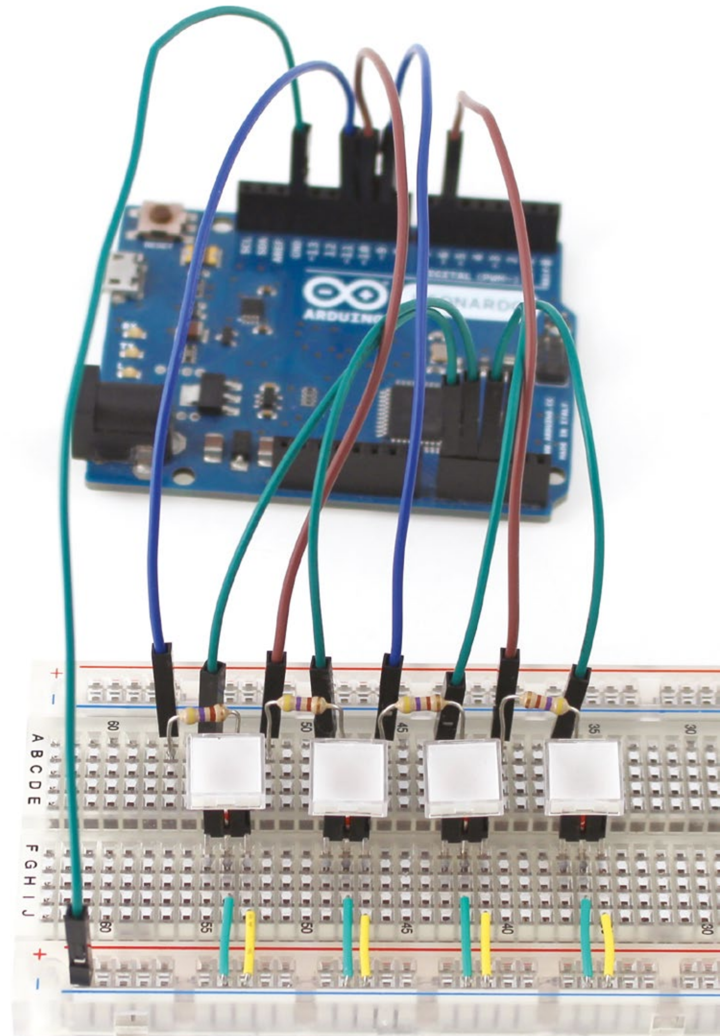
Sous réserve de toutes modifications.

DÉBUTEZ EN PROGRAMMATION SUR RASPBERRY PI !

NE LE MANQUEZ PAS CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

```
    case 1:
      // "Tout sélectionner"
      // attention QWERTY q=a !
      Keyboard.press('q');
      break;
    case 2:
      // "Copier"
      Keyboard.press('c');
      break;
    case 3:
      // "Coller"
      Keyboard.press('v');
  }
  delay(100);
  Keyboard.releaseAll();
}

// juste pour faire beau
void animation() {
  for(int i=0; i<4; i++) {
    digitalWrite(led[i], HIGH);
    delay(150);
  }
  for(int i=0; i<4; i++) {
    digitalWrite(led[i], LOW);
    delay(150);
  }
}

void setup() {
  // les leds en sortie
  for (int i=0; i<4; i++) {
    pinMode(led[i], OUTPUT);
  }

  // les bouton en entrée
  for (int i=0; i<4; i++) {
    pinMode(bouton[i], INPUT_PULLUP);
  }

  Keyboard.begin();

  animation();
}

void loop() {
  // on scanne les boutons
  for (int i=0; i<4; i++) {
    if(!digitalRead(bouton[i])) {
      // led allumée
     digitalWrite(led[i], HIGH);
      // envoi du raccourci
      raccourcis(i);
      // on éteint doucement la led
      // tout en gérant les rebonds
      for(int j=250; j>=0; j=j-10) {
        analogWrite(led[i],j);
        delay(10);
      }
    }
  }
}
```



La bibliothèque **Keyboard** est très simple d'utilisation, une fois celle-ci correctement initialisée avec un petit **Keyboard.begin()** dans la fonction **setup()**, une série de fonctions sont alors utilisables. Nous pouvons, comme ici, dans **raccourcis()**, envoyer des pressions sur les touches avec **Keyboard.press()** et relâcher ces touches le moment venu avec **Keyboard.release()** ou **Keyboard.releaseAll()** (relâcher toutes les touches précédemment enfoncées).

Pour pouvoir rendre le croquis plus modulaire et facile à étendre, nous plaçons les valeurs des broches utilisées dans des tableaux et notre fonction d'envoi de séquence de touches

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

prendra en argument une valeur. Ceci nous permet de reposer massivement sur l'utilisation de boucles `for` pour gérer à la fois les boutons et les leds. Ainsi si vous souhaitez ajouter ou retirer des boutons, il vous suffit de modifier le contenu des tableaux avec les bonnes valeurs de broches ainsi que la fonction `raccourcis()`.

`Keyboard` nous permet donc d'enfoncer des touches, directement désignée par un symbole alphanumérique (caractère), mais également par un nom pour les touches qui n'ont pas de symboles, comme `KEY_LEFT_CTRL` (touche Ctrl de gauche). La liste complète des noms (ou macros) est disponible sur **https://www.arduino.cc/en/Reference/KeyboardModifiers**. Vous remarquerez qu'elle est relativement limitée, mais nous y reviendrons dans les projets suivants.

Nous pouvons également envoyer des chaînes de caractères avec `Keyboard.print()` et `Keyboard.println()`, à la manière des méthodes équivalentes pour `Serial`. Notez cependant que, pour toutes les fonctions désignant les touches par un caractère, c'est l'organisation QWERTY US qui sera utilisée. Si vous envoyez un `"a"` c'est dont un “q” qui apparaîtra sur l'ordinateur...

3. LE CLAVIER, MAIS PAS SEULEMENT

Ce bref, mais suffisant à mon goût, aperçu de la bibliothèque `Keyboard` laisse entrevoir des possibilités intéressantes sans pour autant permettre des réalisations très poussées. Il en va de même pour la bibliothèque sœur, `Mouse`, permettant de faire de votre carte Arduino une souris USB. Là, je dois avouer que, bien qu'il soit très amusant de faire se déplacer le pointeur de la souris ainsi (avec `Mouse.move()`), l'intérêt pratique m'échappe un peu (à part pour faire une blague à quelqu'un).

Pour faire des choses un peu plus intéressantes avec les cartes Leonardo et Micro en USB, il faut aller au-delà de ces bibliothèques. Chose que je vous propose de faire dans les pages qui suivent... DB

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

NE MANQUEZ PAS LA NOUVELLE FORMULE !

LINUX PRATIQUE n°100

CONTRÔLEZ L'ACCÈS AU WEB !

ACTUELLEMENT DISPONIBLE
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
http://www.ed-diamond.com

CRÉEZ UN CONTRÔLEUR DE VOLUME POUR VOTRE ORDINATEUR

Denis Bodor

Nous venons de voir que simuler un clavier ou une souris à l'aide d'un Arduino micro, ou Leonardo, n'est pas quelque chose de bien difficile. Mais dès lors qu'on sort des sentiers battus, les choses peuvent rapidement se corser. Notre objectif ici sera d'émuler l'utilisation des touches d'un clavier multimédia pour contrôler le volume audio. Autrement dit, créer un « bouton » de volume qui fonctionne avec n'importe quel ordinateur et n'importe quel système !

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

La bibliothèque **Keyboard** que nous venons de voir permet un certain nombre d'opérations que je qualifierai de basiques : envoyer des caractères et utiliser quelques touches spéciales comme MAJ, Alt, Ctrl, etc. La bibliothèque cependant est construite d'une manière assez particulière et tout à fait inadaptée pour une utilisation avancée.

En effet, chaque touche d'un clavier est désignée par un code (*keycode*). C'est l'ordinateur auquel est connecté le clavier qui se charge d'associer ce code avec un symbole alphanumérique donné, en fonction de la disposition linguistique du périphérique. C'est entre autres pour cette raison que vous devez configurer vous-même la langue d'un clavier connecté à une Raspberry Pi. Ceci ne peut être déterminé automatiquement.

Or ce que fait la bibliothèque **Keyboard** se place un peu entre ces deux étapes. En regardant le contenu de **Keyboard.cpp** dans les fichiers de l'environnement Arduino, on peut voir des choses comme ceci :

```
size_t Keyboard_::press(uint8_t k)
{
  uint8_t i;
  if (k >= 136) { // it's a non-printing key (not a modifier)
    k = k - 136;
  } else if (k >= 128) {  // it's a modifier key
    _keyReport.modifiers |= (1<<(k-128));
    k = 0;
  } else { // it's a printing key
    k = pgm_read_byte(_asciimap + k);
    if (!k) {
      setWriteError();
      return 0;
    }
    if (k & 0x80) { // it's a capital letter or other character reached with shift
      _keyReport.modifiers |= 0x02;   // the left shift modifier
      k &= 0x7F;
    }
  }
```

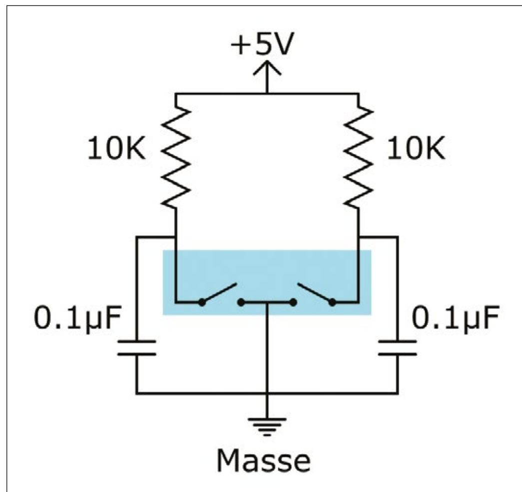
Cette méthode **press()** prend en argument une valeur **k** qui peut être soit un caractère, soit le code d'une touche. Pour déterminer la nature du contenu de **k**, différents tests sont faits. Si **k** est supérieur à 136, il est estimé qu'il ne s'agit pas d'un caractère, mais d'un code auquel on soustrait alors 136. Mais si **k** est entre 128 et 136, alors il doit s'agir d'un modificateur (Maj, Ctrl, etc.). Et enfin si tout ceci n'est pas vérifié, la bibliothèque considère alors que c'est un caractère et utilise le code correspondant recherché dans le tableau **_asciimap** en mémoire flash.

Ceci est très intéressant et pratique pour simplement utiliser **Keyboard**, mais qu'en est-il des codes supérieurs à 136 qu'on souhaiterait effectivement utiliser ? La réponse est simple, avec la bibliothèque dans cet état, ça n'est pas possible !

1. TORTURER LA BIBLIOTHÈQUE POUR RIEN

Instinctivement, la première chose à laquelle on pense devant cet état de fait, est de chercher à tout simplement modifier la bibliothèque pour sauter les différents tests. Bien entendu, modifier le fichier **Keyboard.cpp** directement dans le répertoire où est installé l'environnement est hors de question. À la prochaine mise à jour, nos modifications seraient écrasées (et puis ce n'est absolument pas propre).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Un encodeur rotatif, zone bleue, n'est rien d'autre qu'une paire d'interrupteurs actionnés alternativement en tournant la molette. Pour éviter les phénomènes de rebonds, on ajoute simplement une résistance entre une broche et l'alimentation, et un condensateur entre la broche et la masse. Ceci a pour effet de « lisser » le changement d'état sur une brève période et donc d'obtenir un signal franc, débarrassé de toutes interférences mécaniques.

On peut également envisager de copier la bibliothèque (le répertoire `Keyboard` et les fichiers `Keyboard.cpp` et `Keyboard.h`) dans le répertoire `libraries` de son carnet de croquis. Il faudrait alors réviser tout le code pour changer son nom et éviter les conflits avec la bibliothèque originale. C'est beaucoup de travail et ça ne nous avance pas beaucoup.

Dernière option, nous pouvons tout simplement prendre `Keyboard.cpp` et `Keyboard.h` et les copier dans le répertoire de notre croquis tout en y spécifiant `#include "Keyboard.h"` et non `#include <Keyboard.h>`. Ainsi, dans l'environnement, nous obtenons trois onglets, un par fichier, et nous pouvons librement modifier la bibliothèque sans rien casser. Là, il devient possible d'ajouter deux nouvelles méthodes, `Keyboard_::pressRaw(uint8_t k)` et `Keyboard_::releaseRaw(uint8_t k)`, en copiant/collant le code déjà existant et en supprimant les différents tests (sans oublier les prototypes dans `Keyboard.h`). Ainsi, en appelant `Keyboard.pressRaw()` en lieu et place de `Keyboard.press()`, nous pouvons envoyer les codes de notre choix...

Ceci fonctionnera très bien pour des codes standards, comme par exemple ceux du pavé numérique, mais ne fonctionnera absolument pas pour notre cas particulier : les boutons « Vol+ », « Vol- » et « Muet » présents sur certains claviers multimédias. En cherchant les codes sur le Web et en les utilisant ainsi, rien ne se passe, c'est l'échec total. Il est temps d'appeler une Raspberry Pi à la rescousse et d'éluder ce mystère...

2. RAPPORTS HID ET ESPIONNAGE DE CLAVIER

Pour comprendre la source du problème, il faut avant tout se faire une idée précise de la manière dont fonctionne un clavier USB. Ce type de périphériques appartient à une gamme plus vaste dite HID pour *Human Interface Device* (périphérique d'interface pour humain). Sans entrer outre mesure dans le détail, cette classe de périphériques comprenant claviers, souris, contrôleur de jeux, etc., communique sur le bus USB avec un protocole particulier basé sur la notion de rapports (*reports*).

Lors de la connexion à un ordinateur, un périphérique USB est analysé, on parle d'énumération. Durant cette phase, un matériel compatible HID fournit non seulement des informations sur sa nature, mais également sur ses fonctionnalités et la façon de communiquer avec lui, par l'intermédiaire d'un descriptif de rapport HID. Dans les grandes lignes, il détaille la forme que doivent prendre les messages qu'il reçoit ainsi que ceux qu'il émet. Ce descriptif de rapport permet également à l'ordinateur d'avoir un modèle des messages à venir et peut donc les comprendre et les vérifier.

Cette méthode de fonctionnement rend les périphériques HID très souples, car les constructeurs peuvent alors utiliser ces rapports et ce format plus ou moins comme ils le souhaitent tout en fournissant des fonctionnalités communes à des types entiers de périphériques. Ceci explique non seulement pourquoi n'importe quel clavier peut fonctionner avec n'importe quel ordinateur, mais que certaines touches ne fonctionnent pas sans un pilote spécifique, et en même temps que la classe HID est également souvent utilisée pour des matériels qui ne sont pas des interfaces pour humain (programmeur de cartes, station météo USB, notificateurs à led, etc.).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Il existe, sous GNU/Linux, des outils très intéressants pour obtenir ce genre d'informations d'un périphérique et en particulier **usbhid-dump** (paquet **usbutils**). Nous pouvons donc nous en servir pour voir ce que raconte un clavier disposant des touches qui nous intéressent. Notre victime (volontaire) sera ici un des claviers que j'affectionne tout particulièrement, le G230 de Cherry (alias le Cherry Stream).

Nous devons tout d'abord trouver les identifiants USB avec :

```
$ lsusb
Bus 001 Device 005: ID 046a:0023 Cherry GmbH CyMotion Master Linux Keyboard G230
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast
Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Notre matériel a comme ID vendeur **046a** et comme ID produit **0023**. Nous pouvons alors utiliser **usbhid-dump** pour obtenir le descriptif de rapport :

```
$ sudo usbhid-dump -m 046a:0023
001:005:001:DESCRIPTOR         1482164105.066372
05 0C 09 01 A1 01 85 01 19 00 2A FF 03 15 00 26
FF 03 95 01 75 10 81 00 C0 05 01 09 80 A1 01 85
02 19 81 29 83 15 00 25 01 75 01 95 03 81 02 95
05 81 01 C0

001:005:000:DESCRIPTOR         1482164105.091371
05 01 09 06 A1 01 05 07 19 E0 29 E7 15 00 25 01
75 01 95 08 81 02 95 01 75 08 81 01 95 03 75 01
05 08 19 01 29 03 91 02 95 05 75 01 91 01 95 06
75 08 15 00 26 FF 00 05 07 19 00 2A FF 00 81 00
C0
```

Voilà qui n'est pas très parlant, mais nous montre déjà qu'il y a deux descriptifs disponibles. Ceci ne peut signifier qu'une chose : on peut communiquer avec ce clavier de deux façons, car même s'il s'agit bien d'un seul et même périphérique USB, il se compose de plusieurs interfaces (chose confirmée par un **lsusb -v -d 046a:0023**). Il y a deux descriptions disponibles parce que le clavier possède une interface « normale » et une permettant un ensemble minimum de fonctionnalités utilisées, par exemple, dans la configuration du BIOS d'un PC.

Nous pouvons ensuite voir ce qui se passe lorsque nous utilisons le clavier en ajoutant les options **-es** à notre ligne de commandes :

```
$ sudo usbhid-dump -m 046a:0023 -es
Starting dumping interrupt transfer stream
with 1 minute timeout.

001:005:000:STREAM          1482164475.019037
00 00 14 00 00 00 00 00

001:005:000:STREAM          1482164475.075153
00 00 00 00 00 00 00 00
```

Ces deux messages, qui sont des rapports HID, apparaissent lorsque j'appuie sur la touche « A » du clavier. Il y a un rapport pour la pression sur la touche, et un second lors du relâchement. Nous avons ici une série de 8 octets qui composent le message. Gardez bien cela en tête.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

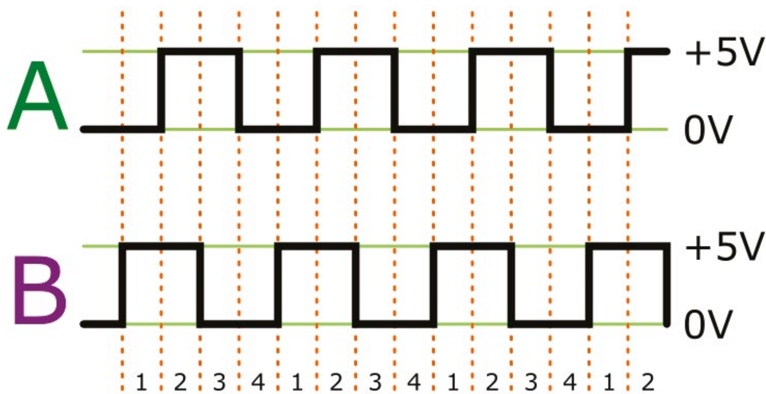
Voyons maintenant ce qu'il se passe lorsque j'appuie sur le bouton « - » permettant de réduire le volume audio :

```
001:005:001:STREAM          1482164683.768131
01 EA 00

001:005:000:STREAM          1482164683.768374
00 00 00 00 00 00 00 00

001:005:001:STREAM          1482164684.008132
01 00 00

001:005:000:STREAM          1482164684.008308
00 00 00 00 00 00 00 00
```

Voilà qui est intéressant ! Nous n'avons pas un message pour la pression et un autre pour le relâchement, mais deux. Toujours un rapport de 8 octets, mais précédé d'un autre de 3. La logique cependant semble identique. Lorsque la touche est enfoncée, les rapports indiquent des chiffres, mais lorsqu'elle est relâchée, ils sont remplacés par 0. De plus, le second rapport, lui, est toujours à 0... Notez également que l'entête affiché avant les rapports se compose ainsi `bus:périphérique:interface horodatage`. Les rapports de 3 octets proviennent de l'interface 1 et ceux de 8 octets de l'interface 0. Ce qui confirme la supposition précédente basée sur la sortie de `lsusb -v`. Il y a bien « deux claviers » dans le clavier...

3. MAIS QUE FAIT LA BIBLIOTHÈQUE ARDUINO KEYBOARD ?

Nous savons que le clavier n'utilise pas les mêmes rapports pour les touches « standards » et les touches multimédias, c'est un début de piste intéressant, mais nous devons aller fouiller dans la bibliothèque `Keyboard` pour en avoir le cœur net et trouver la source du dysfonctionnement.



Ce diagramme représente les changements d'état successifs qui apparaissent sur les broches d'un encodeur rotatif à mesure que la molette est tournée. Les deux signaux sont déphasés et il suffit de réagir à un changement d'état sur une broche pour détecter un mouvement, mais c'est la comparaison entre l'état actuel de la seconde broche et son état précédent qui indique le sens de rotation.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

La méthode press() utilise la fonction sendReport(&_keyReport) pour envoyer un rapport après avoir vérifié quelques points à propos de la fameuse variable k. sendReport() est déclaré dans le même fichier Keyboard.cpp un peu plus haut :

```
void Keyboard_::sendReport(KeyReport* keys)
{
  HID().SendReport(2,keys,sizeof(KeyReport));
}
```

Il s'agit d'un appel à la méthode SendReport() de la bibliothèque HID (la classe plus précisément). On voit également qu'un des arguments utilisés est keys, un pointeur sur une variable de type KeyReport qui est déclarée dans Keyboard.h :

```
typedef struct
{
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keys[6];
} KeyReport;
```

Ceci commence à prendre des allures de jeu de piste, mais nous touchons au but. Ce qui compose le rapport du point de vue de la bibliothèque Keyboard est donc un tableau de 6 octets (uint8_t) et deux autres octets réunis dans une structure. Le tout envoyé avec HID().SendReport() directement à la machine où est branché la carte Arduino. Ceci n'a rien à voir avec ce que fait le clavier G230, il y a un problème.

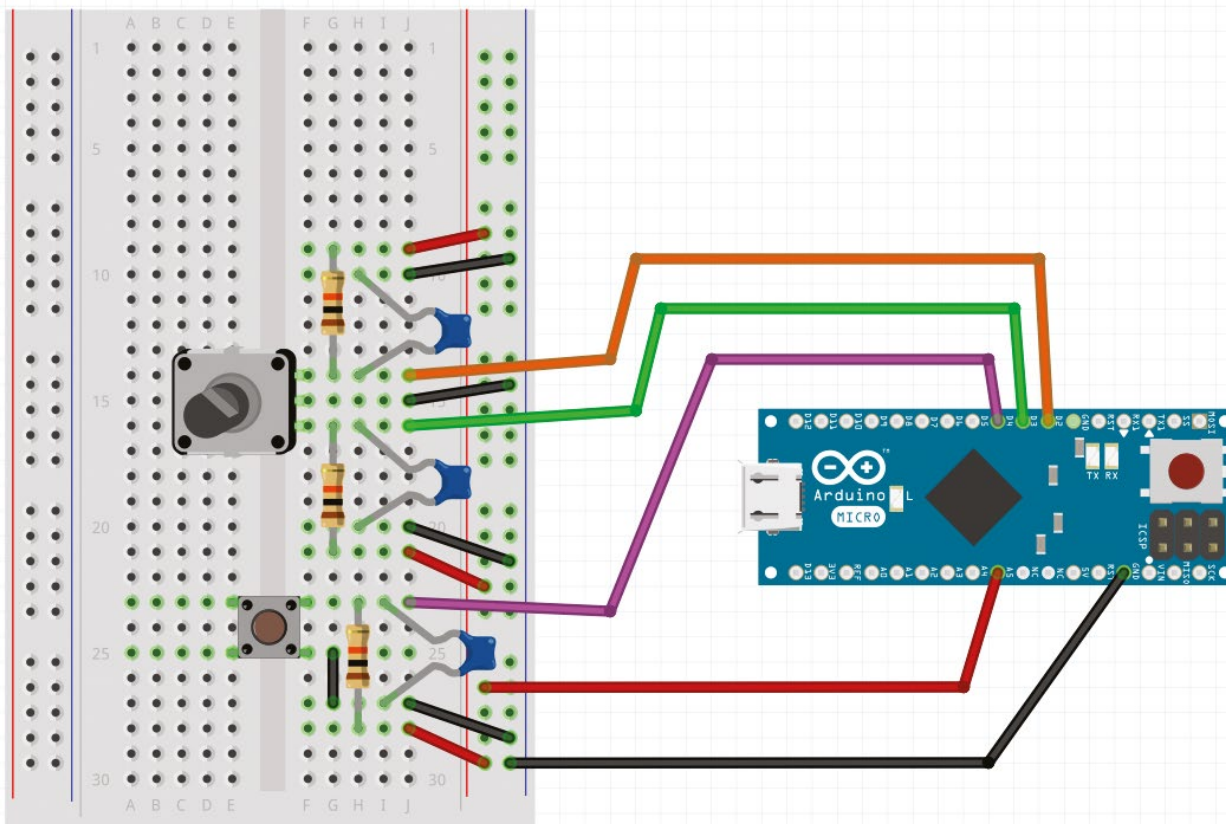
Chose confirmée en programmant notre carte Arduino avec un croquis simulant l'utilisation d'une touche « A » et en observant les rapports envoyés :

```
$ sudo usbhid-dump -m 2341:8036 -es
Starting dumping interrupt transfer stream
with 1 minute timeout.

001:007:002:STREAM              1482222681.700084
02 00 00 04 00 00 00 00 00

001:007:002:STREAM              1482222681.800063
02 00 00 00 00 00 00 00 00
```

Notre Keyboard envoie donc uniquement des rapports d'une taille précise qui ne peut être celle correspondant à un rapport pour notre fameuse touche de contrôle du volume. 9 octets dans le rapport, le 2 passé en premier argument de HID().SendReport() et les 8 octets de la structure KeyReport. Ceci bien entendu fonctionne, bien qu'étant différent de notre G230, parce que le descriptif de rapport utilisé par le croquis de la carte Arduino, obtenu avec :

```
$ sudo usbhid-dump -m 2341:8036
001:007:002:DESCRIPTOR          1482223010.139500
05 01 09 06 A1 01 85 02 05 07 19 E0 29 E7 15 00
25 01 75 01 95 08 81 02 95 01 75 08 81 03 95 06
75 08 15 00 25 65 05 07 19 00 29 65 81 00 C0
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Un grand nombre d'encodeurs rotatifs permettent de « cliquer » en appuyant sur la molette. Ici, j'ai simplement représenté cette fonctionnalité en ajoutant un simple bouton poussoir, connecté à la broche 4 de la carte Arduino. Que le bouton soit embarqué dans l'encodeur ou comme ici ajouté, le montage dans son ensemble ne change pas et le croquis associé non plus.

le spécifie ainsi. Rappelez-vous, les rapports correspondent au descriptif associé, mais ne sont pas identiques entre les périphériques, même de même nature. Notre souci donc va au-delà d'un simple code de touche mal utilisé. Nous avons un souci de rapport, de descriptif de rapport et c'est toute la bibliothèque `Keyboard` qui devient inutilisable. Catastrophe !

Pas de panique. Ce que peut faire `Keyboard` en reposant sur `HID` nous pouvons, nous aussi, le faire dans un croquis. Mieux encore, nous avons même un exemple puisque `Keyboard`, dans les grandes lignes, fait presque le travail qu'on lui demande de faire.

Note : j'avoue avoir un peu romancé ici le processus de recherche pour rendre cela plus attrayant et ludique. En réalité, pour avoir déjà programmé des outils reposant sur USB HID, dès les rapports affichés avec `usbhid-dump` je savais que ceux créés, formatés et envoyés par `Keyboard` n'étaient pas adaptés et qu'il y avait clairement un problème dans la souplesse de configuration de cette bibliothèque. Mais je tenais ici à décrire un processus typique de ce genre d'investigation. La réponse est toujours dans le code, toujours (« *Use the source, Luke* »).

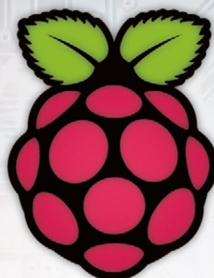
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

4. UN CONTRÔLE DE VOLUME QUI APPUIE SUR DES TOUCHES

Avant d'attaquer le croquis, résumons rapidement en quoi consiste l'objectif à atteindre. Nous souhaitons ici créer un contrôle de volume, plus ou moins comme on en trouve sur les appareils Hifi, avec une molette de réglage. Cette molette sera un encodeur rotatif que j'ai très brièvement abordé dans le numéro 6, mais que nous allons voir plus en détail. Cette molette nous permettra d'augmenter le volume en la tournant dans un sens et le réduire en tournant dans l'autre sens. À chaque *clic* dans la rotation, nous enverrons un rapport HID simulant l'action sur une touche d'un clavier multimédia. Il ne sera donc aucunement nécessaire d'installer un quelconque pilote sur l'ordinateur utilisé puisque celui-ci, quel que soit son type ou son système (Windows, GNU/Linux, macOS, etc.) sera persuadé d'avoir affaire à un simple clavier. Cela fonctionnera même avec la plupart des tablettes et smartphones (testé avec un Samsung Galaxy Nexus, un Galaxy S6 Edge et une Nvidia Shield).

Un encodeur rotatif n'est rien d'autre qu'une paire d'interrupteurs actionnés par un mécanisme rotatif. Il ne fournit donc aucune position ni aucune valeur comme le ferait, par exemple, un potentiomètre. Il n'a pas à proprement parler d'état et il ne fait que signaler des changements, des événements. Ce sont les états successifs des deux interrupteurs qui déterminent l'événement en cours, la rotation, et le sens de cette rotation.

Vend.
28/04/17



Sam.
29/04/17

Deuxièmes rencontres nationales
RASPBERRY PI
Ateliers Conférences Rencontres
NEVERS (58)
Infos : www.crrep.fr/raspberry et #RNRPI2

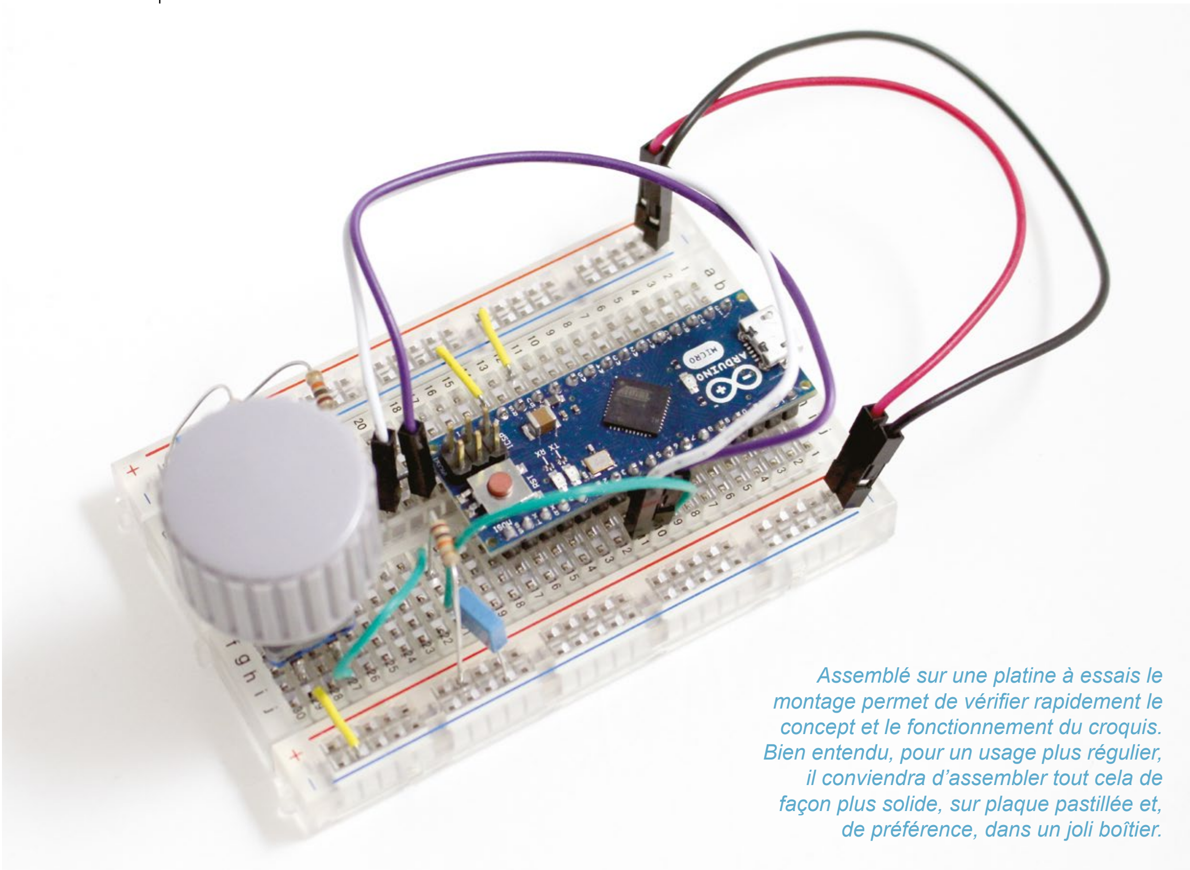
nevers AGGLOMÉRATION
CRREP Centre de Ressources en Robotique Educative et Professionnelle
Framboise 314
KUBII
Nevers cité ducale

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Pour prendre en charge un encodeur rotatif, il faut donc utiliser deux entrées d'une carte Arduino. Mais qui dit interrupteur dit également rebond. C'est un phénomène qui, à l'échelle humaine n'est pas perceptible, mais qui pour un microcontrôleur cadencé à 16 Mhz comme celui équipant une carte Arduino Micro, pose problème. Il découle du fait que, mécaniquement, le contact lors de la fermeture de l'interrupteur n'est pas franc. La lamelle métallique rebondie sur le support et génère non pas un changement d'état, mais plusieurs en un temps très court. Pour contourner le problème, on utilise généralement soit un code qui prend ce phénomène en compte, soit un montage à base de résistances et de condensateurs pour le masquer totalement.

Les broches d'un encodeur rotatif sont généralement au nombre de trois avec un commun et une broche pour chaque interrupteur/contacteur interne (A et B). Il arrive également qu'en plus de la rotation du bouton, un clic soit possible par pression. Ce contacteur supplémentaire propose alors deux autres broches pour cet usage. Ceci correspond parfaitement à nos présents besoins : la rotation pour monter/descendre le volume et le clic pour la touche à bascule « muet ».

En tournant l'encodeur, les deux interrupteurs internes changent de position et font apparaître sur les broches une succession d'états. En lisant ces deux états et en ayant connaissance des états précédents, il est possible de déterminer le sens de la rotation, la seule information utile, puisqu'il n'y a pas de notion de position absolue ici.



Assemblé sur une platine à essais le montage permet de vérifier rapidement le concept et le fonctionnement du croquis. Bien entendu, pour un usage plus régulier, il conviendra d'assembler tout cela de façon plus solide, sur plaque pastillée et, de préférence, dans un joli boîtier.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

5. LE CROQUIS ET LA BIBLIOTHÈQUE HID

Avant toutes choses, nous devons prendre en main l'utilisation de la bibliothèque `HID` sur laquelle repose `Keyboard`. Ceci n'est pas aussi complexe qu'il peut sembler au premier abord, c'est même très simple. L'utilisation de la bibliothèque `HID` se compose de deux parties. Dans un premier temps, nous devons spécifier le descriptif de rapport dans `setup()` en déclarant un objet de type `HIDSubDescriptor`, lui-même utilisant un tableau d'octets contenant les données de cette description.

Nous n'avons pas réellement besoin de nous soucier de la signification de ces octets et donc de lire des dizaines de pages de spécifications du standard HID, si nous pouvons simplement réutiliser quelque chose qui existe. Or, justement, nous avons cela sous la main : notre clavier G230. Nous avons vu précédemment que nous pouvons obtenir à la fois la description de rapport et des exemples de rapport pour chaque pression des touches qui nous intéressent. Nous n'avons donc qu'à honteusement copier ce qui existe.

Pour nous aider dans cette tâche et rendre cela plus lisible, nous pouvons nous servir du site **http://eleccelerator.com/usbdescreqparser**. Celui-ci propose, en effet, de préciser une suite d'octets telle que nous l'affiche `usbhid-dump` et nous retourne, en échange, des données bien formatées et commentées directement utilisables dans notre croquis. Notez que ceci n'est pas une absolue nécessité, mais c'est tout de même plus facile que de reprendre chaque valeur et d'ajouter manuellement des `0x` et des virgules partout (un bon programmeur est un programmeur feignant).

Via ce site nous obtenons donc de quoi remplir le tableau que nous déclarons alors ainsi :

```
static const uint8_t hidReportDescriptor[] PROGMEM = {
  0x05, 0x0C,        // Usage Page (Consumer)
  0x09, 0x01,        // Usage (Consumer Control)
  0xA1, 0x01,        // Collection (Application)
  0x85, 0x01,        //   Report ID (1)
  0x19, 0x00,        //   Usage Minimum (Unassigned)
  0x2A, 0xFF, 0x03,  //   Usage Maximum (0x03FF)
  0x15, 0x00,        //   Logical Minimum (0)
  0x26, 0xFF, 0x03,  //   Logical Maximum (1023)
  0x95, 0x01,        //   Report Count (1)
  0x75, 0x10,        //   Report Size (16)
  0x81, 0x00,        //   Input (Data,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
  0xC0,              // End Collection
  0x05, 0x01,        // Usage Page (Generic Desktop Ctrls)
  0x09, 0x80,        // Usage (Sys Control)
  0xA1, 0x01,        // Collection (Application)
  0x85, 0x02,        //   Report ID (2)
  0x19, 0x81,        //   Usage Minimum (Sys Power Down)
  0x29, 0x83,        //   Usage Maximum (Sys Wake Up)
  0x15, 0x00,        //   Logical Minimum (0)
  0x25, 0x01,        //   Logical Maximum (1)
  0x75, 0x01,        //   Report Size (1)
  0x95, 0x03,        //   Report Count (3)
  0x81, 0x02,        //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
  0x95, 0x05,        //   Report Count (5)
  0x81, 0x01,        //   Input (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
  0xC0,              // End Collection
};
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Nous pouvons alors composer notre fonction **setup()** en incluant le nécessaire pour la prise en charge de l'encodeur rotatif :

```
#include <HID.h>

#define PINA 3
#define PINB 2
#define CLIC 4

// dernier état de A
int derniereA = LOW;
// état actuel
int n = LOW;
int clic = HIGH;

void setup() {
  pinMode(PINA,INPUT);
  pinMode(PINB,INPUT);
  pinMode(CLIC,INPUT);

  static HIDSubDescriptor node(hidReportDescriptor,
sizeof(hidReportDescriptor));
  HID().AppendDescriptor(&node);
}
```

Nous déclarons un objet **node** de type **HIDSubDescriptor** en spécifiant le tableau contenant nos octets ainsi que sa taille. Un pointeur (notez-le **&**) sur **node** est ensuite utilisé en argument de la méthode **AppendDescriptor()** de la classe **HID()** pour que le descripteur de rapport soit utilisé.

Nous pouvons alors nous pencher sur **loop()**, mais avant tout nous devons faire comme **Keyboard** et prévoir une structure qui contiendra les octets composant les rapports pour chaque touche. Encore une fois, nous faisons appel à **usbhid-dump** (avec **-es**) pour capturer ce qui nous intéresse depuis le clavier G230 :

- touche « volume- » : **01 EA 00**,
- touche « volume+ » : **01 E9 00**,
- touche « muet » : **01 E2 00**,
- relâchement : **00 00 00**.

Nous parierons ici que le second rapport, provenant de l'autre interface proposée par le périphérique USB, n'est pas important, car de toute façon tous les octets sont à 0, ce qui correspond à un « aucune pression de touche ». Contrairement au clavier G230 donc, nous n'enverrons qu'un seul type de rapport.

Notre structure inspirée de celle présente dans **Keyboard** sera donc :

```
typedef struct {
  uint8_t modifiers = 0;
  uint8_t key = 0;
} KeyReportMini;
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Tout ce que nous avons à faire à présent est d'envoyer le bon contenu de rapport au bon moment avec **HID().SendReport()**, qui prend en argument un identifiant numérique (**id**) qui est le premier octet du rapport, un pointeur sur les données à envoyer (notre variable de type **KeyReportMini**) et la taille de ces données.

Voici donc notre fonction **loop()** écrite dans ce sens :

```
void loop() {
  // données du rapport
  KeyReportMini report;

  // gestion du clic "muet"
  if((digitalRead(CLIC) == LOW)) {
    if(clic == HIGH) {
      clic = LOW;
      // on change la valeur du second octet du rapport
      report1.modifiers = 0xe2;
      // envoi du rapport
      HID().SendReport(1,&report,sizeof(KeyReportMini));
      // temps de pression sur la touche
      delay(20);
      // touche relâchée
      report1.modifiers = 0x00;
      // envoi du rapport
      HID().SendReport(1,&report,sizeof(KeyReportMini));
    }
  } else {
    clic = HIGH;
  }

  // Gestion de la rotation de l'encodeur
  // lecture A
  n = digitalRead(PINA);
  // est-ce que l'état de A a changé ?
  if ((derniereA == LOW) && (n == HIGH)) {
    // oui et B est bas, donc c'est une rotation antihoraire
    if (digitalRead(PINB) == LOW) {
      // on décrémente
      report1.modifiers = 0xea;
      HID().SendReport(1,&report,sizeof(KeyReportMini));
      delay(20);
      report1.modifiers = 0x00;
      HID().SendReport(1,&report,sizeof(KeyReportMini));
    } else { // oui et B est haut, donc c'est une rotation horaire
      // on incrémente
      report1.modifiers = 0xe9;
      HID().SendReport(1,&report,sizeof(KeyReportMini));
      delay(20);
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
            report1.modifiers = 0x00;
            HID().SendReport(1,&report,sizeof(KeyReportMini));
        }
    }
    // l'état actuel devient l'état précédent
    derniereA = n;
}
```

Les commentaires dans le croquis parlent d'eux-mêmes. Les changements d'état sur les broches A et B de l'encodeur rotatif nous permettent, une fois combinés et comparés aux précédents, de savoir dans quel sens la rotation vient d'être faite, et nous envoyons les rapports en conséquence, juste après avoir modifié l'octet qui nous intéresse.

6. CE N'EST QU'UN DÉBUT

Nous venons d'apprendre plusieurs choses grâce à ce projet. Premièrement, il est important de ne pas s'en tenir aux fonctionnalités offertes par les bibliothèques standards livrées avec l'environnement. Ici nous avons vu que « Keyboard » propose certaines fonctionnalités intéressantes et tout comme sa bibliothèque « sœur » `Mouse` pour émuler un périphérique de pointage, est construit sur `HID`, mais que cette dernière bibliothèque ne dispose pas d'exemples dédiés. Il est donc très intéressant, tantôt, de prendre le temps de ne pas simplement utiliser ce qui est mis à notre disposition, mais également d'en comprendre le fonctionnement interne.

Nous avons également constaté que cette façon de voir les choses est très bénéfique et nous permet de régler des problèmes qui peuvent paraître bloquants au premier abord. La ténacité est une qualité importante chez un programmeur et un bidouilleur. Ce n'est pas parce que quelque chose semble impossible qu'il faut baisser les bras. La plupart du temps, il suffit de « descendre d'un cran » et de comprendre comment fonctionne et ce qu'utilise ce qui ne marche pas.

Enfin, et c'est peut-être là le plus important, nous avons ici utilisé davantage d'outils que ceux dont nous disposons initialement. Ce n'est pas parce que votre projet est basé sur Arduino qu'une Raspberry Pi ne peut pas vous rendre de grands services dans vos recherches. Ceci est tout aussi vrai dans ce sens que dans le sens opposé, une carte Arduino peut être d'un grand secours pour un projet basé sur une Pi. Garder l'esprit ouvert et utiliser toutes les ressources à sa disposition est souvent la meilleure approche. Rien ne vous empêche d'expérimenter sur une plateforme pour ensuite transposer le savoir acquis sur une autre.

Et bien sûr, ce que nous avons fait ici pour un simple contrôle de volume peut être fait pour n'importe quelle fonctionnalité d'un clavier multimédia ou autre. Vous pouvez associer n'importe quelle touche ou séquence de touches à n'importe quelle entrée d'une carte Arduino Micro ou Leonardo. Lecture/pause de vidéo ou de piste audio, avance rapide, contrôles associés à un lecteur multimédia comme VLC ou à n'importe quelle application... Tout ce que vous pouvez faire avec un clavier peut être transposé ainsi dans un montage utilisant des encodeurs rotatifs, des boutons, des capteurs ou encore des potentiomètres. Les possibilités sont pratiquement illimitées... **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

À PARTIR DU 1ER MARS, CONNECT ÉVOLUE !

LISEZ CE NUMÉRO ET PLUS DE 15 AUTRES EN LIGNE !

ACTUELLEMENT SUR CONNECT :

- **CE NUMÉRO**
- **et + de 15 autres numéros de Hackable**

- **1 numéro Hors-Série de Hackable**

TOUT CELA À PARTIR DE 149 € TTC*/AN !

* Tarif France Métropolitaine

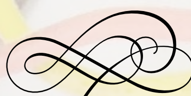
Rendez-vous sur **connect.ed-diamond.com** pour découvrir Connect !

Pour tous renseignements complementaires, contactez-nous :
· via notre site internet : **www.ed-diamond.com**
· par téléphone : **03 67 10 00 20**
ou envoyez-nous un mail à **connect@ed-diamond.com** !

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

TRANSFORMEZ UN VIEUX MATÉRIEL DE 30 ANS EN CLAVIER USB

Denis Bodor



Rien se perd, tout se transforme... en particulier quand on prend l'habitude de ne rien jeter qui pourrait, un jour, servir à quelque chose. C'est précisément le cas de ce clavier IBM XT de 1985 qui attendait patiemment dans mon grenier que je daigne lui donner une seconde vie. L'article qui va suivre est tout autant une description technique de l'adaptation d'un clavier, qu'un retour d'expérience pouvant, je l'espère, vous aider dans vos propres expérimentations.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

L'histoire commence avec un terrible regret, celui d'avoir abandonné il y a quelques années déjà, à l'occasion d'un déménagement, un magnifique IBM PC 5160, véritable point de départ de l'ère « PC » dans laquelle nous nous trouvons encore. En effet, dans votre machine dernier cri se trouvent encore des éléments, réels ou émulés, hérités de cette machine légendaire dont la production a commencée en 1981. Regardez simplement votre clavier par exemple, non loin du pavé numérique se trouve une touche, « Arrêt défil » (*Scroll Lock*), quasiment inutilisée de nos jours (sauf rares exceptions), celle-ci est un héritage de l'IBM PC original (qui l'a lui-même hérité des terminaux IBM). Ceci n'est qu'un exemple visible, le bus ISA du tout premier PC, par exemple, est toujours présent même sur les architectures PC 64 bits actuelles, ce qui n'est bien entendu pas le cas avec une Raspberry Pi (qui n'est donc pas à proprement parler un PC).

Mais l'ensemble, avec écran, unité centrale et clavier pesant quelques 22 kilos et occupant une place non négligeable, n'est pas le genre d'objet qu'il est facile de conserver et celui-ci a donc, il y a fort longtemps et à contrecœur, été abandonné à son sort. Je ne saurai exprimer à quel point je regrette cette décision aujourd'hui, mais je trouve une certaine forme de réconfort dans le fait d'avoir, tout de même, conservé le clavier, qui pour une raison inconnue s'est cependant vu amputé de son câble (sans doute pour en faire un connecteur pour un Commodore 64).

Quoi qu'il en soit, j'avais donc en ma possession un fantastique, magnifique, sale et pesant (~3 kg) clavier IBM modèle « F » XT de quelques 30 ans d'âge ne demandant qu'à reprendre du service, en faisant un adorable « clic » à chaque pression d'une touche. Mais les choses ne sont pas aussi simples que cela...

1. PRISE DE CONTACT ET FLORILÈGE D'ERREURS

Une trentaine d'années, ce n'est pas grand-chose à l'échelle de l'Histoire, un peu davantage à l'échelle d'une vie, mais une véritable éternité lorsqu'on parle d'informatique. Dites-vous bien qu'il y a 30 ans, il n'y avait pas d'Internet pour le grand public, pas de réseau Ethernet, pas de Wifi... que le Macintosh venait de naître et que la souris tout comme les disques durs étaient encore quelque chose de relativement rare pour un chanceux (et riche) propriétaire de PC.

Voici la bête ! Un clavier IBM modèle « F » pour PC/XT de 1985. C'est, tristement, tout ce qu'il me reste de la machine originale qui est maintenant une véritable pièce de musée. Ce genre de choses se traite avec respect et humilité, car c'est littéralement un morceau d'histoire...

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Le clavier mériterait en principe un bon nettoyage méticuleux et prudent. Si vous en doutiez ou l'ignoriez, oui, le terme « PC » est bel et bien une référence directe au Personal Computer d'IBM. C'est même marqué dessus !

La première étape, quasi archéologique, consiste donc à retrouver le dialecte utilisé à l'époque entre le clavier et l'unité centrale. Instinctivement, et après une recherche bien trop sommaire sur le Web, on a tendance à tirer des conclusions hâtives. Ainsi, sachant qu'une carte Arduino est en mesure de prendre en charge un clavier de type PS/2 et que, de mémoire, il existait des adaptateurs DIN vers PS/2 permettant de connecter un ancien clavier à une machine moderne (pré-2000), on est enclin à penser qu'il n'y a pas de grandes différences entre ces différents types de claviers.

C'est là une grossière erreur, car même si effectivement, la bibliothèque *PS2Keyboard* est bien en mesure d'interpréter les messages d'un clavier à l'aide de deux simples connexions, ceci se limite au seul protocole PS/2 et donc à des périphériques qui ne sont de loin pas aussi vieux.

Mon premier conseil donc pour vous éviter bien des ennuis dans ce genre d'expérimentations est de ne surtout pas faire de raccourcis et ne pas supposer des vérités là où vous n'avez absolument aucune preuve pour les étayer.

Ces premières tentatives d'utilisation avec la bibliothèque *PS2Keyboard* ont également été l'occasion d'une autre bonne leçon que je vous invite à retenir : ne faites JAMAIS confiance à la couleur des câbles !

Le clavier, qui s'avère être une véritable pièce de collection, a bien failli vivre ses derniers jours lors de la première connexion à une carte Arduino. Après un démontage de la bête par retrait du panneau métallique arrière retenu par deux énormes vis, la surprise fut grande de constater que le circuit imprimé animant l'objet dispose d'un connecteur interne. La coupure du câble n'était donc pas un problème grave nécessitant soudure, mais simplement l'affaire d'une connexion aisée à l'aide de câbles/jumpers femelles.

Le connecteur 10 broches (2×5) en question et son morceau de câble présentaient 4 connexions : rouge, noir, blanc et brun. En toute logique, le rouge est synonyme d'alimentation et le noir de masse... Grave erreur que celle-ci, car chez IBM on ne semblait, à l'époque, n'accorder que peu d'intérêt à ce genre de convention. La connexion erronée du clavier à la carte Arduino n'apportant aucun résultat, une vérification des connexions me donna l'occasion de constater que le circuit intégré du clavier semblait anormalement chaud. Là, il n'y a aucune question à se poser, ce genre de phénomène n'appelle qu'à une seule réaction : tout déconnecter dans un mouvement de panique totale !

À ce stade, la conclusion est évidente et mon conseil à votre attention tout autant : ne faites pas confiance à la couleur des câbles, ni selon les conventions d'usage, ni selon les documentations en ligne et assurez-vous toujours que la masse est là où vous pensez qu'elle se trouve.

La masse est généralement facile à trouver puisque présente un peu partout dans un périphérique et généralement reliée à toutes les masses métalliques en présence (blindage, etc.). Un simple multimètre en mode « continuité » vous permettra de vérifier si un connecteur est effectivement relié à la masse. Dans le cas de ce clavier, l'énorme plaque métallique servant de support et de châssis

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

aux touches est reliée au circuit imprimé et à la broche centrale du connecteur, initialement branché... au câble rouge ! Oui, vous avez bien lu, le rouge était la masse et, de ce fait, j'ai donc appliqué +5V directement à cet endroit, risquant de détruire le circuit intégré !

Une fois la masse trouvée, l'étape suivante consiste à faire de même pour la tension d'alimentation. Là, en l'absence de schéma ou de documentation technique des composants, il faut procéder par déduction et vérifier ses suppositions. En toute logique, un tel circuit, connecté à l'aide d'un long câble doit s'assurer que la tension d'alimentation est exempte de « bruit ». Il y a donc forcément un condensateur placé entre la tension d'alimentation et la masse, non loin du connecteur.

La masse étant identifiée, nous pouvons également la retrouver sur les circuits intégrés présents. C'est le cas, par exemple, des deux circuits logiques présents à côté de l'énorme contrôleur clavier : deux circuits intégrés de 14 pattes, dont un 74239 de *Texas Instruments* (un décodeur/démultiplexeur 2-vers-4). Ce type de composants a un brochage standardisé avec la masse en bas à gauche et la tension d'alimentation en haut à droite. Il suffit alors d'utiliser, encore une fois, le testeur de continuité pour :

- vérifier que les deux circuits intégrés sont connectés par leur masse et à la masse,
- qu'ils sont connectés par la broche en haut à droite supposée être la tension d'alimentation (VCC),
- qu'au moins un des condensateurs est à la fois branché à la masse et à la tension d'alimentation supposée et que ceci correspond effectivement à l'éventuel marquage sur ce condensateur.

Après cette procédure, il s'est avéré que le câble original de couleur brune était la tension d'alimentation et que le câble noir était connecté à un signal. J'avais donc connecté ma masse à un signal en sortie du clavier, ce qui, là encore, aurait pu poser un problème (moindre que l'alimentation sur la masse, certes).

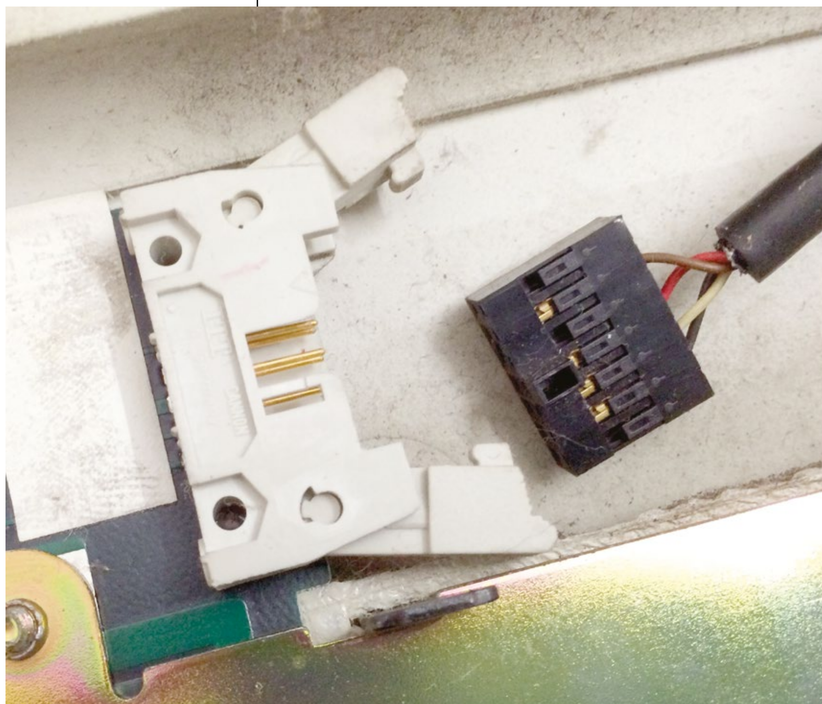
À l'intérieur du clavier, un simple circuit imprimé double face, un énorme et solide support pour les touches et, non présents sur la photo faite après nettoyage, 30 ans de poussière accumulée, des miettes à profusion et une araignée séchée (d'époque sans doute).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Une fois deux des connecteurs identifiés, il ne reste plus que les signaux. Wikipédia, sur la page anglaise concernant l'*IBM PC keyboard*, nous indique qu'il n'existe qu'une seule configuration utilisant uniquement 4 connexions pour les claviers PC (en dehors de l'USB) : le clavier du PC XT type 2. Cette page indique également que les deux autres connexions sont destinées aux signaux d'horloge et de données provenant du clavier et à destination du PC, et que la communication est unidirectionnelle (pas de communication du PC vers le clavier).

Le lecteur attentif remarquera sans doute qu'il aurait été plus simple d'observer le circuit imprimé pour retrouver les connexions plus rapidement. Ceci est parfaitement vrai dans de nombreux cas, mais malheureusement difficilement applicable lorsqu'une masse de mousse tout aussi vieille que le matériel lui-même est collée sur l'une des faces du circuit. Je n'ai pas souhaité retirer cette mousse fragile pour ne pas avoir à trouver une solution pour la remplacer au moment de refermer définitivement le périphérique.

Une charmante attention du constructeur : le circuit imprimé dans le clavier est relié au câble par un connecteur d'excellente facture et non simplement soudé. C'est normal, fut un temps, le matériel informatique était prévu pour pouvoir être réparé. Fut un temps...



2. RETROUVER UNE LANGUE PARLÉE IL Y A 30 ANS

Après reconnexion correcte de l'alimentation et de la masse, l'aventure ne fait que commencer. Voyant le fait de n'avoir pas détruit l'électronique du clavier comme un signe du destin très encourageant, c'est à ce moment que j'ai commis une autre erreur : supposer que « qui peut le plus peut le moins » ou en d'autres termes que le standard PS/2 devait sans doute être compatible avec les claviers AT qui eux-mêmes devaient l'être avec les claviers XT.

Niveau connectique, et ne sachant pas quel câble était le signal d'horloge et lequel était destiné à la transmission des données, je me suis naïvement dit qu'il me suffirait de tester les deux combinaisons possibles avec la bibliothèque *PS2Keyboard*. Le résultat fut... étonnant. Quelle que soit la connexion, le clavier semblait bel et bien émettre des données, mais de façon incohérente. Une pression sur une touche ne donnait pas de résultats, mais une série de pressions sur plusieurs touches oui. De la même façon, une action répétée sur une même touche affichait également un résultat dans le terminal série, mais... sans cesse différent !

Un coup d'œil sur la page Wikipédia précédemment citée montre en effet une grande différence entre le protocole utilisé par un clavier XT, 2 bits de *start*, 8 bits de données et un bit de stop, et celui utilisé par un clavier PS/2, 1 bit de *start*, 8 bits de données, un bit de parité, et un bit

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

de stop. Ce dernier format/protocole est également celui des claviers AT, ce qui explique pourquoi un adaptateur DIN-5 vers miniDIN-6 permettait aux propriétaires de claviers AT d'utiliser ce matériel sur des machines plus récentes avec un port miniDIN (qui existe encore tantôt sur les machines actuelles).

C'est à ce stade qu'on peut se rendre compte que la bibliothèque *PS2Keyboard* ne pourra pas être utilisée dans l'état et que sa modification n'apportera pas forcément une solution. Il en va de même pour la bibliothèque *PS2KeyAdvanced* plus souple d'utilisation et offrant davantage de fonctionnalités, mais comportant le même problème : au niveau le plus bas, les données émises par le clavier n'ont pas le bon format.

La solution consiste donc à développer un croquis permettant, au minimum, de régler un premier problème : distinguer le signal d'horloge de celui des données.

Le principe est relativement simple, nous avons un connecteur qui change d'état régulièrement pour « donner la cadence » et signaler qu'un bit peut être lu. L'autre connecteur présente le bit en question. Sur la base des informations données par Wikipédia, nous sommes donc censés pour chaque « événement » (pression ou relâchement d'une touche), avoir 11 bits à lire et donc 11 changements d'état de la ligne d'horloge. Là encore, faire des raccourcis maladroits et croire tout ce qu'on lit ou pense est ce qui fera perdre un temps non négligeable.

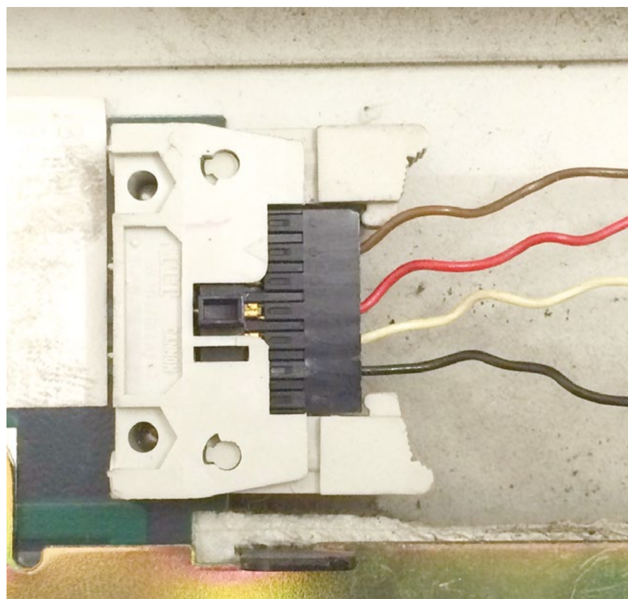
L'approche à adopter ici consiste à écrire un croquis qui va scruter l'état de ce qu'on suppose être la ligne d'horloge et à chaque changement d'état bas vers haut (0V à +5V), lire l'état de la broche supposée des données pour collecter chaque bit et former une valeur entière (`int`) sur 11 bits.

Deux façons de faire peuvent être envisagées : littéralement scruter l'état de la broche dans une boucle (technique du *polling*) ou utiliser une interruption qui déclenchera l'exécution d'une fonction en cas de changement d'état. Cette seconde solution étant la plus élégante, c'est donc celle qui sera utilisée avec le croquis suivant :

```
#define CLOCK 2
#define DATA  8

// données cumulées
volatile int b_data;
// compteur d'horloge
volatile int cpt;
// drapeau prêt/ready
volatile int rdy;

// routine d'interruption
void clksig() {
  b_data |= digitalRead(DATA) << cpt ;
  cpt++;
  if(cpt > 10) {
    cpt=0;
    rdy=1;
  }
}

void setup() {
  Serial.begin(115200);
  while(!Serial){;}
  Serial.println("Go go go!");

  pinMode(CLOCK, INPUT_PULLUP);
  pinMode(DATA, INPUT_PULLUP);

  // installation de la routine
  attachInterrupt(digitalPinToInterrupt(CLOCK), clksig, RISING);
}

void loop() {
  // données prêtes ?
  if(rdy) {
    // horodatage
    Serial.print(millis());
    Serial.print(" : ");
    // bits collectés
    Serial.println(b_data,BIN);
    rdy=0;
    b_data=0;
  }
}
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Là, je ne comprends pas. Qu'est-ce qui peut pousser quelqu'un à décider, quelque part dans un laboratoire ou une usine, qu'associer un câble de couleur rouge à la masse est une bonne idée ?! Car c'est un fait, le fil rouge est ici la masse, le brun la tension d'alimentation (+5V) et le noir comme le blanc des signaux TTL.



Nous configurons ici la carte Arduino de manière à appeler `clksig()` dès que l'état de la broche 2 passe de bas à haut (`RISING`) avec `attachInterrupt()` et en précisant le nom de l'interruption grâce à la valeur retournée par `digitalPinToInterrupt()` à qui on passe en argument le numéro de broche concernée.

J'utilise ici une carte Arduino Micro, car l'objectif est, à terme, de faire apparaître le montage comme un clavier USB. Sur ce modèle de carte, seules les broches 0, 1, 2, 3 et 7 peuvent être utilisées pour ce type de choses. Pour une UNO, seules les broches 2 et 3 peuvent ainsi générer des interruptions.

Lorsque `clksig()` est appelée, nous lisons l'état de la broche 8 et stockons chaque bit obtenu décalé du nombre de fois où la routine a été appelée. Le premier bit sera donc le bit de poids le plus faible (totalement à droite dans la variable de type `int`) et le 11ème sera à la position 10. Si le compteur atteint 11 c'est que le prochain bit devrait être le 12ème et, dans ce cas, notre valeur lue est complète. Nous repassons `cpt` à 0 et plaçons 1 dans `rdy` (*ready*).

Dans la boucle principale `loop()`, nous nous contentons d'attendre que `rdy` soit non nul, indiquant qu'une valeur complète est lue. Si tel est le cas, nous affichons cette valeur en binaire dans le moniteur série et réinitialisons à la fois la variable contenant la valeur et `rdy` à zéro pour collecter les prochains bits.

Tout semble en ordre, il ne nous reste plus qu'à tester le croquis en appuyant de façon répétée sur une même touche pour obtenir des données. Ce qui nous donne :

```
Go go go!
3919 : 1001011
4711 : 1010100010
4817 : 1100000001
5274 : 110000000
5365 : 1011000010
5913 : 101100001
6005 : 10110010
6766 : 1001011000
6873 : 100101100
7527 : 10010110
```

Voilà qui n'est pas du tout ce que nous espérions. La même touche, enfoncée et relâchée, doit OBLIGATOIREMENT envoyer les deux mêmes valeurs (une pour l'appui et une autre pour le relâchement). Là, vous avons à chaque fois une autre valeur, ça n'a pas de sens. Ce n'est pas grave, les signaux d'horloge et de données doivent simplement être inversés. Second essai et... non, c'est encore pire ! En inversant les broches, tantôt nous n'avons pas de données du tout et les valeurs sont encore plus changeantes. Quelque chose va vraiment de travers...

3. PARFOIS UN PEU D'AIDE NE FAIT PAS DE MAL

À ce stade, la confusion et le doute règnent en maîtres. L'inversion de polarité lors du premier branchement a-t-elle endommagé l'électronique du clavier ? S'agit-il d'une autre forme de communication ? Notre Arduino a-t-il un problème ? Ou... est-on simplement allé trop vite en besogne ?

Dans ce genre de situations, il faut savoir revenir à la base et

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Abonnez-vous !

HACKABLE MAGAZINE

M'abonner ?

Me réabonner ?

Compléter ma collection en papier ou en PDF ?

Pouvoir consulter la base documentaire de mon magazine préféré ?

C'est simple... c'est possible sur

http://www.ed-diamond.com

... OU SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE AU VERSO ET RENVOYEZ CE DOCUMENT COMPLET À L'ADRESSE CI-DESSOUS !

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

❑ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : http://**boutique.ed-diamond.com/content/3-conditions-generales-de-ventes** et reconnais que ces conditions de vente me sont opposables.

HACKABLE MAGAZINE

Les Éditions Diamond
Service des Abonnements
10, Place de la Cathédrale
68000 Colmar – France
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

....

....

Ce document est la propriété exclusive de Alex Arnaud(bahnuxdroid@gmail.com)

Bon d'abonnement

CHOISISSEZ VOTRE OFFRE !

SUPPORT

Prix TTC en Euros / France Métropolitaine*

Offre	ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Tarif TTC
HK	6 n° Hackable	☐ HK1	39,-	☐ HK13	169,-
	LES COUPLAGES AVEC NOS AUTRES MAGAZINES				
i	6 n° Hackable + 6 n° MISC	☐ i1	79,-	☐ i13	419,-
i+	6 n° Hackable + 6 n° MISC + 2 n° Hors-Série	☐ i+1	99,-	☐ i+13	439,-
J	11 n° GNU/Linux Magazine France + 6 n° Hackable	☐ J1	105,-	☐ J13	399,-
J+	6 n° Hackable + 11 n° GNU/Linux Magazine France + 6 n° Hors-Série	☐ J+1	159,-	☐ J+13	459,-
K	6 n° Hackable + 6 n° Linux Pratique	☐ K1	75,-	☐ K13	329,-
K+	6 n° Hackable + 6 n° Linux Pratique + 3 n° Hors-Série	☐ K+1	99,-	☐ K+13	359,-
	LA TOTALE DIAMOND !				
L	11 n° GLMF + 6 n° HK* + 6 n° LP + 6 n° MISC	☐ L1	189,-	☐ L13	839,-
L+	11 n° GLMF + 6 n° HS + 6 n° HK* + 6 n° LP + 3 n° HS + 6 n° MISC + 2 n° HS	☐ L+1	289,-	☐ L+13	939,-

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | HK = Hackable

Particuliers = **CONNECTEZ-VOUS SUR :**

http://www.ed-diamond.com

pour consulter toutes les offres !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

Professionnels = **CONNECTEZ-VOUS SUR :**

http://proboutique.ed-diamond.com

pour consulter toutes les offres !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

vérifier ce sur quoi on aura basé notre expérience. Nous avons besoin ici de vérifier deux choses : où est le signal d'horloge et combien de bits sont effectivement envoyés lors d'une pression sur une touche.

La méthode la plus simple est de bénéficier de l'aide d'un oscilloscope ou d'un analyseur logique (voir *Hackable n°5*). En branchant les sondes de l'appareil sur les deux connecteurs mystérieux, nous aurons alors très rapidement les deux réponses en une fois. Dans les faits c'est précisément ce que j'ai décidé de faire et sans doute par là que j'aurai dû commencer. Il y a ici aussi une leçon à retenir : utiliser tout l'équipement à sa disposition pour collecter des informations avant de faire quoi que ce soit.

Comme il est possible que vous ne disposiez pas d'un tel matériel, que ce soit un oscilloscope numérique (~350€) ou un analyseur logique (~120€ pour un *Salae Logic* 4 canaux et ~20€ pour un clone), voici une méthode alternative. Ce que nous cherchons à détecter c'est un nombre récurrent de changements d'état, quelle que soit la touche du clavier utilisée. Les données ne nous intéressent pas, nous voulons simplement compter les *ticks* d'horloge. Or, pour cela, un croquis Arduino suffit.

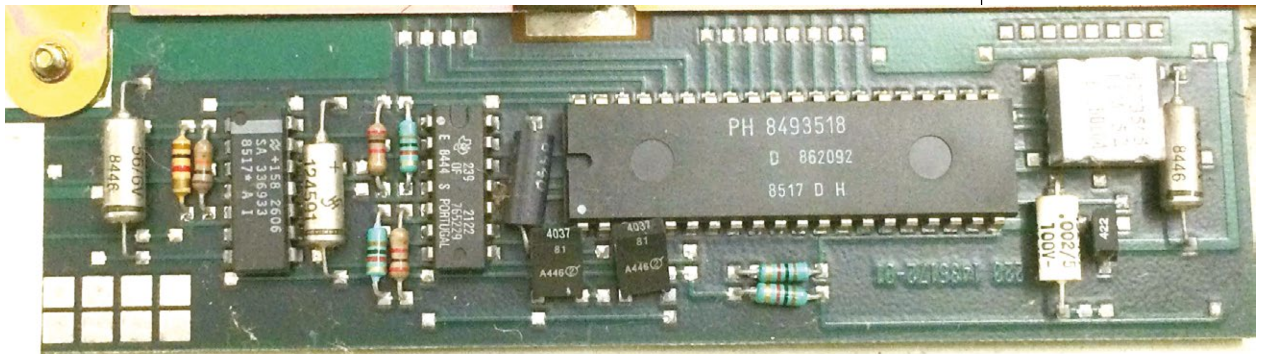
Nous pouvons donc modifier notre croquis initial ainsi :

```
void clksig() {
  cpt++;
}
```

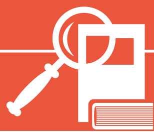
et

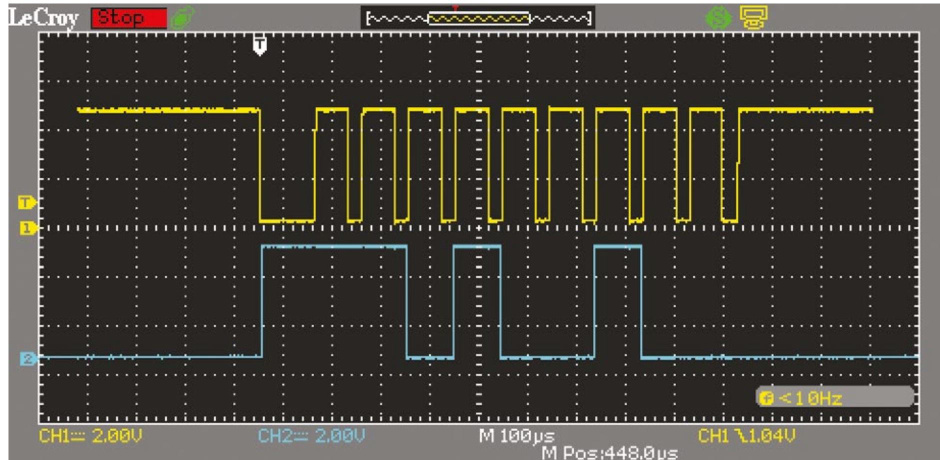
```
unsigned long previousMillis = 0;
const long interval = 10;

void loop() {
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    if(cpt) {
      Serial.print(millis());
      Serial.print(" : ");
      Serial.println(cpt);
      cpt=0;
    }
  }
}
```

Retrouver la masse n'est pas difficile, elle est présente un peu partout. Pour la tension d'alimentation, il faut s'aider du condensateur sur la gauche et du brochage des deux circuits logiques de 14 pattes à gauche de l'énorme contrôleur.



Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)





Pour trouver la signification et l'usage des connexions restantes, le plus simple est encore d'utiliser un oscilloscope en réglant un niveau de déclenchement (trigger) et une capture unique (single shot). On voit ici en haut en jaune le signal d'horloge et en bas en cyan les données découlant de la pression sur la touche « K ».

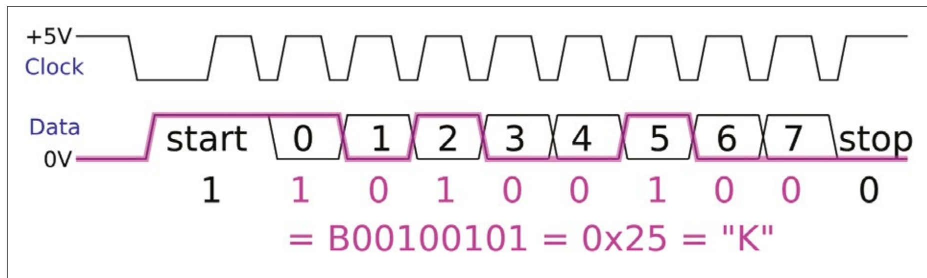
Notre routine d'interruption ne fait plus qu'une seule chose : incrémenter une variable `cpt`. Nous reposons sur la fonction `millis()` pour régulièrement afficher et remettre à zéro cette variable (si elle est différente de 0). Il ne s'agit ni plus ni moins que de l'exemple *BlinkWithoutDelay* de l'environnement Arduino avec, en prime, notre routine d'interruption.

Après enregistrement du croquis dans la carte, une broche nous donne des valeurs différentes à chaque fois, mais l'autre nous affiche ceci :

```
11650 : 10
11791 : 10
12704 : 10
12855 : 10
14090 : 10
14240 : 10
15154 : 10
15335 : 10
```

Ah ! Voici donc notre signal d'horloge, constant, propre et bien discipliné. Les changements d'état sur la broche de données sont forcément souvent différents d'une touche à l'autre, en fonction du nombre de successions de 1 ou de 0, mais l'horloge, elle, est régulière. Nous avons donc identifié notre signal et c'est cette broche qui devra être connectée à la broche 2 de la carte Arduino.

L'autre point intéressant est la valeur affichée : 10. Que l'interruption soit déclenchée par un changement de bas vers haut (`RISING`) ou de haut vers bas (`FALLING`), le résultat est le même. Nous avons 10 flancs montants ou descendants et donc 10 bits, non 11 ! Donc, soit il n'y a pas deux bits de *start* (destinés à signaler l'arrivée de données), mais un, soit il n'y a pas de bit de stop. On peut raisonnablement penser que les bits de données sont effectivement au nombre de 8, pour former un octet (mais là aussi ça pourrait être une erreur).



La capture faite à l'oscilloscope permet de confirmer une des sources de documentation sur l'encodage des données. Ici le diagramme reproduit à partir des informations de kbdbabel.org avec, en rose, le code correspondant à la touche K, en tout point similaire au signal capturé. C'est la lumière au bout du tunnel... Enfin !

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Pourtant, une telle configuration n'existe pas d'après Wikipédia et il faut aller creuser dans le fin fond d'Internet pour trouver un autre son de cloche. Après une bonne heure de recherche avec, sous les yeux mon écran d'oscilloscope, je finis alors par tomber sur **http://www.kbdbabel.org**. D'aspect spartiate, le site pourtant regorge d'informations techniques très intéressantes à propos des protocoles de divers claviers du Macintosh à l'Apple Lisa en passant par l'Amiga, le Tandy 1000, l'AT ou encore l'IBM PC/XT. Et là, surprise ! Il est effectivement question de 10 bits avec 1 bit de *start*, 8 de données et 1 de stop.

Encore une leçon qui s'ajoute à celles déjà reçues : ne jamais faire confiance à une source unique d'information fut-elle Wikipédia.

Une cherche complémentaire à propos du flanc me dirige vers un forum où un utilisateur en 2006 confirme mes soupçons et ce que montre à la fois mon oscilloscope et **kbdbabel.org** : les données sont valides sur un flanc montant du signal d'horloge, contrairement aux claviers AT qui utilisent le flanc descendant. Chose confirmée, de plus, par la « *PC Keyboard FAQ* » (**http://ilkerf.tripod.com/c64tower/F_Keyboard_FAQ.html**), un fichier texte de 1995 regroupant également une masse impressionnante de données (deux demi-leçons de plus, toujours privilégier les données au plus proche de l'époque concernée et plus c'est moche, mieux c'est !).

Fort de ces informations, il ne nous reste plus qu'à valider tout cela avec une nouvelle évolution de notre croquis en adaptant notre routine d'interruption :

```
// routine d'interruption
void clksig() {
  // on saut le bit de start
  if(cpt > 0)
    // stockage
    b_data|=digitalRead(DATA) << cpt-1 ;
  cpt++;
  // on ignore le bit de stop
  if(cpt > 9) {
    cpt=0;
    rdy=1;
  }
}
```

Cette fois, nous gérons le bon nombre de bits et ajoutons une condition pour sauter le premier bit. Nous ne stockons donc dans `b_data` que les 8 bits de données pour former une valeur. Dès que nous arrivons au bit de stop, notre variable est utilisable et ceci est traité dans `loop()`, que nous étoffons un peu au passage :

```
void loop() {
  // données prêtes ?
  if(rdy) {
    // horodatage
    Serial.print(millis());
    Serial.print(" : ");
    // en hexa
    Serial.print(b_data,HEX);
    Serial.print(" : ");
    // en décimal
    Serial.print(b_data,DEC);
    Serial.print(" : ");
    // en binaire
    Serial.println(b_data,BIN);
    rdy=0;
    b_data=0;
  }
}
```

La valeur hexadécimale affichée va nous permettre de vérifier la mention dans l'image présentée sur **kbdbabel.org** montrant que la touche « K » correspond à la valeur `0x25`. Et effectivement :

```
Go go go!
2113 : 25 : 37 : 100101
2189 : A5 : 165 : 10100101
2585 : 25 : 37 : 100101
2646 : A5 : 165 : 10100101
3102 : 25 : 37 : 100101
3163 : A5 : 165 : 10100101
3604 : 25 : 37 : 100101
3650 : A5 : 165 : 10100101
```

Victoire !

Notez dans le résultat obtenu que le bit de poids le plus fort (à gauche) indique s'il s'agit d'une pression sur la touche (`0`) ou d'un relâchement de celle-ci (`1`). Les 7 autres bits sont identiques. `0xA5` est `0x25` avec le bit 7 à 1.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

4. DIRECTION USB !

Nous avons maintenant un résultat probant et utilisable. Chaque touche enfoncée ou relâchée peut être identifiée par son code et associée à un caractère. Mais nous n'avons en réalité pas besoin de faire une telle chose, car même si la bibliothèque *Keyboard* disponible pour les cartes Arduino Micro et Leonardo permet d'envoyer directement du texte comme si celui-ci avait été saisi au clavier nous pouvons trouver un raccourci.

En effet, chaque touche d'un clavier est associée à un symbole par l'ordinateur et uniquement l'ordinateur. Du point de vue du clavier lui-même, la sérigraphie des touches importe peu, ce qui compte c'est le code de la touche et plus exactement le *scancode*. À la charge de l'ordinateur de faire le rapprochement avec le symbole adéquat. Ceci permet l'utilisation de plusieurs organisations de clavier sans pour autant revoir l'électronique d'un périphérique. Seule la phase finale d'assemblage en usine change et c'est l'impression des caractères sur les touches qui détermine la « nationalité » du clavier.

Le travail à faire n'en sera pas moins pénible pour obtenir un clavier USB. Il faut, en effet, utiliser le précédent croquis et associer le numéro obtenu pour chaque touche avec le nom de la touche tel que spécifié dans la bibliothèque que nous allons utiliser.

À ce propos d'ailleurs, même si la bibliothèque *Keyboard* est fort sympathique pour des petits projets, elle possède un certain nombre de limitations. Nous l'avons vu dans un autre article de ce numéro, certaines touches manquent à l'appel par exemple. Ici, plutôt que de nous amuser à jouer avec des rapports USB HID comme précédemment, nous allons reposer sur une autre bibliothèque : *HID-Project* de NicoHood (cf. l'article sur HoodLoader2).

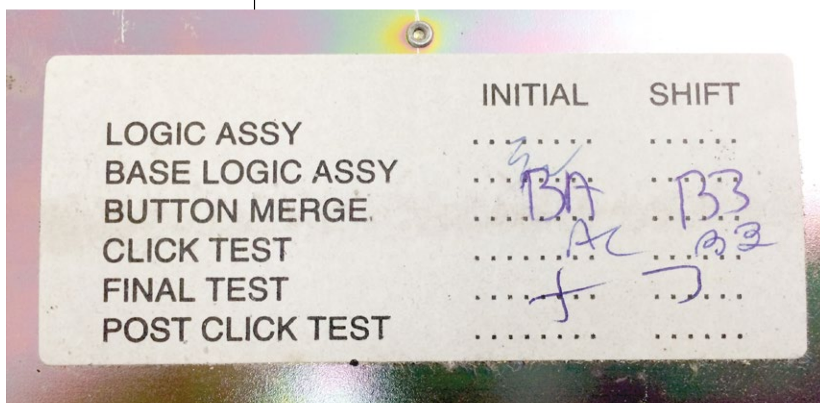
Celle-ci est disponible via le gestionnaire de bibliothèques de l'environnement Arduino et s'installera donc d'un simple clic. Le principal avantage de cette bibliothèque, dans le cas qui nous occupe ici, est la définition d'une énumération (ou liste énumérée) dans **ImprovedKeylayouts.h**. Celle-ci contient les noms de toutes les touches imaginables et leur valeur associée sous la forme de **KeyboardKeycode** (qui ne sont pas les *scancodes* de notre clavier).

La tâche pénible consiste donc à faire ceci 83 fois :

- appuyer sur une touche pour obtenir son code,
- chercher dans **ImprovedKeylayouts.h** le nom correspondant dans l'énumération,
- placer ce nom dans un tableau que nous intègrerons à notre croquis, dans l'ordre croissant des codes.

Après quelques looooooongues et éprouvantes dizaines de minutes et de nombreuses fautes de frappe et d'inattention, nous finissons pas obtenir ceci :

L'arrière du châssis clavier possède une étiquette avec le paraphe des personnes ayant procédé aux tests. Je ne peux m'empêcher de me demander si ces gens se doutaient un seul instant que 32 ans plus tard le fruit de leur travail serait encore utilisable et utilisé...



Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
const byte xtmap[] = {
  KEY_RESERVED, KEY_ESC, KEY_1, KEY_2, KEY_3, KEY_4, KEY_5,
  KEY_6, KEY_7, KEY_8, KEY_9, KEY_0, KEY_MINUS, KEY_EQUAL,
  KEY_BACKSPACE, KEY_TAB, KEY_Q, KEY_W, KEY_E, KEY_R, KEY_T,
  KEY_Y, KEY_I, KEY_U, KEY_O, KEY_P, KEY_LEFT_BRACE,
  KEY_RIGHT_BRACE, KEY_RETURN, KEY_LEFT_CTRL, KEY_A, KEY_S,
  KEY_D, KEY_F, KEY_G, KEY_H, KEY_J, KEY_K, KEY_L, KEY_SEMICOLON,
  KEY_QUOTE,KEY_TILDE, KEY_LEFT_SHIFT, KEY_BACKSLASH, KEY_Z,
  KEY_X,  KEY_C, KEY_V, KEY_B, KEY_N, KEY_M, KEY_COMMA,
  KEY_PERIOD, KEY_SLASH, KEY_RIGHT_SHIFT, KEY_PRINTSCREEN,
  KEY_LEFT_ALT, KEY_SPACE, KEY_CAPS_LOCK, KEY_F1, KEY_F2,
  KEY_F3, KEY_F4, KEY_F5, KEY_F6, KEY_F7, KEY_F8, KEY_F9,
  KEY_F10, KEY_NUM_LOCK, KEY_SCROLL_LOCK, KEYPAD_7, KEYPAD_8,
  KEYPAD_9, KEYPAD_SUBTRACT, KEYPAD_4, KEYPAD_5, KEYPAD_6,
  KEYPAD_ADD, KEYPAD_1, KEYPAD_2, KEYPAD_3, KEYPAD_0, KEYPAD_DOT
};
```

Vous trouvez ça difficile à lire ? Je vous rassure, ce n'est rien à côté du fait de l'écrire ! La version téléchargeable du croquis complet dans le dépôt GitHub de ce numéro inclut une version plus lisible, parce que je suis quelqu'un de vraiment très gentil...

Cette variable **xtmap** peut être déclarée directement dans le croquis, mais j'ai préféré le faire dans un fichier **xtmap.h** qui prendra place dans un nouvel onglet de l'environnement Arduino. L'autre onglet, quant à lui, contiendra notre croquis :

```
#include <HID-Project.h>

#include "xtmap.h"

#define CLOCK 2
#define DATA  8

// données cumulées
volatile byte touche;
// compteur d'horloge
volatile int cpt;
// drapeau prêt/ready
volatile int rdy;

// routine d'interruption
void clksig() {
  // on saute le bit de start
  if(cpt > 0)
    // stockage
    touche |= digitalRead(DATA) << cpt-1 ;
  cpt++;
  // on ignore le bit de stop
  if(cpt > 9) {
    cpt=0;
    rdy=1;
  }
}
```

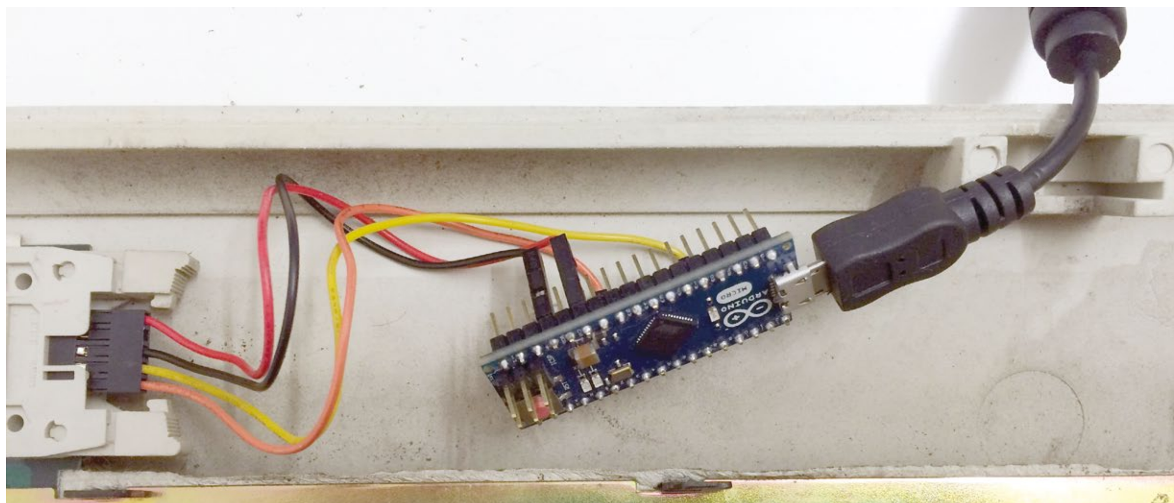
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
void setup() {
  Serial.begin(115200);

  // bloquant si aucune connexion série
  //while(!Serial){;}
  delay(2000);
  Serial.println("Go go go!");

  Keyboard.begin();

  pinMode(CLOCK, INPUT_PULLUP);
  pinMode(DATA, INPUT_PULLUP);

  delay(2000);

  // installation de la routine
  attachInterrupt(digitalPinToInterrupt(CLOCK), clksig, RISING);
}

void loop() {
  // données prêtes et touche valide ?
  if(rdy && touche > 0 && touche < 83) {
    // scancode
    Serial.print("0x");
    Serial.print(touche,HEX);
    Serial.print(": touche [");
    // KeyboardKeycode
    Serial.print(xtmap[touche & 127]);
    Serial.print("]");
    if(touche & 128) {
      // si bit 7 alors relâchement
      Serial.println(" relacher");
      Keyboard.release(KeyboardKeycode(xtmap[touche & 127]));
    } else {
      // sinon pression
      Serial.println(" presser");
      Keyboard.press(KeyboardKeycode(xtmap[touche & 127]));
    }
    rdy=0;
    touche=0;
  }
}
```

La grande majorité du code est identique au croquis d'analyse précédent. La seule différence réside dans l'utilisation des fonctions propres à la bibliothèque de Nicohood et en particulier :

- `Keyboard.press()` qui permet de simuler l'appui sur une touche en précisant le *keycode* correspondant,
- `Keyboard.release()` qui fait de même, mais pour le relâchement d'une touche,
- et `KeyboardKeycode()` qui permet d'obtenir un *keycode* à partir d'un code brut, car vous n'êtes pas censé utiliser les valeurs numériques directement (et donc un des énumérateurs non plus).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Oh comme c'est gentil ! Les personnes qui ont conçu ce périphérique nous ont laissé suffisamment de place pour ajouter une carte Arduino à l'intérieur (et même une Raspberry Pi Zero si besoin).

La logique du croquis est donc très simple, dès qu'une valeur de touche est lue, celle-ci sert d'index pour obtenir le *keycode* correspondant qui est utilisé pour simuler la pression ou le relâchement d'une touche. L'ensemble des messages affichés sur le moniteur série ne sont là qu'à des fins de mise au point du croquis.

Notez à ce propos qu'une perte de temps notable provenait d'une mauvaise compréhension de ma part de l'utilisation de `if(Serial)` (ici `while(Serial)`). Ceci ne permet pas d'attendre que la liaison série soit utilisable, la condition est VRAIE lorsqu'une liaison est effectivement établie. Ce type de choses n'est possible qu'avec les Arduino utilisant un microcontrôleur ATmega32u4 et, si aucun outil n'accède à la carte via la liaison série, cette instruction peut être bloquante et passablement énervante (comme par exemple quand vous mettez au point et que tout marche, mais que le montage refuse de « parler » dès qu'il est branché à un ordinateur de test).

Tout ce qui vient d'être décrit ici peut être reproduit avec n'importe quel clavier de n'importe quel type (ou presque). Du moment que la communication entre le périphérique et la carte peut se faire sous forme d'un lien série synchrone (avec un signal d'horloge), tout ceci est directement transposable (claviers AT, Amiga, Amstrad PC1512, IBM 3104, Mac Classic, Wyse, NeXT, Sun, etc.).

5. POUR FINIR

Parfois le voyage est plus important que la destination et je pense que c'est clairement le cas ici. Faire fonctionner un clavier d'une trentaine d'années avec une Raspberry Pi, un PC actuel ou un Mac est amusant, certes, mais j'ai surtout beaucoup appris, et je l'espère vous aussi. Très souvent certains projets se limitent à la réutilisation de bibliothèques et à un assemblage de pièces où le résultat rapidement obtenu est ce qui apporte satisfaction.

Bien entendu, lorsque cela est trop facile, ce n'est pas vraiment amusant, même si un peu de plaisir est au rendez-vous. Parfois cependant l'aboutissement en lui-même devient l'élément primordial. Passé un certain stade, le résultat n'est plus de faire fonctionner le montage, mais de faire fonctionner son cerveau malgré les erreurs, l'absence de documentation, les suppositions trompeuses, etc.

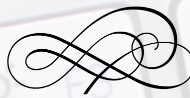
Ce n'est qu'après avoir creusé jusqu'aux fondations des choses, d'en avoir dépilé chaque pièce et tiré la quintessence, pour ensuite tout assembler dans le bon ordre, que viennent le réconfort et le sentiment d'accomplissement. Pour ce projet, ce n'est pas tant le fait de voir s'afficher le texte saisi au clavier qui est important, mais plutôt la signification et la justification des efforts qui se cachent derrière chaque pression sur une touche. Mieux encore, ceci révèle l'importance de faire des erreurs et de ne jamais jeter l'éponge.

Je vous le dis sans détour et en toute sympathie, ce que je vous souhaite c'est de faire vous aussi des erreurs, car c'est tout simplement le meilleur moyen d'apprendre... **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

CONFIGUREZ UN CLAVIER BLUETOOTH POUR VOTRE PI

Denis Bodor



Qu'il s'agisse de la Raspberry Pi 3 ou d'un modèle précédent équipé d'un adaptateur USB Bluetooth, l'un des usages les plus courants pour ce type de connectivité se résume souvent à l'utilisation de périphériques d'entrée comme un clavier ou une souris. Ceci est très facile à configurer via l'interface graphique, mais qu'en est-il lorsqu'on utilise Raspbian Lite et qu'on préfère la ligne de commandes ?

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Lorsqu'on joue avec plusieurs Pi pour différents projets qui tantôt nécessitent une intervention, une reconfiguration ou une mise à jour, il n'est pas toujours agréable de s'amuser à brancher/débrancher un clavier, en particulier lorsque l'installation est bien propre et dans un coffret par exemple. La solution consiste donc à utiliser une liaison sans fil et, si l'on souhaite ne pas devoir jongler avec les *dongles*, cette solution est tout naturellement le Bluetooth.

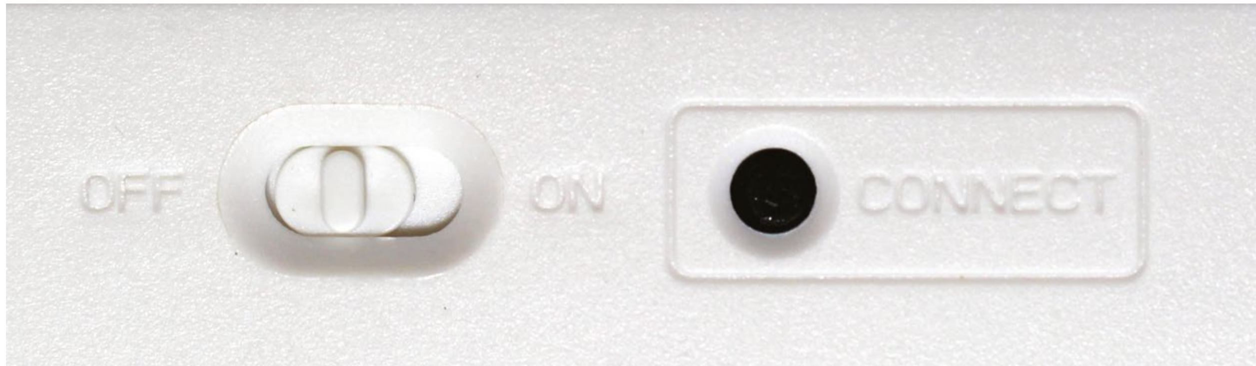
Ceci est pris en charge très facilement via l'interface graphique PIXEL (qui se résume à une adaptation/personnalisation de LXDE) puisqu'il est directement possible de configurer un clavier et une souris depuis une petite icône située en haut de l'écran.

Mais voilà, lorsqu'on décide d'utiliser Raspbian Jessie Lite et non la version avec interface graphique, plus lourde, ce n'est pas pour devoir installer tout ce petit monde sous prétexte de devoir configurer un clavier (chose qui peut effectivement être fait en installant le paquet `raspberrypi-ui-mods` et donc toutes ses dépendances). Mieux vaut se creuser un peu les méninges, se documenter et faire cela en ligne de commandes.

Comme souvent, on retrouve sur le Web, quelques guides, billets de blog et *howto* qui détaillent ce genre de configuration et, comme souvent, il ne s'agit que de copier/coller d'autres guides, billets de blog et *howto* qui n'expliquent rien et assument que certaines manipulations font effectivement partie de la procédure, alors que ce n'est pas le cas. Ainsi, pour le sujet qui nous intéresse ici, on retrouve généralement la modification de deux fichiers système, `/etc/default/bluetooth` et `/etc/default/bluetooth`, alors que ceux-ci n'ont rien à voir avec la configuration d'un périphérique comme un clavier ou une souris. Pire encore, cette modification n'est souvent pas nécessaire et en partie totalement obsolète...

Un clavier Bluetooth de bonne facture sera le compagnon idéal d'une Raspberry Pi 3, puisque celle-ci intègre de base la connectivité adaptée ainsi que le Wifi. Pour les précédents modèles, il vous faudra acquérir un ou des adaptateurs USB Bluetooth.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Les claviers Bluetooth disposent généralement d'un petit bouton à l'arrière permettant de rendre le périphérique découvrable pour procéder à l'appairage. Il faudra ensuite taper un code au clavier pour confirmer l'association.

1. CONNEXION ET APPAIRAGE

Le matériel de démonstration utilisé ici est un sympathique clavier Bluetooth AZERTY de marque Perixx et plus exactement un Periboard-804 KT-1063. Je ne peux guère vous en dire plus puisque je possède ce périphérique depuis bien longtemps et ne sais plus où, quand et comment j'en ai fait l'acquisition. Mais ceci n'est pas important, car tous les périphériques de saisie (clavier, souris, etc.) Bluetooth fonctionnant de la même façon, ce qui va suivre pourra s'appliquer à n'importe quel produit récent ou ancien.

Avec l'implémentation actuelle du support Bluetooth sous GNU/Linux nommée BlueZ (version 5.23 dans Raspbian), tout est directement configurable via une simple commande : `bluetoothctl`. On lancera donc celle-ci accompagnée de l'option `-a` de façon à ce qu'elle intègre directement un agent qui nous permettra de gérer l'authentification lors de l'appairage (on peut également lancer la commande sans l'option puis utiliser `agent on`).

Une fois la commande lancée, on commencera par un petit scan destiné à rechercher des appareils Bluetooth pouvant être découverts. Côté clavier, un petit bouton à l'arrière du matériel permet de rendre le périphérique temporairement visible :

```
[bluetooth]# scan on
Discovery started
[CHG] Controller B8:27:EB:10:1F:A8 Discovering: yes
[NEW] Device 90:7F:61:80:8A:9E Bluetooth Keyboard
```

Le clavier a été détecté par la Pi et nous pouvons nous renseigner sur ce dernier avec `info` suivi de son adresse :

```
[bluetooth]# info 90:7F:61:80:8A:9E
Device 90:7F:61:80:8A:9E
        Name: Bluetooth Keyboard
        Alias: Bluetooth Keyboard
        Class: 0x002540
        Icon: input-keyboard
        Paired: no
        Trusted: no
        Blocked: no
        Connected: no
        LegacyPairing: yes
```

Le périphérique n'est ni appairé, ni connecté, ni de confiance. Nous commençons donc par l'appairage avec :

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
[bluetooth]# pair 90:7F:61:80:8A:9E
Attempting to pair with 90:7F:61:80:8A:9E
[CHG] Device 90:7F:61:80:8A:9E Connected: yes
[agent] PIN code: 358206
```

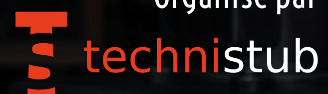
L'association se fera après authentification. L'idée est ici de prouver qu'on a physiquement accès aux deux matériels. Le code qui nous est alors présenté à l'écran par l'agent doit être utilisé sur le clavier. Cette procédure peut être différente d'un matériel à l'autre. Ici, il suffit de taper les six chiffres sur le clavier (qui est alors en QWERTY et ne demande donc pas d'utiliser la touche Maj pour entrer les chiffres) et de finir par la touche *Entrée*.

Ceci fait, **bluetoothctl** nous confirme le succès de l'opération :

```
[CHG] Device 90:7F:61:80:8A:9E Paired: yes
Pairing successful
```

Pour que la liaison puisse s'établir automatiquement la prochaine fois que les périphériques (Pi et clavier) seront réunis, nous devons faire confiance au clavier. Nous utilisons donc la commande **trust** suivie de l'adresse du périphérique pour entériner la liaison :

```
[bluetooth]# trust 90:7F:61:80:8A:9E
[CHG] Device 90:7F:61:80:8A:9E Trusted: yes
Changing 90:7F:61:80:8A:9E trust succeeded
```

MAKER FIGHT #3
Combats de robots, ateliers et bricolage 2.0
Samedi 8 Avril 2017
la Fonderie - Mulhouse
ENTREE GRATUITE
organisé par


www.technistub.org
infos et inscription sur
www.makerfight.fr

Ce document est la propriété exclusive de Alex Arnaud (@gmail.com)

Enfin, on jette un dernier coup d'œil à la configuration :

```
[bluetooth]# info 90:7F:61:80:8A:9E
Device 90:7F:61:80:8A:9E
        Name: Bluetooth Keyboard
        Alias: Bluetooth Keyboard
        Class: 0x002540
        Icon: input-keyboard
        Paired: yes
        Trusted: yes
        Blocked: no
        Connected: no
        LegacyPairing: yes
```

Les lignes **Paired** et **Trusted** sont à **yes**, le périphérique est donc bien appairé et la liaison est de confiance. Dès cet instant, le clavier doit être utilisable exactement comme un modèle connecté en USB. Il vous sera peut-être nécessaire cependant de configurer son agencement de clavier en lançant **sudo raspi-config** et en faisant un tour dans ***Internationalisation Options*** et ***Change Keyboard Layout*** pour choisir un clavier français AZERTY.

Notez la ligne **Connected** dans la sortie. Ceci pourra vous permettre de vous assurer que le clavier Bluetooth est bel et bien connecté, en cas de problème d'utilisation. N'oubliez pas, non plus, que les périphériques Bluetooth passent généralement rapidement en veille et qu'une frappe sur un clavier qui est endormi mettra parfois quelques secondes pour le réveiller et rétablir la connexion.

En termes de qualité de fabrication et de solidité, ce qu'on trouve sur le Web est assez variable. Ici au premier plan un clavier très peu cher (~15€), peu solide, avec une frappe très moyenne et une alimentation via une paire de piles AAA. En arrière-plan, un modèle de meilleure qualité, rechargeable via une connexion micro-USB.

2. CONFIGURATION DU CLAVIER

En dehors de l'agencement des touches, il n'y a strictement rien à configurer pour faire fonctionner votre clavier. D'où sortent donc ces informations concernant la modification de certains fichiers ?

Le premier, **/etc/default/bluetooth**, permet de régler un problème rencontré avec certains adaptateurs Bluetooth. Le matériel moderne s'utilise généralement avec un protocole appelé HCI pour *Host Controller Interface*. Celui-ci décrit la façon dont l'hôte (votre Pi) et le contrôleur (l'adaptateur Bluetooth) communiquent. Cependant,

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

certains modèles, lorsque vous les connectez à la machine, apparaissent comme des périphériques HID (*Human Interface Device*), une classe de périphériques USB correspondant généralement aux périphériques de saisie.

Pour contrôler l'adaptateur, nous devons lui parler HCI et non HID, ces adaptateurs doivent donc changer de mode de communication pour être utilisés. C'est là qu'intervient un outil appelé `hid2hci` (placé dans `/lib/udev`). L'idée est, lors de la détection de l'adaptateur USB, de demander au système d'automatiquement lancer `hid2hci` pour le faire changer de mode (via *udev*). Ceci se configure en changeant une ligne dans le fichier `/etc/default/bluetooth` :

```
HID2HCI_ENABLED=1
```

Dans le cas d'une Raspberry Pi 3 et son adaptateur Bluetooth intégré, le changement de mode n'est pas nécessaire puisqu'il communique, de base, en HCI. Je pense que la confusion concernant l'utilisation de ce fichier vient du fait qu'il est question de HID, terme qui rappelle les périphériques USB HID et donc les claviers/souris USB. Pourtant ceci n'a rien à voir avec le périphérique et la prise en charge de claviers Bluetooth, cela ne concerne que l'adaptateur et sa connexion.

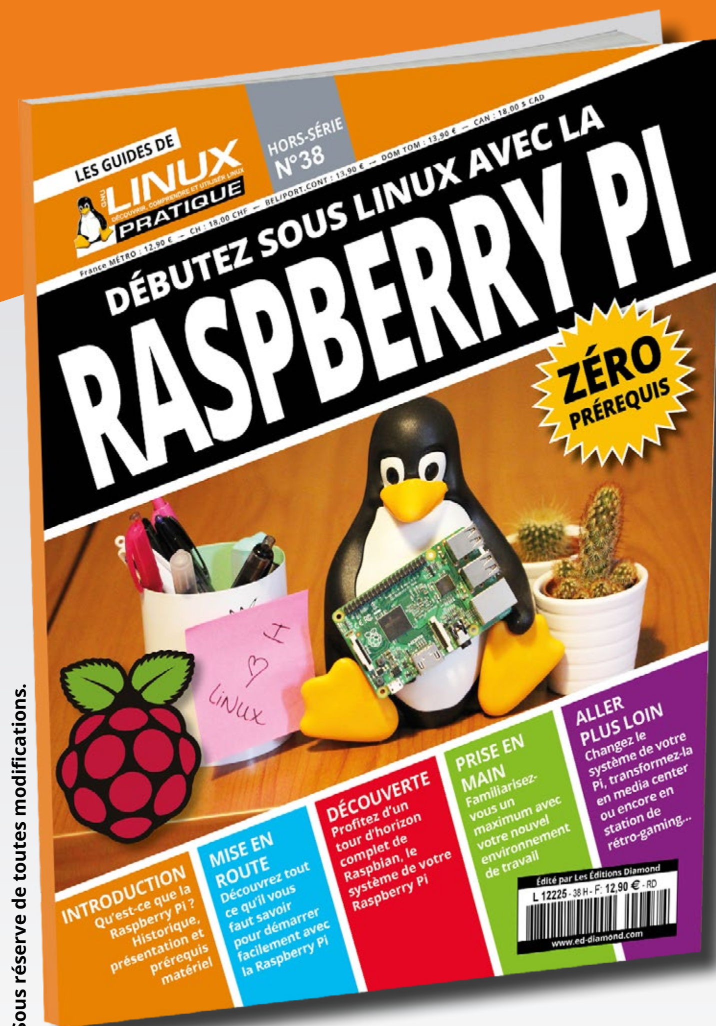
L'autre fichier généralement cité est `/etc/bluetooth/hcid.conf`. Là, c'est encore pire, car ce fichier n'est tout simplement pas utilisé. Il s'agissait, avec BlueZ 3.x, du fichier de configuration de `hcid`, le démon de gestion HCI. Celui-ci, avec l'arrivée de la version 4.0 de BlueZ (en août 2008 !), a laissé place à `bluetoothd`, encore utilisé aujourd'hui. `hcid.conf` n'a aucune utilité puisque `hcid` n'est plus là depuis plus de 8 ans...

Moralité, mieux vaut regarder les dates des pages (biller, wiki, etc.) avant de suivre une procédure et s'assurer que ce qui est décrit correspond bien à la documentation **officielle**. Car, bien entendu, en ajoutant un `hcid.conf` et en modifiant son `/etc/default/bluetooth`, cela fonctionne, mais en ajoutant un `pouet.conf` dans `/etc/bluetooth`, cela fonctionnera tout aussi bien... **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

DISPONIBLE DÈS LE 24 MARS !

LINUX PRATIQUE HORS-SÉRIE n°38



Sous réserve de toutes modifications.

DÉBUTEZ SOUS LINUX AVEC LA RASPBERRY PI

NE LE MANQUEZ PAS
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

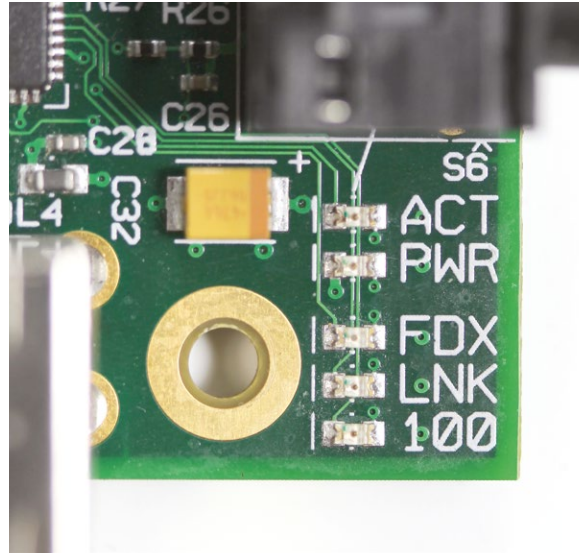
http://www.ed-diamond.com

CHANGEZ LA CONFIGURATION DES LEDS DE VOTRE RASPBERRY PI

Denis Bodor

... ou de n'importe quel autre ordinateur monocarte du même type. Les leds équipant généralement les cartes comme la Raspberry Pi, la BeagleBone Black, l'OrangePi, ou encore la NanoPi sont généralement configurées par défaut pour indiquer un certain type d'activité. Ceci peut cependant être changé relativement facilement pour adapter le comportement de ces leds à vos besoins.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



La toute première Raspberry Pi disposait de 5 leds dont trois contrôlées par l'interface réseau, une directement branchée à l'alimentation et une, ACT, pilotable comme détaillée dans cet article.

Avant que le système dont nous allons parler soit ajouté dans Linux (et je parle ici du noyau), disposer d'une led témoin d'une activité quelconque n'était pas une mince affaire. Il fallait soit écrire un module noyau, soit un programme utilisateur se chargeant de changer l'état de la, ou des leds. Ce n'était pas une mince affaire et cela demandait pas mal de connaissances et d'expérience. Puis fut introduit, avec la version 3.4 du noyau, le support pour la classe de périphériques LED permettant une prise en charge de ce genre de fonctionnalités directement par le noyau, mais de façon configurable par l'utilisateur.

Bon nombre de plateformes utilisent à présent ce mécanisme pour toutes sortes de choses, les Raspberry Pi bien sûr pour les leds verte et rouge placées sur la carte elle-même (verte uniquement pour la Pi 3), bien d'autres nano-ordinateurs monocartes et même les ordinateurs de type PC, portables ou non. De plus, certains matériels exposent également ce type de périphériques, comme par exemple les clés USB Wifi, permettant à l'utilisateur de configurer librement l'utilisation et la signification de ces leds.

1. JOUONS AVEC LES LEDS

L'objet même de la classe de périphérique LED est de vous permettre de manipuler leur état depuis la ligne de commandes, à l'aide de simples outils mis à disposition par le système. Avant toutes choses, précisons ici que ce qui va suivre s'applique, dans le cas de nos chères Pi, aux modèles 2, B, B+ et A+ en ce qui concerne les deux leds intégrées à la carte : la rouge libellée PWR pour *power* et la verte notée ACT comme *activity*. La Raspberry Pi Zero n'a pas de led rouge « PWR » et la Raspberry Pi 3 en dispose, mais elle est physiquement connectée à l'alimentation (donc non contrôlable par le système). La Raspberry Pi première du nom (celle avec un connecteur RCA jaune) dispose de 5 leds, mais seule celle marquée ACT est, là aussi, contrôlable de la façon que je vais m'empresser de détailler.

Pour accéder à l'état de cette ou ces leds, rendez-vous simplement dans le répertoire `/sys/class/leds`. Là vous y trouverez un sous-répertoire pour chaque led pilotable de cette façon : `led0` pour la verte et `led1` pour la rouge (si utilisable). Si vous avez connecté une clé USB Wifi, par exemple, il est également possible que d'autres répertoires apparaissent à cet endroit : `rt73usb-phy0::assoc`, `rt73usb-phy0::quality`, et `rt73usb-phy0::radio`, par exemple, dans le cas de ma Linksys WUSB54GC (toutes les leds ne sont pas forcément visibles, mais le pilote fourni néanmoins un accès dans `/sys/`).

Tous ces répertoires contiennent la même structure et les mêmes fichiers :

```
$ ls -F
brightness
device@
max_brightness
power/
subsystem@
trigger
uevent
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Notez qu'il ne s'agit pas de réels fichiers et répertoires présents sur la carte SD ou microSD, mais d'une façon pour le système de vous donner un accès à ces informations. Il en va de même pour `/sys` que pour `/proc`, `/dev` ou encore le contenu de `/tmp` ou `/run`.

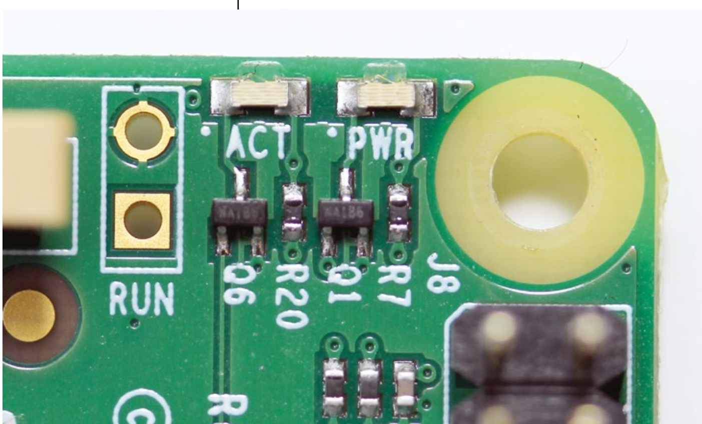
Le fichier important, déterminant la façon dont se comporte la led concernée par le répertoire choisi, est `trigger`. Celui-ci permet de connaître et de modifier ce qui déclenche (*trigger* en anglais) un changement d'état de la led. Nous pouvons consulter le contenu du fichier avec :

```
$ cat /sys/class/leds/led0/trigger
none kbd-scrollock kbd-numlock kbd-capslock
kbd-kanalock kbd-shiftlock kbd-altgrlock
kbd-ctrllock kbd-altlock kbd-shiftllock
kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock
[mmc0] mmc1 timer oneshot heartbeat backlight
gpio cpu0 cpu1 cpu2 cpu3 default-on input
rfkill0 rfkill1
```

Ce qui se présente alors à l'écran est l'ensemble des déclencheurs utilisables avec, entre crochets, celui qui est actuellement configuré : `mmc0`, une activité sur première carte mémoire ou, en d'autres termes, la microSD de cette Raspberry Pi 3. Le contenu du fichier est dépendant des pilotes actuellement chargés par le noyau et de la carte utilisée, mais un certain nombre de déclencheurs sont généralement présents par défaut :

- `mmc0` : activité sur le support de stockage (SD, microSD, eMMC, etc.). Très pratique pour s'assurer que la Pi « travaille » ;
- `heartbeat` : produit un battement cardiaque dont le rythme est proportionnel à la charge système. Plus la Pi fait de choses, plus le rythme est important ;
- `timer` : permet une configuration personnalisée en choisissant la durée active et inactive de la led en millisecondes ;
- `gpio` : permet de changer automatiquement l'état de la led en fonction de celui d'une broche de la Pi (GPIO), comme témoin d'activité en somme ;
- `cpu*` : configure la led de façon à montrer l'activité d'un processeur (ou d'un cœur). Ceci permet avec suffisamment de leds disponibles d'avoir un témoin d'activité par CPU/cœur. Le nombre de déclencheurs (ici 4) est fonction du nombre de cœurs disponibles ;
- `none` : aucun déclencheur ;
- `default-on` : toujours active.

Les déclencheurs dont le nom débute par `kbd` font référence à l'état des éventuelles leds d'un clavier. Il est ainsi possible de configurer la led concernée par le répertoire où se



Voici les deux leds intégrées à une Pi modèle B+ et pilotables, l'une comme l'autre directement depuis le système via la manipulation de simples fichiers dans /sys. Remarquez les transistors Q1 et Q6 placés juste dessus permettant le pilotage à l'aide de simples GPIO.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

trouve **trigger** pour suivre l'état de ces dernières. Enfin, tout comme les leds physiques de certains périphériques sont exposés via **/sys/class/leds**, on retrouve tantôt dans **triggers** les déclencheurs supplémentaires correspondants, proposés par le pilote du périphérique. Dans le cas de ma clé wifi Linksys, par exemple, les déclencheurs **phy0rx**, **phy0tx**, **phy0assoc**, et **phy0radio** sont ainsi également disponibles (émission, réception, association et radio wifi active).

Pour changer le déclencheur, rien de plus simple, il suffit de placer dans le fichier celui qu'on souhaite utiliser. Il faut pour cela disposer des permissions adéquates, le fichier appartenant au super-utilisateur **root** :

```
$ cd /sys/class/leds/led0
$ sudo -s
# echo timer > trigger
```

Nous venons ici de configurer le déclencheur *timer* et ce faisant, deux nouveaux fichiers sont apparus dans le répertoire, **delay_off** et **delay_on**. Ceux-ci permettent de choisir le délai des deux états de la led :

```
# cat delay_on
500
# cat delay_off
500
```

Ici, la led sera allumée 500ms, éteinte 500ms, et ainsi de suite. Nous pouvons, avec **echo** et une redirection, changer ces valeurs à loisir :

```
# echo 50 > delay_on
# echo 1000 > delay_off
```

Nous aurons alors une brève pulsation lumineuse de 50 millisecondes toutes les secondes. Bien entendu, en fonction du déclencheur configuré et placé dans **trigger** ceci changera et d'autres fichiers feront (ou pas) leur apparition dans le répertoire. Notez également qu'en cas de changement de déclencheur, la configuration précédente ne sera pas conservée. Si vous basculez de *timer* à *mmc0* puis revenez à *timer*, les valeurs dans **delay_off** et **delay_on** reviendront à 500ms par défaut. Bien entendu, il en ira de même en cas de redémarrage du système...

2. CONFIGURER LE COMPORTEMENT PAR DÉFAUT SUR UNE PI

Par défaut, les Raspberry Pi sont configurées avec une led verte libellée ACT (**led0**) notifiant d'une activité sur le support de stockage (**mmc0**). On retrouve un peu partout sur le Web différentes méthodes permettant de changer cet état de fait : commandes à placer dans **rc.local**, passage de paramètres au noyau ou encore bidouillage ignoble dans des fichiers systèmes...

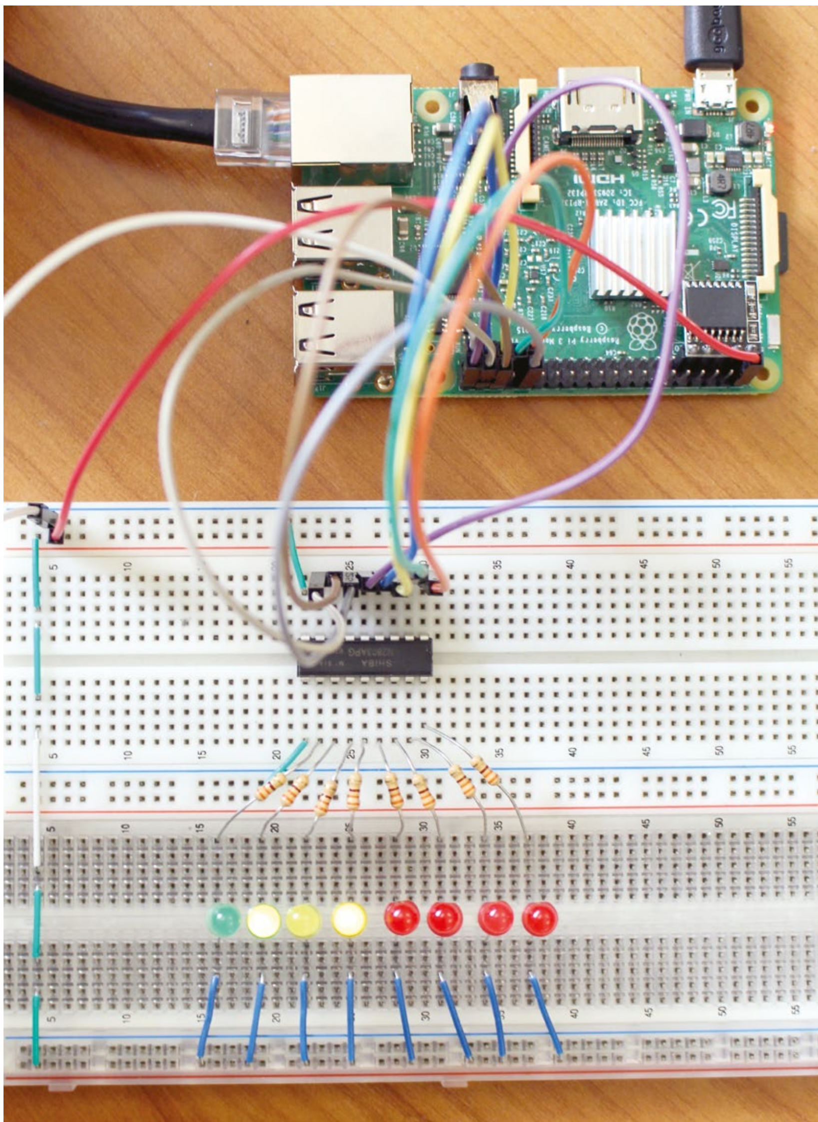
Depuis quelques versions de Raspbian pourtant, accompagnée d'un noyau supportant le *device tree* (voir article sur la configuration de deux minis écrans SPI dans *Hackable n°9*), il n'est pas nécessaire de faire ces choses, aussi honteuses que dégoûtantes. Dans ce genre de situation, il est important de ne pas se poser la question « *comment je peux avoir ce que je veux ?* », mais plutôt « *comment ceci est actuellement configuré ?* ».

On peut supposer, en effet, que les développeurs ont poursuivi exactement le même objectif que vous en configurant le comportement par défaut et que, de ce fait, ils ont sans doute appliqué la solution la plus optimale possible. En d'autres termes, la première approche à envisager doit être teintée d'humilité...

Or justement, cette fameuse led verte n'est pas du tout configurée par un script de démarrage ou un programme quelconque, elle fait partie des périphériques non détectables définis dans ce qu'on appelle le *device tree*. Contrairement aux périphériques USB par exemple, dont la connexion déclenche la mise en œuvre d'un processus de détection et de sélection de pilotes, certains matériels et composants d'un ordinateur qu'il s'agisse d'une Pi ou d'un PC, ne peuvent être détectés automatiquement.

Jusqu'à l'arrivée du *device tree*, cette configuration spécifique était embarquée

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Ajouter des leds configurées de la même façon que celles intégrées à la carte se résume à une connexion classique de ce genre de composants sur les broches d'une Pi. La partie délicate concerne surtout la configuration logicielle, mais rien n'est insurmontable dans ce domaine...

directement dans le plus important des programmes : le noyau. Puis quelques développeurs ont eu l'idée de rationaliser cela et de faciliter la vie de tous leurs confrères (et des utilisateurs par la même occasion) en créant un format unique permettant de décrire précisément la composition d'une machine : le *device tree*.

Celui-ci est, dans un format binaire, chargé directement au tout début du démarrage et le noyau l'utilise pour savoir précisément à quel matériel il a affaire. Sur une Raspberry Pi, ceci prend la forme d'un fichier placé sur la carte SD/microSD, sur la partition FAT, accessible via le répertoire **/boot** une fois le système démarré :

```
$ ls /boot/*.dtb
/boot/bcm2708-rpi-b.dtb
/boot/bcm2708-rpi-b-plus.dtb
/boot/bcm2708-rpi-cm.dtb
/boot/bcm2709-rpi-2-b.dtb
/boot/bcm2710-rpi-3-b.dtb
/boot/bcm2710-rpi-cm3.dtb
```

Il y a un fichier par modèle de Pi. Les modèles B et A ne se différenciant que par l'absence de quelques éléments, connectés en USB et donc détectables, il n'y a de distinctions qu'entre les modèles B, B+, 2B, 3B, *compute module* et *compute module 3*.

Pour être utilisable, le fichier du *device tree*, qui se présente initialement sous la forme de texte structuré, doit être transformé (compilé) en un fichier binaire utilisable par le noyau. On utilise pour cela un outil nommé **dtc** (paquet **device-tree-compiler**). Comme il peut être relativement pénible de devoir faire cette transformation en cas de changement de configuration comme celle qui nous intéresse ici, un autre mécanisme a été mis en place. Il est possible de compléter la configuration initiale à l'aide de « modules » additionnels : les *overlays*.

Ceux-ci, placés dans **/boot/overlays**, permettent d'ajouter des configurations de périphériques non détectables sans avoir à éditer et recompiler un nouveau fichier binaire du *device tree*. Si vous voulez ajouter

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

une horloge temps réel type DS3231 à votre configuration, par exemple, vous n'avez qu'à utiliser **i2c-rtc.dtbo**.

Pour ce faire, il vous suffit d'ajouter deux petites lignes dans **/boot/config.txt** :

```
# activer le bus I2C
dtparam=i2c_arm=on
# prendre en charge la RTC
dtoverlay=i2c-rtc,ds3231
```

Il en va exactement de même pour la configuration du comportement de la ou des leds soudées à votre Pi. Là, cependant, il n'est pas nécessaire d'utiliser le mécanisme d'*overlays*, car il ne s'agit pas d'un ajout de matériel, mais de la configuration d'un matériel existant. L'option **dtparam** permet ainsi d'activer et paramétrer des éléments déjà pris en charge. Ainsi, pour automatiquement configurer le déclencheur associé à la led ACT, nous pouvons simplement ajouter au fichier **config.txt** :

```
dtparam=act_led_trigger=heartbeat
```

Dans le *device tree*, cette led est nommée **act_led** et un paramètre **act_led_trigger** détermine le déclencheur à utiliser par défaut. Remplacez simplement **heartbeat** par l'un des déclencheurs dont nous avons pris connaissance précédemment et le tour est joué. Pour les Raspberry Pi supportant également la configuration de la led rouge (**led1** dans **/sysfs**), nous pouvons procéder de même en utilisant **pwr_led_trigger** en lieu et place de **act_led_trigger**.

Redémarrez ensuite votre Pi et vous pourrez constater que dès les premières étapes du démarrage, votre déclencheur sera utilisé. Pas besoin de scripts, pas besoin de bidouille... le seul fichier modifié est celui qui est précisément là pour ajuster la configuration matérielle à vos besoins. C'est propre et sans bavure.

Note : remarquez que les choses ont un peu évolué depuis mon article dans *Hackable n°9*. À l'époque, les fichiers *overlays* étaient nommés **NOM-overlay.dtb** et étaient spécifiés avec **NOM** dans **config.txt**. À présent, les fichiers ont simplement une extension **.dtbo** et leur nom, sans l'extension, est utilisé dans la configuration.

Notez également que la led rouge, sur les Pi où elle est configurable, n'est pas branchée de la même façon que sa consœur verte. Cette led est active avec un signal à l'état bas. De ce fait, si vous souhaitez rendre votre Pi plus discrète en désactivant les deux leds, vous devrez ajouter ceci dans **config.txt** :

```
# Eteindre la led verte
dtparam=act_led_trigger=none

# Eteindre la led rouge
dtparam=pwr_led_trigger=none
dtparam=pwr_led_activelow=off
```

Cette dernière ligne précise que la logique est inversée et qu'avec aucun déclencheur configuré (**none**), la led est effectivement éteinte. Cette ligne devra également être précisée si vous voulez que votre déclencheur fournisse visuellement l'effet attendu et non son opposé.

3. CONFIGURER LE COMPORTEMENT SUR D'AUTRES CARTES

Les versions récentes du système Raspbian reposent sur le mécanisme de *device tree*, mais il n'en a pas toujours été ainsi. D'autre part, toutes les plateformes ne l'utilisent pas forcément. Le système Debian installable sur les NanoPi, par exemple, n'en fait pas (encore) usage. Comment alors modifier le comportement par défaut des leds pourtant effectivement accessibles via **/sys/class/leds** ?

Là encore, inutile de chercher à bidouiller dans les coins et se montrer humble est l'attitude à adopter. Mieux vaut se dire « *les développeurs ont sûrement pensé à quelque chose* » plutôt que « *ils ont oublié de prévoir quelque chose* ». Ici, il ne

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

s'agit pas des développeurs du noyau ou de ceux travaillant sur le système de la Raspberry Pi, mais des développeurs Debian.

Il est, en effet, possible, même sur une Pi si vous ne voulez pas toucher à `config.txt`, d'installer un simple paquet pour pouvoir à loisir déterminer une configuration par défaut appliquée au démarrage dans n'importe quel fichier de `/sys` : `sysfsutils`.

Après installation de ce paquet avec un petit `sudo apt-get install sysfsutils`, il vous suffira alors de créer un nouveau fichier de configuration. Comme on souhaite changer le contenu du fichier `/sys/class/leds/led0/trigger`, par exemple, nous créons simplement un fichier `/etc/sysfs.d/led0.conf` (le nom importe peu du moment qu'il finisse par `.conf`), contenant :

```
class/leds/led0/trigger = heartbeat
```

On précise simplement le fichier à modifier et la valeur à y placer. Notez que `/sys` n'est pas précisé puisque `sysfsutils` ne concerne que le contenu de `/sys`. Il suffit ensuite de redémarrer le service installé par `sysfsutils` pour appliquer les changements :

```
$ sudo service sysfsutils restart
```

En cas de problème, les outils de *systemd* vous permettront de savoir d'où vient une éventuelle erreur (comme celle consistant à utiliser bêtement des guillemets autour du nom du déclencheur) :

```
$ sudo systemctl status sysfsutils.service
$ sudo journalctl -xn -n 30
```

4. DEUX LEDS NE VOUS SUFFISENT PAS ? IL SUFFIT D'EN AJOUTER

Disons-le de suite, là les choses vont devenir un peu plus délicates et complexes. Mais à vaincre sans péril, on triomphe sans gloire, paraît-il.

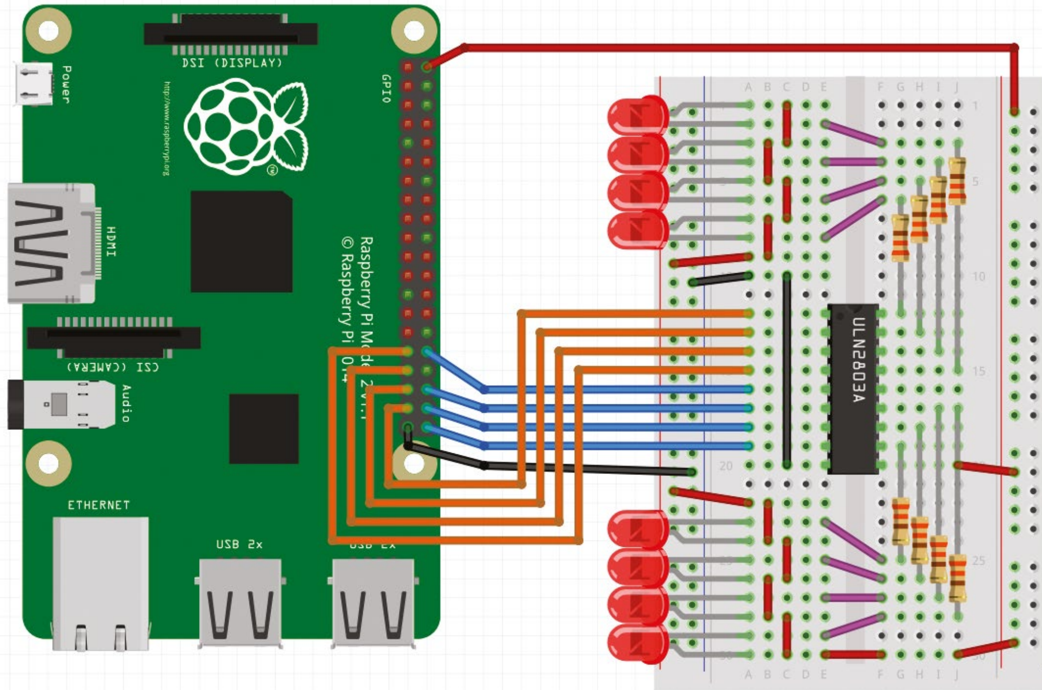
Sur ma Pi 3, je dispose d'un processeur possédant 4 cœurs et, en toute logique, je dispose donc des déclencheurs `cpu0` à `cpu3`. L'idée même de littéralement voir les cœurs s'exciter au rythme des commandes et programmes exécutés me semble tout bonnement merveilleuse ! Seulement voilà, sur une Raspberry Pi 3, la seule led que je peux utiliser ainsi est la led verte ACT, et le pilotage de la led rouge ne changerait pas grand-chose à l'affaire. Il nous faut plus de leds (oui, il faut **toujours** plus de leds)

C'est frustrant ! Inacceptable ! Être ainsi privé d'une telle stimulation visuelle est à la limite du supportable. Il faut agir !

Je l'ai précisé précédemment, le support de ce périphérique est pris en charge par le mécanisme de *device tree*. De ce fait, quelque part dans le fichier textuel ou dans l'un des fichiers *overlays* est forcément défini qu'une broche est connectée à cette led verte et qu'elle doit être utilisée de cette façon précise par le noyau.

Pour accéder aux structures du *device tree* sous une forme lisible, il est nécessaire de s'en référer aux sources du noyau (ou d'utiliser `dtc` pour décompiler un binaire, mais on perd les

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Piloter des leds avec une Raspberry Pi n'est pas très compliqué, mais il faut bien prendre en considération que les GPIO ne peuvent pas en supporter plus d'un certain nombre faute de courant suffisant. L'une des solutions possibles consiste à utiliser un ULN2803A, un circuit intégré comprenant huit transistors Darlington, chacun capable d'accepter jusqu'à 500mA.

commentaires). Pas de panique, nous n'avons pas à y faire de modification ni même à produire un nouveau paquet à cet effet (si vraiment vous y tenez, jetez un coup d'œil dans *Hackable n°10*), tout ce que nous voulons c'est savoir comment ceci est mis en place.

Et effectivement dans le fichier `linux/arch/arm/boot/dts/bcm2710-rpi-3-b.dts`, la version texte du fichier `bcm2710-rpi-3-b.dtb` présent dans votre `/boot`, nous pouvons lire ceci :

```
&leds {
        act_led: act {
                label = "led0";
                linux,default-trigger = "mmc0";
                gpios = <&virtgpio 0 0>;
        };
};
[...]
                act_led_gpio = <&act_led>,"gpios:4";
                act_led_activelow = <&act_led>,"gpios:8";
                act_led_trigger = <&act_led>,"linux,default-trigger";
```

Nous avons donc la confirmation de ce que nous supposions, l'association GPIO/led est bien dans le *device tree* et non dans le noyau lui-même. Ceci implique également qu'il nous est donc possible d'écrire de toutes pièces un *overlay*, le compiler et l'ajouter dans `/boot/overlays` pour pouvoir avoir bien plus de leds accessibles via `/sysfs`.

Je vous ferai grâce ici de la démarche complète de recherche et vous fournirai simplement le contenu du fichier à créer, que nous appellerons `mesleds.dts` :

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
/dts-v1/;
/plugin/;

/ {
   compatible = "brcm,bcm2708", "brcm,bcm2709", "brcm,bcm2710";

   fragment@0 {
       target = <&leds>;
       __overlay__ {
           ma_led0: maled0 {
               label = "maled0";
               gpios = <&gpio 26 0>;
               linux,default-trigger = "cpu0";
           };
           ma_led1: maled1 {
               label = "maled1";
               gpios = <&gpio 19 0>;
               linux,default-trigger = "cpu1";
           };
           ma_led2: maled2 {
               label = "maled2";
               gpios = <&gpio 13 0>;
               linux,default-trigger = "cpu2";
           };
           ma_led3: maled3 {
               label = "maled3";
               gpios = <&gpio 6 0>;
               linux,default-trigger = "cpu3";
           };
       };
   };
};
```

Nous définissons ici 4 nouveaux périphériques, **maled0** à **maled3** respectivement connectés aux GPIO 26, 19, 13, et 6 de la Pi (broches 37, 35, 33 et 31). Nous définissons également le déclencheur par défaut pour chaque led de façon à les voir s'activer dès que le cœur correspondant du processeur est actif.

Il ne nous reste alors plus qu'à créer un fichier binaire à partir de cette description textuelle avec :

```
$ sudo dtc -@ -I dts -O dtb \
-o /boot/overlays/myled.dtbo myled.dts
```

Nous lançons **dtc** en précisant directement le nom du fichier de sortie, **myled.dtbo**, à l'emplacement où il pourra être utilisé (**/boot/overlays**). Enfin, nous ajoutons une ligne dans notre **config.txt** afin de prendre en charge cet *overlay* :

```
dtoverlay=myled
```

Au prochain redémarrage, cette modification sera utilisée et il nous suffit d'éteindre la Pi proprement pour y connecter nos leds. En fonction du nombre de leds utilisées et de leurs caractéristiques, il ne vous sera pas possible de les piloter directement, car la Pi ne sera pas capable de fournir le courant nécessaire. Mieux vaudra passer par un petit montage reposant sur un

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

ULN2803A par exemple, des MOSFET ou encore des transistors. Ma solution repose sur un ULN2803A puisque ce composant est parfaitement adapté à un tel usage.

Une fois les leds connectées, il vous suffira de démarrer votre Pi pour admirer le magnifique clignotement rythmé par l'activité du processeur. Un petit coup d'œil dans `/sys` nous confirmera qu'effectivement, nous venons d'ajouter ces périphériques que nous pourrons manipuler exactement comme la led ACT depuis la ligne de commandes :

```
$ ls /sys/class/leds/
led0  maled0  maled1  maled2  maled3

$ ls /sys/class/leds/maled0/
brightness  device  max_brightness
power  subsystem  trigger  uevent

$ cat /sys/class/leds/maled0/trigger
none kbd-scrollock kbd-numlock kbd-capslock
kbd-kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock
kbd-altlock kbd-shiftllock kbd-shiftrlock
kbd-ctrlllock kbd-ctrlrlock mmc0 mmc1 timer oneshot
heartbeat backlight gpio [cpu0] cpu1 cpu2 cpu3
default-on input rfkill0 rfkill1
```

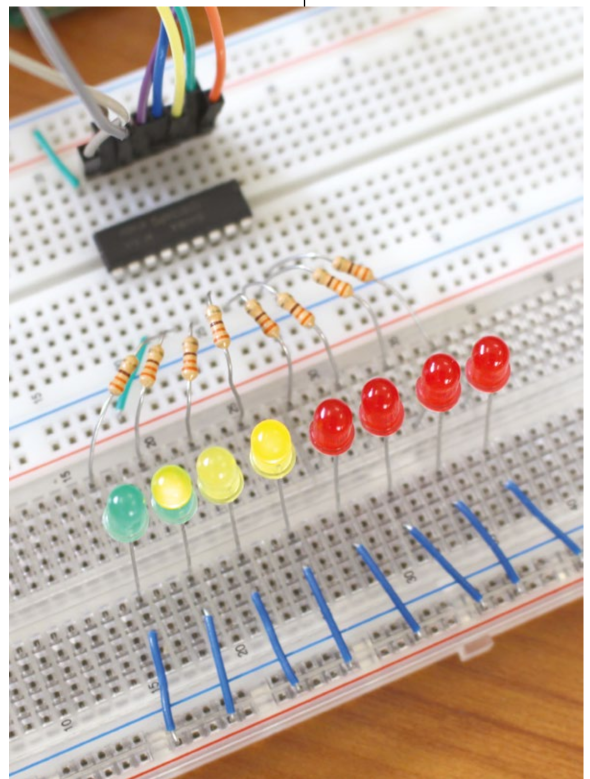
Là, vous pouvez vous amuser à lancer des commandes et voir physiquement chaque cœur s'exténuer docilement pour accomplir vos souhaits... Un pur bonheur !

5. POUR FINIR

La morale de cet article est relativement simple : faites confiance aux développeurs et ne partez jamais du principe que quelque chose a été mal pensé ou oublié. Bien sûr, il est possible qu'une telle chose puisse arriver, là n'est pas la question. Si ce n'était pas le cas, il n'y aurait pas de correction de bogues et encore moins d'évolution des versions. Les erreurs existent, mais il est généralement plus intelligent de tout d'abord remettre ses propres compétences en cause, reposer sur celles des développeurs pour ensuite, et seulement ensuite, envisager une erreur ou un oubli.

Je finirai en précisant que le dernier point de l'article est perfectible. Notre *overlay* n'est pas configurable comme d'autres, mais ceci peut être ajouté pour plus de souplesse. Je vous recommande donc de fouiller dans les fichiers `.dts` présents dans les sources du noyau disponibles sur **https://github.com/raspberrypi/linux**, et de vous inspirer de ce qui s'y trouve pour tenter d'accomplir cela par vous-même... **DB**

Pour piloter 8 leds comme celle-ci depuis les broches d'une Raspberry Pi, il serait déraisonnable d'envisager une connexion directe. Mieux vaut utiliser un circuit intégré spécialisé comme l'ULN2803A capable de prendre en charge le courant nécessaire, si d'aventure toutes les leds étaient allumées en même temps.



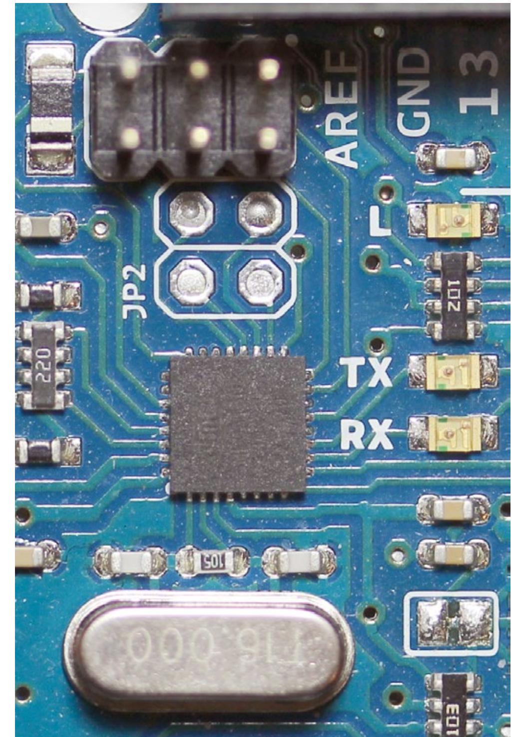
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

UTILISEZ VOTRE ARDUINO UNO COMME PÉRIPHÉRIQUE USB

Denis Bodor

Les cartes Arduino Leonardo et Micro utilisent un microcontrôleur bien différent des modèles UNO ou Mega, un ATmega32U4. De ce fait non seulement ces cartes n'ont plus besoin d'une puce faisant la liaison série/USB, mais elles peuvent également apparaître comme des périphériques USB sur-mesure. Les Arduino UNO semblent donc un peu en reste de ce point de vue, mais c'était sans compter un certain NicoHood et son HoodLoader2 ! Les Uno aussi peuvent devenir des périphériques USB. Suivez le guide !

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Le microcontrôleur Atmel AVR ATmega16U2 sert initialement de relais entre la connexion USB et le microcontrôleur animant la carte Arduino UNO. Il peut cependant être utilisé pour vos croquis. Ici de bas en haut, son oscillateur à quartz, le microcontrôleur lui-même, à sa droite les leds qu'il contrôle, 4 ports utilisables à souder et le connecteur ICSP pour le programmer.

Précisons de suite que je vais parler ici des modèles récents d'Arduino UNO et plus exactement des R3 (alias REV3). Les premières UNO utilisant une puce FTDI en guise de pont de communication entre le microcontrôleur AVR ATmega328P et le port USB ne sont pas compatibles avec ce qui va suivre, pas plus que les clones utilisant généralement des puces USB/série CH340/CH341.

La raison pour laquelle seules les cartes UNO (et Mega2560) officielles et récentes peuvent être utilisées est simple : il y a deux microcontrôleurs sur ces cartes, un ATmega238P (ou un ATmega2560) et un ATmega16U2. Nous nous concentrerons ici sur la carte UNO, mais ce qui va suivre est également applicable (mais non testé par mes soins) sur une Mega2560.

Le microcontrôleur ATmega16U2 est une déclinaison plus modeste de l'ATmega32U4 utilisé sur les cartes Leonardo et Micro. Celui-ci, sur une UNO, est programmé avec un croquis opérant exactement comme une puce FTDI ou CH340 : faire apparaître la carte comme un port USB/série et transmettre tout ce qu'il reçoit à l'ATmega238P et, inversement, envoyer en USB tout ce que veut bien raconter l'ATmega238P. Ce qui est cependant très intéressant dans cette histoire c'est qu'un ATmega16U2 n'est pas un petit composant insignifiant : 16 Ko de mémoire flash pour les programmes, 512 octets de mémoire vive, 512 octets d'EEPROM intégrée, fonctionnement à 16 Mhz... c'est presque la moitié des ressources offertes par l'ATmega238P qui forme la base d'une carte UNO !

Autre précision que je tiens également à apporter, le fait de se servir de ce composant n'a rien de nouveau. Sans doute avez-vous déjà remarqué le connecteur à 6 broches se trouvant à proximité du port USB et du bouton *reset*, en tous points identique à celui marqué « ICSP » à l'autre bout de la carte. C'est le connecteur permettant de programmer l'ATmega16U2 à l'aide d'un programmeur AVR-ISP, exactement comme il est possible de le faire pour l'ATmega238P via son connecteur dédié.

La nouveauté apportée par NicoHood et son HoodLoader2 réside dans le fait de ne plus avoir besoin d'utiliser un matériel spécialisé comme un programmeur AVR-ISP (ou une carte Arduino utilisée comme telle), mais de pouvoir programmer simplement les deux microcontrôleurs depuis l'environnement Arduino, en choisissant simplement dans un menu à quel microcontrôleur on souhaite s'adresser.

Enfin, pour en finir avec cette introduction, il est important de vous préciser que le fait de tritouiller l'ATmega16U2 et son bootloader (voir ci-après) n'est pas une opération sans risque. En cas de maladresse, de déconnexion intempestive lors de l'installation ou de problème, votre carte Arduino UNO pourrait se retrouver totalement muette. Rien de matériellement catastrophique et d'irréparable, mais si cela devait vous arriver, il vous sera alors nécessaire d'utiliser une autre carte Arduino comme programmeur AVR-ISP (ou un vrai programmeur AVR-ISP) pour réinstaller tous les éléments dans la mémoire de l'ATmega16U2 et peut-être de l'ATmega238P. Vous voici prévenu.

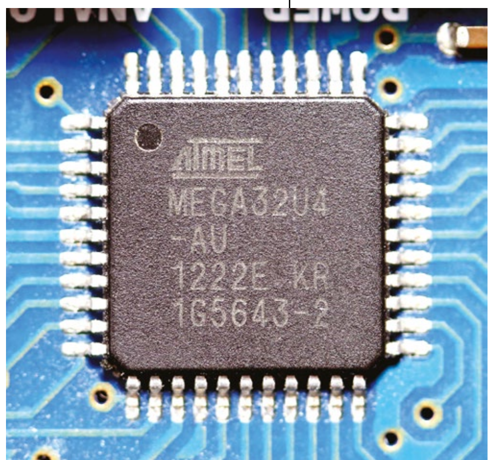
Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

1. BOOTLOADER ET PRINCIPE DE FONCTIONNEMENT

Posons dans un premier temps correctement le décor. Un microcontrôleur comme l'AVR d'Atmel qui est au cœur d'une carte Arduino, se programme normalement d'une certaine façon à l'aide d'un programmeur dédié. C'est la raison d'être du connecteur ISCP de 6 broches en bordure de la carte. Pour simplifier les choses et rendre la programmation du composant indépendante d'un matériel spécifique, un autre mode de programmation peut être utilisé : un bootloader.

Les AVR disposent d'une fonctionnalité permettant de démarrer, dès la mise sous tension, un tout petit programme avant de passer la main à un autre, chargé de faire le gros du travail : ce code c'est le bootloader. Celui-ci dispose de fonctionnalités qu'un programme standard n'a pas et en particulier la capacité d'écrire dans la mémoire flash. L'idée est, après une réinitialisation par exemple, que le bootloader attende un certain temps une éventuelle communication, si celle-ci n'intervient pas, il passe le relai. Mais si cette communication est initiée dans les temps, le bootloader va alors servir de relais pour la programmation de la mémoire et charger le programme reçu avant de redémarrer. C'est précisément ce qui arrive lorsque vous cliquez sur le bouton ***Téléverser*** dans l'environnement Arduino, le bootloader Optiboot s'active et vous permet d'enregistrer votre croquis sur la carte via USB.

L'ATmega32U4 au cœur des Arduino Leonardo (obsolète) et Micro/(Pro) équivaut à un l'ATmega328P d'un Arduino UNO en termes de ressources. Il intègre en plus des fonctionnalités permettant de créer des périphériques USB comme des émulations de clavier, souris, instruments de musique MIDI, séquenceur MIDI, etc., à l'aide de simples croquis.



Sans le bootloader dans l'ATmega238P, il n'est pas possible de charger un croquis dans la mémoire via le câble USB. Ceci explique par exemple pourquoi si vous retirez l'ATmega238P de votre carte et le remplacez par un composant tout neuf, il ne vous sera pas possible de le programmer de cette façon. Par défaut, les AVR neufs sont vierges et ne sont pas configurés pour exécuter un bootloader. Il faut tout d'abord programmer et paramétrer tout cela en utilisant le connecteur ICSP et utiliser l'entrée ***Graver la séquence d'initialisation*** (bootloader en bon français tout joli) du menu ***Outils*** de l'IDE.

L'ATmega16U2 qui sert de relais USB/série n'utilise par défaut pas de bootloader, mais ce microcontrôleur peut être configuré dans ce sens (en fait si, il y a un bootloader DFU, mais ce n'est pas important ici). De ce fait, avec un bootloader adapté il devient possible de non seulement utiliser l'ATmega16U2 comme relais entre l'USB et l'ATmega238P, mais également de l'utiliser pour y placer ses propres croquis. La logique serait la suivante :

- Si l'ATmega16U2 démarre sur le bootloader « maison », il sert de relais pour la programmation standard de la carte Arduino (et donc de l'ATmega238P), fonction remplie par le croquis standard dans la configuration d'origine.
- S'il démarre normalement et que le bootloader passe la main au croquis dans l'ATmega16U2, les ceux croquis fonctionnent de façon indépendante, mais disposent d'une liaison série entre eux. Dans ce cas de figure, le croquis dans l'ATmega238P n'a plus accès au moniteur série de l'environnement Arduino, mais communique avec

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

l'ATmega16U2. Sauf, bien entendu, si le croquis dans l'ATmega16U2 sert de relais, mais là, il n'y a plus vraiment d'intérêt à changer le comportement d'origine qui est exactement le même.

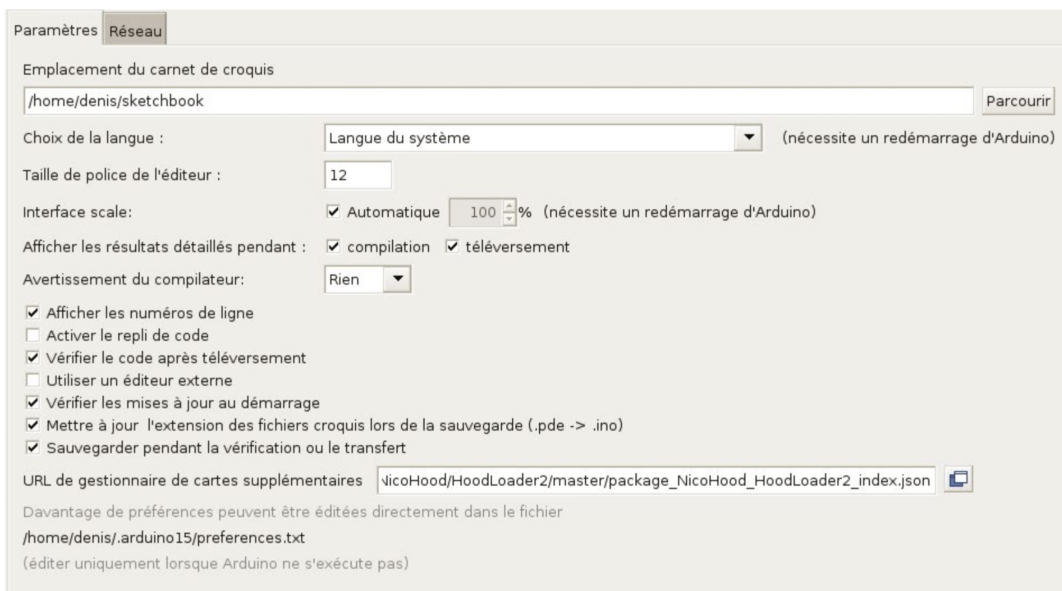
Ce comportement est précisément celui de HoodLoader2, si ce n'est que NicoHood ne s'est pas limité à cela. En effet, il a non seulement développé le bootloader en question, mais également tout le nécessaire pour simplifier au maximum l'installation et l'utilisation au sein de l'environnement Arduino. Une fois HoodLoader2 installé dans l'ATmega16U2 de votre carte Arduino, il vous suffit de spécifier le bon type de carte dans le menu ***Outils*** pour programmer soit l'ATmega16U2, soit l'ATmega238P, sans autre manipulation nécessaire. C'est là justement toute la beauté de la réalisation de NicoHood par rapport à d'autres solutions similaires.

2. INSTALLATION DE HOODLOADER2

Je me baserai ici sur la dernière version à jour de l'environnement Arduino (1.8.1 à cette date), mais ces manipulations devraient également fonctionner avec n'importe quelle version égale ou supérieure à 1.6.6 (testé par mes soins avec la 1.6.13).

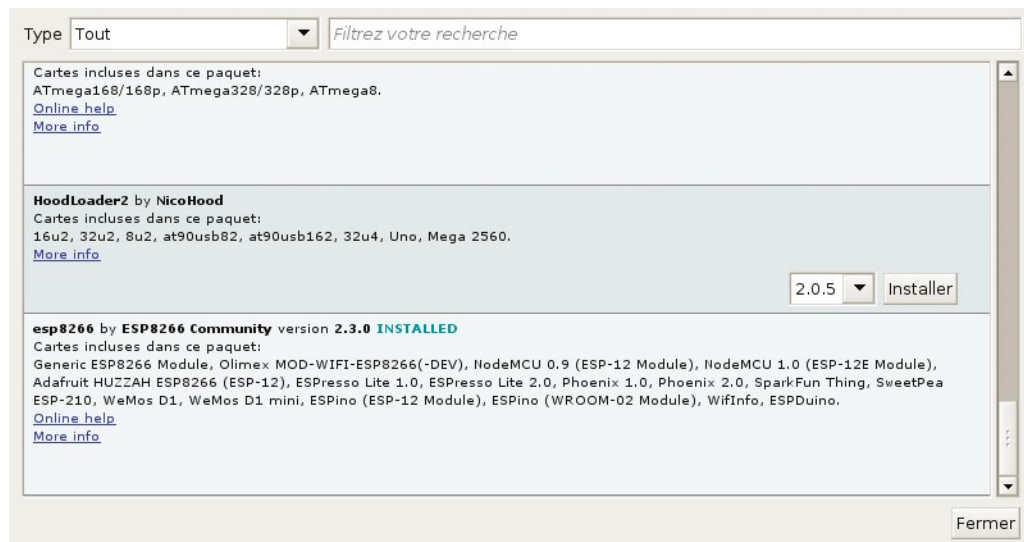
Pour pouvoir profiter des avantages de HoodLoader2, deux manipulations importantes doivent être opérées : préparer l'environnement Arduino pour supporter ces modifications et reprogrammer le contenu de l'ATmega16U2 de la carte UNO.

L'installation des éléments logiciels se fera entièrement (ou presque) dans l'environnement lui-même. Commencez par afficher les préférences de l'environnement via le menu ***Fichier***. Là, vous remarquerez une ligne ***URL de gestionnaire de cartes supplémentaires***. Un clic sur le bouton à droite du champ de saisie vous ouvrira une fenêtre permettant de saisir une URL par ligne. Ces lignes indiquent où l'environnement Arduino, et plus précisément le gestionnaire de cartes, doit chercher les informations pour installer le support d'autres plateformes (ESP8266, ATtiny, etc.). Pour pouvoir installer le support HoodLoader2, utilisez l'URL suivante : **https://raw.githubusercontent.com/NicoHood/HoodLoader2/master/package_NicoHood_HoodLoader2_index.json**.



La fenêtre des préférences de l'environnement Arduino est l'endroit où vous pouvez spécifier des URL permettant l'installation du support pour d'autres cartes que les Arduino officielles et compatibles.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



La liste des plateformes proposées par le gestionnaire de cartes est fonction des modèles supportés par défaut, mais également des URL que vous avez spécifiées dans les préférences.

Aucun redémarrage de l'environnement ne sera nécessaire et vous pourrez alors lancer le gestionnaire de cartes via le menu ***Outils*** et ***Type de carte***, puis parcourir la liste à la recherche de ***Hoodloader2 by NicoHood*** puis cliquer sur ***Installer*** pour ajouter le support de cette plateforme. La version actuelle est la 2.0.5, mais il est possible qu'assez rapidement celle-ci soit remplacée par une nouvelle qui verra le problème dont je vais parler dans un instant corrigé. Si, lorsque vous lirez cet article, la version est supérieure à 2.0.5, ce qui va suivre ne s'appliquera sans doute pas.

En principe à ce stade, tout est prêt pour passer à la phase « matérielle ». La prochaine étape consiste à installer le bootloader dans l'ATmega16U2. Plusieurs solutions sont à votre disposition pour ce faire, mais une seule d'entre elles n'implique que la carte Arduino UNO elle-même. L'idée est la suivante : programmer un croquis dans l'ATmega328P de façon à provoquer une reprogrammation de l'ATmega16U2 en reliant certaines broches des connecteurs ICSP. De cette façon, l'ATmega328P va faire un *reset* de l'ATmega16U2, reprogrammer sa mémoire et configurer ses « fusibles » pour activer la fonctionnalité bootloader.

C'est la méthode que nous allons utiliser ici et NicoHood met à disposition un croquis pour ce faire, celui-ci est même compris dans les fichiers qui ont été installés avec le support de la plateforme via le gestionnaire de cartes. Petit problème cependant, ce croquis, ainsi que d'autres formant des exemples d'utilisation de Hoodloader2 se trouve à un endroit que l'environnement Arduino ne « voit » pas par défaut comme contenant des exemples.

Notez que ce problème est corrigé dans la version de développement actuellement disponible sur GitHub, grâce à une contribution d'un utilisateur en septembre dernier, mais que le correctif en question n'est pas intégré dans la version 2.0.5, mise à disposition quelques jours seulement avant la contribution de cet utilisateur. Lorsque la 2.0.6 sera disponible, il ne sera donc plus nécessaire de corriger cela à la main.

Pour procéder à la correction, vous devrez vous rendre dans le répertoire de gestion des données de l'environnement Arduino. Celui-ci, sous Windows, se trouve dans le répertoire `AppData` (caché par défaut) de votre répertoire utilisateur, sous-répertoire `Local` puis `Arduino15` (ce qui correspond à `~/.arduino15/` sous GNU/Linux, et à `~/Library/Arduino15` sous MacOS).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Dans ce répertoire, vous trouverez les sous-répertoires `packages`, `HoodLoader2`, `hardware`, `avr` puis `2.0.5`. Là, se trouve un sous-répertoire `examples` contenant les croquis qui nous intéressent et qui ne sont, pour l'instant, pas visibles dans l'environnement Arduino. Vous devrez alors :

- créer un répertoire `libraries` et un sous-répertoire `HoodLoader2`,
- déplacer `examples` et son contenu dans `HoodLoader2`,
- créer un fichier `HoodLoader2.h` à ce même endroit, contenant une simple ligne de commentaire (`//HoodLoader2` par exemple).

Le fichier `HoodLoader2.h` est important pour éviter un message d'erreur de l'environnement Arduino précisant que la bibliothèque n'est pas valide. Une fois ces manipulations effectuées et l'IDE Arduino redémarré, en basculant sur n'importe quelle carte Hoodloader2 via ***Outils*** et ***Type de carte***, les exemples seront visibles dans ***Fichier***, ***Exemples***.

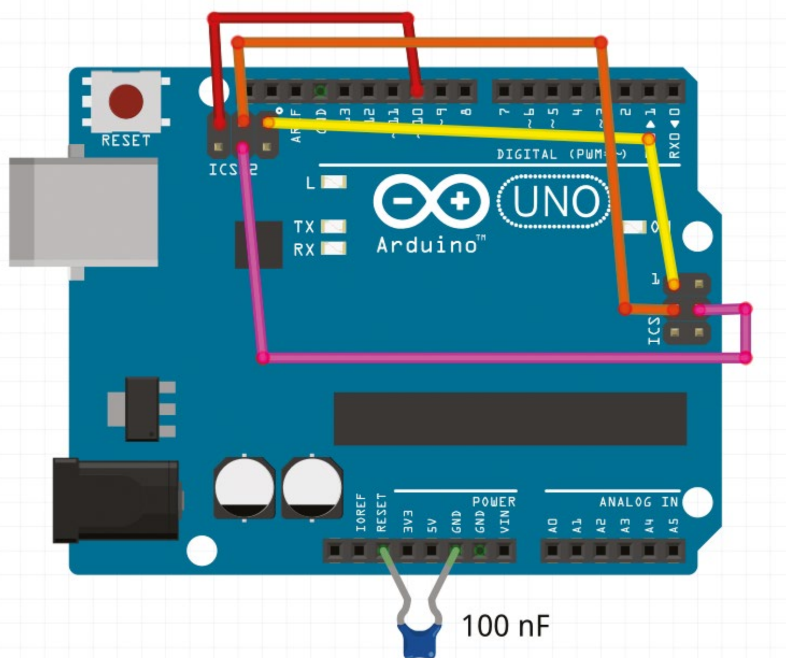
À présent, procédez comme suit :

- sélectionnez n'importe quelle carte Hoodloader2,
- ouvrez le croquis ***Installation_Sketch*** via le menu ***Fichier***, ***Exemples***, ***Hoodloader2***,
- sélectionnez à nouveau le type de carte « Arduino/Genuino Uno ».

En tout début de ce croquis d'installation se trouve une ligne (11) précisant `#define HEXFILE HEXFILE_HoodLoader2_0_5_Uno_atmega16u2_hex`. Cette déclaration de la macro `HEXFILE` spécifie le « fichier » du bootloader à programmer. Les lignes qui suivent dans le croquis sont également des déclarations de macros, il y en a une pour chaque type de carte et de microcontrôleur supporté (Hoodloader2 ne se limitant pas à l'Arduino UNO). Dans le cas qui nous intéresse ici, `HEXFILE_HoodLoader2_0_5_Uno_atmega16u2_hex` est la macro à utiliser et c'est celle présente par défaut (du moins avec la version 2.0.5).

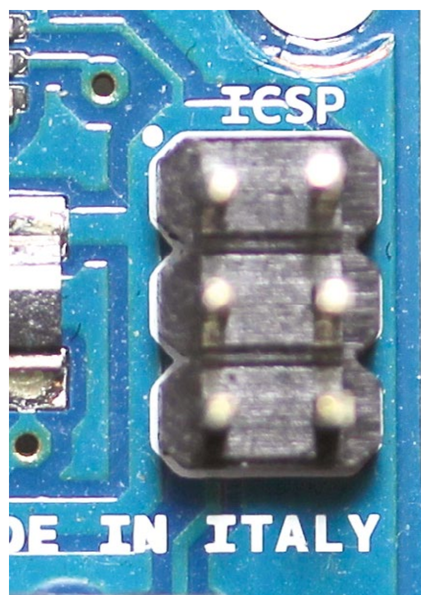
L'autre macro qui pourrait vous être utile est `HEXFILE_Arduino_COMBINED_dfu_usbserial_atmega16u2_Uno_Rev3_hex_Full` (ligne 34). En modifiant la déclaration de `HEXFILE` pour la faire correspondre à cette macro, vous pouvez réinstaller le croquis initial dans l'ATmega16U2 et revenir à un fonctionnement « normal » de la carte UNO.

Une fois `HEXFILE` à la bonne valeur ou que vous vous soyez simplement assuré que la valeur par défaut est la bonne, il ne vous reste plus qu'à programmer votre Arduino UNO avec ce croquis. Celui-ci démarrera automatiquement, mais ce n'est pas grave, puisque les deux connecteurs ICSP ne sont pas encore reliés.



Pour que le croquis d'installation puisse faire son travail, il faut relier les deux connecteurs ICSP de cette façon et ajouter un condensateur de 100nF entre la masse et la broche RESET de la carte Arduino (notez que certains clones Arduino utilisant un ATmega16U2 ont tantôt le connecteur ICSP tourné à 180°).

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Le connecteur ICSP présent sur toutes les cartes Arduino à base d'Atmel AVR permet une méthode alternative de programmation du microcontrôleur. C'est aussi par ce biais que vous pouvez programmer un AVR totalement vierge pour le rendre compatible avec l'environnement de développement Arduino (installation du bootloader et configuration des fusibles).

Vient alors l'étape la plus importante, la programmation de l'ATmega16U2. Débranchez votre carte Arduino du port USB et procédez au câblage des deux connecteurs ICSP : MISO sur MOSI, MOSI sur MISO, SCK sur SCK et le RESET de l'ATmega16U2 sur la broche 10 de la carte Arduino. Placez également un condensateur de 100nF entre la masse et la broche RESET de la carte (celui-ci est destiné à empêcher un reset de l'ATmega328P par l'ATmega16U2). Vérifiez les connexions plusieurs fois puis branchez l'Arduino UNO en USB et attendez.

Le croquis programmé dans l'ATmega328P va alors s'exécuter et reprogrammer l'ATmega16U2. Cette opération ne prendra pas plus d'une trentaine de secondes maximum. Dans un premier temps, la led « L » de la carte clignotera lentement (une fois par seconde), puis la programmation aura lieu et la led se mettra à clignoter rapidement (dix fois par seconde) pour indiquer que l'opération a réussi. SOYEZ PATIENT ! Si la led ne clignote pas rapidement au bout de plus de 30 secondes, c'est que la programmation a échoué. Vérifiez les connexions et la présence du condensateur (au minimum 100nF), puis éventuellement, accédez au moniteur série de l'environnement Arduino pour voir les messages du croquis (115200 bps) et vous assurer qu'il se lance correctement.

Une fois Hoodloader installé et la led « L » clignotant rapidement, débranchez la carte du port USB puis retirez les câbles reliant les deux connecteurs ICSP ainsi que le condensateur. Votre Arduino UNO est maintenant prêt à être utilisé d'une nouvelle façon et avec les deux microcontrôleurs que nous désignerons maintenant de la même manière que la documentation de Hoodloader2 : « USB MCU » pour l'ATmega16U2 et « I/O MCU » pour l'ATmega328P, ou en d'autres termes, respectivement, le microcontrôleur chargé de l'USB et celui chargé des entrées/sorties.

3. UTILISEZ HOODLOADER2 ET L'ARDUINO UNO MODIFIÉ

À ce stade, vous disposez sur la carte de deux microcontrôleurs facilement programmables. Pour sélectionner l'un ou l'autre, il vous suffit de choisir la bonne « carte » dans le menu ***Outils*** :

- « Hoodloader2 Uno » pour programmer l'I/O MCU (l'ATmega328P). En choisissant cette carte, l'environnement Arduino va procéder à un reset de l'ATmega16U2 qui va démarrer sur son bootloader chargé de faire la liaison avec l'ATmega328P qui sera, à son tour, redémarré et qui lui aussi démarrera sur son bootloader pour pouvoir procéder au chargement de votre croquis. Notez qu'il ne vous est plus possible d'utiliser la carte « Arduino/Genuino Uno » puisque l'ATmega16U2 ne fait office de relais que lorsqu'il démarre sur son bootloader.
- « Hoodloader2 16u2 » pour programmer l'USB MCU (l'ATmega16U2). Là, l'environnement Arduino s'adresse directement au microcontrôleur USB sans s'occuper de l'ATmega328P.

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Pour asseoir tout cela, voyons un petit exemple simple. Dans un premier temps, choisissez « Hoodloader2 16u2 » puis chargez le croquis suivant :

```
void setup() {
  pinMode(LED_BUILTIN_RX, OUTPUT);
  pinMode(LED_BUILTIN_TX, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN_RX, LOW);
  digitalWrite(LED_BUILTIN_TX, LOW);
  delay(100);
  digitalWrite(LED_BUILTIN_RX, HIGH);
  digitalWrite(LED_BUILTIN_TX, HIGH);
  delay(1000);
}
```

L'USB MCU n'a pas accès à la led « L » mais, par contre, c'est lui qui contrôle les leds « RX » et « TX » de la carte et ses broches sont directement référencées par les macros `LED_BUILTIN_RX` et `LED_BUILTIN_TX`. Nous n'avons donc ici rien de plus que le croquis *Blink*, adapté à l'ATmega16U2. Notez que les niveaux logiques sont inversés, un `LOW` allume les leds et un `HIGH` les éteint, car les deux sont connectés aux broches correspondantes et à la tension d'alimentation (et non à la masse).

Une fois le croquis compilé et enregistré dans la mémoire, basculez sur la carte « Hoodloader2 Uno » et utilisez le croquis exemple *Blink* standard. Compilez-le et chargez-le dans la mémoire.

Le résultat sera, en toute logique, un clignotement bref et simultané des deux leds « RX » et « TX », et un clignotement d'une seconde sur « L ». Les deux premières leds sont contrôlées par l'USB MCU et la dernière par l'I/O MCU. Nous avons donc deux croquis différents fonctionnant indépendamment sur les deux microcontrôleurs d'une même carte.

Il est important de comprendre que lorsque l'USB MCU exécute son croquis, il ne fait plus son travail de relais entre le port USB et l'ATmega328P que vous aviez l'habitude d'utiliser sur votre Arduino UNO. Ce n'est que lorsqu'il démarre sur son bootloader qu'il remplit cette tâche. Vous pouvez revenir à un fonctionnement standard sans inverser toute la procédure en utilisant le croquis suivant pour l'USB MCU (également disponible sous forme d'exemple sous le nom « RunBootloader ») :

```
#include <avr/wdt.h>

void setup() {
  cli();
  *(uint16_t*)MAGIC_KEY_POS = 0x7777;
  wdt_enable(WDTO_120MS);
}

void loop() {}
```

Ces quelques lignes provoquent une réinitialisation de l'ATmega16U2 directement sur son bootloader. Lorsque vous brancherez la carte en USB, les deux microcontrôleurs vont se

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

réinitialiser normalement, puis ce croquis sera exécuté sur l'USB MCU qui se réinitialisera une seconde fois et restera sur le code du bootloader. Ce faisant, la carte pourra à nouveau être utilisée avec l'entrée « Arduino/Genuino Uno ».

Note : Attention, en jonglant entre des types de cartes, à ce qui semble être un bug mineur de l'environnement Arduino. Lorsque vous ouvrez un exemple associé à un type de carte et y faites des modifications, une tentative de sauvegarde affiche un message vous précisant qu'il faut l'enregistrer dans votre carnet de croquis. C'est le comportement normal. Cependant, si vous ouvrez un exemple pour une carte, changez de plateforme puis tentez de sauvegarder, l'opération réussira, écrasant par la même occasion l'exemple original. Faites donc attention en jouant avec les exemples et les changements de cartes et éventuellement, prenez l'habitude de toujours enregistrer un exemple modifié avec Ctrl+Maj+S et non Ctrl+S.

Un autre point important en rapport avec ce nouveau mode de fonctionnement concerne le port série de l'I/O MCU. Celui-ci est relié, sur le circuit, à l'USB MCU qui fait normalement la liaison avec l'USB. Or, à présent, cette liaison n'est plus faite si l'USB MCU exécute lui aussi un croquis. Ceci signifie qu'un `Serial.print()`, par exemple, dans un croquis exécuté par l'I/O USB ne vous sera plus retransmis dans le moniteur série (sauf en forçant l'USB MCU à redémarrer sur son bootloader avec le croquis précédent par exemple).

Il y a donc deux modes d'utilisation de votre carte Arduino UNO à présent. Soit vous chargez ce mini-croquis dans l'USB MCU et la carte se comporte comme avant la modification, soit vous faites travailler de concert les deux microcontrôleurs sachant que le `Serial` de l'I/O MCU est connecté au `Serial1` de l'USB MCU et que le `Serial` de l'USB MCI est connecté via le port USB à votre ordinateur.

Ceci peut, certes, rendre la mise au point de croquis complexes un peu plus difficile mais, d'un autre côté, vous ajoutez des fonctionnalités USB à une carte qui initialement n'en possède pas. On ne peut pas tout avoir, le beurre, l'argent du beurre et la bibliothèque USB HID de la crémière...

4. UN EXEMPLE CONCRET : L'ARDUINO UNO COMME CLAVIER USB

Vous l'avez compris, avec Hoodloader2 vous pouvez utiliser l'ATmega16U2 à la manière d'un ATmega32U3 d'une carte Arduino Micro et donc obtenir les mêmes fonctionnalités, dont l'émulation de clavier et de souris USB (classe HID). Il y a cependant un petit souci ou plus exactement un point auquel il faut faire très attention : l'ATmega16U2 dispose de ressources limitées. Non seulement une très grande partie de ses broches ne sont pas utilisables (juste 2 leds sur 17 et 18, et 4 broches à souder sur 4, 5, 6 et 7 (PWM)), mais la mémoire vive disponible est vraiment réduite (512 octets). De plus, la mémoire flash pour les croquis peut sembler suffisante avec ses 16 Ko, mais il ne reste en réalité que 12288 octets pour vos croquis après installation du bootloader.

L'ATmega16U2 n'offrira donc que peu de possibilités pour des projets intéressants. Ce qu'il est possible de faire, en revanche, c'est de confier toute la partie entrée/sortie et le « gros oeuvre » à l'ATmega328P et de laisser le soin à l'ATmega16U2 de se charger de la communication.

Pour illustrer cette répartition des tâches, nous allons utiliser deux croquis selon le cahier des charges suivant : l'ATmega328P

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

va envoyer du texte sur son port série toutes les secondes et l'ATmega16U2 va utiliser celui-ci pour émuler une saisie au clavier. C'est simple, mais efficace. Commençons donc par le premier croquis destiné à ATmega16U2 :

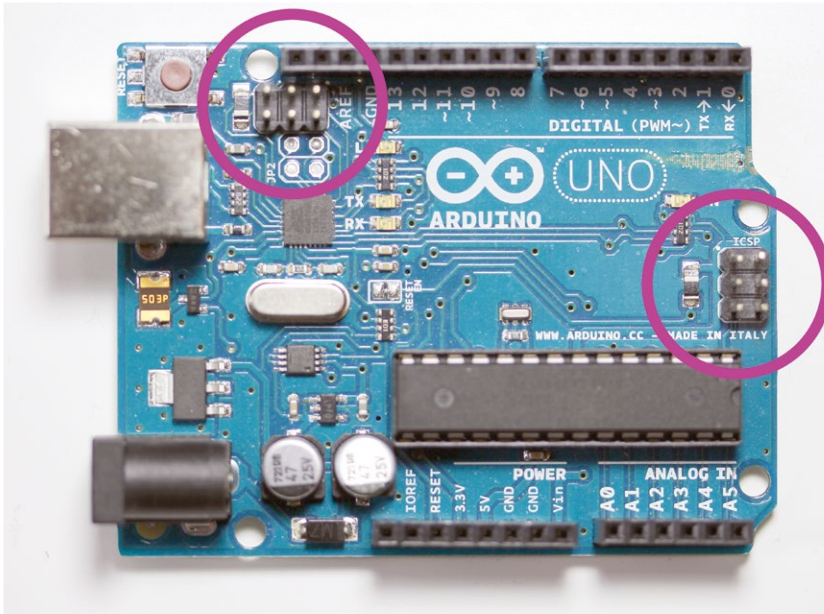
```
#include <Keyboard.h>

void setup() {
  // configuration des leds et extinction
  pinMode(LED_BUILTIN_RX, OUTPUT);
  pinMode(LED_BUILTIN_TX, OUTPUT);
  digitalWrite(LED_BUILTIN_RX, HIGH);
  digitalWrite(LED_BUILTIN_TX, HIGH);

  // Liaison série vers l'ATmega328P à 2400 bps
  Serial1.begin(2400);

  // Mise en route du clavier USB HID
  Keyboard.begin();
}

void loop() {
  // A-t-on des données à lire ?
  if (Serial1.available() > 0) {
    // récupération d'un caractère
    char c = Serial1.read();
    // on envoie le caractère comme tapé au clavier
    Keyboard.print(c);
  }
}
```

Rien de bien complexe dans l'ensemble, nous nous contentons d'initialiser à la fois la liaison série et l'émulation clavier puis, dans la boucle principale, nous prenons chaque caractère qui arrive et l'envoyons comme une saisie clavier.

Côté ATmega328P, c'est tout aussi simpliste :

```
void setup() {
  // led L en sortie
  pinMode(LED_BUILTIN, OUTPUT);
  // Liaison série vers l'ATmega16U2 à 2400 bps
  Serial.begin(2400);
  // Petite pause pour laisser le temps de démarrer
  delay(5000);
}

void loop() {
  // Ecriture de texte sur le port série
  Serial.print("Coucou ");

  // Clignotement de la led à chaque envoi
  digitalWrite(LED_BUILTIN, HIGH);
  delay(100);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



La carte Arduino UNO possède deux connecteurs ICSP, un pour l'ATmega328P et un autre pour l'ATmega16U2. Pour programmer le bootloader Hoodloader2 dans l'ATmega16U2 sans avoir recours à un équipement complémentaire comme un programmeur AVR-ISP, il vous suffit de relier les deux connecteurs et d'utiliser un croquis spécial d'installation.

Après configuration de la liaison série et de la led « L », nous envoyons environ toutes les secondes le texte « coucou » en faisant clignoter la led (comme indicateur en cas de problème).

Les difficultés sont plus subtiles qu'il n'y paraît. En termes de ressources par exemple, le croquis pour l'ATmega16U2 consomme à lui seul 47% de la mémoire flash (5800 octets) et la même proportion pour la mémoire vive (245 octets). Ce qui démontre qu'on est effectivement rapidement limité avec ce microcontrôleur.

De plus, la gestion des délais et de la chronologie est importante. Remarquez la faible vitesse de communication utilisée. 115200 bps semblait clairement trop élevé et une partie des caractères n'étaient pas envoyés en USB. Ce n'est pas une question de vitesse de communication, mais le fait que le croquis de l'ATmega16U2 doit créer un rapport HID et l'envoyer pour chaque caractère reçu.

Enfin, le délai de 5 secondes à la fin de la fonction `setup()` du croquis pour l'ATmega328P est également important. Il est destiné à laisser le temps au bootloader de l'ATmega16U2 de passer la main au croquis qui doit ensuite initialiser la communication HID. De plus, l'ordinateur sur lequel est branchée la carte UNO doit également énumérer les périphériques USB proposés et les prendre en charge. Tout ceci prend du temps et nous ne pouvons pas envoyer sauvagement du texte sans une petite pause.

Le maître mot dans l'utilisation d'une carte Arduino de cette façon est « économie ». Vous devez garder à l'esprit qu'il est nécessaire d'optimiser au mieux le croquis de l'ATmega16U2 et de le maintenir le plus concis possible.

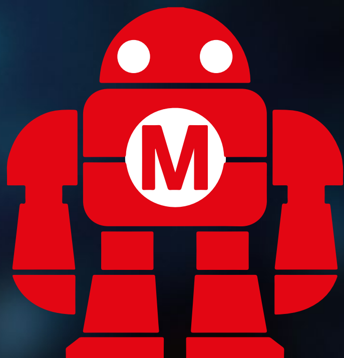
5. POUR FINIR... OU COMMENCER...

Il est amusant de constater qu'au sein même des cartes Arduino se cachent des ressources intéressantes et tantôt très utiles. Il est certes indéniable que les manipulations décrites ici ne sont pas des plus évidentes, mais le résultat est bien là : il est possible d'utiliser une carte Arduino UNO là où normalement une Leonardo ou une Micro serait nécessaire.

Nous n'avons fait qu'entrevoir le travail énorme et admirable de NicoHood qui se doit d'être salué comme il le mérite (tant pour HoodLoader2/1, que pour sa bibliothèque HID-Project ou encore IRLremote). Son code est une véritable source d'inspiration et de motivation, et cela ouvre la porte à bien des possibilités. Il vous suffit de tester, échouer et recommencer, et vous finirez par arriver à vos fins.

NicoHood le précise d'ailleurs sur l'une des pages de documentation de HoodLoader2 : « *J'ai tout appris par moi-même. Apprendre en faisant. J'ai débuté en partant de zéro fin janvier 2014 dans le domaine Arduino/électronique. Donc vous pouvez aussi le faire, je crois en vous :)* ». Bien dit ! **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Maker Faire® Paris

9 > 11 juin 2017

Cité des Sciences et de l'Industrie

Appel aux Makers

JUSQU'AU 15 AVRIL

Maker Faire est à la fois une fête de la science, une foire populaire et un événement de référence pour l'innovation. Ce concept regroupe stands de démonstration, ateliers de découverte, spectacles et conférences autour des thèmes de la créativité, de la fabrication, et des cultures Do It Yourself.
Aujourd'hui, plus de 200 éditions réunissent dans 38 pays **des communautés de passionnés, experts ou débutants, qui partagent l'envie de créer, fabriquer et apprendre les uns des autres.**
La 4ème édition de **Maker Faire Paris** se tiendra à la Cité des Sciences et de l'Industrie.

Vous souhaitez participer ?
Rejoignez la communauté des Makers en vous inscrivant à notre appel !

Clôture des inscriptions le 15 avril 2017 à minuit !

Un événement

Présenté par

Partenaire

Inscription sur
paris.makerfaire.com

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

LES CODES TOURNANTS OU COMMENT NE PAS ENVOYER LE MÊME MESSAGE DEUX FOIS

Denis Bodor

Lorsqu'on utilise une télécommande que ce soit à fréquence ou infrarouge, un message est transmis de l'émetteur au récepteur. Cela fonctionne très bien, mais si quelqu'un capte un message, le copie et l'envoie à votre place, le voici capable de piloter votre installation. Ceci s'appelle le rejeu et pour contrer ce type d'attaque, l'astuce consiste à ne jamais envoyer deux fois le même message. C'est précisément ce que font, plus ou moins toujours selon les modèles, les télécommandes pour le verrouillage centralisé des voitures par exemple et ceci repose sur une notion particulière : les codes tournants. Chose que nous allons découvrir en pratique dans cet article...

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Dans le numéro précédent, nous avons vu qu'avec un matériel comme le HackRF One ou n'importe quel autre émetteur/récepteur SDR (ou montage fait maison), il était possible de capturer un signal radio, l'enregistrer et le rejouer pour prendre la place de la télécommande à fréquence initialement utilisée. Ce genre de pratique est applicable pour bon nombre de dispositifs du carillon sans fil à la porte de garage en passant par la télécommande IR de votre téléviseur.

Le fait de chiffrer les données transmises ne règle pas le problème. Si vous chiffrez un message « ouvre garage » en « eofjdlksfuir » et que le récepteur le déchiffre pour retrouver le message initial, ceci n'empêchera absolument pas le fait que quelqu'un puisse capturer « eofjdlksfuir » et l'envoyer à votre place. Le chiffrement n'est donc pas la solution au problème de rejeu.

La solution consiste à utiliser des codes tournants, *rolling codes* ou *hopping codes* en anglais et son fonctionnement n'a rien de bien complexe une fois le mécanisme général compris. Nous allons mettre en pratique cette technique avec une carte Arduino et réaliser notre propre implémentation dans la suite de cet article. Comme vous allez le découvrir, ce principe ne se limite pas aux télécommandes de toutes sortes, mais rejoint également la notion d'authentification à deux facteurs consistant à saisir son nom d'utilisateur et son mot de passe, puis à confirmer son identité en fournissant un code, souvent numérique, provenant soit d'une application pour smartphone, soit d'un petit porte-clé doté d'un afficheur. Ce type de système est, par exemple, proposé par *Blizzard* pour son service de jeux en ligne *Battle.net* (WoW, StarCraft, Diablo III, OverWatch, etc.).

Avant de poursuivre, je tiens à préciser ici que l'objet de l'article est de vous présenter et d'illustrer par la pratique ce qu'est un système de codes tournants et en **AUCUN CAS** de fournir une solution sécurisée. La création d'un système sécurisé robuste repose sur une compréhension parfaite et l'utilisation de mécanismes et de principes cryptographiques qui sortent totalement du cadre de cet article (et je dois l'avouer, de mes compétences).

Les codes tournants ou rolling codes sont généralement très utilisés, en compagnie de systèmes de chiffrement, sur les systèmes de verrouillage centralisé des véhicules récents. C'est également quelque chose qu'on voit peu à peu apparaître dans les télécommandes de portes de garage, même si c'est encore relativement rare...

1. LE PRINCIPE DES CODES TOURNANTS

Dans l'absolu, le cahier des charges permettant de mettre en œuvre des codes tournants est d'une grande simplicité : l'émetteur et le récepteur utilisent une série de valeurs ou de codes identiques et, à chaque fois qu'un code est utilisé, on passe au suivant. Dans les faits, à chaque message l'émetteur envoie le prochain code dans la liste, peu importe que le récepteur le reçoive effectivement ou non, la communication est à sens unique. Un code envoyé n'est jamais réutilisé, on ne revient jamais en arrière dans la série.

Côté récepteur en revanche il faut prendre en compte le fait que l'émetteur a peut-être envoyé un code de la liste alors qu'on ne l'a pas reçu. Le code effectivement réceptionné pourrait parfaitement ne

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

pas être le prochain, mais l'un de ceux qui se trouvent plus loin dans la série. Cependant cette série est connue de part et d'autre, il suffit au récepteur de comparer le code reçu, non pas uniquement au prochain de la série, mais également aux 5, 10, ou 50 suivants.

Si le code est effectivement trouvé, le récepteur considère qu'un certain nombre de codes ont été perdus et que le prochain qui devra être réceptionné se trouve après le code validé (on ne revient jamais en arrière dans la série, oui, je me répète, c'est fait exprès).

Là, deux problèmes doivent vous venir en tête : il faut une liste monstrueuse de valeurs/codes d'un côté comme de l'autre de la transmission, et la question de ce qui se passe si l'émetteur a envoyé davantage de code de la série que la taille de la plage de recherche du récepteur.

Nous verrons le premier point et sa solution dans un instant, mais penchons-nous d'abord sur cette captivante question. La réponse est évidente : ça ne marchera pas car, comme il n'est pas possible de revenir en arrière dans la série, la synchronisation est alors définitivement perdue. Dans la pratique avec des systèmes comme ceux des télécommandes de verrouillage de voitures, la plage ou fenêtre de cherche, côté récepteur, est tout simplement suffisamment grande pour éviter les problèmes.

256 codes envoyés et non reçus est une valeur souvent utilisée et est généralement bien suffisante pour éviter l'effet de quelques pressions accidentelles sur les boutons (dans la poche ou dans un sac) ou même découlant de tentatives pour retrouver son véhicule sur un parking. Bien entendu, si vous appuyez pour vous amuser ou par réflexe nerveux (vous savez comme ces gens énervants qui ne peuvent s'empêcher de faire clic-clic avec leur stylo), il n'est pas impossible que vous finissiez par atteindre la limite. Mais là encore, les constructeurs proposent des solutions pour resynchroniser émetteur et récepteur comme par exemple en accédant au véhicule mécaniquement (serrure) et en opérant une manipulation spécifique alors que la clé est sur le contact (pression longue sur le bouton par exemple).

L'autre problème concerne la série elle-même et il est, bien entendu, impensable d'utiliser réellement une liste faute de place, quelle que soit la plateforme. De telles listes ont, par définition, une fin et surtout, peuvent être copiées. La solution consiste donc à créer un bout de programme qui, à chaque appel, nous donnera une nouvelle valeur qui paraîtra aléatoire. En d'autres termes, nous pouvons remplacer une vraie série par un générateur de nombres pseudo-aléatoires.

2. LES GÉNÉRATEURS DE NOMBRES PSEUDO-ALÉATOIRES

Comme vous le savez, les ordinateurs, les microcontrôleurs et les processeurs fonctionnent de manière déterministe et prévisible (plus ou moins selon le système utilisé et le nombre de mails que vous avez ouverts provenant de soi-disant généreux princes ou diplomates africains). Obtenir des valeurs réellement aléatoires avec ce genre d'équipement est tout bonnement impossible (à moins d'utiliser du matériel spécifique).

Au mieux, il est possible d'utiliser des programmes et/ou des fonctions fournissant des suites de valeurs ayant quelques caractéristiques au fruit du hasard, comme une certaine indépendance des valeurs les unes par rapport aux autres ou encore l'absence de propriétés communes à toutes les valeurs générées. Il ne s'agit cependant pas de valeurs réellement aléatoires qui ne peuvent en aucun cas être

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

obtenues par des méthodes purement arithmétiques. C'est pour cette raison qu'on parle de générateurs de nombres **pseudo**-aléatoires ou PRNG en anglais pour *PseudoRandom Number Generator*.

Ces générateurs utilisent des algorithmes et des méthodes mathématiques pour créer des suites de nombres, mais ils doivent être initialisés à partir d'une valeur précise qu'on appelle graine (*seed*) qui, elle, doit être le plus aléatoire possible. Ainsi dans un croquis Arduino, nous allons généralement utiliser quelque chose comme ceci :

```
void setup() {
  Serial.begin(115200);
  while(!Serial){;}

  int graine = analogRead(0);
  Serial.print("Graine: ");
  Serial.println(graine);

  randomSeed(graine);
  for(int i=0; i < 10; i++) {
    Serial.println(random(1000));
  }
}

void loop() {
}
```

Le générateur de nombres pseudo-aléatoires, que je vais maintenant désigner par le terme PRNG, est initialisé, via `randomSeed()`, avec une graine découlant d'une valeur obtenue par une lecture de l'entrée analogique A0 sur laquelle rien n'est connecté (entrée flottante). Dans la boucle `for` qui s'en suit, `random()` est utilisé pour obtenir, à chaque tour, une valeur pseudo-aléatoire entre 0 et 999 que nous affichons sur le moniteur série.

À chaque exécution du croquis, une série de 10 valeurs est ainsi affichée, précédée par la valeur lue sur A0, à chaque fois différente. Mais si nous connectons A0 à +5V via une résistance de rappel de quelques 10k ohms, le comportement du croquis change du tout au tout. La valeur lue est systématiquement 1023 et la série de valeurs obtenues devient : 561, 29, 195, 691, 775, 975, 456, 176, 211 et 775... à chaque fois !

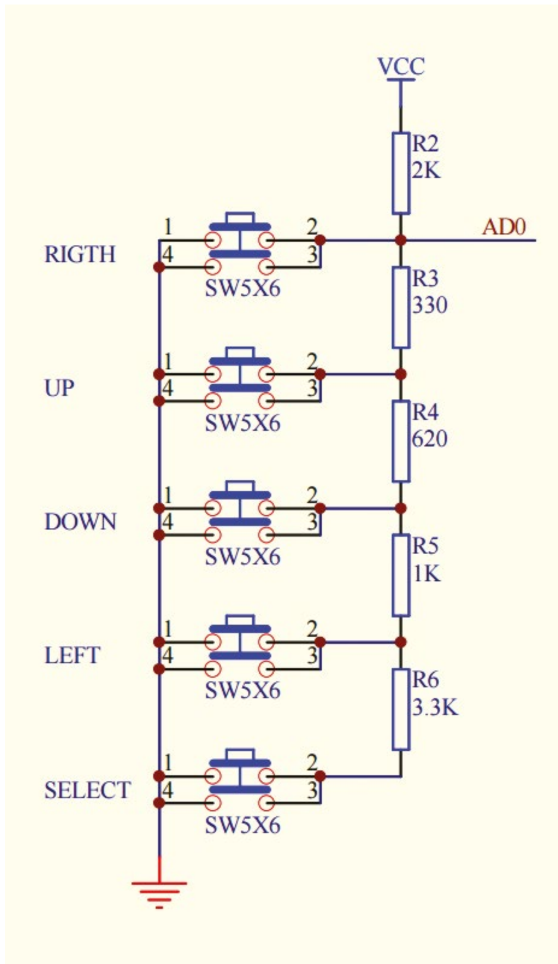
La suite n'a rien de très aléatoire, car la valeur de la graine est tout le temps la même. Pourtant, les valeurs ne semblent pas avoir de rapport entre elles autres, il n'y a pas de suite logique, et elles ne semblent pas non plus répondre à des caractéristiques communes (ce ne sont pas des multiples d'un même nombre, etc.).

La série est donc parfaitement prédictible et reproductible si `randomSeed()` est utilisé avec une valeur connue et fixe. Mieux encore, le même croquis ainsi utilisé fournira des résultats identiques sur deux cartes Arduino utilisant un microcontrôleur AVR. Une Arduino Due en revanche nous fournira une autre suite (925, 41, 600, 964, 424, 705, 455, 202, 763, 938), car le PRNG utilisé est différent. Les bibliothèques standards (libC) utilisées ne sont pas les mêmes pour les Arduino AVR et les Arduino ARM Cortex-M, et l'algorithme est différent.

Néanmoins, nous avons là précisément ce que nous cherchons : une fonction qui génère une série de valeurs pseudo-aléatoires qui peut être à la fois identique côté émetteur comme récepteur et pouvant être initialisée avec une valeur de notre choix (la graine). Le fonctionnement même du PRNG derrière `random()` sort du cadre de cet article, mais en voici une version adaptée pouvant être intégrée dans un croquis :

```
unsigned long x = 1023;

long mon_random() {
  long hi, lo;

  if (x == 0)
    x = 123459876L;
  hi = x / 127773L;
  lo = x % 127773L;
  x = 16807L * lo - 2836L * hi;
  if (x < 0)
    x += 0x7fffffffL;
  return x;
}
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Ce montage de boutons et de résistances, utilisé sur le shield de DERobot, permet de distinguer quel bouton est utilisé à l'aide d'une unique entrée analogique de la carte Arduino.

La variable globale **x** est initialisée, avant le premier appel à cette fonction, avec la valeur de la graine, puis tous les appels consécutifs changeront sa valeur (qui est également retournée par la fonction). Pour obtenir un comportement identique à `random(1000)`, il nous suffirait d'appeler `mon_random() % 1000` (`%` étant l'opérateur modulo, le calcul du reste de la division par 1000). La suite de valeurs obtenues avec **x** initialisé à 1023 est exactement la même que celle dans notre croquis exécuté sur un Arduino à base d'AVR (testé sur Arduino Due).

3. ALLONS-Y !

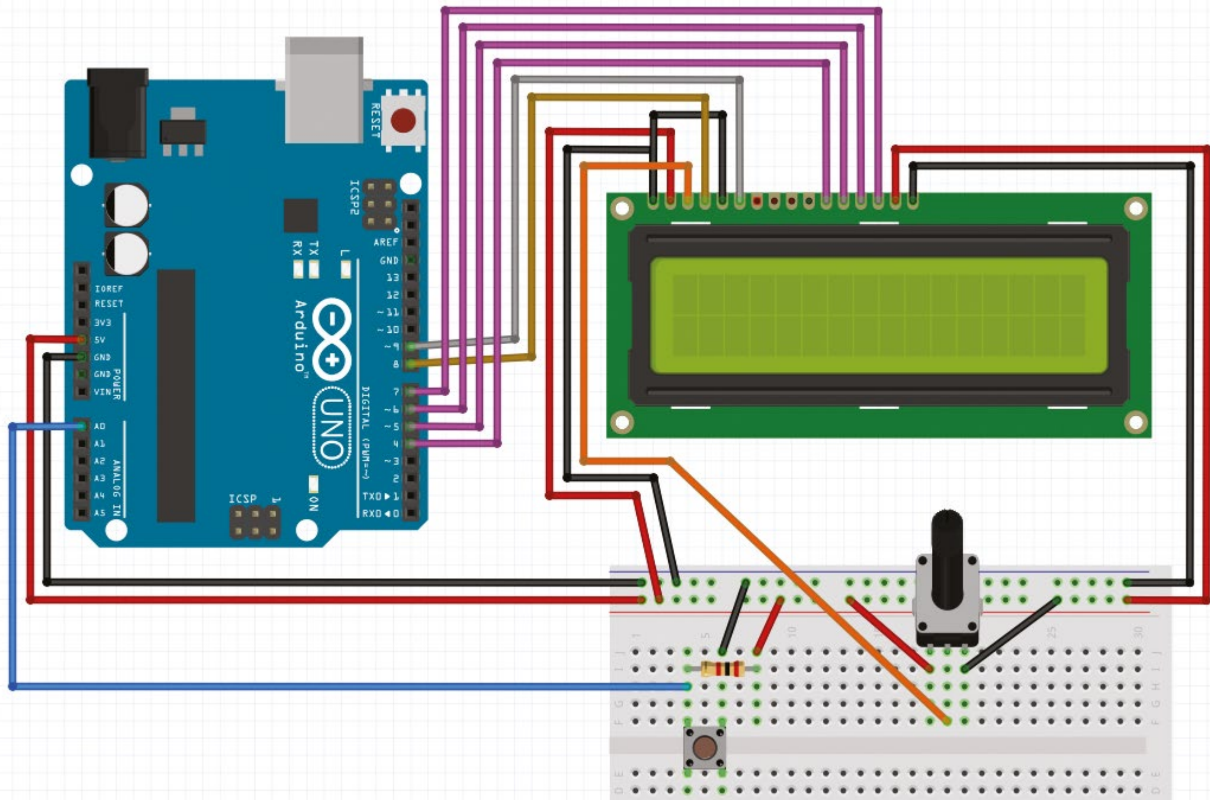
Nous avons maintenant toutes les briques nécessaires pour créer notre implémentation d'un système à codes tournants. L'idée sera donc la suivante en reposant sur deux Arduino :

- un montage « émetteur » autonome disposant d'un afficheur LCD et d'un bouton permettant l'affichage successif des différentes valeurs de la série ;
- un montage « récepteur » maintenant en interne une liste de futures valeurs utilisables, connecté via le moniteur série et permettant d'entrer une valeur et de la vérifier/valider.

Commençons par le montage et le croquis le plus simple, celui de l'émetteur :

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(8,9,4,5,6,7);

void setup() {
  lcd.begin(16, 2);
  lcd.setCursor(0,0);
  lcd.print("Hackable 17");

  randomSeed(42424242);
}

void loop() {
  if (analogRead(0) < 50) {
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Code tournant:");
    lcd.setCursor(0,1);
    lcd.print(random(1,65536));
    delay(500);
  }
}
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



Voici un schéma du montage émetteur. Malgré la complexité apparente, il ne s'agit que d'une liaison classique d'un afficheur LCD alphanumérique compatible HD44780 accompagné d'un bouton poussoir connecté à A0. Les connexions présentées ici sont celles du shield « LCD KeyPad » de DFRobot, si ce n'est que les 4 autres boutons et leurs résistances (+ reset) ne sont pas illustrés ici.

L'afficheur LCD alphanumérique utilisé ici est un shield à 10€ de chez DFRobot comprenant l'écran LCD ainsi d'une série de boutons-poussoirs connectés sur la broche A0 de la carte Arduino. L'afficheur compatible HD44870 est connecté en mode 4 fils à la carte via les broches : 8 (RS), 9 (EN), 4 (D4), 5 (D5), 6 (D6) et 7 (D7). C'est la configuration la plus économique en termes de brochage, mais également la plus lente pour piloter l'afficheur et celle proposant le moins de fonctionnalités. Il est impossible de définir ses symboles personnalisés, par exemple, puisque la broche RW est reliée à la masse.

Les 5 boutons-poussoirs en revanche n'utilisent qu'une seule broche de la carte Arduino grâce à l'utilisation de résistances reliées en série. Ainsi en lisant simplement la valeur d'A0, il est possible de déterminer quel bouton est enfoncé puisque la tension présente dépend du nombre de résistances connectées entre la tension d'alimentation et la masse (principe du diviseur de tension).

Le fonctionnement global du croquis est relativement simple. Nous initialisons l'afficheur ainsi que le PRNG avec une valeur arbitrairement choisie puis, à chaque pression sur l'un des boutons (« RIGHT »), nous affichons simplement la valeur retournée par `random(1,65536)` qui sera donc comprise entre 1 et 65535. Chaque nouvelle utilisation du bouton nous fera donc avancer d'une valeur dans la série pseudo-aléatoire.

L'émetteur a un travail relativement simple puisqu'il ne fait qu'émettre une nouvelle valeur de la série. Notez que je parle d'émettre puisque, même s'il ne s'agit que d'un affichage sur un écran LCD, l'opération consiste effectivement à envoyer une valeur sur un canal de communication. Il se trouve simplement que

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

le canal en question se résume à vos yeux et vos doigts copiant cette information de l'afficheur au moniteur série. L'opération fonctionnera de la même manière en vous remplaçant par un dispositif infrarouge, à fréquence ou même audio (encodage/décodage DTMF).

Le récepteur a à sa charge de prendre en compte une éventuelle perte d'un code et, si tel est le cas, d'assurer un fonctionnement malgré ce problème. Pour ce faire, comme détaillé précédemment, il doit donc disposer d'une solution pour tester une valeur reçue non pas uniquement avec la prochaine valeur attendue dans la série, mais avec un groupe de futures valeurs. Nous devons donc avoir à notre disposition une partie de cette série de valeurs.

Pour cela, nous utiliserons un tableau et commencerons par le remplir :

```
#define TAILLE_BUF 10

unsigned int buf[TAILLE_BUF];

void setup() {
  Serial.begin(115200);
  while(!Serial){;}

  randomSeed(42424242);

  Serial.println("Go go go");

  for(int i=0; i < TAILLE_BUF; i++) {
    buf[i] = random(1,65536);
  }
}
```

La macro `TAILLE_BUF` déterminera la taille de cette « fenêtre de test » et sera utilisée tout au long du croquis. Notre tableau, `buf[]` sera rempli de valeurs dans la fonction `setup()` après avoir initialisé le PRNG avec la même graine que celui de l'émetteur. Une simple boucle `for` permet de parcourir l'ensemble du tableau pour y placer consécutivement les valeurs obtenues via `random(1,65536)`.

À ce stade, nous savons que les 10 premières valeurs que nous recevrons de la part de l'émetteur seront celles du tableau ou, éventuellement, une partie d'entre elles. Afin de ne pas rendre le croquis illisible, nous créons des fonctions permettant deux opérations : chercher une valeur dans le tableau et mettre à jour le tableau si cette valeur est trouvée et donc que le code est accepté.

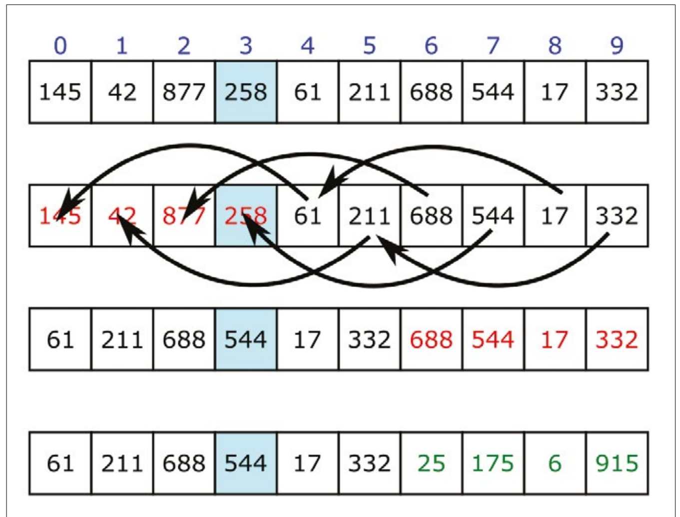
La recherche est relativement simple puisqu'elle consiste à parcourir le tableau et comparer chaque valeur à celle reçue :

```
int cherchecode(unsigned int code) {
  for(int i=0; i < TAILLE_BUF; i++) {
    if(buf[i] == code)
      return i;
  }
  return -1;
}
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Si la valeur passée en argument de l'appel à cette fonction est trouvée, la position dans le tableau est retournée. Si tel n'est pas le cas ou, en d'autres termes, si la valeur reçue ne fait pas partie des 10 (`TAILLE_BUF`) prochaines valeurs attendues, elle retournera -1 (une valeur qui ne peut en aucun cas être celle d'une position dans le tableau).

Vient ensuite la problématique de la mise à jour. Si la valeur reçue est effectivement trouvée, toutes les valeurs à des positions précédant la sienne deviennent obsolètes et inutiles. Nous pouvons donc décaler les valeurs dans le tableau de manière à mettre en première position la prochaine valeur attendue, déplacer les suivantes et compléter par de nouvelles valeurs fournies par `random(1,65536)` :



La mise à jour du tableau se fait à l'aide d'une simple boucle. En haut, le tableau avant mise à jour avec, en bleu, la position du code trouvé. Toutes les valeurs sont copiées aux nouveaux emplacements en écrasant les valeurs inutiles (en rouge) puis le tableau est complété de nouvelles valeurs pseudo-aléatoires (en vert) en écrasant celles déjà présentes (en rouge également).

```
void majcode(int pos) {
  for(int i=0; i < TAILLE_BUF; i++) {
    if(i < TAILLE_BUF-pos)
      buf[i] = buf[i+pos+1];
    else
      buf[i] = random(1,65536);
  }
}
```

La boucle `for` parcourt ici aussi tous les emplacements du tableau, mais pour les positions avant celles correspondant au nombre de valeurs à conserver, on copie la valeur se trouvant à une distance égale à `pos`+1. Une fois passée cette position, il suffit de faire appel à `random(1,65536)` pour stocker les prochaines valeurs de la série. On se retrouve, en sortie de la boucle, avec un tableau tout neuf, avec en première position le code qu'on est censé recevoir la prochaine fois.

Ces deux fonctions utilitaires peuvent ensuite être utilisées dans la fonction `loop()` chargée de réceptionner et traiter les codes via le moniteur série :

```
void loop() {
  while (Serial.available() > 0) {
    unsigned int valeur = Serial.parseInt();

    if(!valeur)
      return;

    Serial.print("recu: ");
    Serial.println(valeur);

    int ret = cherchecode(valeur);
    if(ret >= 0) {
      Serial.print("Code valide en position: ");
```

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

```
        Serial.println(ret);
        majcode(ret);
      } else {
        Serial.println("Code non trouve");
      }
    }
}
```

Si des données sont disponibles sur la liaison série, on appelle la méthode **parseInt()** afin d'obtenir immédiatement un entier (**int**) et non une chaîne de caractères. Cette méthode retourne 0 à la fois si « 0 » est reçu ou si une chaîne ne se traduit pas en nombre. Voilà pourquoi nous utilisons **random()** avec une valeur minimum de 1, il ne serait pas possible sinon de faire la différence entre un code **0** et une entrée **"coucou"** sur le moniteur série par exemple. Si l'entier obtenu est zéro, la fonction **loop()** se termine là, nous écartons la saisie.

Il ne nous reste ensuite plus qu'à chercher le code obtenu avec notre **cherchecode()** et agir en fonction du résultat. Si la valeur retournée est

Matériellement, notre montage se résume à un afficheur LCD et un bouton. La partie vitale est la génération, la gestion et les manipulations faites dans les croquis. Il vous sera possible d'appliquer le principe des codes tournants à n'importe quel type d'échanges, que ce soit par les ondes, une saisie manuelle, une transmission infrarouge ou encore audio...

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

INNO ROBO EVENT

16-18 MAI 2017

PARIS FRANCE

WHERE INNOVATION GROWS

Plongez au **cœur** de l'**innovation robotique**

Chercheurs, créateurs et utilisateurs vous dévoilent le monde de demain

UNE APPROCHE HUMAINE DE LA ROBOTIQUE

SMART HOMES

INDUSTRY 4.0 & SUPPLY CHAIN SERVICES

MEDICAL & HEALTH

TECHNOLOGIES & FORESIGHT

SMART CITIES

FIELD ROBOTICS

HOSPITALITY RETAIL & TOURISM

www.**innorobo.**com

theinnorobo

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

positive, alors il s'agit d'une position dans le tableau et le code est valide. Dans ce cas, nous mettons à jour le tableau avec `majcode()` en passant la position du code trouvé. Dans le cas contraire, nous ne touchons pas au tableau et nous rejetons le code avec un message d'erreur.

Pour tester notre réalisation, rien de plus simple : il suffit de programmer le premier croquis dans l'Arduino émetteur et le second dans le récepteur, puis d'ouvrir le moniteur série. Une pression sur le bouton nous affichera un code que nous pouvons entrer dans le moniteur pour validation. On peut également tester la plage de recherche en appuyant plusieurs fois sur le bouton et en testant le code obtenu.

Bien entendu, si le nombre de valeurs affichées, mais non saisies dépasse la taille du tableau, ou en d'autres termes la fenêtre de validation, les deux montages sont alors désynchronisés et la seule solution est de faire un *reset* de l'un comme de l'autre. Exactement comme avec les télécommandes de verrouillage centralisé des voitures. Notez que le croquis de réception est conçu pour définir la taille de la fenêtre de validation via la valeur de la macro `TAILLE_BUF`. Pour plus de souplesse, il suffit d'augmenter la valeur et le reste du code s'adaptera en conséquence.

4. ALLER PLUS LOIN

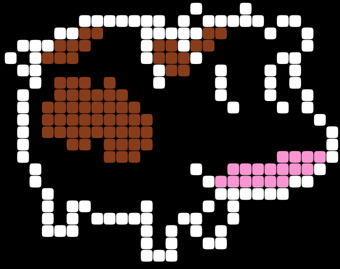
Comme précisé en introduction de l'article, ceci n'est bien entendu absolument pas sécurisé. Non seulement les codes sont visibles, mais surtout, le PRNG est relativement simple. Pire encore, son fonctionnement même (en dehors du modulo) fait qu'il est possible de prédire relativement facilement le prochain code. Il suffit de connaître le croquis pour arriver à mettre le système en péril. Or dans ce domaine, le fait que le fonctionnement du système soit secret n'est pas une sécurité. La sécurité par obscurantisme ne fonctionne jamais...

Pour améliorer le système, il faut non seulement utiliser un meilleur PRNG ou du moins en augmenter l'entropie (65535 valeurs différentes possibles ne sont clairement pas suffisantes), mais surtout il est nécessaire de rendre l'échange illisible. En utilisant des valeurs possibles plus importantes et en chiffrant les échanges, les codes tournants deviennent impossibles à prédire. Mais attention, il faut vraiment savoir ce qu'on fait car, dans la pratique, même des applications commerciales sont souvent mises à mal en raison d'une maladresse de conception ou d'implémentation. La conception de ce genre de système est un vrai métier.

De plus, le fonctionnement général des codes tournants peut être perverti. Dans le cas d'une émission du code par radio comme dans le cas des télécommandes pour véhicules, une certaine forme d'attaque par rejeu reste possible : brouiller le signal de manière à empêcher le récepteur d'obtenir les codes, intercepter un premier message, attendre une seconde tentative, intercepter le second message et là, envoyer le premier message au récepteur. On introduit alors un décalage entre émetteur et récepteur, et le second code capturé peut alors être utilisé ultérieurement.

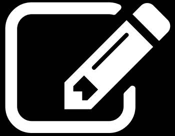
Enfin, vous l'avez sans doute remarqué, ces montages démonstratifs ne supportent pas un *reset*, car tout est stocké en mémoire vive. Il est cependant possible d'envisager une solution en jouant avec le PRNG (non, je ne l'avais pas intégré dans l'article par hasard) et en stockant en EEPROM la valeur de la variable globale utilisée, et/ou les valeurs du tableau, pour ensuite reprendre le processus là où il s'était arrêté avant redémarrage. Ceci peut être un exercice amusant et très enrichissant pour faire évoluer ses croquis... **DB**

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)



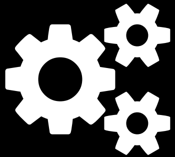
Cowlab

IDE pour l'embarqué



Développez

Editeur de code en ligne: vous accédez à vos projets de partout et à tout moment.



Compilez

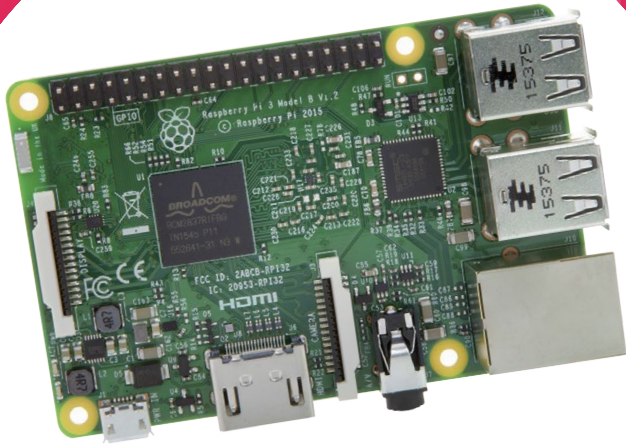
De nombreux compilateurs disponibles pour vos cibles préférées (gcc, yosys, spinalHDL, ...)

Collaborez

Cowlab est en plein développement. Nous sommes à l'écoute de vos avis, souhaits et remarques pour que cet outil fasse demain ce dont vous revez aujourd'hui.

http://www.cowlab.fr

AGILACK



Votre boutique On-line

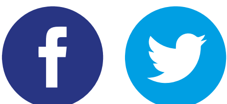
Raspberry Pi

Avec Raspberry Pi, une multitude de fonctionnalités s'offrent à vous ...

Découvrez l'univers du **Raspberry Pi** et plus de **250 accessoires !**

Code KDO : **HACKABLE**

5€ pour 100€ d'achat



www.kubii.fr

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)

Ce document est la propriété exclusive de Alex Arnaud(balinuxdroid@gmail.com)