N°9 | NOV. - DÉC. 2015

HACKABLE MAGAZINE

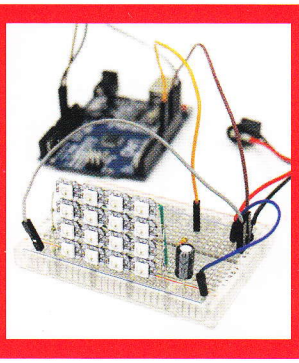
DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

France METRO : 7,90 € – CH : 13 CHF – BEL/PORT.CONT : 8,90 € – DOM TOM : 8,50 € – CAN : 14 $ cad – TUNISIE : 18 TND – MAR : 100 MAD

LEDS

Simulez un téléviseur en votre absence pour dissuader les cambrioleurs

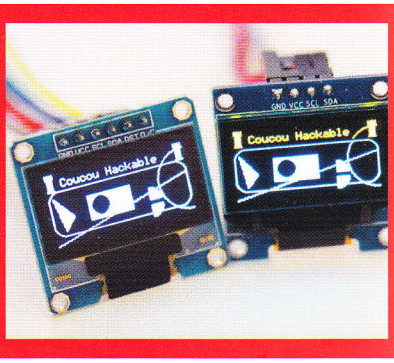
p. 20



AFFICHEURS

Utilisez des mini-écrans OLED SPI ou i2c pour vos projets Arduino

p. 04



CONFIGURATION

Connectez et gérez deux écrans SPI sur Raspberry Pi grâce au Device Tree

p. 76

RADIO

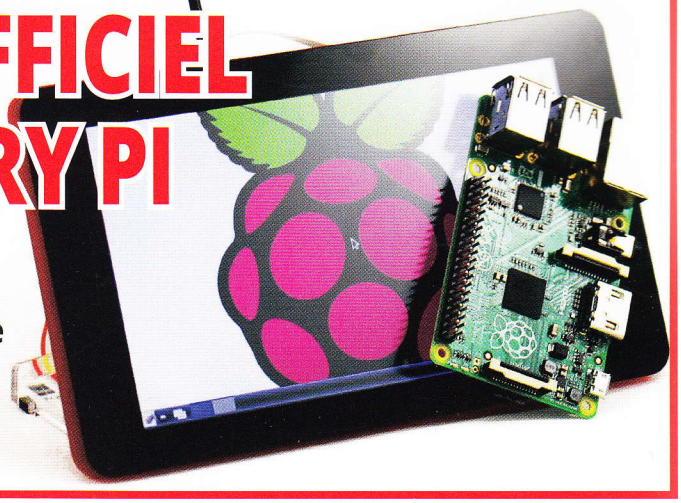
Tracez une « carte » des ondes avec votre récepteur SDR/DVB-T et RTL-Power

p. 88

Enfin une solution d'affichage 7 pouces de qualité pour la Pi ! p. 62

ÉCRAN OFFICIEL RASPBERRY PI

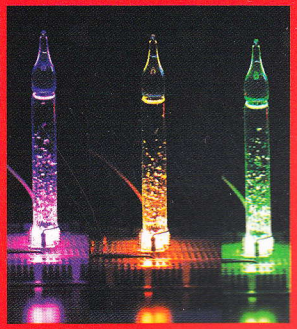
- Tour d'horizon du matériel et connexion
- Configuration et test de l'écran tactile
- Comparaison avec une solution 7 pouces HDMI



DÉCORATION

Les Bubble lights ou comment faire bouillir un solvant avec des résistances

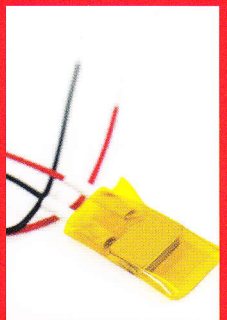
p. 40



CONTRÔLE

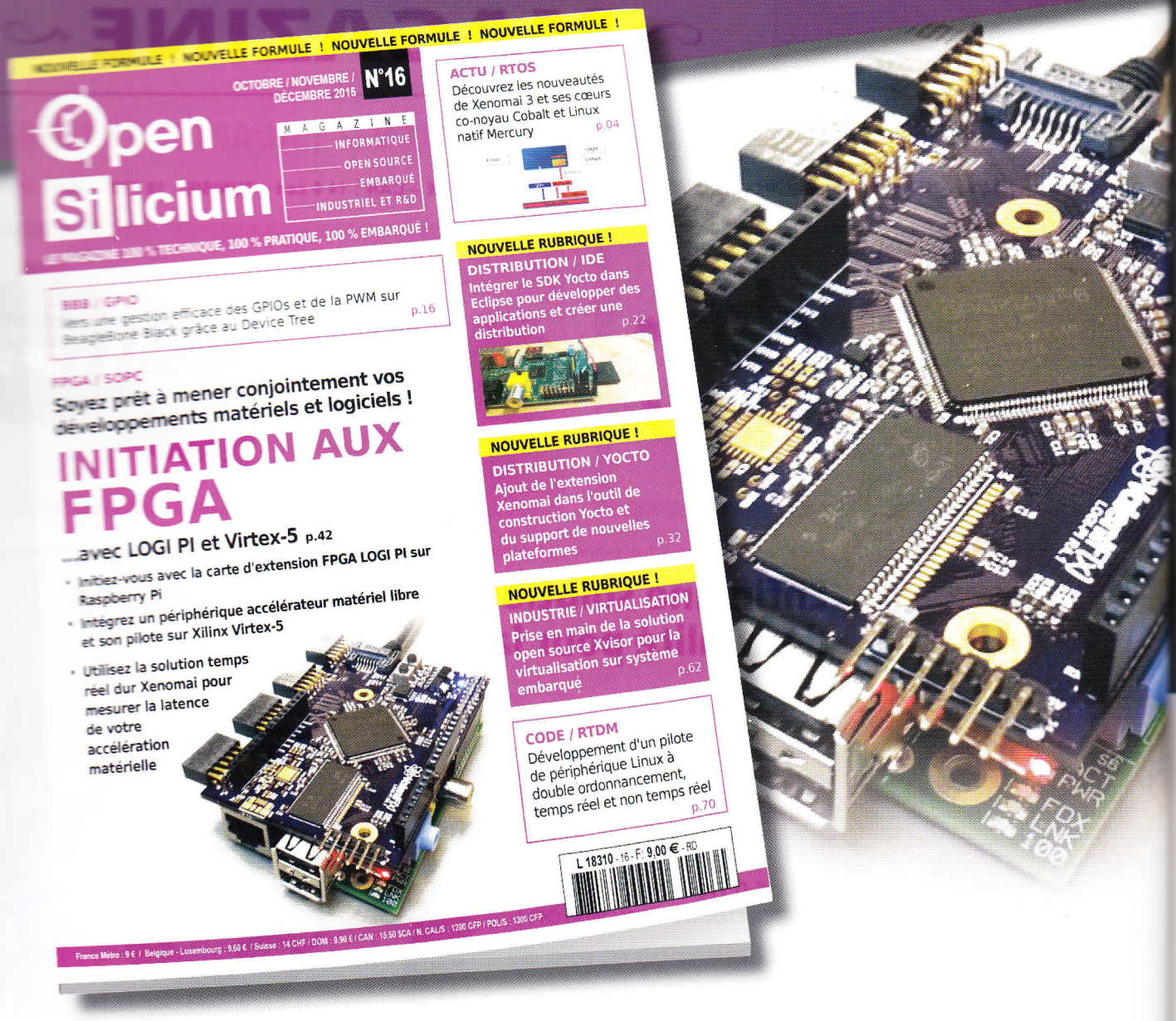
Réglez précisément et efficacement un système chauffant grâce à la régulation PID

p. 56



L 19338 - 9 - F: 7,90 € - RD

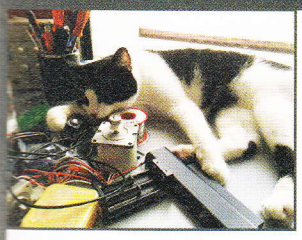
NE MANQUEZ PAS
OPEN SILICIUM N°16 !



SOYEZ PRÊT À MENER CONJOINTEMENT VOS DÉVELOPPEMENTS MATÉRIELS ET LOGICIELS !

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

ÉDITO



Bienvenue dans ce numéro 9 couvrant la période de novembre et décembre.

Pourquoi préciser cette période de parution ? Tout simplement parce que « fin d'année » rime avec « fêtes de fin d'année » et donc plus précisément avec « Noël ». Je ne suis personnellement pas particulièrement sensible à ce genre de festivité (en dehors des marchés de Noël bien de chez nous et de ce qu'on peut y boire et manger), mais Noël est souvent une excuse pour se faire plaisir et faire plaisir. Et par la même occasion, trop souvent se torturer l'esprit pour trouver la réponse à l'annuelle question « qu'est-ce que je vais faire comme cadeau ? ».

Quoi de mieux alors que d'offrir tout un univers en guise de présent plutôt qu'un gros ballotin de pralinés soi-disant belges ? Je parle bien entendu d'une carte populaire de type Arduino ou Raspberry Pi, qui finalement ne vous coûtera pas plus cher, mais ouvrira bien des perspectives à celui ou celle qui la recevra. Et quoi de plus normal après tout que de donner ou recevoir une carte pour Noël ?

Comment ça vos convives de Réveillon ne sont pas des « geeks » (terme totalement galvaudé, j'en conviens) ? Il n'y a pas de prérequis pour s'initier, pour peu que l'on soit un minimum accompagné. Il suffit d'y voir un intérêt et le reste vient de lui-même. Même si certains et certaines sont peut-être prédisposés en raison d'une inexplicable attirance mystique envers le domaine (voir photo, je vous présente M. Copper, compagnon monochrome présentant à l'évidence une telle attirance), pour la plupart il suffit d'un élément déclencheur pour transformer une poussée de curiosité en passion.

Bien entendu, les premiers bénéficiaires naturels qui viennent à l'esprit pour ce qui est de cadeaux, sont les enfants (ou ados, ou pré-ados, ou post-ados, ou pré-post-ados (?), etc.). Mais il suffit de parcourir un catalogue de jouets pour crouler sous les Tamagotchis, les panoplies d'espion, les kits d'initiation à l'électronique, les jouets avec trois ridicules leds qui clignotent, etc. Pourquoi ne pas leur faire découvrir « le vrai truc » et leur donner l'opportunité de créer leurs propres jouets, bien plus complets et riches que ceux qui, finalement, ne sont que des solutions « clé en main ».

En y réfléchissant un instant, la véritable question est : « pourquoi les (vrais) kits électroniques et les cartes type Arduino ne sont pas présents dans ces magasins et les rayons jouets des grandes surfaces ? ». C'est pourtant là que se trouvent, par définition, les cadeaux de Noël de la future génération d'électroniciens, d'ingénieurs et de programmeurs... Une passion qui se révèle après l'ouverture un paquet sous le sapin est, sans l'ombre d'un doute, le plus beau des cadeaux pour toute une vie.

Sur ces quelques mots, je vous souhaite d'excellentes choses de fin d'année et vous donne rendez-vous pour le prochain numéro en 2016.

Denis Bodor

Hackable Magazine

est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

10, Place de la Cathédrale - 68000 Colmar
Tél. : 03 67 10 00 20 – Fax : 03 67 10 00 21
E-mail : lecteurs@hackable.fr
Service commercial : cial@ed-diamond.com
Sites : www.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 €

MIXTE
Papier issu de sources responsables
FSC® C015136

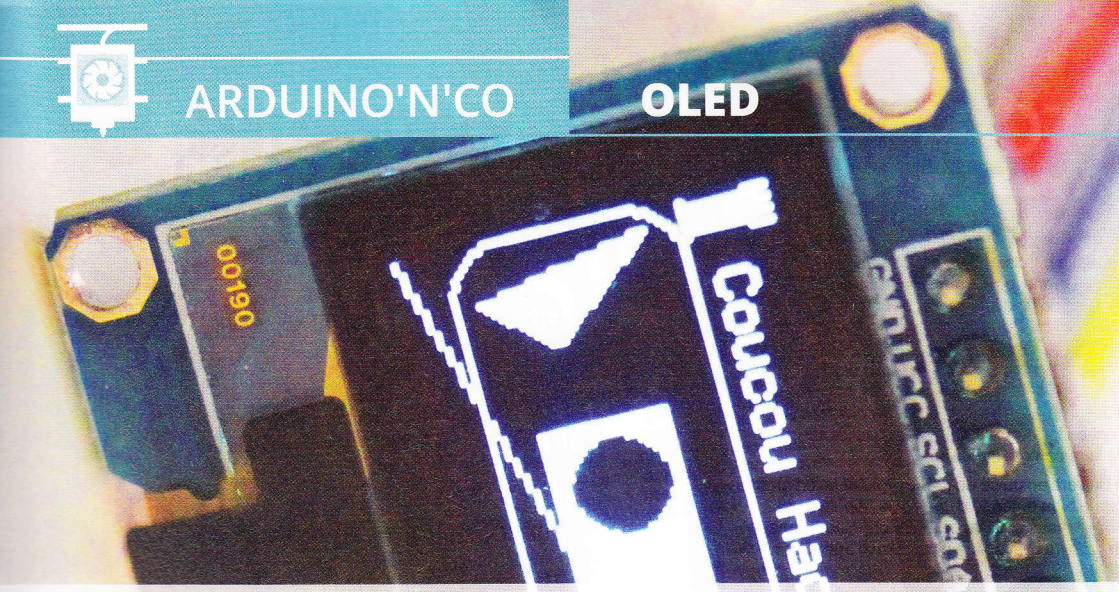
Suivez-nous sur Twitter
@hackablemag

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.

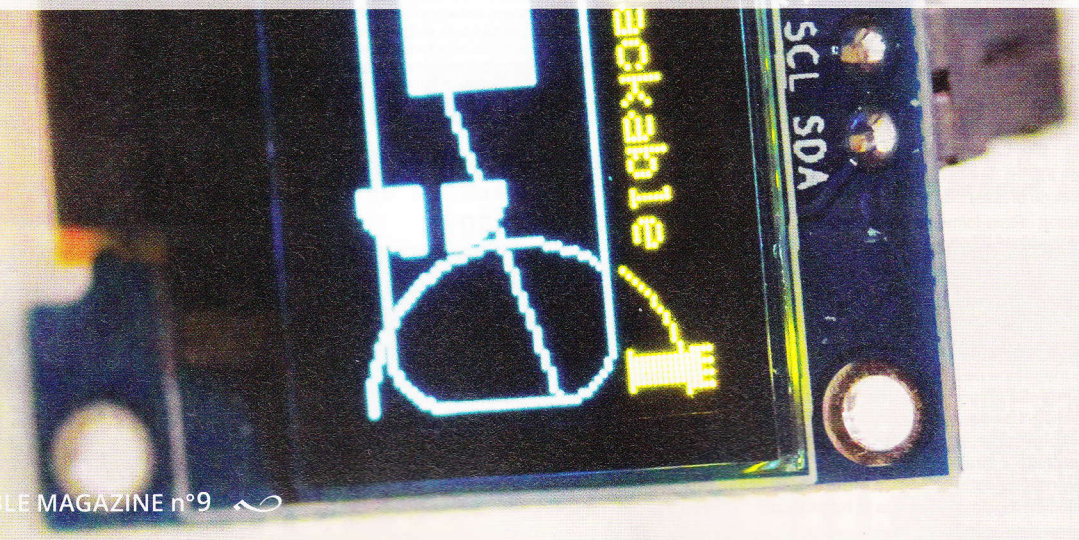
SOMMAIRE



UN ÉCRAN OLED MINIATURE POUR VOS PROJETS

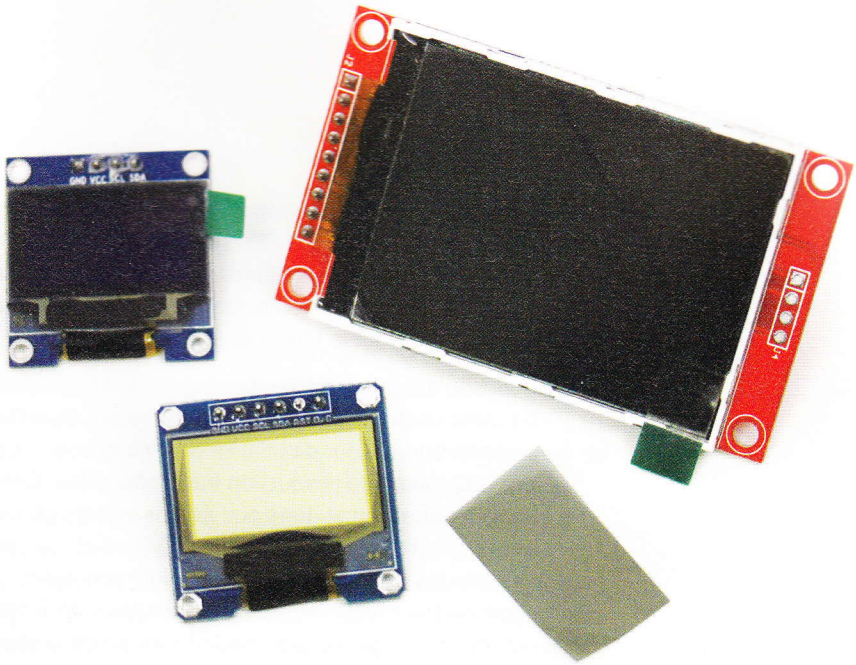
Denis Bodor

Nous avons déjà fait connaissance dans ces pages avec des afficheurs LCD texte ainsi que les écrans TFT graphiques couleur ILI9340 en SPI. L'un et l'autre type d'écran se prêtent très bien à la plupart des projets, mais parfois le cahier des charges est particulier : on souhaite un écran vraiment petit, avec une forte luminosité et un excellent contraste. Dans ce cas, la solution tient en un mot : OLED !



Vous connaissez sans l'ombre d'un doute les leds, ces petits composants semi-conducteurs adorables générant de la lumière ou un rayonnement infrarouge ou UV. Vous connaissez également les écrans à cristaux liquides, ou LCD (pour *Liquid Crystal Display*). Les deux technologies ont une longue histoire derrière elles et se sont bonifiées au fil du temps. Les leds affichent ainsi un rendement qui ne cesse de s'améliorer et offrent aujourd'hui des intensités lumineuses remarquables. La technologie LCD actuelle permet des contrastes plus importants que par le passé et une densité d'affichage importante tout en corrigeant les problèmes d'il y a quelques années (comme la stabilité du comportement en fonction des températures).

Cependant, depuis quelques années, le monde de l'affichage a subi une petite révolution avec l'arrivée des leds organiques. Il s'agit également de semi-conducteurs, mais reposant sur un composé organique (base carbone) et non plus sur un élément nativement semi-conducteur comme le silicium dopé, maintenant appelé semi-conducteur inorganique. Les caractéristiques de ces nouveaux composants sont similaires à celles des semi-conducteurs inorganiques, mais présentent un certain nombre d'avantages lorsqu'il est question de leds : plus légers, moins fragiles, plus faciles à produire, moins chers...

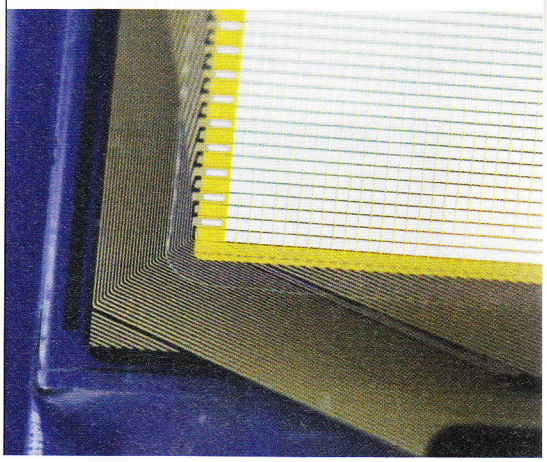


Trois écrans utilisables facilement avec les cartes Arduino (ou Raspberry Pi). En haut à droite (en rouge), l'écran LCD TFT 320×240 en SPI dont nous avons déjà parlé à plusieurs reprises dans le magazine, à titre de comparaison avec un écran OLED 0,96" i2c/TWI (à gauche) et un modèle identique en SPI avec son filtre polarisant retiré (sacrifié pour les jolies photos).

Les facilités d'assemblage et de production associées aux leds organiques ont ainsi permis une évolution des systèmes d'affichage. Le fonctionnement d'un écran LCD couleur repose sur un assemblage de couches : un éclairage arrière fournit la lumière, celle-ci passe par un filtre polarisant et une grille de filtres de couleurs (3 par pixel : rouge, vert et bleu), puis passe par les cristaux liquides et enfin un dernier filtre polarisant... En faisant circuler un courant dans les cristaux liquides, ceux-ci dévient la lumière et affectent sa polarité. Il est ainsi possible d'activer ou désactiver un élément rouge, vert ou bleu pour chaque pixel et ainsi composer une image.

Remplacez maintenant ces trois composants de couleur contrôlables par des leds RVB. Le rétro-éclairage devient inutile, car les leds émettent la lumière, le contraste est plus important (les cristaux liquides ne bloquent pas la totalité de la lumière) et la logique est inversée : un écran

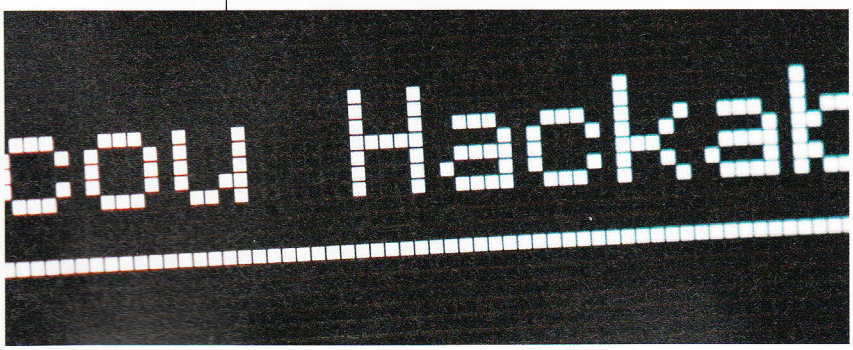
Très gros plan sur la matrice d'affichage. Chacun de ces petits carrés est une led et il y en a 8192 en tout, arrangés en une grille de 128×64. Le contrôle de cet ensemble est l'affaire du SSD1306 avec lequel vous dialoguez par l'intermédiaire d'une bibliothèque.



tout noir est un écran où toutes les leds sont éteintes alors qu'avec un écran LCD il s'agit d'alimenter chaque « pixel » pour bloquer la lumière (certes les cristaux liquides nécessitent moins de courant, mais le rétro-éclairage est tout le temps actif).

Jusqu'à il y a quelques années, il était impensable de créer un écran à leds, car cela revenait à produire un semi-conducteur de la taille de l'écran et soulevait des problèmes de production insurmontables. Cette technique se limitait donc aux écrans géants où les pixels sont littéralement des leds RVB soudées sur des circuits formant des panneaux qui sont alors combinés et pilotés individuellement pour créer l'écran et former l'image (écrans publicitaires ou de stades). Mais la technologie des semi-conducteurs organiques a changé tout cela, il est maintenant possible de produire « facilement » un panneau de quelques pouces ou plus précisément une matrice active de leds organiques, *Active-Matrix Organic Light-Emitting Diode* en anglais, ou plus simplement AMOLED. Ce type de technologie est maintenant courante sur smartphone et fait également son apparition sur les grandes tablettes comme la Samsung Galaxy Tab S 10.5 dont l'écran est littéralement composé de 2560x1600x3 leds, soit quelques 12 millions de leds !

La technologie est prometteuse, car les semi-conducteurs organiques peuvent être des polymères offrant encore d'autres avantages comme le fait d'obtenir des écrans souples. Quelques démonstrations technologiques ont déjà été faites dans ce sens, sans pour autant avoir atteint, pour l'instant, le stade de la fabrication en masse ou de l'intégration dans un produit fini. Nul doute cependant que les écrans OLED « enroulables » ne sont pas loin...



Zoom sur une partie de l'écran OLED blanc. L'affichage est net, clair et parfaitement homogène. Le tout avec un angle de vision presque impossible à obtenir avec un affichage à cristaux liquides et définitivement inaccessible aux écrans d'entrée de gamme.

1. ET NOTRE MONDE DE L'ÉLECTRONIQUE ALORS ?

Les écrans OLED sont encore relativement coûteux qu'il s'agisse de pièces de remplacement pour smartphone ou de systèmes d'affichage complets. Il existe des modules proposant des écrans couleurs de quelques pouces en OLED (chez Liquidware par exemple), mais à un prix prohibitif (~250€ pour un 4,3"). Rien d'équivalent à l'écran LCD dont nous avons déjà parlé dans ce magazine. Attention, certains vendeurs n'hésitent pas à glisser un « OLED » ou un « AMOLED » dans leurs titres d'annonces eBay alors qu'il s'agit en réalité d'écrans LCD TFT tout à fait standards. Un écran 4 pouces OLED couleur pour 20€ ça n'existe pas (du moins pour l'instant) !

À notre portée cependant, nous avons une classe plus modeste d'écrans : affichage monochrome, résolution de 128*64 ou 128*32, taille réduite de 0,96" (~2,5cm) ou 1,3" (~3,4cm)... Ce type d'afficheurs est initialement prévu pour être intégré dans des baladeurs MP3 d'entrée de gamme ou encore des téléphones mobiles économiques. Ils présentent une particularité intéressante puisqu'ils intègrent un contrôleur directement dans l'écran, un SSD1306. Bien entendu, les chaînes de production chinoises n'ont pas traîné et ont rapidement proposé ces afficheurs sous forme de

modules offrant une connectique plus facile à utiliser qu'un simple flat-flex (connecteur souple).

On retrouve donc ce type d'afficheurs OLED un peu partout, à commencer par eBay, pour un prix situé entre 5 et 7 euros. Notez au passage que ces modules sont des clones de ceux vendus par Adafruit à $19,50 (0.96") et $24,50 (1,3") et qu'ils présentent tantôt des petits défauts, en particulier au niveau du brochage.

2. À PROPOS DES CLONES ET DU BROCHAGE

Si vous souhaitez opter pour la sécurité et faciliter la mise en œuvre, moyennant finances, le plus simple est bien entendu de vous tourner vers Adafruit ou un de ses distributeurs comme Mouser. Si en revanche un peu de bidouille et de risque ne vous font pas peur, vous pouvez choisir la « loterie » eBay et les vendeurs chinois avec des modules à quelques euros (ports gratuits). C'est précisément ce que j'ai fait et, à part un peu de patience, d'énervement et d'expérimentations à tâtons, cela s'est relativement bien passé.

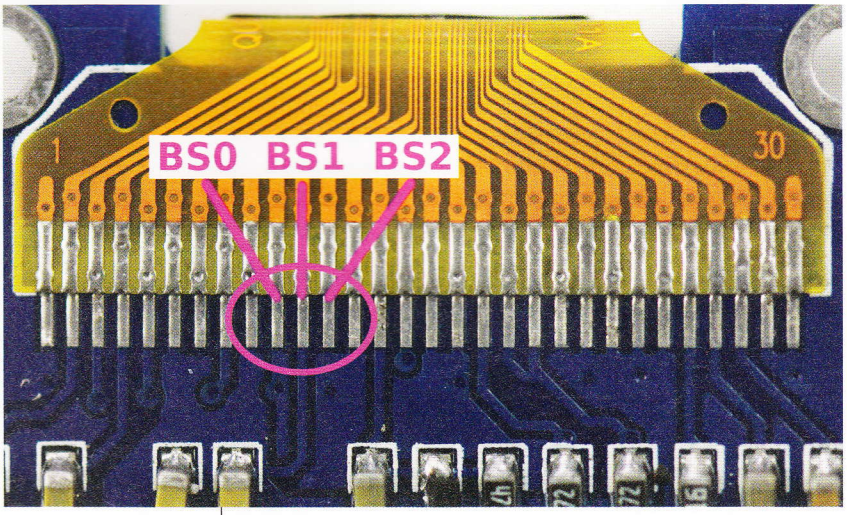
Les annonces de ventes aux enchères, en particulier sur ce matériel, sont souvent assez sommaires et bourrées d'informations contradictoires. Les modules d'affichage OLED contrôlés par un SSD1306 peuvent être interfacés de plusieurs façons parmi lesquelles, la liaison parallèle, SPI 3 fils, SPI et i2c (alias TWI pour *Two Wire Interface*). Ce sont ces deux derniers modes qui sont le plus souvent utilisés pour les modules, et aussi ceux évoqués n'importe comment dans les titres et les descriptions des annonces.

Le SSD1306 est configuré en fonction de l'état haut (VCC) ou bas (masse) de trois de ses broches : BS0, BS1 et BS2. Pour l'i2c, nous avons respectivement 0,1 et 0, et pour SPI, 0, 0 et 0. Ces connexions font généralement partie intégrante du circuit imprimé du module et ne sont que rarement configurables (sauf à jouer du fer à souder et du scalpel pour couper des pistes). Les modules chinois sont donc obligatoirement soit SPI, soit i2c. La mention des deux protocoles dans le descriptif n'est justifiée qu'en rapport avec le SSD1306 et non vis-à-vis du produit lui-même. Ceci ne veut pas obligatoirement dire que le module supporte les deux bus.

J'ai commandé plusieurs modèles 0,96" chinois entre 5 et 8 euros et la conclusion est la suivante : vous ne savez pas réellement ce que vous recevez. Tous les modules réceptionnés ont finalement fonctionné, mais certains étaient en SPI alors qu'annoncés en i2c, d'autres étaient en SPI comme décrit dans l'annonce, mais avaient des broches libellées avec une nomenclature i2c et d'autres enfin, avaient des signaux manquants (CS ou RST).

Au final, tous ces modules semblent avoir en commun l'afficheur lui-même composé du contrôleur SSD1306 embarqué dans la matrice de leds. Il s'agit d'un afficheur fabriqué par une société appelée *EastRising Technology* dont la documentation technique (*datasheet*) est disponible sur le site HobbyTronics (**http://www.hobbytronics.co.uk/datasheets/ht/ER-OLED0.96.pdf**). Cet afficheur se décline en trois versions de même taille, même résolution et même contrôleur :

- ER-OLED0.96-3W : affichage entièrement blanc ;
- ER-OLED0.96-2B : affichage entièrement bleu (plus lumineux) ;
- ER-OLED0.96-1YB : affichage jaune et bleu. L'écran n'est pas bicolore dans le sens où on peut instinctivement l'entendre, mais est divisé en une zone supérieure jaune de 128*16 pixels et une zone bleue inférieure de 128*48 pixels. Les deux zones sont séparées physiquement d'une distance équivalente à un pixel, mais qui ne fait pas partie de la matrice d'affichage qui reste donc de 64 pixels de haut. Ce genre de division bichromique peut être très intéressante soit pour afficher des données différentes soit, par rotation à 90°, pour fournir une



En cas de doute sur le protocole à utiliser pour se connecter au module, la méthode la plus simple, pour ce type d'afficheur, est de vérifier les connexions des broches 10, 11 et 12 correspondant à BS0, BS1 et BS2. Ici l'écran est i2c, avec masse/VCC/masse.

« marge » colorée pour préfixer ou suffixer des lignes de texte (comme pour un curseur sur le côté d'un menu par exemple) ou encore accompagner une jauge d'une icône explicite.

En quoi le fait de connaître le fabricant précis de l'afficheur est important ? C'est tout simple, parmi les 30 broches du connecteur en plastique souple (flat-flex) on retrouve les lignes BS0 à BS2 (broches 10 à 12) qui nous permettent alors de précisément savoir quelle interface est utilisée pour communiquer. C'est grâce à cette information que mes afficheurs OLED blancs possédant des broches SCL et SDA typiques d'une liaison i2c se sont avérés être interfacés en SPI (BS0 à BS2 à la masse). Il faut savoir, en effet, que les lignes de données (D0 à D7 en mode parallèle, broches 18 à 25) sont partiellement utilisées en SPI et en i2c :

- SPI : D0 correspond à SCLK et D1 à MOSI ;
- i2c/TWI : D0 est SCL, D1 et D2 sont reliés entre eux et forment SDA.

L'alimentation des modules est également un problème puisque souvent aucune information n'est fournie (ou suffisamment crédible). Du point de vue de l'afficheur, plusieurs tensions sont à l'œuvre :

- VDD/VSS : alimentation et masse pour le circuit logique (le SSD1306 lui-même), 3,3V max (on parle ici des maximums recommandés, non des maximums absolus) ;
- VCC : tension d'alimentation des leds, 9,5V max ;
- VLSS : la masse pour les circuits analogiques (convertisseur de tension) ;
- VBAT : tension d'alimentation du convertisseur de tension (*DC/DC converter*), 4,2V max.

La documentation présente deux cas de figure. Le premier consiste à alimenter l'écran avec d'un côté VCC pour les leds et de l'autre VDD pour le SSD1306. Généralement ce type de montage n'est pas présent sur les modules (mais on ne sait jamais). La seconde option repose sur le convertisseur de tension intégré dans le SSD1306 et reposant sur des condensateurs externes permettant d'augmenter la tension pour l'alimentation des leds (convertisseur à transfert de charges). Il est donc de bon ton de jeter un œil à l'arrière du module pour vérifier la présence des condensateurs de 1µF entre les broches 4/5 et 2/3. Si c'est le cas, il est fort probable que la tension pour les leds soit générée en interne (attention à ne pas confondre avec les condensateurs de filtrage entre VLSS/masse et VCC/VDD).

Mais ce n'est pas tout. Les fabricants de modules peuvent également ajouter leur grain de sel en intégrant un régulateur de tension sur le circuit. Il faut donc se montrer très prudent et surtout ne pas avoir peur de prendre le risque de griller le module si les informations dans le descriptif de l'objet ne sont pas justes (ou prendre le temps d'analyser le circuit du module et mesurer chaque résistance). Pour tout dire, tous les modules mis en vente sur eBay que j'ai consultés affichent joyeusement « *3V ~ 5V DC* » ou des choses horribles comme « *Soutien large de tension : sans*

aucune modification, soutient directement 3V ~ 5V DC ». Tous ceux réceptionnés ont également parfaitement fonctionné avec une carte Arduino Uno utilisant donc des niveaux de tension 0/5V.

Quoi qu'il en soit, pour faire simple, vous n'avez que deux options : croire le descriptif ou analyser le circuit. Dans le premier cas, vous risquez de perdre 5€ à 7€ et dans le second, 30 à 60 minutes ou bien plus... Réjouissez-vous, en plus d'un module, vous avez le frisson et l'aventure, merci Shenzhen ! Si vous n'êtes pas amateur de telles sensations, vous pouvez également acheter un module à $20 bien documenté et sérigraphié.

Terminons cette partie effrayante et mystérieuse en parlant des broches manquantes. Parmi les modules réceptionnés, deux modèles possédaient un brochage « insuffisant » :

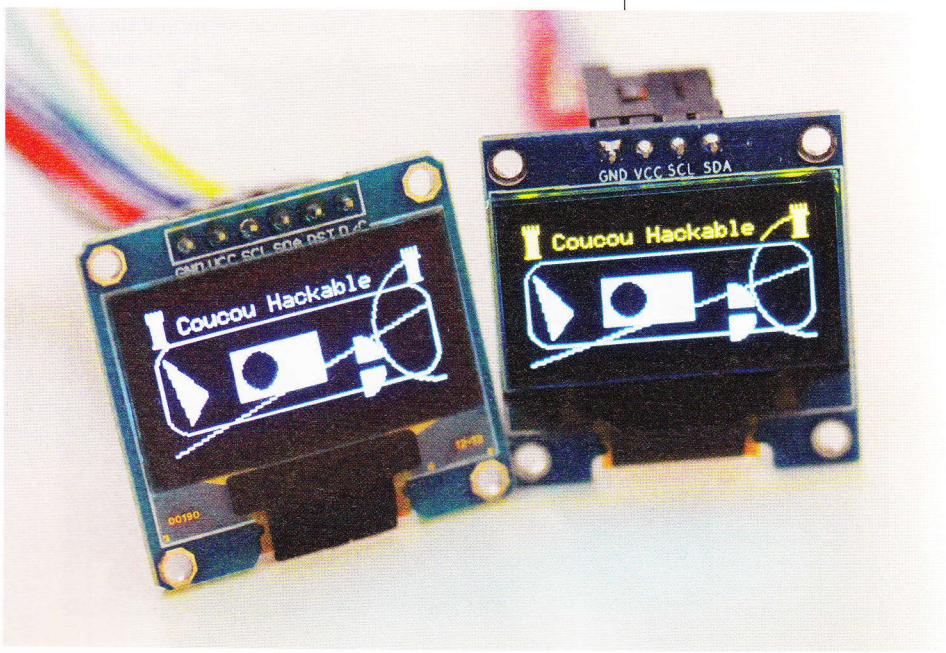
- Le module SPI libellé avec « SCL » / « SDA » n'avait pas de ligne CS. Rappelons que ce signal permet d'asservir un périphérique sur un bus SPI. On met la ligne CS du périphérique auquel on veut parler à la masse pour « le faire écouter ». En l'absence d'accès à ce signal, connecté « en dur » à la masse sur le module, le périphérique pense que c'est toujours à lui qu'on parle. En d'autres termes, il n'est pas possible de l'utiliser sur un bus SPI avec d'autres périphériques.
- Un module interfacé en i2c ne présente que VCC, GND, SCL et SDA. Il n'y a pas de ligne RST permettant la réinitialisation du contrôleur. Ceci ne semble pas poser problème avec les deux bibliothèques Arduino les plus courantes (Adafruit SSD1306 + Adafruit_GFX et U8glib) pour ce type d'utilisation, mais la bibliothèque Adafruit utilise une sortie pour instancier l'objet qui représente l'afficheur. En d'autres termes, une sortie sera utilisée pour rien...

3. UTILISATION AVEC UNE CARTE ARDUINO

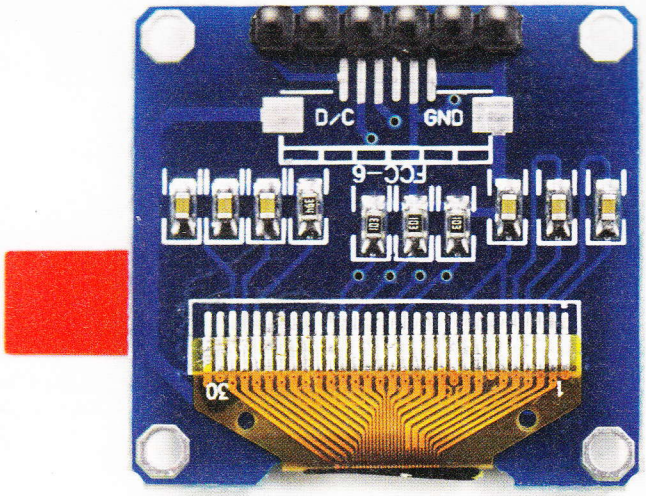
Une fois qu'on est est peu près sûr du type du protocole utilisé, on peut s'attacher à la connexion du module d'affichage à une carte Arduino.

Dans le cas du SPI, en principe, nous avons sept connexions :

- VCC : tension d'alimentation ;
- GND : masse ;
- MOSI : données en provenance sur maître (Arduino) et à destination de l'esclave (module) ;
- SCLK : signal d'horloge pour cadencer la communication ;

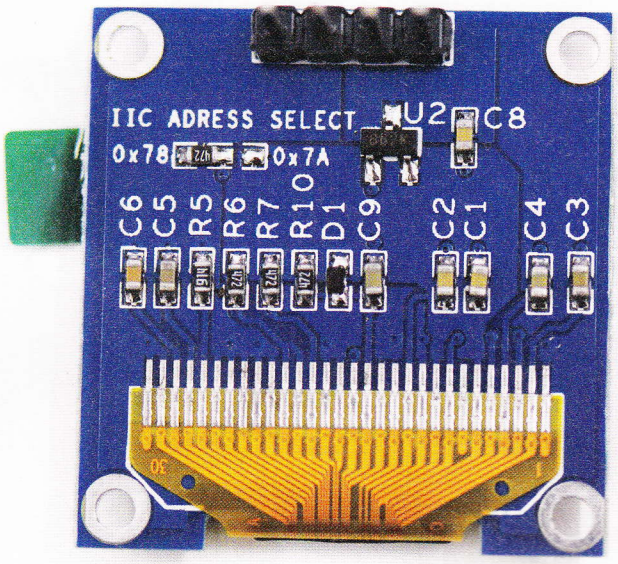


Deux des quelques modèles réceptionnés et mis en œuvre pour la réalisation de cet article. Notez la non-uniformité de l'image sur l'écran de droite. Ceci n'est pas visible à l'œil nu, mais un simple artefact parasite dû à la fréquence de rafraîchissement et à l'appareil photo.



L'écran OLED interfacé en SPI est indubitablement de qualité moindre que celle de son comparse i2c. La soudure du connecteur flat-flex est brouillonne, le tracé du circuit clairement auto-routé et les connecteurs sont mal référencés (SDA et SCL en lieu et place de MISO et SCLK).

À l'arrière de l'écran i2c on trouve un ensemble de composants passifs, ce qui semble être clairement un régulateur de tension, et une résistance permettant le choix de l'adresse sur le bus.



- RST : reset permettant de réinitialiser le SSD1306 ;
- D/C : signal permettant de préciser si le maître envoie des données ou des commandes (vaguement équivalent au RS des HD44780), cette broche est tantôt appelée A0 ;
- CS : *chip select* permettant d'activer l'esclave pour lui envoyer des ordres ou des données.

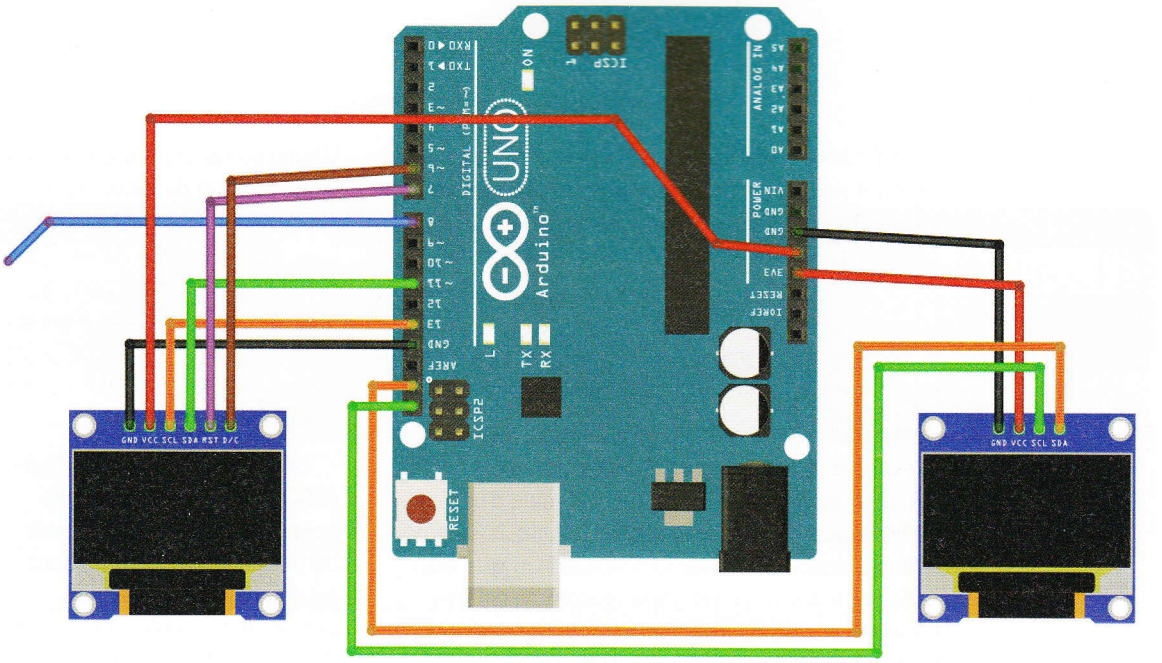
Avec l'afficheur mal libellé, CS est simplement absent, « SDA » correspond à MOSI et « SCL » à SCLK.

Avec la version i2c/TWI, normalement nous avons 5 signaux :

- VCC : tension d'alimentation ;
- GND : masse ;
- SCL : signal d'horloge pour cadencer la communication ;
- SDA : données pour la communication bidirectionnelle ;
- RST : reset (absent sur les modules testés).

Dans un cas comme dans l'autre, la connexion à une carte Arduino est directe et ne nécessite pas d'autres composants :

- SDA et SCL sont directement référencés sur une Arduino UNO par exemple, mais les lignes sont communes avec A5 et A4 (les entrées analogiques). Il ne vous est donc pas possible d'utiliser le bus i2c et ces broches en même temps ;
- toujours sur UNO, les signaux SCLK et MOSI correspondent respectivement aux broches 13 et 11. Les deux autres signaux du bus SPI, MISO et SS sont sur 12 et 10. Notez que SS (alias CS) est utilisé lorsque la carte Arduino est esclave sur le bus SPI et non-maître. Dans ce cas, nous pouvons choisir la broche qui nous chante pour la ligne CS du module (quand elle est disponible).



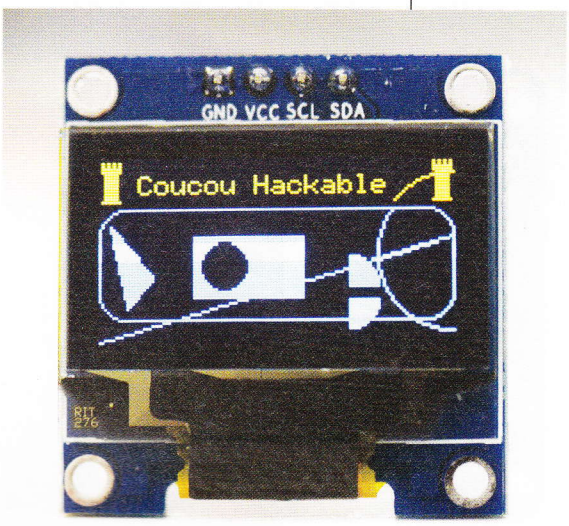
Connexion des modules en SPI et i2c/TWI sur une carte Arduino UNO. Notez que l'afficheur connecté en SPI est ici présenté exactement comme réceptionné, avec une sérigraphie des broches fausse (SDA correspond au signal MOSI et SCL à SCLK>) et la broche CS absente. Le câble bleu connecté à la broche 8 de l'Arduino illustre le fait de devoir utiliser une sortie pour rien dans le croquis.

Que faire lorsque des signaux manquent à l'appel, me demanderez-vous. Il suffit de leur attribuer des sorties comme si la connexion était possible. Ainsi, les bibliothèques d'Adafruit ou U8glib attendent une broche pour CS en SPI, il suffit d'en préciser une. Il en va de même pour la connexion i2c avec la ligne RST, dans le cas de la bibliothèque d'Adafruit (U8glib n'utilise pas RST pour le SSD1306).

4. LES BIBLIOTHÈQUES À UTILISER

Côté support logiciel, trois solutions s'offrent à vous. Vous avez tout d'abord la piste noire consistant, sur la base de la communication SPI ou i2c, à envoyer des ordres au SSD1306 après avoir patiemment épluché, étudié et assimilé sa documentation. Cette méthode universelle est applicable à n'importe quel composant ou module attaché à une carte Arduino, mais c'est aussi la plus difficile et la plus longue.

L'alternative consiste à reposer sur ce même travail, mais fait par quelqu'un d'autre qui aura très sympathiquement écrit une bibliothèque. En ce qui concerne

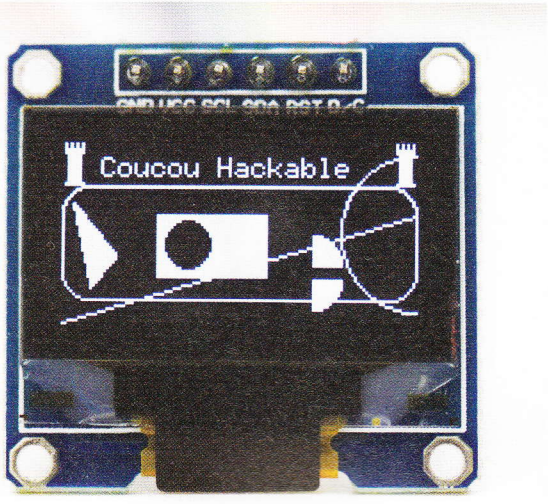


Notre croquis exemple à l'œuvre sur l'afficheur jaune/bleu en i2c, piloté par la bibliothèque U8glib. Ici encore, la barre plus claire au centre de l'écran n'est qu'un problème de prise de vue.

le SSD1306, il en existe deux. L'une, proposée par Adafruit pour leurs propres matériels est directement accessible via le gestionnaire de bibliothèques présent dans l'environnement de développement Arduino, menu ***Croquis***, ***Include Library*** et ***Manage Libraries*** (chercher « SSD1306 »). Il vous faudra également installer la bibliothèque GFX d'Adafruit fournissant les primitives graphiques.

L'autre bibliothèque, installable de la même façon, est U8glib. Il ne s'agit pas d'un support direct pour les afficheurs à base de SSD1306, mais d'une bibliothèque graphique universelle. U8glib n'est donc ni une solution **que** pour le SSD1306, ni **que** pour Arduino, mais un projet générique couvrant une bonne trentaine de contrôleurs (SSD1306, ST7920, HT1632 (matrice de leds), KS0108, PCF8812, etc.). La liste complète des afficheurs supportés est disponible sur **https://code.google.com/p/u8glib/wiki/device**). Il en va de même pour les plateformes supportées et, même si le « 8 » de U8glib signifie initialement « 8 bit », celles-ci vont des Arduino aux ARM Cortex-M en passant par la programmation AVR, Microchip PIC32, etc.

Un certain nombre de différences existent entre les bibliothèques Adafruit et U8glib en termes de fonctionnalités, de compacité du code, de rapidité, de choix techniques et de philosophie de développement. À mon sens, U8glib se rapproche davantage d'une logique de bibliothèque graphique (type SDL) alors que la bibliothèque Adafruit est plus accessible pour obtenir rapidement un résultat avec un mécanisme proche de ce qu'on trouve pour le pilotage d'écrans LCD texte (HD44780).



Le même exemple utilisant U8glib, mais cette fois sur un écran OLED blanc interfacé en SPI. Pour vous composer une interface, vous devrez utiliser de simples primitives graphiques (lignes, cercle, boites, bitmaps, texte, disques, triangle, etc.).

Les bibliothèques AdaFruit (SSD1306 + GFX) et U8glib ne se limitent pas au contrôle du matériel, mais fournissent également des fonctions permettant le dessin et la gestion du texte. Ceci vous permet d'aller plus loin que la simple activation/désactivation des pixels en manipulant directement des fonctions avancées de tracé : rectangles, cercles, disques, lignes, triangle, etc. Ce à quoi s'ajoutent des fonctions de rendu de texte. En effet, l'afficheur lui-même n'a aucune notion de caractère ou de texte. C'est à la bibliothèque de gérer la conversion et la transformation en pixels. Sur ce point particulier, U8glib offre nettement plus de fonctionnalités avec une belle gamme de polices de caractères très complètes déclinées en plusieurs tailles.

Les deux solutions sont donc en mesure de contrôler les afficheurs qu'ils soient connectés en SPI ou en i2c et dans les grandes lignes permettent les mêmes opérations. Le choix est majoritairement une affaire de préférences personnelles. Voyons donc deux exemples commentés proposant (presque) le même résultat, mais avec l'une et l'autre bibliothèque.

4.1 U8glib

Fichier Édition Croquis Outils Aide

```
#include "U8glib.h"

// Déclaration pour SPI matériel
// SCK sur 13 et MOSI sur 11
// arguments : CS, D/C, RST
//U8GLIB_SSD1306_ADAFRUIT_128X64 u8g(8, 6, 7);

// Déclaration pour I2C / TWI matériel
U8GLIB_SSD1306_128X64 u8g(
  U8G_I2C_OPT_DEV_0|
  U8G_I2C_OPT_NO_ACK|
  U8G_I2C_OPT_FAST);

// Déclaration d'une image bitmap en flash
// Notez le U8G_PROGMEM, un PROGMEM propre à u8glib
// La macro B pour spécifier des valeurs en binaire
// est ici très utile
const uint8_t tour_bitmap[] U8G_PROGMEM = {
  B00000000,
  B01010101,
  B01010101,
  B01111111,
  B01111111,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B00111110,
  B01111111
};

// notre fonction de dessin
// contenant les instructions de tracé
void draw(void) {
  // on place l'icône bitmap
  // arg : x, y, nombre d'octets pour une ligne, nombre de lignes
  // Oui, on est coincé avec des multiples de 8 pixels en largeur
  u8g.drawBitmapP(0, 0, 1, 16, tour_bitmap);
  u8g.drawBitmapP(119, 0, 1, 16, tour_bitmap);
```

```
  // un cadre à coins arrondis
  // arg : x, y, largeur, hauteur, rayon coins
  u8g.drawRFrame(0, 16, 128, 40, 10);

  // un rectangle plein
  // arg : x, y, largeur, hauteur
  u8g.drawBox(34, 26, 40, 22);

  // un disque (cercle plein)
  // arg : centre x, cente y, rayon, segment à afficher
  // Les modificateurs de segment possibles sont :
  // U8G_DRAW_UPPER_RIGHT : haut/droite
  // U8G_DRAW_UPPER_LEFT : haut/gauche
  // U8G_DRAW_LOWER_LEFT : bas/gauche
  // U8G_DRAW_LOWER_RIGHT : bas/droite
  // U8G_DRAW_ALL : tout, par défaut en l'absence de modificateur
  // ici haut/droite :
  u8g.drawDisc(90, 43, 10, U8G_DRAW_UPPER_RIGHT);
  // idem, mais bas/droite :
  u8g.drawDisc(90, 48, 10, U8G_DRAW_LOWER_RIGHT);

  // une ligne
  // arg : départ x, départ y, fin x, fin y
  u8g.drawLine(0, 63, 127, 26);

  // un triangle plein
  // arg : x et y de chaque sommet
  u8g.drawTriangle(3,20, 10,52, 20,40);

  // un cercle (vide)
  // arg : centre x, centre y, rayon
  // les modificateurs de segment sont identiques à drawDisc
  u8g.drawCircle(127, 32, 27);

  // un texte
  // On commence par choisir la police
  // (voir la liste dans libraries/U8glib/src/clib/u8g.h)
  u8g.setFont(u8g_font_6x10);
  // le point de placement est relatif au plus haut pixel du texte +1
  // A utiliser *après* le choix de la police
  u8g.setFontPosTop();
  // on dessine le texte
  // arg : x, y, chaîne de caractères (pointeur)
  u8g.drawStr(14, 5, "Coucou Hackable");

  // Nous pouvons aussi dessiner en noir
  // Il faut spécifier la couleur dans l'index
  // Sur écran monochrome 0=fond, 1=avant-plan
  u8g.setColorIndex(0);
  // dessin d'un disque
  u8g.drawDisc(45, 36, 8);
  // on revient sur la couleur d'avant-plan
  // (juste une bonne habitude)
  u8g.setColorIndex(1);
}

void setup() {
  // il n'a rien à faire ici
}
```

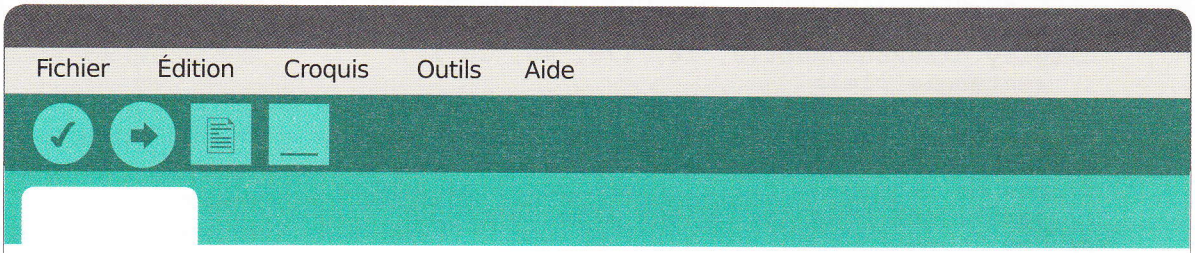
```
void loop() {
  // début de la boucle d'affichage
  u8g.firstPage();

  // boucle d'affichage d'U8glib
  do {
    // appel répété à notre fonction de dessin
    draw();
  // fin de boucle
  } while( u8g.nextPage() );
  // nextPage() retourne 0 si le travail est fini
  // et 1 si un rafraîchissement est encore nécessaire
  // Ce mode de fonctionnement permet d'économiser de
  // la mémoire vive (SRAM).

  // petite pause
  delay(1000);
}
```

Arduino

4.2 Adafruit SSD1306 et AdaFruit GFX




```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// déclaration pour SPI matériel
// SCK sur 13 et MOSI sur 11
// argument ; D/C, RST, CS
Adafruit_SSD1306 display(6, 7, 8);

// déclaration pour i2c / TWI matériel
// argument : RST
//Adafruit_SSD1306 display(7);

const uint8_t PROGMEM tour_bitmap[]= {
  B00000000,
  B01010101,
  B01010101,
  B01111111,
  B01111111,
  B00111110,
  B00111110,
```

```
    B00111110,
    B00111110,
    B00111110,
    B00111110,
    B00111110,
    B00111110,
    B00111110,
    B00111110,
    B01111111
};

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC);

  // la mémoire de l'afficheur est initialisée automatiquement
  // avec le logo de Adafruit par la bibliothèque.
  // Donc on efface d'abord l'écran
  display.clearDisplay();
  // puis on rafraîchit l'affichage
  display.display();

  // Les primitives d'affichage utilisent des arguments
  // similaires à ceux d'U8glib, mais avec, en plus, à la fin
  // l'index de la couleur utilisée (1 = avant-plan, 0 = fond)

  // affichage icônes
  display.drawBitmap(0, 0, tour_bitmap, 8, 16, 1);
  display.drawBitmap(119, 0, tour_bitmap, 8, 16, 1);
  // cadre à coins ronds
  display.drawRoundRect(0, 16, 128, 40, 10,1);
  // rectangle plein
  display.fillRect(34, 26, 40, 22, 1);
  // disque (plein)
  display.fillCircle(90, 43, 10, 1);
  // pas de segments. fillCircleHelper() ne semble
  // servir que pour les boites à coins ronds
  // Une ligne
  display.drawLine(0, 63, 127, 26, 1);
  // un triangle
  display.fillTriangle(3,20, 10,52, 20,40, 1);
  // un cercle
  display.drawCircle(127, 32, 27, 1);

  // Il n'y a pas de gestion de polices
  // une seule est disponible en une seule taille
  // Avec une taille >1 le texte est simplement grossi
  // et donc pixelisé
  display.setTextSize(1);
  // Définition de la couleur du texte.
  // WHITE est défini à 1 dans Adafruit_SSD1306.h et BLACK à 0
  display.setTextColor(WHITE);
  // position du texte
  display.setCursor(16,5);
  // Notez l'utilisation de la classe Print
  // pour l'affichage sur l'écran. On obtient un comportement
  // identique à Serial.print, incluant l'affichage de valeurs.
  // U8glib le fait aussi, mais GFX ajoute le saut de lignes
  // automatique (mais pas le défilement du texte).
  display.println("Coucou Hackable");
```

```
  // un disque en noir sur le blanc du rectangle plein
  display.fillCircle(45, 36, 8, 0); // <- 0 = BLACK = fond

  // pas de boucle ici
  // on ordonne simplement le rafraîchissement
  display.display();
}

void loop() {
  // rien ici
}
```

Arduino

4.3 U8glib ou Adafruit SSD1306+GFX ?

Comme précisé précédemment, la réponse est une question de préférence personnelle, et la mienne penche clairement vers U8glib. La présence de nombreuses polices de caractères, déclinées proprement dans plusieurs tailles est un avantage conséquent. Quelques fonctions supplémentaires comme la segmentation des cercles et disques sont également appréciables, tout comme comme la cohérence et l'élégance du code de la bibliothèque elle-même. Le fait qu'U8glib prenne en charge énormément de types d'afficheurs et de contrôleurs est une option sur l'avenir non négligeable (je ne suis pas marié avec le SSD1306).

Côté ressources utilisées, les deux croquis présentés affichent également des différences qui font pencher la balance en faveur d'U8glib. Le code utilisant cette bibliothèque nous montre après compilation une utilisation de 9688 octets (30%) de flash (mémoire programme) et 314 octets (15%) de SRAM (mémoire vive). L'exemple utilisant les bibliothèques Adafruit utilise 13184 octets de flash (40%) et 1384 octets de SRAM (67%). Ceci découle directement des choix techniques faits pour la gestion de l'affichage puisque la bibliothèque Adafruit SSD1306 alloue un buffer de 1 Ko en SRAM quoiqu'il arrive (pour un 128×64 pixels) alors qu'U8glib joue l'économie avec un buffer plus petit et un affichage en plusieurs morceaux (d'où la boucle reposant sur `nextPage()`). La SRAM est précieuse, surtout lorsqu'on n'en a que 2 Ko comme sur l'ATmega328P d'une carte Arduino UNO.

Enfin, et ceci est plus subjectif qu'autre chose, je n'aime pas du tout le fait que la bibliothèque Adafruit SSD1306 initialise d'office l'afficheur avec le logo de la société. Je comprends tout à fait le principe et le message inclus en commentaire dans les croquis exemples est tout à fait sincère (je pense) :

```
Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!
```

« *Adafruit investit du temps et des ressources pour fournir ce code open source, s'il vous plaît aidez Adafruit et l'open source matériel en achetant des produits chez Adafruit* ». On aime ou on n'aime pas ce genre de discours de la part d'une entreprise, là n'est pas la question. Un commentaire est un simple message, mais un écran qui s'initialise avec le logo d'une société est, je trouve, un peu trop intrusif à mon goût.

5. XBM, UN FORMAT ANCIEN, MAIS TRÈS PRATIQUE

XBM ou X BitMap est un format d'image non compressé très intéressant. Il est né avec le système X Window (alias X11) il y a plus de 30 ans et possède une particularité qui le rend absolument adorable pour le sujet qui nous intéresse ici : le format décrit les images sous la forme de déclarations de variables en C. Ceci permet non seulement de manipuler très facilement les données d'une image, qui n'est qu'un tableau de **unsigned char**, mais s'accompagne de la possibilité d'intégrer directement les données dans un code en C ou C++ (et donc dans un croquis Arduino).

À titre d'exemple, voici le logo monochrome du magazine en un carré de 64 pixels de côté, ceci est littéralement le contenu du fichier XBM :

```
#define logoMM_width 64
#define logoMM_height 64
static unsigned char logoMM_bits[] = {
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0x7f, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f, 0x00,
   0x00, 0xf0, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0xc0, 0xff, 0xff,
   0xff, 0x7f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0x1f, 0x00, 0x07,
   0x08, 0x00, 0xfc, 0xff, 0xff, 0x0f, 0xf0, 0x87, 0x36, 0x00, 0xf0, 0xff,
   0xff, 0x03, 0x80, 0xc3, 0xfa, 0x30, 0xe0, 0xff, 0xff, 0x01, 0x00, 0x02,
   0x46, 0x70, 0x80, 0xff, 0xff, 0x00, 0x00, 0x80, 0xd1, 0x00, 0x00, 0xff,
   0x7f, 0x00, 0x00, 0x80, 0x55, 0x00, 0x00, 0xfe, 0x3f, 0xe0, 0x1f, 0x40,
   0x80, 0x00, 0x00, 0xfc, 0x1f, 0xf8, 0xdf, 0xc0, 0xb6, 0x00, 0x0f, 0xf8,
   0x0f, 0xfe, 0xdf, 0x83, 0x32, 0xe0, 0x3f, 0xf0, 0x07, 0xff, 0xd8, 0x07,
   0x0c, 0xf0, 0xff, 0xe0, 0x87, 0x1f, 0xc6, 0x0f, 0x00, 0x78, 0xf0, 0xe1,
   0x83, 0xcf, 0x07, 0x0f, 0x00, 0x3c, 0xe0, 0xc1, 0xc3, 0xf7, 0x73, 0x1e,
   0x00, 0x1e, 0xc0, 0xc3, 0xe1, 0xcb, 0xfc, 0x1e, 0x00, 0x1e, 0xc0, 0xc3,
   0xe1, 0x09, 0xfe, 0x3c, 0x1c, 0x1e, 0xc0, 0xc7, 0xe1, 0x8d, 0xff, 0x3d,
   0x1c, 0x1e, 0xc0, 0x87, 0xe1, 0xcd, 0xff, 0x39, 0x0c, 0x1e, 0xc0, 0x87,
   0xe1, 0xcc, 0xff, 0x39, 0x1c, 0x3e, 0xe2, 0x87, 0x01, 0xc0, 0xff, 0x3b,
   0x1c, 0x3e, 0xe6, 0xc7, 0xe1, 0xcc, 0xff, 0x39, 0x1c, 0x7e, 0xe2, 0xc3,
   0xe1, 0xcd, 0xff, 0x3d, 0x1c, 0x7e, 0xf0, 0xc3, 0xe1, 0x8d, 0xff, 0x3d,
   0x3c, 0x7c, 0xf0, 0xc1, 0xe1, 0x49, 0xf8, 0x1c, 0x3c, 0x78, 0xf0, 0xe0,
   0xc3, 0xf3, 0xf8, 0x1e, 0x7e, 0x70, 0x70, 0xe0, 0xc3, 0xf7, 0x70, 0x1f,
   0x7e, 0x60, 0x30, 0xf0, 0x83, 0xcf, 0x8f, 0x0f, 0xfe, 0x00, 0x00, 0xf8,
   0x07, 0x3f, 0xc0, 0x07, 0x3f, 0x00, 0x00, 0xfc, 0x0f, 0xfe, 0xdf, 0x83,
   0x0f, 0x00, 0x00, 0xff, 0x0f, 0xfc, 0xdf, 0x81, 0x07, 0x1c, 0xc0, 0xff,
   0x1f, 0xf8, 0xdf, 0xc0, 0x03, 0xff, 0xfd, 0xff, 0x3f, 0xc0, 0x1f, 0xe0,
   0x80, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x70, 0xe0, 0xff, 0xff, 0xff,
   0xff, 0x01, 0x00, 0x3c, 0xf0, 0xff, 0xff, 0xff, 0xff, 0x03, 0x00, 0x1f,
   0xf8, 0xff, 0xff, 0xff, 0xff, 0x03, 0x00, 0x0f, 0xfc, 0xff, 0xff, 0xff,
   0xff, 0x81, 0x07, 0x06, 0xfe, 0xff, 0xff, 0xff, 0xff, 0xe1, 0x0f, 0x02,
   0xff, 0xff, 0xff, 0xff, 0xff, 0xe0, 0x17, 0x82, 0xff, 0xff, 0xff, 0xff,
   0xff, 0xd0, 0x0f, 0xc0, 0xff, 0xff, 0xff, 0x7f, 0xff, 0x90, 0x24, 0xe0,
   0xff, 0xff, 0xff, 0x3f, 0xff, 0x90, 0x24, 0xf0, 0xff, 0xff, 0xff, 0x0f,
   0xff, 0xd0, 0x1f, 0xf8, 0xff, 0xff, 0xff, 0x07, 0xff, 0xe0, 0x1c, 0xf8,
   0xff, 0xff, 0xff, 0x03, 0xff, 0x41, 0x08, 0xfc, 0xff, 0xff, 0xff, 0x01,
   0xff, 0x01, 0x00, 0xfe, 0xff, 0xff, 0xff, 0x00, 0xff, 0x03, 0x00, 0xfe,
   0xff, 0xff, 0x7f, 0x00, 0xff, 0x07, 0x00, 0xff, 0xff, 0xff, 0x3f, 0x00,
   0xff, 0xff, 0x0f, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xff, 0x7f, 0x80, 0xff,
   0xff, 0xff, 0x0f, 0x00, 0xff, 0xff, 0x83, 0xff, 0xff, 0xff, 0x07, 0x00,
   0xff, 0xff, 0xc3, 0xff, 0xff, 0xff, 0x01, 0x00, 0xff, 0xff, 0xc1, 0xff,
   0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0xe1, 0xff, 0xff, 0x7f, 0x00, 0x00,
   0xff, 0xff, 0xe1, 0xff, 0xff, 0x3f, 0x00, 0x00 };
```

Pour utiliser une telle image directement dans notre croquis, rien de plus simple puisqu'il suffit de renommer le fichier en lui donnant l'extension **.h** en lieu et place de **.xbm** puis de le placer dans le répertoire de votre croquis. Il apparaîtra alors automatiquement dans un nouvel onglet en ouvrant le projet. Vous pouvez également, bien entendu, simplement copier/coller le contenu du XBM dans votre code.

Dans le cas de la bibliothèque U8glib, la fonction **drawXBM()** pourra être alors utilisée en lui passant en argument la position en x, en y, la largeur de l'image, la hauteur de l'image et un pointeur sur les données. On préférera cependant la fonction **drawXBMP()** utilisant des données placées en mémoire flash (**PROGMEM**) plutôt qu'en SRAM. Cette fonction utilise les mêmes arguments que **drawXBM()**.

Pour cela, il conviendra de légèrement modifier la déclaration de la variable en ajoutant **U8G_PROGMEM** juste avant le **=**. Dans notre cas, nous aurons donc quelque chose comme :

```
#define logoMM_width 64
#define logoMM_height 64
static unsigned char logoMM_bits[] U8G_PROGMEM = {
   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff....
```

Notez les macros `logoMM_width` et `logoMM_height` qui font partie intégrante du format XBM et précisent la largeur et la hauteur de l'image. Ces informations peuvent être directement utilisées avec `drawXBMP()` ainsi :

```
u8g.drawXBMP(0,              // x
             0,              // y
             logoMM_width,   // largeur
             logoMM_height, // hauteur
             logoMM_bits);  // variable (pointeur)
```

Comment obtenir un fichier XBM vous demandez-vous peut-être ? Le moyen le plus simple est d'utiliser le logiciel de traitement d'images The Gimp (disponible sous GNU/Linux bien sûr, mais aussi Windows et Mac OS X). Le processus pour obtenir un rendu de qualité est le suivant :

- ouvrir l'image originale ;
- la convertir en niveaux de gris ;
- la redimensionner à la taille souhaitée pour un affichage sur l'écran OLED ;
- ajuster la luminosité et le contraste de l'image pour obtenir un résultat en noir et blanc. Le contraste est poussé au maximum et on joue sur la luminosité pour affiner le résultat ;
- passer le mode en couleurs indexées (1 bit) ;
- exporter le tout au format XBM (The Gimp vous permet de préciser le nom de la variable auquel il ajoute automatiquement `_bits`).

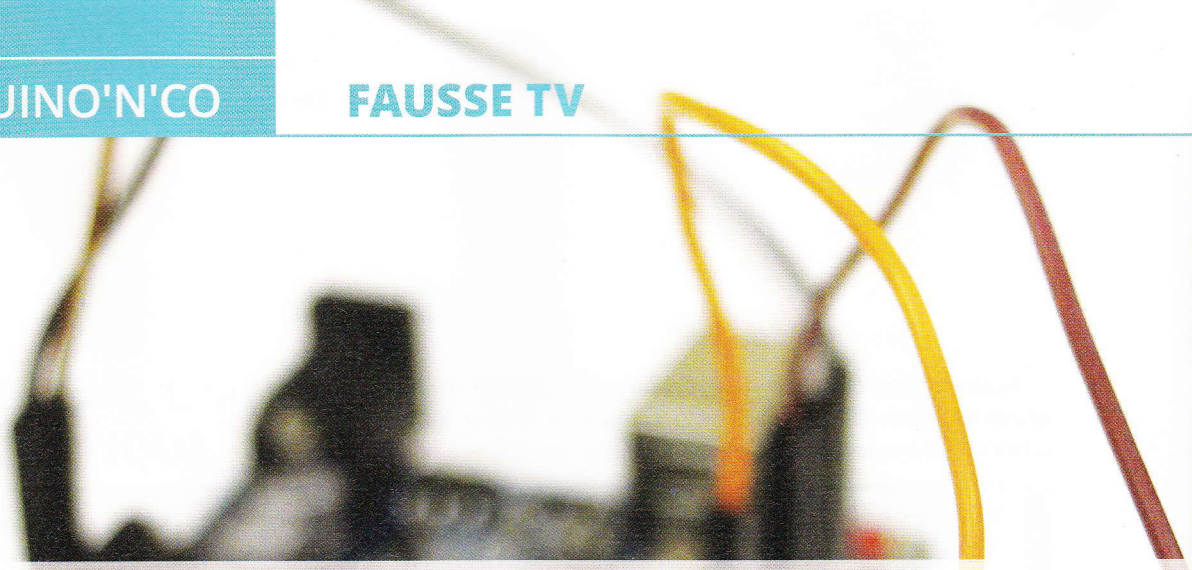
La manipulation d'images XBM est également possible avec la bibliothèque Adafruit GFX et sa fonction `drawXBitmap()`.

CONCLUSION

Nous avons, je pense, appris énormément de choses ici. Ce type d'écran n'est finalement pas si difficile à prendre en main et présente des avantages intéressants (contraste, compacité, luminosité, nombre de ports utilisés très réduit, etc.). Il est regrettable que les vendeurs sur eBay ne connaissent pas vraiment ce qu'ils proposent et que le matériel soit un peu pénible à prendre en main pour cette raison. Mais ne cherchons pas trop la petite bête, car le prix proposé compense assez largement les désagréments de ces produits chinois.

Pour conclure, je vous recommanderai simplement de « profiter de ce qui arrive ». En effet, en commandant mes modules j'étais persuadé qu'un écran divisé en deux zones de couleurs différentes était certainement moins utile que la version intégralement blanche. À l'usage, il s'avère que ce sont ces modules, achetés « juste pour voir » qui sont graphiquement les plus plaisants (même si la liaison i2c est un peu plus lente qu'en SPI). En électronique numérique, finalement, comme pour bien d'autres choses, il suffit de se laisser surprendre et de voir les choses du bon côté... DB

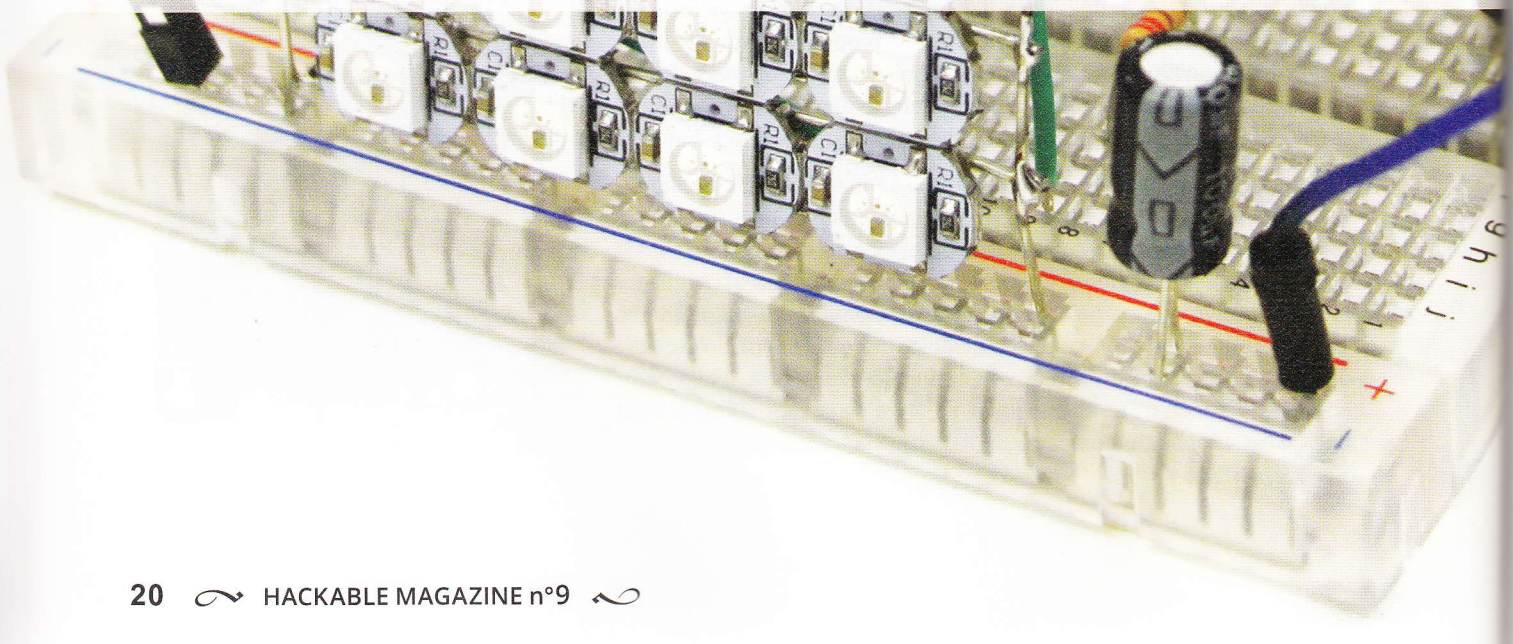
Contre toutes attentes, l'écran bichromique s'avère bien plus pratique que les versions monochromes. Ici interfacé en i2c sur un bus comprenant également un DS3231, nous pouvons avec 4 simples fils (GND/VCC/SCL/SDA) assembler un sympathique thermomètre/horloge.



CRÉEZ UN SIMULATEUR DE TV

Denis Bodor

Vous connaissez l'une des techniques permettant d'éviter de se faire cambrioler presque à coup sûr ? Rester chez soi et garder l'œil ouvert prêt à défendre bec et ongles votre territoire. Malheureusement, ceci n'est pas vraiment possible en raison de cycles biologiques pénibles et de la nécessité de sortir « chasser » les ressources nécessaires à votre survie. Il vous reste donc l'option de l'alarme ou celle consistant à simuler votre présence de façon intelligente...



Les alarmes sont populaires et parfois économiques mais, entre nous, elles ne sont pas vraiment d'une grande utilité. Soit il s'agit d'un système équipé d'une sirène et dans ce cas, généralement, les voisins se terrent chez eux effrayés ou simplement pestent contre vous, le voisin pénible avec son alarme. Parfois même sans appeler les forces de l'ordre. Soit vous utilisez une alarme connectée qui prévient directement, sous certaines conditions, un employé d'astreinte d'une agence de sécurité. Dans ce cas, les choses sont un peu plus efficaces mais, encore une fois, pas suffisamment. En effet, le plus souvent les voleurs peuvent être classés dans deux grosses catégories :

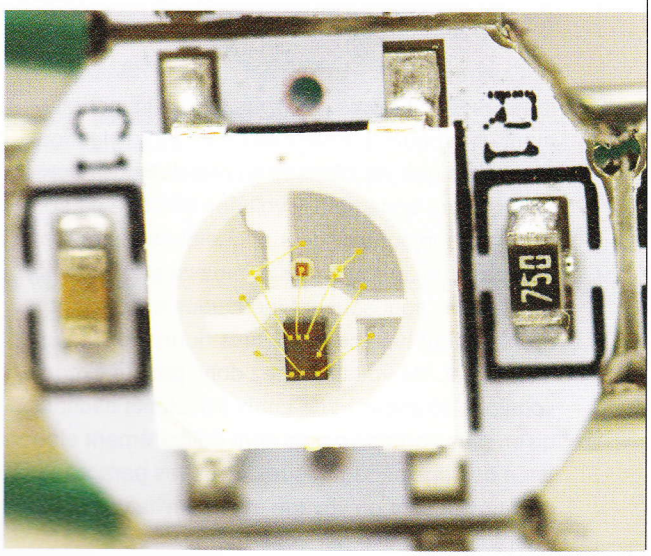
- Le petit délinquant : il ne sait pas ce qu'il cherche, rentre par effraction, repère rapidement quelque chose de valeur et de transportable puis file avec non sans avoir tout saccagé autour de lui s'il est frustré.
- Le professionnel : il repère le terrain, sait ce qu'il cherche et sait où cela se trouve. Il a un objectif et l'atteint, puis quitte les lieux.

Dans les deux cas, le plus souvent, lorsque les renforts arrivent, ils n'ont plus qu'à constater les dégâts et le délit.

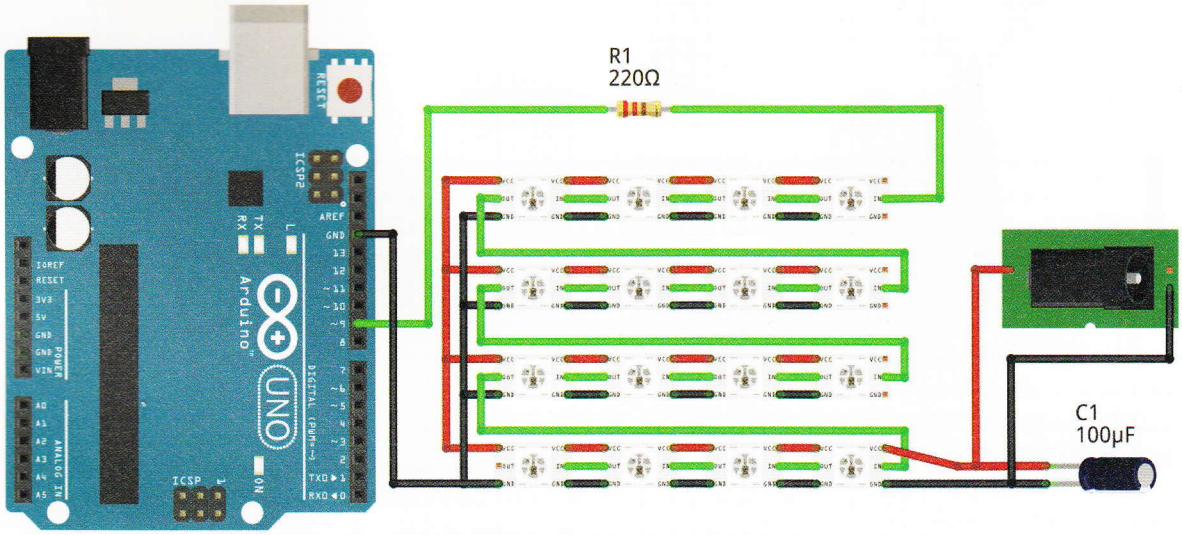
Pour ceux tentés de suggérer l'option « caméra », sachez que les voleurs peuvent utiliser un équipement de haute technologie, issu de recherches très poussées, qui offre une contre-mesure parfaite que ce soit en lumière visible ou en infrarouge : cela s'appelle... un passe-montagne et ça coûte environ 2€, ce qui est plutôt rentable pour mettre en défaut une installation à des centaines d'euros. Vous aurez juste en prime la vidéo d'un gugusse cagoulé se baladant dans votre salon (de quoi faire quelques vues sur YouTube peut-être).

Une bonne solution consiste donc à prévenir les risques d'intrusion. Les installations domotiques et d'alarmes plus « intelligentes » offrent des fonctionnalités de « simulation de présence ». Ceci consiste à faire croire qu'une personne occupe actuellement les lieux alors que ce n'est pas le cas. Ceci peut passer par le contrôle des luminaires, la mise en route de systèmes audio, l'activation de volets ou de stores et parfois même par des installations électro-mécaniques simulant des pas ou des bruits domestiques.

Côté alternatives, une habitude parfois courante consiste à laisser une radio ou un téléviseur allumé. En dehors de l'efficacité douteuse de la stratégie (une TV allumée 24h/24 est relativement peu convaincante), le coût en énergie ainsi que l'éventuelle nuisance pour le voisinage (le même qui peste contre un déclenchement d'alarme) en font une idée plutôt mauvaise. Simuler en revanche un téléviseur ou un écran d'ordinateur, avec un comportement similaire aux habitudes de l'occupant des lieux, via une carte Arduino par exemple, est économique et bien plus efficace.



La led « intelligente » WS2812b est désormais un compagnon très populaire de tous bidouilleurs électronique. De par son prix et sa facilité d'utilisation, il s'agit d'une solution presque parfaite pour toutes sortes d'application. Notez le montage de ce modèle sur un circuit accompagné d'un condensateur de filtrage et d'une résistance. Préférez toujours ce type de module aux WS2812b seules.



L'assemblage du montage est relativement simple puisqu'il suffit d'alimenter la chaîne de leds en 5V et d'utiliser une seule sortie de la carte Arduino. Notez toutefois que pour une telle quantité de leds, la carte Arduino n'est pas capable de fournir le courant nécessaire et l'usage d'un adaptateur secteur 5V de qualité sera indispensable.

Enfin pour ceux d'entre vous qui se diraient qu'une arbalète à déclenchement automatique, une fosse peuplée de crocodiles ou des douves remplies de rats mutants explosifs dressés à tuer, sont de bonnes solutions, je vous arrête tout de suite, ce n'est tout simplement pas légal. Même un seau au-dessus d'une porte, histoire de couvrir le voleur de peinture rose et de paillettes disco, placé délibérément dans ce but, peut vous envoyer devant le juge et vous coûter très cher. Il en va de même si vous tentez d'enfermer le voleur sur place, cela s'appelle un piège et c'est tout simplement interdit.

1. LE CODE ET SON ÉVOLUTION

Contrairement à ce qui est d'usage dans le magazine, je ne vais pas vous fournir immédiatement un croquis tout fait contenant toutes les fonctionnalités que j'ai incluses au projet. Nous allons plutôt suivre ensemble l'évolution du code de son simple « bonjour monde » à, quelques générations suivantes, un croquis plus complexe et plus étoffé. Ces étapes sont souvent masquées et pourtant représentent bien ce processus normal de la vie d'un projet. Ne vous étonnez donc pas de circonvolutions étranges et de choix troublants sachant qu'en plus certains points sont délibérément ajoutés à des fins pédagogiques. Bref, les paragraphes qui suivent sont un voyage et une invitation à le poursuivre ensuite par vous-même...

1.1 Au départ tout est simple

Le montage initial n'a rien de compliqué. Il s'agit, tout simplement d'utiliser les fameuses led WS2812b, agencées en un carré de 4 par 4. Pour rappel (voir *Hackable n°6*), ces petits composants sont des led tricolores « intelligentes » ne nécessitant que trois connexions : une masse, une alimentation 5V et un signal pour les contrôler. Chaque led dispose d'une entrée pour ce signal ainsi qu'une broche en sortie pour passer les données à sa voisine. Car, en effet, ces leds sont chaînées, la sortie de données de l'une étant reliée à l'entrée de la suivante, et ainsi de suite.

Pour les contrôler, quel que soit leur nombre, on doit générer un signal dans lequel est encodé la couleur qu'elles doivent prendre. Ceci se fait très simplement en reposant sur une simple sortie d'une carte Arduino et en mettant en œuvre la bibliothèque Adafruit NeoPixel.

Notre premier croquis sera donc aussi simple qu'inutile :

Fichier Édition Croquis Outils Aide

```
#include <Adafruit_NeoPixel.h>

// nombre de leds WS2812b chaînées
#define PIX 16
// intensité maximum
#define MAXB 255
// intensité minimum
#define MINB 35

// pixels est l'objet représentant notre chaîne
// connectée à la broche 9
Adafruit_NeoPixel pixels =
  Adafruit_NeoPixel(PIX, 9, NEO_GRB + NEO_KHZ800);

// Cette variable contient les valeurs de rouge, de
// vert et de bleu pour chaque "pixel"
// 16 pixels et trois valeurs pour chacune = 48 valeurs
// Notez le PROGMEM afin de stocker tout cela en flash
// et non en SRAM
const unsigned char simple[48] PROGMEM  = {
  255,255,255,  255,255,255,  255,255,255,  255,255,255,
    0,  0,255,    0,  0,255,    0,  0,255,    0,  0,255,
    0,255,  0,    0,255,  0,    0,255,  0,    0,255,  0,
    0,255,  0,    0,255,  0,    0,255,  0,    0,255,  0
};

// Notre fonction d'envoi des valeurs
void pmotif(int tempo) {
  int r,g,b;
  // On boucle sur le nombre de leds
  for(int j=0; j<PIX; j++) {
    // Comme les données sont en flash nous devons
    // les récupérer
    r =  pgm_read_byte_near(simple + (j*3)+0);
    g =  pgm_read_byte_near(simple + (j*3)+1);
    b =  pgm_read_byte_near(simple + (j*3)+2);
    // On donne la couleur à chaque led
    pixels.setPixelColor(j, r, g, b);
  }
  // puis on valide l'envoi en "affichant" les données
  pixels.show();
  // et on fait une pause
  delay(tempo*100);
}

void setup() {
  // Initialisation
  pixels.begin();
  // Tous les pixels éteints
```

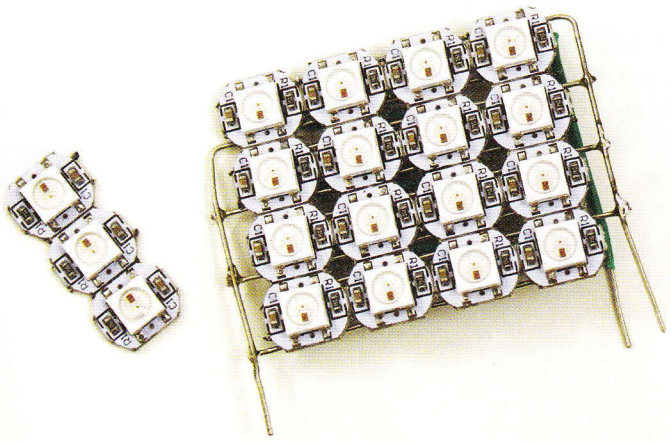
```
  pixels.clear();
  // On règle l'intensité lumineuse globale
  // Ceci est très pratique pour ne pas se tuer les
  // yeux en affinant et testant le code
  pixels.setBrightness(MAXB);
  // On envoie les données
  pixels.show();
  // et on marque une pause
  // Là encore ceci est pratique pour tester le code
  // afin qu'on puisse reconnaître l'initialisation par
  // un "vide" de 2 secondes
  delay(2000);
}

void loop() {
  // On ne fait qu'appeler notre fonction en boucle
  pmotif(2);
}
```

Arduino

Selon la source d'approvisionnement de vos modules WS2812b, il est possible, comme ici, que vous les receviez montées sur un seul circuit (PCB). Ceci peut être utilisé à votre avantage puisqu'en conservant cet agencement et en ne séparant pas chaque composant, vous disposez déjà de votre matrice qu'il vous suffira de souder.

Et là, vous vous dites « mais c'est complètement stupide », et vous avez raison. Ce n'est qu'un squelette puisque nous passons notre temps à appeler toujours la même fonction qui ne fait qu'afficher le même motif. L'idée est, bien entendu, de développer d'autres fonctions, générant d'autres effets et changements sur les leds et de les appeler en boucle afin d'alterner les différents rendus de façon aléatoire.



1.2 D'autres fonctions

Nous commençons par intégrer une fonction que nous avons déjà vue dans le magazine, permettant d'obtenir les valeurs de rouge, de vert et de bleu dans les bonnes proportions en fonction de critères de teinte, de saturation et de valeur (TSV ou HSV en anglais). L'avantage de ce système colorimétrique est de disposer d'un cercle chromatique de 360 degrés couvrant toutes les couleurs (la teinte). Comme il s'agit d'un cercle, nous pouvons à l'aide d'une simple boucle, « tourner » pour passer en douceur d'une couleur à l'autre.

Nous n'allons pas reprendre ici le code de cette fonction puisque nous l'avons déjà vu dans un précédent numéro et que le croquis de notre actuel projet sera accessible sur GitHub (**https://github.com/Hackable-magazine**). Voici, en revanche notre nouvelle fonction d'affichage pour notre croquis :

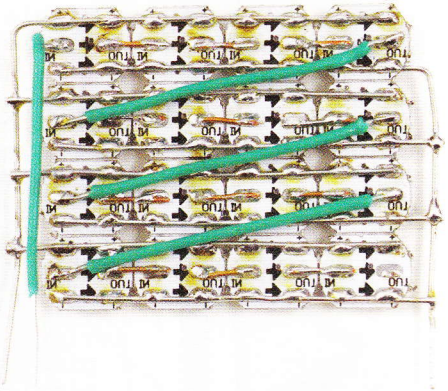
```
void arcenciel(int tempo) {
  int r,g,b;
  // On boucle dans le cercle
  for(int i=0; i<360; i++) {
    // On passe TSV et on récupère RVG
    HSVtoRGB( &r, &g, &b, i, 255, 255);
    // On fait le tour des leds
    for(int j=0; j<PIX; j++) {
      // On donne la même valeur aux 16 leds
      pixels.setPixelColor(j, pixels.Color(r,g,b));
    }
    // Envoi
    pixels.show();
    delay(tempo);
  }
}
```

Il s'agit d'une simple boucle **for** parcourant le cercle de couleurs de 0 à 359 degrés et reposant sur les valeurs issues de **HSVtoRGB()**. Notez que :

- **&r**, **&g** et **&b**, passés en argument, sont des pointeurs sur les variables **r**, **g** et **b**. Ce sont les emplacements de ces variables et non leur contenu qui est passé en paramètre. La fonction utilise ces informations pour stocker les valeurs à ces emplacements et après son exécution, les variables contiennent les bonnes valeurs.
- **i** est incrémenté de 0 à 359 et passé en argument de la fonction, c'est la teinte à utiliser.
  La saturation et la valeur sont fixées au maximum, on obtient donc des couleurs très vives.
- **tempo** est utilisé de façon différente ici. On utilise toujours sa valeur comme temporisation, mais pour la transition d'une couleur à une autre et non comme temps de pause après l'envoi des données.

Nous avons à présent deux fonctions d'affichage qui utilisent un argument de type identique (ce n'est pas un hasard). Nous pouvons créer d'autres fonctions d'affichage et dérivés, comme par exemple celle affichant seulement un morceau d'arc-en-ciel :

```
void marcenciel(int tempo) {
  int r,g,b;
  int debut = random(0,360);
  int fin = random(debut,360);
  for(int i=debut; i<=fin; i++) {
    HSVtoRGB( &r, &g, &b, i, 255, 255);
    for(int j=0; j<PIX; j++) {
      pixels.setPixelColor(j, pixels.Color(r,g,b));
    }
    pixels.show();
    delay(tempo);
  }
}
```



En reposant sur l'agencement initial, il nous suffit d'interconnecter les broches d'alimentation et de masse, puis de chaîner les broches OUT et IN de chaque module. Remarquez ici la connexion de l'alimentation par ligne et non sous la forme d'une chaîne. Dans le cas contraire, avec une importante quantité de leds, les composants au bout de la chaîne pourraient voir leur alimentation fortement perturbée/déstabilisée...

C'est assez similaire à ce que nous venons de voir, à la différence que les valeurs de départ et de fin pour la boucle sont choisies aléatoirement avec `random()` prenant en argument une valeur minimum et une valeur maximum (exclusive). On pensera à ajouter un appel à `randomSeed(analogRead(0))` dans notre fonction `setup()` afin d'initialiser (ensemencer) la suite de valeurs pseudo-aléatoire.

Ajoutons-en encore une pour donner une couleur fixe et deux autres qui ne touchent pas aux couleurs, mais font varier l'intensité globale :

```
void couleur(int tempo) {
  int r,g,b;
  HSVtoRGB( &r, &g, &b, random(0,360), 255, 255);
  for(int j=0; j<PIX; j++) {
    pixels.setPixelColor(j, pixels.Color(r,g,b));
  }
  pixels.show();
  delay(tempo*100);
}

void fondu_out(int tempo) {
  // On boucle entre le maximum et le minimum
  for(int i=MAXB; i>MINB; i--) {
    // Cette fonction impacte l'ensemble des leds
    pixels.setBrightness(i);
    pixels.show();
    delay(tempo*2);
  }
  // On revient au maximum pour qu'une autre fonction
  // puisse toujours afficher correctement
  pixels.setBrightness(MAXB);
  pixels.show();
}

void fondu_in(int tempo) {
  // On boucle entre le minimum et le maximum
  for(int i=MINB; i<=MAXB; i++) {
    pixels.setBrightness(i);
    pixels.show();
    delay(tempo*2);
  }
}
```

À ce stade, nous avons à notre disposition les fonctions `arcenciel()`, `marcenciel()`, `pmotif()`, `couleur()`, `fondu_out()` et `fondu_in()`. La question est à présent de savoir comment les appeler dans `loop()`. La première idée qui vient à l'esprit est de simplement les lister avec, pourquoi pas, un appel à `random()` pour varier les temporisations. Le problème qui se pose alors est le fait que ce sera toujours la même série de motifs qui sera répétée. Nous devons ajouter un caractère aléatoire à ces appels.

1.3 Des histoires de hasard et de pointeurs

La solution simple pour rendre les appels aux fonctions aléatoires serait l'algorithme suivant :

- choisir une valeur aléatoire dans un intervalle borné ;
- utiliser cette valeur avec une série de conditions `if`/`else if` ou un `switch`/`case`.

Il n'y a pas vraiment de problème avec cette démarche si ce n'est que le code peut s'avérer relativement long et source d'erreur en cas de modification. En effet, si on ajoute une fonction, il faut revoir notre enchaînement de conditions ou ajouter une clause **case** dans notre **switch/case**. C'est pénible et pas vraiment amusant (parce que l'aspect ludique du code est très important).

Nous allons plutôt utiliser une fonctionnalité du langage C/C++ pour pouvoir utiliser quelque chose comme ceci en guise de fonction **loop()** :

```
void loop() {
  fonction ftable[6] = {
    arcenciel,marcenciel,pmotif,couleur,fondu_out,fondu_in
    };
  ftable[random(0,6)](random(1,10));
}
```

Ce code déclare **ftable** qui est un tableau de variables de type **fonction**, initialisé avec nos fonctions déclarées par ailleurs dans le croquis, puis utilise aléatoirement un élément du tableau comme s'il s'agissait... d'une fonction ?! Est-ce de la sorcellerie ? Non, c'est du C et un usage intéressant des pointeurs sur fonctions.

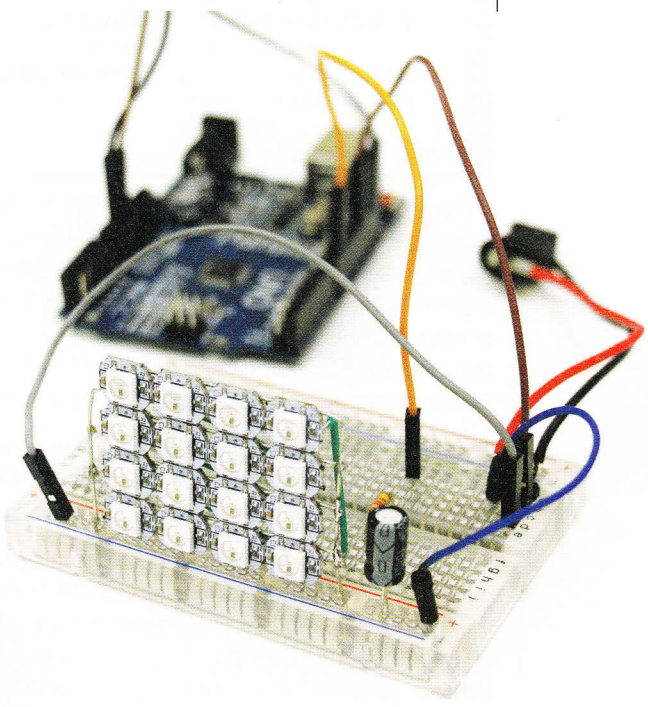
Mais commençons par le début. Si vous êtes lecteur fidèle du magazine, vous le savez déjà, les variables peuvent être vues comme des bocaux contenant des choses d'un certain type (caractère, entier, etc.) et ces bocaux sont rangés quelque part à un emplacement précis. On peut accéder au contenu d'un bocal par son nom, le nom de la variable, mais également utiliser l'emplacement lui-même en faisant précéder le nom d'un **&**. Il est également possible de réserver un emplacement de bocal, qui sera alors désigné par un nom et on accédera alors au contenu en faisant précéder le nom d'un *****. Un bocal est une variable et un emplacement est une adresse mémoire. Un pointeur est une variable contenant cette adresse.

Ce n'est pas tout, et c'est là que cela devient très amusant : les fonctions ont aussi un emplacement en mémoire et donc une adresse. Elles peuvent donc être utilisées dans des pointeurs... sur fonction.

Pour comprendre le contenu de notre **loop()**, il faut avant tout parler du type **fonction**. Ce type de variable n'existe pas, je l'ai créé par ailleurs dans ce croquis :

```
typedef void (*fonction) (int t);
```

typedef est une instruction permettant de créer des types de données qui ne sont pas fournis par défaut. C'est une très bonne solution pour se simplifier la vie (surtout avec les structures).



Le montage complet sur platine à essais serait presque suffisant pour un usage ponctuel du simulateur. Pour une version définitive, on prendra soin de permettre une circulation d'air pour refroidir les 16 WS2812b et éviter tout problème. L'ajout d'une lentille de Fresnel, type loupe/règle de poche, permettra de diffuser de manière plus homogène la lumière et ainsi améliorer l'effet TV.

void est généralement utilisé dans une déclaration de fonction lorsque celle-ci ne retourne rien. Ici c'est un peu différent, nous définissons un type nommé **fonction** qui n'a pas de type, c'est un pointeur générique, littéralement une simple adresse, un endroit en mémoire. C'est un pointeur sur une fonction prenant un argument de type **int** (entier).

Pour comprendre l'utilisation de ce genre de déclaration, rien de tel qu'un exemple plus simple :

```
// Définition de notre type
typedef void (*fonction) (int t);
// Fonction simple prenant un argument
void coucou(int num) {
  for(int i=0; i<num; i++) {
    Serial.println("Coucou monde!");
  }
}
void setup() {
  // Initialisation du port série
  Serial.begin(115200);
  delay(2000);
  // On appelle notre fonction
  Serial.println("---- coucou(2):");
  coucou(2);
  // On déclare une variable de type fonction
  // et on l'initialise avec l'adresse de coucou
  fonction pouet = coucou;
  // On appelle la fonction pouet
  Serial.println("---- pouet(3):");
  pouet(3);
}
void loop() {}
```

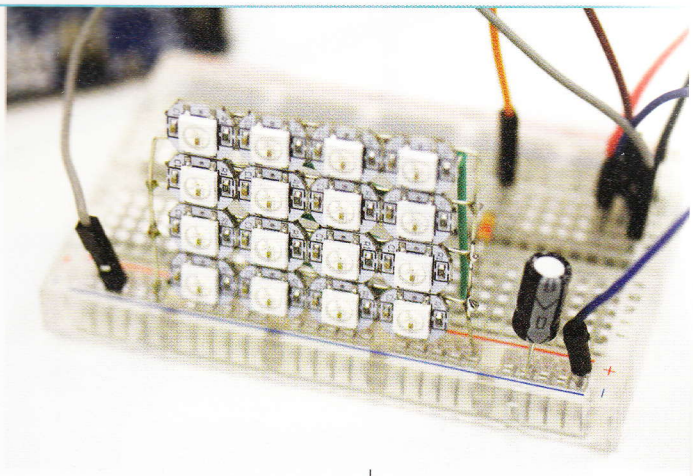
Nous avons là une fonction **coucou()** parfaitement normale. Dans **setup()** nous en faisons usage pour afficher un message via le moniteur série. Nous déclarons ensuite une autre variable, **pouet**, qui est du type **fonction** et donc un pointeur sur une adresse d'une fonction prenant en argument une variable de type **int**. **pouet** est donc destiné à contenir une adresse de fonction et nous l'initialisons avec l'adresse de **coucou**. En effet, lorsque vous déclarez une fonction, vous déclarez en réalité un pointeur sur une fonction retournant un certain type de données (ici **void**) et acceptant en argument un certain nombre de variables (ici, une, de type **int**).

pouet pointe sur exactement la même adresse que **coucou**. Il n'y a donc rien d'étonnant au fait qu'on puisse alors l'utiliser exactement de la même manière que **coucou**, avec le même type d'argument et le même résultat. C'est littéralement le même code qui est appelé.

Notre **ftable[]** n'est pas différent, mais plutôt que de déclarer une seule variable nous déclarons un tableau de six **fonction** et l'initialisons avec les pointeurs vers chacune des fonctions que nous avons déclaré précédemment. Le tableau contient donc 6 pointeurs vers des fonctions ne retournant rien (**void**) et prenant en argument un **int**. Pour utiliser l'une d'entre elles, il nous suffit de spécifier la position dans le tableau et l'argument. Par exemple, **ftable[3](5)** appellera la fonction pointée par l'adresse stockée à la position 3 qui est celle de **couleur()**, avec l'argument 5.

Pour ajouter l'aspect aléatoire, il nous suffit alors d'intégrer **random()** dans l'appel : **ftable[random(0,6)](random(1,10))**.

Mais nous pouvons encore nous simplifier les choses. Nous déclarons un tableau de 6 éléments que nous initialisons avec 6 pointeurs vers nos fonctions, puis nous réutilisons cette



En faisant preuve d'un peu d'astuce et de connecteurs rigides, l'assemblage de la chaîne de leds pourra également servir de support pour la mise au point sur platine à essais. Notez également le condensateur sur la droite permettant de stabiliser la tension d'alimentation (les WS2812b sont très sensibles), ainsi que la résistance de 220 ohms en arrière-plan, placée entre la sortie de la carte Arduino et le IN de la première led.

valeur dans l'appel à **random()** en guise de maximum. Si nous ajoutons une fonction et passons notre tableau à 7 éléments et ne changeons pas l'appel à **random()**, nous n'utiliserons jamais la nouvelle fonction. Mais il y a pire !

Si nous passons notre tableau à 5 éléments et oublions de changer **random()**, c'est le début de gros gros problèmes. En effet, dans cette situation il y a une chance que nous appelons la fonction qui se trouve à l'adresse pointée par le sixième élément du tableau. Or, nous n'avons pas la moindre idée de ce qui se trouve effectivement à **ftable[5]** ! Il y a bien quelque chose, mais sans doute pas l'adresse d'une fonction que nous avons choisie. Le code peut faire absolument n'importe quoi selon ce qui se trouve là. C'est là précisément ce qui fait qu'un certain nombre de programmeurs ont une peur bleue des pointeurs (personnellement c'est une forme de proximité avec le matériel que j'affectionne énormément).

Nous pouvons éliminer cette source d'erreur en n'ayant qu'une seule occurrence de la taille du tableau, dans sa déclaration, et en calculant le reste. Une fonction est à notre disposition pour cela : **sizeof()** (« taille de »). Celle-ci retourne la taille d'une variable, mais encore faut-il savoir ce qu'on cherche. **sizeof(ftable)** retournera la taille de **ftable** en octets. Il ne s'agit pas du nombre d'éléments du tableau, mais de la taille totale. Pour obtenir le nombre d'éléments, nous devons diviser cette valeur par la taille d'un seul élément, soit **sizeof(ftable[0])** et donc faire **sizeof(ftable)/sizeof(ftable[0])**. Ce qui dans notre appel deviendra :

```
ftable[random(0,
  sizeof(ftable)/sizeof(ftable[0])
  )](random(1,10));
```

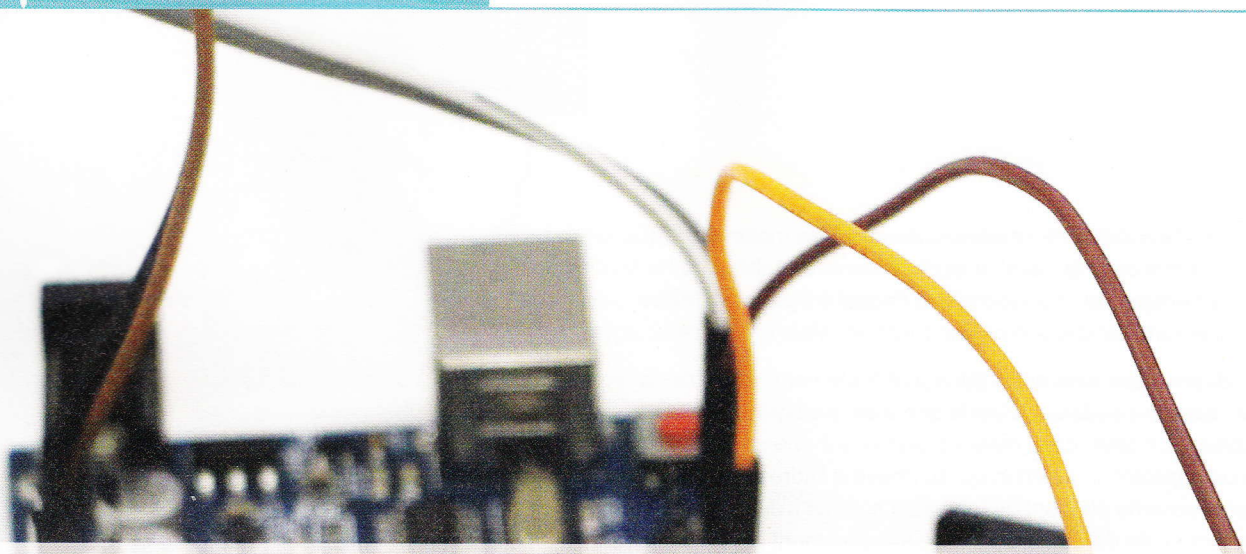
Mais ce n'est pas tout ! Nous avons oublié quelque chose de très important (ou plutôt avons été un peu tête-en-l'air) : lorsqu'on initialise un tableau au moment de sa déclaration, nous n'avons pas besoin de définir sa taille ! Notre **fonction ftable[6] = {arcenciel,...** peut parfaitement se résumer à **fonction ftable[] = {arcenciel,...** Nous avons donc totalement éliminé cette notion de taille du tableau et pouvons alors nous contenter de lister les fonctions à utiliser ! (oui, c'était délibéré)

À présent, tout ce que nous avons à faire est de démultiplier les fonctions en faisant preuve d'imagination et de modifier notre tableau en conséquence. Automatiquement, celles-ci seront alors utilisées aléatoirement.

2. ÉVOLUTIONS

Une évolution possible du code concerne le contrôle de cet aspect aléatoire. Nous avons ici six fonctions générant des effets lumineux, mais certains d'entre eux sont peu courants au regard du comportement habituel d'un téléviseur. Je n'ai pas de TV, mais je vois rarement des transitions entre toutes les couleurs du spectre visible dans mes séries préférées. Quel que soit le type d'émission télévisée à simuler, certains effets sont plus fréquents que d'autres.

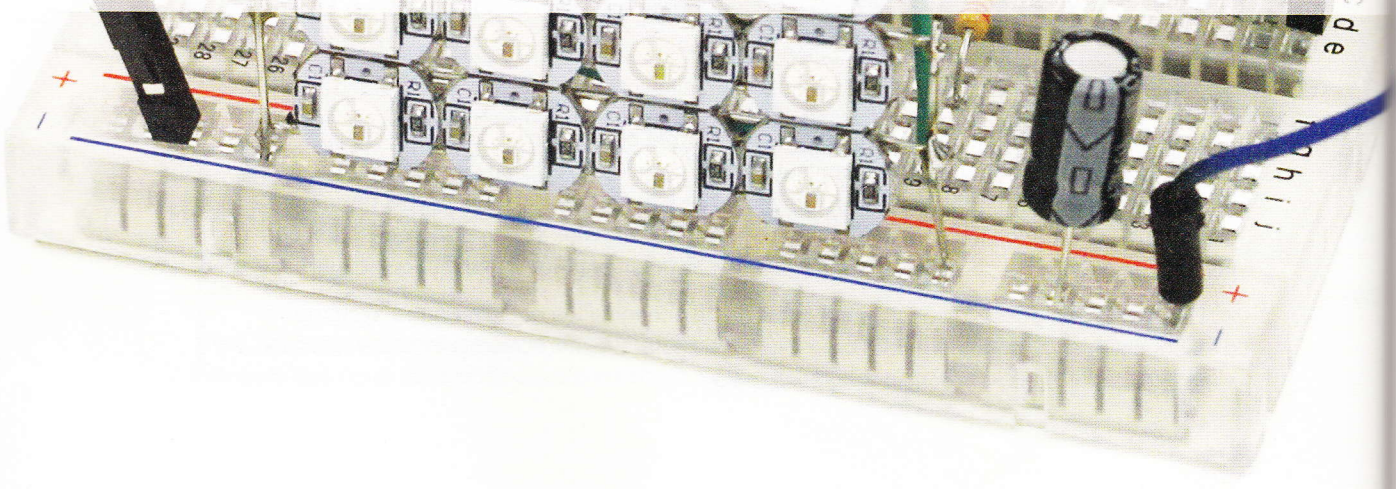
Une autre voie possible consiste à ajouter des fonctionnalités et étayer notre système d'affichage en se basant sur une source d'images fiable. C'est précisément ce que nous allons découvrir dans le prochain article. DB



AMÉLIOREZ VOTRE SIMULATEUR DE TV

Denis Bodor

Nous avons, dans l'article précédent, esquissé les grandes lignes de notre simulateur et en avons profité pour découvrir le monde merveilleux des pointeurs sur fonction. Nous allons à présent nous pencher sur deux autres fonctionnalités utiles et amusantes : l'extraction d'images réelles comme base d'affichage et la programmation horaire.



L'idée du simulateur de téléviseur n'est pas nouvelle. Il existe en effet, différents modèles commercialisés sous plusieurs marques et dénominations : *fake TV*, *TV Simulator*, *Burglar deterrent*, *Dummy televisor*... à des prix allant de 5 à 40 euros. Je n'ai pas la moindre idée de la logique ou des algorithmes en œuvre dans ces produits, mais le but ici n'est pas de copier l'existant, mais de faire travailler son imagination et sa créativité.

La tâche n'est pas aisée, car s'il est question de simuler un téléviseur, il faut avant tout déterminer ce qui fait qu'un tel appareil diffuse une lumière si caractéristique. Il faut ensuite extraire ces spécificités et les reproduire sur un montage de taille bien plus réduite qu'un téléviseur et composé de seulement quelques 16 leds tricolores.

Mon approche de ce problème est la suivante : une émission TV est une succession de scènes composées d'images. En extrayant une série d'images d'une telle émission et en réduisant la résolution en 4×4 pixels, il est donc possible d'obtenir de vraies données « de terrain » utilisables pour composer nos affichages.

Il nous faut cependant revoir à cet effet la fonction écrite pour afficher un ensemble de pixels sur ce que je qualifierai désormais d'écran.

1. DES IMAGES UN PEU PLUS NOMBREUSES

Pour rappel, notre fonction et ses données dans notre croquis ressemblent actuellement à ceci (les commentaires sont retirés afin de rester concis) :

Fichier Édition Croquis Outils Aide

```
const unsigned char simple[48] PROGMEM  = {
  255,255,255,  255,255,255,  255,255,255,  255,255,255,
    0,  0,255,    0,  0,255,    0,  0,255,    0,  0,255,
    0,255,  0,    0,255,  0,    0,255,  0,    0,255,  0,
    0,255,  0,    0,255,  0,    0,255,  0,    0,255,  0
};

void pmotif(int tempo) {
  int r,g,b;
  for(int j=0; j<PIX; j++) {
    r =  pgm_read_byte_near(simple + (j*3)+0);
    g =  pgm_read_byte_near(simple + (j*3)+1);
    b =  pgm_read_byte_near(simple + (j*3)+2);
    pixels.setPixelColor(j, r, g, b);
  }
  pixels.show();
  delay(tempo*100);
}
```

Arduino

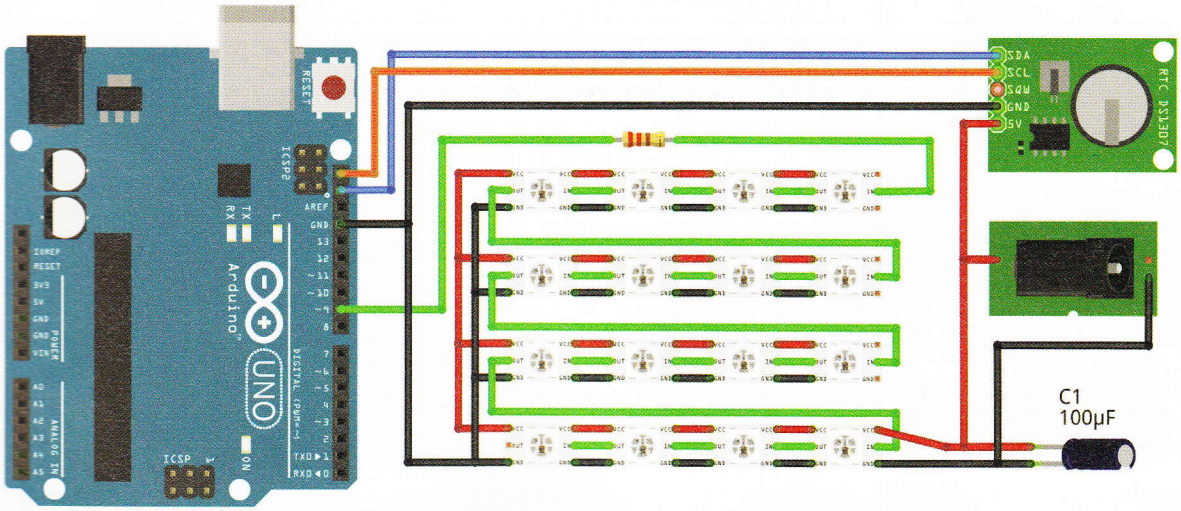
Si nous utilisons ce stratagème pour inclure dans notre code quelques dizaines d'images, ceci va rapidement devenir ingérable puisqu'il faudrait autant de variables (tableaux de 48 **unsigned char**) et surtout de fonctions que d'images. C'est complètement stupide. Mieux vaut tout regrouper dans une seule variable, traitée par une seule fonction qui choisira aléatoirement les données à afficher sur l'écran.

Nous créons alors une autre fonction (ou transformons celle existante) ainsi :

```
void gmotif(int tempo) {
  // Variables pour rouge, vert et bleu
  int r,g,b;
  // Choix aléatoire d'une image
  // NBRIMG est le nombre d'images présentes dans
  // la variable/tableau images[]
  int idx=random(0,NBRIMG)*48;
  // On boucle sur les 48 valeurs à envoyer
  for(int j=0; j<PIX; j++) {
    r =  pgm_read_byte_near(images + (j*3)+idx+0);
    g =  pgm_read_byte_near(images + (j*3)+idx+1);
    b =  pgm_read_byte_near(images + (j*3)+idx+2);
    pixels.setPixelColor(j, r, g, b);
  }
  // On affiche
  pixels.show();
  // Petite pause
  delay(tempo*100);
}
```

Le principal avantage de l'utilisation d'un module RTC DS1307 est sa simplicité de connexion. Il s'agit en effet d'un bus i2c (également appelé TWI) ne nécessitant que deux connexions sur les broches dédiées des cartes Arduino. Ne perdez cependant pas de vue que, comme ici avec une Arduino UNO, les broches SDA et SCL sont partagées respectivement avec A4 et A5 qui ne sont alors plus utilisables en tant que tel.

Initialement, une image est stockée sous la forme d'une série de trois valeurs par « pixel » : rouge, vert et bleu. La boucle consiste à aller de 0 au nombre de leds (**PIX**) moins 1 (**j<PIX**). Comme nous avons trois valeurs par pixel, la valeur de rouge se



trouve toutes les trois valeurs du tableau, soit `j*3`. Pour obtenir le vert, ce sera donc `(j*3)+1` ou en d'autres termes, la valeur se trouvant à la position juste après le numéro du pixel fois trois. Idem pour le bleu, avec `(j*3)+2`.

Si nous étendons ce fonctionnement à un tableau contenant plusieurs images, nous avons donc une nouvelle image de 16 pixels de 3 valeurs toutes les 48 positions. Il nous suffit alors de choisir un numéro d'image, multiplier ce numéro par 48 pour obtenir sa première valeur dans le tableau et procéder comme avec une seule image, mais en utilisant le décalage (`idx`) à chaque récupération de valeur.

Il nous faut à présent construire notre énorme variable `image[]` et définir `NBRIMG`. Notez que vous pouvez vous débarrasser de `NBRIMG` avec `sizeof()`, mais utiliser une macro (`#define`) est plus économique et moins risqué qu'avec notre tableau de pointeurs sur fonctions.

Nous n'allons pas créer la variable manuellement. Un bon programmeur est avant tout un paresseux avec un processus cognitif digne de l'agent Smith : n'envoyez jamais un humain faire le travail d'une machine.

1.1 Utiliser les données d'une vidéo

Quoi de mieux pour obtenir une sélection d'images de 4×4 pixels à destination d'un simulateur de TV que d'extraire celles-ci d'une véritable émission TV ? C'est exactement ce que nous allons faire et ceci implique l'utilisation d'un système riche en outils pour :

- extraire automatiquement des images d'une vidéo à intervalles réguliers ;
- convertir ces images à un format de 4 pixels de côté ;
- obtenir les valeurs RVB de chacun des 16 pixels de chaque image ;
- formater ces données avec une syntaxe utilisable pour notre croquis ;
- et stocker tout cela dans un fichier importable dans l'IDE Arduino.

Ce système est bien entendu GNU/Linux et vous pouvez à loisir utiliser une version présente sur votre PC ou celle d'une Raspberry Pi. Les paquets nécessaires sont `ffmpeg` et `imagemagick`. Le premier est une trousse à outils pour la vidéo et le second pour la manipulation d'images (notez que l'utilisation sur Pi sera bien plus lente qu'avec un PC).

En guise d'exemple, nous utiliserons la vidéo *Big Buck Bunny* disponible sous licence Creative Commons et étant principalement une démonstration artistique et technologique de la Fondation Blender (un logiciel libre de modélisation, d'animation et de rendu en 3D open source sous licence GPL). Ce qui va suivre est applicable à n'importe quel média à un format/codec supporté par les bibliothèques Libav.

Après récupération de la vidéo dans le format le plus petit possible (62 Mo tout de même) et installation des paquets, nous commençons par transformer la vidéo en une série d'images :

```
$ mkdir img
$ ffmpeg -i BigBuckBunny_320x180.mp4 -r 1/60 img/img%03d.png
```

Nous utilisons ici les options `-i` pour spécifier un fichier en entrée, `-r` pour indiquer un débit en image par seconde et enfin un fichier en sortie. Deux points sont remarquables :

- `-r 1/60` précise « un soixantième d'image toutes les secondes », soit une image par minute ;

- `img%03d.png` permet de générer non pas un fichier, mais une série de fichiers nommés `img001.png`, `img002.png`, `img003.png`, etc.

Nous obtenons ainsi une douzaine de fichiers PNG de 320 par 180 pixels que nous pouvons afficher pour y faire un brin de ménage. Les images de couleur unies (noires en particulier) ne nous intéressent pas. Il en va de même pour le générique que personne ne regarde jamais (ou « ne regardait jamais » jusqu'à ce que quelqu'un ait la brillante idée de placer une scène bonus juste après, histoire de vous obliger à lire les 75 noms de l'équipe technique et celui du gars qui préparait le café sur le tournage).

Une fois la sélection d'images réduite, nous nous plaçons dans le répertoire et transformons tout cela en une série de fichiers texte :

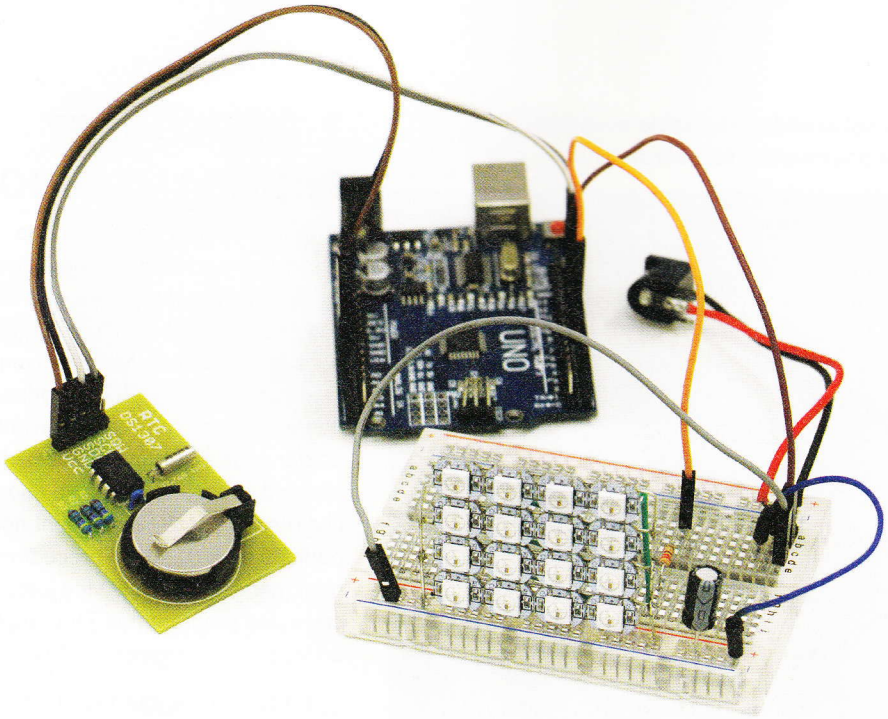
```
$ cd img
$ mkdir txt
$ for i in *.png; do convert -resize 4x4\! $i txt/$i.txt; done
```

Le shell Bash est quelque chose d'adorable et merveilleux. Cette dernière ligne est une boucle `for` parcourant la liste des fichiers PNG et lançant une commande pour chacun d'eux. `$i` est une variable contenant le nom du fichier dans le corps de la boucle et nous utilisons l'outil `convert` d'ImageMagick pour redimensionner l'image (`-resize`) en 4×4 pixels (le `!` force la non-proportionnalité et `\` évite que le caractère soit interprété par le shell) et le convertir en texte. Cette conversion est propre à ImageMagick qui utilise l'extension du fichier pour déterminer le format en sortie, il s'agit d'une description textuelle du contenu de l'image. Exemple :

```
# ImageMagick pixel enumeration: 4,4,255,srgb
0,0: (101,116, 67)  #657443  srgb(101,116,67)
1,0: ( 69, 87, 46)  #45572E  srgb(69,87,46)
2,0: ( 44, 65, 37)  #2C4125  srgb(44,65,37)
3,0: ( 35, 50, 38)  #233226  srgb(35,50,38)
0,1: ( 67, 93, 74)  #435D4A  srgb(67,93,74)
1,1: (106,108, 71)  #6A6C47  srgb(106,108,71)
2,1: ( 64, 78, 65)  #404E41  srgb(64,78,65)
3,1: ( 34, 56, 52)  #223834  srgb(34,56,52)
0,2: (108,129, 62)  #6C813E  srgb(108,129,62)
1,2: (133,130, 48)  #858230  srgb(133,130,48)
2,2: ( 69, 86, 44)  #45562C  srgb(69,86,44)
3,2: ( 59, 90, 66)  #3B5A42  srgb(59,90,66)
0,3: (106,127, 40)  #6A7F28  srgb(106,127,40)
1,3: (107,127, 41)  #6B7F29  srgb(107,127,41)
2,3: (137,147, 40)  #899328  srgb(137,147,40)
3,3: (166,172, 48)  #A6AC30  srgb(166,172,48)
```

Nous avons en première ligne un commentaire (`#`) puis une succession de lignes décrivant chaque pixel avec tout d'abord sa position puis plusieurs syntaxes décrivant sa couleur. Le premier pixel en haut à gauche de l'image est donc 101/255 rouge, 116/255 vert et 61/255 bleu (ou `#657443` en notation HTML). C'est exactement ce dont nous avons besoin pour notre croquis !

Il ne nous reste donc plus qu'à mettre tout ça en forme. Pour cela, nous pouvons soit écrire une longue ligne de commandes, soit créer un script shell, qui sera bien plus lisible. Nous créons donc un fichier texte, `gopix.sh` dans notre répertoire personnel par exemple, contenant :



Le montage final assemblé pour test et mis au point du code, en compagnie du module RTC. Notez le connecteur d'alimentation relié à la platine à essais. Il est en effet indispensable d'alimenter la chaîne de leds avec une source 5V capable de fournir suffisamment de courant (comme un bloc secteur 5V/2A par exemple).

```
echo "#define NBRIMG `ls -1 | wc -l`"
echo "const unsigned char images[$((`ls -1 | wc -l`*48))] PROGMEM = {"
for i in *.txt
do
  echo "// $i"
  cat $i | egrep -v "^#" | sed "s/.*srgb(//;s/)/,/"
  echo -e ""
done
echo "};"
```

Nous avons ici plusieurs commandes permettant d'obtenir et de formater les informations dont nous avons besoin. `ls -1 | wc -l` par exemple utilise la bien connue commande `ls` pour lister les fichiers, mais redirige le résultat vers `wc -l` qui retourne le nombre de lignes, donc le nombre de fichiers et donc le nombre d'images.

Cette même commande est utilisée la ligne d'après, mais cette fois dans un calcul effectué par le shell, d'où le `$((` et le `))`. Nous multiplions le nombre de fichiers par 48 et obtenons ainsi la taille du tableau que nous devons déclarer. Le tout est utilisé pour créer la ligne de code en question.

Enfin, nous bouclons sur la liste des fichiers textes du répertoire courant pour prendre le contenu de chacun d'eux et l'envoyer à `egrep` qui éliminera toutes les lignes débutant par un `#`. Les autres lignes seront ensuite envoyées à `sed` qui élimine tout le texte jusqu'à `srgb(` puis la dernière parenthèse fermante.

À chaque tour dans la boucle, on utilise également le contenu de `$i` pour créer un commentaire avec le nom du fichier et ajouter une ligne vide entre chaque donnée d'image. Enfin, nous n'oublions pas d'envoyer `};` à la fin pour terminer proprement la déclaration/initialisation.

Notre script sera rendu exécutable avec `chmod +x ~/gopix.sh` et pourra alors être utilisé ainsi :

```
$ cd txt
$ ~/gopix.sh
#define NBRIMG 8
const unsigned char images[384] PROGMEM = {
// img004.png.txt
123,160,118,
180,190,184,
193,199,209,
202,209,225,
66,110,64,
180,179,162,
190,188,162,
214,212,183,
167,178,92,
161,173,102,
161,171,93,
189,199,102,
169,188,57,
158,179,54,
140,164,51,
126,148,41,

// img005.png.txt
230,219,218,
180,183,204,
180,174,183,
209,210,227,
[...]
22,36,17,
8,17,4,
28,47,42,
45,65,58,
41,51,36,
25,34,22,

};
```

Ça marche (bien sûr) ! Il nous suffit enfin de relancer le script en redirigeant la sortie dans un fichier en faisant `~/gopix.sh > /tmp/images.h`, et nous obtenons un fichier `images.h` dans le répertoire temporaire `/tmp` que nous pouvons transférer sur notre machine de développement Arduino pour l'intégrer à notre projet (menu ***Croquis*** et ***Ajouter un fichier***). Celui-ci apparaîtra alors sous la forme d'un onglet et vous pourrez corriger le petit défaut qui traîne en fin de fichier (la virgule et la ligne vide inutiles).

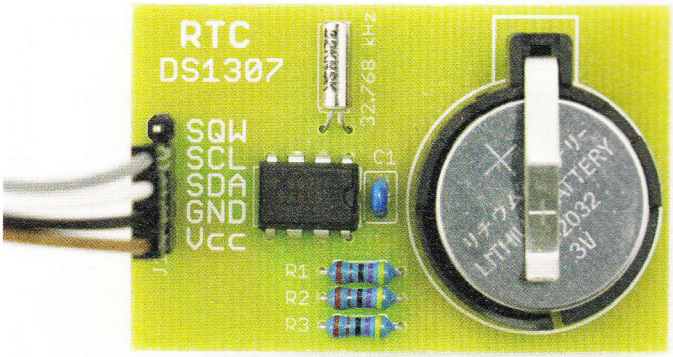
Il ne vous restera plus qu'à ajouter une ligne `#include "images.h"` dans votre croquis et voici vos données intégrées et prêtes à servir.

2. HORLOGE ET PROGRAMMATION

Je pense qu'à ce stade nous avons très certainement déjà des fonctionnalités plus riches que celles des modèles tout faits qu'on trouve dans le commerce. Nous pouvons maintenant étendre le fonctionnement de notre projet en utilisant davantage de données et surtout produire des séries d'images typiques de chaque divertissement télévisuel dont nous parlions en fin d'article précédent. On peut même pousser le vice jusqu'à créer des programmes TV complets en fonction du moment de la journée.

En parlant de journée et de moments, il semblerait que les modèles du commerce disposent de fonctionnalités très limitées en termes de programmation. Il n'est bien entendu pas question de laisser fonctionner le simulateur en permanence, ceci ne serait pas crédible, pas plus que le fait qu'il se déclenche tout seul la nuit tombée. Ce qu'il nous faut c'est une programmation digne de ce nom. Après tout, si vous comptez simuler l'habitat d'un légume de canapé, il faut être crédible et c'est généralement quelque chose comme 19h-23h en semaine et 10h-01h le week-end qui devra s'appliquer. Ceci est bien au-delà de la simple temporisation au choix avant extinction automatique des feux.

Il faut en revanche prendre en considération un autre point. Je l'ai dit dans le précédent article, la thématique est ici non seulement de parler technique, mais également de suivre l'évolution d'un projet et ceci passe par la phase que je qualifie de « overkill » ou « surextermination



Les modules RTC incluant le DS1307 (ou DS1338), un oscillateur à quartz, un support de pile bouton et quelques composants passifs, sont économiques et très faciles à trouver. Celui-ci est un peu ancien, mais des produits plus récents, au même prix (quelques euros) affichent un niveau de finition bien supérieur (platine, sérigraphie, support de fixation, compacité, etc.).

fonctionnelle ». Nous avons un projet et un montage riches, avons mis en œuvre des techniques avancées et avons même utilisé des outils de traitement de données pour produire des données pertinentes.

Pourtant à ce stade, la machine peut s'emballer car là, à cet instant précis, je suis en train d'imaginer non seulement l'intégration d'une horloge (RTC) dans le montage, ce qui reste raisonnable, mais également de faciliter la programmation en plaçant le « planning de diffusion » au format texte sur une petite carte SD de récupération. Bien entendu, ceci ouvre la voie à l'utilisation du support SD pour le stockage des images et par extension, à la conversion de formats vidéos classiques en un format personnalisé super-compact et pourquoi pas au passage de 16 leds à 48 (8×6) ou 192 (16×12)... Ceci ne ressemble plus à un simulateur de TV, mais commence à prendre l'apparence d'un player vidéo. Nous sommes hors cahier des charges !

La leçon ici est simple : il faut garder à l'esprit le but poursuivi par le projet en cours. Dans le cas contraire, vous vous engagez dans une course folle aux fonctionnalités avec le risque, si vous ne savez pas vous arrêter, d'oublier le projet initial sans pour autant faire aboutir la nouvelle création. Au final, vous n'aurez plus de simulateur de TV, mais ne terminerez peut-être pas non plus ce qui sera devenu un écran géant de 4800 leds WS2812b soulevant des problèmes (alimentation, synchronisation, rapidité du microcontrôleur, etc.) qui n'auraient jamais dû voir le jour. Mieux vaut garder son sang-froid, finir le simulateur et éventuellement, ensuite, partir de cette base à l'attaque d'un nouveau projet aux proportions bibliques.

Gardant cela à l'esprit, nous allons donc simplement nous en tenir à l'intégration du support pour une RTC sous la forme d'un module courant comprenant un DS1307, un oscillateur à quartz et une pile bouton CR2032. Ce type de modules se trouve un peu partout pour 2 ou 3 euros (voire moins si vous êtes patient). La connexion est simplissime : GND à la masse, Vcc au +5V, SDA sur SDA et SCL sur SCL.

Côté code, il suffira d'ajouter la bibliothèque DS1307RTC dans l'environnement Arduino via le *Library Manager* et d'intégrer les lignes suivantes en début de croquis :

```
#include <Wire.h>
#include <Time.h>
#include <DS1307RTC.h>
```

Un exemple est livré avec la bibliothèque afin d'initialiser la date et l'heure dans le DS1307 (**SetTime**) à partir des informations du PC de développement. Dès lors, l'horloge est fonctionnelle et continuera de fonctionner même déconnectée de l'alimentation, grâce à sa pile bouton.

Afin de signifier d'un problème avec le module RTC, on créé une fonction simple permettant de faire clignoter rapidement toutes les leds en rouge ou en orange :

```
void RTCerror(int err) {
  int r = 255;
  int g = 0;
  if(err!=0) g=100;
  while(1) {
    for(int j=0; j<PIX; j++) {
      pixels.setPixelColor(j, pixels.Color(r,g,0));
    }
    pixels.show();
    delay(75);
    pixels.clear();
    pixels.show();
    delay(75);
  }
}
```

Ce code est bloquant. Comprenez par là que si une erreur survient, qu'il s'agisse de l'absence de module (rouge, 0) ou d'une mauvaise date/heure (orange, 1), celle-ci déclenche le clignotement sans fin qui nécessite une réinitialisation du montage. C'est un choix délibéré de ma part, mais vous pouvez tout aussi bien décider d'un comportement différent, comme utiliser une ou deux leds standards en plus tout en laissant la simulation en route 24h/24 en guise de plan B.

On changera ensuite le corps de notre fonction **loop()** ainsi :

```
void loop() {
  // tableau de pointeurs
  fonction ftable[6] = {marcenciel, gmotif, pmotif, couleur, fondu_out, fondu_in};
  // variable de stockage de la date/heure
  tmElements_t tm;
  // Puis-je lire les infos ?
  if (RTC.read(tm)) {
    // RTC ok, on teste la plage horaire
    if(tm.Hour > 18 && tm.Hour < 23) {
      // Nous sommes entre 19h00 et 22h59, on simule
      ftable[random(0,sizeof(ftable)/sizeof(ftable[0]))](random(1,10));
    } else {
      // Hors plage, on éteint les leds et on pause 5s
      pixels.clear();
      pixels.show();
      delay(5000);
    }
  } else {
    // Nous ne pouvons pas récupérer la date/heure
    if (RTC.chipPresent()) {
      // Le DS1307 répond mais la date est invalide,
```

```
        // il n'est pas initialisé ou la pile a été retirée sans VCC
        RTCerror(1); // orange
      } else {
        // Le DS1307 est introuvable, aïe ! aïe !
        RTCerror(0); // rouge
      }
    }
}
```

3. ET LA SUITE ?

Notre support des plages horaires mériterait davantage de travail, mais cet article est déjà suffisamment long. Le test de l'heure est simpliste, mais toujours plus fonctionnel que ce que proposent les produits du commerce avec leur ridicule temporisateur de quelques heures.

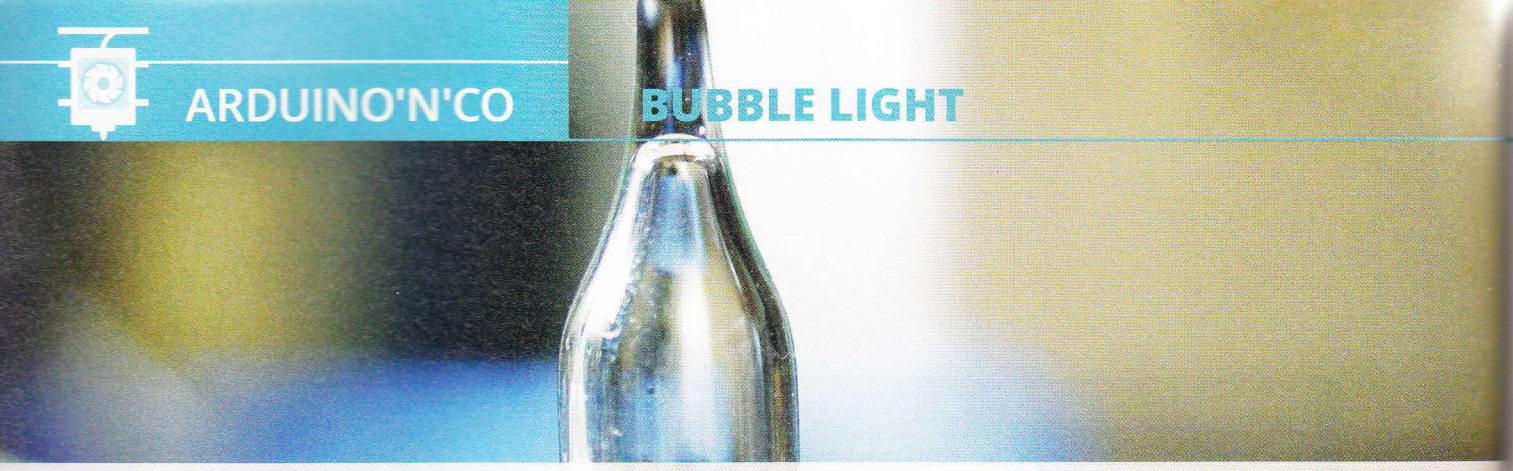
Si nous voulons quelque chose de plus étoffé, une succession et imbrication de **if/else** n'est sans doute pas une bonne solution. Personnellement, j'opterai pour la création d'un tableau à deux dimensions avec un jour de la semaine par ligne et sous forme de colonne : heure et minutes de mise en fonction et heure et minute d'extinction.

Une autre approche peut également être envisagée en définissant des valeurs de temps discrètes (comme des segments de 15 mn) et une succession de bits à 1 ou 0 pour chaque segment (comme sur les programmateurs rotatifs). Le code n'aura alors qu'à lire l'heure et tester le bit correspondant toutes les 15 minutes.

En termes d'évolutions possibles (en dehors de la petite folie concernant la carte SD) on peut envisager plusieurs choses :

- ajouter des relais et simuler une activité humaine via un contrôle de luminaire (oui, je parle des pauses « pipi » ou « argh, les cacahuètes ! ») ;
- élargir la matrice de leds (en restant raisonnable) pour non seulement s'approcher d'un format 16/9, mais en plus en profiter pour simuler le passage d'une personne devant l'écran (une colonne éteinte animée d'un côté à l'autre de la matrice) ;
- ajouter un élément aléatoire dans la gestion du moment de la mise en route et de l'arrêt du simulateur ;
- ajouter une interface permettant de configurer les plages horaires sans avoir à reprogrammer le microcontrôleur (liaison série et stockage en EEPROM interne par exemple) ;
- multiplier les effets disponibles bien sûr (explosions, transitions, silhouettes, motifs, défilements horizontaux ou verticaux, etc.) ;
- éventuellement trouver d'autres usages pour un tel système lumineux (pourquoi pas connecter un micro sur une entrée analogique et en faire un générateur d'effets lumineux).

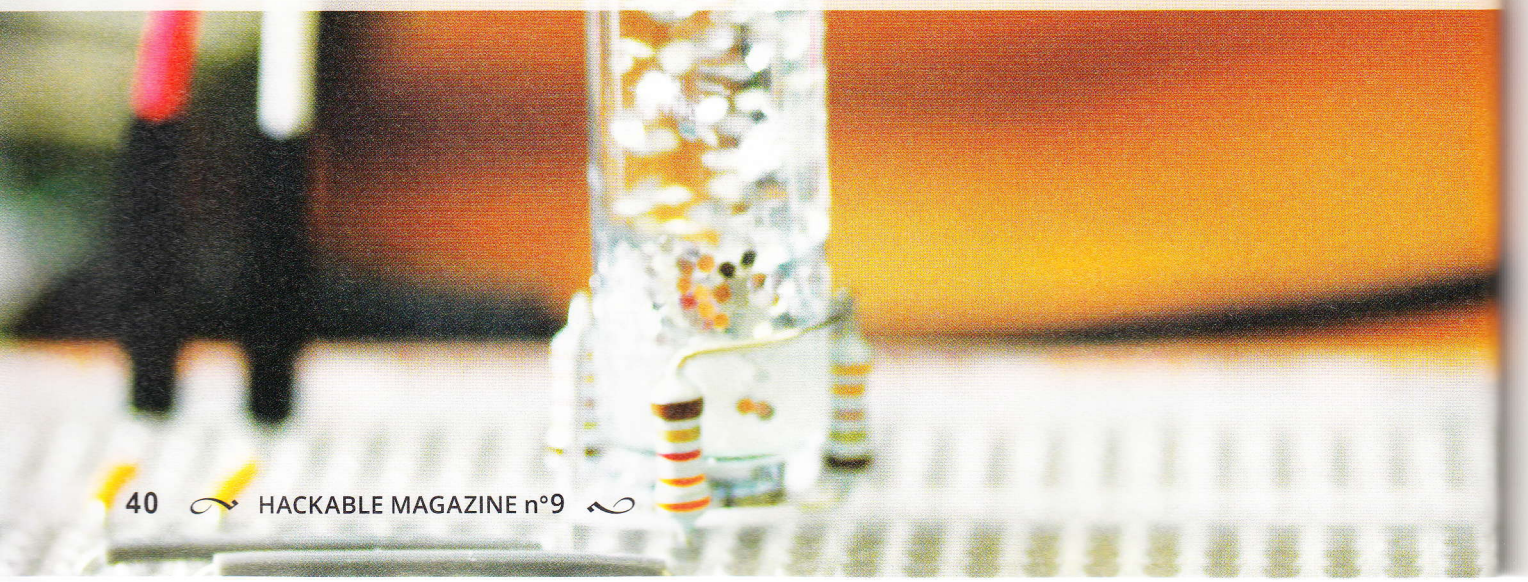
Dans tous les cas, si vous poursuivez et étoffez ce projet, n'hésitez pas à nous faire part de vos réalisations, expérimentations et évolutions via Twitter ou par mail afin que nous puissions en faire profiter tous les lecteurs... DB



CRÉER UNE NOTIFICATION ORIGINALE : FAIRE BOUILLIR UN LIQUIDE À 35°C

Denis Bodor

Tout, absolument tout, est source d'inspiration pour un esprit imaginatif. Prenez une décoration de Noël d'il y a plus de 50 ans par exemple et plus exactement une ampoule contenant un liquide qui possède la fantastique caractéristique d'entrer en ébullition à basse température. Pourquoi ne pas en faire un système de notification original ? Voici comment une petite idée amusante va nous conduire à un sujet bien plus sérieux et utile. Mais commençons par le début...



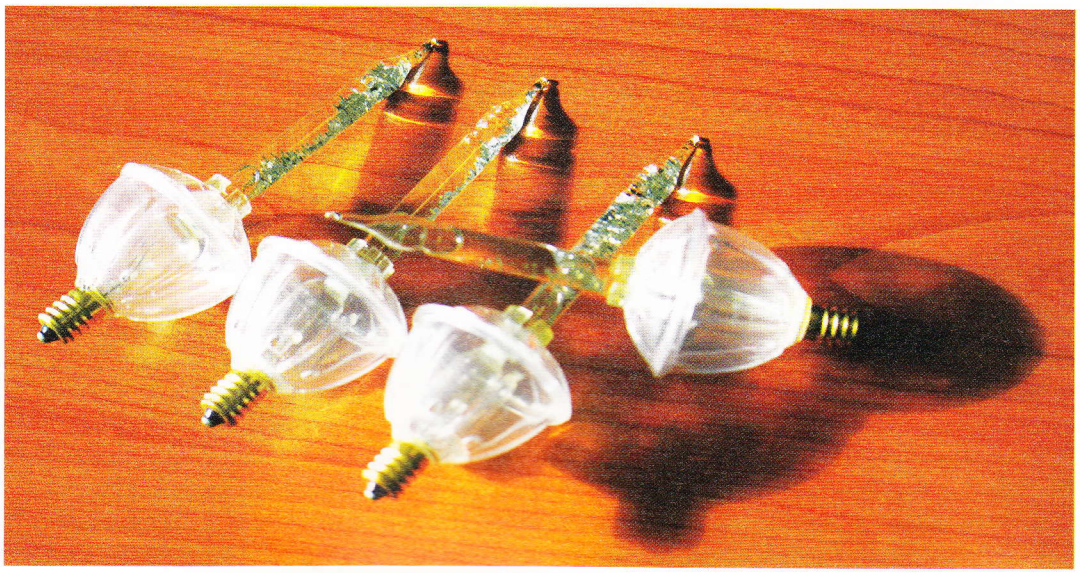
Les décorations de Noël actuelles reposent massivement sur les leds et des microcontrôleurs permettant de générer des effets sachant capter le regard et égayer les fêtes de fin d'année. Avant cette poussée technologique massive, ces décorations lumineuses se limitaient généralement à de simples ampoules 12V branchées en séries, éventuellement complétées d'une ampoule clignotante d'une simplicité extrême. Dans le bulbe de verre se trouvait un filament qui, chauffé, émettait de la lumière comme dans une ampoule classique, mais complété d'une lamelle métallique se déformant sous l'effet de la chaleur et coupant l'alimentation. En refroidissant, le contact était à nouveau établi et le cycle recommençait. La guirlande clignotait. Vous vous souvenez peut-être de ces pénibles (ou amusants) moments lors de la préparation du sapin où le but du jeu consistait à trouver l'ampoule clignotante et/ou celle qui avait grillée dans la guirlande. Mais ceci était avant les leds et avant que *Hackable* fasse que Noël devienne bimestriel.

À une époque plus reculée encore, un autre type de décoration était presque inconnu en Europe, mais relativement populaire dans les années 50 à 70 en Amérique du Nord : la *Bubble light*. C'est une invention britannique des années 20, mais c'est aux USA que le brevet déposé par Carl W. Otis (**http://www.google.com/patents/US2383941**) fut transformé en produit par NOMA, alors le plus gros fabriquant de décorations de Noël du pays.

1. LA BUBBLE LIGHT

Le principe de fonctionnement de cette antique décoration est fort simple : une ampoule de verre en forme de bougie était remplie de dichlorométhane (ou chlorure de méthylène) en laissant un espace occupé par de l'air à basse pression. Dans le fond de l'ampoule est placée une petite quantité de sel gemme (ou halite) non soluble dans le dichlorométhane, destiné à faciliter le fonctionnement de l'ensemble.

Le dichlorométhane est un solvant qui présente une caractéristique intéressante : à température ambiante, il se présente comme un liquide incolore, mais son point d'ébullition de situe autour de 40°C (au niveau de la mer, plus bas avec une pression moindre). Il est très volatil et est toxique



Voici 2 des 14 lampes de Noël dites « Bubble lights ». Celles-ci contiennent des paillettes dans l'ampoule de dichlorométhane, mais sont relativement sobres dans l'ensemble. D'autres modèles existent bien plus colorés.

BUBBLE LIGHT

Après avoir faire preuve d'une douceur et d'une prudence digne du barbare du donjon de Naheulbeuk avec un premier objet, on finit par développer une technique plus sûre et efficace.

par inhalation ce qui le rend bien plus dangereux que des produits comme le *white spirit* ou l'essence de térébenthine plus communément utilisés comme solvants domestiques.

Mais dans le cas de la *bubble light* ceci ne présente pas, en principe, un risque important, car le produit est hermétiquement contenu dans une ampoule scellée. Il est cependant possible que cette décoration ait vu sa popularité décroître en raison d'accidents domestiques (un solvant puissant ne fait généralement pas bon ménage avec un tapis lors d'un réveillon de Noël). Une autre raison possible concerne la méthode de fabrication qui semble-t-il consistait à créer la faible pression dans l'ampoule par succion à l'aide d'un tube tenu en bouche par l'ouvrière (dans le même ordre d'idée passablement douteuse, je vous conseille à l'occasion de rechercher les termes « *Radium Girls* » sur le Web).

La décoration en elle-même consistait à une telle ampoule attachée à une lampe à incandescence classique (~110V ou 12V). En plus d'illuminer l'ensemble, l'ampoule faisait chauffer le fond de la bougie de verre et le dichlorométhane entrait en ébullition. Les versions encore fabriquées aujourd'hui incluent tantôt dans le liquide des paillettes argentées qui sont agitées par l'ébullition. Il existe également des versions colorées des *bubble lights* soit par ajout d'un colorant dans le dichlorométhane, soit par utilisation d'un verre coloré ou peint.

Le dichlorométhane est également utilisé avec le même principe dans certaines décorations de juke-box des années 60 sous la forme de longs tubes de verre ou encore, d'une autre façon, pour mettre en évidence un certain nombre de principes physiques sous la forme du jouet « l'oiseau buveur ».

Les *bubble lights* ne sont presque plus produites sauf en petites quantités à destination d'acheteurs à la recherche de l'ambiance des Noëls de leur jeunesse Le *vintage* est de plus en plus tendance, au point que certains n'hésitent pas à faire du neuf avec du vieux... avec du neuf (vous savez des meubles façon années 60/70 en pseudo-bois (MDF ou contreplaqué) à 400€ alors que vous en trouvez des vrais en parfait état à 30€ à Espoir ou chez Emmaüs). Il ne serait guère étonnant donc qu'un beau jour ces décorations reviennent à la mode avec le statut de nouveauté de l'année...

Pour l'heure, la façon la plus sûre de se procurer ces décorations est, une fois n'est pas coutume, eBay. Les offres sont nombreuses et les prix très variables puisqu'il s'agit presque d'objets de collection sinon d'antiquités. En étant suffisamment patient et prudent, on arrive toutefois à s'en sortir à un prix presque acceptable (mais tout est relatif). Personnellement, j'ai craqué pour deux guirlandes de 7 *bubble lights*

$20,95 pièce, ce qui avec le port depuis le Colorado et les frais de douane monte le total pour 14 *bubble lights* à quelques 60€. Il s'agit de modèles embarquant des paillettes (*silver glitter*), chose que je n'avais pas remarquée à l'achat, et sont vendus en boîte encore à cette date (07/10) par *noveltylightsinc* sous la dénomination « *7 Foot Bubble Light Set - Christmas Tree -Clear with Silver Glitter- 7 Light Set* ».

Je fais mention de la présence de paillettes car, selon moi, l'effet est bien plus impressionnant avec un simple liquide transparent dans une ampoule. La faible pression interne et la différence de température entre le bas et le haut du conteneur suffisent à provoquer l'ébullition en tenant simplement l'objet du bout des doigts. En d'autres termes, on peut faire littéralement bouillir le contenu avec la simple chaleur corporelle et ça, c'est tout bonnement émerveillant !

De là à chercher le moyen de contrôler ce phénomène de façon électronique via une carte Arduino, il n'y a qu'un pas. La question est donc...

Note : Vous pouvez également vouloir fabriquer vos *bubble lights* si vous arrivez à mettre la main sur du dichlorométhane. Si tel est le cas, je vous recommande de faire un tour sur la chaîne YouTube du fort sympathique Clive Mitchell, alias *bigclivedotcom*, qui y est parvenu avec le verre de tubes fluorescents, une torche à souder à main chinoise, une seringue et beaucoup de ténacité. J'ajouterai également que la lecture de la fiche toxicologique de l'INRS à propos du dichlorométhane est tout simplement incontournable, ne serait-ce que pour être clairement être au fait des risques encourus ou vous dissuader d'une telle manipulation si vous n'êtes pas prêt à prendre les mesures de sécurité nécessaires (**http://www.inrs.fr/dms/inrs/FicheToxicologique/TI-FT-34/ft34.pdf**).

2. COMMENT OBTENIR DE LA CHALEUR EN ÉLECTRONIQUE ?

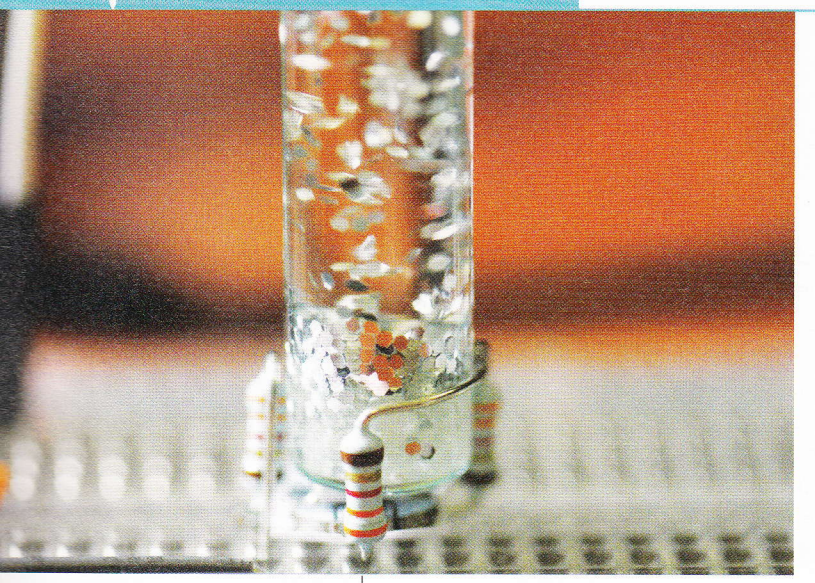
Disons-le de suite : l'utilisation de *bubble lights* même si c'est un sujet fort ludique et captivant, constitue ici principalement une introduction à une thématique plus générale que nous explorerons dans les deux articles qui suivent. Il s'agit de celui du contrôle d'un phénomène ou plutôt d'une grandeur physique à l'aide d'un système de régulation. Le présent article peut être vu comme une mise en situation dans le but d'appréhender un domaine et des principes qui pourront s'appliquer à une vaste gamme d'applications, de la régulation de température au contrôle de moteurs.

En ce qui concerne le présent article la question qui se pose à nous est simple puisque nous cherchons simplement à produire un peu de chaleur dans une mesure très modeste. Il ne s'agit pas d'atteindre 250°C, ni même 90°C, mais simplement une température équivalente à celle de la peau, soit environ 30°C à 35°C.

Comment un composant électronique, quel qu'il soit (ou presque) produit-il de la chaleur ? Le plus naturellement possible, il dissipe généralement un excédent d'énergie par effet Joule. Le composant le plus représentatif d'un tel phénomène est bien sûr la résistance.

L'objet même d'une résistance est de résister au passage d'un courant électrique. Les électrons (porteurs de charges) qui se déplacent dans un matériau interagissent avec les atomes qui freinent leur déplacement. Pour faire circuler une quantité d'énergie précise, on fournit un courant plus important et l'excédent découlant de l'interaction avec les atomes du matériau sera dissipé en énergie thermique. C'est l'effet Joule et c'est précisément ainsi qu'une résistance d'une valeur donnée permet le passage d'un courant donné sous une tension donnée, selon la formule simple U=R*I : la loi d'Ohm (U est la tension en volts, R la résistance en ohms et I l'intensité du courant en ampères).

Vous l'avez compris, ces petits composants à rayures que sont les résistances, chauffent même si dans un usage normal vous ne vous en rendez pas vraiment compte. C'est d'ailleurs pour cette raison que les résistances sont conçues pour une puissance précise (le plus souvent 1/4 de watt) qui correspond à la quantité d'énergie qu'elles peuvent dissiper avant que leurs caractéristiques ne changent. Le calcul de la puissance devant être dissipée par une résistance se fait avec la formule P=U*I, avec la puissance en watts (et par extension $P=R*I^2$).



Le montage chauffant composé de trois résistances 1/4 de watt ne provoque qu'une ébullition timide difficile à capturer avec un appareil photo. Il ne s'agit cependant pas seulement de convection, ce sont bien des bulles de vapeur de dichlorométhane qui animent les paillettes.

L'idée ici, qui est inspirée d'une des vidéos de *BigClive*, est de simplement utiliser des résistances 1/4 watts standards en guise de résistance chauffante de fortune. En fouillant dans mon stock (acheter des lots vraiment pas chers sur eBay permet d'augmenter rapidement votre collection et tantôt avec des valeurs assez exotiques), j'ai donc choisi d'utiliser des résistances de 33,2 ohms car :

```
U = R*I
5 = (33,2*3)*I
5/(33,2*3) = I = 0,0502 = 50 mA

P = U*I
P = 5*0,0502 = 0,2510 W
P = 251 mW ~= 1/4 W
```

Trois résistances en série à placer autour de la base de la *bubble light* permettront la circulation d'un courant d'environ 50mA sous 5 volts pour une puissance dissipée totale d'un peu plus d'un quart de watt. Nous sommes donc bien en dessous des limites des composants (chaque résistance dissipera en effet que (5/3)*0,0502 watts).

Les cristaux à la base de l'ampoule sont, selon toutes vraisemblances, du sel gemme. C'est la forme minérale pure du chlorure de sodium (qui n'est qu'une composante du sel de cuisine). Ceci semble favoriser l'apparition et la régulation des bulles dans le tube. Juste au-dessus se trouvent les paillettes métallisées de forme hexagonale qui sont agitées lors de l'ébullition.

Il n'est, en revanche, pas si simple de calculer la température à partir des données à notre disposition (tension, puissance, courant, résistance, etc.). Énormément de paramètres entrent en ligne de compte comme la température extérieure, la forme du matériau résistif, le coefficient de diffusivité thermique, le flux d'air autour du montage, etc. Pour les résistances chauffantes, tous ces calculs sont fournis par le fabriquant dans la documentation du composant (*datasheet*) sous la forme d'un rapport (ou d'une courbe) température/puissance. Avec les résistances type conducteurs ohmiques utilisées en électronique (les bébêtes à rayures) la donnée fournie est encore plus simple puisqu'il s'agit de la puissance maximum à ne pas dépasser (le fameux 1/4 de watt).

En vérité, le plus simple ici est tout simplement d'essayer. Il sera toujours possible de choisir des résistances de valeurs différentes (un peu plus de 10 ohms) afin de s'approcher au mieux du 1/4 watt et donc générer davantage de chaleur tout en restant sous le maximum. Gardons cependant à l'esprit que ces résistances ne sont absolument pas destinées à cet usage.

Une fois les résistances assemblées en série de manière ingénieuse de façon à utiliser leurs pattes comme support de fixation de la *bubble light* et à la platine à essais, il nous suffit d'alimenter tout cela pour valider la théorie.
Il faut un petit temps pour que la chaleur s'accumule au bas de l'ampoule, mais une fois l'ébullition démarrée le processus se poursuit de façon stable, mais néanmoins assez calme.

Si comme moi vous n'avez trouvé, à un prix intéressant, que des véritables *bubble lights* en guirlande, l'étape pénible et risquée consistera à retirer l'ampoule de dichlorométhane de son support. Dans mon cas, celles-ci étaient collées à la colle epoxy et il m'a été nécessaire

d'user de la pince pour les extraire avant de gratter méticuleusement et prudemment le résidu de colle sur le verre. Sur les 14 ampoules, aucune n'a été détruite ou n'a fuit, mais l'une d'elles présente un impact dans le verre qui me dissuade totalement d'en faire usage. Vous aurez peut-être plus de chance que moi en trouvant des ampoules seules à un prix raisonnable... et sans paillettes.

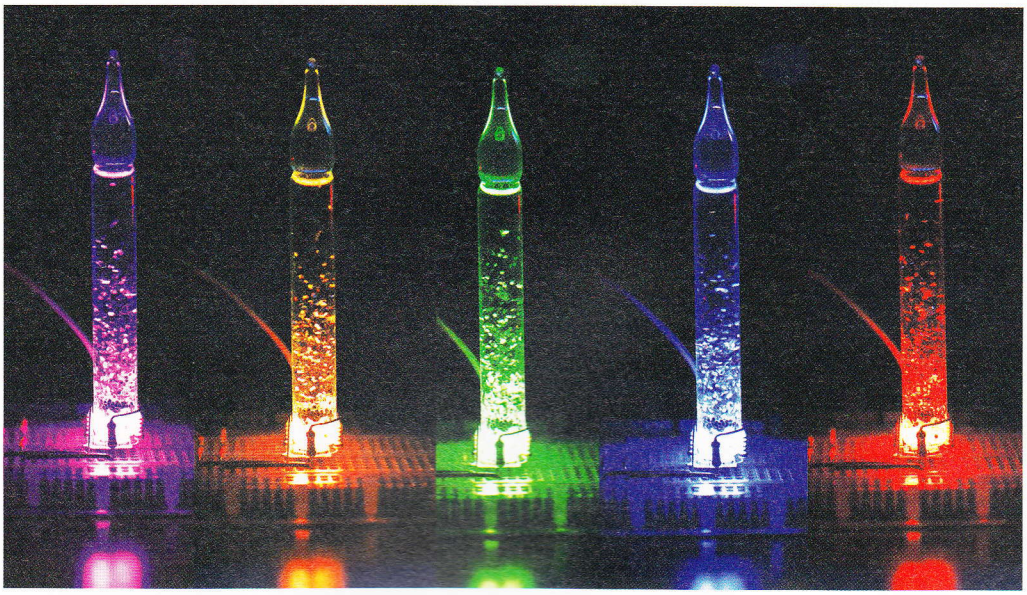
3. RÉSULTAT, PROBLÈMES, LIMITATIONS ET CONCLUSION

Plusieurs choses découlent de ce premier essai, certaines positives d'autres non. En premier lieu, oui il est possible d'utiliser la chaleur de simples résistances pour chauffer une *bubble light* et déclencher l'ébullition. On peut même s'amuser à placer une led type WS2812b à sa base pour améliorer l'effet et en plus augmenter la chaleur.

Un problème cependant assombrit le tableau : 50 mA c'est beaucoup et en particulier plus que ne peut en fournir une broche en sortie d'une carte comme Arduino (40 mA). Pour contrôler l'ébullition, il faudra donc utiliser un composant supplémentaire comme un transistor ou un MOSFET capable de contrôler davantage de courant.

L'ébullition elle-même n'est pas très intense. Ceci est parfaitement compréhensible, car en y regardant de plus près, l'ampoule de dichlorométhane est initialement fixée contre une ampoule à incandescence qui, une fois extraite également à la pince, s'avère libellée « 120V 5W ». 5 watts ce n'est pas beaucoup pour ce type d'élément (les ampoules de réfrigérateurs ou de fours sont de 10, 15 ou 25 watts), mais cela générera sans doute bien plus de chaleur qu'une main ou trois modestes résistances de quelques dizaines d'ohms. Pour bien faire, il faudra alimenter une de ces ampoules en ~110V et mesurer la température pour obtenir une valeur qu'on considérera comme notre maximum absolu.

Pour chauffer notre dichlorométhane de façon à provoquer une vive réaction, mais de façon contrôlée, il est préférable d'utiliser un composant et un montage dédiés : une résistance chauffante, pilotée par un MOSFET, le tout contrôlé en PWM sur la base d'une mesure de température via un capteur DS18B20. Voici précisément ce qui nous catapulte dans le monde merveilleux de la régulation de température, à même de nous enseigner énormément de nouvelles choses sympathiques que nous pourrons réutiliser dans bien des contextes. Ceci est l'objet de l'article qui suit... **DB**

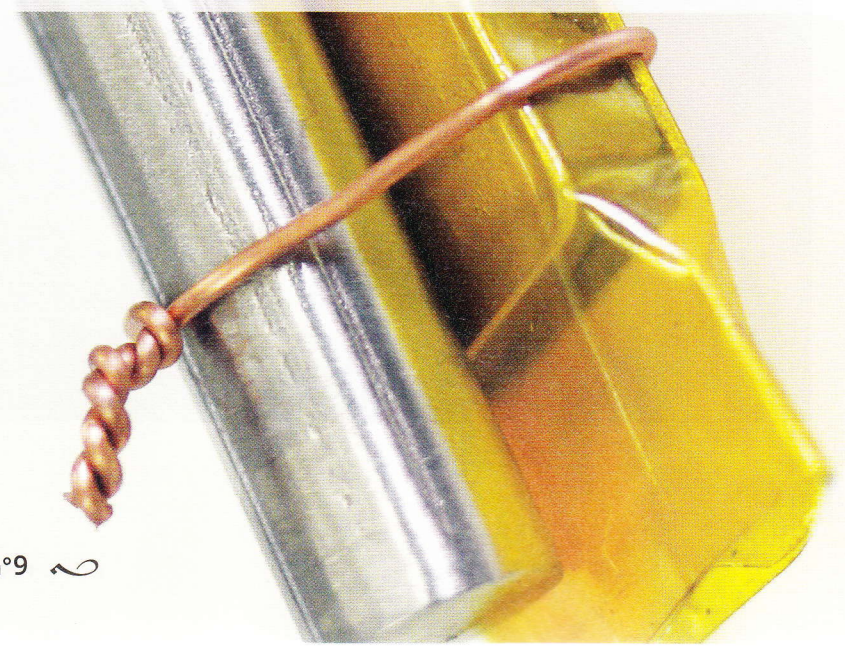


En ajoutant simplement une led WS2812b à la base de l'ampoule de dichlorométhane, non seulement on obtient un effet très intéressant, mais en plus on augmente la température et donc l'intensité de l'ébullition. Ajouter un contrôle de couleur par la carte Arduino et vous obtenez une base très originale pour un effet lumineux.

CONTRÔLEZ UN ÉLÉMENT CHAUFFANT ET SURVEILLEZ LE FONCTIONNEMENT AVEC PROCESSING

Denis Bodor

Le mot Processing ne doit pas vous être totalement inconnu. Il s'agit en effet d'une des bases de développement d'Arduino et plus exactement la source d'inspiration de son environnement de développement. Dans le précédent article, nous nous sommes heurtés à une problématique consistant à contrôler avec précision le chauffage d'un élément. Nous allons ici voir comment Arduino et Processing nous permettront de découvrir le monde merveilleux de la régulation.



Processing, anciennement Proce55ing, est un environnement de développement écrit en Java doublé d'un ensemble de bibliothèques, destiné à la création numérique. Il fonctionne, à l'instar de l'IDE Arduino, aussi bien sous Windows, Mac OS X et GNU/Linux. Jusqu'à l'arrivée de la version 3.0 le 1er octobre de cette année, l'environnement Processing ressemblait énormément à l'IDE Arduino. Cette nouvelle version apporte un certain nombre de changements ergonomiques importants, mais l'ensemble fonctionne d'une manière qui devrait, dans les grandes lignes, vous permettre de rapidement transposer votre expérience Arduino.

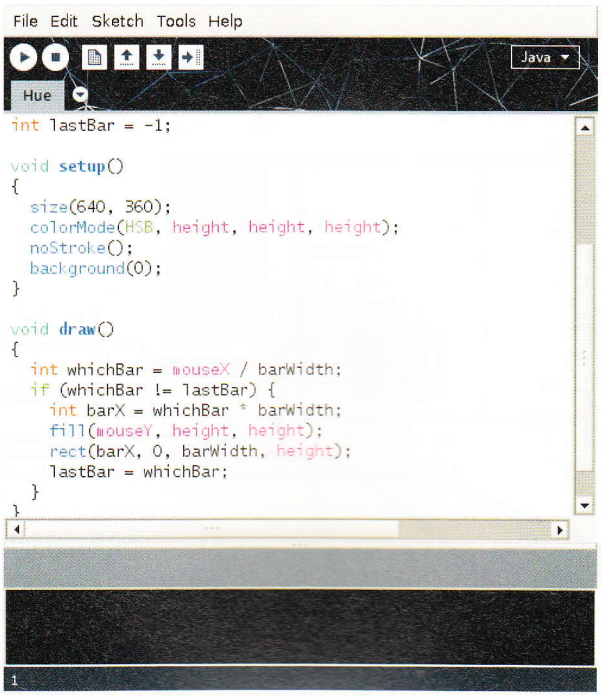
Tout comme Arduino dans le monde de l'électronique numérique, Processing s'est fixé un principe majeur qui dirige tout son fonctionnement : la simplicité. Arduino permet de rapidement obtenir un résultat satisfaisant dans la découverte de la programmation de microcontrôleurs sur la base du langage C/C++ édulcoré par certaines « facilités » syntaxiques. Processing fait de même, mais sur la base du langage Java et dans le domaine de la création graphique.

En quelques lignes de code, on peut donc obtenir rapidement quelque chose qui fonctionne et surtout qui fait ce qu'on lui demande. Exemple :

```
void setup(){
  size(300, 300);
  background(0);
}

void draw(){
text("Bonjour monde!", width/2, height/2);
}
```

Ceci vous rappelle quelque chose ? C'est normal, Processing sert de base à Wiring qui lui-même est le point de départ logiciel d'Arduino. En toute logique, on retrouve donc une philosophie similaire. La fonction **`setup()`**, par exemple, est destinée exactement au même usage qu'avec Arduino, mais en lieu et place de **`loop()`** nous trouvons la fonction **`draw()`** qui, elle aussi, sert un dessin identique : servir de boucle principale au croquis ou *sketch* (oui, la terminologie aussi est identique).



L'interface de développement de Processing 2.2.1 est très similaire à celle de l'IDE Arduino. Il en va de même pour la syntaxe du langage lui-même malgré le fait que celui-ci repose sur Java et non C/C++ comme Arduino.

Ce croquis crée une fenêtre, la remplit de noir en guise de fond, et trace le texte spécifié à la moitié de sa hauteur et largeur. C'est le **`Blink`** de Processing...

1. NOS MOUTONS

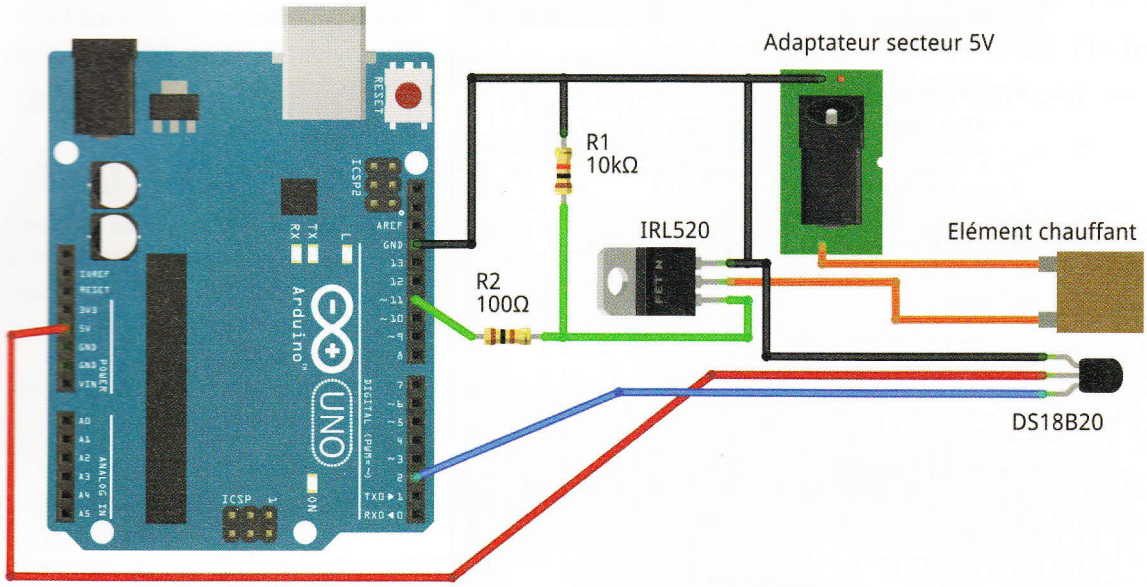
Revenons maintenant au problème qui nous préoccupe et à l'évolution de notre première solution. Notre chauffage de *Bubble light* s'est transformé en problématique de régulation de température. Les composants mis en œuvre seront les suivants :

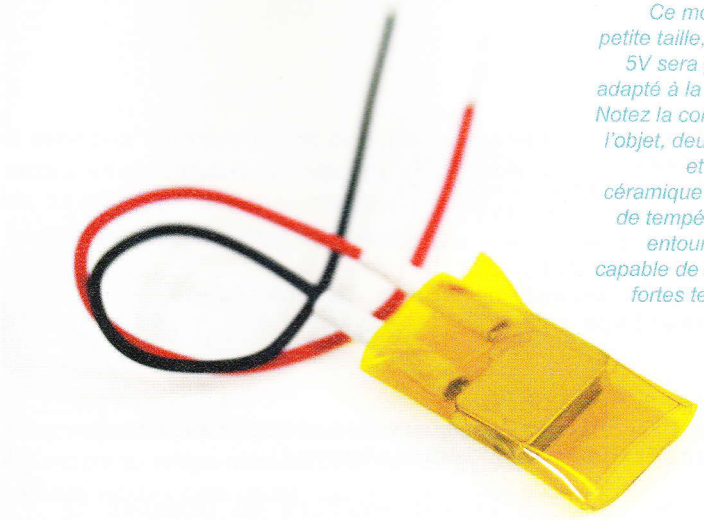
- Une résistance chauffante 5V : le principe de fonctionnement est totalement identique à celui d'une résistance classique, mais l'élément est conçu pour produire

Le montage de notre système de contrôle de température est relativement simple : un MOSFET-N IRL520 est commandé par le port 11 de la carte Arduino pour alimenter la résistance chauffante et le capteur DS18B20 est connecté au port 2. R1 est une résistance de rappel et R2 limite le courant sur la grille du MOSFET.

de la chaleur de façon efficace. On fait simplement circuler un courant dans le composant et celui-ci chauffe à une température de... mystère ! Le composant utilisé ici en test, vendu sur eBay par « pedgington » (UK) pour quelques 4 euros, est censé fournir une température de 40°C. Mais il ne s'agit que d'un élément chauffant en céramique et plus exactement un PTC pour *Positive Temperature Coefficient*, un matériau dont la résistance va augmenter en même temps que sa température. La description de l'objet s'accompagne simplement d'une mention « *5V, 40° ± 10°, 0.3 - 1 Watt* ». Normal, le composant lui-même n'est pas régulé, sa température dépendra du courant qui y circule et impactera sa résistance qui, elle, influera sur le courant, etc. Pour obtenir une température arbitraire, le composant seul ne suffira pas.

- Un capteur de température DS18B20 avec lequel nous avons déjà fait connaissance, sur plateforme ESP8266, dans le précédent numéro. Interfacé en 1-wire ce capteur peut s'attacher très facilement à n'importe quelle carte Arduino et mesurer une température précise. En plaçant la sonde à proximité de la résistance chauffante, nous obtenons alors le retour d'information qu'il nous faut.
- Un MOSFET très courant : IRL520. Il n'est pas possible d'alimenter directement la résistance à partir d'un port de la carte Arduino, car celle-ci ne peut fournir le courant nécessaire. 1 watt tel que précisé dans la description de la résistance, en 5 volts, cela nous donne, P=U*I, 1=5*I, 1/5=I, 200 mA. Même l'alimentation +5V de la carte suffirait à peine à délivrer le courant nécessaire. Nous allons donc depuis la carte Arduino, piloter le MOSFET qui contrôlera l'alimentation de la résistance provenant d'un bloc secteur 5V/2A. Pour régler la puissance délivrée, nous utiliserons simplement une sortie analogique (PWM) et pourrons ainsi, si nécessaire, ajuster l'alimentation à un niveau compris entre 0 et 255 (0-100% du rapport cyclique, voir *Hackable n°1* pour plus de détail sur la technique de PWM ou modulation en largeur d'impulsions).





Il existe plusieurs types d'éléments chauffants. Ce modèle de très petite taille, alimenté en 5V sera parfaitement adapté à la bubble light. Notez la construction de l'objet, deux électrodes et un matériau céramique à coefficient de température positif entouré de Kapton capable de résister à de fortes températures.

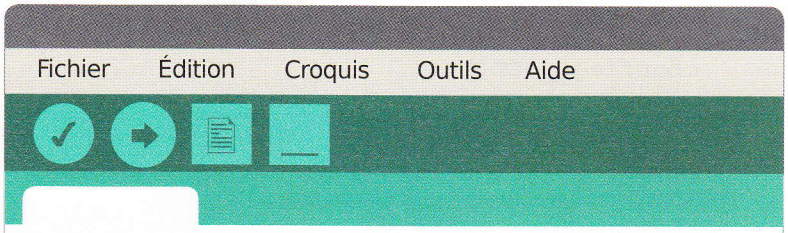
Côté matériel, tout est relativement simple. Personnellement, j'ai préféré vérifier les données et pseudo-caractéristiques concernant la résistance chauffante. Il m'a suffi de connecter le composant à une alimentation de laboratoire et de régler la tension sur 5 volts. J'ai ensuite graduellement augmenté le courant jusqu'à arriver à un régime de régulation de tension. Il apparaît ainsi que cette résistance chauffante laisse circuler un courant de 225mA, dissipant effectivement autour de 1W en 5V (1,125 plus précisément). Il est possible de faire une mesure équivalente avec un simple multimètre.

Le travail de la carte Arduino et du croquis qu'elle contient consistera en l'algorithme suivant :

- mesurer la température de la résistance grâce au DS18B20 ;
- comparer cette valeur à la température que nous souhaitons (la consigne) ;
- si la valeur est sous la consigne, on active le MOSFET et on chauffe ;
- si la valeur est au-dessus de la consigne, on coupe l'alimentation de la résistance.

Nous avons décidé d'utiliser une sortie « analogique » (PWM) et, de ce fait, la notion d'activer/couper n'est pas nécessairement tout ou rien. Afin d'affiner le fonctionnement, nous pouvons tout aussi bien décider que la mise en chauffage correspond à un rapport cyclique de 75% (**192**) et l'arrêt à 25% (**64**). Mais oubliez cela pour l'instant, ce n'est qu'un exercice pour se rendre compte que le réglage manuel n'est pas la meilleure solution.

Notre croquis sera donc le suivant :




```
#include <OneWire.h>
#include <DallasTemperature.h>

// le bus est sur la broche 2
#define ONE_WIRE_BUS 2

// On contrôle avec la broche 11
// On choisi une sortie PWM au cas où
#define OUTP 11

// Déclaration du bus 1-wire
OneWire oneWire(ONE_WIRE_BUS);
// déclaration des capteurs
// (il pourrait y en avoir plusieurs)
DallasTemperature sensors(&oneWire);

// Fonction chauffage
void chauffage_on() {
  // on peut jouer avec la PWM si nécessaire
  analogWrite(OUTP, 255);
}

// Arrêt chauffage
void chauffage_off() {
  analogWrite(OUTP, 0);
}
```

```
void setup(void) {
  Serial.begin(115200);
  // démarrage capteurs
  sensors.begin();
  // broche 11 en sortie
  pinMode(OUTP, OUTPUT);
  // par défaut on coupe le chauffage
  chauffage_off();
}

void loop(void) {
  // Demande de lecture des températures
  sensors.requestTemperatures();

  // On affiche la température du premier
  // capteur (le seul dans notre cas)
  Serial.print(sensors.getTempCByIndex(0));
  // si on est au-dessus de la consigne
  if(sensors.getTempCByIndex(0) > 39) {
    // affiche valeur bidon pour le graph
    Serial.println(":34");
    // on  ne chauffe pas
    chauffage_off();
  // si on est en dessous
  } else if(sensors.getTempCByIndex(0) < 39) {
    // affiche valeur bidon pour le graph
    Serial.println(":45");
    // on chauffe
    chauffage_on();
  }
  // si on est à la consigne,
  // on ne change rien,
  // dans un cas comme dans
  // l'autre la température
  // changera en plus
  // ou en moins.
}
```

Processing 3 est depuis le 1er octobre la nouvelle version stable de l'environnement. Parmi les nouveautés, une belle refonte de l'interface ajoutant énormément de fonctionnalités, dont la vérification du code à la volée avec affichage des erreurs lors de la saisie, ou encore la complétion du code grâce au raccourci Ctrl+espace.

Nous utilisons également le moniteur série pour afficher une liste de données avec la température mesurée et deux valeurs arbitrairement choisies correspondant à l'activation et la désactivation de l'élément chauffant. 34 et 45 n'ont pas de signification précise, mais correspondent à une astuce bien utile pour l'autre croquis... dans Processing 3 !

2. LES LISTES C'EST BIEN, UN GRAPHIQUE C'EST MIEUX

En mettant en route le montage et en surveillant les données, nous verrons s'afficher une liste de valeurs avec un intervalle correspondant au temps nécessaire à chaque mesure (pas loin d'une seconde). Nous pourrions noter ces valeurs dans un tableau ou même envisager de copier/coller tout cela pour produire un graphique à l'aide d'un tableur. Mais l'énoncé même de la méthode parait des plus ennuyants...

Nous allons donc nous tourner vers Processing de façon à écrire un croquis, côté PC, qui recevra les lignes envoyées par la liaison série de l'Arduino, extraira la température et nos valeurs arbitraires et tracera pour nous, en temps réel, un graphique de l'évolution du système.

Voici le croquis utilisé :

Fichier Édition Croquis Outils Aide

```
import processing.serial.*;
// objet représentant le port série
Serial lePort;

// position courante sur l'axe x
float xPos = 0;
float num;
float numcmd;

void setup () {
  // taille de la fenêtre
  // (pas d'accélération graphique, on utilise le moteur
  // de java. Il semblerait que mon pilote graphique sous
  // Linux pose problème avec les rendus P3D et P2D)
  size(1600, 1000, JAVA2D);

  // Afficher la liste des ports séries détectés
  printArray(Serial.list());
  // Ceci peut être plus simple sous Windows/Mac :
  //String portName = Serial.list()[5];
  //lePort = new Serial(this, portName, 115200);

  // Ouverture du port série
  lePort = new Serial(this, "/dev/ttyUSB1", 115200);
```

```
  // fond noir + effacement
  background(#000000);
  // vert
  stroke(#00FF00);
  // lignes horizontales
  line(0, height-410, width, height-410);
  line(0, height-400, width, height-400);
  // jaune
  stroke(#FFFF00);
  // Cette ligne est la consigne 39degC x 10
  line(0, height-390, width, height-390);
  // re-vert
  stroke(#00FF00);
  line(0, height-380, width, height-380);
}

// Equivalent au loop() d'Arduino
void draw () {
  // configuration de la police de caractères
  PFont f;
  f = createFont("Arial",12,true);
  textFont(f);

  // on retrace les lignes horizontales (repères)
  // vert
  stroke(#00FF00);
  line(0, height-410, width, height-410);
  line(0, height-400, width, height-400);
  // jaune pour la consigne
  stroke(#FFFF00);
  line(0, height-390, width, height-390);
  stroke(#00FF00);
  line(0, height-380, width, height-380);

  // rouge pour les températures
  stroke(#FF0000);
  point(xPos, height - num*10);
  // cyan pour la valeur PWM (commande)
  stroke(#00FFFF);
  point(xPos, height - numcmd*10/4);

  // noir
  stroke(#000000);
  // remplissage des formes en noir
  fill(#000000);
  // rectangle noir plein pour effacer le tracé
  // avant la position courante
  rect(xPos+1, 0, 42, height);
  // pas de remplissage
  noFill();

  // ajout texte
  fill(#FF0000);
  text(str(num),xPos+8, height - num*10);
  fill(#00FFFF);
  text(str(int(numcmd)),xPos+8, height - numcmd*10/4);
  noFill();
}
```

```
// Quelque chose se passe sur le port série
void serialEvent (Serial lePort) {
  String maChaine = null;

  // On récupère toute la ligne
  maChaine = lePort.readStringUntil(10);
  // si ce n'est pas une ligne vide
  if (maChaine != null) {
    // récupération de la position du ":"
    int p = maChaine.indexOf(":");
    // récupération chaîne température
    String temp = maChaine.substring(0, p);
    // récupération chaîne pwm/commande
    String cmd = maChaine.substring(p+1);

    // chaîne température en valeur à virgule flottante
    num=float(temp);
    // chaîne pwm/commande en valeur à virgule flottante
    numcmd=float(cmd);
    // affichage en console
    print(num);
    print("    --    ");
    println(numcmd);

    // Si on arrive au bord de la fenêtre (à droite)
    if (xPos >= width) {
      // On revient au début (à gauche)
      xPos = 0;
    } else {
      // sinon, on avance la position courante d'un pixel
      // à chaque nouvelle ligne reçue sur le port série
      // Le tracé se fait dans draw()
      xPos++;
    }
  }
}
```

Arduino

Les commentaires intégrés dans le code et les connaissances que vous avez certainement acquises de la programmation Arduino devraient vous permettre de rendre cela parfaitement clair. Il s'agit de Java et non de C/C++, mais vous remarquerez sans difficulté que bon nombre de fonctions, de méthodes ou de types de données sont identiques. **`print()`** et **`println()`** affichent simplement leurs messages dans la console et non sur le port série, les manipulations de chaînes (**`String`**) sont identiques, tout comme les déclarations de variables, les opérations arithmétiques, etc.

En dehors du fait de troquer **`loop()`** contre **`draw()`** la particularité du présent croquis concerne surtout la gestion du port série. Avec Arduino, la question ne se pose pas quant au choix du port. En effet, la plupart des cartes n'en possèdent qu'un et surtout les caractéristiques matérielles ne changent pas. Sur PC ou Mac, la configuration matérielle n'est pas figée, le simple fait de brancher une carte Arduino UNO ajoute un port série. Le lien avec le matériel et le système d'exploitation ne peut ainsi être identique sur toutes les plateformes.

Pour mesurer au plus juste la température de l'élément chauffant, rien de plus simple : on attache les deux à l'aide d'un fil en cuivre (très bon conducteur de chaleur). Un rien bricolo, mais ça marche parfaitement...

Dans mon cas, j'utilise Processing 3 sous GNU/Linux. La désignation du port série se fait donc sous la forme `/dev/ttyUSB*` ou `/dev/ttyACM*`. Ayant connaissance du nom exact du fichier représentant le port série lié à l'Arduino, il me suffit d'utiliser le chemin complet dans `Serial()`. Notez à ce propos une spécificité Processing provenant de Java : pour créer (instancier) un nouvel objet, comme ici `lePort`, il faut utiliser l'opérateur `new`.

En fonction du système d'exploitation cependant, il sera sans doute plus simple d'ouvrir le port série adéquat après avoir obtenu la liste de ceux présents (`Serial.list()`) puis récupéré son nom en spécifiant sa position dans la liste (`[5]` ici en commentaire). On utilisera ensuite ce nom pour ouvrir le port avec `new Serial()` qui, dès lors, sera représenté par l'objet `lePort` (équivalent au `Serial` avec Arduino).

L'autre point concernant le port série concerne la manière de réagir à l'arrivée de données. Ceci est géré par une routine `serialEvent()` appelée automatiquement dès que des données peuvent être lues. Pour traiter les données, tout ce que nous avons à faire est d'écrire cette fonction qui collectera les données avec `readStringUntil()` permettant d'obtenir une chaîne jusqu'à la première occurrence d'un caractère précis (ici `10'`, le caractère LF comme *Line Feed*, le saut de ligne). Notez que l'utilisation de `serialEvent()` est également possible avec Arduino, mais vous devrez appeler la fonction dans `loop()` alors qu'avec Processing ceci se fait automatiquement.

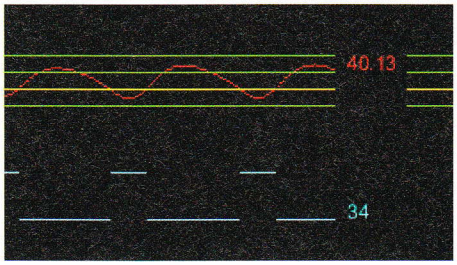
Le reste du croquis concerne la manipulation de la fenêtre graphique, le remplissage de zone, le tracé de lignes, de points et de textes. L'ensemble permet d'obtenir une fenêtre de 1600×1000 pixels avec en rouge la température mesurée, en cyan l'activation/désactivation du chauffage (vous comprenez maintenant le pourquoi des valeurs arbitraires), en jaune la consigne à 39°C et en vert quelques repères visuels espacés d'un degré.

Notez qu'il ne vous est pas possible d'accéder au port série en même temps depuis l'IDE Arduino et Processing et donc ne pouvez pas :

- utiliser le moniteur série Arduino en même temps que votre croquis Processing ;
- reprogrammer la carte Arduino avec le croquis Processing en marche ;
- ou lancer le croquis Processing avec le moniteur série Arduino ouvert.

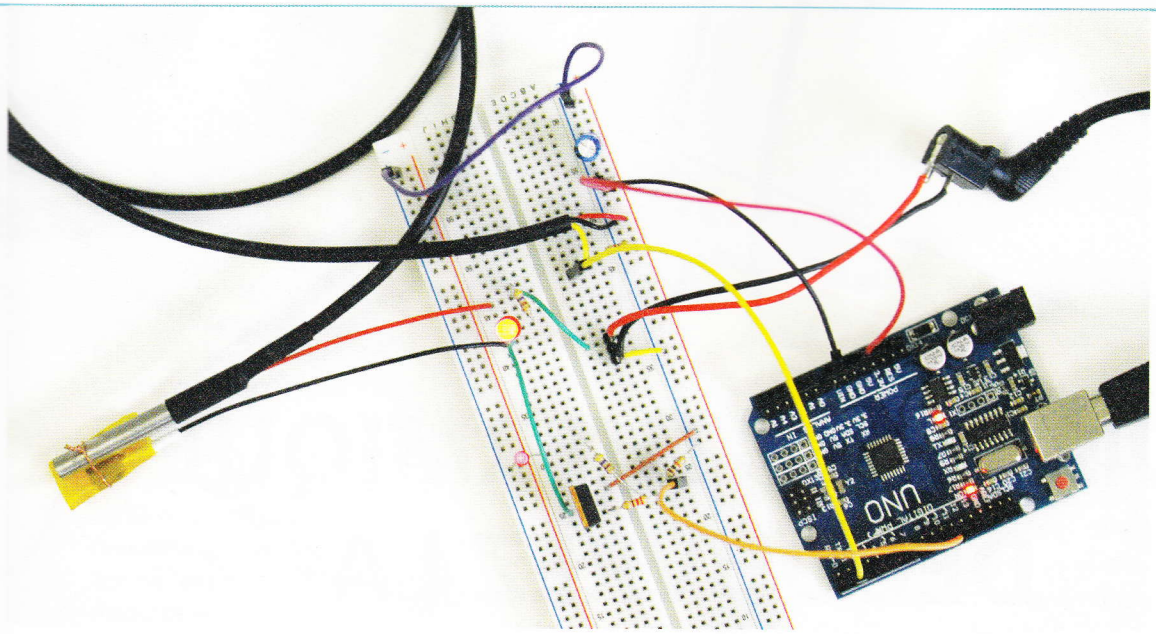
3. RÉSULTAT ET DÉCEPTION

Voici ce que le graphique produit va vous afficher (il est ici redimensionné pour les besoins de la mise en page) :



La température va se mettre à osciller au rythme des activations/désactivations de l'élément chauffant. Nous avons un décalage entre la mesure et la commande, ou plus exactement une importante inertie thermique. Au moment où la température passe sous 39°C l'élément se met à chauffer, mais l'effet n'est pas immédiat et la température va continuer à descendre un certain temps. Lorsque celle-ci passe au-dessus de la consigne, le chauffage est coupé, mais la température grimpe encore de plus d'un degré et l'ensemble ne se refroidit qu'ensuite. Le système oscille sans jamais rester à la consigne...

Il ne s'agit pas d'un problème inhérent à notre montage, nos croquis ou à un composant défectueux, mais tout simplement d'un phénomène physique naturel. Énormément de systèmes fonctionnent de la sorte : votre four, la plupart des



Vue d'ensemble du montage de test. L'élément chauffant est alimenté par un bloc indépendant et contrôlé par le MOSFET piloté par la carte Arduino. Deux leds ont été ajoutées (une sur la sortie, l'autre en parallèle à l'élément chauffant) afin de garder un œil sur le fonctionnement... et parce qu'il faut toujours ajouter des leds partout pour qu'un montage ait du style.

radiateurs électriques, les plaques de cuisson, les réfrigérateurs, les ventilations...

L'algorithme implémenté dans notre croquis Arduino est simple et c'est cette simplicité qui motive généralement le choix d'une telle solution. Il n'est même pas nécessaire d'utiliser un microcontrôleur pour ce genre de choses. Rappelez-vous de l'introduction du précédent article, des anciennes guirlandes de Noël, de la petite ampoule clignotante 12V et sa lamelle se déformant avec la chaleur... Le principe est exactement le même et il n'y a pas une trace d'électronique.

Les thermostats ont longtemps fonctionné de cette manière en « tout ou rien », en particulier pour les thermostats d'ambiance utilisés pour le chauffage domestique (électrique ou à eau chaude). Une solution est apparue ensuite sous la forme de robinets ou têtes thermostatiques. Là, le principe est sensiblement différent, car la régulation devient proportionnelle : plus la température est éloignée de la consigne, plus la commande est importante. En d'autres termes, si la consigne est à 39°C et la température mesurée à 20°C, le chauffage sera plus intense que si elle était mesurée à 35°C.

C'est exactement le même principe que lorsque vous êtes dans votre voiture : vous regardez le compteur de vitesse et appuyez plus ou moins fort sur l'accélérateur jusqu'à atteindre la vitesse souhaitée.

Toute la difficulté repose alors sur la proportionnalité. L'élément chauffage doit-il être deux fois plus important à 5 degrés d'écart ou trois fois plus important qu'il ne l'est à 2 degrés d'écart ? Ce rapport température/puissance est-il linéaire ? Est-il le même si on se trouve sous la consigne et au-dessus de la consigne ? En voiture la question ne se pose pas, le régulateur c'est votre cerveau qui contrôle votre pied, mais intégrer un cerveau humain dans un montage électronique n'est pas encore possible (et peut-être pas vraiment souhaitable selon le cerveau choisi).

Rappelez-vous que nous avons choisi d'utiliser une sortie analogique (PWM) de la carte Arduino. Il est donc possible d'adapter notre croquis à cette notion de simple proportionnalité. Ceci cependant vous sera laissé en guise d'exercice, car nous allons explorer une voie bien plus intéressante.

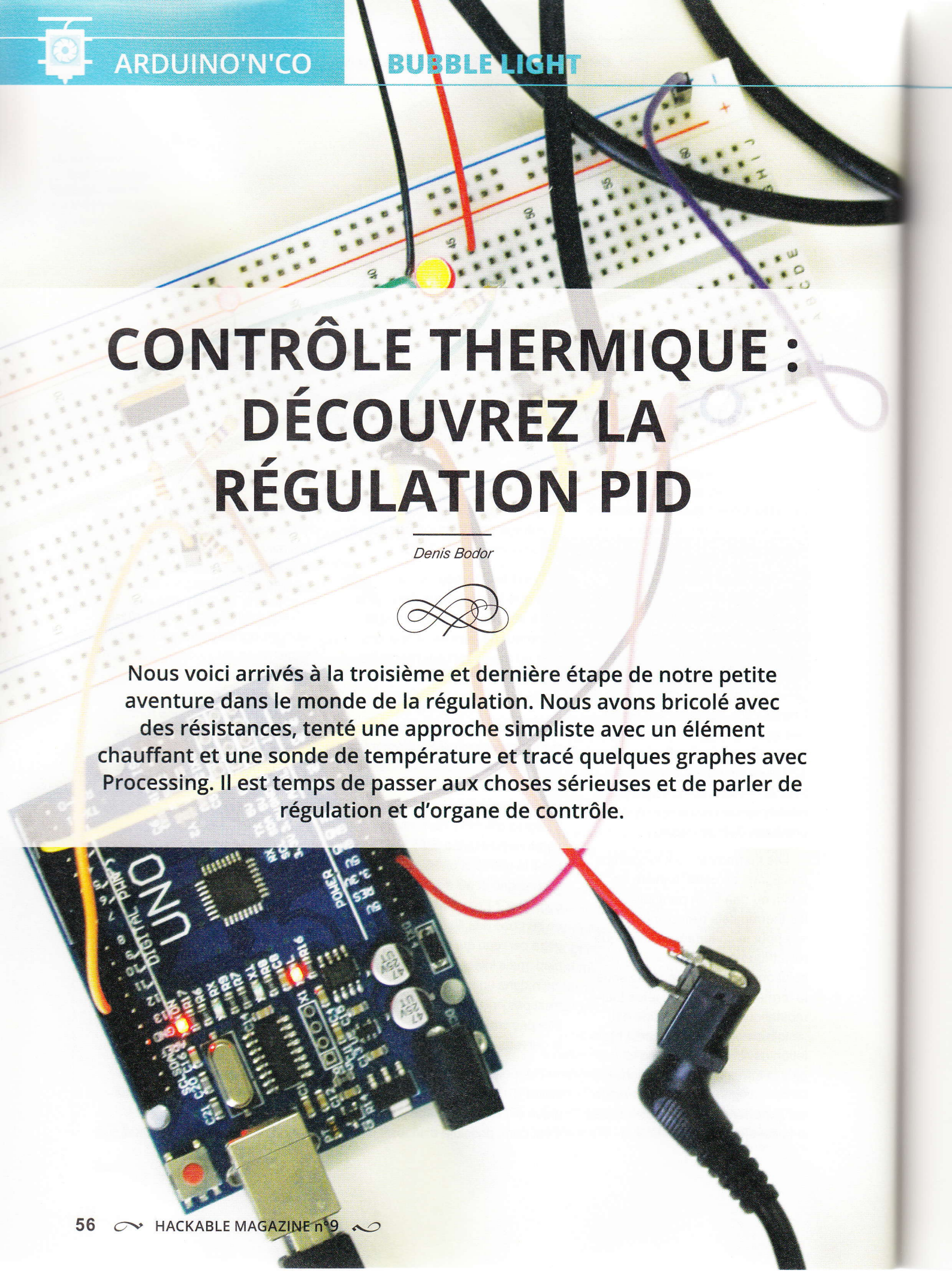
Dans certains cas, cette régulation proportionnelle est bien suffisante, mais les choses changent depuis quelques années. Des considérations économiques et/ou environnementales font qu'il est devenu important, dans presque tous les domaines liés à l'énergie, de réguler de la façon la plus efficace possible, de manière à non seulement atteindre précisément la consigne fixée, mais également le faire le plus rapidement possible et sans oscillation.

La régulation proportionnelle n'est pas suffisante, ceci est connu de longue date et la solution que nous allons explorer dans le prochain article n'est pas nouvelle. Elle était simplement réservée jusqu'alors à des domaines où la précision était de mise. Ce type de régulation efficace porte un nom, qu'on retrouve dans bien des applications et qui n'est en rien limité aux domaines thermiques : c'est la PID pour Proportionnel Intégral Dérivé. **DB**

CONTRÔLE THERMIQUE : DÉCOUVREZ LA RÉGULATION PID

Denis Bodor

Nous voici arrivés à la troisième et dernière étape de notre petite aventure dans le monde de la régulation. Nous avons bricolé avec des résistances, tenté une approche simpliste avec un élément chauffant et une sonde de température et tracé quelques graphes avec Processing. Il est temps de passer aux choses sérieuses et de parler de régulation et d'organe de contrôle.



À l'étape précédente, nous avons constaté que le fait de simplement activer ou désactiver l'alimentation de l'élément chauffant en fonction de la température mesurée ne permettait pas d'obtenir un système stable du fait de l'inertie thermique. Nous avons également envisagé de contrôler l'intensité du chauffage en fonction de l'écart entre la température mesurée et la consigne de 39°C.

Ce type de régulation est dite « proportionnelle » et ce que nous allons explorer à présent n'est applicable qu'à cette condition : il faut que l'organe de commande qui contrôle le système puisse être manipulé de manière proportionnelle, non simplement allumé ou éteint, tout ou rien, on/off, etc. Ce type de commande peut être, comme ici, une PWM ajustant l'intensité d'un élément chauffant, mais également n'importe quel autre type de dispositif :

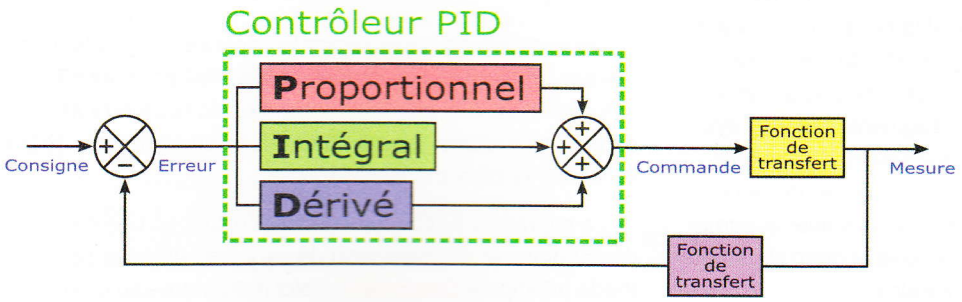
- une vanne dont on commande l'ouverture ;
- la vitesse de rotation d'une ventilation ;
- l'intensité lumineuse d'une source ;
- la pression exercée sur un élément mécanique ;
- un angle d'inclinaison ;
- la hauteur d'un dispositif ;
- le remplissage d'un conteneur quelconque ;
- etc.

Tout ce qui peut être contrôlé sous la forme d'un ordre de grandeur peut être une commande du système. Il nous faut également un moyen de mesurer le résultat de nos commandes. Dans notre cas, ce rôle est joué par la sonde de température DS18B20, mais là encore il peut s'agir de n'importe quelle donnée quantifiable : un volume, un poids, une vitesse, une inclinaison, un taux hygrométrique, une pression, une vibration, un niveau...

1. UN PEU DE THÉORIE, JUSTE UN PEU

Comme nous l'avons vu précédemment, le mode tout ou rien conduit à une oscillation de la température. La première chose à envisager est donc de rendre proportionnelle l'intensité du chauffage par rapport à l'écart de température par rapport à la consigne que nous fixons, à 39°C. L'idée est de chauffer « plus fort » si nous sommes à 20°C, un peu moins si nous mesurons 30°C et beaucoup moins si nous arrivons à 37°C.

Le problème qui se pose alors est typique de la régulation proportionnelle. À 39°C, l'élément chauffant est coupé puisqu'il n'est actif qu'en rapport avec l'écart entre la température mesurée et la consigne. Si l'écart (ou erreur) est nul, alors le chauffage est désactivé. Mais dans ce cas, la température va descendre. Finalement, le système va bien se stabiliser, mais en dessous de la consigne puisque mathématiquement, il FAUT une erreur pour que le chauffage fonctionne. Il est bien entendu possible de tricher et régler une consigne au-dessus de la



Représentation schématique simplifiée d'un contrôleur ou régulateur PID.

valeur effectivement souhaitée. Cela fonctionnera si les conditions ne changent pas, mais dès que la température de la pièce où se trouve le montage sera différente, notre tricherie ne fonctionnera plus et le système se stabilisera à nouveau à une autre température que celle souhaitée.

Oui, on peut imaginer l'ajout d'un second capteur mesurant la température ambiante et réglant notre « fausse consigne » en conséquence. La question est alors : comment allez-vous réguler cette valeur de triche ? C'est un retour à la case départ...

Il nous faut arriver à prendre en compte cet écart entre la température où le système se stabilise et la consigne. La solution consiste à utiliser l'erreur résiduelle (la différence avec la consigne) et la cumuler à intervalle régulier. On additionne ensuite cette erreur cumulée avec notre valeur proportionnelle pour obtenir la valeur qui contrôle le chauffage. Nous obtenons une régulation PI, comme Proportionnelle-Intégrale.

Tant que la consigne n'est pas atteinte, l'erreur s'accumulera et contribuera à la correction de la même façon que l'écart proportionnel entre la température mesurée et la consigne. Lorsque la consigne est atteinte, l'erreur à additionner est égale à 0 et la somme n'augmente plus. Le système se stabilise à la consigne donnée. En cas de dépassement de la consigne, la valeur accumulée est négative et donc soustraite du « compteur ».

La régulation PI est adaptée pour le contrôle de systèmes thermiques à eau par exemple, avec une inertie thermique importante et des actions dont les conséquences mettent du temps à apparaître. La composante proportionnelle est celle qui fait la majeure partie du travail, celle d'intégration se charge d'affiner la régulation et permet d'atteindre la consigne.

Avec d'autres systèmes, comme la climatisation par exemple, l'impact de la régulation est très rapide. Il est alors nécessaire d'ajouter une troisième composante de régulation dans le système, qui prendra en compte la vitesse d'évolution (ou de dérive) de l'erreur : la Dérivée. Cela semble compliqué, mais en réalité ceci revient à simplement mesurer l'erreur actuelle et lui soustraire la valeur de l'erreur précédente. Cette composante est alors additionnée aux deux autres pour contrôler la régulation et former une régulation Proportionnelle-Intégrale-Dérivée ou PID.

Pour contrôler la façon dont un système PID fonctionne, chaque composante se voit associer un coefficient, ou gain, permettant de régler l'influence de chacune d'elles pour un système donné. L'intervalle de temps pour l'intégration et la dérivée est également à prendre en compte. Le choix de ces différentes variables est l'étape de mise au point du régulateur PID ou *tuning* en anglais. Celle-ci dépendra du système que nous mettons en place, de l'effet de la commande sur le résultat, de l'inertie du système et bien entendu du résultat souhaité (favoriser la stabilité, la réactivité et/ou la précision).

2. BIBLIOTHÈQUE ARDUINO ET MISE EN ŒUVRE

Le principe théorique du fonctionnement maintenant décrit, nous pouvons nous pencher sur la mise en œuvre. Il ne nous est pas utile de changer quoi que ce soit au montage initial. Le capteur DS18B20 et l'élément chauffant commandé en PWM, nous avons tout le nécessaire pour mettre en place une régulation PID.

Nous pourrions implémenter le contrôleur/régulateur nous-mêmes, mais *Brett Beauregard* l'a déjà fait pour nous sous la forme d'une bibliothèque installable via le *Library Manager* dans l'IDE Arduino. Il vous suffira donc de chercher « PID » et d'installer la bibliothèque adéquate.

La migration du code consistera surtout en une série de déclarations et d'initialisation de variables utilisées pas la méthode principale `Compute()`. Voici notre nouveau croquis :

Fichier Édition Croquis Outils Aide

```
#include <OneWire.h>
#include <DallasTemperature.h>
#include <PID_v1.h>

// le bus 1-wire est sur la broche 2
#define ONE_WIRE_BUS 2
// On contrôle avec la broche 11
#define OUTP 11

// Variables pour PID
//   La température souhaitée :
double Consigne;

//   La température mesurée :
double Temp;

//   La valeur PWM 0-255 :
double Mosfet;

// déclaration du bus 1-wire
OneWire oneWire(ONE_WIRE_BUS);
// déclaration des capteurs
DallasTemperature sensors(&oneWire);


// Déclaration de l'objet PID
// Les arguments sont les variables de gestion
// puis les gains pour P, I et D
// et enfin,le mode
PID myPID(&Temp, &Mosfet, &Consigne, 6, 1, 0.25, DIRECT);

void setup(void) {
  // broche 11 en sortie
  pinMode(OUTP, OUTPUT);
  // activation port série
  Serial.begin(115200);
  // démarrage capteurs
  sensors.begin();
  // consigne à 39°C
  Consigne = 39;
  // contrôle entre 0-255
  myPID.SetOutputLimits(0, 255);
  // on active la régulation PID
  myPID.SetMode(AUTOMATIC);
}

void loop(void) {
  // demande de mesure
  sensors.requestTemperatures();
  // obtention de la température du premier
  // et seul capteur DS18B20
```

```
    Temp = sensors.getTempCByIndex(0);

    // calcul PID, la variable Mosfet contient
    // ensuite la valeur calculée
    myPID.Compute();

    // écriture PWM
    analogWrite(OUTP, Mosfet);

    // envoi des informations temp:PWM à Processing
    Serial.print(sensors.getTempCByIndex(0));
    Serial.print(":");
    Serial.println(Mosfet);
}
```

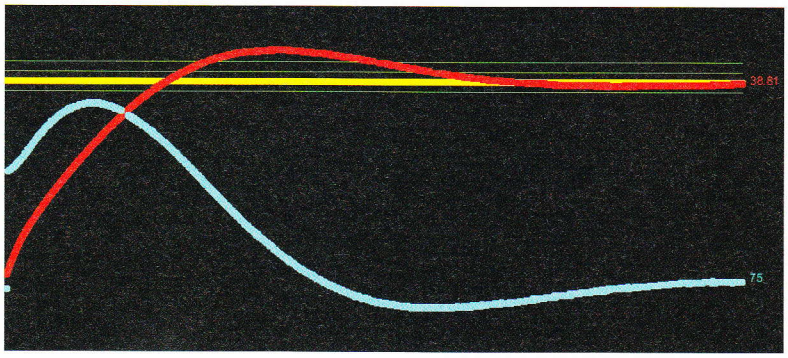
Arduino

À chaque tour de boucle **loop()** nous calculons la valeur destinée à **analogWrite()** grâce à la méthode **Compute()** de notre objet **myPID**. Bien entendu, ceci vient juste après avoir mesuré et stocké la température du capteur DS18B20 dans **Temp**. Après l'appel, **Mosfet** contiendra la valeur à utiliser entre 0 à 255, qui correspond au rapport cyclique que le contrôleur PID juge adapté.

Les paramètres importants sont les coefficients précisés lors de la création de **myPID**. Nous avons ici choisi 6 pour la composante proportionnelle, 1 pour l'intégration et 0.25 pour la dérivé. Ceci signifie que l'erreur mesurée (l'écart de la température actuelle par rapport à la consigne) aura un impact très important, alors que le cumul d'erreurs aura une importance bien moindre. La vitesse à laquelle l'erreur dérive n'est pas importante, car le système met un temps non négligeable à réagir. Nous sommes davantage dans une logique proche de la régulation d'un chauffage à eau que d'une climatisation d'un flux d'air.

3. RÉSULTATS

Après programmation de notre carte Arduino UNO et lancement du croquis Processing, que nous n'avons pas besoin de modifier, le graphique en temps réel nous permet de constater qu'effectivement, non seulement le système n'oscille plus du tout (merci P), mais finit par atteindre la consigne sans difficulté et assez rapidement (le graphique a été redimensionné et retracé pour les besoins de la mise en page) :



Très rapidement, alors que la température est autour de 20°C, la PWM saute presque directement autour de 160/255. La température augmente alors progressivement. Avant même que la température n'atteigne la consigne, le rapport cyclique se réduit. Puis arrive le dépassement de consigne ou *overshoot* en anglais, mais le régulateur est déjà sur la bonne voie lorsqu'on passe à nouveau la barre des 39°C. Le rapport cyclique augmente alors légèrement et nous arrivons au régime stationnaire avant de rester, à 0,1°C près, à la consigne.

Pour perturber le système et/ou nous permettre d'ajuster nos paramètres PID, il suffit de refroidir rapidement le capteur. Pour cela, rien de tel qu'un petit coup de bombe à air sec tête en bas. ATTENTION,

soyez très prudents ! Certes, la décompression rapide du gaz produit une chute de température importante, mais :

- il est expressément indiqué qu'il faut utiliser l'aérosol en position verticale ;
- le gaz est inflammable et ne doit pas être dirigé vers un élément incandescent ou une flamme ;
- vous pouvez gravement vous brûler (gelure/brûlure par le froid) ;
- et enfin, selon la température et le type d'élément chauffage, un tel refroidissement peut créer un choc thermique et littéralement briser le composant.

Une solution plus sûre, surtout en ce moment, est de simplement ouvrir une fenêtre et ventiler un peu d'air frais vers le duo élément/capteur. On peut également envisager de simplement mettre un glaçon dans un petit sachet plastique et le placer à proximité. Le but n'est cependant pas de perturber au maximum le système puisque cela ne correspond en rien aux conditions d'utilisation normales du montage. Pour rappel, l'idée était simplement de chauffer à une température choisie une *bubble light* pour faire apparaître des bulles...

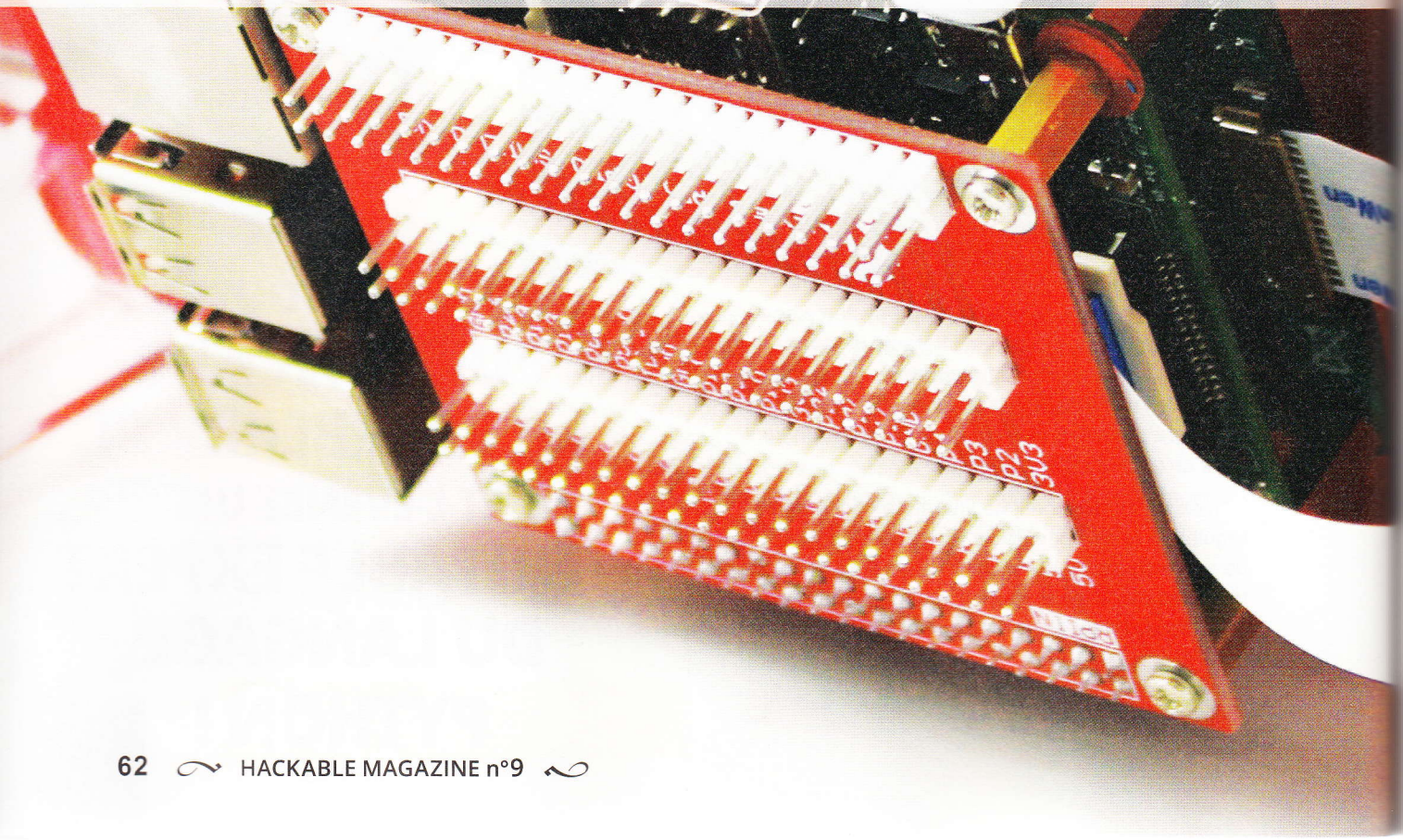
Bien sûr, ceci était surtout une excuse pour s'initier à la régulation PID, mais à présent vous êtes à même d'utiliser cette technique et cette bibliothèque avec tout et n'importe quoi. Le contrôle de moteurs par exemple est une excellente suite à cette initiation. Contrôlez le moteur de la même façon que l'élément chauffant (sans oublier d'ajouter une diode de roue libre, cf. article sur les relais dans *Hackable n°7*) et ajoutez, en guise de mesure, un capteur optique ou à effet Hall pour compter le nombre de tours par seconde. Ce système se comportera de manière très différente de notre exemple, car la variation du rapport cyclique aura un effet quasi immédiat sur la vitesse du moteur et cela changera grandement les valeurs de gain pour la mise au point du régulateur PID.

Il existe énormément de littérature concernant la régulation et les contrôleurs PID, aussi bien du point de vue mathématique que pratique. Je ne saurai que trop vous en conseiller la lecture, car ceci pourra vous être fort utile dans bien des domaines. DB

ÉCRAN LCD TACTILE 7 POUCES : L'OFFICIEL OU SOLUTION HDMI ?

Denis Bodor

Dernièrement, la fondation Raspberry Pi a annoncé la commercialisation d'un écran officiel dédié aux Raspberry Pi A+, B+ et 2. Ici à la rédaction, nous sommes d'un naturel méfiant et nous avons donc à la fois craqué pour le périphérique officiel, mais également pour un écran générique disposant d'un connecteur HDMI standard, histoire de comparer tout cela.



Lors de son arrivée il y a maintenant de cela presque 4 ans, le nano ordinateur monocarte (SBC en anglais pour *Single-Board Computer*) était principalement destiné à une utilisation en tant que machine éducative, exactement comme un PC miniature sous GNU/Linux. Le réflexe était donc naturellement d'y brancher un clavier, une souris et un écran HDMI. Pour autant, depuis les toutes premières versions, la carte était équipée de deux connecteurs en surface permettant de brancher de futurs modules caméra et écran.

Le module caméra est déjà commercialisé depuis quelque temps et voici venir maintenant l'écran officiel de 7 pouces de diagonale (l'annonce a été faite le 8 septembre dernier sur le blog de la fondation). Cependant, comme nous l'avons vu dans de précédents numéros du magazine, cette voie n'est pas la seule utilisable pour obtenir un affichage plus compact qu'un moniteur HDMI/DVI. Il est possible de brancher des modules SPI, d'utiliser la sortie composite (RCA) avec un écran (système de vision ou multimédia de véhicule), de faire usage d'un moniteur miniature HDMI ou encore d'opter pour des modules se connectant au port DSI et permettant de s'interfacer avec un écran d'iPhone ou un moniteur HDMI/DVI.

Le connecteur DSI, pour *Display Serial Interface* répond aux spécifications éponymes émises par l'alliance MIPI (*Mobile Industry Processor Interface*) visant à standardiser l'interfaçage compact d'un système d'affichage. Il ne s'agit donc pas, en principe, d'une spécificité des cartes Raspberry Pi, mais il ne faut pas oublier en revanche que les spécifications décrivent non seulement un jeu d'instructions générique (*Display Command Set* ou DCS), mais aussi un jeu d'instructions propre à chaque périphérique (*Manufacturer Command Set* ou MCS). C'est un peu comme le standard LVDS, disposer d'une carte et d'un écran ayant chacun le bon connecteur ne signifie par pour autant qu'une compatibilité existe. C'est pour cette raison qu'il est généralement recommandé, dans le cas d'un écran LVDS par exemple, d'acquérir la carte contrôleur (entrée HDMI ou DVI) et la dalle LCD en même temps.

L'écran équipé du cadre optionnel transpire indubitablement la qualité. On est loin de la dalle LCD fixée sur un circuit imprimé avec une salade de câbles en guise de connectique.

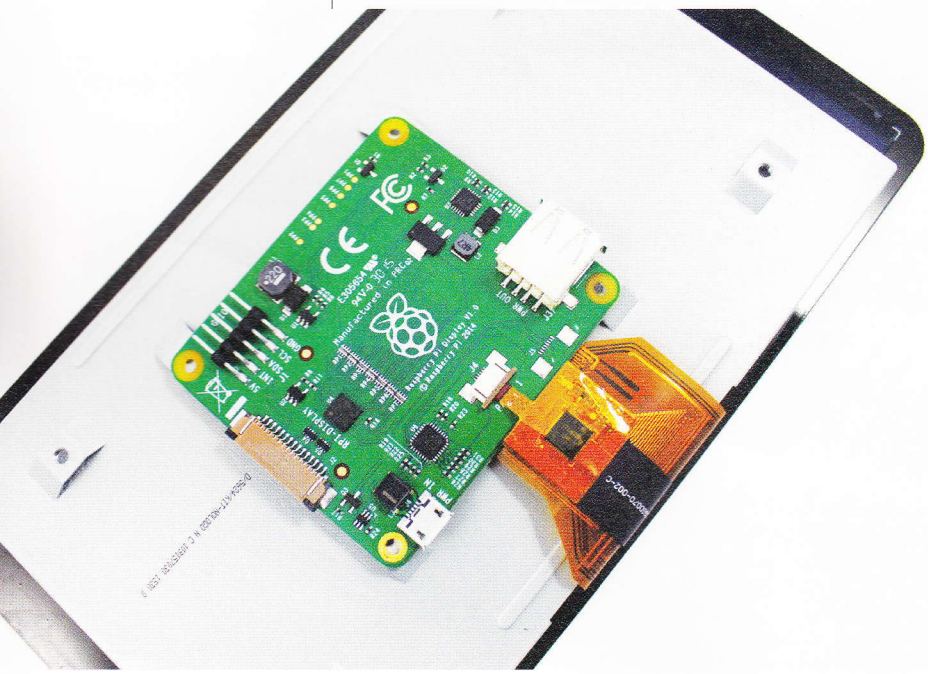
L'arrière de l'écran à proprement parler est équipé d'une structure métallique sur laquelle se fixe la carte contrôleur et la Raspberry Pi. Les emplacements de fixation les plus à l'extérieur permettent la fixation du cadre ou éventuellement l'intégration dans un boîtier.

L'objet de cet article est de faire un tour d'horizon de cet écran officiel et de le comparer avec l'une des solutions disponibles depuis quelque temps déjà. En effet, quelques constructeurs se rendant parfaitement compte de l'engouement autour de la Raspberry Pi et d'autres plateformes disposant d'une connexion HDMI ou micro HDMI, ont rapidement commencé à distribuer des écrans de 7 ou 10 pouces.

Tous ces matériels sont généralement vendus comme des écrans Raspberry Pi et se déclinent sous plusieurs formes :

- des écrans standards à un format réduit et entrées composites/HDMI/DVI/VGA ;
- des ensembles dalle LCD + contrôleur + carte annexe n'étant très clairement rien d'autre que l'intérieur d'un moniteur ;
- des écrans avec toute l'électronique embarquée, plus compacts, mais également vendus nus sans boîtier ou cadre, comme le touchscreen SainSmart 7 pouces et ses clones.

En ce qui me concerne, pour procéder à la comparaison la plus juste possible, j'ai opté pour ce dernier type d'écran disposant d'un format et d'une finition la plus proche possible de l'écran officiel. Les deux périphériques ont été achetés sur eBay à des vendeurs anglais pour une raison évidente : c'est un canal de distribution efficace. Les produits officiels (mais il faut vérifier le vendeur) sont généralement plus vite disponibles que par les canaux habituels et les produits chinois comme le clone d'écran SainSmart sont difficiles (voire impossibles) à obtenir autrement. Les prix indiqués dans la suite de l'article font directement référence à ce canal d'achat. Enfin, pour être le plus exhaustif possible, précisons que l'écran officiel a été acheté à « *the_pi_hut* » et le modèle générique à « *1buylong* ».

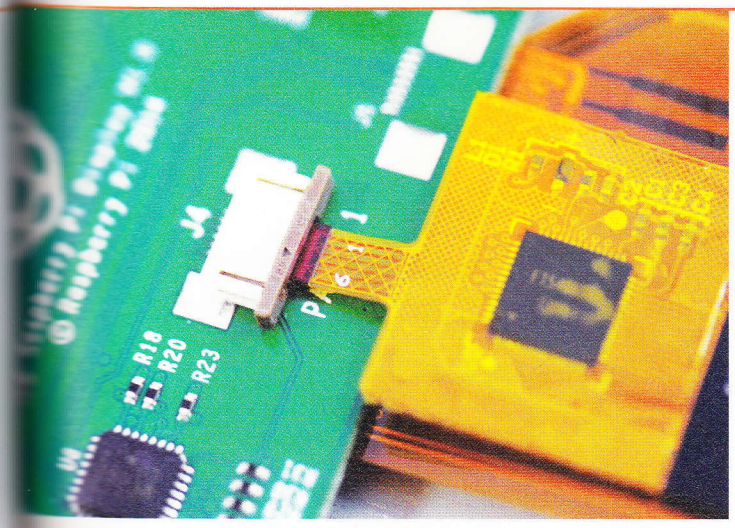


La carte contrôleur, après connexion de l'écran de part et d'autre du circuit, se loge directement sur le métal du châssis. Remarquez à droite le connecteur USB pouvant fournir une alimentation à la Pi et dans le coin supérieur gauche une série de broches avec +5V et masse pour une alternative d'alimentation et l'accès au périphérique i2c (les Pi modèles A+, B+ et 2 n'en ont pas besoin).

1. ÉCRAN OFFICIEL

1.1 Caractéristiques et présentation

- Prix : £54 + £3 (port) + £8 (cadre) soit 75€ + 4€ + 11€ = 90€
- Diagonale : 7 pouces
- Résolution : 800×480
- Touchscreen : capacitif multi-touch (10 points) contrôleur FT5406 de FocalTech
- Couleurs : 24 bit RGB
- Rétroéclairage : leds
- Connexion : flatflex (câble ruban) DSI
- Inclus : une carte d'interface, un connecteur flatflex, visserie, 4 câbles femelle/femelle
- Emballage : une jolie boîte blanche et rouge avec les différents éléments dans des emplacements mousse prédécoupés



La connexion de l'écran à la carte contrôleur demande précision, minutie et douceur. Des petits doigts et un environnement calme sont fortement recommandés pour cette opération.

- Option : cadre en plastique transparent teinté (noir, vert, bleu, violet, rouge ou orange) à assembler, avec pieds.

La qualité et les finitions du matériel sont exemplaires, tout autant que l'emballage dans lequel il est livré : une jolie boîte aux couleurs Raspberry Pi contenant des emplacements dans la mousse prédécoupés pour chaque élément. L'ensemble arrive avec, en guise de documentation, uniquement les recommandations de sécurité. Il faut aller sur le Web pour consulter un PDF ou lire une vidéo d'assemblage.

De face, l'écran se présente comme un afficheur qui n'est pas sans rappeler les écrans de MacBook avec en façade une surface transparente entourée d'une bordure noire relativement conséquente. Mais l'élément qui est le plus impressionnant est sans doute l'arrière du matériel, car en lieu et place de l'habituel circuit imprimé se trouve un châssis métallique couvrant la majorité de l'ensemble (tout ce qui se trouve dans le cadre).

Ceci permet, bien entendu, de fixer la carte contrôleur sur l'écran puis la Raspberry Pi elle-même en utilisant les entretoises en laiton fournies. Deux connecteurs souples flatflex dépassent de la bordure supérieure et sont destinés à être branchés sur une face et l'autre du contrôleur. Celui-ci est ensuite relié par un autre flatflex sur le côté de la carte au connecteur DSI de la Pi.

L'impression générale de qualité et de solidité du produit fait écho au billet sur le blog de la fondation, parlant du choix d'un écran de qualité industriel. C'est ce même billet qui indique que l'écran lui-même est interfacé en DPI (*Display Parallel Interface*) et que le contrôleur est un design de la fondation (d'où le logo sérigraphié) assurant la conversion DSI (Pi) vers DPI (écran) ainsi que l'intégration des signaux du contrôleur de la surface tactile.

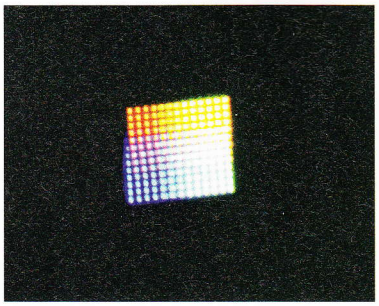
Quelques éléments sont sensiblement dérangeants si on est un peu tatillon : le connecteur plat DSI occupe une place importante et est susceptible d'être arraché par mégarde. Il en va de même si l'on décide d'alimenter la Raspberry Pi via les longs câbles à brancher au connecteur 40 broches. Le cadre en plastique optionnel me semble également très important, car même si l'écran lui-même est renforcé par le châssis métallique, toute la bordure extérieure n'est qu'une mince plaque minérale qui semble relativement fragile. Une autre solution est d'intégrer l'écran dans une boîte « maison » et le fixer à l'aide des 4 emplacements pour vis M3 initialement prévus pour le cadre optionnel. Enfin, et c'est sans doute le plus dérangeant, l'utilisation du cadre optionnel gène la connexion de périphériques USB. Sur les 4 ports USB, deux sont directement dans l'axe d'un des pieds du cadre, ce qui interdira l'utilisation sans rallonge de clés USB un peu grosses.

1.2 Connexion et configuration

La connexion ou plutôt l'assemblage du matériel avec la Raspberry Pi demande une certaine dextérité. L'ensemble se compose d'un écran, d'une carte et de connectique. Avant de vous lancer dans l'opération, il est important de bien comprendre un point important du système d'alimentation. La carte contrôleur doit être reliée à l'écran et à la Raspberry Pi, mais l'alimentation peut se faire de diverses façons :

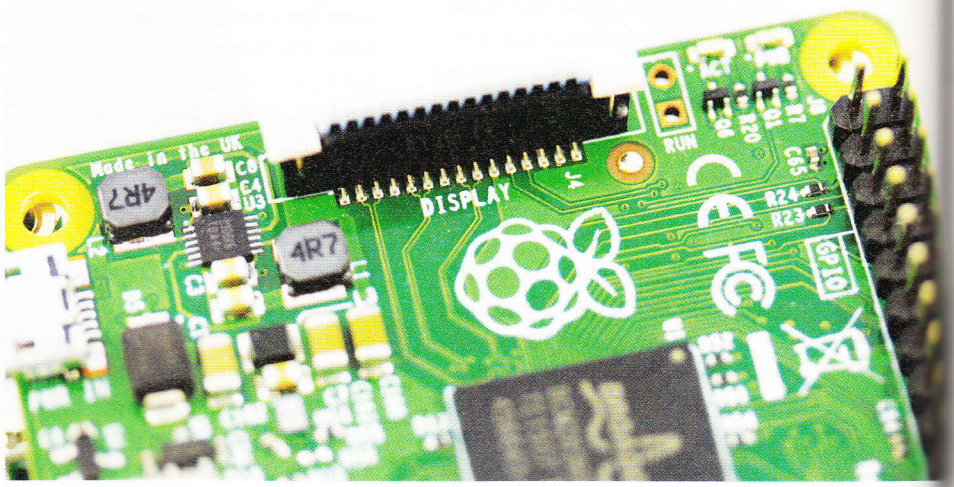
- une alimentation via un connecteur USB micro B sur la carte contrôleur et deux des câbles fournis reliant VCC (+5V) et la masse depuis la carte vers la Pi. Dans ce cas de figure, l'alimentation arrive sur le contrôleur d'écran et une tension régulée est délivrée à la Raspberry Pi sur le connecteur 40 broches ;
- une alimentation via un connecteur USB micro B sur le contrôleur et une connexion USB A vers USB micro B de la carte vers la Pi. La carte contrôleur dispose en effet d'un connecteur USB A femelle permettant de fournir le courant demandé par la Pi (attention au câble utilisé, si le connecteur est trop long, le cadre optionnel gêne la connexion) ;
- deux alimentations distinctes sous la forme de deux connexions USB micro B : une sur la carte contrôleur et une sur la Pi.

Le point important à prendre en compte est le courant nécessaire à l'ensemble. Avec une seule source, il est indispensable d'utiliser un bloc d'alimentation, de bonne qualité, et capable de fournir 2A. Dans le cas de deux sources, 500mA seront suffisants par connexion, mais gardez à l'esprit qu'avec des blocs secteurs de mauvaise qualité, vous irez au-devant de problèmes de stabilité. Dans tous les cas, retenez en premier lieu ceci : c'est l'écran qui peut alimenter la Pi et non l'inverse. Ne branchez pas l'écran sur l'un des ports USB hôtes de la Raspberry Pi !

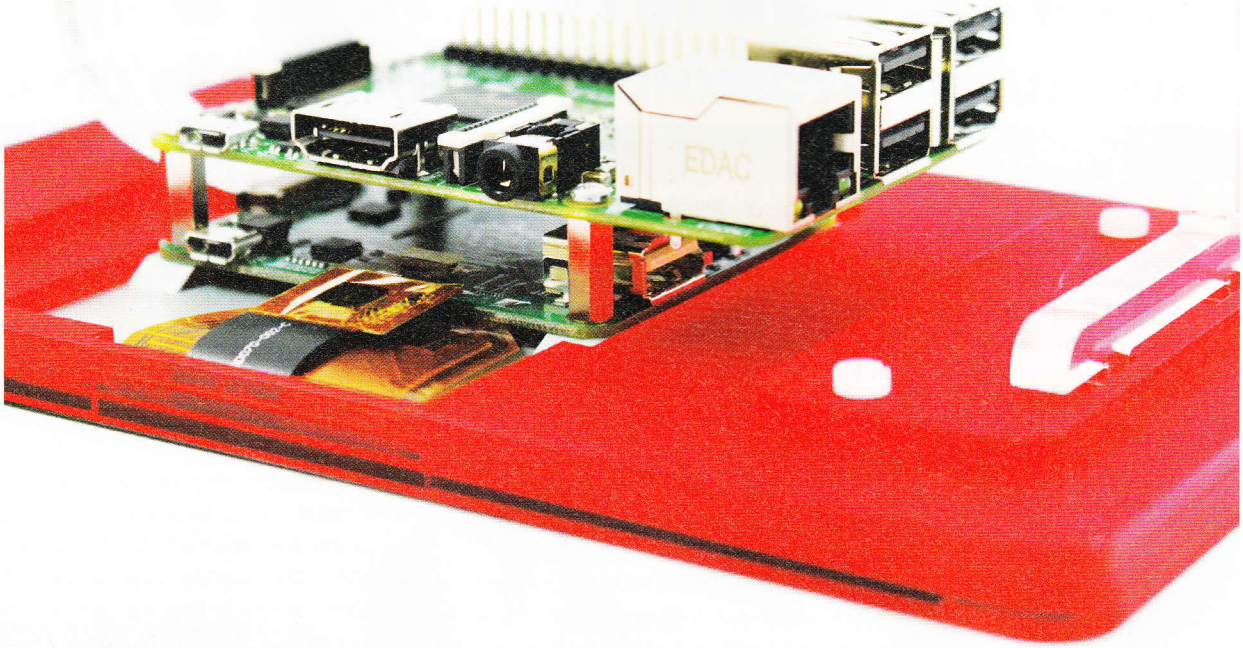


Cette petite icône, alias le « Rainbow square » sur les forums, signale un problème d'alimentation. Si celui-ci apparaît en haut à droite de l'écran, c'est signe qu'une chute de tension est détectée. Dans ce cas, soit vous n'utilisez pas une alimentation capable de fournir 2A en 5V, soit vous alimentez beaucoup trop de périphériques depuis la Raspberry Pi (USB ou autre).

Le système intègre un signal d'avertissement en cas de problème d'alimentation, sous la forme d'une icône multicolore apparaissant en haut à droite de l'écran. Ceci est un pictogramme (*rainbow square*) apparaissant en surimpression à partir du modèle B+. Si ce carré arc-en-ciel s'affiche, cela signifie que le courant fourni n'est pas suffisant pour « maintenir » la tension nominale (>4,65V). Une telle chute de tension s'appelle un *brownout* dans le jargon. Notez que si le carré est rouge, il s'agit d'un indicateur de surchauffe (>80°C). Il est possible de désactiver cette notification en ajoutant `avoid_warnings=1` dans le `config.txt` (ceci n'est pas recommandé).



Enfin ce connecteur DSI sert à quelque chose ! Cela paraît incroyable, mais toutes les données à afficher ainsi que les informations concernant la surface tactile, transitent par ce connecteur 15 broches à l'aspect fragile.



La Raspberry Pi est fixée à bonne distance de la carte contrôleur avec la visserie fournie. Notez la position prééminente du flatflex (câble plat) DSI ainsi que les connecteurs USB et Ethernet de la Raspberry Pi juste dans l'axe d'un des pieds du cadre.

Après avoir suivi les indications de connexions fournies par le distributeur (comme celui-ci par exemple **http://www.farnell.com/datasheets/1960197.pdf**) l'ensemble écran + contrôleur + Raspberry Pi forme un tout compact. Si comme moi nous avez ajouté quelques euros pour un cadre, vous vous retrouvez avec un résultat qui ressemble à s'y méprendre à un cadre photo numérique. Un conseil à ce propos : ajoutez les éléments du cadre juste après avoir fixé le contrôleur à l'arrière de l'écran et avant de connecter le flatflex DSI et fixé la Pi. Ceci vous évitera des jongleries pénibles.

Côté logiciel et système, si tout est à jour, la configuration se résume à... rien ! Un simple démarrage de la Raspberry Pi devrait automatiquement afficher exactement la même chose que sur un moniteur HDMI (carré multicolore, une ou plusieurs framboises selon le modèle de Pi et les messages de démarrage).

Si ce n'est pas le cas, la première chose à faire est de procéder à une mise à jour via les commandes :

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Ceci aura pour effet d'installer un noyau Linux récent, les firmwares et les outils adéquats pour la prise en charge du matériel.

À ce stade, il est fort probable que la commande **fbset**, affichant des informations sur le *framebuffer* vous retourne quelque chose comme ceci :

```
$ fbset
mode "752x448"
    geometry 752 448 752 448 16
    timings 0 0 0 0 0 0 0
    rgba 5/11,6/5,5/0,0/16
endmode
```

Avant de crier à la tromperie parce qu'on vous a vendu un écran 800×480 et qu'on vous livre du 752×448, tournez-vous vers le contenu du fichier **/boot/config.txt**. Là, repérez la ligne **#disable_overscan=1** puis supprimez le **#**. Ceci aura pour effet de désactiver la fonction *overscan* ajoutant une bordure autour de l'écran, tantôt utile pour un moniteur HDMI. Notez que cette modification nécessite un redémarrage du matériel.

Prenez également garde aux informations se trouvant en fin de fichier qui ont certainement été ajoutées si vous avez installé le système avec NOOBS. Celles-ci sont généralement précédées d'une ligne en commentaire `#NOOBS Auto-generated Settings`. Il sera sans doute nécessaire de les supprimer ou de les faire précéder d'un `#` puisqu'elles comprennent généralement des instructions concernant l'*overscan* et la sortie HDMI.

L'écran tactile capacitif sera également directement pris en charge et en cas de démarrage graphique (ou lancement via `startx`) vous pourrez ainsi immédiatement contrôler le système par ce biais. Notez qu'il s'agit par défaut d'un des deux modes de fonctionnement et en particulier de l'émulation de souris. Pour un usage en tant que surface tactile multi-touch, vous devrez reposer sur des applications spécifiques comme celles développées avec la bibliothèque Python Kivy (**http://kivy.org/**).

Enfin, une dernière information intéressante, il vous est possible de retourner l'écran plus ou moins comme il vous plaît. Il vous suffit d'éditer le `config.txt` et d'ajouter une des lignes suivantes :

```
# normal
display_rotate=0
# rotation à 90° (portait)
display_rotate=1
# rotation à 180° (paysage)
display_rotate=2
# rotation à 270° (portait)
display_rotate=3
# retournement horizontal (flip)
display_rotate=0x10000
# retournement vertical (flip)
display_rotate=0x20000
```

Les deux dernières instructions peuvent être intéressantes si l'écran est vu par réflexion sur un miroir ou un morceau de verre, comme un système de vision tête-haute par exemple.

2. ÉCRAN GÉNÉRIQUE HDMI

2.1 Caractéristiques et présentation

- Prix : £39,69 + £10,00 (port) soit 55€ + 14€ = 69€
- Diagonale : 7 pouces
- Résolution : 800×480
- Touchscreen : capacitif USB (contrôleur Goodix GT811 5 points)
- Couleurs : 65536
- Rétroéclairage : leds
- Connexion : HDMI
- Inclus : un câble plat HDMI, un câble USB, un DVD, visserie
- Emballage : sachet antistatique et papier bulle (*bulk*)

J'ai reçu cet écran avant le produit officiel de la fondation et ai été agréablement surpris au déballage. Certes, celui-ci était livré comme le sont très souvent les modules chinois que l'on commande ainsi sur eBay : un carton, rembourré avec un peu de « bulles », des sachets pas vraiment antistatiques et aucune explication. C'est cependant l'aspect même du matériel qui est surprenant avec un circuit imprimé propre, des soudures correctes et une sérigraphie cohérente.

Il s'agit ici d'un produit brut, mais que je qualifierai de très acceptable au regard de ce que je commande généralement. Ce n'est qu'en le comparant à l'écran officiel qu'on se rend compte de l'écart qualitatif les séparant. Il ne faut toutefois pas oublier que cet écran HDMI est vendu pour être intégré dans un projet, voire un boîtier, et que les modèles plus économiques se résument généralement au contenu désassemblé d'un moniteur.

En termes d'affichage, encore une fois, le résultat peut sembler plus qu'acceptable à première vue, si l'on ne le place pas à côté de l'écran officiel. Les couleurs sont bien plus ternes et le contraste bien moins important.

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

HACKABLE MAGAZINE

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

Vos remarques :

VOICI TOUTES LES OFFRES COUPLÉES AVEC HACKABLE !

POUR LE PARTICULIER ET LE PROFESSIONNEL ...

Prix TTC en Euros / France Métropolitaine

CHOISISSEZ VOTRE OFFRE !

Prix en Euros / France Métropolitaine

Offre	SUPPORT / ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + PDF (PDF 1 lecteur) Réf	PAPIER + PDF Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE Tarif TTC	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Réf	PAPIER + PDF + BASE DOCUMENTAIRE Tarif TTC
HK	6n° HK*	HK1	39,-	HK12	58,-				
LES COUPLAGES « EMBARQUÉ »									
D	6n° HK* + 4n° OS	D1	65,-	D12	98,-	D13	85,-*	D123	118,-*
E	6n° HK* + 4n° OS + 6n° MISC	E1	105,-	E12	158,-	E13	179,-*	E123	232,-*
E+	6n° HK* + 4n° OS + 6n° MISC + 2n° HS	E+1	119,-	E+12	179,-	E+13	193,-*	E+123	253,-*
F	6n° HK* + 4n° OS + 11n° GLMF	F1	125,-	F12	188,-	F13	229,-*	F123	292,-*
F+	6n° HK* + 4n° OS + 11n° GLMF + 6n° HS	F+1	183,-	F+12	275,-	F+13	287,-*	F+123	379,-*
G	6n° HK* + 4n° OS + 6n° LP	G1	100,-	G12	150,-	G13	164,- *	G123	214,-*
G+	6n° HK* + 4n° OS + 6n° LP + 3n° HS	G+1	129,-	G+12	194,-	G+13	193,-*	G+123	258,-*
LES COUPLAGES « GÉNÉRAUX »									
H	6n° HK* + 4n° OS + 6n° LP + 6n° MISC + 11n° GLMF	H1	200,-	H12	300,-	H13	402,-*	H123	499,-*
H+	6n° HK* + 4n° OS + 6n° LP + 3n° HS + 6n° MISC + 2n° HS + 11n° GLMF + 6n° HS	H+1	301,-	H+12	452,-	H+13	493,-*	H+123	639,-*

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

* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

N'hésitez pas à consulter les détails des offres ci-dessus sur : www.ed-diamond.com !

L'écran chinois est sobre et, si on le considère seul, relativement sympathique. On est loin du niveau de finalisation de l'écran officiel, mais la compatibilité HDMI en fait un écran générique intéressant. Ça, c'est l'impression de départ. Les choses prennent vite une autre tournure ensuite...

Cette perception s'intensifie encore dès lors qu'on n'est pas bien en face de l'écran. L'angle de vue, non spécifié par le vendeur, est très réduit et l'écran posé à plat sur un bureau sera tout simplement inutilisable.

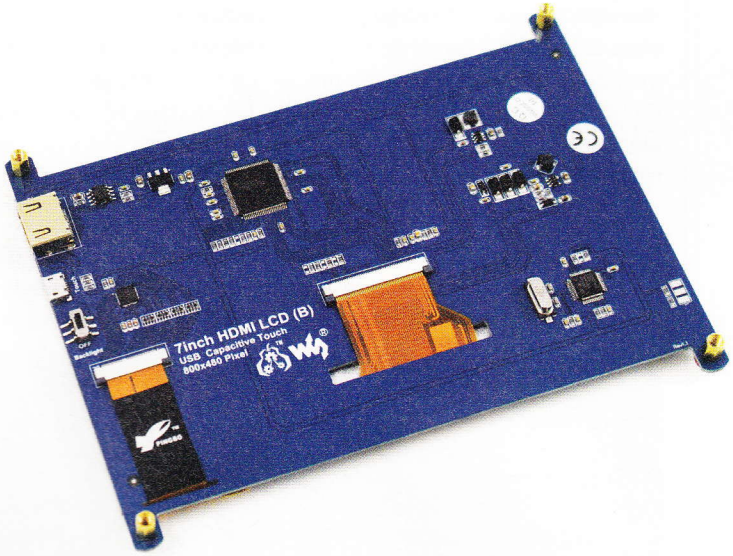
Vous remarquerez également que parmi les caractéristiques présentées ici, qui sont principalement tirées de l'annonce eBay, il est fait mention de seulement 64K couleurs. Ceci est très surprenant et semble surtout totalement faux. Une telle réduction de couleur doit être clairement visible en affichant un simple dégradé, or ce n'est pas le cas. Il s'agit très certainement d'une erreur de description. Il en va de même pour le contrôleur de la surface tactile qui est annoncé comme étant un STM32. Un STM32 est un microcontrôleur et non un composant dédié. Même si un STM32F1 est bien présent sur le périphérique, il se trouve à bonne distance du connecteur USB micro B (alimentation et touchscreen) qui lui-même est juste à côté d'une puce i2c GOODIX GT811 qui est le composant dédié à cette fonction (le datasheet ne nous apprend pas grand-chose, il est en chinois).

2.2 Connexion et configuration

La connexion de l'écran à la Pi est d'une simplicité enfantine puisqu'il suffit de brancher le câble HDMI fourni ainsi que l'alimentation sous la forme d'un câble USB (A mâle vers micro B). Cette connexion suppose que la machine hôte utilisée est susceptible de fournir le courant nécessaire, car si l'on alimente l'écran ainsi via un bloc secteur, on perd l'interface tactile (qui n'est de toute façon pas accessible, voir ci-après).

Le premier démarrage après connexion est un peu effrayant. À la mise sous tension, l'écran affiche un fond blanc rapidement

À l'arrière de l'écran, on distingue sur la gauche de haut en bas, le connecteur HDMI, l'alimentation USB servant également d'accès au touchscreen et un bouton permettant de couper le rétroéclairage. En bas à droite se trouvent le microcontrôleur STM32F1 et son quartz. C'est ce composant qui est chargé de fournir un accès à la surface tactile et qui est la source de bien des maux.



parasité par une série de lignes et de points multicolores qui laissent penser qu'un gros problème vient d'apparaître ou qu'on vient de se faire royalement avoir. Une fraction de secondes plus tard, l'affichage est réinitialisé et les messages systèmes apparaissent enfin.

Un certain nombre de problèmes d'affichage se font également jour (image étirée, résolution étrange, bordure vide, etc.). Quelque chose ne tourne pas rond au niveau de la configuration. Il faut en effet se pencher sur le contenu du fichier **/boot/config.txt** et procéder à un certain nombre de réglages :

On désactive l'*overscan* :

```
disable_overscan=1
```

On précise une utilisation non standard de l'alimentation du bus USB :

```
max_usb_current=1
```

Dès lors, vous êtes censé utiliser une alimentation qui soit capable de fournir 2A. Ici, contrairement à l'écran officiel, c'est la Raspberry Pi qui alimente l'écran et non l'inverse. Enfin, nous devons configurer différents éléments concernant la gestion HDMI dont la résolution et les *timings* en œuvre :

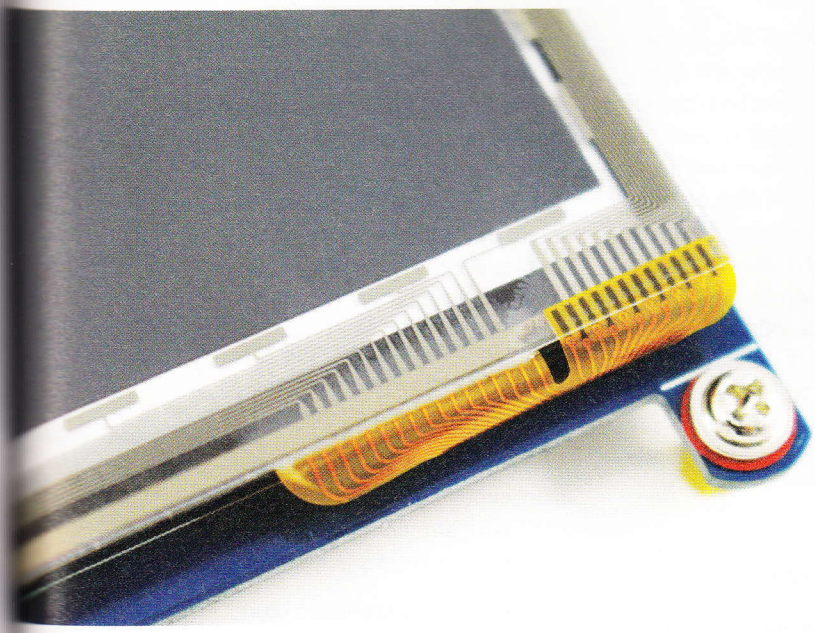
```
hdmi_force_hotplug=1
hdmi_group=2
hdmi_mode=1
hdmi_mode=87
hdmi_cvt 800 480 60 6 0 0 0
```

Ces lignes ne sortent pas de nulle part. Elles correspondent en effet aux données de configuration d'un écran étrangement similaire vendu par Adafruit pour quelques $89.95 (**http://www.adafruit.com/product/2407**). J'ai apparemment donc fait une grosse économie finalement, d'autant que le produit Adafruit dispose d'une surface tactile résistive et non capacitive.

Dès ces modifications apportées et la Raspberry Pi redémarrée, l'écran se comporte enfin de manière adéquate. L'image perturbante due à la non-initialisation est toujours présente à la mise sous tension et à l'arrêt du système, mais c'est quelque chose que l'on retrouve également, dans une moindre mesure, avec l'écran officiel. La raison est facile à imaginer puisqu'en l'absence de pilotage de l'afficheur LCD au niveau matériel, les cristaux liquides se comportent de manière plus ou moins aléatoire. C'est quelque chose que vous ne voyez pas avec un moniteur ou un téléviseur, car le rétroéclairage est tout simplement coupé lorsque l'effet se produit.

Nous arrivons maintenant à la partie déplaisante de l'essai : la surface tactile. Disons-le clairement, il n'y a tout simplement pas de support pour le périphérique dans le noyau par défaut fourni avec Raspbian. Pour constater que le périphérique est bien détecté, il nous suffit de lister les composants sur le bus USB :

```
$ lsusb
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
Bus 001 Device 004: ID 05e3:0610 Genesys Logic, Inc. 4-port hub
Bus 001 Device 005: ID 05e3:0610 Genesys Logic, Inc. 4-port hub
Bus 001 Device 006: ID 0eef:0005 D-WAV Scientific Co., Ltd
```



Gros plan sur la connexion de la surface tactile. À défaut d'arriver à la faire fonctionner sans d'ignobles bidouilles, on peut toujours s'extasier sur la beauté des électrodes transformant votre doigt en condensateur...

Accessoirement, notez que dans cette sortie de la commande `lsusb` apparaît ici deux hubs 4 ports (physiquement un hub 7 port D-Link). Il est, en effet, très fortement recommandé de connecter l'écran sur un hub actif (alimenté) et non directement sur la Raspberry Pi (malgré le `max_usb_current=1`).

Le périphérique `0eef:0005` est notre touch-screen fournissant une interface HID (*Human Interface Device*) normalement prise en charge par le noyau. Une petite recherche de ces identifiants nous apprend qu'il s'agit d'un *eGalax TouchScreen*. En creusant un peu, on se rend compte, contre toutes attentes, que le microcontrôleur STM32 semble effectivement utilisé comme interface (*glue logic*) entre le GT811 et le port USB.

Le problème majeur qui se pose à nous trouve son origine justement dans le code embarqué dans le MSP432. Ce firmware émule un contrôleur eGalax qui ne semble référencé nulle part, le *idProduct* `0005` est systématiquement rattaché à des périphériques à problèmes, d'origine assez douteuse.

Le périphérique n'existant pas, il ne dispose forcément d'aucun pilote et le noyau à jour présent sur la Raspberry Pi le détecte sous forme de matériel HID non standard *hidraw* (il sait que c'est de l'HID parce que la classe du périphérique USB le précise). Comme le rapporte la documentation du noyau, hidraw permet un accès direct au matériel sans passer par le *parser* HID (*hiddev*) et offre l'opportunité de communiquer avec le périphérique via une application en espace utilisateur.

Plusieurs solutions existent pour « forcer » le support du touchscreen, mais toutes passent par :

- l'installation d'un script Python (**https://github.com/derekhe/wavesahre-7inch-touchscreen-driver**) ;
- la compilation d'un module noyau (**https://github.com/ArhiChief/eleduino_ts**) ;
- l'installation des sources du noyau et/ou des fichiers de développement du noyau Raspberry Pi (**https://github.com/notro/rpi-source**) ;
- une bidouille ignoble dans `/sys/module` pour associer un pilote arbitrairement choisi avec le périphérique ;
- et/ou d'autres hacks qui demandent des compétences techniques et une certaine expérience du fonctionnement d'un système GNU/Linux.

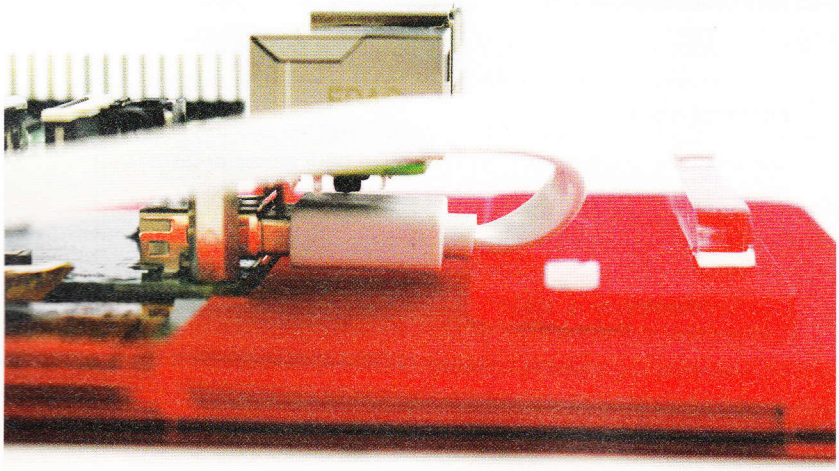
Entre nous, même avec mes quelques 20 ans d'expérience d'utilisation de ce système, je n'ai guère de motivation pour faire fonctionner ce périphérique. Non seulement la solution ne serait pas viable dans le temps, mais en plus, elle est nécessaire en raison d'une « magouille » technique du constructeur (le périphérique `0eef:0005` n'existe pas vraiment).

La comparaison avec l'écran officiel, qui lui fonctionne immédiatement, intégralement et à merveille, ne pousse aucunement à l'exploration en ce sens et au contraire incite simplement à n'utiliser ce périphérique douteux que comme un simple moniteur HDMI nécessitant déjà un brin de bidouille. Le sentiment à la mise en œuvre se résume à une grosse déception mêlée d'énervement, car la promesse d'un touch-screen HDMI/USB se transforme en une suite infinie de bricolages qui ne survivraient pas à la moindre mise à jour. Un tel acharnement pourrait être envisagé, mais uniquement à la condition qu'il n'existe pas d'autres solutions, un poil plus chères, mais presque clé en main.

3. FAIRE SON CHOIX

Le prix de l'écran officiel dépend du canal d'achat. Farnell/Element14 vend en effet le produit à 55€ hors taxe (soit 66€ TTC), mais au moment où cet article est rédigé il n'est pas disponible en stock. Il faut ajouter à cela, bien entendu, le coût du port, mais même en révisant tout cela à la baisse, nous avons toujours un différentiel minimum de 10€ dans le pire des cas et quelques 20€ dans le meilleur (à l'avantage du clone chinois).

Des trois façons d'alimenter le duo Pi + écran, ma préférée est le câble USB. À condition bien entendu d'en trouver un qui arrive à se loger dans le port dédié malgré le plastique du cadre, et qui soit suffisamment court pour que l'ensemble reste compact.



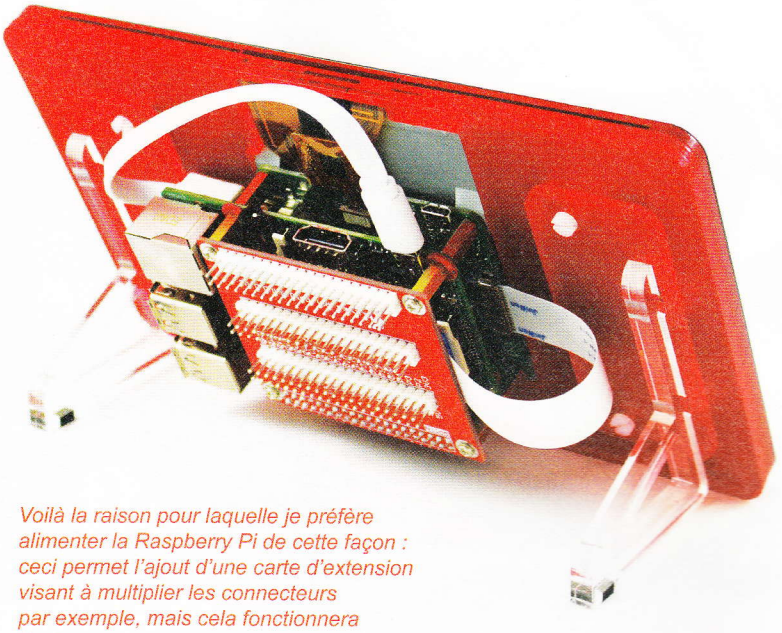
La qualité générale de l'écran officiel ne fait absolument aucun doute, qu'il s'agisse du matériel lui-même ou du soin apporté à son emballage (et donc sa protection durant le transport). De ce point de vue, la différence de prix, même de 20€, justifie largement l'option officielle :

- l'écran est clairement plus contrasté et offre un affichage bien meilleur ;
- il ne s'agit pas d'un simple circuit imprimé, mais d'une structure complète avec cadre et renforts métalliques ;
- il peut être complété d'un cadre complet optionnel ;
- il ne nécessite que peu de configuration et est directement pris en charge (et le sera encore longtemps).

Fonctionnellement, le seul point positif en faveur du clone chinois tient en sa connectique HDMI qui lui procure un caractère générique. L'écran fonctionnera, dans le pire des cas comme un système d'affichage, mais celui-ci ne se limitera pas à la Raspberry Pi. Après tout, il ne s'agit de rien d'autre qu'un écran HDMI.

Inversement, l'écran officiel avec sa connectique DSI est peu ou prou lié à la petite framboise. Il n'est sans doute pas totalement impossible de connecter cet écran à une Banana Pi par exemple, mais ceci demanderait un travail important comparé à une simple connexion HDMI (à moins que les développeurs apportent une solution clé en main pour le support du produit).

Tout dépendra donc de l'usage que vous comptez faire de l'écran, avec une balance qui penche dans la majorité des cas vers le matériel officiel. Il semblerait bien que, pour une fois, la concurrence du bas de gamme chinois ait atteint ses limites. Si vous cherchez un écran pour votre Raspberry Pi, la question ne se pose même pas, le choix est évident.



Voilà la raison pour laquelle je préfère alimenter la Raspberry Pi de cette façon : ceci permet l'ajout d'une carte d'extension visant à multiplier les connecteurs par exemple, mais cela fonctionnera également avec les « HAT » compatibles.

4. BONUS : NE PAS FAIRE SON CHOIX

Vous l'avez compris, à l'issue de cet article, je me retrouve avec un moniteur 7 pouces HDMI bas de gamme plus ou moins sur les bras (même si tout finit presque toujours dans un projet). Naturellement, la première idée qui vient à l'esprit est : pourquoi ne pas utiliser les deux écrans ?

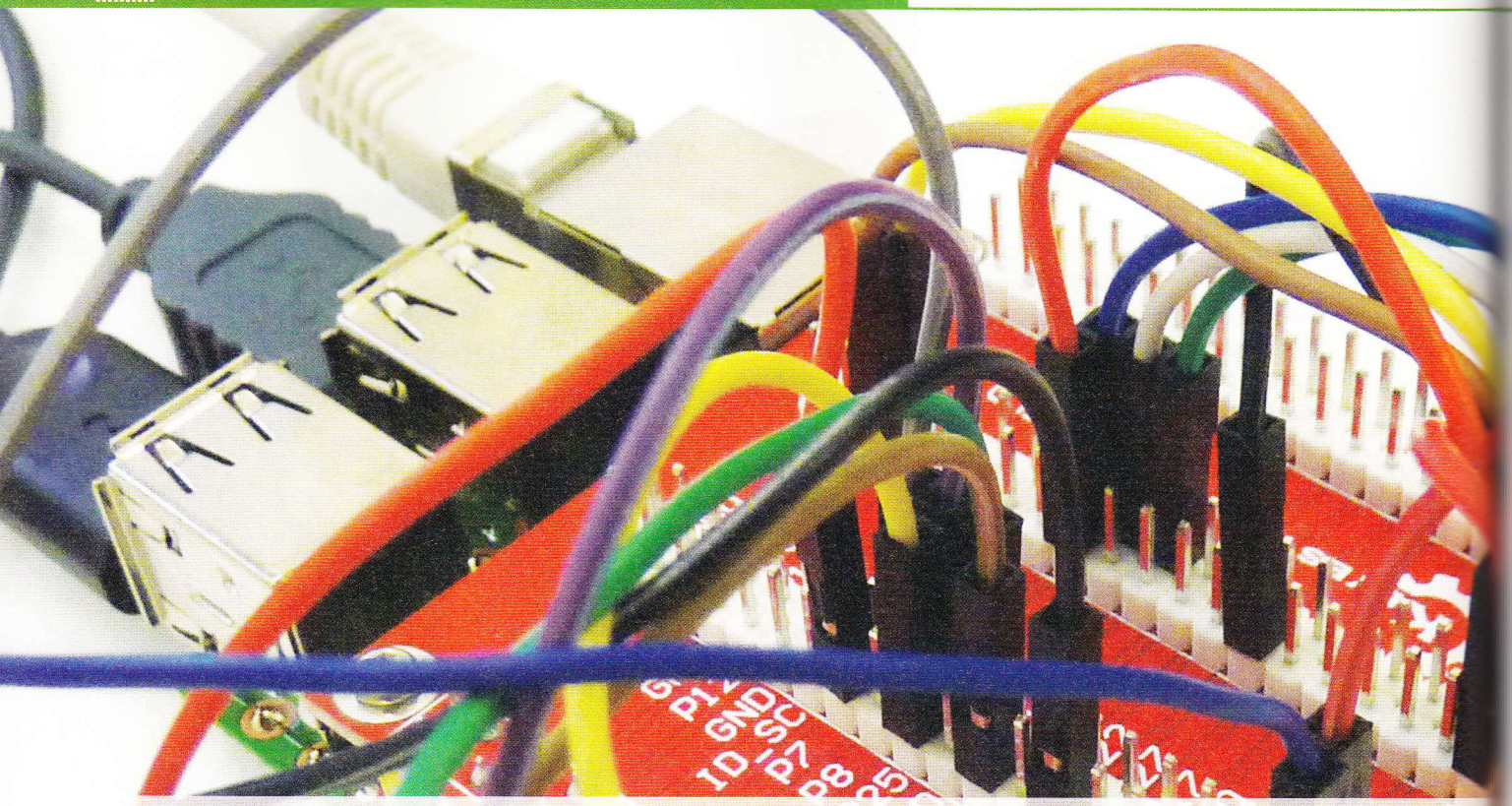
Le billet dans le blog officiel fait légèrement mention d'une telle utilisation ou plutôt de l'écran DSI en compagnie d'un moniteur HDMI standard. Lorsque l'écran DSI est connecté, celui-ci est utilisé par défaut. Il faut ajouter une ligne `display_default_lcd=0` dans le `/boot/config.txt` pour que la sortie HDMI retrouve son statut initial.

Ce n'est malheureusement pas suffisant pour l'ensemble des usages, du moins pour l'instant. Bien que la Raspberry Pi démarre alors naturellement sur la sortie HDMI, un seul et unique *framebuffer* est disponible (contrairement à l'ajout d'écran SPI par exemple). Comme nous n'avons que `/dev/fb0`, il n'est pas possible de lancer une session graphique en *dual-display* (ou *dual-head*).

La seule option à notre disposition est de nous tourner vers des applications compatibles, comme le lecteur multimédia `omxplayer`. Celui-ci permet en effet d'utiliser une option `--display` prenant en argument un identifiant d'écran (5 pour HDMI et 4 pour l'écran LCD). Il faudra également s'assurer que la mémoire allouée au GPU (processeur graphique) soit égale ou supérieure à 128 Mo (ligne `gpu_mem=128` dans `config.txt`).

Pour l'heure, les possibilités offertes s'arrêtent là. Il manque un pilote permettant d'avoir deux framebuffer distincts (`/dev/fb0` et `/dev/fb1`) afin de non seulement pouvoir utiliser X avec deux sorties, mais également utiliser l'un des deux périphériques au choix avec PyGame par exemple. Le fait est que les sorties DSI et HDMI sont toutes deux contrôlées par le GPU. Nous ne sommes pas dans une situation qui, sur PC, reviendrait à insérer deux cartes graphiques dans la machine, mais plutôt une seule carte avec deux sorties vidéo utilisables. En l'absence de support noyau pour une fonctionnalité de double framebuffer, nous sommes coincés. Mais ceci viendra peut-être avec de futures mises à jour.

En conclusion, le verdict est évident. L'écran chinois ne fait pas le poids face au produit officiel et ne constituerait une option que s'il ne fallait choisir qu'entre des périphériques du même tenant, ou si l'option officielle était deux fois plus chère. **DB**



CONFIGUREZ DEUX ÉCRANS LCD MINIATURES SUR RASPBERRY PI

Denis Bodor

Vous vous souvenez de l'article détaillant l'utilisation d'un écran miniature économique 320×240 pixels ? Les choses depuis la publication ont légèrement changé sur bien des points, tout en restant compatibles. Pourquoi revenir sur le sujet ? Parce que c'est l'occasion de voir ensemble une autre nouveauté, intégrée dans les noyaux Linux utilisés désormais par les Raspberry Pi : le device tree.

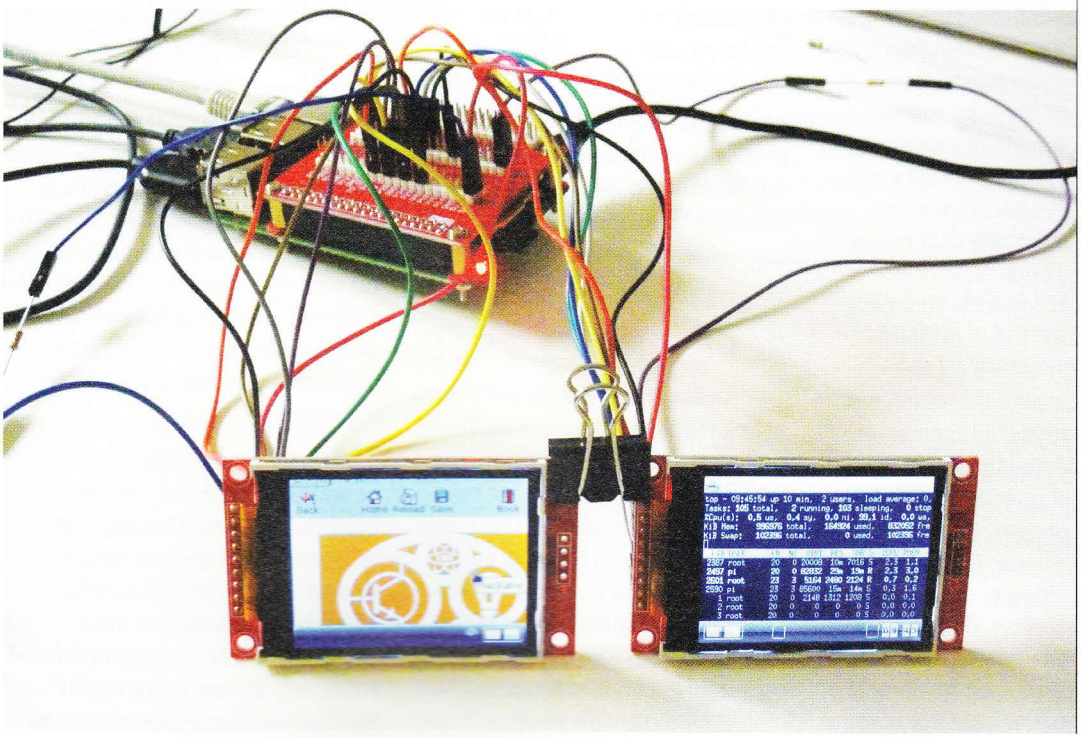
Dans un précédent article, nous avons détaillé la manière de prendre en charge un écran économique 2 pouces LCD en SPI. Il était alors question de mettre à jour le système pour obtenir un noyau compatible puis de charger un module (pilote) pour ajouter le support pour l'écran. Depuis la parution de l'article, l'énorme travail de développement accompli par Noralf Trønnes (alias Notro) pour supporter le contrôleur de cet écran (et bien d'autres) a fait son bonhomme de chemin jusqu'à intégrer le noyau Linux officiel (dit « vanilla » ou *mainline*).

1. INTÉGRATION DU SUPPORT FBTFT DANS LE NOYAU OFFICIEL

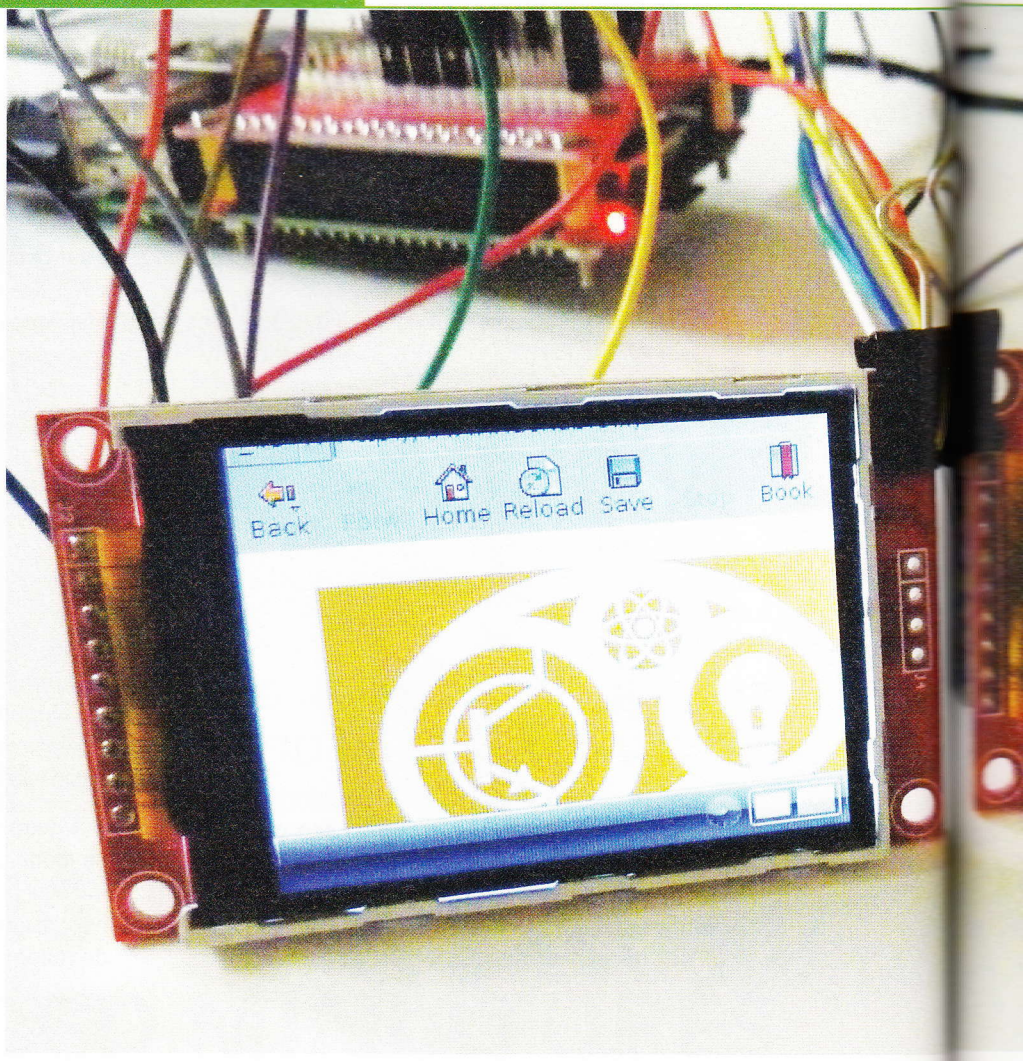
Ceci n'est pas une mince affaire, car l'intégration de codes et de fonctionnalités dans le noyau Linux officiel est un processus long et difficile visant à assurer un niveau important de qualité et surtout à vérifier que les changements ne cassent rien du travail qui a déjà été accompli depuis plus de deux décennies (diantre, le temps passe vite !).

Pour l'heure, le support *fbtft* est présent dans le noyau, mais dans sa branche dite « staging » où il poursuit son évolution, maintenu par Noralf Trønnes et Thomas Petazzoni. La branche staging est destinée à accueillir les pilotes qui ne sont pas encore totalement prêts pour l'intégration, mais de qualité suffisante pour être mis à disposition des utilisateurs. Les pilotes présents dans « staging » font partie de l'arbre de développement principal (*Linux kernel tree*), ce qui les rend plus faciles à utiliser.

L'autre conséquence de ce mode d'intégration est le fait qu'il n'est plus forcément nécessaire, pour un utilisateur, de recompiler un noyau pour



La connexion des deux écrans est un peu fastidieuse, mais le résultat est là. Un X démarré en dual-screen avec deux bureaux sous le gestionnaire de fenêtre e17 (Enlightenment 0.17+). À gauche, le navigateur Dillo et à droite une fenêtre de terminal.



Disposer de deux écrans de ce genre n'est qu'un petit plus pour une utilisation de type environnement graphique. La résolution de 320×240 avec uniquement 262K ou 65K couleurs ne permet pas vraiment de faire des miracles même en multipliant par deux, mais après tout nous n'avons sous les yeux qu'une dizaine d'euros...

profiter des nouveautés. Si les pilotes de staging sont compilés dans le noyau, alors ils sont disponibles pour l'utilisateur et c'est précisément le cas du noyau utilisé par Raspbian sur Raspberry Pi. Les pilotes fbtft sont donc désormais présents dans le système, s'il est à jour.

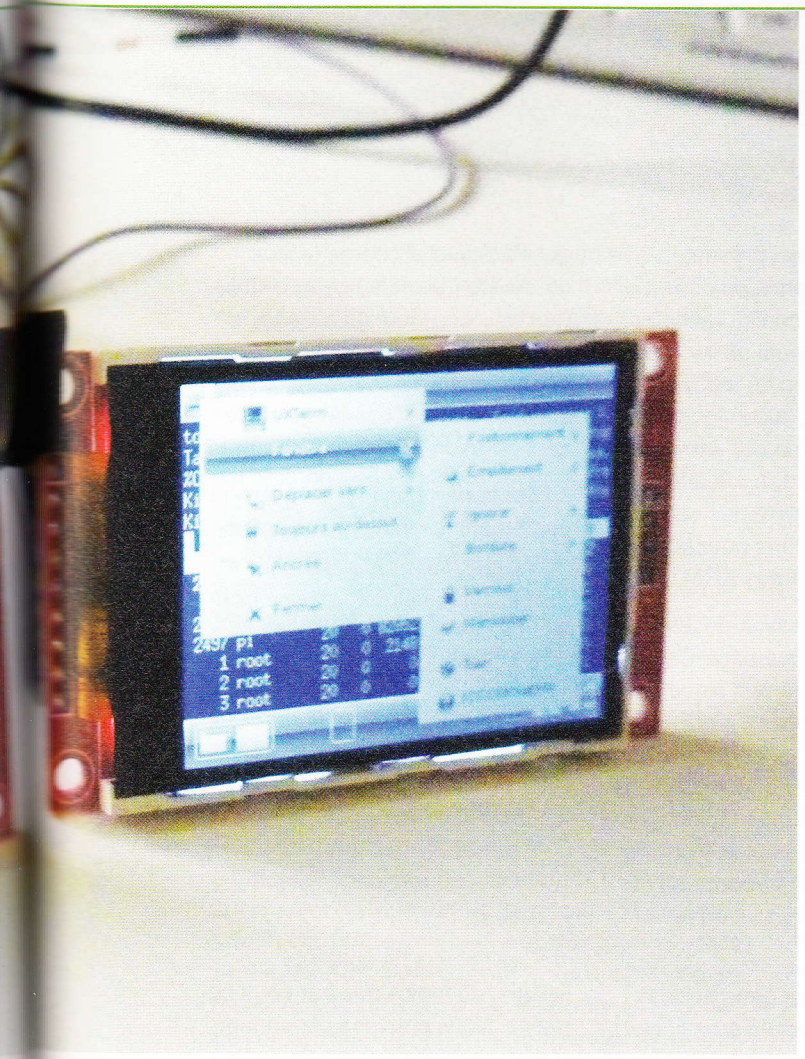
Enfin pour conclure cette introduction, j'aimerai simplement préciser quelque chose qui me paraît très important. Le noyau Linux est le résultat du travail de centaines de développeurs, de genre, d'origine, et d'orientations (politique, sociale, sexuelle) énormément diversifiées et littéralement disséminées à travers le monde. Pourtant le développement est exemplaire, les fonctionnalités intégrées admirables et le niveau de qualité du code excessivement élevé. Nombreux sont ceux qui voient dans Linux et GNU/Linux de formidables réussites techniques, mais c'est, selon moi, surtout une réussite humaine et la parfaite démonstration qu'il est effectivement possible de réaliser de grandes choses, de manière totalement ouverte, libre et communautaire (dans le sens « travail en commun » pas « communauté hippie »). Bien sûr cette constatation ne s'applique pas seulement au noyau Linux, mais celui-ci et ses 18 millions de lignes de code produites et maintenues par un ensemble de quelques 5000 développeurs (personnes ou sociétés) est, selon moi, l'exemple ultime.

2. LE DEVICE TREE

Une autre nouveauté, qui n'en est pas vraiment une, est l'intégration du support du *device tree* dans le système maintenant standard pour Raspberry Pi. C'est une nouveauté pour la Pi, mais pas pour Linux. Pour comprendre l'intérêt de ce système, il est important de se souvenir, tout d'abord, comment les choses fonctionnaient il y a bien longtemps.

Un ordinateur n'est pas différent d'un microcontrôleur d'un point de vue architectural. Nous avons un processeur (unité de calcul), de la mémoire et des périphériques. Le processeur exécute le code machine présent en mémoire pour manipuler des données et utiliser les périphériques. C'est simplissime.

Pendant longtemps, le code permettant de gérer les périphériques était intégré dans le système. Un Commodore 64 (même s'il n'y avait pas de système d'exploitation, mais simplement des routines en ROM) n'avait pas besoin de logiciels annexes pour permettre



l'utilisation du clavier par exemple. Bien entendu, il n'y avait qu'un seul type de clavier utilisable sur cette machine et donc aucun intérêt à envisager une architecture modulaire. C'était l'époque des ordinateurs familiaux et personne ne parlait de compatibilité.

Avec les premiers PC et en particulier les PC AT (processeurs 80286), il est devenu courant d'ajouter des cartes dans la machine afin d'étendre ses fonctionnalités. L'exemple typique est celui de la carte son *Creative Labs SoundBlaster* (je suis sûr que « 220 7 1 » rappelle des souvenirs à certains) ou Gravis Ultrasound (la bonne vieille GUS). L'un des avantages du PC était sa modularité, mais cela impliquait aussi que le code prenant en charge le nouveau matériel devait pouvoir s'intégrer au système. Sont ainsi apparus les pilotes de périphériques et leur florilège de conflits et de problèmes de configuration.

Puis est arrivé la révolution (ou la bonne blague diront certains) du *Plug and Play* promettant à l'utilisateur de se débarrasser de ces problèmes. L'idée était simple puisqu'il s'agissait pour l'ordinateur de déterminer tout seul comment gérer un périphérique, le configurer et trouver les pilotes adaptés, le tout sans l'intervention de l'utilisateur. On branchait et ça marchait... ou pas.

Il faut toutefois faire la différence entre la perception qu'a l'utilisateur et la mécanique en œuvre pour la machine. Pour accomplir ce petit miracle, l'ordinateur doit pouvoir identifier le périphérique ou disposer d'un moyen de l'interroger sur son identité. PCI et USB sont deux exemples de technologies permettant ce genre de choses, car intégrant un système de négociation et d'identification duquel découle tout le reste.

Aujourd'hui, il n'y a guère plus rien d'étonnant à voir immédiatement fonctionner une clé USB, un clavier ou encore une carte Ethernet PCIe dès sa connexion. Si le pilote est présent dans le système, il est tout simplement utilisé et ça s'arrête là (au moins pour une utilisation basique). Le noyau Linux fait cela très bien pour une très vaste majorité de périphériques, parfois bien mieux que d'autres systèmes.

Mais en dehors du monde PC/Mac, tous les périphériques ne sont pas Plug and Play. Si nous prenons le cas de notre écran LCD TFT sur bus SPI, il n'y a tout simplement aucun moyen pour le système de déterminer tout seul la façon dont il est connecté et le pilote à utiliser : il faut que l'utilisateur précise explicitement ces informations.

Dans un système GNU/Linux, cette prise en charge se faire en chargeant un pilote en mémoire sous la forme d'un module. On peut ainsi préciser le pilote, le modèle d'écran, le bus SPI à utiliser, les broches concernées et tout un tas de paramètres (taux de rafraîchissement, vitesse, verbosité, rotation, etc.).

Un module est, grossièrement, un morceau de noyau qui peut être intégré dynamiquement. Techniquement, ce n'est pas différent du même pilote directement présent dans le noyau. Il est d'ailleurs parfaitement possible de configurer et compiler un noyau, sans gestion de module,

embarquant uniquement le support du matériel présent. C'était quelque chose d'assez courant pendant bien des années pour les systèmes embarqués où la configuration est fixe et invariable, mais cela impliquait également qu'il y avait autant de combinaisons de configurations que de modèles de cartes. Pire encore, dans la plupart des cas, la configuration précise devait être faite dans le code lui-même par une description sous la forme d'une structure (`platform_device`). Avec l'augmentation du nombre et de la diversité des cartes, ceci est rapidement devenu ingérable, y compris pour les développeurs Linux travaillant sur les plateformes ARM.

La solution a pris la forme d'une externalisation de la description du matériel non détectable automatiquement. Ces informations sont donc sorties du noyau et ont pris place dans un fichier distinct : le *device tree blob* (blob pour *Binary Large OBject*, un gros tas de données binaires). Il est donc devenu bien plus simple de prendre en charge les périphériques sans avoir à toucher au code du noyau et donc de plus rapidement adapter un système d'une configuration à une autre. Par effet de bord, ceci a permis également de factoriser énormément de code et de rationaliser les développements tout en améliorant la qualité. Bref, que du bonheur !

Et comme les développeurs Linux sont des gens brillants et fantastiques, cette solution n'est pas un simple gros fichier de configuration, mais une structure permettant une grande modularité. Au final, le blob est un fichier `.dtb` chargé en même temps que le noyau Linux au démarrage et, vous ne le savez peut-être pas, votre carte Raspberry Pi démarre très certainement ainsi.

3. UN PEU D'EXPLORATION POUR COMPRENDRE

Comme le précise un billet sur le blog de la fondation Raspberry Pi, les dernières versions des systèmes Raspbian et NOOBS (qui installe en fait Raspbian) utilisent le *device tree*. Il ne s'agit pas encore d'un support complet puisqu'une partie des périphériques est encore décrite dans le noyau lui-même, mais c'est une première évolution qui permettra de faciliter la gestion des *hats* (chapeaux), sortes d'équivalents aux *shields* Arduino pour la Pi.

Votre Raspberry Pi démarre en utilisant un code en ROM qui charge le fichier `bootcode.bin` depuis la partition FAT de la carte SD ou microSD. Ce fichier, le bootloader de second niveau, charge alors `start.elf` (anciennement `loader.bin`) qui s'exécute et charge le blob du *device tree* et le noyau Linux (`kernel.img`) tout en utilisant le contenu de `cmdline.txt` comme argument pour son exécution et le fichier `config.txt` en guise de configuration.

Sur la partition FAT se trouvent également les fichiers suivants :

- `bcm2708-rpi-b.dtb` : le fichier binaire du *device tree* pour le modèle B ;
- `bcm2708-rpi-b-plus.dtb` : celui pour le B+ ;
- `bcm2708-rpi-cm.dtb` : celui pour le Raspberry Pi Compute Module ;
- `bcm2709-rpi-2-b.dtb` : et celui pour la carte Raspberry Pi 2.

Ces fichiers ne sont pas lisibles, du moins pas par nous humains. Nous pouvons cependant utiliser un outil pour manipuler le fichier et le convertir en device tree « source » : `dtc` (du paquet `device-tree-compiler`).

On peut alors convertir le fichier en utilisant la commande `dtc -I dtb -O dts -o pi2.dts /boot/bcm2709-rpi-2-b.dtb`. Les arguments sont :

- `-I` le format en entrée, `dtb` pour le fichier binaire ;
- `-O` le format en sortie, `dts` pour le fichier source (et donc texte lisible) ;
- `-o` pour préciser le nom de fichier en sortie ;
- et enfin le nom du fichier en entrée (ici le fichier binaire).

Nous obtenons ainsi un fichier `pi2.dts` dans le répertoire courant, que nous pouvons

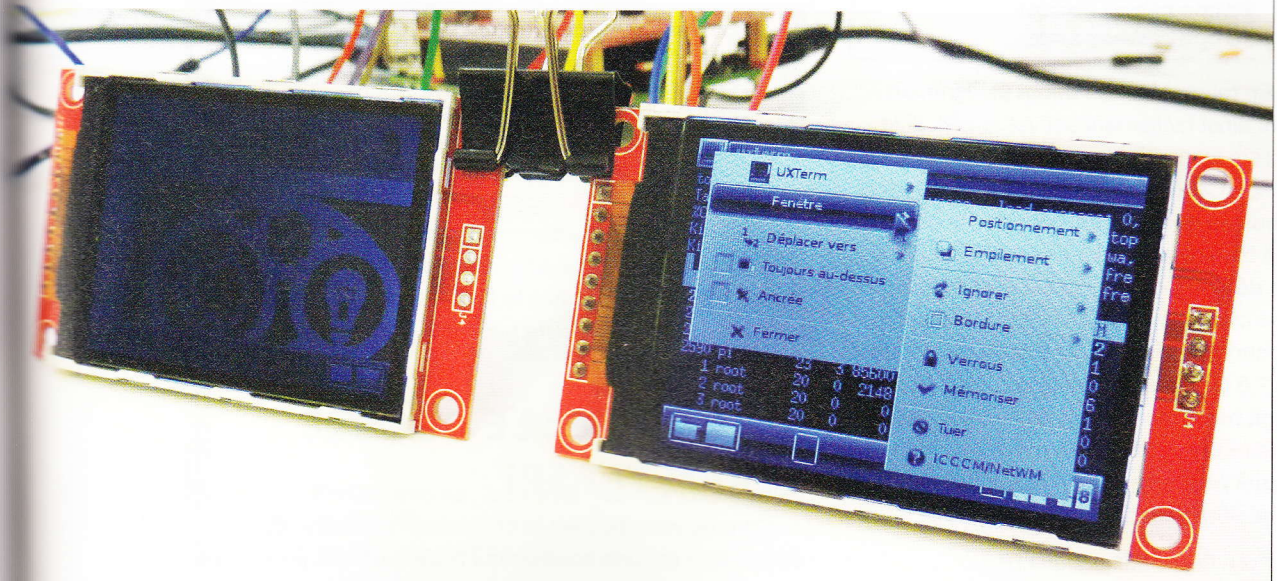
consulter ou modifier. Si vous souhaitez simplement voir le contenu de la structure du device tree, l'outil `fdtdump` pourra être utilisé avec `fdtdump /boot/bcm2709-rpi-2-b.dtb` ou plus judicieusement `fdtdump /boot/bcm2709-rpi-2-b.dtb | less`. Notez qu'en procédant à l'opération inverse (DTS vers DTB (compilation), il est recommandé d'utiliser l'option `-@` permettant de gérer les symboles.

En jetant un œil au contenu, nous avons une vision de la structure arborescente (d'où le terme *tree* signifiant « arbre ») composée de branches et de nœuds. Tout le matériel « statique » est ainsi décrit, morceau par morceau. Nous ne rentrerons pas dans le détail de cette structure et de sa sémantique ici. François Mocq couvre très bien le sujet dans un billet sur son blog (**http://www.framboise314.fr/un-point-sur-le-device-tree/**) et vous pourrez compléter cette lecture avec le support de présentation de Thomas Petazzoni sur le sujet (**http://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf**) ainsi qu'avec la documentation Raspberry Pi (**https://www.raspberrypi.org/documentation/configuration/device-tree.md**).

4. OVERLAYS ET PARAMÈTRES DU DEVICE TREE

En portant votre attention sur le fichier `config.txt` de la partition FAT vous remarquez que deux nouvelles directives ont fait leur apparition avec la mise en place du support device tree de la Pi : `dtoverlay` et `dtparam`.

Pour changer le contenu du device tree et donc personnaliser le support des périphériques de la carte, vous pouvez utiliser le fichier binaire, le convertir en fichier source, l'éditer pour faire vos modifications et enfin le recompiler (toujours avec `dtc`, mais en inversant les valeurs `-I` et `-O`) en fichier binaire. Ça fonctionne, mais ce n'est pas très pratique, en particulier lorsqu'on tâtonne pour obtenir une bonne configuration. De plus, pour créer un binaire à partir de sources, on peut utiliser des noms ou macros simplifiant l'écriture, mais ces facilités disparaissent dans le sens inverse (tout



L'un des gros problèmes des écrans LCD et en particulier des modèles économiques bas de gamme est leur faible angle de vue. On voit ici clairement que l'écran de gauche est presque totalement illisible sous cet angle.

comme les commentaires) et vous vous retrouvez avec des valeurs, des registres et des adresses bien plus difficiles à lire et interpréter. Si vous voulez vraiment jouer avec le device tree, il est recommandé de démarrer avec le fichier DTS original (inclus dans les sources du noyau) plutôt que le résultat obtenu à partir du DTB.

Deux solutions sont alors possibles pour apporter une modification au device tree sans tout régénérer. Le fichier **config.txt** est utilisé par **start.elf** pour charger les éléments nécessaires au démarrage et donc le device tree. Rappelons que l'objet même du device tree binaire est d'être une représentation structurée du matériel à prendre en charge. Rien d'étonnant donc qu'il soit possible d'y faire des modifications à la volée et c'est précisément ce que fait **start.elf**. Il permet ainsi d'activer ou désactiver certaines parties de la configuration avec la directive **dtparam**. Par exemple, dans le fichier **config.txt** on peut ainsi trouver les lignes suivantes, par défaut en commentaire :

```
#dtparam=i2c_arm=on
#dtparam=i2s=on
#dtparam=spi=on
```

En retirant le **#** en début de ligne, on peut ainsi facilement activer le SPI, l'i2c ou l'i2s (alias IIS, pour les périphériques audio). Ceci permet à **start.elf** d'activer le bus en question qui est dans le device tree binaire en **status = "disabled"** par défaut. Comment **start.elf** procède à ce changement ? J'aimerai vraiment répondre, mais les sources de ce binaire de 2,5 Mo ne sont pas disponibles et il ne s'agit pas d'un élément en logiciel libre (cf. **LICENCE.broadcom** en guise de piqûre de rappel pour se souvenir que les Raspberry Pi ne sont pas 100% open source).

Pour des modifications plus importantes, nous avons la directive **dtoverlay** permettant de spécifier un nom de fichier device tree binaire contenant une structure s'ajoutant au device tree **bcm270*.dtb**. On peut voir ceci comme un device tree partiel destiné à faciliter la gestion de l'écosystème de *hats* en évitant aux utilisateurs de devoir recompiler un device tree pour chaque circuit/shield/module enfichable : il suffit de spécifier une ligne **dtoverlay=** suivie du nom de fichier à utiliser, en omettant **-overlay.dtb**.

Sur la partition FAT de la carte SD ou microSD, vous trouverez en effet un répertoire **overlays/** contenant un ensemble de fichiers livrés par défaut :

```
$ ls /boot/overlays
ads7846-overlay.dtb
bmp085_i2c-sensor-overlay.dtb
dht11-overlay.dtb
[...]
uart1-overlay.dtb
w1-gpio-overlay.dtb
w1-gpio-pullup-overlay.dtb
```

Ainsi qu'un fichier **README** décrivant l'utilisation et les paramètres de chaque *overlay*. Le principe consiste à spécifier un overlay en précisant les paramètres ainsi : **dtoverlay=lirc-rpi:gpio_out_pin=16,gpio_in_pin=17,gpio_in_pull=down**, ou à procéder en plusieurs fois de façon à charger l'overlay puis préciser les arguments avec **dtparam** :

```
dtoverlay=lirc-rpi
dtparam=gpio_out_pin=16
dtparam=gpio_in_pin=17
dtparam=gpio_in_pull=down
```

lirc-rpi ici correspond au fichier **/boot/overlays/lirc-rpi-overlay.dtb** permettant la prise en charge du support de réception infrarouge Lirc. Les **dtparam** spécifient les ports utilisés, s'ils sont différents de ceux par défaut déjà présents dans le fichier DTB. Au passage, un petit **fdtdump** est toujours très instructif...

5. L'EXCUSE : LE DOUBLE ÉCRAN TFT SPI

Comme vous pouvez le constater, l'utilisation du device tree est une solution qui règle bien des problèmes et bien des tracas concernant les périphériques « non découvrables ». La fondation Raspberry Pi repose ainsi

sur un mécanisme très performant créé par les développeurs Linux et l'étend en ajoutant ce qui permet d'atteindre la modularité souhaitée pour le support simplifié des cartes d'extensions appelées *HATs*, comme « chapeaux » en anglais, mais officiellement acronymes de « *Hardware Attached on Top* » (« matériel attaché au-dessus », sérieux ?!).

Mais cela règle indirectement un autre problème. Nous avons par le passé détaillé la connexion et la configuration d'un écran LCD TFT SPI. L'opération est relativement simple, mais que se passe-t-il si nous voulons brancher non pas un écran, mais deux ? En utilisant le module **fbtft_device**, il n'y a tout simplement pas de solution. Celui-ci peut utiliser le premier périphérique SPI (broche CS0) ou le second (CS1), mais pas les deux et nous ne pouvons charger qu'une seule fois le module...

La seule solution possible jusqu'alors consistait à intégrer le pilote « en dur » dans le noyau et donc de le recompiler, ce qui n'est ni aisé ni vraiment amusant, surtout si l'on ne souhaite pas compiler directement sur la Raspberry Pi, mais plutôt sur un PC puissant (cross-compilation).

Mais le device tree change complètement la façon d'envisager le problème. Il nous suffit, en effet, d'écrire un fichier device tree source (DTS) en nous inspirant de ceux existants et d'en faire un binaire (DTB) avec **dtc** pour enfin l'utiliser comme overlay via **config.txt**.

Voici donc le fichier en question :

```
/*
* Device Tree overlay SPI TFT ili9341
*/

/dts-v1/;
/plugin/;

/ {
  // ça marche pour Pi, Pi CM et Pi 2
  compatible = "brcm,bcm2835", "brcm,bcm2708", "brcm,bcm2709";

  // activation du bus SPI
  fragment@0 {
    target = <&spi0>;
    __overlay__ {
      status = "okay";

      // on désactive spidev pour périph 0 et 1
      spidev@0{
        status = "disabled";
      };
      spidev@1{
        status = "disabled";
      };
    };
  };

  // Nous avons besoin de GPIOs
  fragment@1 {
    target = <&gpio>;
    __overlay__ {
      rpi_display_pins: rpi_display_pins {
```

```
        // broches pour led, reset et dc
        brcm,pins = <18 25 24>;
        brcm,function = <1 1 1>; /* en sortie */
      };
      rpi_display_pins1: rpi_display_pins1 {
        // broches pour led, dc et reset
        brcm,pins = <12 16 19>;
        brcm,function = <1 1 1>; /* en sortie */
      };
    };
};

// configuration des périphériques sur le bus SPI
fragment@2 {
  target = <&spi0>;
  __overlay__ {
    #address-cells = <1>;
    #size-cells = <0>;

    rpidisplay: rpi-display@0{
      compatible = "ilitek,ili9341";
      // adressé pas CS0
      reg = <1>;
      pinctrl-names = "default";
      pinctrl-0 = <&rpi_display_pins>;

      // arguments pour le pilote
      spi-max-frequency = <48000000>;
      rotate = <90>;
      bgr;
      fps = <30>;
      buswidth = <8>;
      reset-gpios = <&gpio 25 0>;
      dc-gpios = <&gpio 24 0>;
      led-gpios = <&gpio 18 1>;
      debug = <0>;
    };

    rpidisplay1: rpi-display@1{
      compatible = "ilitek,ili9341";
      // adressé pas CS0
      reg = <0>;
      pinctrl-names = "default";
      pinctrl-0 = <&rpi_display_pins1>;

      // arguments pour le pilote
      spi-max-frequency = <48000000>;
      rotate = <90>;
      bgr;
      fps = <30>;
      buswidth = <8>;
      reset-gpios = <&gpio 19 0>;
      dc-gpios = <&gpio 16 0>;
```

```
            led-gpios = <&gpio 12 1>;
            debug = <0>;
        };
    };
  };
  __overrides__ {
    speed =   <&rpidisplay>,"spi-max-frequency:0";
    rotate =  <&rpidisplay>,"rotate:0";
    fps =     <&rpidisplay>,"fps:0";
    debug =   <&rpidisplay>,"debug:0";
  };
};
```

Il nous suffit alors d'un petit **`sudo dtc -@ -I dts -O dtb -o /boot/overlays/lcdgeneric22-overlay.dtb mon_fichier.dts`** pour produire directement le fichier DTB au bon emplacement puis éditer **`/boot/config.txt`** en ajoutant la ligne :

```
dtoverlay=lcdgeneric22
```

Le démarrage suivant de la Pi nous donnera accès à trois *framebuffer* avec **`fb0`** pour la sortie HDMI, et **`fb1`** et **`fb2`** pour les écrans :

```
$ fbset -fb /dev/fb1

mode "320x240"
    geometry 320 240 320 240 16
    timings 0 0 0 0 0 0 0
    nonstd 1
    rgba 5/11,6/5,5/0,0/0
endmode

$ fbset -fb /dev/fb2

mode "320x240"
    geometry 320 240 320 240 16
    timings 0 0 0 0 0 0 0
    nonstd 1
    rgba 5/11,6/5,5/0,0/0
endmode
```

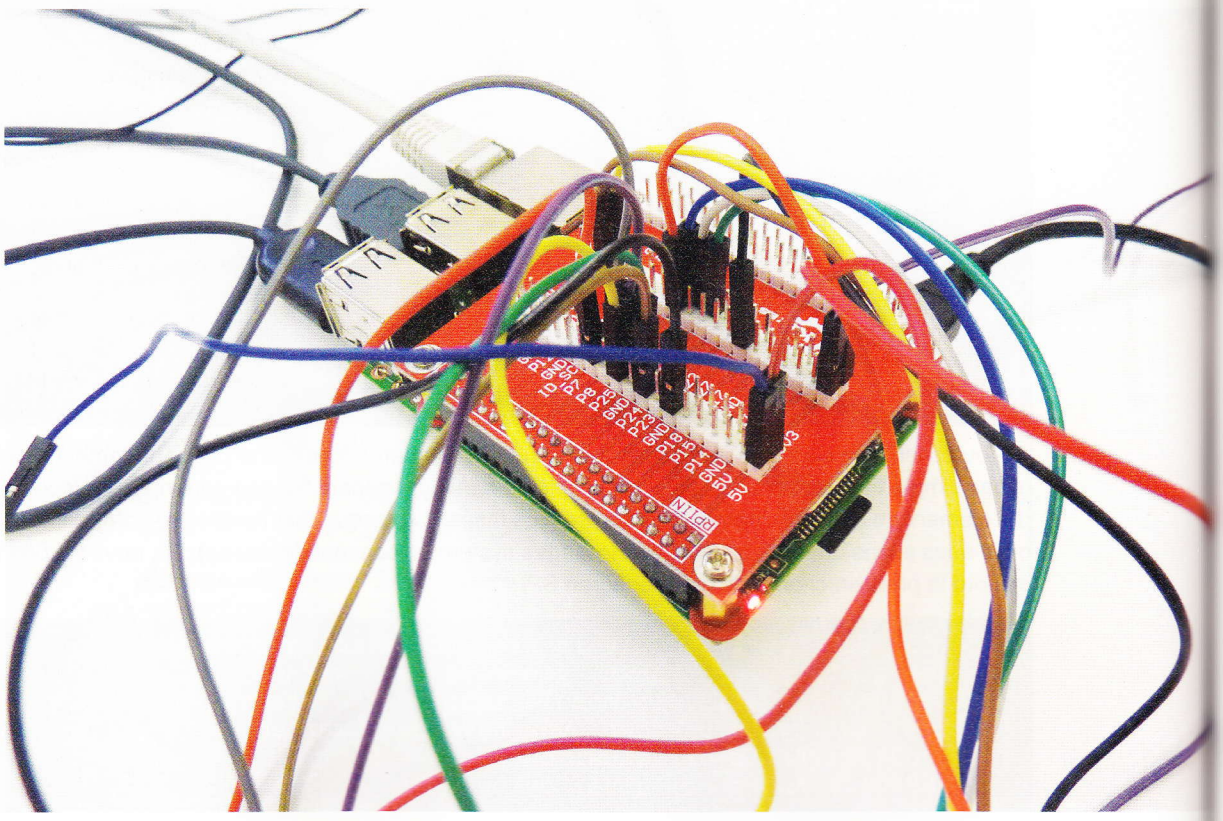
Toutes les manipulations vues dans les précédents numéros fonctionneront parfaitement, nous avons simplement un écran en plus. Un problème bénin cependant fait son apparition et découle directement du pilote lui-même. Il passe presque inaperçu puisque tout fonctionne correctement, mais nous avons une erreur dans les messages systèmes (commande **`dmesg`**). Là, nous voyons tout d'abord la prise en charge du premier écran SPI :

```
[4.052762] fbtft: module is from the staging directory,
  the quality is unknown, you have been warned.
[4.072049] fb_ili9341: module is from the staging directory,
  the quality is unknown, you have been warned.
[4.089737] fbtft_of_value: buswidth = 8
[4.096042] fbtft_of_value: debug = 0
```

```
[4.103243] fbtft_of_value: rotate = 90
[4.110609] fbtft_of_value: fps = 30
[4.509070] graphics fb1: fb_ili9341 frame buffer, 320x240,
  150 KiB video memory, 16 KiB DMA buffer memory, fps=33,
  spi0.0 at 48 MHz
```

Le premier message est parfaitement normal. Les pilotes dans staging sont toujours explicitement signalés en tant que tels. Le problème survient à la prise en charge du second périphérique :

```
[4.535186] fbtft_of_value: buswidth = 8
[4.541379] fbtft_of_value: debug = 0
[4.546502] fbtft_of_value: rotate = 90
[4.552708] fbtft_of_value: fps = 30
[4.862705] WARNING: CPU: 0 PID: 277 at fs/sysfs/dir.c:31
  sysfs_warn_dup+0x68/0x84()
[4.862711] sysfs: cannot create duplicate filename
  '/class/backlight/fb_ili9341'
[4.863763] fb_ili9341 spi0.1: cannot register
  backlight device (-17)
[5.480285] graphics fb2: fb_ili9341 frame buffer, 320x240,
  150 KiB video memory, 16 KiB DMA buffer memory,
  fps=33, spi0.1 at 48 MHz
```



Utiliser deux périphériques SPI avec leurs connexions supplémentaires finit forcément et invariablement en salade de câbles, même en utilisant un « hat » permettant de démultiplier les connexions (qui n'en est pas vraiment un puisqu'un HAT, au sens strict du terme, intègre une EEPROM contenant sa configuration).

Un avertissement nous est donné concernant non pas l'écran lui-même ou le framebuffer, mais le rétroéclairage. En effet, une sortie de la Raspberry Pi est utilisée pour la broche LED des afficheurs. Cette prise en charge passe par le sous-système fournissant `/sys/class/backlight`. Le premier écran crée un point d'entrée dans `/sys` et lorsque le second est pris en charge, ce pseudo-fichier existe déjà et ne peut donc être créé une seconde fois. Utiliser un chemin différent pour chaque instance de ce type de périphérique serait une correction possible, mais ceci doit être fait dans les sources du pilote. En ce qui nous concerne, cela reste un problème mineur et symptomatique d'un pilote staging (les développeurs ne peuvent pas penser immédiatement à tout, c'est pour cela qu'on faire des rapports de bogues).

Comme notre double affichage fonctionne, pourquoi ne pas utiliser l'interface graphique Xorg en *dual-screen*. Oubliez `FRAMEBUFFER=/dev/fb1 startx`, il nous faut quelque chose de plus conséquent pour ce type de choses. Nous créons donc un fichier de configuration pour Xorg, `/etc/X11/xorg.conf` :

```
Section "Device"
    Identifier  "FBDEV 1"
    Driver      "fbdev"
    Option      "fbdev" "/dev/fb1"
EndSection

Section "Device"
    Identifier  "FBDEV 2"
    Driver      "fbdev"
    Option      "fbdev" "/dev/fb2"
EndSection

Section "Screen"
    Identifier  "ILI0"
    Device      "FBDEV 1"
    Monitor     "Monitor name 0"
EndSection

Section "Screen"
    Identifier  "ILI1"
    Device      "FBDEV 2"
    Monitor     "Monitor name 1"
EndSection

Section "ServerLayout"
    Identifier  "Default Layout"
    Screen      0 "ILI0"
    Screen      1 "ILI1" LeftOf "ILI0"
EndSection
```

Et voilà ! À présent, un simple `startx` et nous voici avec un bureau sur deux écrans. Nous pouvons même déplacer les fenêtres de l'un à l'autre ou les étendre sur l'ensemble de la surface disponible.

Notez le `LeftOf` dans la section `ServerLayout` du fichier de configuration. Cette directive nous permet de spécifier qu'un écran est à gauche de l'autre. Nous pouvons également utiliser `RightOf` (à droite de), `Above` (au-dessus) ou encore `Below` (sous). `Relative` nous permettra même d'ajuster précisément l'alignement si les deux écrans sont décalés, afin que le pointeur de la souris reste à la même hauteur physique en passant de l'un à l'autre.

POUR CONCLURE

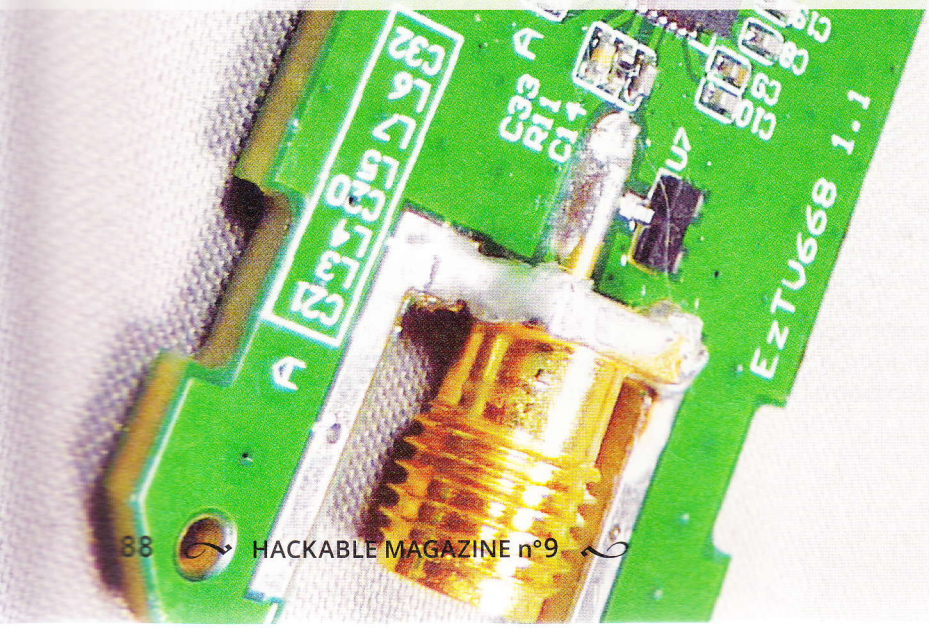
Nous n'avons vu qu'un exemple d'utilisation du device tree mais, vous l'avez compris, au-delà de la simple prise en charge de hats, ceci ouvre des perspectives très intéressantes. Le nombre de pilotes Linux pour des composants interfacés en i2c ou en SPI est impressionnant et il devient très simple de se composer un système prenant en charge une ribambelle de capteurs de toutes sortes, sans jamais avoir à jongler avec des modules.

Les bénéfices pour les utilisateurs moins amateurs de configuration avancée sont également tout à fait notables. Au final, lorsque la gamme d'extensions matérielles grossira davantage, la configuration se limitera à la connexion du module et l'édition du `config.txt`. Pour l'heure, les hats ne sont pas nombreux et le contenu de `/boot/overlays/` est majoritairement en relation avec des composants simples (RTC, bus 1-Wire, écran SPI, DAC audio, etc.), mais nul doute que nous aurons bientôt autant de hats qu'il existe de shields pour les cartes Arduino... DB

RTL POWER OU COMMENT SURVEILLER LES ONDES AVEC VOTRE RASPBERRY PI

Denis Bodor

Si vous êtes propriétaire d'un récepteur RTL-SDR (voir *Hackable n°2*) et avez fait connaissance avec le monde de la radio logicielle, les termes « graphique waterfall » vous disent certainement quelque chose. Que diriez-vous d'obtenir la même chose, plus large, plus long et surtout sans rien faire pendant que vous dormez ? En d'autres termes, il s'agit de garder un œil sur les ondes et d'obtenir une image de tout ce qui rayonne autour de vous...



Lorsque vous utilisez un logiciel SDR comme Gqrx ou SDR#, l'interface vous présente généralement deux représentations des signaux reçus. Le graphique FFT montre l'ensemble de la bande de réception avec en abscisse les fréquences et en ordonnée l'intensité du signal pour chacune d'elles. Ceci vous permet de voir où se situent plus ou moins précisément les émissions captées en temps réel.

Mais nombreux sont les signaux qui ne sont pas constants. Comme l'ont montré les différents articles déjà parus dans le magazine, la plupart du temps les émissions sont ponctuelles. L'exemple de la télécommande de garage est explicite, il n'y a une émission sur 27 Mhz que lorsqu'on appuie sur un bouton, le reste du temps c'est le silence radio. Que se passe-t-il alors si vous n'aviez pas l'œil rivé au graphique FFT à ce moment précis ?

Pour régler ce point, une seconde représentation de la bande de réception est affichée avec, toujours en abscisse, les fréquences, mais en ordonnée le temps. La puissance du signal est quant à elle représentée par une couleur ou un niveau de gris. L'ensemble du graphique défile vers le bas vous affichant en permanence cette représentation pour les quelques secondes qui viennent de s'écouler. Ainsi, une émission ponctuelle sur une fréquence donnée laissera une trace même si vous ne l'avez pas vue sur la représentation FFT. Vous savez qu'il y a eu un signal et vous pouvez agir en conséquence pour éventuellement capter, démoduler et décoder le prochain.

La représentation waterfall dans un logiciel est très pratique, mais pour certains usages elle souffre d'un certain nombre de limitations :

- nous sommes contraints à la largeur de bande du récepteur (*bandwidth* liée à la fréquence d'échantillonnage) et ne pouvons donc observer qu'une mince portion des fréquences sur lesquelles peut effectivement se caler le récepteur ;
- nous ne pouvons voir qu'une petite plage temporelle. Éloignez-vous de l'écran une minute et vous avez peut-être raté quelque chose d'important qui n'arrive qu'une fois par jour ;
- la résolution de votre écran vous empêche de voir des émissions trop brèves, il n'y a tout simplement pas assez de pixels verticalement (ou le défilement est trop rapide) ;
- la précision du résultat en fréquence est également liée à la résolution (horizontale), un *bin* ou échantillon spectral ne peut être inférieur à un pixel ;
- il n'y a pas assez de niveaux de gris ou de couleurs dans une palette pour afficher précisément la puissance des signaux reçus.

Bien entendu, tout cela découle directement des impératifs d'affichage du logiciel utilisé puisque l'information a un caractère dynamique.

Un outil livré avec le support RTL-SDR utilise une approche différente (et typiquement UNIX, faire une chose, mais la faire bien) : `rtl_power`. L'idée n'est plus de créer un graphique waterfall en temps réel, mais de simplement surveiller une bande de fréquences et d'enregistrer le résultat, à la résolution souhaitée, sur le disque dur sous la forme d'un fichier texte au format CSV (de l'anglais *Comma-Separated Values*). Ce format simple et universel bien connu des utilisateurs de tableurs est juste une liste de valeurs séparées par des virgules.

Cette approche fait tomber presque toutes les restrictions précédentes :

- bande de fréquences uniquement limitée à celle de votre tuner ;
- aucune limite de temps d'enregistrement (sauf au regard de l'espace disque disponible) ;
- résolution en fréquence (*bins*) illimitée ;
- le niveau de signal n'est plus dépendant de l'affichage ;
- et surtout les ressources processeur et mémoire nécessaires pour faire fonctionner l'outil n'ont rien à voir avec celles d'un logiciel SDR comme Gqrx, SDR# ou GNUradio.

Ce dernier point est très important, car `rtl_power` ne procède à aucun traitement de signal et se contente d'écouter et d'enregistrer des valeurs dans un fichier. Il est

donc possible de le faire fonctionner sur une toute petite machine placée à un endroit où un PC puissant ne saurait trouver sa place. Il pourra donc s'agir d'un vieil ordinateur portable, d'une tour composée de matériel de récupération ou... d'une carte Raspberry Pi bien sûr ! Alimentez votre Pi avec un bloc batterie 5V (*USB power bank*) et vous obtenez un système autonome et mobile.

La logique utilisée pour cet article sera donc la suivante : installer RTL-SDR sur une Raspberry Pi (de préférence B, B+ ou 2), collecter les informations avec **rtl_power**, transférer les données sur un PC et générer un énorme graphique waterfall couvrant une plage de fréquences et une durée impossible à obtenir autrement.

1. INSTALLATION RAPIDE DE RTL-SDR SUR RASPBERRY PI

Les « pilotes » et outils RTL-SDR ne sont, semble-t-il, pas disponibles dans les dépôts standards de Raspbian. Ce n'est pas grave, ce sera l'occasion de rapidement construire un paquet. Notez au passage que la quasi-totalité des tutoriels en ligne décrit une compilation et une installation « à la sauvage » de façon indépendante du système de gestion de paquets. Manipulations que je qualifierai très sommairement par un « c'est paaaaas bien ».

Faire propre n'est pas bien compliqué, permet d'installer tout ce qu'il faut là où il faut, et la manipulation tient en une poignée de commandes. Mais avant toutes choses, un conseil : NE BRANCHEZ PAS LE RECEPTEUR DVB-T À LA RASPBERRY PI ! Avant d'installer le paquet que nous allons créer, une connexion du périphérique USB provoquera sa prise en charge par le noyau, or la fonction première de l'adaptateur USB est d'être un récepteur TNT. Le noyau chargera donc naturellement des modules comme **dvb_usb_rtl28xxu**, **rtl2832**, **dvb_core**, **dvb_usb_v2**, etc., qui seraient utiles pour regarder la TV, mais bloquent l'utilisation en tant que récepteur SDR. Si c'est trop tard et que vous avez déjà fait preuve d'impatience, ce n'est pas grave, vous pouvez au choix décharger les modules avec **sudo rmmod** ou tout simplement retirer le périphérique et redémarrer la Pi.

Pour récupérer les sources du paquet, on pointe simplement son navigateur sur le site de Debian (**https://packages.debian.org**) et plus précisément sur la page concernant le paquet **rtl-sdr** dans sa version Sid (instable) afin d'avoir la dernière version existante (chez Debian, pas la version de développement OsmocomSDR). Sur la droite, nous avons une série de liens sous le titre « *Download Source Package* ». Nous utilisons alors ces liens avec la commande **wget** directement sur la Raspberry Pi :

```
$ cd
$ mkdir rtldeb
$ cd rtldeb

$ wget http://http.debian.net/debian/pool/main/r/rtl-sdr/rtl-sdr_0.5.3-5.dsc
$ wget http://http.debian.net/debian/pool/main/r/rtl-sdr/rtl-sdr_0.5.3.orig.tar.gz
$ wget http://http.debian.net/debian/pool/main/r/rtl-sdr/rtl-sdr_0.5.3-5.debian.tar.xz
```

Ces trois fichiers récupérés et placés dans le répertoire courant correspondent respectivement à la description du paquet source, aux sources d'origine et aux ajouts/modifications apportés par Debian. On repose sur ces trois éléments pour désarchiver le tout en utilisant :

```
$ dpkg-source -x rtl-sdr_0.5.3-5.dsc
gpgv: Signature faite le dim. 23 août 2015 19:46:06 CEST
  avec la clef RSA d'identifiant 1F44E090
gpgv: Impossible de vérifier la signature :
 clef publique introuvable
dpkg-source: avertissement: impossible de vérifier
  la signature sur ./rtl-sdr_0.5.3-5.dsc
dpkg-source: info: extraction
```

```
  de rtl-sdr dans rtl-sdr-0.5.3
dpkg-source: info: extraction
  de rtl-sdr_0.5.3.orig.tar.gz
dpkg-source: info: extraction
  de rtl-sdr_0.5.3-5.debian.tar.xz
[...]
dpkg-source: info: mise en place
  de improve-librtlsdr-pc-file
dpkg-source: info: mise en place
  de improve-scanning-range-parsing
```

Un message nous signale que la signature électronique embarquée dans la description ne peut être vérifiée. Il nous manque en effet la clé publique correspondante, mais ce n'est pas gravissime. Les sources sont tout de même décompressées et les modifications apportées sous la forme d'une série de *patchs*. Au final, nous obtenons un répertoire **`rtl-sdr-0.5.3/`** contenant tout le nécessaire ou presque. En effet, des éléments annexes doivent être ajoutés pour compiler et construire le paquet : les dépendances de construction. Une commande est à notre disposition pour les lister :

```
$ cd rtl-sdr-0.5.3
$ dpkg-checkbuilddeps
dpkg-checkbuilddeps : dépendances de construction
non trouvées : cmake libusb-1.0-0-dev
```

Il nous manque ici deux paquets : l'outil CMake orchestrant la compilation et les fichiers de développement de la bibliothèque libUSB 1.0. Nous pouvons installer cela rapidement avec un simple **`sudo apt-get install cmake libusb-1.0-0-dev`**. Ceci fait, il est temps de construire le paquet :

```
$ dpkg-buildpackage -b
dpkg-buildpackage: paquet source rtl-sdr
dpkg-buildpackage: version source 0.5.3-5
dpkg-buildpackage: source changé par A. Maitland
  Bottoms <bottoms@debian.org>
dpkg-buildpackage: architecture hôte armhf
dpkg-source --before-build rtl-sdr-0.5.3
fakeroot debian/rules clean
dh clean
   dh_testdir
[...]
dpkg-deb : construction du paquet " librtlsdr-dev "
  dans " ../librtlsdr-dev_0.5.3-5_armhf.deb ".
dpkg-deb : construction du paquet " librtlsdr0 "
  dans " ../librtlsdr0_0.5.3-5_armhf.deb ".
dpkg-deb : construction du paquet " rtl-sdr "
  dans " ../rtl-sdr_0.5.3-5_armhf.deb ".
dpkg-genchanges -b >../rtl-sdr_0.5.3-5_armhf.changes
dpkg-genchanges: envoi d'un binaire - aucune
  inclusion de code source
dpkg-source --after-build rtl-sdr-0.5.3
dpkg-buildpackage: envoi d'un binaire
  seulement (aucune inclusion de code source)
```

Au final, nous obtenons non pas un paquet, mais trois, dans le répertoire parent :

```
$ ls ../*.deb
../librtlsdr0_0.5.3-5_armhf.deb
../librtlsdr-dev_0.5.3-5_armhf.deb
../rtl-sdr_0.5.3-5_armhf.deb
```

Nous avons respectivement, la bibliothèque RTL-SDR sur laquelle repose les outils et permet à des logiciels comme Gqrx de contrôler le périphérique, le paquet de développement de cette même bibliothèque et enfin, les outils RTL-SDR eux-mêmes. Nous ne cherchons pas la petite bête et installons dans la foulée les trois paquets avec :

```
$ sudo dpkg -i ../*.deb
[...]
Paramétrage de librtlsdr0:armhf (0.5.3-5) ...
Paramétrage de librtlsdr-dev (0.5.3-5) ...
Paramétrage de rtl-sdr (0.5.3-5) ...
Traitement des actions différées
(" triggers ") pour " man-db "...
```

Le paquet **librtlsdr0** installe, parmi d'autres choses, les fichiers **/etc/modprobe.d/rtl-sdr-blacklist.conf** interdisant le chargement des modules/pilotes pour la réception TNT et **/lib/udev/rules.d/60-librtlsdr0.rules** qui permet d'ajuster automatiquement les permissions afin qu'un utilisateur standard (comme **pi**) puisse accéder au périphérique. C'est après l'installation de ces deux fichiers qu'il est alors possible de sereinement brancher le récepteur DVB-T à la Raspberry Pi.

On s'empressera de tester rapidement la prise en charge avec :

```
$ rtl_test
Found 1 device(s):
  0:  Realtek, RTL2838UHIDIR, SN: 00000001

Using device 0: Generic RTL2832U OEM
Found Rafael Micro R820T tuner
Supported gain values (29): 0.0 0.9 1.4 2.7 3.7
  7.7 8.7 12.5 14.4 15.7 16.6 19.7 20.7 22.9
  25.4 28.0 29.7 32.8 33.8 36.4 37.2 38.6 40.2
  42.1 43.4 43.9 44.5 48.0 49.6
[R82XX] PLL not locked!
Sampling at 2048000 S/s.

Info: This tool will continuously read from the
device, and report if samples get lost. If you
observe no further output, everything is fine.

Reading samples in async mode...
^CSignal caught, exiting!
User cancel, exiting...
```

Le matériel est correctement détecté, ici une clé à base de tuner Rafael Micro R820T, et le test fonctionne. Nous sommes prêts pour nous pencher sur **rtl_power** installé gracieusement en compagnie des autres outils.

2. AUCUNE LIMITE À MON POWER !

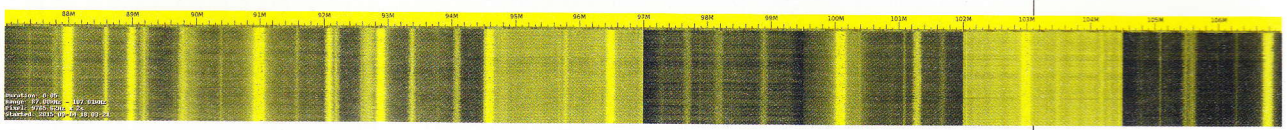
`rtl_power` n'est pas très difficile à utiliser, mais le nombre d'options disponibles pour améliorer le résultat demande quelques explications. De base, la principale option est celle permettant de spécifier la plage de fréquences à surveiller ainsi que la résolution `-f` :

```
$ rtl_power -f 87M:107M:12.5k FM.csv
Number of frequency hops: 8
Dongle bandwidth: 2500000Hz
Downsampling by: 1x
Cropping by: 0.00%
Total FFT bins: 2048
Logged FFT bins: 2048
FFT bin size: 9765.62Hz
Buffer size: 16384 bytes (3.28ms)
Reporting every 10 seconds
Found 1 device(s):
  0:  Realtek, RTL2838UHIDIR, SN: 00000001

Using device 0: Generic RTL2832U OEM
Found Rafael Micro R820T tuner
Tuner gain set to automatic.
Exact sample rate is: 2500000.107620 Hz
```

Cette option prend en argument un paramètre avec la syntaxe **`fréquence_départ:fréquence_fin:taille_bin`**. Alors que les deux premiers éléments sont évidents, le second, le pas (ou *bin* pour la largeur de l'échantillon spectral pour la FFT) est plus subtil. Il s'agit très grossièrement de la taille en hertz d'un pixel dans le rendu final.

Ici nous surveillons la bande radio FM commerciale avec un « pas » de 12,5 Khz largement suffisant pour voir les différentes stations. Notez les unités utilisées sur la ligne de commandes avec `G` pour gigahertz, `M` pour mégahertz et `k` pour kilohertz. Ceci est interchangeable et nous aurions tout aussi bien pu utiliser `87000k:0.107G:12500` pour décrire exactement la même chose.

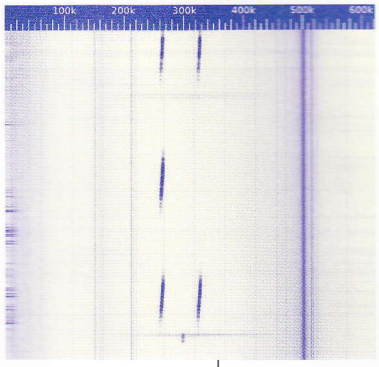


Exemple typique d'une représentation des données issues de rtl_power avec heatmap.py. Nous avons là la bande FM commerciale et on peut distinguer clairement chaque station.

En fin de ligne, le dernier argument est le nom du fichier dans lequel seront placées les données au format CSV. Ne l'oubliez pas, car dans le cas contraire ces données arriveront sur la sortie standard (le terminal). Ceci peut être intéressant dans certaines situations en redirigeant cette sortie vers un autre outil avec un `|` (*pipe*).

Enfin, sachez que `rtl_power` une fois lancé pourra être arrêté avec Ctrl+C. Il terminera alors la dernière « passe » en cours et vous rendra la main. Une autre solution consiste à utiliser l'option `-e` et en précisant une valeur en minutes (`30m`), en secondes (`42s`) ou en heures (`12h`). `rtl_power` s'arrêtera alors automatiquement une fois le temps écoulé. Ceci peut être très pratique si, par exemple, vous souhaitez lancer la commande quotidiennement et générer régulièrement un nouveau fichier.

Une autre option intéressante est `-g` permettant de spécifier le gain qui par défaut est automatique. `rtl_power` n'a pas de pouvoir magique, il ne fait rien qu'un logiciel SDR ne



Un morceau de graphique en couleurs inversées. Ces traces semblant étrangement se décaler au fil du temps sont des satellites. Le décalage en fréquence est dû à l'effet Doppler, exactement comme lorsque vous entendez passer une ambulance. Quand la sirène se rapproche, sa fréquence est plus importante que lorsqu'elle s'éloigne...

fasse et en particulier utiliser un taux d'échantillonnage et donc une largeur de bande plus importante que celle du matériel (généralement 2,5 Mhz). Comment alors peut-il produire une sortie unique pour une plage de 87 Mhz à 107 Mhz ? C'est simple, il le fait en plusieurs fois. Le problème qui se pose alors est double :

- ceci prend du temps et plus la plage est grande, plus vous devrez intégrer de données pour composer une ligne (par défaut l'intégration se fait sur 10 secondes) ;
- ensuite, à chaque fois que `rtl_power` change le réglage du tuner, le gain est réglé automatiquement par rapport aux émissions captées. Le résultat final sur une grande plage de fréquences peut alors paraître incohérent, irrégulier et non homogène.

Pour le premier point, il n'y a malheureusement rien à faire, si ce n'est être très prudent en jouant avec l'option `-i` spécifiant la durée pour l'intégration des données. Pour le second en revanche, il suffit de spécifier une valeur de gain avec `-g`.

Nous pouvons donc revoir notre commande ainsi `rtl_power -f 87M:107M:12.5k -g 25 -i 2s -e 5m FM2.csv`. Même plage, gain de 25, intégration sur 2 secondes, durée de 5 minutes et enregistrement dans `FM2.csv`.

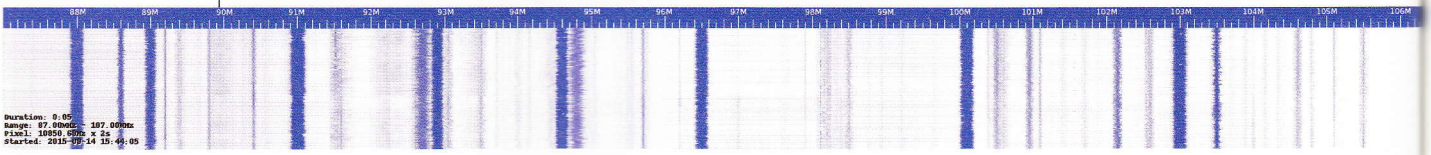
Une fois ces quelques paramètres et leur utilité correctement assimilés, il ne vous reste plus qu'à vous amuser. Un bon test est, comme ici, une petite surveillance de la bande radio FM commerciale. On passera ensuite, par exemple, à l'ensemble de la bande de réception du périphérique DVB-T. Les clés USB reposant sur un tuner RT820T ou un RT820T2 peuvent se caler sur des fréquences de 24 Mhz à 1760 MHz, mais peut-être est-ce une meilleure idée que de faire cela en plusieurs fois. En effet, un paramètre comme `-f 24M;1760M;10k` revient à faire des « passes » de 173600 ((1760M-24M)/0,01M) « pixels », ce qui prend du temps et générera un graphique absolument démesuré. En travaillant avec de larges bandes ainsi, il vaut mieux, soit segmenter, soit spécifier un *bin* plus important.

Parmi les fréquences intéressantes, nous avons toutes les bandes ISM (Industrielle, Scientifique et Médicale) dont 26,957-27,283 MHz et 433,05-434,79 MHz (2,4-2,5 GHz est malheureusement inaccessible par la plupart des récepteurs DVB-T). Les fréquences utilisées par des satellites météo (~137 Mhz pour NOAA) sont également intéressantes, car on peut littéralement les voir passer et constater les décalages de fréquence en raison de l'effet Doppler. Dans les 466 Mhz, nous avons également les *pagers* POCSAG dont nous avons déjà parlé dans le magazine (*Hackable n°2*), ainsi que de l'ADS-B des avions sur 1090 Mhz.

3. TRANSFORMER LE CSV EN IMAGE

Vous avez fait vos premiers essais et laissez votre Raspberry Pi travailler doucement, au grenier, durant plusieurs nuits. Vous avez donc une petite collection de fichiers CSV qu'il vous faut maintenant transformer en images. La solution la plus utilisée consiste à faire usage d'un script Python écrit par Kyle Keen et disponible

Graphique obtenu avec les données de la commande rtl_power -f 87M:107M:12.5k -g 25 -i 2s -w hamming -c 20% -e 5m FM3.csv (les couleurs de l'image ont ici été inversées et le contraste augmenté pour les besoins de l'impression).



sur **https://github.com/keenerd/rtl-sdr-misc/tree/master/heatmap**. Le fichier `heatmap.py` pourra être utilisé sur n'importe quelle machine GNU/Linux, Mac ou Windows pour peu que vous disposiez d'un interpréteur Python et des modules dont le script a besoin (dont PIL, la *Python Imaging Library*).

Dans sa syntaxe la plus simple, le script s'utilise avec `heatmap.py fichier.csv image.png`. En fonction de la quantité de données présentes dans le CSV, le processus de création pourra prendre plus ou moins de temps et provoquer l'utilisation de plus ou moins de mémoire. À titre d'exemple, un fichier `toto_20150425.csv` contenant les données d'une plage 130-150 Mhz, sur 13 heures, avec un *bin* de 5 Khz génère une image de 3696×4705 pixels avec un pic d'utilisation de mémoire à 75 Mo, le tout en environ 1 minute 30 sur un PC équipé de deux Intel Xeon E5520 à 2.27GHz (le script n'est pas multithread, avoir 2 CPU de 4 cœurs et 2 threads par cœur ne vous apportera rien).

C'est une très mauvaise idée que d'utiliser `heatmap.py` sur la Pi, mais en principe vous pouvez le faire, si vous aimez attendre (il vous faudra également installer le paquet `python-imaging`). Ce même fichier `toto_20150425.csv` a été traité sur une Raspberry Pi 2 B en... 17 minutes et 20 secondes !

L'idée générale est, bien entendu, de limiter l'utilisation de la Pi à l'activité de réception et d'utiliser un PC puissant pour la partie gourmande en ressources.

4. PETIT BONUS : TESTER VOTRE RÉCEPTEUR

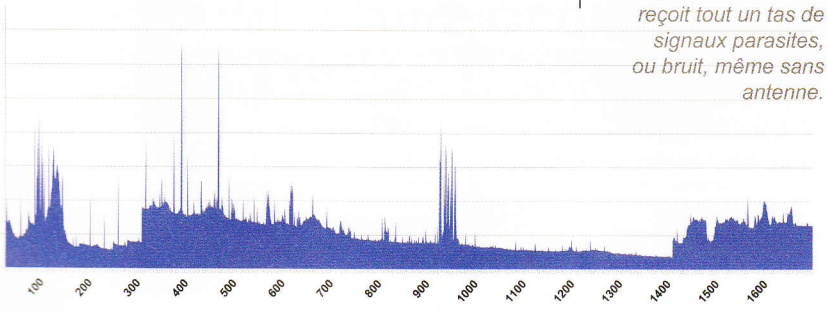
Un bouchon de terminaison à mettre en lieu et place de l'antenne. Il s'agit tout simplement d'un connecteur équipé d'une résistance correspondant à l'impédance en entrée du récepteur. Ici, un bricolage maison avec un vieux connecteur Ethernet 10base2 et deux résistances de 150 ohms en parallèle.

Comme le précise l'un des auteurs de `rtl_power` (qui est aussi celui de `heatmap.py`), Kyle Keen, l'une des activités principales des utilisateurs de récepteurs DVB-T pour la SDR, en dehors de l'écoute, consiste à chercher des astuces pour réduire le bruit dans la réception (utilisation d'un ordinateur portable sur batteries, boîte en métal, tore de ferrite sur le câble USB, changement d'oscillateur, remplacement du circuit d'alimentation, etc.).

`rtl_power` peut être utile dans le sens où il permet de tester le matériel. Il vous suffit pour cela de remplacer l'antenne par une résistance de 75 ohms et d'utiliser quelque chose comme :

```
$ rtl_power -f 24M:1.7G:1M -g 50 -i 15m -1 bruit.csv
```

On procède à une surveillance de toute la bande de réception de l'adaptateur, en poussant le gain au maximum et en spécifiant une intégration sur 15 minutes. Comme il n'y a pas d'antenne, mais une résistance de terminaison, idéalement, vous n'êtes pas censé capter quoi que ce soit. Notez la présence de l'option `-1` permettant de ne faire qu'une « passe » d'intégration (*one shot*). Le fichier CSV ne contiendra alors qu'une seule ligne que vous pourrez transformer en graphique, non pas avec `heatmap.py`, mais en ouvrant le fichier dans un tableur et en utilisant les fonctions graphiques de ce dernier.



Graphique réalisé avec LibreOffice.org et beaucoup de souffrance à partir des mesures faites par rtl_power et un récepteur équipé d'un « terminator » (ou « bouchon de terminaison » pour ceux qui se souviennent des réseaux Ethernet 10base2 coaxiaux) de 75 ohms. On voit clairement que ce récepteur non modifié reçoit tout un tas de signaux parasites, ou bruit, même sans antenne.

Règle 1147b du bidouilleur : tout devient subitement plus propre une fois couvert avec de la gaine thermorétractable. Cette règle s'applique également au bricolage en général avec la peinture.

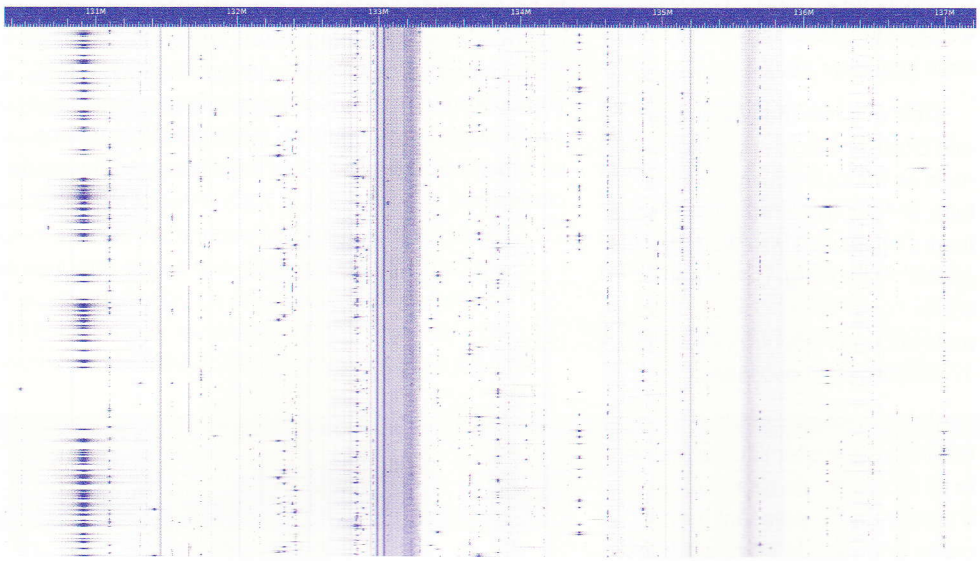
5. QUELQUES CONSEILS POUR FINIR

Les récepteurs USB DVT-T utilisables avec RTL-SDR sont généralement de qualité moyenne et surtout équipés d'oscillateurs à quartz standard (par opposition aux TCXO ou oscillateurs compensés en température). Sauf à acheter un récepteur spécialement prévu pour la SDR (comme ceux vendus sur **rtl-sdr.com**), il est déconseillé de brancher son périphérique et de lancer immédiatement une « capture » de quelques minutes. En fonction, le récepteur va chauffer, la fréquence va dériver et vous vous retrouverez avec des jolies courbes dans votre waterfall. Laissez donc le matériel monter en température et optez pour des mesures longues.

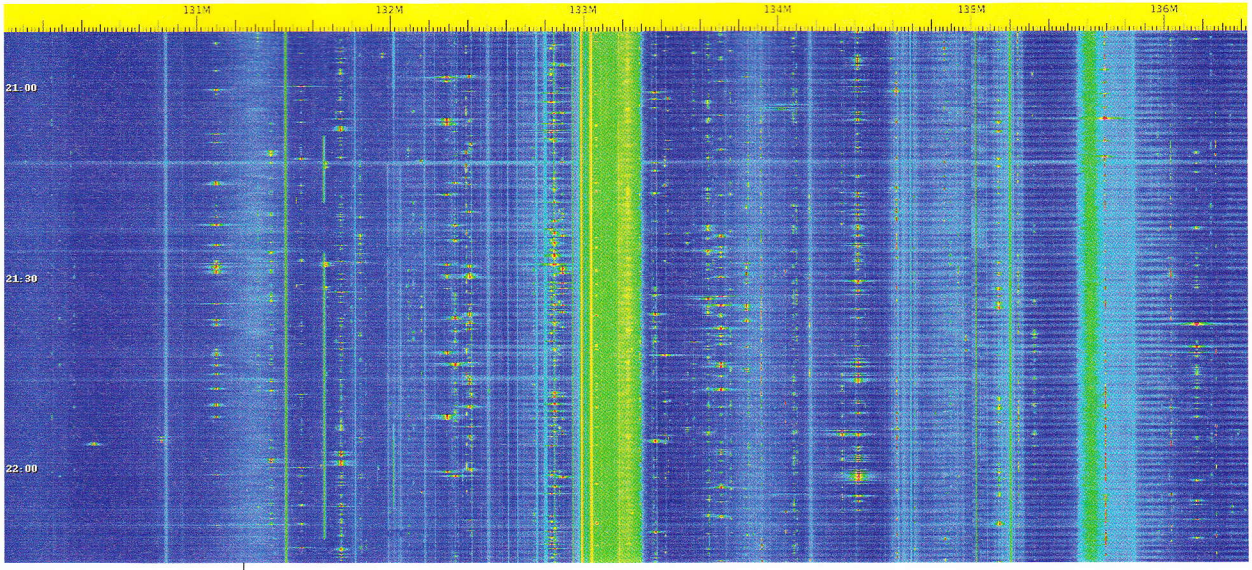
Toujours à propos de dérives de fréquence, pensez à calibrer votre récepteur DVB-T et spécifiez une valeur PPM de correction avec l'option `-p` (cf. *Hackable n°7*, page 50). La calibration doit, bien entendu, se faire dans les mêmes conditions que l'utilisation. Encore une fois, laissez le matériel se stabiliser en température.

`rtl_power` dispose d'options expérimentales. Utilisez-les ! Essayez par exemple les options de fenêtrage (`-w`) permettant d'utiliser d'autres algorithmes que celui du fenêtrage classique rectangulaire (`-w rectangle`) : `hamming`, `blackman`, `blackman-harris`, `hann-poisson`, `bartlett`, `youssef`.

Une autre de ces options est le *croping* (`-c`) permettant de spécifier un pourcentage de fréquences à écarter en bordure de la bande de réception. Votre récepteur, sur une largeur de 2 Mhz, ne reçoit pas avec la même qualité au centre de la bande et sur les fréquences les plus distantes de ce centre. Le *cropping* permet de ne garder qu'un morceau central de cette bande pour procéder aux mesures. Une valeur entre 20% et 50% est généralement recommandée, mais attention, plus vous écartez de données, plus il y aura de « tranches » à faire et donc plus la mesure de la plage de fréquences que vous avez indiquée (avec `-f`) prendra de temps. Ceci peut poser des problèmes avec



Une partie du rendu des données en exemples collectées sur 13 heures (nuit) et en couleurs inversées. Apparemment il y a beaucoup de choses qui se passent dans les airs autour de moi entre 130 Mhz et 137 Mhz pendant que je dors...



Une légère modification du script heatmap.py nous permet de créer des graphiques plus colorés, tantôt plus lisibles que ceux produits par défaut (notez en plus l'horodatage obtenu en utilisant l'option --ytick 30m).

la valeur passée en argument à l'option `-i` précisant l'intervalle en secondes pour l'intégration. Si cette valeur est plus petite que le temps nécessaire au parcours de la plage de fréquence, l'outil va boguer (c'est un problème connu et mentionné dans l'aide en ligne, accessible via l'option `-h`).

Adaptez votre antenne à la plage de fréquences que vous comptez surveiller. La conception d'antennes est un monde à part entière, mais une solution aisée et économique (et donc, brouillonne) consiste à acheter une antenne télescopique à 8 ou 10 segments pouvant s'étirer sur quelques 90 cm. Vous pouvez alors régler la taille de l'antenne sur 1/4 de la longueur d'onde ou 1/2 pour améliorer la réception. Pour rappel, la longueur d'une onde est égale à la vitesse de la lumière en mètres par seconde divisée par la fréquence en hertz (exemple pour 433 Mhz, 299792458/433000000=0,6923 soit 69,23 cm).

Et enfin, n'hésitez pas à regarder les sources des programmes. Ceci est valable pour `rtl_power.c` bien sûr, mais aussi et surtout pour `heatmap.py` qui est sans doute plus aisé à appréhender. C'est en y jetant un œil que j'ai découvert que son auteur avait déjà prévu de quoi adapter les couleurs du rendu final. Ainsi, on trouve une définition de fonction `rgb2(z)` (ligne 293) définissant la couleur d'un pixel mais, une paire de lignes plus bas, `rgb3(z)` (ligne 297) utilisant une palette totalement différente (via `colorsys.hsv_to_rgb()`). Il vous suffit de remplacer les appels à la fonction `rgb2()` par `rgb3()`, lignes 356 et 364, pour obtenir un graphique totalement différent et bien plus coloré. Moralité, ne jamais oublier de regarder les sources !

Toujours sur ce point, un autre développeur, Bruno Adele, est également allé fouiller dans le script, au point d'y faire un brin de ménage, de restructurer une partie du code et d'ajouter une fonctionnalité intéressante (intégration de notes dans l'image provenant d'un fichier JSON). Il a proposé ses modifications (*pull request*) au créateur de `heatmap.py` mais elles n'ont pas encore été intégrées. Vous pourrez trouver cette version sur **https://github.com/badele/rtl-sdr-misc/tree/master/heatmap**.

Et pour finir, j'ai moi-même créé ma version dérivée du script en ajoutant une option (`--gradient`) permettant d'utiliser un fichier image contenant un dégradé personnalisé pour produire le diagramme : **https://github.com/Lefinnois/SDR-heatmap**. Ceci peut permettre, par exemple, de rendre les signaux puissants plus visibles ou tout simplement d'obtenir un résultat plus joli. DB

NE MANQUEZ PAS

GNU/LINUX MAGAZINE HORS-SÉRIE N°81 !

LE GUIDE POUR
APPRENDRE À PROGRAMMER EN JAVA EN 5 JOURS !

Toutes les bases de Java pour pleinement utiliser le langage de Processing !

DISPONIBLE DÈS LE 13 NOVEMBRE

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

www.ed-diamond.com

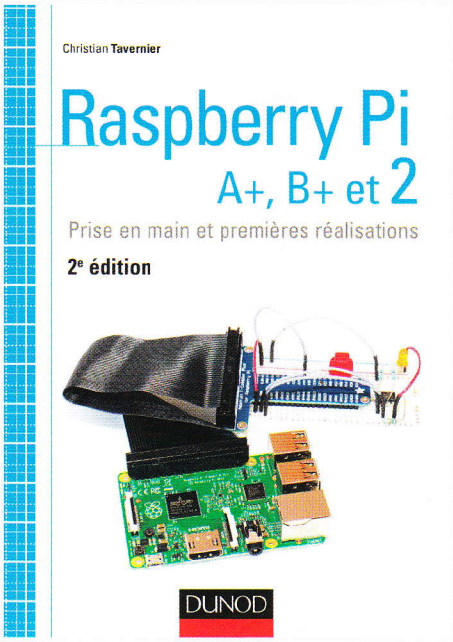
TOUS MAKERS !

RÉVÉLEZ VOTRE POTENTIEL PAR LA CRÉATION

9782100742561, 160 p., 19,90 €

PASCAL LIÉGEOIS

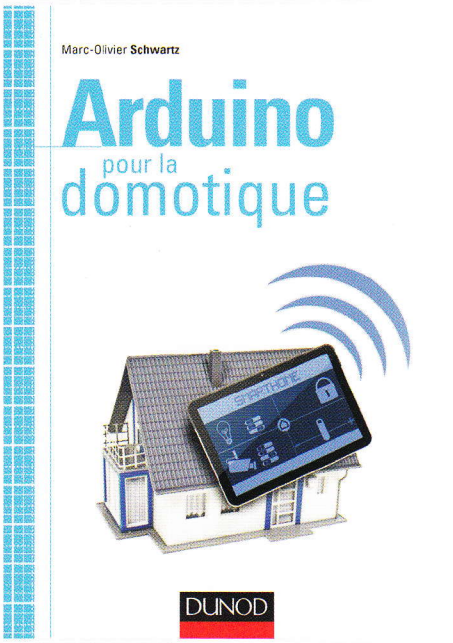
Pour apprendre à réaliser un robot radiocommandé (drone terrestre) contrôlable à vue ou à distance et pouvant embarquer un appareil photo ou une caméra.



9782100742639, 352 p., 24,90 €

CHRISTIAN TAVERNIER

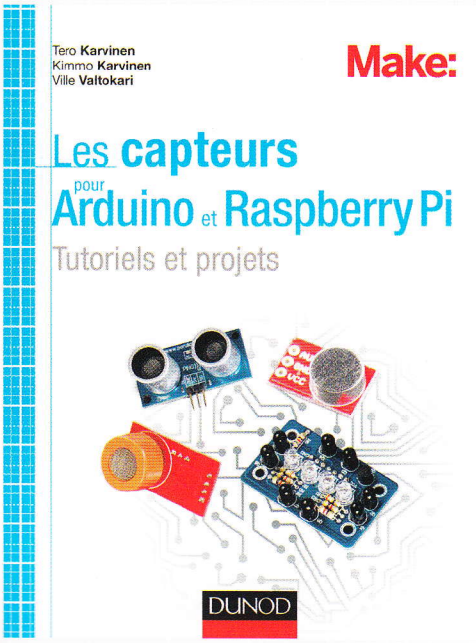
Cet ouvrage décrit les différentes versions de Raspberry Pi et explique comment le configurer et le paramétrer correctement.



9782100727117, 256 p., 27,50 €

MARC-OLIVIER SCHWARTZ

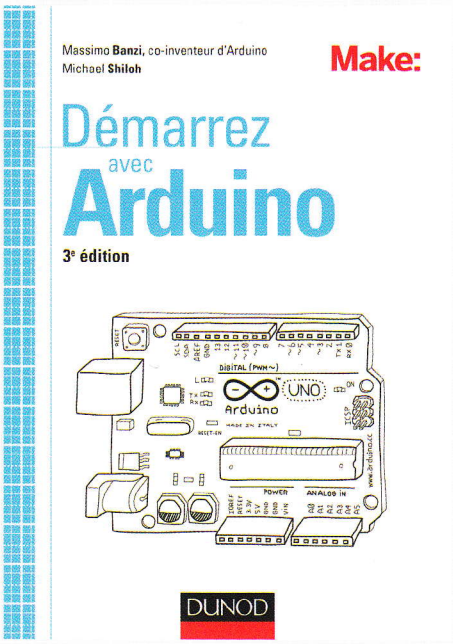
Le guide pas-à-pas de projets concrets avec des exemples de code, des schémas et des photos pour réaliser vous-même des applications domotiques.



9782100717934, 304 p., 29,90 €

TERO KARVINEN ET AL.

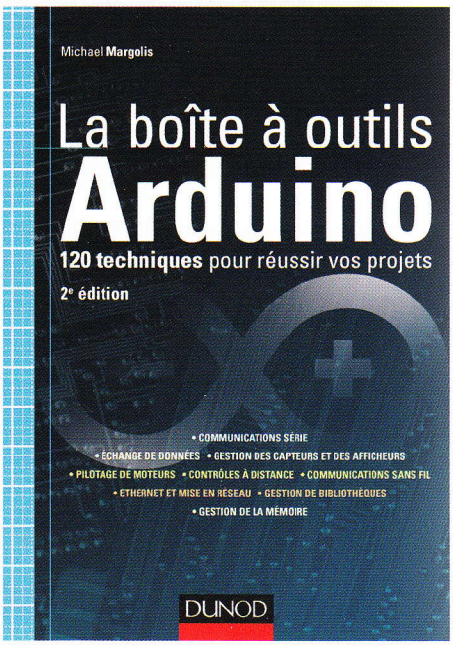
Bien utiliser les capteurs pour concevoir des montages avec Arduino ou Raspberry Pi qui interagissent avec leur environnement.



9782100727391, 192 p., 19,90 €

MASSIMO BANZI, MICHAEL SHILOH

Une présentation accessible d'Arduino et les bases en électronique et programmation pour sa mise en œuvre immédiate.



9782100727124, 480 p., 37 €

MICHAEL MARGOLIS

120 solutions pour réaliser des applications concrètes avec une carte programmable Arduino.

TOUT LE CATALOGUE SUR **WWW.DUNOD.COM**

DUNOD
ÉDITEUR DE SAVOIRS