N°5 | MARS-AVRIL 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

L 19338 - 5 - F: 7,90 € - RD

ÉDITO

Bienvenue dans ce cinquième numéro !

Bricoler, bidouiller, explorer, hacker rapidement un bout de code sous le coup d'une idée subite... On fait cela tous les jours et on obtient des résultats. Bref, ça marche et on est heureux.

Pourtant, dans certains cas, le sujet mérite un peu plus qu'une soirée ou une nuit d'intense activité. Il faut alors revoir sa copie avec d'autres priorités que le simple fait d'arriver à dominer le matériel. Il faut faire propre, beau et pratique afin que d'autres puissent profiter de vos efforts. Voilà exactement l'idée qui se cache derrière le trio d'articles en vedette sur la couverture de ce numéro : examiner le parcours d'un projet de la prise en main d'un module/composant à la réalisation d'une bibliothèque Arduino en passant par l'incontournable phase de réflexion permettant d'améliorer la qualité de son code.

Je vous laisse à présent découvrir tout cela et le reste des pages par vous-même car, en ce qui me concerne, la rédaction de ce numéro a étrangement déclenché l'apparition d'une quantité non négligeable d'idées et de pistes à explorer pour les prochains numéros. Je me lance donc sans attendre dans la suite et de nouveaux articles après avoir fait maturer tout cela, et vous donne directement rendez-vous pour le numéro 6 d'ici deux mois. Appelons cela de l'expérimentation et de la rédaction compulsive, mais j'assume totalement :)

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

B.P. 20142 – 67603 Sélestat Cedex
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
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution,
N° ISSN : en cours
Commission paritaire : en cours
Périodicité : bimestriel
Prix de vente : 7,90 €

Suivez-nous sur Twitter
@hackablemag

SOMMAIRE

L'ANALYSEUR LOGIQUE : ESPIONNER LES BUS EN TOUTE SIMPLICITÉ

Denis Bodor

Qu'on utilise Arduino ou Raspberry Pi, on en arrive toujours à brancher des modules et périphériques utilisant des protocoles comme SPI, i2c, CAN, 1-Wire, etc. En cas de problème, savoir si oui ou non les données que vous voulez envoyer transitent réellement sur les connecteurs est capital. Mieux encore, si le montage n'est pas de votre fait, il s'agit là d'un fantastique moyen de comprendre comment il fonctionne et comment le réutiliser.

Précisons que nous parlons ici d'espionner des bus, pas des bus. Autrement dit des bus de données série ou parallèles et non des bus municipaux (autobus). Ne riez pas, ce n'est pas totalement une boutade (17% boutade), car c'est possible. Oona Räisänen sur son fantastique blog (**windytan.com**) l'a parfaitement montré, avec un récepteur RTL-SDR et beaucoup d'expérience dans le domaine, elle a été en mesure de décoder le protocole DARC du système IBus utilisé à Helsinki et donc à recevoir les messages des bus avec leur temps d'arrivée à un arrêt... depuis chez elle (avec les hivers finlandais je pense qu'on préfère attendre le bus chez soi) ! Il est donc effectivement possible d'espionner les bus, et non pas seulement les bus. Je vous recommande fortement la lecture des billets de son blog (en anglais) dans lesquels vous trouverez peut-être des choses qui sont également utilisées chez nous (dans une Cafétéria U par exemple).

Un bus de données qu'il soit série ou parallèle est une méthode souvent standardisée pour échanger des données entre un système et un périphérique. Capteur de température SPI, RTC en i2c ou encore simplement brancher et piloter un écran LCD 2×16 caractères HD44780 se fait en utilisant un bus. Espionner cette connexion permet de s'assurer de la validité des données qui circulent et donc de confirmer le fonctionnement de ses programmes. Mais il devient également possible de se lancer dans une épopée plus importante en étudiant des données mystérieuses et en arrivant, à terme, à déduire le protocole utilisé et ses commandes et informations utiles. Ensuite, il sera possible de réutiliser le périphérique ou module et le contrôler à partir d'un autre système comme une carte Arduino ou une Raspberry Pi.

1. AVANT-PROPOS

L'outil dont nous allons parler ici est l'analyseur logique et il est judicieux, avant toutes choses, de préciser quelques points importants. Il ne faut, tout d'abord, pas confondre analyseur logique et oscilloscope numérique car, sur bien des points, au niveau fonctionnel, ils présentent des similitudes. Les deux sont en effet des équipements destinés à mesurer et collecter des données en fonction du temps. Ils disposent donc de sondes ou *probes* qui seront reliées à un système ou un montage « vivant » afin d'en comprendre ou vérifier le comportement.

Les deux appareils procèdent donc à des mesures, traitent les données et se chargent de leurs représentations à destination de l'utilisateur. Cette vision très grossière, mais néanmoins assez juste fait qu'un grand nombre d'oscilloscopes numériques permettent d'embarquer des fonctionnalités d'analyse logique et, inversement, les analyseurs logiques peuvent tantôt proposer des fonctions relevant du travail d'un oscilloscope, comme la représentation d'un signal analogique récurrent.

Mais il existe cependant une différence fondamentale si l'on s'en tient au travail premier de chaque équipement : l'oscilloscope mesure des phénomènes analogiques et l'analyseur numérique se cantonne aux signaux logiques et donc à des valeurs discrètes de tension (certaines valeurs dans un ensemble prédéfini). Même si l'un peut potentiellement faire le travail de l'autre, on peut résumer cela en affirmant simplement que :

- un oscilloscope numérique pourra faire office d'analyseur logique d'appoint,
- et un analyseur logique pourra éventuellement, selon le modèle, faire office d'oscilloscope modeste.

Il faut également ajouter à cela que, comme bien des équipements, les deux appareils se déclinent en version « de laboratoire » en fonctionnement autonome et en version « périphérique PC/Mac » utilisable conjointement à un ordinateur équipé du bon logiciel. Là encore, mais c'est dépendant de l'usage effectif et du domaine, une différence existe : pour l'oscilloscope on préférera généralement la

version « équipement de laboratoire » au périphérique PC/Mac, mais pour l'analyseur logique, la version périphérique sera privilégiée (surtout pour des questions budgétaires).

Enfin, il est important de préciser le cadre d'utilisation qui nous concerne. En tant qu'amateur éclairé ou qu'ingénieur spécialisé dans les circuits numériques (microcontrôleurs, systèmes embarqués, etc.), l'intérêt premier de l'analyseur logique est d'obtenir des informations sur des données numériques à un niveau supérieur d'abstraction. En effet, les analyseurs logiques peuvent également être d'une grande aide pour les problèmes de *timing* et de traitement des signaux numériques au plus bas niveau. Comme souvent, si votre domaine de prédilection est, par exemple, celui de la transmission de données vidéos numériques (type HDMI/DVI) ou n'importe quelle liaison très haut débit où une différence de quelques nanosecondes est capitale, je pense que vous n'avez rien à apprendre de cet article et connaissez déjà parfaitement l'équipement qu'il vous faut (et accessoirement avez la capacité financière professionnelle ou personnelle pour l'acquérir). Avec un budget fixé à moins d'une centaine d'euros, disons-le clairement, il ne faut pas compter pouvoir espionner un bus comme l'USB, le PCI Express ou l'HDMI. C'est simplement hors de notre portée.

Ce qui nous occupera ici sera donc exclusivement en rapport avec les technologies généralement utilisées sur des plateformes comme la Raspberry Pi ou les cartes Arduino et consorts. En d'autres termes, les échanges de données entre ces systèmes et leurs périphériques, à une vitesse et avec un débit propre aux plateformes concernées et aux périphériques les plus généralement utilisés (capteurs, afficheurs, servomoteurs, etc.).

2. L'ANALYSEUR LOGIQUE

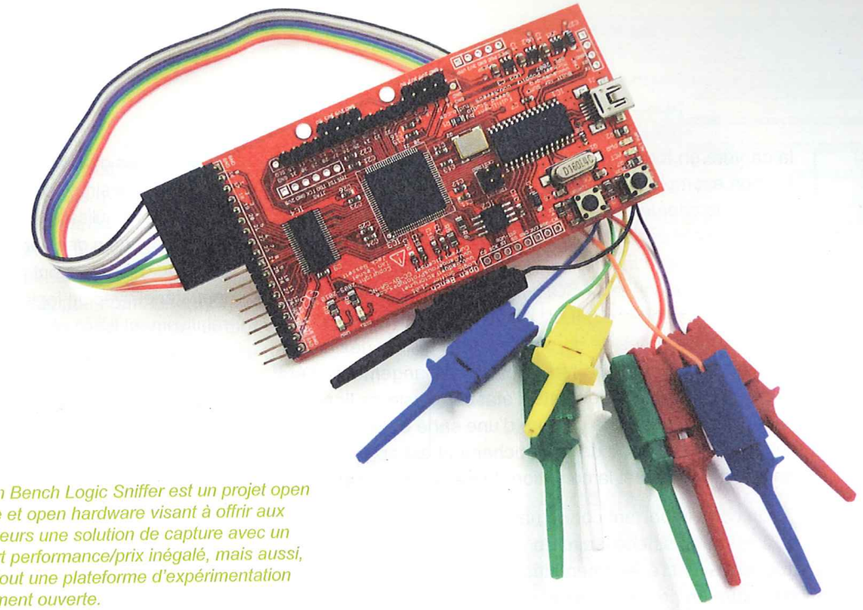
Vous l'avez compris, le type de matériel qui nous occupe à présent est un périphérique à connecter à un ordinateur. Son travail consiste à mesurer l'état de ses entrées (haut ou bas) au fil du temps. Après cette collecte, l'ensemble pourra être traité et représenté sous la forme d'un diagramme sur lequel l'utilisateur pourra littéralement voir l'état de chaque entrée sur une période donnée.

Si on s'arrêtait là, il y aurait déjà un intérêt notable puisqu'il serait possible de « lire » cette succession d'états hauts et d'états bas pour en déduire les données qui viennent de circuler. Mais il y a mieux. Puisque les informations sont là, et représentées par un logiciel, pourquoi ne pas directement faire en sorte que cette analyse soit faite par l'ordinateur ? L'utilisateur n'a alors qu'à choisir le protocole et les paramètres qu'il souhaite utiliser pour interpréter les signaux/données et la traduction lui apparaîtra sous forme d'une succession de valeurs décimales, binaires ou hexadécimales, ou encore directement en équivalence ASCII (cf série d'articles sur le LTC5926 dans le présent numéro).

Les analyseurs logiques se présentant sous forme de périphériques peuvent être classés en plusieurs catégories, ceux avec mémoire et ceux sans. Pour capturer des signaux logiques de manière efficace, la vitesse est un élément crucial. En effet, la vitesse à laquelle le périphérique est en mesure de lire l'état de ses entrées doit être plus importante que la vitesse à laquelle ces entrées changent d'état. Dans le cas contraire, on rate des transitions et donc forcément des données. Cette vitesse est fixée en nombre de fois par seconde où les entrées sont lues, il s'agit donc d'une fréquence et plus exactement d'une fréquence d'échantillonnage.

Pour arriver à ces performances, il faut donc que la partie matérielle de l'analyseur logique repose sur des circuits et des composants très rapides. Mais ce n'est pas tout, lire des données est une chose, encore faut-il pouvoir les conserver aussi rapidement qu'on les lit. C'est là qu'il faut distinguer deux écoles :

- Celles des analyseurs logiques intégrant une mémoire : lorsque la lecture est déclenchée, l'intelligence embarquée dans le



L'Open Bench Logic Sniffer est un projet open source et open hardware visant à offrir aux utilisateurs une solution de capture avec un rapport performance/prix inégalé, mais aussi, et surtout une plateforme d'expérimentation totalement ouverte.

périphérique se charge de procéder à la lecture de l'état des entrées à la fréquence choisie et stocke ces informations dans la mémoire interne. Une fois la lecture stoppée ou la mémoire pleine, les données sont transférées au PC/Mac via USB (le plus souvent), présentées à l'utilisateur et analysées. La quantité de mémoire intégrée détermine donc la masse d'information qu'il est possible de collecter et donc la durée totale de lecture. Parfois, cela peut être insuffisant, voire largement insuffisant, car plus la fréquence de capture est importante, plus la mémoire est rapidement consommée.

- Celles des analyseurs logiques sans mémoire : plutôt que de stocker les données lues dans une mémoire rapide embarquée, elles sont transmises directement au PC/Mac. Le principal problème de cette approche est le goulot d'étranglement que représente la connexion avec l'ordinateur. La vitesse de collecte des informations ne sera jamais supérieure à celle de la liaison périphérique/ordinateur. En revanche, la limitation concernant la quantité de données capturées disparaît et la durée de la capture n'est dépendante que de la capacité de stockage de l'ordinateur : énorme.

Dans les deux cas, une « astuce » permet de contourner le problème dans une proportion toute relative. Il faut en effet différencier deux façons de stocker les données. Soit les captures sont faites et enregistrées de manière régulière à la fréquence choisie par l'utilisateur et donc de façon successive, soit ce sont uniquement les changements d'état et le moment de leur apparition qui donnent lieu à un enregistrement. Cette technique pour gagner en espace mémoire est considérée comme une méthode de compression et elle peut intervenir à plusieurs stades du traitement des données. On parle généralement de *Run Length Encoding* (RLE).

Cette différenciation de méthodes de stockage et de techniques utilisées nous amène à aborder un autre sujet : le déclenchement des captures. Quelles que soient les fonctionnalités de l'analyseur logique, le moment choisi pour débuter la capture est critique. Il est toujours possible, à la manière d'un enregistreur audio de simplement laisser l'utilisateur cliquer un bouton « *record* », mais il y a bien plus efficace. L'idée est de tout simplement laisser l'analyseur logique déclencher lui-même

la capture en fonction de conditions qui lui sont imposées. Un bon exemple, pour la capture sur un bus SPI consiste à débuter la capture lorsque la ligne /CS (destinée à asservir le composant esclave) passe de l'état haut à l'état bas. Le simple fait que le maître désigne l'esclave déclenche alors la capture sur les lignes SCK, MISO et MOSI, capturant ainsi la communication. Il est également possible, avec des analyseurs logiques de bonne facture, d'utiliser des conditions plus complexes comme plusieurs changements d'état successifs, des changements d'état sur plusieurs lignes ou encore, directement, la capture d'une série de bits en particulier. Cette méthode de déclenchement est appelée *triggering* et un *trigger* est la condition du départ de la capture.

Un point important concernant les *triggers* est le fait qu'il est parfois nécessaire de capturer les données qui se trouvent avant l'événement déclencheur. En effet, lorsqu'on rencontre un problème qui nécessite l'usage d'un analyseur logique, il est indispensable de connaître, dans une certaine mesure, les conditions du problème. On *trigger* donc généralement sur un symptôme du problème tout en collectant des données avant et après le moment où il survient. Ceci est possible de la part de l'analyseur logique, par la mise en place d'une mémoire tampon. Les données capturées avant et après le déclenchement sont appelées respectivement *pre* et *post triggering*.

Pour conclure cette partie introductive, résumons donc les éléments clés importants qui caractérisent un bon analyseur logique :

- la fréquence d'échantillonnage ou autrement dit le nombre de fois par seconde où les entrées peuvent être lues,
- la mémoire intégrée qui détermine le nombre d'échantillons (*samples*) qui peuvent être collectés, ou l'absence de mémoire,
- le nombre d'entrées disponibles impactant directement le type et le nombre de bus utilisables,
- les différentes options permettant de débuter automatiquement la capture (*triggering*),
- la possibilité d'utiliser une compression.

Ce à quoi s'ajoutent naturellement le prix, le type de périphérique et la disponibilité d'applications compatibles de qualité. Ce dernier point est très important, en particulier dans notre cas, car notre objectif est clairement d'espionner des bus et non de faire de l'analyse temporelle (*timing analysis*). La partie applicative est celle qui embarque le traitement et l'analyse des données, et donc la traduction de simples signaux en informations utilisables. L'approche utilisée par bien des constructeurs consiste généralement à vendre le matériel avec un logiciel relativement limité et à proposer des traducteurs pour d'autres protocoles sous forme d'options complémentaires payantes. Bien entendu, la communauté open source propose une autre approche, bien plus ouverte.

3. OPENBENCH LOGIC SNIFFER

Le premier périphérique dont nous allons parler est l'*Openbench Logic Sniffer* (ou OLS) qui est un projet communautaire open source. Développé conjointement par *Gadget Factory* et *Dangerous Prototypes* ce matériel est né d'une discussion dans des commentaires d'un billet sur le blog de *Dangerous Prototypes* (les créateurs du fantastique *Bus Pirate* dont nous parlerons prochainement).

Le projet se compose de trois éléments entièrement open source : le matériel, le protocole et le logiciel. Côté matériel, l'OLS est un analyseur logique à mémoire connectable en USB et répondant aux caractéristiques suivantes :

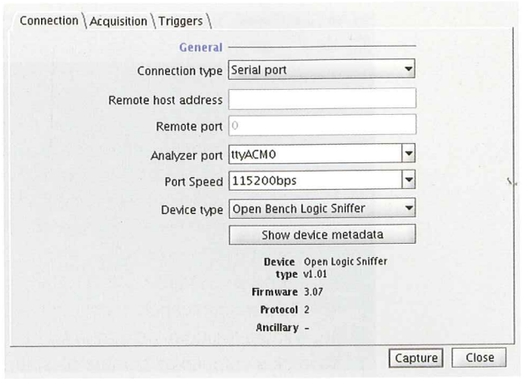
- fréquence d'échantillonnage jusqu'à 70 Mhz,
- 32 entrées (canaux), dont 16 avec *buffer* et tolérant 5V (les autres sont en 3,3V),
- possibilité d'utiliser la carte comme plateforme de développement FPGA.

L'OLS est construit autour de deux principaux composants. Nous avons d'une part un FPGA (*Field-Programmable Gate Array*) qui est, grossièrement, un composant programmable pouvant matérialiser n'importe quel circuit logique. Un FPGA n'est pas comme le microcontrôleur de votre Arduino, il ne fait rien de lui-même et n'est, sans être programmé, qu'une matrice de transistors. En programmant un FPGA, on le configure de manière à ce qu'il prenne la forme d'un circuit complet. Concrètement, un microcontrôleur exécute le programme d'un utilisateur, mais un FPGA **est** le circuit que l'utilisateur souhaite qu'il devienne. Le principal intérêt des FPGA est leur capacité à traiter des informations en parallèle et ils sont donc d'une grande efficacité pour traiter des entrées multiples comme sur un analyseur logique. Le FPGA utilisé ici est un Xilinx Spartan 3E.

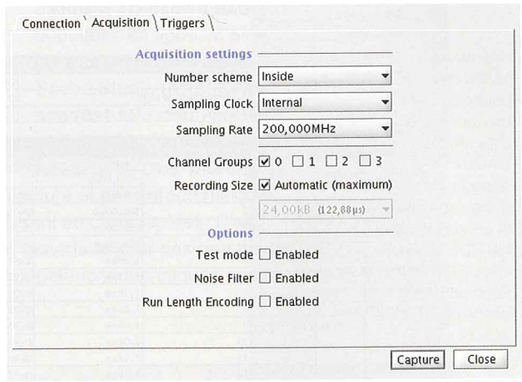
L'autre composant de l'OLS est un microcontrôleur Microchip PIC18F24 chargé de faire office d'interface avec le PC/Mac et de piloter le FPGA. Les deux « firmwares » sont majoritairement open source (à l'exception de la partie USB propre à Microchip, gratuite, mais non distribuable).

L'OLS et en particulier l'aspect logiciel peut être déroutant pour l'utilisateur, car plusieurs projets s'imbriquent pour former le produit final. SUMP, par exemple est à la fois le logiciel Java utilisable sur l'ordinateur pour piloter l'OLS, mais également le code pour le FPGA. En effet, le projet SUMP (sump.org) vise à permettre au propriétaire d'une plateforme de développement FPGA d'utiliser son matériel comme analyseur logique. SUMP fournit également une bibliothèque Java permettant de gérer la communication avec le matériel. Ce a quoi s'ajoutent plusieurs « clients » ou applications dérivées de SUMP qui peuvent également être utilisés avec l'OLS. L'un d'entre eux, *Logic Sniffer* était un temps une alternative au client SUMP, mais est depuis devenu l'applicatif par défaut. Le choix des noms pour la plateforme et le logiciel n'aide, bien entendu pas, à clarifier les sources de confusions.

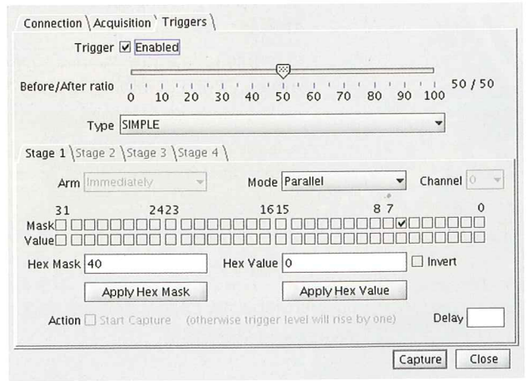
Ainsi, aujourd'hui nous avons l'OLS d'une part et *Logic Sniffer* de l'autre. C'est donc sur **http://www.lxtreme.nl/ols/** qu'il faudra pointer votre navigateur pour récupérer la dernière version en date : 0.9.7 SP2



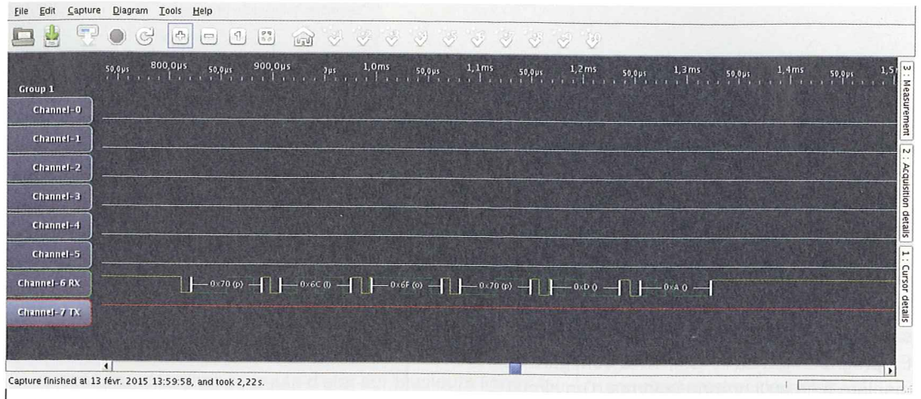
OLS 1 : Pour capture des données, on commence par choisir le matériel utilisé, le port de connexion, la vitesse de communication, etc.



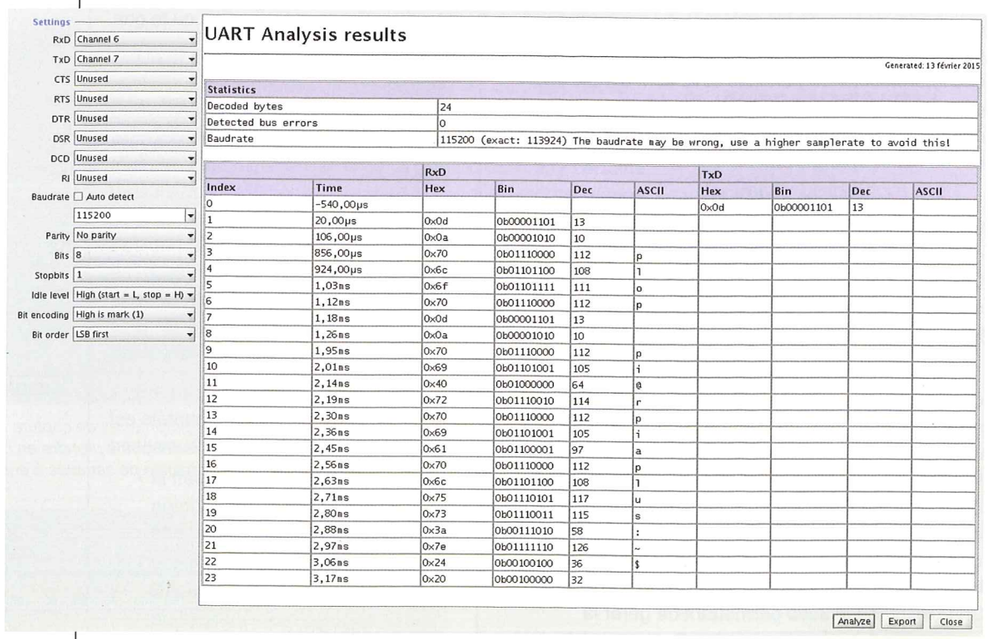
OLS 2 : On règle ensuite les paramètres de capture comme la fréquence d'échantillonnage, les entrées à prendre en compte, la source d'horloge, la quantité de samples à enregistrer...



OLS 3 : On active éventuellement l'utilisation de trigger(s) afin de déclencher automatiquement la capture et on attend que l'événement survienne...



OLS 4 : Une fois les données capturées, elles apparaissent dans l'interface utilisateur sous forme de diagrammes.



Settings

RxD: Channel 6
TxD: Channel 7
CTS: Unused
RTS: Unused
DTR: Unused
DSR: Unused
DCD: Unused
RI: Unused
Baudrate: Auto detect — 115200
Parity: No parity
Bits: 8
Stopbits: 1
Idle level: High (start = L, stop = H)
Bit encoding: High is mark (1)
Bit order: LSB first

UART Analysis results

Generated: 13 février 2015

Statistics	
Decoded bytes	24
Detected bus errors	0
Baudrate	115200 (exact: 113924) The baudrate may be wrong, use a higher samplerate to avoid this!

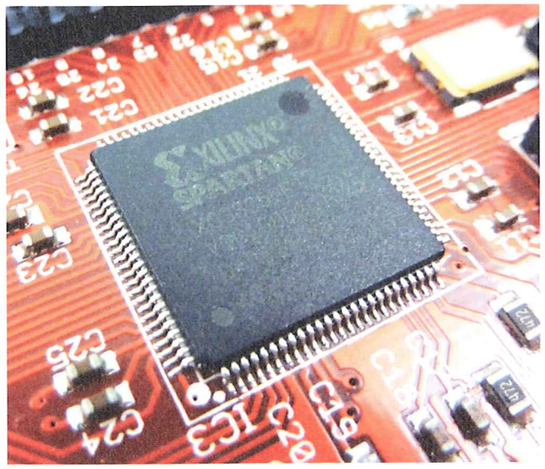
		RxD				TxD			
Index	Time	Hex	Bin	Dec	ASCII	Hex	Bin	Dec	ASCII
0	-540,00µs					0x0d	0b00001101	13	
1	20,00µs	0x0d	0b00001101	13					
2	106,00µs	0x0a	0b00001010	10					
3	856,00µs	0x70	0b01110000	112	p				
4	924,00µs	0x6c	0b01101100	108	l				
5	1,03ms	0x6f	0b01101111	111	o				
6	1,12ms	0x70	0b01110000	112	p				
7	1,18ms	0x0d	0b00001101	13					
8	1,26ms	0x0a	0b00001010	10					
9	1,95ms	0x70	0b01110000	112	p				
10	2,01ms	0x69	0b01101001	105	i				
11	2,14ms	0x40	0b01000000	64	@				
12	2,19ms	0x72	0b01110010	114	r				
13	2,30ms	0x70	0b01110000	112	p				
14	2,36ms	0x69	0b01101001	105	i				
15	2,45ms	0x61	0b01100001	97	a				
16	2,56ms	0x70	0b01110000	112	p				
17	2,63ms	0x6c	0b01101100	108	l				
18	2,71ms	0x75	0b01110101	117	u				
19	2,80ms	0x73	0b01110011	115	s				
20	2,88ms	0x3a	0b00111010	58	:				
21	2,97ms	0x7e	0b01111110	126	~				
22	3,06ms	0x24	0b00100100	36	$				
23	3,17ms	0x20	0b00100000	32					

Analyze | Export | Close

OLS 5 : On peut ensuite lancer l'analyseur de protocoles en spécifiant les correspondances signal/broche et voir apparaître les données capturées. Ici, nous avons connecté les broches 6 et 7 sur RX et TX d'un adaptateur USB/série relié à une Rapsberry Pi. On voit clairement dans la colonne TxD la validation avec un caractère 0x0D et la réponse de la Pi avec un retour à la ligne suivi de « plop », un autre retour à la ligne puis « pi@rpiablus:~$ » qui est tout simplement l'invite de la ligne de commandes.

du 2 janvier 2015. L'application est disponible pour GNU/Linux, Windows et Mac OS X.

Dès le matériel connecté, celui-ci est accessible via un port série virtuel fourni par le microcontrôleur PIC. Il suffit alors de lancer *Logic Sniffer* et s'assurer que « OpenBench LogicSniffer » est bien sélectionné dans le menu ***Capture > Device***. Il suffit alors d'un clic sur le bouton de capture (icône avec la flèche vers le bas) pour faire apparaître la boite de dialogue de configuration.



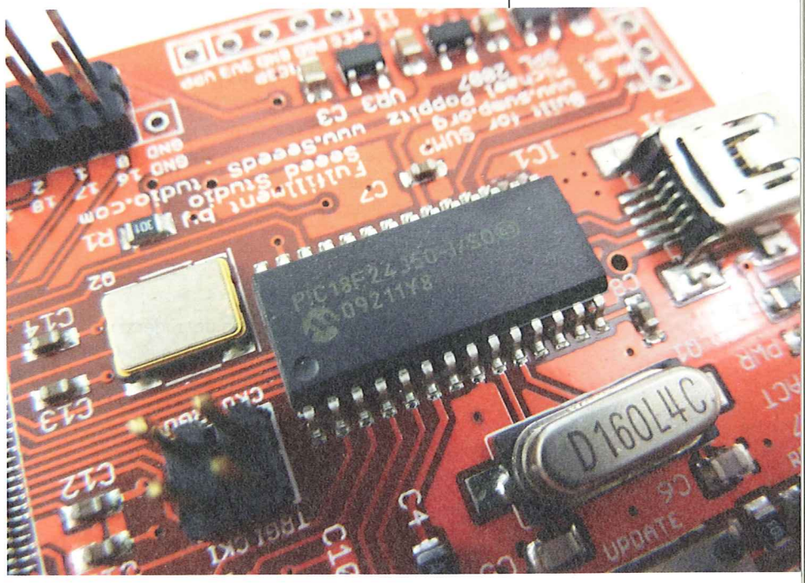
This is Spartaaaaaaaan ! (désolé, celle-ci était tout simplement obligatoire). Ce Xilinx Spartan 3E équipant l'OLS se charge de la capture des données qui se doit d'être très rapide, mais c'est aussi le composant qui fait de ce matériel une plateforme permettant d'expérimenter le développement sur FPGA.

permet, par défaut, de gérer 1-wire, Asm45, DMX512, i2c, SPI, JTAG, UART. L'ajout de protocoles est possible via l'utilisation d'un système de greffons. Ici, contrairement aux applications propriétaires, il ne faut pas les acheter, mais les écrire et les partager.

Open Bench Logic Sniffer de Dangerous Prototypes est disponible dans plusieurs boutiques en ligne, la plus connue étant sans doute Seeed Studio (**http://www.seeedstudio.com**) qui le propose à environ 44 euros. Notez que l'OLS vise clairement, de par son architecture et ses caractéristiques, les utilisateurs s'intéressant aux bus rapides. C'est une excellente solution matérielle pour des personnes ayant déjà certaines compétences techniques et des besoins correspondants.

Le microcontrôleur Microchip PIC18F24 fournit l'interface USB permettant le contrôle et la gestion des captures et des triggers. Son firmware à l'exception de la partie USB est disponible sous forme de code source et librement modifiable.

Là, on pourra régler les paramètres concernant la connexion à l'OLS, mais aussi et surtout ceux dictant comment la capture des données doit se dérouler et l'utilisation éventuelle de *triggers*. Bien entendu, la quantité maximum de données capturées est dépendante de la fréquence d'échantillonnage et de la mémoire disponible. Pour une communication lente comme un port série à 115200 bps, 200Khz sera suffisant, mais la quantité d'informations obtenues sera de toute façon relativement réduite (24k *samples*). L'applicatif bénéficie énormément de l'aspect communautaire, en particulier du point de vue de l'analyse de protocole. La version 0.9.7 SP2 de *Logic Sniffer* que nous avons testé



Le Saleae Logic, le vrai ! Un analyseur logique créé avec un souci de qualité et de robustesse. L'un des premiers matériels de ce type susceptible de pouvoir être acheté par un amateur ou un hobbyiste pour des besoins semi-professionnels.

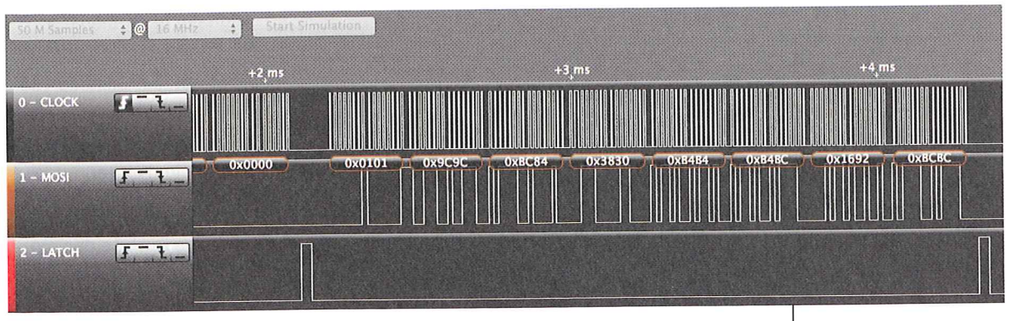
Il est difficile de voir l'OLS comme un produit « fini » ou un appareil d'usage courant bien qu'il propose des fonctionnalités exceptionnelles pour un tel prix. Vendu nu, il faut plutôt le voir comme une plateforme de développement et un concept performant principalement destiné aux hackers et autres bidouilleurs.

4. SALAE LOGIC

Lorsqu'il est question de parler d'un produit à destination de l'utilisateur final en termes d'analyse logique abordable, deux noms ressortent immédiatement du lot USBee et Saleae Logic. Les deux produits ont une approche différente de celle de l'OLS en reposant sur l'utilisation d'un microcontrôleur spécialisé. Notez que les explications et éléments qui vont suivre sont principalement basés sur l'exemplaire en ma possession, acquis il y a quelque temps déjà, d'un modèle qui n'est plus disponible : le Saleae Logic (tout court). Une nouvelle gamme de 4 produits remplace désormais les deux uniques modèles de l'époque (Logic et logic16). Parmi les modèles actuels Logic4, Logic8, Logic Pro 8 et Logic Pro 16 (respectivement $109, $219, $479 et $599 directement sur la boutique Saleae), le Logic4 est celui qui se rapproche le plus du modèle que je possède. Les trois autres modèles utilisent également un FPGA (Xilinx Spartan 6) en complément du microcontrôleur Cypress.

Contrairement à l'OLS, la famille Logic forme une gamme de « vrais » produits de mesure et se compose d'analyseurs ne stockant pas les données, mais les *streamant* au PC/Mac auquel il est connecté. Comme détaillé précédemment, ceci permet de collecter une masse bien plus importante d'informations. Le site Saleae fournit un simulateur en faisant la démonstration. Ainsi, avec un PC disposant de quelques 8Go de RAM, il est possible de capturer les informations d'un bus SPI (SCK à 60Khz) avec une fréquence d'échantillonnage de 12MS/s (méga-samples par seconde) pendant presque 5 heures !

Le nouveau Logic4 dispose de 4 entrées numériques (logiques) et 4 analogiques permettant de marcher sur les plates-bandes des oscilloscopes.



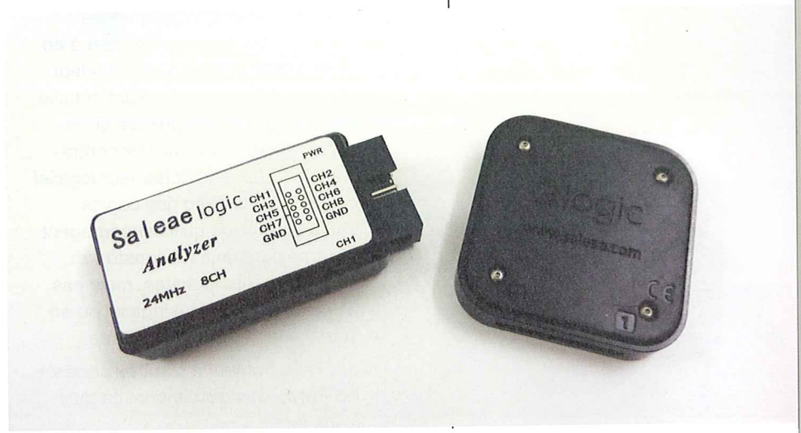
Cette capture montre les données échangées par une carte Arduino et un afficheur à leds. Le Saleae Logic nous a été utile dans la rédaction des articles formant le sujet principal en couverture du magazine. Comme le module d'affichage réagissait de manière étrange, nous avons donc décidé d'espionner les données et il s'est avéré qu'un signal (LAT) arrivait effectivement au mauvais moment (trop tôt). Sans analyseur logique, trouver la source du problème aurait été autrement plus aléatoire.

Le Logic quant à lui ne disposait pas de ces dernières et utilisait donc ses 8 entrées de manière purement numérique. Cependant, l'ensemble fonctionne exactement de la même manière.

Accompagnant l'appareil, une application Windows, Mac OS X et GNU/Linux propriétaire téléchargeable gratuitement permet de collecter les données et décoder les protocoles (CAN, DMX-512, i2c, i2c/PCM, 1-Wire, série asynchrone, parallèle, SPI, UNI/O). Très ergonomique et intuitif, ce logiciel permet de rapidement faire ses premières captures et d'analyser ses données.

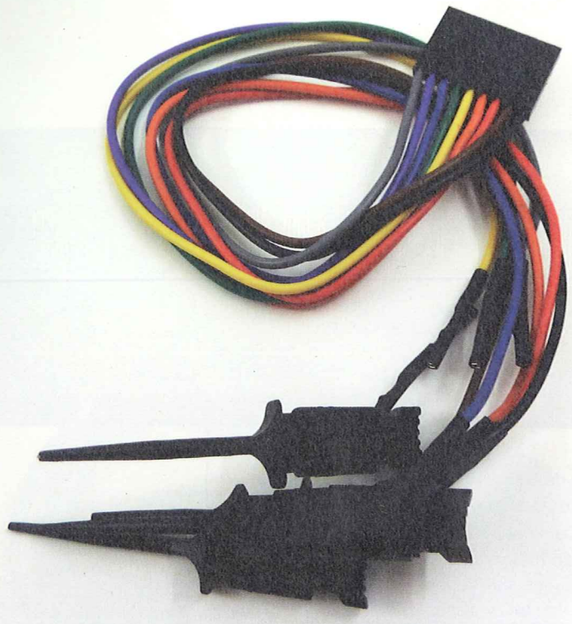
Certes, avec un prix de départ approchant les 100€, les Logic de Saleae ne sont pas accessibles à toutes les bourses et demandent un vrai besoin. Ce n'est pas un jouet, mais un véritable appareil de mesure qui demande à être utilisé régulièrement.

Je n'ai pas testé les nouveaux modèles de Saleae, car mon Logic fonctionne parfaitement et répond à mes besoins, mais je n'ai aucun doute sur leur qualité. Ce qui nous amène au problème des copies. Nous aborderons ce point dans la section suivante, mais c'est ici que je souhaite mettre en lumière la réaction de Saleae. Très rapidement des clones du Logic ont en effet été commercialisés. Vraisemblablement, certaines personnes ont été capables d'extraire les informations sur le fonctionnement du périphérique et en ont violé



Le Logic de Saleae et l'un des clones. À droite, l'original dans son boîtier aluminium fraisé et anodisé et à gauche, le clone dans une boite plastique avec un simple sticker. Ouf, heureusement je dispose de dons d'investigation journalistique exacerbés, pour un peu je me serai fait avoir tant la copie est parfaite !

ANALYSEUR LOGIQUE



Lorsqu'il s'agit de capter des signaux, conserver au maximum leur intégrité est capital. Et ceci passe par l'utilisation de sondes de qualité permettant un contact franc avec le bus espionné et avec un niveau de qualité qui en font des éléments durables. C'est une partie mécanique qui s'use, elle doit être adaptée à la tâche.

tout simplement les droits d'auteur. Le tout en assemblant des circuits qui étaient joyeusement libellés Saleae.

Les fondateurs de Saleae, les frères Joe et Mark Garrison ont publié une page sur leur site (**https://www.saleae.com/counterfeit**) précisant que ces contrefaçons causaient du tort à ce qu'ils tentaient de réaliser et à leur entreprise familiale. Le billet détaille leur point de vue et précise qu'ils comptent travailler sur des contre-mesures pour empêcher leur logiciel de fonctionner avec des copies, tout en précisant qu'ils encouragent d'autres personnes à construire des analyseurs logiques, mais pas en créant leur propre logiciel ou en aidant à un projet open source.

Contrairement à d'autres sociétés, les frères Garrison ont fait ce qu'il fallait faire :

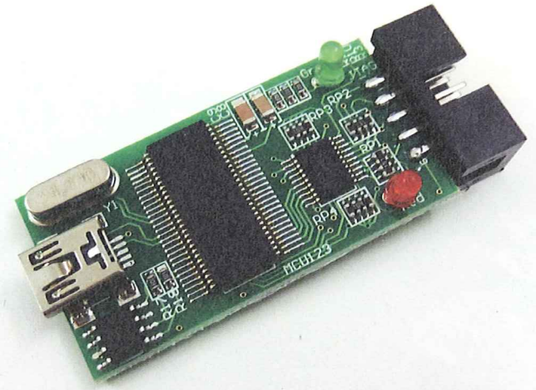
- 1 : signaler le problème à leurs clients,
- 2 : préciser leur point de vue et leurs motivations,
- 3 : faire part des actions qu'ils comptaient entreprendre,
- 4 : innover !

Je pense que l'arrivée des modèles Logic8, Logic Pro 8 et Logic Pro 16, intégrant un FPGA, et ajoutant des fonctions analogiques pour l'ensemble des modèles n'est pas étrangère à ce problème de copie. Améliorer son produit, ajouter des fonctionnalités, être à l'écoute des clients et communiquer, voilà exactement les actions à mener pour couper l'herbe sous le pied des copieurs malhonnêtes.

5. LA GUERRE DES CLONES

Derrière cette désignation se cachent des histoires bien plus intéressantes et mouvementées que celles du film éponyme (ce qui n'est pas difficile). Les clones des USBee et Saleae Logic sont généralement des reproductions des circuits utilisés par les périphériques originaux et réutilisant le firmware et le logiciel original. Ceci semble avoir dérangé bien plus de monde que les sociétés concernées puisque les développeurs du projet open source Sigrok ont, à leur manière mis un point d'arrêt à la copie illégale (du moins en partie, certains pourraient y voir un acte contre-productif).

Sigrok est un projet de développement d'une suite logicielle d'outils de traitement



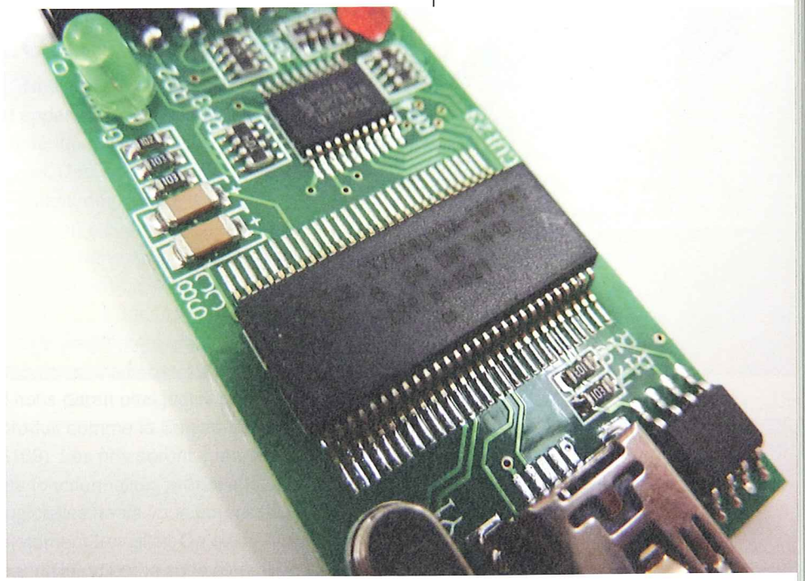
Ce clone utilise apparemment le même microcontrôleur Cypress que le Salae Logic, mais avec les outils Sigrok, on mettra en œuvre le firwmare fx2lafw en lieu et place du code de Saleae. Pour passer dans la légalité, il suffit donc de ne plus utiliser l'application Saleae, mais Pulseview de Sigrok.

de signaux. L'objectif poursuivi n'est pas de créer un logiciel pour une plateforme donnée, mais d'arriver à un ensemble susceptible de fonctionner avec une vaste gamme de matériels et non pas seulement des analyseurs logiques. Le projet propose ainsi un support pour quelques 50 analyseurs logiques, 18 périphériques dits *mixed-signal* (analogique+numérique), 27 oscilloscopes, 68 multimètres, 7 sonomètres, 15 thermomètres, 4 hygromètres, etc.

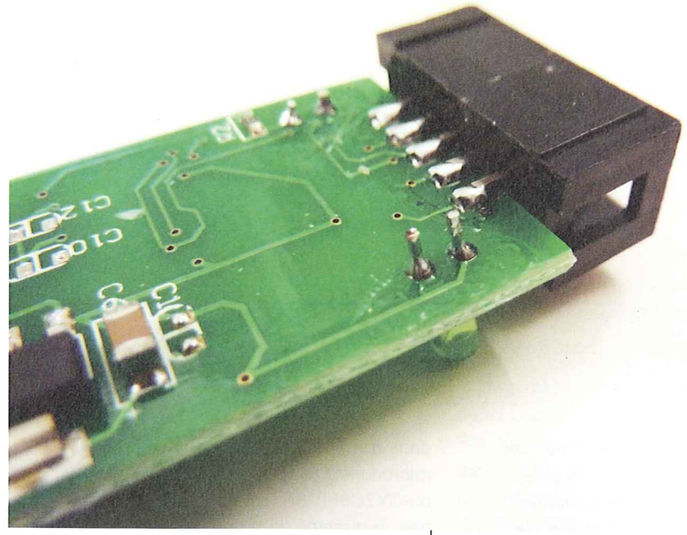
Ce support prend la forme de bibliothèques, d'outils en ligne de commandes et d'interfaces utilisateur (*fronted*) graphiques. Le tout scriptable en Python 3 pour compléter la gamme de protocoles décodés, et offrant des fonctions d'import/export de données avec une vaste variété de formats.

Dans cette myriade de composants, il en est un qui règle le problème de la contrefaçon, du moins partiellement (coller une étiquette « Saleae » sur un produit qui ne vient pas de Saleae reste un acte de contrefaçon et de tromperie) : le firmware fx2lafw. Il s'agit d'un code sous licence GPL pouvant être compilé et chargé dans n'importe quel produit à base de microcontrôleur Cypress CY7C68013 (FX2) ou CY7C68013A (FX2LP). N'en doutez pas, la quantité de produits de ce type est impressionnante (environ 40) et compte 10 produits originaux (dont le Logic et l'USBee), et deux cartes d'évaluation, le reste étant des clones. Vous trouverez la liste complète sur le Wiki de Sigrok (**http://sigrok.org/wiki/Fx2lafw**). On remarquera au passage que les cloneurs ne falsifient pas uniquement le produit final, mais également les puces, en ajoutant un « A » à la sérigraphie d'un microcontrôleur FX2 (CY7C68013) pour le

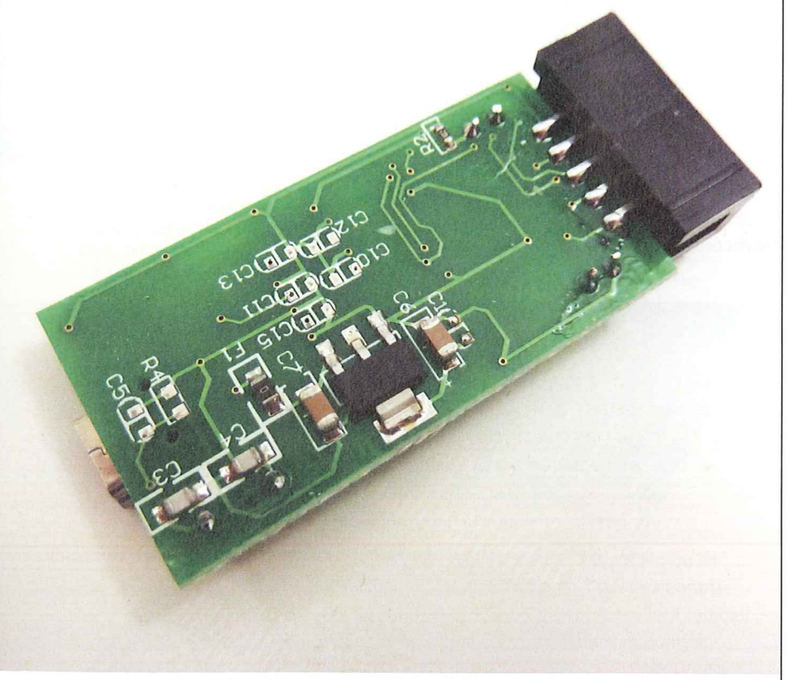
L'intérieur d'un clone est révélateur du niveau général de qualité du produit. À l'intérieur d'un boîtier plastique bas de gamme se trouve un simple circuit auto-routé avec un microcontrôleur, un buffer et une mémoire. La motivation du fabricant n'est ici pas la qualité ou l'innovation, mais uniquement les bénéfices.



ANALYSEUR LOGIQUE



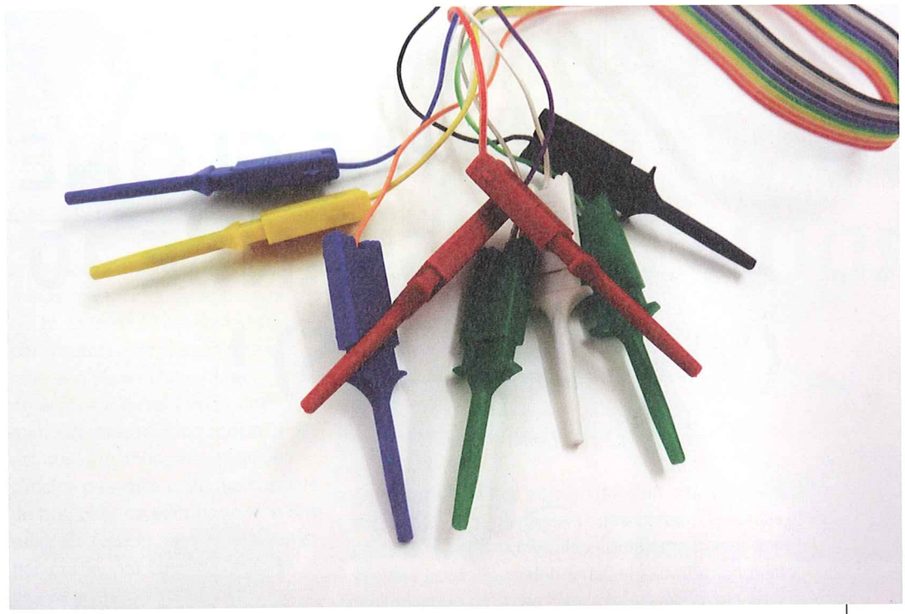
Le principal problème avec les clones est leur faible qualité. Le soin apporté aux soudures et au placement de composants laisse vraiment à désirer. Sur le dessous du circuit du clone de MCU123, on voit clairement qu'il s'agit certainement d'un circuit routé automatiquement.



faire passer pour un FX2LP (CY7C68013A). Heureusement, le firmware fonctionne avec les deux modèles.

Il est donc possible, en ayant acheté l'un de ces clones frauduleux de devenir un utilisateur responsable en flashant le matériel avec un firmware open source et en utilisant Sigrock, en particulier son interface graphique Pulseview en guise d'application (en lieu et place de l'application Saleae). Attention cependant, lorsqu'on dit « flashé » ce n'est pas dans le même sens qu'avec une carte Arduino. Le microcontrôleur Cypress utilisé, à l'inverse de l'AVR d'un Arduino, dispose de 16 Ko de RAM et pas de flash. Le firmware n'est donc pas vraiment flashé, mais plus exactement uploadé dans le périphérique. Ainsi, il suffit d'utiliser Pulseview ou les outils Sigrok en ligne de commandes pour ne plus violer les droits d'auteur.

Mais nous nous devons ici d'être clairs. Le projet Sigrok est massif et va bien au-delà du simple pilotage d'un analyseur logique. L'interface graphique



Exemple de sondes livrées avec les produits bas de gamme : connecteurs fragiles, câbles d'une finesse ridicule, plastique peu résistant... Idéales pour faire des mesures, autant des parasites que des signaux.

Pulseview bien qu'utilisable n'est en réalité qu'un élément de l'ensemble. Pour tirer profit de la pleine puissance et de la souplesse de Sigrock, l'utilisation de la ligne de commandes (`sigrok-cli`) est indispensable. Généralement, on capture avec `sigrok-cli` et les paramètres adéquats pour ensuite ouvrir le fichier contenant les données (`.sr`) dans Pulseview.

Il faudra donc attendre un peu avant que le projet ne produise un outil aussi intuitif que celui de Saleae par exemple, car l'accent est actuellement principalement mis sur l'aspect fonctionnel du projet. Ceci est facile à constater puisqu'il suffit de jeter un œil à la liste des modules de décodage de protocoles. Le wiki de Sigrok en liste 45 allant de la simple liaison SPI ou i2c à des protocoles de bien plus haut niveau comme ceux des RTC, des DAC, des capteurs comme le DS28EA00 (thermomètre) et même le Nunchuk de la Nintendo Wii.

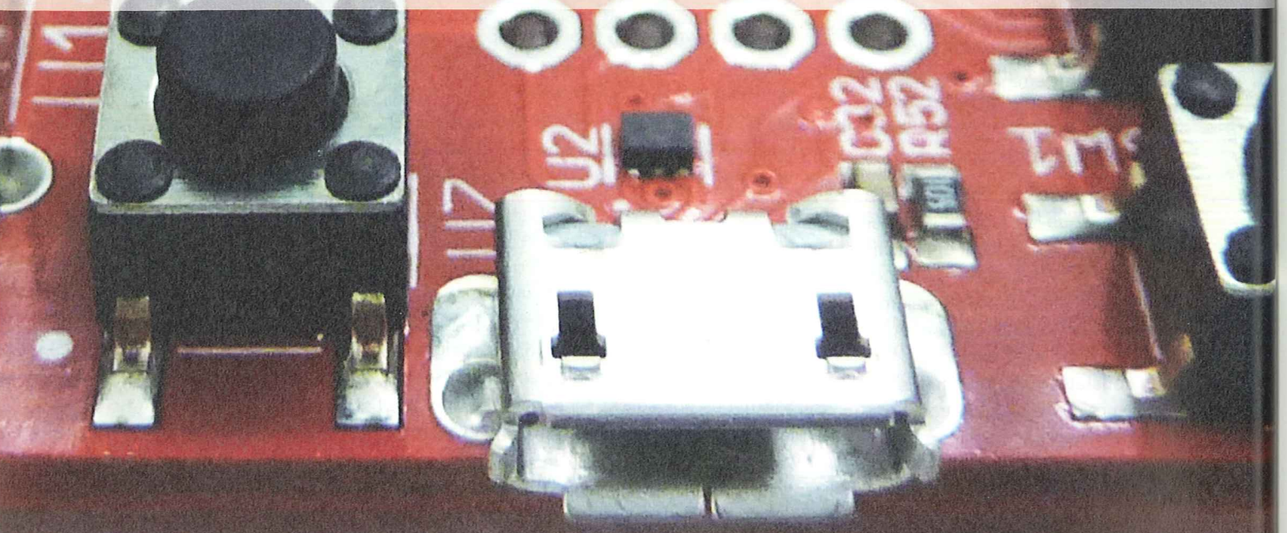
CONCLUSION

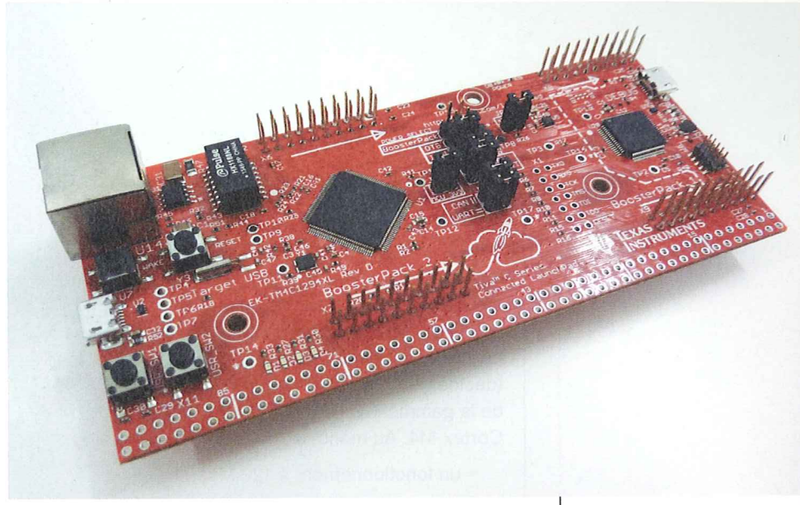
Nous avons eu un aperçu des différents produits et appareils accessibles à l'amateur d'électronique numérique et la conclusion est pour l'instant sans appel. Des solutions entièrement open source nécessitent des connaissances et une certaine expérience en électronique et en ligne de commandes. L'OLS comme les fruits du projet Sigrok sont des plateformes puissantes et souples, mais ne sont pas à mettre dans toutes les mains. Si votre objectif est de simplement espionner un bus pour affiner ou déboguer vos codes pour Arduino ou Raspberry Pi, il nous paraît plus judicieux de vous diriger vers un produit comme le Saleae Logic ou l'USBee (SX à $169). Les prix seront, bien entendu en rapport avec les fonctionnalités proposées, matérielles comme logicielles, mais vous pourrez directement et immédiatement travailler. Ce qui ne vous empêche pas, en parallèle ou par la suite d'acquérir un OLS. DB

TI LAUNCHPAD CONNECTÉ OU LE CLONE ARDUINO PUISSANCE 10 (VOIRE PLUS) !

Denis Bodor

Oui bon, c'est vrai que c'est un peu racoleur comme titre, mais les faits sont les faits. La carte EK-TM4C1294XL LaunchPad de Texas Instruments est littéralement impressionnante de par les fonctionnalités offertes, et ce pour un prix inférieur à celui d'une carte Arduino Uno. Voyons ensemble ce que vaut cette petite bestiole...





Comme toute la gamme Launchpad de Texas Instruments, cette carte est facilement reconnaissable de par son circuit imprimé rouge vif.

Vous connaissez l'histoire et la manière dont tout cela fonctionne, un groupe a une idée et la réalise associant de manière indélébile un nom à un ensemble d'émotions et d'éléments marquants. Si le succès est là, la formule est réutilisée par d'autres dans l'espoir de se faire une petite place dans ce qui devient rapidement un phénomène de masse (« de masse » est quelque chose de relatif, je n'inclus pas ici la collègue capable de brancher un téléphone IP à une prise FT (vécu)). Seulement voilà, par inertie, par crainte de sortir des sentiers battus ou par adhésion à une idée ou un concept, les outsiders n'arrivent généralement pas à se faire plus qu'une place minoritaire, pour peu que l'initiateur du « mouvement » ne fasse pas trop de bêtises.

C'est ainsi que, dans ce petit monde de 0 et de 1, de 0V et de +5V, « Arduino » est encore et toujours le mot qui revient et qui rassure. Qu'il s'agisse de logiciel ou de matériel, quand bien même ils sont issus de géants du domaine, cela ne sera perçu que comme des alternatives au grand Arduino. Tenir tête à la figure emblématique est un travail de longue haleine. Égaler le leader en terme d'image prend encore davantage de temps. Et enfin, le dépasser pour devenir calife à la place du calife est un objectif presque hors d'atteinte.

Dans le jeu du « qui cours derrière Arduino », nous trouvons entre autres le géant *Texas Instruments* et son écosystème *Launchpad*. Il s'agit aujourd'hui d'un ensemble assez impressionnant de choses : des cartes, des outils, des partenaires et des modules complémentaires appelés *booster pack* (équivalents des shields Arduino), etc. L'initiative ne date pas d'hier puisque la première carte MSP430 Launchpad a commencé à être distribuée en juillet 2010. Cette petite bébête rouge existe toujours et bien que le composant central ait évolué en gamme depuis, la structure est la même : un microcontrôleur, une vraie interface de programmation intégrée, deux leds, deux boutons-poussoir (dont un pour reset), un ensemble de broches et un environnement type IDE Arduino appelé Energia.

La version actuelle de cet « *original Launchpad* », le MSP-EXP430G2, repose sur un microcontrôleur 16 bits MSP430G2553 qui fournira :

- une horloge à 16 Mhz,
- 16 Ko de flash pour le code,
- 512 octets de RAM
- 16 entrées/sorties,
- un convertisseur analogique/numérique 16 bits 8 canaux,
- deux timers,
- des ports (i2c, SPI, UART).

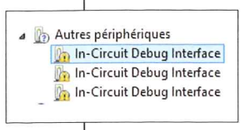
Tout cela pour moins de 10€ !

Mais la pièce qui nous intéresse aujourd'hui est plutôt dans le haut de la gamme : le EK-TM4C1294XL LaunchPad. Gâchons de suite l'effet de surprise (*spoiler*), ce haut de gamme, ne vous coûtera qu'environ 20 euros !

Dans le cas de Windows 8 et 8.1, les choses se compliquent encore davantage étant donné que les pilotes fournis par Texas Instruments (**http://www.ti.com/tool/stellaris_icdi_drivers**) ne sont pas signés et de ce fait, Windows refusera de les installer. Il faudra donc avant toutes choses connecter votre Launchpad, attendre que Windows cherche des pilotes et n'en trouve pas, puis aller sur la page web de Texas Instruments pour récupérer l'archive `spmc016.zip`. Vous désarchiverez celle-ci n'importe où et obtiendrez un répertoire `stellaris_icdi_drivers` (« Stellaris » est grossièrement l'ancien nom de la gamme Tiva, mais on le retrouve un peu partout (pilotes, documentations, logiciels, etc.)).

Pour installer ces pilotes et les « associer » aux périphériques non gérés par Windows 8/8.1, vous devrez ensuite passer par une succession d'étapes :

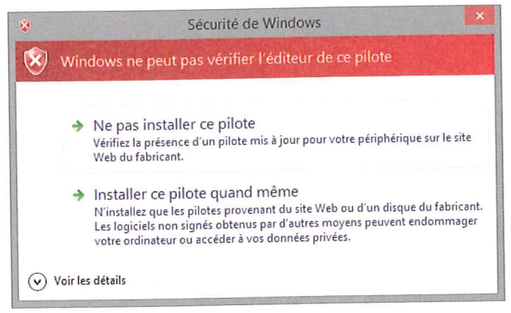
- depuis l'interface Metro (le truc moche avec les pavés aux couleurs qui piquent les yeux) choisissez, sur la droite, ***Paramètres***,
- cliquez sur ***Modifier les paramètres du PC***,
- dans la nouvelle « fenêtre » qui s'ouvre, choisir ***Mise à jour et récupération***,
- puis ***Récupération***,
- et sous ***Démarrage avancé***, cliquez sur ***Redémarrer maintenant***,
- après quelques secondes, vous arrivez sur un nouvel écran et vous choisissez ***Dépannage***,
- puis ***Options avancées***,
- ***Paramètres***,
- et cliquez sur le bouton ***Redémarrer***,
- vous arrivez sur un autre écran avec une liste numérotée et devez choisir ***7 - Désactiver le contrôle obligatoire des signatures de pilotes*** en tapant le numéro au clavier.



Les différents périphériques apparaissant à la connexion de la carte Launchpad sous Windows 8. Passage par la case « installation de pilotes manuellement » obligatoire.

Ce n'est pas fini. Là, vous venez juste de désactiver l'interdiction d'installer des pilotes non signés. Il vous faut maintenant spécifier les pilotes à associer aux périphériques :

- dans le panneau de configuration, choisissez ***Matériel et audio***,
- puis ***Gestionnaire de périphériques*** sous ***Périphériques et imprimantes*** (ne cliquez pas sur le titre, mais juste sur ***Gestionnaire de périphériques***),
- dans la liste de périphériques, repérez ceux sous ***Autres périphériques*** désignés par « In-Circuit Debug Interface » avec un point d'exclamation sur leurs icônes (voir capture),
- cliquez avec le bouton droit de la souris sur le premier et choisissez ***Mettre à jour le pilote***,
- puis choisissez ***Rechercher un pilote sur mon ordinateur***,
- et recherchez le répertoire `stellaris_icdi_drivers` où se trouve le contenu du fichier Zip téléchargé,
- après avoir confirmé l'emplacement, Windows signale qu'il ne peut pas vérifier l'éditeur de ce pilote (sans déconner ?) et vous devez confirmer l'installation en cliquant sur ***Installer ce pilote quand même***,
- recommencez cette mise à jour de pilotes pour les autres périphériques « In-Circuit Debug Interface » encore problématiques (3 en tout),
- enfin, hurlez « *Mais ils sont débiles ?! C'est plus compliqué que d'installer Linux !* » en gesticulant les bras :)



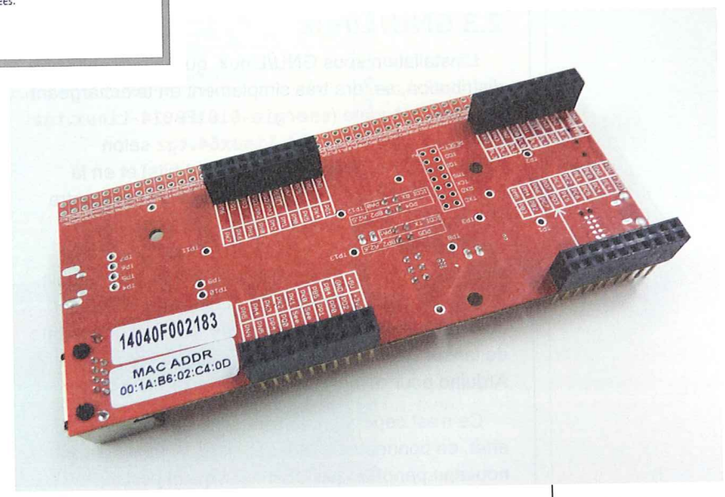
Oh mon dieu ! Vous voulez installer un pilote non signé ?! Microsoft considère cela comme vraiment très dangereux. C'est amusant, les pilotes FTDI qui rendaient inutilisables les périphériques à base de puces clonées, eux, étaient bel et bien signés...

Après tout ce travail, la carte Launchpad doit apparaître, avec un affichage par connexion, sous la forme d'un périphérique USB composite auxquels sont rattachés les périphériques suivants :

- *Stellaris ICDI DFU Devices* qui est le périphérique de mise à jour du contenu ou de programmation du Launchpad (DFU pour *Device Firmware Upgrade*),
- *Stellaris ICDI JTAG/SWD Interface* qui est l'interface pour déboguer et tracer l'exécution du code avec les environnements avancés/pro,
- et *Stellaris Virtual Serial Port,* suivi d'un numéro de port (ici COM4) qui correspond au port utilisé par le moniteur série.

Une fois toutes les manipulations correctement effectuées, on peut enfin profiter de son nouveau jouet. Ici, la carte fait apparaître correctement les différents éléments dans le gestionnaire de périphériques.

Vous pouvez maintenant aller sur le site d'Energia pour récupérer le fichier `energia-0101E0014-windows.zip` et le décompresser.



Le dessous de la carte avec les deux connecteurs femelles pour les Booster Packs. Contrairement aux shields Arduino, ceux-ci peuvent être empilés sur ou sous la « carte mère ».

Dans le répertoire que vous obtenez se trouvent l'environnement de programmation et l'icône pour le lancer.

Bien entendu, toutes ces péripéties pourraient être évitées si Texas Instruments signait son pilote, ou mieux encore, si celui-ci était intégré dans les pilotes pouvant être installés automatiquement par Windows (la seconde option impliquant la première). Mais ceci impliquerait très certainement des accords entre Texas Instruments et Microsoft, ce qui suppose des implications financières...

2.2 Mac OS X

L'installation pour Mac OS est des plus simples puisqu'il vous suffit de télécharger le fichier `energia-0101E0014-macosx.dmg` qui est une image disque Mac. Un simple double-clic permettra d'ouvrir et monter ce disque virtuel. Dans la fenêtre qui s'ouvre automatiquement se trouve l'application Energia ainsi qu'un raccourci vers le dossier Applications. Prenez l'application, glissez-la sur le raccourci et celle-ci sera copiée dans les applications installées. Enfin, éjectez le disque `energia-0101E0014-macosx` via le Finder et c'est terminé.

Il ne vous reste plus qu'à trouver Energia dans les applications et éventuellement la glisser dans le Dock pour un accès rapide.

2.3 GNU/Linux

L'installation sous GNU/Linux, quelle que soit votre distribution, se fera très simplement en téléchargeant l'archive adéquate (`energia-0101E0014-linux.tgz` ou `energia-0101E0014-linux64.tgz` selon l'architecture et le système 32 ou 64 bits) et en la désarchivant à un endroit quelconque comme votre répertoire personnel. Ce faisant, vous obtiendrez un répertoire `energia-0101E0014` contenant l'ensemble de l'environnement ainsi que l'exécutable `energia` qui est en réalité un script shell invoquant Java avec les arguments adaptés. Il vous suffira donc de lancer ce programme, tout comme avec l'IDE Arduino pour obtenir l'interface de programmation.

Ce n'est cependant pas la seule chose à faire. En effet, en connectant votre Launchpad, Linux détecte un nouveau périphérique USB ainsi qu'un port série (`/dev/ttyACM*`). Cependant, par défaut les permissions concernant l'accès à ce périphérique sont limitées et seuls les utilisateurs autorisés peuvent y lire et écrire. Comme il est absolument hors de question d'utiliser le super utilisateur `root` pour lancer l'environnement Energia (via `sudo`), il est nécessaire de changer une configuration.

Sans entrer dans le détail du fonctionnement de ce genre de mécanisme, un programme fonctionnant en permanence permet de gérer ce type de choses : `udev`. En fonction de certaines règles, `udev` ajuste les permissions, les opérations à déclencher à la connexion/déconnexion, etc. Pour faire en sorte que les utilisateurs puissent accéder au Launchpad, il faut donc définir une nouvelle règle. Ceci se fait dans le répertoire `/etc/udev/rules.d` en ajoutant un fichier dont le nom se termine par `.rules`.

Ajoutez donc un fichier `/etc/udev/rules.d/TiLaunchpad.rules` contenant :

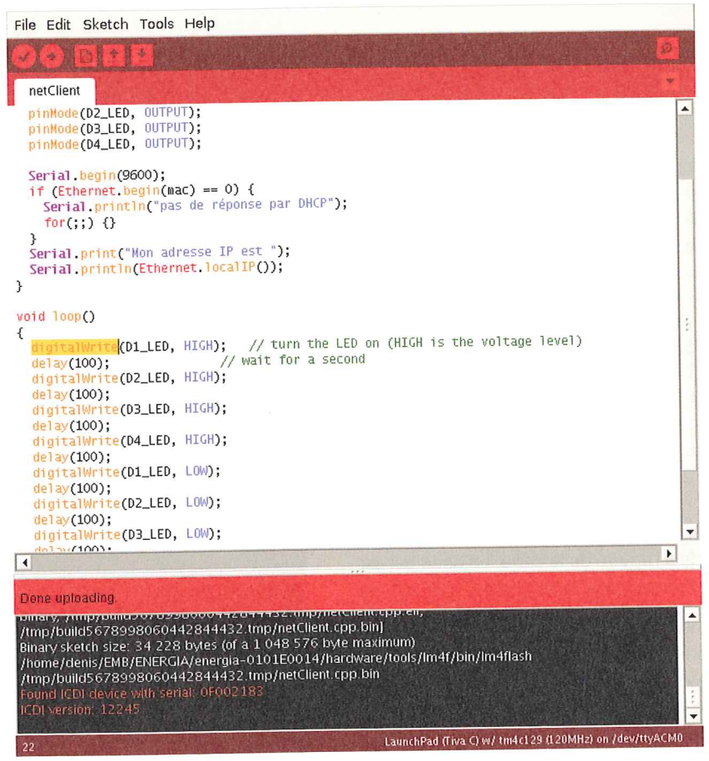
```
ATTRS{idVendor}=="1cbe",
ATTRS{idProduct}=="00fd",
GROUP="plugdev", MODE="0660"
```

Ceci aura pour effet de créer une nouvelle règle définissant que le périphérique USB identifié par `1cbe:00fd` (VendorID:ProductID) pourra être lu et écrit par tous les utilisateurs du groupe `plugdev`. Ce groupe présent dans les distributions Ubuntu est celui des utilisateurs ayant accès aux périphériques « connectables ». Pour une autre distribution comme Debian, Arch ou encore Fedora, vous devrez spécifier un nom de groupe dont vous faites parti ou éventuellement remplacer `GROUP="plugdev"` par `OWNER="votre_nom_utilisateur"` (mais c'est nettement moins appréciable puisque là, seul, vous pourrez accéder au périphérique et non tous les utilisateurs d'un groupe).

Une fois la règle ajoutée, vous devrez redémarrer le service `udev` avec `sudo service udev restart`, `sudo /etc/inid.d/udev restart` ou encore `sudo udevadm control --reload-rules`. Si le Launchpad est encore connecté, vous devrez sans doute le débrancher et le rebrancher pour que la règle soit appliquée.

3. DÉCOUVERTE D'ENERGIA ET PREMIER CROQUIS

Dès son lancement, Energia ne manquera pas de vous rappeler quelque chose. En effet, l'environnement est presque totalement identique à celui d'Arduino si l'on fait abstraction de la couleur dominante. On retrouve ainsi les mêmes zones, les mêmes menus



En dehors de la couleur, ceci ne vous rappelle rien ?

et les mêmes icônes permettant de gérer les croquis, procéder à la vérification (compilation) et au téléversement (programmation du microcontrôleur).

Dans le menu ***Tools***, puis ***Board*** vous pourrez choisir le modèle de Launchpad qui est le vôtre : « LaunchPad (Tiva C) w/tm4c129 (120 Mhz) ». Le menu ***Serial Port*** devra aussi être ajusté pour le moniteur série même si le port n'est pas utilisé pour la programmation (tout se passe via l'interface ICDI).

Concernant le code des croquis exemples, la compatibilité avec Arduino est quasi parfaite grâce au travail du projet Energia et au portage de Wiring. Mais plutôt que de simplement faire clignoter une led, utilisons un exemple plus avancé pour démontrer cela.

Que diriez-vous de tester le fonctionnement de l'interface réseau ? Rien de plus simple en vérité puisque le croquis suivant est suffisant :

```
#include <Ethernet.h>

byte mac[] = {0x00,0x00,0x00,0x00,0x00,0x00};

void setup() {
  // communication avec le moniteur
  Serial.begin(9600);

  // démarrage de l'interface
  if (Ethernet.begin(mac) == 0) {
    Serial.println("pas de réponse par DHCP");
    // pas de réponse, on boucle ici
    for(;;) {}
  }

  // envoi des infos au moniteur série
  Serial.print("Mon adresse IP est ");
  Serial.println(Ethernet.localIP());
}

void loop()
{
  // on fait rien en boucle
}
```

Si vous avez déjà eu le loisir de jouer avec un shield Ethernet pour Arduino, vous conviendrez avec moi qu'il n'y a là qu'une subtile différence. Notre croquis ne fait pas grand-chose à part demander à obtenir une adresse sur le réseau et afficher celle ainsi obtenue (via DHCP) sur le moniteur série.

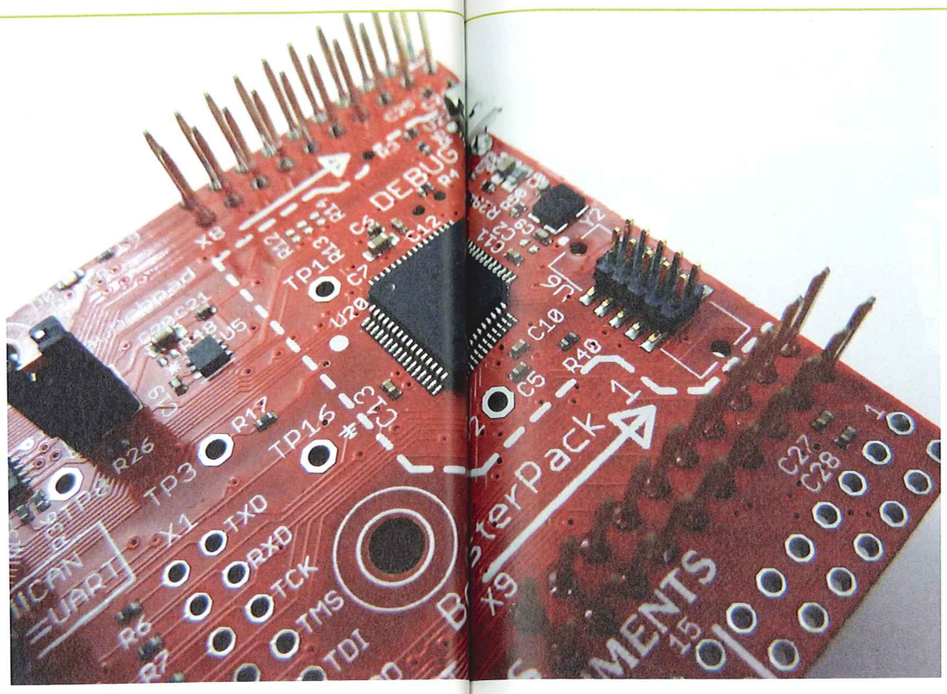
Remarquez cependant ce avec quoi nous initialisons la variable `mac`. Avec un shield Arduino, cette variable qui est ensuite passée à la méthode `begin` permet de préciser une adresse matérielle pour l'interface. Ici, nous n'avons que des zéros. En réalité, nous pourrions mettre n'importe quoi, ceci ne changerait rien au résultat. En effet, le microcontrôleur Tiva-C de notre Launchpad intègre une interface Ethernet qui possède une vraie adresse « en dur ». Celle-ci est d'ailleurs précisée sur une étiquette à l'arrière de la carte. Le code qui est généré par l'environnement utilise cette adresse directement.

Ceci n'est bien entendu qu'un simple exemple, mais représente très bien le niveau de compatibilité pour des croquis basiques. De la même manière, si vous souhaitez jouer avec les 4 leds présentes sur la carte, celles-ci sont définies dans les bibliothèques sous les noms `D1_LED` à `D4_LED`. Elles s'utiliseront en sortie exactement comme avec Arduino :

```
pinMode(D1_LED, OUTPUT);
digitalWrite(D1_LED, HIGH);
```

Ceci est déclaré dans le fichier `hardware/lm4f/variants/launchpad_129/pins_energia.h` tout comme l'ensemble des broches disponibles, incluant le monstrueux connecteur à souder libellé X11, mais aussi les connecteurs pour les Booster Packs (X6 à X9). Ainsi, pour utiliser la broche 23 du connecteur X11, on utilisera simplement `X11_23` correspondant pour le microcontrôleur à `PC_4`, soit la ligne 4 du port C.

Normalement devant de telles bonnes nouvelles un doute devrait s'installer dans votre esprit : plein de ports c'est bien, mais quid de la tension ? Le microcontrôleur Tiva-C de cette Launchpad fonctionne en 3,3V comme beaucoup de microcontrôleurs ARM Cortex-M. Cependant, celui-ci est tolérant au 5V sur toutes ces entrées/sorties (contrairement au STM32 par exemple où seules certaines broches sont tolérantes). Vous n'avez donc pas à vous inquiéter à ce niveau et une tension en entrée de 2V à 5V ne posera aucun problème. Bien entendu, un niveau haut en sortie sera 3,3V et il faut donc que les composants et modules connectés, eux aussi, le tolèrent (ce qui n'est généralement pas le cas des afficheurs LCD alphanumériques).



4. SHIELDS, BIBLIOTHÈQUES ET COMPATIBILITÉ

Sur le papier, la gamme Launchpad et la carte qui fait l'objet de cet article sont absolument fantastiques, en particulier grâce à Energia et sa compatibilité avec Arduino. Sur le papier...

Les bienfaits de la compatibilité ne peuvent être obtenus qu'avec du code qui a été pensé pour être facilement transférable et adaptable d'une plateforme à une autre. On parle généralement de « portabilité ». Un code portable qu'il s'agisse de croquis ou de bibliothèques, évite au maximum d'être dépendant d'une carte ou d'une fonctionnalité en particulier.

Comme beaucoup de cartes de développement disponibles, les Launchpad intègrent toutes une interface de programmation et de débogage utilisable via différents outils non seulement pour placer le code dans la mémoire flash, mais également pour analyser son fonctionnement. Un élément non pris en charge par l'environnement Energia, à l'instar de ce qui se fait avec Arduino.

Ceci est généralement déjà un problème lorsqu'on travaille sur du code fonctionnant sur système d'exploitation et l'est bien davantage lorsqu'il s'agit d'interagir avec du matériel.

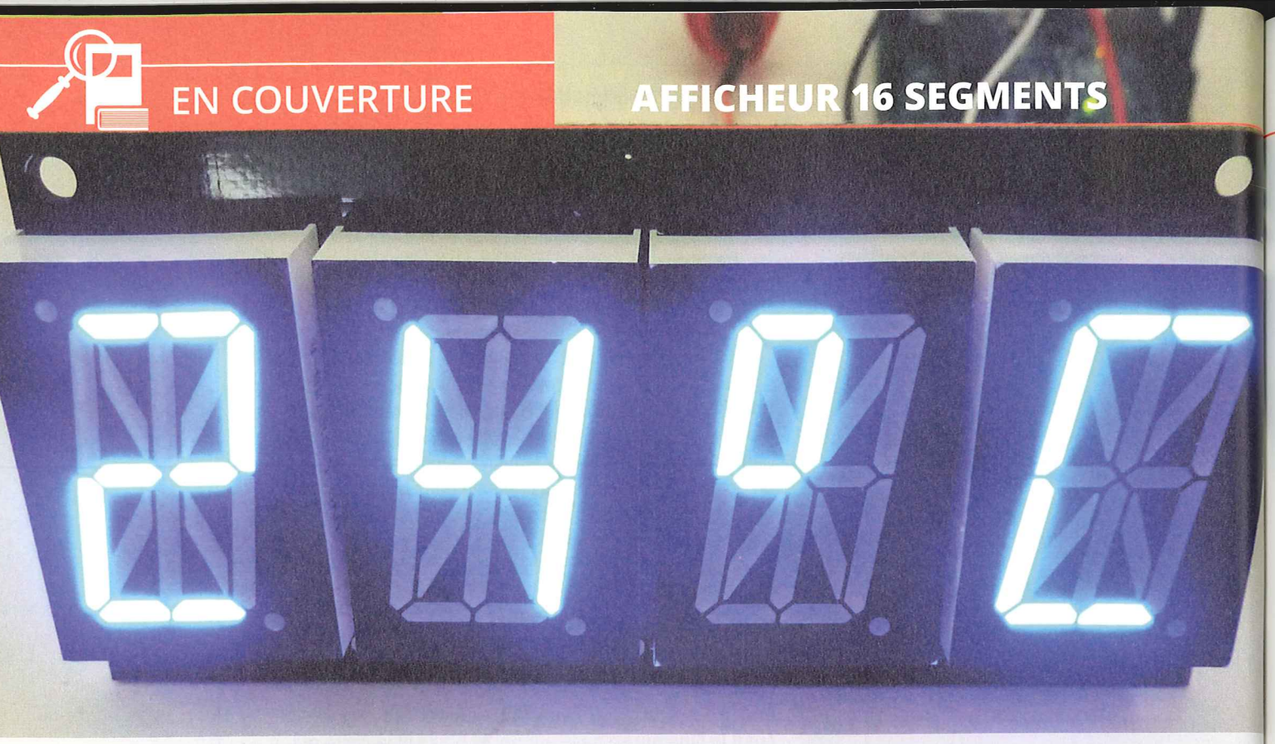
Ainsi, bien qu'une majeure partie des fonctionnalités soient disponibles de la même manière sur Arduino et Launchpad, la compatibilité se limite à un niveau bien précis d'utilisation. Des choses comme `pinMode()`, `digitalWrite()`, `analogRead()`, `SPI.transfer()` ou encore `Wire.beginTransmission()` (i2c) fonctionneront sans problème. Descendez plus proche du matériel et les choses changent, montez d'un niveau d'abstraction pour utiliser des bibliothèques comme Ucglib (*Universal uC Color Graphics Library*) et là encore les problèmes se font jour. Même des bibliothèques comme *LCD_screen Library Suite* affichant un important niveau de compatibilité entre les différentes Launchpad s'en tiennent, pour l'instant, aux modèles MSP430 et Stellaris (identique à Tiva-C, mais d'une génération différente). Une tentative pour faire fonctionner un écran LCD comme celui présenté dans le précédent numéro, théoriquement supporté, débouchera sur un message comme « `#error Platform not supported` ». Le microcontrôleur de notre Launchpad n'est tout bonnement par encore pris en charge...

Bien sûr, ceci n'a rien d'étonnant en vérité. Les Launchpad Stellaris se basaient sur le microcontrôleur éponyme (sur base ARM Cortex-M3 héritée de Luminary Micro), les Arduino sont majoritairement des Atmel AVR (sauf le Due qui se base sur un SAM3X) et la carte qui nous concerne ici un microcontrôleur Tiva-C (ARM Cortex-M4). C'est un fait, en basculant votre projet d'une carte à l'autre, vous aurez des problèmes et ceci vaut également si vous comptez utiliser des « briques » conçues pour une plateforme avec une autre.

En résumé, sans pour autant dire qu'un retour à `Blink.ino` est indispensable, la prise en main d'une nouvelle carte passera obligatoirement par un apprentissage et de longues heures à essayer d'obtenir quelque chose d'au moins équivalent à ce que vous aviez l'habitude d'avoir.

Cette Lauchpad dispose de capacités phénoménales (Ethernet, USB, etc.) à un prix qu'on pourrait presque qualifier de dérisoire. Une vaste majorité de celles-ci et de la puissance du microcontrôleur ne sera réellement accessible qu'avec un environnement de développement « professionnel » (IAR, Keil, Mentor, CCS ou Vim+GCC) et dans une moindre mesure avec Energia et une grosse part d'implication personnelle. Si l'idée de tout simplement prendre vos croquis Arduino pour les utiliser avec une Laucnhpad vous a effleuré l'esprit, chassez-la rapidement et dites-vous qu'on n'a jamais rien sans effort...

Énormément de réalisations et de bibliothèques existent pour Arduino et seule une petite partie est utilisable actuellement avec cette Launchpad. Une partie plus importante est portable et pourra facilement être adaptée, mais la plus grande partie des travaux existants ont été faits sur ou pour Arduino. Là, il n'y aura pas de miracle, il faudra que les développeurs initiaux ou des utilisateurs motivés revoient chaque projet et bibliothèque en pensant à la portabilité... DB



COMMENT PILOTER 64 LEDS OU PLUS AVEC 4 FILS : LE REGISTRE À DÉCALAGE

Denis Bodor

Tout le monde aime les leds ! C'est un fait, la preuve, il y en a partout... mais partout ! Amusez-vous une fois à les compter autour de vous, c'est impressionnant. En contrôler une, c'est facile. 2, 3, 10... 15... 30. Là ça commence à devenir difficile et plusieurs techniques différentes peuvent être mises en œuvre. L'une d'elles consiste à utiliser un composant particulier. Un circuit intégré appelé le registre à décalage.

Une solution courante pour piloter des dizaines de leds consiste à les connecter en matrice. Avec techniques et astuces, il est possible sans aucun composant additionnel autre que des résistances, d'utiliser le multiplexage et reposer sur la persistance rétinienne. Dans les faits, seule une partie des leds sont effectivement actives à un instant *t*, mais le système de vision humain n'est pas suffisamment efficace pour percevoir ce clignotement. Mais... parce qu'il y a un « mais », on ne peut tricher avec la physique et si nous prenons le cas d'une matrice de 16 lignes de 16 leds, une seule ligne étant active à la fois, on obtient une intensité globale 16 fois inférieure (et donc un rapport cyclique de 1/16). Il est possible d'améliorer les choses puisque les leds peuvent être brièvement alimentées avec davantage de courant (*peak current* dans les documentations) pour une durée et avec une période de repos spécifiée par le fabricant. Bien entendu, on arrive rarement à totalement compenser la perte d'intensité lumineuse de cette manière. Certaines leds, coûteuses, sont même dédiées à cet usage et dans ce cas ne sont pas conçues pour une alimentation continue. C'est une manière de dire que cela règle le problème, mais pour des leds standards, ce n'est pas le cas.

Une autre solution consiste à utiliser un composant dédié qui se charge de piloter les sorties connectées aux leds. L'avantage est de ne plus avoir de rapport cyclique et donc une pleine intensité pour chaque led. Le contrecoup en dehors de l'ajout de composants (relativement économiques, je vous rassure) est la quantité de courant consommé. Là, il ne s'agit plus de clignotement et donc d'alimenter 16 leds à la fois en fournissant par exemple 10mA par led pour un total de 160mA. Avec 16*16 leds à 10mA, on arrive à un total de 2,56 ampères soit, avec une tension de 2 volts, une puissance de pas moins 5 watts !

1. LE REGISTRE À DÉCALAGE SIMPLE

Pour comprendre le principe de fonctionnement de ce type de composants, imageons nos propos. La pizza de la PWM du numéro 1 a bien marché, restons donc dans le milieu alimentaire. Vous voici à la plonge dans un monumental restaurant. Vous avez donc un flot constant d'assiettes sales qui arrivent, mais la personne qui sèche la vaisselle n'accepte que des assiettes propres (logique) et en jolies piles de 16. Mais le patron du restaurant devant le caractère peu avenant de la personne en question a trouvé une méthode.

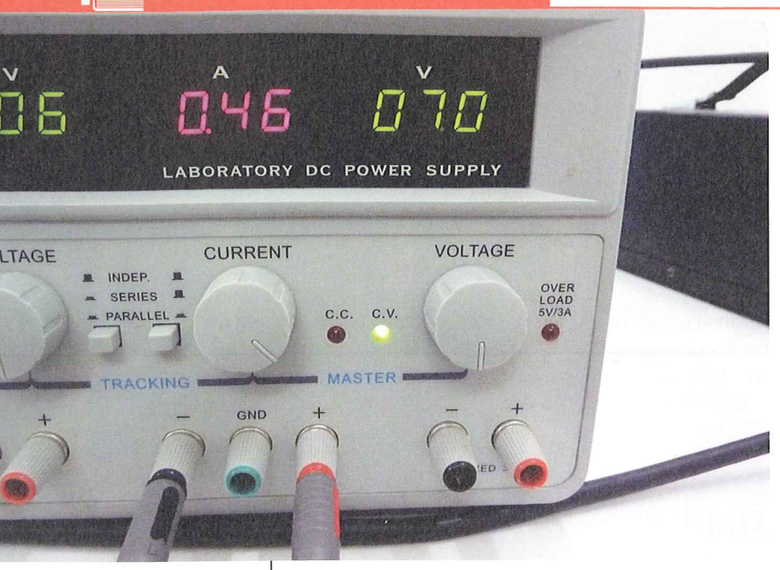
Les serveurs qui débarrassent les tables vous apportent les assiettes. Vous les prenez une à une pour les nettoyer et vous les posez sur le côté de l'évier : clock, clock, clock... Au bout de 16 assiettes propres sur la pile, vous la poussez en direction du sécheur : laaaatchh ! Et c'est reparti pour un tour.

Un registre à décalage, c'est exactement cela. Il faut juste lui dire quand poser l'assiette qu'on lui présente et quand pousser la pile. Il prend donc en entrée une donnée de un bit qui peut être 0 ou 1 (état bas ou état haut). À chaque fois qu'on lui présente une donnée, on lui demande de la prendre en compte avec un signal d'horloge, ou *clock* en anglais, abrégé CLK. Le registre ne sait pas quand la pile est complète. Au bout du nombre qu'il est capable d'accumuler, c'est encore à vous qu'il revient de lui demander de « pousser » l'ensemble dans sa mémoire via une broche appelée *latch* (loquet ou gâchette en français).

Ceci revient donc à suivre le scénario suivant :

- présenter l'assiette,
- dire « prend-la »,
- au bout de *n* assiettes, dire « pousse la pile ».

Que se passe-t-il si on ne *latch* pas ? Le gars de la plonge n'est pas un homme compliqué, il va jeter l'assiette la plus en bas dans la pile (la première qui a été mise). Si on continue le



L'alimentation de laboratoire fournissant le courant au module d'affichage avec un texte standard n'utilisant pas tous les segments. Nous avons là un courant de 460mA en 7V, soit 3,2W. Combien si vous utilisez 4 modules chaînés ? Il suffit de multiplier, presque 13 watts !

processus et qu'on *latch*, c'est la pile de *n* assiettes telle qu'elle existe qui sera poussée, avec la dernière assiette dedans, mais plus la première.

Une catastrophe ? Pas du tout, au contraire. L'assiette/données tout en bas de la pile n'est pas oubliée et elle ne disparaît pas. Elle est simplement présentée sur une sortie du registre à décalage. Ainsi, si l'on connecte cette sortie à l'entrée de données d'un autre registre à décalage et que les deux répondent ensemble au signal d'horloge, nous obtenons juste une pile deux fois plus grande (mais divisée sur les deux registres). En reliant les deux broches */latch*, l'ensemble peut alors se comporter comme un seul composant, deux fois plus grand.

Un exemple d'un tel composant est le 74HC595 qui coûte environ 50 centimes d'euros. C'est un registre à décalage 8 bits série vers parallèle avec latch. Il possède 8 sorties, contrôlées par une gâchette (*latch*) et ce avec une possibilité d'activer les sorties ou non (*output enable*). On présente les données de manière série sur Ds en utilisant le signal d'horloge SHcp. Quand les 8 bits sont entrés, on active le stockage en utilisant la broche STcp ou en d'autres termes, on enregistre l'état. Les états seront présentés sur les sorties Q0 à Q7 dès lors que la broche /OE est à la masse. Ce n'est pas tout, à ce moment la sortie Q7' présente l'état du premier bit entré. Cette broche peut donc être reliée au Ds d'un autre 74HC595 et ainsi de suite...

Le 74HC595 peut fonctionner à une fréquence de 100 Mhz selon conditions (tension, température, etc.), mais cela reste un circuit logique simple. Ainsi, il n'est pas en mesure de fournir (ou recevoir) plus de 20mA par sortie et pas plus de 70mA en totalité (GND/Vcc), avec une dissipation de puissance totale de 500mW. En clair, vous ne pouvez pas directement piloter, par exemple, 8 leds blanches ou bleues à 20mA ou plus chacune. Il vous faudra utiliser des transistors ou des MOSFET pour cela.

En revanche, il existe des composants qui fonctionnent comme un 74HC595, mais qui comprennent des fonctionnalités spécialisées pour le pilotage des leds...

2. LE TLC5926, UN REGISTRE UN PEU À PART

Ce composant est un *16-Channel Constant-Current LED Sink Driver*. En bon français, c'est un circuit intégré avec 16 sorties parallèles, une entrée série et tout le mécanisme propre à un registre à décalage (clock, latch, activation de sortie, etc.). Mais celui-ci a un gros plus : on peut y brancher directement les leds, sans résistance, car le composant est capable de se charger comme un grand de la régulation du courant. Mieux encore, la tension utilisée pour les leds est indépendante de celle de l'interface de communication. En d'autres termes, il vous est possible par exemple de piloter 16 leds avec une tension de 7V depuis un système (Arduino, Raspberry Pi, Launchpad) fonctionnant en 3,3V ou 5V.

Le TLC5926 dispose des broches suivantes :

- GND : la masse,
- VDD : la tension d'alimentation du composant qui détermine aussi les niveaux de tension de l'interface (pour une Raspberry Pi par exemple, le TLC5926 est alimenté en 3,3V),
- SDI : la ligne de données série en entrée,
- CLK : le signal d'horloge qui peut monter jusqu'à 30 Mhz,
- LE : la gâchette (*Latch Enable*),
- /OE : l'activation de la sortie, *Output Enable* (notez la barre de fraction dans la désignation de la broche, ceci signifie que le signal est inversé, actif en étant à la masse, on dit « actif à l'état bas »).

24-PIN
DBQ, DW, OR PWP PACKAGE
(TOP VIEW)

GND	1	24	VDD
SDI	2	23	R-EXT
CLK	3	22	SDO
LE(ED1)	4	21	$\overline{OE}$(ED2)
OUT0	5	20	OUT15
OUT1	6	19	OUT14
OUT2	7	18	OUT13
OUT3	8	17	OUT12
OUT4	9	16	OUT11
OUT5	10	15	OUT10
OUT6	11	14	OUT9
OUT7	12	13	OUT8

Le TLC5926 n'est pas un composant très volumineux ou complexe, et il existe en plusieurs « packages » plus ou moins faciles à utiliser par un hobbyiste (source : documentation TLC592x de Texas Instruments).

Cette broche est tantôt également désignée par « BLANK » puisqu'en étant mise à l'état haut, la sortie est « masquée »,

- OUT0 à OUT15 : les 16 sorties permettant de connecter les **cathodes** des leds. Le TLC5926 ne fournit pas le courant, il l'accepte (d'où le *sink* dans la désignation du composant). On branche donc l'anode (la longue patte) à la tension d'alimentation avec plein de courant disponible et la cathode (la courte) au TLC5926,
- SDO : la sortie série permettant de chaîner les TLC5926 afin de piloter encore plus de leds,
- R-EXT : permet la connexion d'une résistance qui déterminera le courant devant circuler dans les leds.

Vous l'avez compris, R-EXT est un élément commun à toutes les leds. Il n'est pas possible de piloter individuellement le courant régulé pour chaque sortie. Le calcul de la résistance répond à une formule détaillée dans la documentation (*datasheet*) :

```
Vr-ext = 1,26 V * VG
Iref = Vr-ext/Rext
Iout = Iref*15*3^(CM-1)
```

De quoi ?! Pas de panique, il suffit de procéder par étapes :

- Vr-ext est la tension sur R-EXT qui est branchée à la masse via la résistance,
- VG est le gain qui est une valeur programmable par défaut (à la mise sous tension) à 127/128 soit 0,992,
- Iref est le courant circulant déterminé par la valeur de la résistance Rext (loi d'Ohm : U=RI et donc I=U/R),
- Iout est le courant souhaité qui traverse une led,
- CM est le multiplicateur de courant avec une valeur par défaut de 1.

En tripatouillant les formules et en utilisant les valeurs par défaut, on obtient alors :

```
Vr-ext = 1,26 V * 0,992 = 1,25 V
Iout = (1,25V/Rext)*15
```

ou, plus intéressant :

```
Iout/15 = 1,25V/Rext
Rext = 1,25/(Iout/15)
```

Ainsi, pour des leds utilisant 20mA, nous calculons :

```
Rext = 1,25/(0.02/15)
Rext = 1,25/0.001333
Rext = 937.5 =~ 1 Kohm
```

L'avantage du TLC5926 est évident, car même si le composant coûte un peu plus cher qu'un simple registre à décalage (environ 1,5 euros), une simple résistance suffit pour piloter 16 leds avec 4 malheureux fils. Notez également que ce composant n'est pas le seul de sa gamme, nous avons également le TLC5947 et ses 24 sorties ou encore le TLC5940 offrant 16 sorties, mais disposant chacune d'un paramètre de réglage d'intensité (PWM) de 4096 niveaux. Ajoutez à cela que les TLCxxxx, provenant de *Texas Instruments*, ne sont pas les seuls circuits intégrés offrant ce type de fonctionnalités. On pourra également citer le LT3745 de *Linear technology*, le A6282 de *Allegro MicroSystems* ou encore le AS1110 de *AMS AG*.

3. UN PEU DE PRATIQUE : UN AFFICHEUR 4 FOIS 16 SEGMENTS

Pour voir en pratique comment piloter un composant comme le TLC5926, nous allons utiliser un module tout fait de chez *Embedded Adventure*. Celui-ci, le DSP-0401B, se décline en pas moins de cinq versions en fonction de la couleur proposée par les afficheurs leds : rouge, jaune, bleu, blanc ou vert. L'ensemble est relativement simple : quatre afficheurs led 16 segments connectés à autant de TLC5926 chaînés. Le circuit propose des emplacements pour broches à souder libellées :

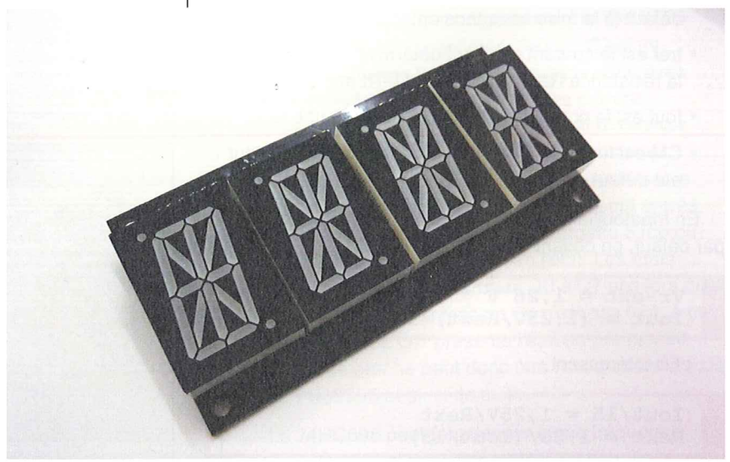
- VCC : alimentation pour les TLC5926,
- LAT : latch/gâchette,
- BLK : *blank* ou /OE,
- CLK : l'horloge,
- SIN : entrée de données série (*Serial IN*),
- GND : la masse.

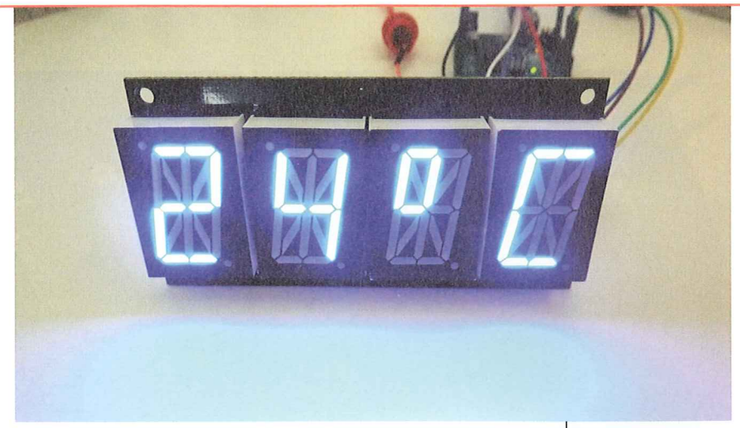
Et de l'autre côté, les mêmes désignations, mais SIN est remplacé par SOUT pour la sortie série. C'est le SDO du dernier TLC5926 qu'il est possible de relier au SDI (SIN sur le circuit) d'un autre afficheur 4 fois 16 segments, en compagnie des autres signaux. De plus, trois autres connecteurs proposent également :

- VCC : toujours l'alimentation des TLC5926,
- GND : l'indispensable masse commune,
- LED_SUP : qui est l'alimentation des leds et qui peut donc être d'une source et d'une tension différentes puisqu'une carte comme une Arduino sera bien incapable de fournir quelques 1,3 ampères en 7 volts (pour les afficheurs à leds bleues) lorsque toutes les leds sont actives.

Les modules d'affichage se satisferont de 5 volts entre LED_SUP et GND pour les versions vertes, rouges et jaunes, mais cette tension devra être au minimum de 7 volts pour les afficheurs bleus et blanc. Une tension plus importante, jusqu'aux 17 volts maximum supportés par le TLC5926, peut être utilisée, mais il faudra prendre en compte un élément important : plus la tension sera importante, plus les régulateurs de courant embarqués devront dissiper l'énergie qui n'ira pas dans les leds. En d'autres termes, les composants vont chauffer. Une sécurité existe avec un arrêt automatique (*overtemperature shutdown*) à 175°C, mais si vous voulez éviter cette situation et que vous tenez absolument à alimenter des leds puissantes avec une tension largement au-delà de ce qui est nécessaire, il faudra prévoir un système de dissipation thermique sur les composants : des radiateurs, voire un système de flux d'air si l'ensemble est dans un espace clos.

Le module d'affichage 4 fois 16 segments. Les afficheurs disposent de points, mais ceux-ci ne sont pas branchés aux TLC5926 faute de sorties suffisantes.

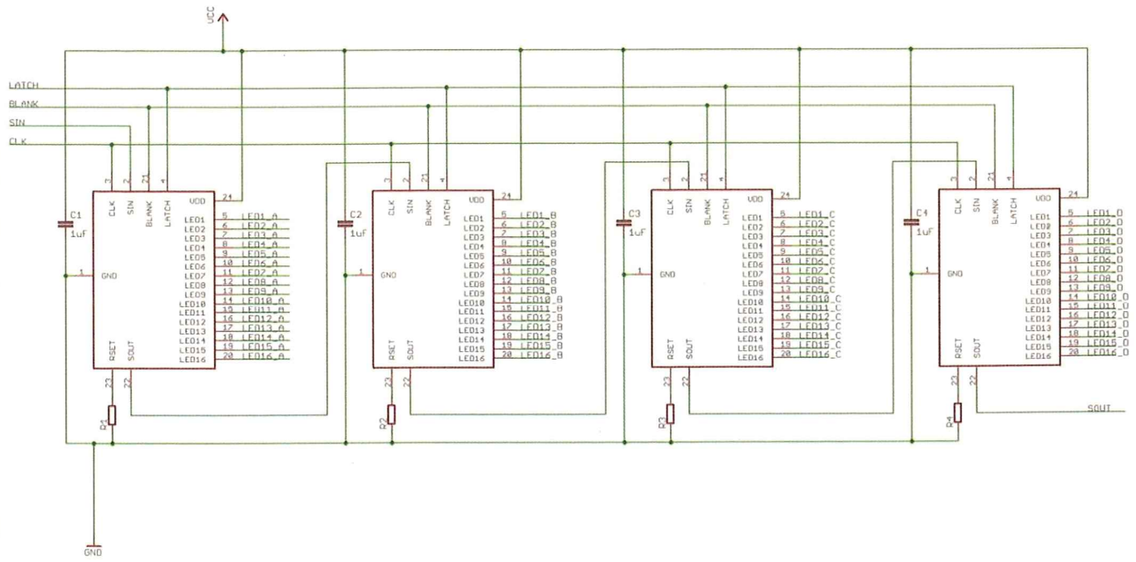




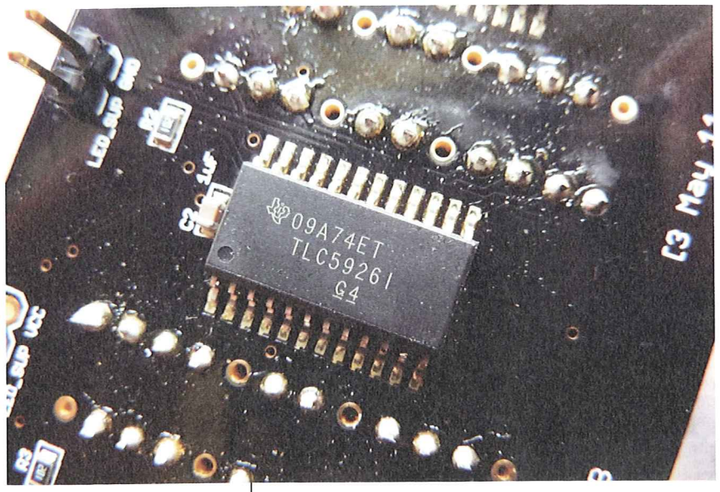
Avec 16 segments, il devient possible d'afficher non seulement des chiffres, mais également des lettres et une certaine quantité de symboles.

Le circuit utilisé pour le module est présenté dans sa documentation et est relativement simple. Nous y trouvons les 4 TLC5926, les afficheurs 16 segments, des résistances de 1 Kohm et des condensateurs de découplage de 1 µF entre Vcc et la masse (protection contre les variations de tension hautes fréquences pouvant perturber le circuit intégré). Et c'est tout.

Tout ce que nous avons à faire pour piloter cet afficheur est de présenter un bit sur SIN, le valider avec une impulsion sur CLK et de répéter l'opération 64 fois avant de « pousser » les données avec une impulsion sur LAT. Enfin, si BLK est à Vcc, l'affichage restera vide, mais en mettant cette patte à la masse le motif composé par nos données sera illuminé.



Schémas du circuit du module d'affichage. Les quatre TLC5926 sont chaînés et on distingue clairement la liaison du SOUT d'un TLC5926 sur le SIN du suivant.



Un TLC5926 de Texas Instruments. On distingue le condensateur céramique de découplage à gauche et plus loin une résistance de 1 Kohm permettant de régler le courant passant dans les leds. Les soudures de qualité moyenne et les traces de flux sur le circuit sont d'origine. À 23€ le module, on aurait espéré un peu plus de soin...

4. UN PEU DE PRATIQUE : CÔTÉ LOGICIEL

Tout à fait honnêtement, piloter les TLC5926 de manière logicielle (il est aussi possible d'utiliser un bus SPI pour cela) est relativement simple, mais la plus grande des vertus pour un programmeur est la paresse. Le réflexe, comme toujours, est donc de se tourner vers le web à la recherche du travail d'un développeur ayant produit une bibliothèque Arduino. Nous trouvons notre bonheur assez rapidement sur GitHub et en particulier la page **https://github.com/2splat/arduino-TLC5926**. Nous trouvons là un fichier Zip contenant une bibliothèque à installer via l'IDE Arduino ou à la main dans votre répertoire `library` de votre *sketchbook*.

4 caractères ce n'est pas assez pour afficher le nom du magazine, mais cela ouvre des possibilités déjà intéressantes...

La bibliothèque s'utilise de plusieurs manières, mais celle que je préfère consiste à contrôler pleinement l'envoi des données (et je n'aime pas particulièrement les choix d'architectures faits pour les autres fonctions et méthodes). Voici un exemple :

```
#include <TLC5926.h>

#define SDI_PIN 2
#define CLK_PIN 3
#define LE_PIN 4
#define OE_PIN 5

TLC5926 shreg;

void setup() {
  shreg.attach(1,
SDI_PIN, CLK_PIN,
LE_PIN, OE_PIN);

  shreg.shift(0xB4B4);
  shreg.shift(0x6868);
  shreg.shift(0xB4B4);
  shreg.shift(0x6868);
  shreg.latch_pulse();
}

void loop() {
}
```

L'accès au module prendra la forme de la variable `shreg` (comme *shift register*) et on utilise en premier lieu la méthode `attach()` permettant de spécifier les broches utilisées pour la connexion. Notez le `1` en premier argument, contrairement à ce qu'on peut penser, ceci ne détermine pas le nombre de TLC5926 chaînés puisque la méthode `send()` qui permet d'envoyer des bits suivis d'une impulsion automatique sur la gâchette (LAT) ne prend en argument qu'une valeur de 16 bits et ce sans utiliser un nombre de TLC5926 chaînés (cette fonction est faite pour un seul TCL5926).

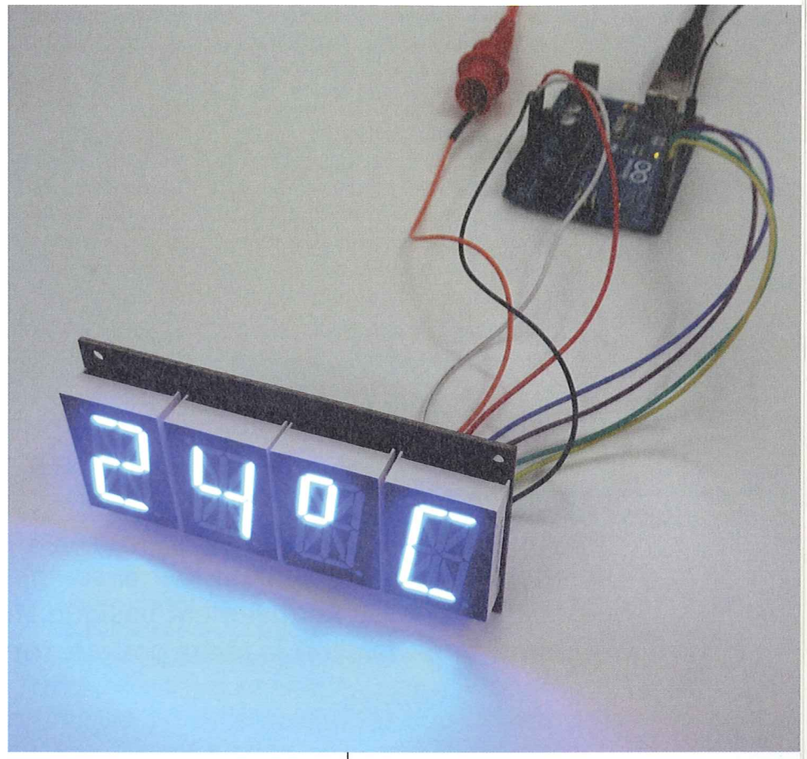
Quoi qu'il en soit, ce qui nous intéresse ici est d'envoyer des données en quantités connues pour satisfaire nos quatre TLC5926, soit 4 fois 16 bits. Chose que nous pouvons faire avec la méthode `shift()` utilisée 4 fois, suivie d'un `latch_pulse()` provoquant deux rapides changements d'état sur la broche désignée par `LE_PIN` (bas/haut et haut/bas). `OE_PIN` étant par défaut à la masse, nos données s'affichent sur les leds : « TOTO » (j'ai pas trouvé mieux).

Notez qu'il vous est possible de régler l'intensité lumineuse de l'ensemble de l'afficheur avec `brightness(x)` où `x` est une valeur entre 0 et 255. C'est en réalité un simple `analogWrite()` qui est utilisé, ce qui implique que `OE_PIN` doit donc être une sortie capable de faire de la PWM (symbole ~ devant le numéro de la broche sur la carte).

Cette bibliothèque est capable de faire plus que cela et fournie, par exemple un accès à la configuration du TLC5926. Rappelez-vous que le gain qui contrôle le courant dans les leds est programmable. Une détection d'erreurs est également disponible ainsi que des fonctionnalités permettant de déboguer votre code. Il semblerait cependant que l'auteur n'ait pas accordé une importance majeure au fait que l'on puisse utiliser les TLC5926 comme dans notre module...

Mais ce qui importe maintenant n'est pas de descendre dans le détail de l'implémentation de cette bibliothèque, mais plutôt de monter à un niveau d'abstraction supérieur. En effet, nous pouvons à présent allumer et éteindre les leds à souhait de manière à former des symboles... à condition de faire toutes ces petites combinaisons de bits à la main. Voilà exactement un travail digne d'un programme et non d'un programmeur : faire faire le sale boulot au code, jamais à vous-même, car comme le dit l'agent Smith : « n'envoyez jamais un humain faire un travail de machines ».

C'est précisément là ce que nous allons voir dans l'article juste après celui-ci... DB



Vue d'ensemble du montage entre le module et une carte Arduino Uno. Les TLC5926 sont alimentés par l'Arduino, mais les leds reçoivent leur courant d'une alimentation de laboratoire. Un bloc d'alimentation suffisamment « costaud » pourrait aussi faire l'affaire à condition que le courant soit bien propre et surtout suffisant.

TLC5926 : AFFICHER ET FAIRE ÉVOLUER VOTRE CODE !

Denis Bodor

Après une première introduction à propos des registres à décalage et du TLC5926 ainsi qu'une petite mise en pratique avec un module d'affichage à leds 4 fois 16 segments, il est temps de profiter de la situation pour parler de logique de programmation. Nous avons en effet ici le socle idéal pour voir ensemble comment construire au-dessus d'un code relativement basique afin de se créer les outils pour nous faciliter la tâche pour de futures utilisations.

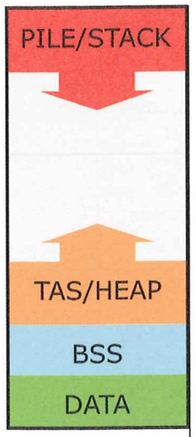
Bien des projets et des réalisations reposent sur des croquis relativement linéaires. En reprenant notre exemple de l'article précédent, il est aisé de l'étoffer pour ajouter des fonctionnalités comme, par exemple, lire un capteur de température, une RTC, l'état de broches en entrées et envoyer en conséquence des motifs sur les afficheurs à leds. Tout ceci peut trouver place dans `loop()` qu'il s'agisse du code pour récupérer les données ou pour l'affichage à grands coups de `shift()` et de `latch_pulse()`. De cette approche résulte généralement une fonction `loop()` assez grosse et monstrueuse et un croquis ne profitant absolument pas des avantages offerts par le langage C/C++ sur lequel est basé le « langage » Arduino/Wiring.

En effet, tout l'intérêt de disposer d'un langage comme le C est bel et bien de faire un usage judicieux des fonctions et ne pas programmer comme s'il s'agissait de BASIC. Vous savez :

```
10 INPUT "NOM?"; NOM$
20 PRINT "COUCOU "; NOM$
30 GOTO 30
```

Mais ce n'est pas tout, en utilisant un microcontrôleur comme celui d'une carte Arduino, il est important de prendre en compte les ressources utilisables. Je parle en particulier du peu de mémoire dont dispose, par exemple, l'ATmega328P de l'Arduino Uno. Celui-ci ne possède que 2 Ko de SRAM, c'est-à-dire de mémoire vive pour les variables et les arguments de fonctions. Ces 2 Ko de SRAM seront utilisés sous la forme de quatre zones distinctes par le code (binaire) exécuté par le microcontrôleur :

- Les variables globales et statiques : il en existe deux types, celles initialisées avec une valeur qui trouve place dans « data » et celles initialisées à zéro (ou NULL) ou non initialisées qui sont placées dans BSS (historiquement, dans les années 50, pour *Block Started by Symbol*). Ces deux zones forment les données statiques (*static data*), car ces espaces sont définis une fois pour toutes et ne changent pas durant le déroulement du programme (leur contenu oui, mais pas les espaces effectivement utilisés).
- Le tas ou *heap* en anglais : accueillera les données des variables définies dynamiquement durant l'exécution du programme avec, par exemple des fonctions comme `malloc()`. Cela peut être utile si, de temps en temps, votre programme a besoin d'une masse de mémoire qu'il n'utilisera que temporairement. On utilise alors `malloc()` et `free()` pour obtenir et libérer un espace mémoire. Certaines fonctions font également ce type de choses lorsqu'elles sont utilisées, on dit qu'elles allouent de la mémoire. D'autres utilisent de la mémoire déjà allouée. Il est peu probable que pour un usage courant vous ayez vous-même à utiliser ce type d'allocation (à moins de savoir exactement ce que vous faites et dans ce cas, il est peu probable que cet article vous apprenne quoi que ce soit), mais les bibliothèques que vous allez utiliser, elles, peuvent s'en servir (c'est le cas de la bibliothèque `SD` livrée avec l'IDE Arduino, par exemple). Vous l'avez compris, la particularité du tas, c'est qu'il peut grossir et rétrécir durant la vie du programme. Pire, à force d'allocation de mémoire et de libération, le tas peut se retrouver avec des trous et donc dans l'absolu, occuper moins de mémoire sans pour autant baisser.
- La pile ou *stack* : cette zone se trouve généralement de l'autre côté de la mémoire et grossit dans le sens inverse du tas. La pile est également dynamique. C'est l'endroit où vont être stockées les variables (non statiques) utilisées localement dans les fonctions ainsi que les arguments des fonctions appelées. Schématiquement, si la fonction `setup()` appelle `toto(42)`, `42` est mis sur la pile et le programme saute à l'adresse où se trouve le code de `toto()`. Comme ce code est exécuté ponctuellement (ou pas du tout), il serait idiot que ses variables occupent toujours un espace de mémoire. C'est donc sur la pile que cet espace est utilisé. Lorsque la fonction se termine, cet espace est à nouveau utilisable, les données sont dépilées et la pile « baisse ». Notez que l'adresse de retour, à laquelle le programme se poursuit est également mise sur la pile. La pile ne peut pas avoir de trou, c'est une pile et on ne peut pas retirer un élément sans retirer ceux qui se trouvent « au-dessus »



Organisation de la mémoire, découpée en zones en fonction du type de variables et de données qui y trouvent place. En vert et bleu, les zones pour les variables statiques et globales, qui ne « bougent » jamais et en rouge et orange les zones dynamiques. Si le tas et la pile se touchent, c'est le crash !

de lui. Notez qu'une pile a, en principe, une taille maximum fixée par le système et le compilateur. Sur GNU/Linux avec GCC par exemple, le maximum par défaut est de 8 Mo (réglable avec `ulimit`). Dans le cas du microcontrôleur AVR d'une carte Arduino, c'est 2048, soit toute la SRAM.

L'architecture selon laquelle le tas et la pile grossissent en sens opposé permet une utilisation dynamique optimale de la mémoire. Mais celle-ci est tout de même une ressource limitée et si, à force de grossir, la pile rencontre le tas ou inversement, c'est la catastrophe ! Les données de la pile peuvent ainsi parfaitement commencer à écraser celles du tas ou celles des données statiques, ce qui mènera à un comportement presque imprédictible du programme (« presque » parce que, dans les faits, ceci peut devenir une faille de sécurité exploitable).

Programmer sur une plateforme comme Arduino ne se fait donc pas de la même manière que sur une machine comme un PC avec des gigaoctets de mémoire disponibles. Un excellent exemple est la récursivité. C'est une technique de programmation consistant, pour une fonction, à s'appeler elle-même, par exemple pour parcourir une structure ou calculer une factorielle. Exemple en pseudo-code :

```
fonction factorielle(entier n) {
  si n = 0
    retourner 1
  sinon
    retourner (n*factorielle(n-1))
}
```

Ceci, implémenté sur une carte Arduino ou toute autre plateforme avec très peu de mémoire est une très mauvaise idée, car cela provoquera une utilisation importante de la pile. Après un nombre donné de récursions, la pile finira par toucher le tas et les données statiques avant de faire un tour de la SRAM et commencer à écraser le bas de la pile... En un mot : boum !

1. OÙ EN ÉTIONS-NOUS ?

Revenons à nos moutons. Nous avons donc l'intention d'ajouter du code de manière judicieuse dans le but de plus facilement afficher du texte, des chiffres et des symboles avec notre module à base de TLC5926. Nous savons que chaque bit envoyé au module correspond à un segment d'un des quatre afficheurs. Pour établir une correspondance, il nous suffit de jeter un œil à la documentation fournie par *Embedded Adventures* (**http://www.embeddedadventures.com/datasheets/DSP-0401B_hw_v4.pdf**).

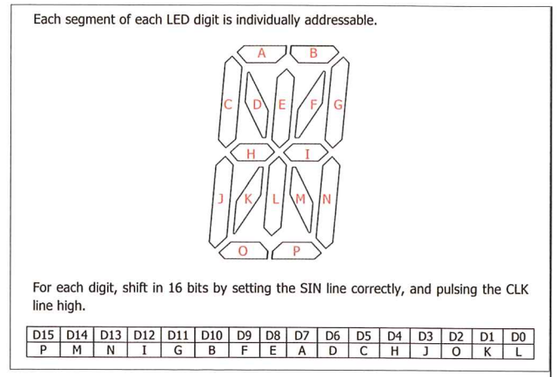
Page 5, on y trouve un sympathique diagramme d'un afficheur suivi d'un tableau avec les positions des bits de 15 à 0 et le segment concerné désigné par une lettre de A à P. Le tableau est trié par position des bits et non des segments. Il nous suffit donc de reporter cela dans un tableur comme celui de LibreOffice et d'user de quelques formules savantes avant de trier le tout par lettre de segment. Ce qui nous donne :

segment	bit	valeur	hex
A	7	128	0x0080
B	10	1024	0x0400
C	5	32	0x0020
D	6	64	0x0040
E	8	256	0x0100
F	9	512	0x0200
G	11	2048	0x0800
H	4	16	0x0010
I	12	4096	0x1000
J	3	8	0x0008
K	1	2	0x0002
L	0	1	0x0001
M	14	16384	0x4000
N	13	8192	0x2000
O	2	4	0x0004
P	15	32768	0x8000

À présent, nous savons que pour allumer le segment A d'un afficheur (celui en haut à gauche), il nous

suffit d'envoyer `0x0080`. Pour le segment M (diagonale inférieure droite), ce sera `0x4000`, et ainsi de suite. Et pour en allumer plusieurs ? C'est simple, il suffit de combiner les valeurs avec un OU logique. Comme il s'agit de valeurs correspondant à une position d'un bit, il n'y a là que des exposants de 2. Nous pouvons donc également les additionner pour obtenir le même résultat. Ainsi, pour faire un "I", nous pouvons allumer E et L, soit `0x0100` et `0x0001`, et donc envoyer `0x0101`. Pour tout allumer sur un afficheur, il faut que tous les bits soient à 1. Cela nous donne une addition de toutes les valeurs possibles et donc `0xFFFF`.

C'est maintenant très facile de dessiner des caractères. Il nous suffit d'utiliser le diagramme d'un afficheur, lister les segments à activer et faire une addition. Un peu de travail et nous pouvons

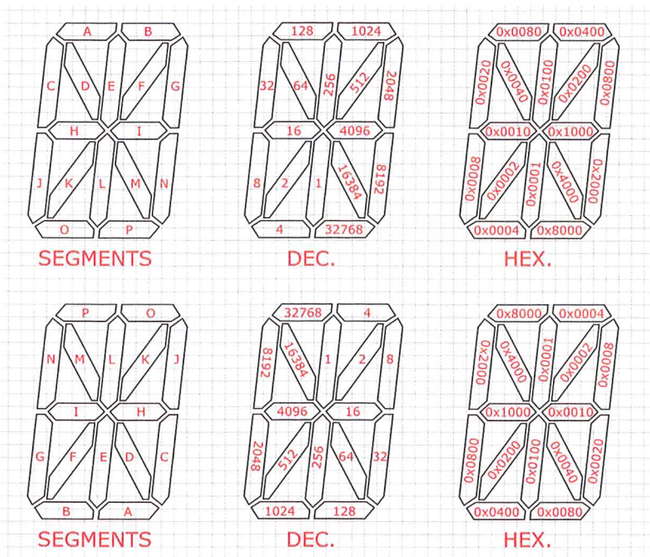


La documentation du module fournit un diagramme ainsi qu'un tableau établissant la relation entre la position de chaque bit et le segment concerné sur l'un des afficheurs. Nous n'avons pas besoin de suivre les pistes sur le schéma.

créer ainsi une liste de valeurs pour créer un alphabet ou plus exactement une table de caractères. Allons-y !

Donc pour faire un beau "A", nous avons A+B+C+G+H+I+J+N, ce qui nous donne 128+1024+32+2048+16+4096+8+8192, soit 15544 et donc `0x3CB8`. Et ça marche... à condition que vous ayez le module posé dans le bon sens. Ou du moins, dans le sens qui fait en sorte que la première valeur 16 bits envoyée soit celle qui finit complètement à gauche et la dernière à droite. Voir les choses dans ce sens semble plus logique puisque le premier symbole envoyé est aussi le premier symbole qui est lu par un observateur. On envoie comme on écrit, de gauche à droite. Bien entendu, si on envoie qu'une seule valeur 16 bits et qu'on latch, le symbole sera complètement à droite, car c'est le premier TLC5926 dans la chaîne...

Il est temps de programmer tout cela pour n'avoir à faire cette gymnastique qu'une fois par symbole.



À chaque segment désigné par une lettre correspond la position d'un bit dans les 16 envoyés sur LTC5926. On peut donc décliner cette nomenclature en utilisant directement le poids de ce bit dans une valeur 16 bits, en décimal et en hexadécimal. Pour créer un caractère, il ne reste plus qu'à faire une addition. En seconde ligne, vous avez la même chose, mais pour un afficheur retourné à 180°.

2. PASSE 1 : BÊTE ET MÉCHANT, OU PRESQUE...

Partons d'une constatation simple : nous avons besoin d'une liste de symboles facilement accessibles, et ce avec leur correspondance avec la valeur du motif à dessiner. Très paresseux, nous ne voulons pas quelque chose comme symbole 1 = "A", symbole 2 = "B", etc. S'il faut retenir la position des symboles en utilisant un chiffre, autant mémoriser la valeur hexadécimale ! Le truc génial serait qu'on définisse directement `"A"=0x3CB8` et qu'on puisse envoyer directement "A", "B", et ainsi de suite. Pour autant, créer des dizaines de variables `lettre_a`, `lettre_b`, etc., ne semble instinctivement pas une très bonne idée.

Ce qu'il nous faut c'est un tableau de caractères/symboles et qu'on puisse désigner les cellules avec une lettre comme s'il s'agissait d'un numéro. Ça tombe bien, les caractères sont aussi des numéros et cela s'appelle la table ASCII (*American Standard Code for Information Interchange*). C'est initialement une table de 128 caractères, encodée avec une valeur sur 7 bits, qui ensuite est passée à 256 caractères en version étendue (et donc 8 bits). Passons les détails historiques, mais sachez que l'ASCII existe depuis les années 60 et la version étendue fut introduite sur IBM PC au début des années 80. Les 128 premiers symboles ou caractères de la table sont toujours les mêmes, mais les 128 autres peuvent varier, en fonction de la page de code utilisée. Aujourd'hui, l'ASCII existe toujours, mais le besoin en caractères « exotiques » a conduit à l'adoption d'une nouvelle norme, l'UTF-8, plus complexe, avec une longueur d'encodage variable, mais tout de même compatible avec ASCII (pour les 128 premiers caractères).

Ce qu'il y a d'absolument génial avec l'ASCII et les ordinateurs qui fondamentalement ne savent que compter et pas lire, c'est que les caractères sont littéralement des nombres. Le croquis Arduino que vous rédigez, trouve place en mémoire et est enregistré sur le disque sous la forme d'une suite de valeurs. Celles-ci ont simplement un sens pour nous, mais pour la machine, ce n'est pas du texte, mais des nombres. Un "A" par exemple **est** la valeur 65, un "W" **est** 87, etc. Même les chiffres sont des nombres ! "4" par exemple, c'est 52. Idem pour quelques symboles, "+" est 43, "?" c'est 63 et "/" c'est 47...

Il est donc possible d'utiliser ces caractères comme s'il s'agissait de nombres : "A"+1="B", parce que 65+1=66="B". Et cela signifie également qu'on peut s'en servir comme index dans un tableau.

Ainsi, à première vue, le croquis suivant est une bonne idée :

```
// notre tableau
word charmap[256];

// initialisation
void initcharmap() {
  // on met tout à 0
  for(int i = 0; i < 256; i++) {
    charmap[i]=0x0000;
  }
  // on définit quelques symboles
  charmap[(uint8_t)'A']=0x3CB8;
  charmap[(uint8_t)'B']=0xB1BC;
  charmap[(uint8_t)'C']=0x84AC;
  charmap[(uint8_t)'D']=0xAD85;
  charmap[(uint8_t)'E']=0x84BC;
  charmap[(uint8_t)'F']=0x04B8;
  charmap[(uint8_t)'G']=0xB4AC;
  charmap[(uint8_t)'H']=0x3838;
  charmap[(uint8_t)'I']=0x8585;
  charmap[(uint8_t)'J']=0xA80C;
  charmap[(uint8_t)'K']=0x4238;
  charmap[(uint8_t)'L']=0x802C;
  charmap[(uint8_t)'M']=0x2A68;
  charmap[(uint8_t)'N']=0x6868;
  charmap[(uint8_t)'O']=0xACAC;
  charmap[(uint8_t)'P']=0x1CB8;
  charmap[(uint8_t)'Q']=0xECAC;
  charmap[(uint8_t)'R']=0x5CB8;
  charmap[(uint8_t)'S']=0xB4B4;
  charmap[(uint8_t)'T']=0x0581;
  charmap[(uint8_t)'U']=0xA82C;
  charmap[(uint8_t)'V']=0x022A;
  charmap[(uint8_t)'W']=0x682A;
  charmap[(uint8_t)'X']=0x4242;
  charmap[(uint8_t)'Y']=0x0242;
  charmap[(uint8_t)'Z']=0x8686;
}

void setup() {
  shreg.attach(4, SDI_PIN, CLK_PIN,
LE_PIN, OE_PIN);
  initcharmap();
  shreg.shift(charmap[(uint8_t)'Z']);
  shreg.shift(charmap[(uint8_t)'U']);
  shreg.shift(charmap[(uint8_t)'L']);
  shreg.shift(charmap[(uint8_t)'U']);
  shreg.latch_pulse();
}

void loop() {}
```

Nous déclarons simplement une variable sous forme d'un tableau de 256 `word` (paire d'octets) et créons une fonction pour l'initialiser. Ceci commence par une boucle permettant de faire le tour des 256 emplacements puis nous utilisons des caractères comme valeur d'index pour préciser des contenus spécifiques. Notez l'utilisation de `(uint8_t)` (*unsigned int* 8 bits) avant de spécifier les caractères. Ceci permet d'explicitement indiquer le type de variable dont il s'agit, on parle de *casting*. Ici cela évite que le compilateur ne considère nos « valeurs » comme autre chose qu'un entier non signé (positif donc) de 8 bits.

Disons-le franchement, ce code est absolument horrible, ignoble et repoussant (c'était le but), même s'il s'écrit facilement. Son principal problème se résume à la toute première ligne : `word charmap[256]`. Nous venons tout juste de déclarer un tableau global de 256 fois 2 octets. En d'autres termes, cette simple ligne, en principe, se traduit par : « bravo, vous venez de consommer 25% de la SRAM d'un coup ». Comme cette variable est déclarée globale, elle trouve place dans les données statiques et cette mémoire sera définitivement consommée pour toute la durée d'exécution.

Ce code est l'exemple parfait de ce qu'il ne faut pas faire, et je ne parle même pas de la boucle...

3. PASSE 2 : LA SRAM EST PRÉCIEUSE

Une première évolution consiste à se débarrasser de la fonction d'initialisation. Si déjà nous savons que nous allons utiliser une table de 256 symboles, autant le faire une fois pour toutes et économiser du code. Il est donc parfaitement possible de directement initialiser notre `charmap[256]` avec une série de 256 valeurs 16 bits :

```
word charmap[256] = {
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x5353, 0x1111, 0x0000, 0x1010, 0x0000, 0x0000,
  0x0000, 0x0101, 0x9C9C, 0xBC84, 0x3830, 0xB4B4, 0xB4BC, 0x1692,
  0xBCBC, 0xBCB4, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0xAC9D, 0x3CB8, 0xB1BC, 0x84AC, 0xAD85, 0x84BC, 0x04B8, 0xB4AC,
  0x3838, 0x8585, 0xA80C, 0x4238, 0x802C, 0x2A68, 0x6868, 0xACAC,
  0x1CB8, 0xECAC, 0x5CB8, 0xB4B4, 0x0581, 0xA82C, 0x022A, 0x682A,
  0x4242, 0x0242, 0x8686, 0x00AC, 0x0000, 0xAC00, 0x0000, 0x8004,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x01B0, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000 };
```

Un grand nombre de symboles ne sont pas définis et ne provoqueront donc pas d'affichage. Nous pourrions nous limiter à l'ASCII « historique » de 128 caractères ou réduire encore davantage la table, mais nous nous laissons une porte ouverte pour ajouter plus que les lettres de A à Z, les chiffres de 0 à 9 et quelques symboles. 16 segments ne permettent pas de dessiner tous les caractères. Les minuscules représentent déjà un défi, tout comme les caractères accentués, mais nous pouvons tricher : remplacer les minuscules par des majuscules, les "é" par des "E" et surtout nous écarter du standard pour établir notre propre table de caractères. Réduire nos prétentions n'est pas une vraie solution, laissons cela dans l'état, 256 symboles c'est très bien.

Pourquoi tant de légèreté alors que nous parlons là de 25% de la SRAM ? Tout simplement parce que nous ne comptons justement pas laisser tout cela en SRAM, mais l'embarquer dans la mémoire flash dédiée normalement au programme lui-même. Nous en avons 16 fois plus, nos 512 octets ne représenteront alors que moins de 2% de l'espace disponible et surtout, c'est possible ! Nous avons d'ailleurs déjà franchi la première étape, car placer des variables en flash est possible, mais uniquement à la condition qu'elles soient en lecture seule. Il n'est donc pas question de créer un code ou une fonction qui va initialiser la table. Tout doit être statique ET constant (comprendre « en lecture seule »).

Pour basculer notre variable globale dans la flash, ou plus exactement, faire en sorte qu'elle ne soit pas copiée en SRAM au début de l'exécution du code dans l'Arduino, nous devons en premier lieu changer la façon de la déclarer. Ainsi,

```
word charmap[256] = {
```

devient :

```
const prog_uint16_t charmap[256] PROGMEM = {
```

`prog_uint16_t` est un type spécifique dédié à ce genre d'usage, défini dans `pgmspace.h`. Mieux vaut éviter, en effet, d'utiliser des types génériques susceptibles de provoquer des bugs difficiles à détecter et corriger. Mais l'élément important est ici le modificateur `PROGMEM` précisant un stockage en flash, autrement dit, en mémoire de programme (*PROGram MEMory*). Enfin, `const` n'est pas une absolue nécessité (quoi que cela soit indispensable avec certaines versions du compilateur), mais toujours une excellente habitude. Ce mot-clé permet de spécifier au compilateur que la variable est constante (et donc pas si variable que ça) afin qu'il puisse optimiser le code en conséquence. Le mot-clé opposé est `volatile` et permet là aussi d'aider le compilateur à laisser correctement accessible une variable qui ne semble jamais utilisée en écriture.

Mais ce n'est pas tout, nous venons de dire que ces données doivent être placées en flash, et donc dans un autre espace d'adressage qu'une variable « normale » en SRAM (architecture Harvard). Il faut donc également changer notre manière d'y accéder. Pour ce faire, nous créons une fonction qui aura pour objectif de lire une de nos données 16 bits et de l'envoyer au module, le tout, à partir d'un index (la position dans la table) passé en argument :

```
void shiftcharmap(uint8_t idx) {
  shreg.shift(pgm_read_word
(charmap+idx));
}
```

L'argument utilisé avec `shift()` est ce que retourne `pgm_read_word()` qui permet de lire une donnée en flash. Cette fonction prend en argument une adresse à laquelle se trouve le mot (`word`) que nous voulons récupérer. Mais où sont passé les `[]` ? Attention, nous venons de le dire, `pgm_read_word()` utilise une **adresse**. `charmap[0]` est la donnée se trouvant à l'endroit désigné, ou pointé, par `charmap`. Où se trouve `charmap[1]` ? Un mot plus loin et donc à `charmap+1`. Le compilateur est assez intelligent pour savoir que nous travaillons avec des mots de 16 bits (`pgm_read_word()` ce n'est pas `pgm_read_byte()` ou `pgm_read_dword()` (*double word*)) et donc que `+1` signifie ici « l'adresse du mot d'après ».

Note : si vous recherchez sur le web, il est possible que vous rencontriez par exemple les fonctions `pgm_read_word_near()` et `pgm_read_word_far()`. `pgm_read_word()` est identique à `pgm_read_word_near()` (une macro) et se différencie de `pgm_read_word_far()` par la taille de l'argument qui lui est passé, 16 bits pour *near* (proche) et 32 pour *far* (loin). Avec un ATmega328 ayant 32Ko de flash, l'adresse à laquelle se trouve notre donnée est forcément dans les 64Ko accessibles sur 16 bits. `pgm_read_word_near()` est suffisant. Mais pour une carte Arduino Mega et son ATmega1280 avec 128Ko de flash, une donnée peut se trouver plus loin que les 64Ko accessibles avec une adresse de 16 bits. Il faut donc utiliser `pgm_read_word_far()` et un argument de 32 bits pouvant adresser 512Ko de mémoire.

Notre nouvelle fonction prend en argument une variable `uint8_t`, nous n'avons plus à *caster* nos caractères et l'utilisation s'en trouve simplifiée. Pour envoyer nos quatre symboles aux TLC5926, nous pouvons alors utiliser :

```
shiftcharmap('H');
shiftcharmap('A');
shiftcharmap('C');
shiftcharmap('K');
shreg.latch_pulse();
```

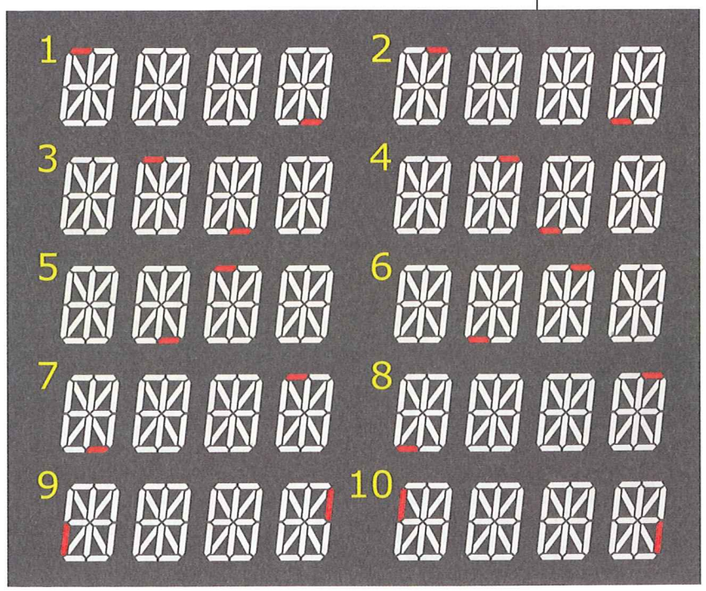
Notre programme utilise 512 octets de SRAM de moins puisque `charmap[256]` ne s'y trouve plus et quelques octets en plus en raison des fonctions supplémentaires utilisées. L'échange est donc très avantageux.

Cette technique est utilisable pour toutes les variables qui ne changent pas et je pense en premier lieu aux chaînes de caractères et en particulier aux messages envoyés sur le moniteur série. L'utilisation des variables en flash doit être un réflexe, de préférence avant même que vous ne rencontriez des problèmes. Notez également que le compilateur (GCC) qui transforme le croquis en code pour le microcontrôleur utilise déjà des techniques d'optimisation. Mais il ne fera jamais totalement le travail à votre place : économiser la mémoire est quelque chose qu'on oublie facilement lorsqu'on travaille avec un PC équipé de 4, 8 ou 16 Go, mais l'univers a une échelle tout à fait différente avec des plateformes comme Arduino.

4. PASSE 3 : FACILITONS-NOUS LA VIE

Nous avons réglé le problème de consommation excessive de SRAM et légèrement simplifié les choses pour afficher des symboles que nous avons déjà définis. Cependant, même en remplissant notre table de 256 symboles, nous arriverons toujours à une situation où nous voudrons dessiner quelque chose de spécial sur le module d'affichage.

Notre animation en 10 images ou frames donnera l'impression de deux segments en rotation horaire autour de l'afficheur. Idéal pour signifier un travail en cours ou une mise en veille...



Un bon exemple serait une petite animation où des segments s'allumeraient successivement donnant l'impression d'un motif en rotation. L'imagination est la seule limite, car avec 4 fois 16 segments et un nombre de motifs uniquement limité par la flash disponible, il y a largement de quoi faire. Et si ce n'est pas suffisant, on peut toujours envisager l'utilisation d'une carte SD, une EEPROM externe ou une flash SPI.

Notre animation consistera donc en 10 « écrans » qui s'afficheront successivement à 50 ms d'intervalle. Mais il ne s'agit pas forcément de motifs stockés dans notre table de caractères. Pour nous simplifier la vie, nous avons donc besoin d'une méthode nous permettant de désigner les segments simplement et directement dans notre code sans passer par la case tableur/papier/crayon.

La solution consiste à désigner les segments directement par leur lettre et de faire en sorte que la valeur correspondante soit utilisée, le tout sans consommer de mémoire. En d'autres termes, nous voulons que, par exemple, `SEG_A` soit utilisé comme si nous écrivions `0x0080`. Ceci n'est pas du domaine du compilateur, mais du préprocesseur, nous devons utiliser des macros :

```
#define SEG_A 0x0080
#define SEG_B 0x0400
#define SEG_C 0x0020
#define SEG_D 0x0040
#define SEG_E 0x0100
#define SEG_F 0x0200
#define SEG_G 0x0800
#define SEG_H 0x0010
#define SEG_I 0x1000
#define SEG_J 0x0008
#define SEG_K 0x0002
#define SEG_L 0x0001
#define SEG_M 0x4000
#define SEG_N 0x2000
#define SEG_O 0x0004
#define SEG_P 0x8000
#define SEG_Z 0x0000
```

`#define` est une directive permettant de définir une macro. Ainsi `SEG_A` est le nom de la macro et `0x0080` la valeur substituée. Chaque fois que l'on utilisera `SEG_A`, le préprocesseur le remplacera par `0x0080` avant que le compilateur ne fasse son travail. Notez bien que ceci n'est pas une variable et n'a pas d'existence passé l'étape du préprocesseur. C'est un simple alias.

Comme `SEG_A` et les autres définitions sont littéralement équivalentes aux valeurs hexadécimales, elles possèdent les mêmes propriétés. Nous pouvons donc les additionner et les combiner avec des opérateurs logiques. `SEG_A|SEG_B` est donc `0x0080` *OU* `0x0400` (ou `0x0080 + 0x0400` dans ce cas) et vaut donc `0x0480`, soit une activation des segments A et B.

Précisons au passage que les macros du langage C/C++ permettent de faire bien plus que cela, d'ajouter des conditions, procéder à des tests, etc.

Notez que nous avons défini un segment Z, `SEG_Z`, qui n'existe pas, mais représente « aucun segment ». Ceci ne nous coûte rien et améliore l'uniformité lors des initialisations.

Dès lors, nous pouvons composer notre animation sous la forme d'un nouveau tableau (stocké en flash bien sûr) :

```
const prog_uint16_t anim[40] PROGMEM = {
SEG_A, SEG_Z, SEG_Z, SEG_P,   // frame 1
SEG_B, SEG_Z, SEG_Z, SEG_O,   // frame 2
SEG_Z, SEG_A, SEG_P, SEG_Z,   // frame 3
SEG_Z, SEG_B, SEG_O, SEG_Z,   // frame 4
SEG_Z, SEG_P, SEG_A, SEG_Z,   // frame 5
SEG_Z, SEG_O, SEG_B, SEG_Z,   // frame 6
SEG_P, SEG_Z, SEG_Z, SEG_A,   // frame 7
SEG_O, SEG_Z, SEG_Z, SEG_B,   // frame 8
SEG_J, SEG_Z, SEG_Z, SEG_G,   // frame 9
SEG_C, SEG_Z, SEG_Z, SEG_N    // frame 10
};
```

C'est bien plus intuitif que des valeurs hexadécimales. La première image ou *frame* est donc « segment A, rien, rien, segment P » respectivement sur les 4 afficheurs leds. Il ne nous reste plus, ensuite, qu'à utiliser cette table dans une boucle, quelque part dans notre croquis :

```
for(int i=0; i<40; i=i+4) {
  shreg.shift(pgm_read_word(anim+i));
  shreg.shift(pgm_read_word(anim+i+1));
  shreg.shift(pgm_read_word(anim+i+2));
  shreg.shift(pgm_read_word(anim+i+3));
  shreg.latch_pulse();
  delay(50);
}
```


À ce stade, nous pouvons même revoir la syntaxe que nous avons utilisée pour créer notre table de caractères initiale de manière à la rendre plus lisible. C'est un gros travail, mais cela ne sera fait qu'une seule fois et surtout, ça ne prendra de place que dans le croquis et non dans le code effectivement utilisé et compilé.

5. PASSE 4 : LA VÔTRE !

Nous n'en n'avons pas fini. En réalité, nous n'en aurons certainement jamais fini. Il reste de la place pour l'optimisation un peu partout. Mais l'idée n'est pas de passer une éternité pour rendre le code toujours plus dense et plus facile à utiliser. Il faut trouver l'équilibre entre efficacité, lisibilité et temps investi.

Si votre code n'est pas intelligible facilement, mais hyper optimisé personne ne le fera évoluer, ce n'est pas rentable.

Si vous avez passé des nuits pour obtenir un code « aux petits oignons », mais que vous ne vous en servez qu'une fois ou deux, ce n'est pas rentable.

Si cela ne vous a pris que quelques minutes, mais que votre code pourrait être deux fois plus petit, ce n'est pas rentable.

Si votre croquis est très facile à comprendre, mais qu'il est d'une lenteur affligeante, ce n'est pas rentable.

Bien entendu, cela dépendra également de l'objectif visé.

Souhaitez-vous diffuser ce code ? Doit-il absolument tenir dans le plus petit espace possible ? Est-ce une rustine ou un hack rapide pour une réalisation ponctuelle et sans avenir ? Ce code est-il juste une excuse pour passer un bon moment à programmer ?

Il y a bien des pistes à explorer. Voici quelques exemples :

- Notre dernière boucle `for` pour l'animation utilise une condition de fin dépendant de la taille de la table. Nous pouvons obtenir cette taille avec `sizeof(anim)/sizeof(prog_uint16_t)` plutôt que de la spécifier « en dur ». Il est aussi possible de créer une fonction plus spécialisée dans l'animation.
- Notre table de caractères est indexée par les valeurs ASCII, mais cela reste une table. Nous pouvons écrire une fonction qui prend en argument une chaîne de caractères, extrait chacun d'eux, et les envoyer à l'afficheur : `envoiled("TITI")`.
- Si nous comptons utiliser plusieurs animations, il est possible de stocker toutes les *frames* dans une seule variable et d'utiliser la position de début et de fin plutôt que d'avoir une variable par animation.
- Nous pouvons écrire une fonction plus générique d'affichage prenant en argument le nombre de TLC5926 utilisés et vérifiant les données qu'on lui passe en argument (pas assez ou trop de symboles) pour savoir quand utiliser `latch_pulse()`.
- Il est possible d'envisager la génération d'animations non pas sous une forme préenregistrée, mais calculée en temps réel.
- Si nous écrivons une fonction qui accepte une chaîne de caractères, nous pouvons en accepter plus que ce que peut afficher un module, découper la chaîne pour l'envoyer en plusieurs fois et la faire défiler sur l'afficheur.
- Nous pouvons animer l'affichage de nos symboles ou caractères en activant un segment après l'autre de façon à créer un petit effet. Il en va de même pour « désafficher » notre motif et ainsi créer un effet de transition.

Comme vous pouvez le voir, il y a de quoi faire et un petit croquis affichant juste « TOTO » peut rapidement devenir quelque chose de plus massif, complet et réutilisable. Mieux encore, vous pouvez en faire une bibliothèque et ainsi faire profiter d'autres utilisateurs du temps et des efforts que vous-même aurez investis dans le projet... DB

CRÉEZ VOTRE BIBLIOTHÈQUE ARDUINO POUR NOTRE AFFICHEUR

Denis Bodor

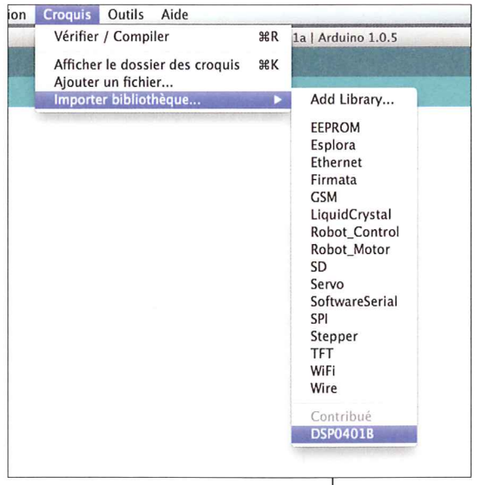
Nous avons fait du chemin, mais nous n'en avons pas fini. Bien que nous disposions des connaissances et du code, il reste un pas à franchir et ce, que vous comptiez ou non diffuser le résultat de votre travail. L'idée est de rendre plus génériques encore nos manipulations et ne laisser apparaître que la partie utilisable de notre travail dans un croquis. En d'autres termes, nous pouvons créer une bibliothèque !

Les bibliothèques, nous les utilisons régulièrement. Je ne parle bien entendu pas du lieu, mais de code. Pour les plus jeunes lecteurs qui ne connaissent pas le lieu, c'est comme Wikipédia, mais tout est imprimé et ça fonctionne sans courant et sans net. Les bibliothèques nous les importons, nous les utilisons très facilement, mais qu'en est-il au-delà de l'aspect un peu magique qui consiste à simplement nous pourvoir en fonctions et en objets apportant de nouvelles fonctionnalités ?

Pour explorer au mieux ce domaine, quoi de mieux que de réaliser sa propre bibliothèque. Ça tombe bien, nous avons exactement ce qu'il nous faut sous la main. En effet, nous venons de factoriser notre code de manière à très simplement envoyer des caractères et autres symboles à notre ou nos afficheurs 4×16 segments à base de TLC5926. Stocker cette « intelligence » dans une bibliothèque pour n'avoir plus qu'à utiliser un `#include` et quelques lignes de code sera donc notre objectif.

1. LES BIBLIOTHÈQUES

L'environnement Arduino supporte la notion de bibliothèques. Elle consiste à avoir à sa disposition un ensemble de fonctions, d'objets et de méthodes groupés en paquets bien séparés. Un certain nombre de ces bibliothèques sont incluses directement dans l'environnement Arduino. Ainsi, pour utiliser le moniteur et envoyer des messages, il n'est pas nécessaire de préciser qu'on souhaite utiliser une bibliothèque, car celle-ci (`HardwareSerial`) est intégrée d'office dans vos croquis.



D'autres bibliothèques en revanche ne font pas partie du lot. Nous pouvons prendre, par exemple, `Adafruit_BMP085` prenant en charge le protocole permettant la gestion et la consultation du capteur barométrique BMP085. Pour faire fonctionner ce composant, il faut bien entendu le connecter à la carte Arduino, mais aussi, et surtout installer la bibliothèque. Celle-ci prend le plus souvent la forme d'un fichier Zip contenant un répertoire du nom de la bibliothèque, lui-même contenant :

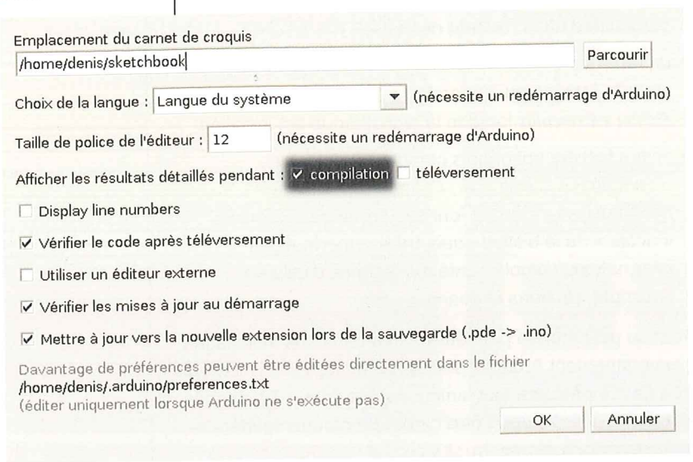
- un ou plusieurs fichiers `.cpp` qui est le code de la bibliothèque écrit en langage C++,
- un ou plusieurs fichiers `.h` pour *header* ou fichier d'en-tête qui contient les différentes déclarations pour la bibliothèque,
- un fichier `keywords.txt` regroupant avec une syntaxe spécifique les termes qui doivent prendre une couleur particulière dans l'éditeur de code,
- un répertoire `examples` contenant un ou plusieurs croquis Arduino qui apparaîtront dans le menu ***Fichier > Exemples*** lorsque la bibliothèque est installée,
- divers fichiers optionnels comme `README.txt` ou encore `License.txt`, `ChangeLog.txt` ou `Buildinfo.txt` qui ne font pas partie des éléments « actifs » de la bibliothèque, mais apportent des informations complémentaires (licence d'utilisation, descriptif, versions et changements).

Tout ce petit monde peut être ajouté dans une installation de l'environnement Arduino, soit via un menu dans l'interface, soit en décompressant tout simplement le fichier Zip dans le répertoire `libraries` de votre carnet de croquis (généralement le répertoire `sketchbook`).

Vous l'avez compris, une bibliothèque c'est surtout un fichier `.cpp` et un `.h`. Ce dernier, un fichier d'en-tête, en C comme en C++ découle d'un besoin évident : en cas d'utilisation de plusieurs fichiers sources (`.c` ou `.cpp`) il fallait trouver un moyen d'éviter de systématiquement débuter chaque fichier avec la liste des déclarations présentes dans les autres fichiers. Aussi, ces déclarations « universelles » et communes trouvent place dans un `.h` (ou plusieurs) qui est alors inclus dans les `.c` ou `.cpp`.

Mais vous n'utilisez qu'un seul fichier sous la forme d'un croquis avec Arduino, n'est-ce pas ? En fait, pas du tout. Lorsque vous vérifiez votre croquis, tout un mécanisme se met en marche et un grand nombre de fichiers sont en réalité traités (compilés). Lorsque vous utilisez le menu ***Croquis > Importer une bibliothèque*** dans l'IDE Arduino et qu'une ou plusieurs lignes `#include` s'ajoutent à votre croquis, nous avons effectivement une situation identique à celle que nous venons de décrire : plein de fichiers sources (`.c`, `.cpp`, `.ino`) et une inclusion des déclarations sous forme de fichiers d'en-tête. Lors du processus de compilation, le fichier `.cpp` de la bibliothèque sera, comme tous les autres, transformé en objet (`.o`) et l'ensemble sera réuni par l'éditeur de liens pour produire votre programme binaire dans un format spécial (ELF pour *Executable and Linkable Format*). Enfin, de ce fichier ELF sera extrait le programme « pur » à destination de la mémoire flash du microcontrôleur de l'Arduino (avec `objcopy`).

Dans les préférences de l'environnement de développement Arduino, vous avez la possibilité d'activer l'affichage des détails du processus de compilation et donc de voir le compilateur et l'éditeur de lien à l'œuvre. Ceci permet également d'afficher les avertissements et non simplement les erreurs.



2. AVANT DE COMMENCER

Les bibliothèques Arduino sont écrites en C++ qui est un langage orienté objet (POO) et procédural (comme le C). On dit que le langage est multiparadigme. Vous pouvez donc, en principe, créer une bibliothèque exactement de la même manière que vous programmez vos croquis. Cependant, il est d'usage d'utiliser le paradigme objet pour les bibliothèques Arduino. Ceci, tout simplement, parce que l'ensemble des utilisateurs a l'habitude de voir et d'utiliser une certaine syntaxe dans leur croquis.

Le C++ utilise ainsi des classes. Une classe est un ensemble de propriétés communes à une classe d'objets. Certaines propriétés représentent l'état de l'objet, ce sont ses attributs. D'autres propriétés représentent le comportement d'un objet : ses méthodes.

Nous partons là dans des considérations très théoriques et devons

impérativement ancrer ces notions par la pratique. Prenons l'exemple de la bibliothèque `LiquidCrystal` permettant de piloter les afficheurs LCD type HD44780. Pour l'utiliser, on commence généralement par quelque chose comme ceci :

```
LiquidCrystal lcd(2,3,4,5,6,7)
```

Nous venons d'instancier un objet `lcd` de la classe `LiquidCrystal` en précisant la liste des broches utilisées en argument. `LiquidCrystal` n'est pas « quelque chose », c'est une classe. Il vaut voir cela comme un moule qui nous permet de fabriquer des objets. Ici `lcd` est un objet moulé d'après `LiquidCrystal`. Rien ne nous empêcherait alors d'utiliser encore une fois cette syntaxe pour créer `totoLCD`, une autre instance de `LiquidCrystal` utilisant d'autres broches de l'Arduino.

Plus loin dans le code, nous utilisons l'afficheur :

```
lcd.print("Coucou");
```

Là, on dit qu'on utilise la méthode `print()` de l'objet `lcd` qui est une instance de `LiquidCrystal`. Une méthode est une fonction membre d'une classe. Ici, c'est une méthode d'instance. Vous êtes perdu ? Traduction : notre `lcd` a été moulé à partir de `LiquidCrystal`, est donc équipé d'une fonction qui était prévue dans le moule. Ceci signifie que `LiquidCrystal`, quelque part, défini `print()` et tous les objets qui sont moulés possèdent cette fonction. `lcd.setCursor(0,1)` est aussi l'utilisation d'une méthode, `setCursor()`. Elle permet de placer le curseur sur une ligne et une colonne spécifique de l'afficheur LCD. Créer une classe consiste donc, entre autres, à créer une série de méthodes qui pourront être utilisées par les objets de cette classe.

`setCursor()` et `print()` sont des méthodes dites publiques, car elles sont utilisables par les clients de la classe comme votre croquis. D'autres méthodes ne sont pas ainsi publiquement utilisables. Ce sont des fonctions qui ne peuvent être appelées que par les membres de la classe elle-même. En d'autres termes, ce sont des fonctions internes que vous ne voulez pas rendre utilisables de l'extérieur (nous passerons ici sur la notion de méthodes protégées et de classes dérivées, pour rester concis).

Enfin, nous avons aussi les attributs qui sont des variables qui peuvent être également soit publiques, soit privées. Les attributs publics sont des variables accessibles par un croquis et, les attributs privés sont purement internes et donc invisibles.

Après ce cours express et simpliste sur la programmation orientée objet et le langage C++, il est temps de nous pencher sur le travail que nous devons réaliser pour créer une bibliothèque. Mais avant cela, il est impératif de savoir comment doit fonctionner notre classe et quels méthodes et attributs doivent être publics et privés. Nous avons besoin de définir un cahier des charges clair et précis.

Ceci revient à répondre à la question simple : que doit proposer notre bibliothèque à un utilisateur ? Nous allons ici tenter de garder les choses le plus simple possible pour ne pas faire de cet article un hors-série complet.

Nous devons également prendre en compte un problème courant avec l'environnement Arduino qui est la difficulté d'utiliser une bibliothèque depuis une autre bibliothèque. Si nous ajoutons à cela qu'une bibliothèque est censée fournir des fonctionnalités faciles à comprendre à l'utilisateur et le fait que nous n'utilisons qu'une petite partie des fonctionnalités de la bibliothèque TLC5926, nous devons prendre une décision radicale : nous allons nous passer de la bibliothèque TLC5926 et intégrer dans la nôtre de quoi communiquer avec les registres à décalage.

Au final, nous nous arrêterons donc sur les fonctionnalités suivantes :

- l'utilisateur peut connecter l'afficheur comme il le souhaite et il doit pouvoir préciser la liste des broches de son choix,

- on doit pouvoir utiliser un nombre quelconque d'afficheurs chaînés entre eux,
- nous réutiliserons le principe de contrôle de luminosité avec la PWM,
- il faut une fonction d'effacement de l'afficheur,
- il faut qu'on puisse envoyer un texte à l'afficheur tout comme on le fait avec `LiquidCrystal`,
- nous sommes des gens cool et les gens cool proposent toujours une fonctionnalité un peu bling-bling. Pour nous, ce sera le fait de pouvoir faire défiler un texte sur l'afficheur, à la vitesse désirée par l'utilisateur.

Cela semble un sacré travail, mais en réalité notre bibliothèque sera relativement modeste. En route, on retrousse ses manches, on finit son mug de café, et on y va !

3. DÉCLARATION, EN-TÊTE ET CLASSE

En C++ et avec Arduino, créer une bibliothèque se fait en définissant la classe dans le fichier d'en-tête et en l'implémentant dans le fichier `.cpp`. Nous allons tout d'abord donner un nom à notre classe et notre bibliothèque : `DSP0401B` (c'est le nom du modèle d'afficheur). De ce fait, notre bibliothèque sera placée dans un répertoire `DSP0401B`, contenant les fichiers `DSP0401B.h` et `DSP0401B.cpp` et définissant la classe `DSP0401B`.

Notre fichier d'en-tête doit contenir la description de la classe, de ses méthodes et de ses attributs. Voyez cela comme l'étiquette avec la composition, comme sur une conserve (les attributs privés ce sont les additifs E-machin). La syntaxe sera la suivante :

```
class nom_de_classe {
   public:
      // méthodes et attributs publics
   private:
      // méthodes et attributs privés
};
```

Rien de bien complexe ici, mais ce n'est pas suffisant. Nous allons compléter cela et le transformer en :

```
#ifndef DSP0401B_h
#define DSP0401B_h

#include "Arduino.h"

class DSP0401B {
   public:
      DSP0401B();
      // méthodes et attributs
      // publics
   private:
      // méthodes et attributs
      // privés
};

#endif
```

Nous utilisons ici une macro un peu plus complexe que celle que nous avons utilisée dans l'article précédent :

```
// si DSP0401B_h est pas définit
#ifndef DSP0401B_h
// on définit DSP0401B_h
#define DSP0401B_h

// on déclare notre classe

// fin de la condition
#endif
```

Ce qui peut se traduire en : « si je n'ai pas encore fait tout cela, alors dire que je l'ai fait et le faire ». Ceci permet d'éviter les problèmes et erreurs si l'utilisateur fait apparaître `#include <DSP0401B.h>` plusieurs fois dans son croquis ou si le fichier est inclus depuis plusieurs endroits. Il n'est pas possible de redéclarer une classe et cette technique permet de nous assurer que ce ne sera fait qu'une fois.

La ligne `#include "Arduino.h"` est également importante. Dans un croquis Arduino, cette ligne est ajoutée automatiquement avant la compilation et permet de fournir à l'utilisateur toutes les classes et fonctions propres à Arduino. Lorsqu'on écrit une bibliothèque, ces fonctions ne sont pas incluses d'office et si nous voulons nous en servir, il faut clairement le préciser.

Vous l'avez sans doute remarqué, une ligne s'est ajoutée dans les déclarations publiques. Cette méthode portant le même nom que la classe et n'ayant pas de type est le constructeur. Toute classe a une fonction spéciale comme celle-ci qui est utilisée pour créer une instance de la classe. Dans `LiquidCrystal`, il y a aussi une méthode nommée `LiquidCrystal`. C'est celle qui a été appelée lorsque nous avons instancié l'objet `lcd` dans notre exemple.

Nous pouvons maintenant nous occuper des méthodes et attributs publics et privés. Les attributs publics comme pour bon nombre de bibliothèques, nous n'en avons pas. Il n'y a pas de variable résidant dans une instance de notre classe qu'un utilisateur puisse lire ou écrire. Pour les méthodes ce n'est pas la même histoire, car c'est bien là le but d'une bibliothèque, proposer des actions à accomplir. Fort heureusement, nous avons prévu un cahier des charges et cela sera donc assez évident :

```
class DSP0401B {
  public:
    // le constructeur
    DSP0401B();
    // initialisation et configuration
    void begin(unsigned int num_disp, unsigned int sdi_pin,
        unsigned int clk_pin, unsigned int le_pin,
        unsigned int oe_pin);
    // réglage de l'intensité
    void brightness(unsigned char brightlvl);
    // effacement de l'affichage
    void clear();
    // envoi d'un texte
    void sendtext(String text);
    // défilement d'un texte
    void slidetext(String text, unsigned long pause);
```

La définition d'une classe est la description de ce qu'elle contient. Nous avons donc là, non pas les fonctions, mais uniquement les prototypes des fonctions (la liste de ce qui est au menu en somme). Nous avons ainsi les méthodes :

- `begin()` qui est typique des bibliothèques Arduino et permet d'initialiser une instance en précisant, comme ici, la liste des broches qui seront utilisées. Nous avons en argument un nombre d'afficheurs (`num_disp`), la broche pour les données (`sdi_pin`), celle pour l'horloge (`clk_pin`), celle pour le *latch* (`le_pin`) et enfin celle de l'activation des leds (`oe_pin`),
- `brightness()` permet de spécifier l'intensité qui sera utilisée avec `analogWrite()` plus tard,
- `clear()` permet d'effacer l'affichage en envoyant le bon nombre de `0x0000`,
- `sendtext()` est la méthode la plus importante (pour l'utilisateur) puisqu'elle permet de spécifier un texte, `"comme ceci"`, et de le faire s'afficher en caractères de leds,
- `slidetext()` est similaire à `sendtext()`, mais provoquera un petit effet sur l'afficheur.

Plusieurs choses demandent des explications ou des remarques. En premier lieu, ce code est bourré d'anglicismes, car n'en déplaise à l'Académie française (et surtout à l'Office québécois de la langue française (pas québécoise)), le monde de l'informatique est anglophone. Nous avons peut-être l'intention de diffuser cette bibliothèque et non de la garder égoïstement pour nous. Il faut donc éviter de faire perdre du temps aux utilisateurs et surtout aux développeurs

qui pourraient faire évoluer notre création. S'ils doivent chercher ce que `broche_sdi` et `effacement()` veulent dire, ils risquent de simplement passer leur chemin. Il en va de même pour les commentaires qui sont ici en français pour le magazine, mais qui seraient en anglais dans un code diffusé sur le net.

`sendtext()` et `slidetext()` prennent toutes deux un argument de type `String` comme « chaîne » de caractères (non, pas comme string). La notion de chaînes de caractères n'existe pas en C/C++, il s'agit simplement de tableaux de caractères terminant par `0x00` et de pointeurs. La classe `String` fait partie de l'environnement Arduino et permet de simplifier la manipulation des chaînes de caractères. Généralement, tableaux et pointeurs (et l'allocation mémoire qui va avec) sont un écueil important lorsqu'on découvre le C/C++. Pour faciliter l'apprentissage, Arduino choisit de masquer ces difficultés en utilisant le plus possible la classe `String` et ses méthodes. Ceci doit également être important pour vous, car comme le précise le guide pour l'écriture de bibliothèques Arduino : « *soyez sympa avec l'utilisateur* » et « *ne partez pas du principe que la notion de pointeur est connue* ». De plus, nous le verrons plus loin, à l'implémentation, cette classe s'avère très pratique dans notre cas (comme dans beaucoup d'autres).

Et quid de la partie privée ? Là, les choses se compliquent. Non en termes techniques pour l'article, mais en ce qui concerne les choix que vous faites. À moins d'avoir une très bonne vision de l'implémentation de votre classe, il est difficile de lister, du premier coup, ce dont vous aurez besoin. Généralement, dans ce genre de situation, on édite le fichier d'en-tête et le code C++ en même temps, en gardant le tout synchronisé. Le compilateur ne manquera pas de vous rappeler à l'ordre si les choses ne sont pas cohérentes.

Ici, la majeure partie du travail a été fait avant même d'entamer la rédaction de l'article et nous avons donc :

```
private:
  // envoi d'un symbole à partir de l'index
  void shiftcharmap(uint8_t idx);
  // impulsion sur LAT
  void latch_pulse();
  // envoi d'une valeur 16 bits
  void shift(unsigned int pattern);
  // les broches
  unsigned int SDI;
  unsigned int CLK;
  unsigned int LE;
  unsigned int OE;
  // nombre d'afficheur
  unsigned int NUM;
```

Tout d'abord, nous avons là un exemple parfait de l'utilité des méthodes privées. `shift()` par exemple n'est pas utilisé depuis le croquis, mais sera appelée par `shiftcharmap()` comme dans les articles précédents, et elle non plus n'a pas de raison d'être visible de l'extérieur de la classe. Nous avons ensuite les attributs privés `SDI`, `CLK`, `LE`, `OE` et `NUM` qui pourtant semblent déjà être des informations que nous avons gérées avec des variables et en particulier celles de la méthode `begin()`.

Mais les arguments de cette méthode ne sont pas disponibles au travers de toute la classe, mais uniquement dans le corps de la fonction `begin()` que nous n'avons pas encore écrite. D'autres méthodes, comme `latch_pulse()` doivent faire usage de ces informations puisqu'il s'agit, entre autres, des numéros de broches à contrôler. Nous avons donc besoin de variables propres à la classe : des attributs privés.

Nous en avons fini, en principe avec la définition de notre classe, mais nous n'avons pas couvert tout ce que peut gérer notre fichier d'en-tête. Nous pouvons tout d'abord y ajouter les macros de l'article précédent permettant de désigner les segments des afficheurs :


```
#define SEG_Z 0x0000
#define SEG_A 0x0080
#define SEG_B 0x0400
#define SEG_C 0x0020
#define SEG_D 0x0040
#define SEG_E 0x0100
#define SEG_F 0x0200
#define SEG_G 0x0800
#define SEG_H 0x0010
#define SEG_I 0x0100
#define SEG_J 0x0008
#define SEG_K 0x0002
#define SEG_L 0x0001
#define SEG_M 0x4000
#define SEG_N 0x2000
#define SEG_O 0x0004
#define SEG_P 0x8000
```

Et nous pouvons également ajouter notre fameuse table de symboles utilisée par `shiftcharmap()`. Pour cela, nous allons utiliser un autre fichier d'en-tête que nous appellerons `charmap.h` et que nous inclurons, juste après `Arduino.h` dans notre `DSP0401B.h`. Ceci nous permettra et permettra à un autre développeur de changer ce fichier sans tripoter le reste de la bibliothèque. Mieux encore, il nous sera possible plus tard de faire évoluer notre bibliothèque de manière à faciliter l'intégration de tables de symboles personnalisées. Le principe est clair : si c'est simple, que ça apporte quelque chose tout en ne coûtant rien, autant préparer le futur et le faire.

Notre classe est maintenant définie, nous pouvons nous intéresser à son implémentation.

4. IMPLÉMENTATION ET MÉTHODES

Notre travail est simple : il faut écrire les méthodes que nous venons de décrire. Pour cela, nous allons rédiger `DSP0401B.cpp` et débuter son contenu en « rappelant » les règles que nous avons établies :

```
#include "DSP0401B.h"
```

Continuons en écrivant cette méthode spéciale qu'est le constructeur de la classe :

```
DSP0401B::DSP0401B() {
}
```

Le nom de cette fonction est particulière et est typique du C++. La première occurrence de `DSP0401B` est le nom de la classe, la seconde la méthode. Nous écrivons donc que ceci est la méthode `DSP0401B` de la classe `DSP0401B`. Cette syntaxe sera utilisée pour toutes les autres méthodes de la classe. Notre constructeur ne fait rien à part instancier implicitement un objet lorsqu'il sera appelé. Le travail effectif a été transféré vers la méthode `begin()` que nous nous empressons d'écrire :

```
void DSP0401B::begin(unsigned int num_disp,
  unsigned int sdi_pin, unsigned int clk_pin,
  unsigned int le_pin, unsigned int oe_pin) {

  // initialisation des attributs privés
  SDI = sdi_pin;
  CLK = clk_pin;
  LE = le_pin;
  OE = oe_pin;
  NUM = num_disp;

  // CLK en sortie
  pinMode(CLK, OUTPUT);
  digitalWrite(CLK, LOW);

  // SDI en sortie
  pinMode(SDI, OUTPUT);

  // LE en sortie
  pinMode(LE, OUTPUT);
  digitalWrite(LE, LOW);

  // OE en sortie
  pinMode(OE, OUTPUT);
  // 100% on = leds off
  analogWrite(OE,255);

  // méthode pour effacer
  clear();
}
```

La majorité des fonctions utilisées sont classiques. Remarquez cependant que `analogWrite()` est utilisé quelle que soit la broche désignée. On part du principe qu'elle supporte la

PWM (sortie « analogique ») et il y a là une amélioration à apporter plus tard grâce à `digitalPinToTimer()` (« j'ai écrit une fonction véritablement merveilleuse que cet article et trop long pour contenir »).

La partie importante de notre méthode réside dans l'initialisation des attributs. `sdi_pin` par exemple est une variable locale à `begin()`, mais `SDI` est globale à la classe tout entière. Le fait de copier la valeur de la première dans la seconde rend cette information disponible pour toutes les méthodes de la classe.

Enfin, nous utilisons `clear()` pour effacer automatiquement l'afficheur lors de l'utilisation de la méthode `begin()`. Notez que le nom de la méthode `clear()` n'est pas précédé de `DSP0401B::`, car nous sommes présentement dans l'implémentation de la classe en question.

Ce qui nous conduit naturellement à son implémentation :

```
void DSP0401B::clear() {
  for(unsigned int i=0; i<NUM*4; i++) {
    shiftcharmap(' ');
  }
  latch_pulse();
}
```

C'est simple, nous envoyons à l'afficheur autant de fois le symbole du caractère « espace » (vide donc) qu'il y a de modules d'affichage chaînés, fois le nombre d'afficheurs 16 segments par module. Cette information est disponible, car précisée lors de l'utilisation de la méthode `begin()`. Après cela, nous activons brièvement la broche LAT pour valider les données.

Nous nous dirigeons en toute logique vers l'implémentation des deux autres méthodes :

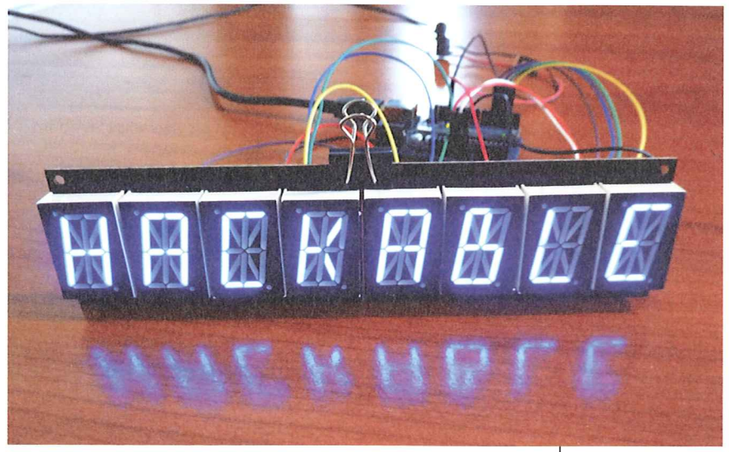
```
void DSP0401B::shiftcharmap(uint8_t idx) {
  shift(pgm_read_word(charmap+idx));
}
```

et

```
void DSP0401B::latch_pulse() {
  delayMicroseconds(30);
  digitalWrite(LE, HIGH);
  delayMicroseconds(30);
  digitalWrite(LE, LOW);
}
```

La première est en tous points similaire à la fonction équivalente du croquis de l'article précédent et la seconde se limite, avec de courts délais, à activer puis désactiver la broche LE. La méthode `shift()` utilisée par `shiftcharmap()` est nouvelle, car précédemment mise à disposition par la bibliothèque TLC5926. C'est d'ailleurs là que nous avons cherché l'inspiration pour l'implémenter :

```
void DSP0401B::shift(unsigned int pattern) {
  shiftOut(SDI, CLK, MSBFIRST, pattern >> 8);
  shiftOut(SDI, CLK, MSBFIRST, lowByte(pattern));
}
```

Cette méthode utilise la fonction standard Arduino `shiftOut()` permettant d'envoyer sur la broche désignée en premier argument le motif 8 bits précisé en dernier argument, en utilisant la broche en second argument comme signal d'horloge. Le troisième argument précise l'ordre dans lequel envoyer les bits. Ici c'est `MSBFIRST` soit « *d'abord le bit de poids fort* ». Dans une valeur 8 bits comme `0b10000111`, le bit de poids fort est celui le plus à gauche, car c'est celui qui a la valeur la plus importante (2^7 soit 128). `MSBFIRST` signifie donc « *commencer par la gauche* ».

Nous avons 16 bits à envoyer, mais `shiftOut()` ne travaille qu'avec des valeurs 8 bits. Nous devons envoyer notre valeur en deux fois. La première occurrence de `shiftOut()` décale donc notre valeur de 8 positions sur la droite (`0b1111110000000000` devient `0b0000000011111100`, mais est tronqué à 8 bits et est donc `0b11111100`).

La seconde occurrence utilise `lowByte()` qui est une fonction qui retourne l'octet de poids le plus faible d'une variable, soit les 8 bits les plus à droite. C'est donc la seconde partie de notre valeur 16 bits. Oui, il existe une fonction `highByte()` que nous aurions pu utiliser et oui, c'est ainsi que la méthode est implémentée dans la bibliothèque TLC5926, mais j'ai trouvé que c'était là une belle occasion de montrer qu'il y a toujours plus d'une façon de faire.

Avant de passer aux deux méthodes les plus intéressantes, voici `brightness()` :

```
void DSP0401B::brightness(unsigned char brightlvl) {
  analogWrite(OE, ~(brightlvl));
}
```

Rien de très surprenant ici, si ce n'est qu'il ne faut pas oublier que OE est un signal actif à l'état bas et de ce fait que le rapport cyclique de la PWM doit être inversé pour fonctionner de manière intuitive.

Nous pouvons nous pencher sur le cœur du problème avec la méthode d'envoi de texte :

```
void DSP0401B::sendtext(String text) {
  for(unsigned int i=0; i<NUM*4; i++) {
    if(i<text.length()) {
      shiftcharmap(text.charAt(i));
    } else {
      shiftcharmap(' ');
    }
  }
  latch_pulse();
}
```

Nous recevons un objet de type `String` et cette classe offre quelques avantages sous la forme de méthodes comme `length()` retournant la longueur de la chaîne de caractères (là il

aurait été très, très, pénible de devoir travailler avec un pointeur sur un tableau de caractères). Nous bouclons simplement sur la quantité d'afficheurs 16 segments disponibles. Dans la boucle, nous testons si notre position correspond toujours à une position existante dans le texte. Si c'est le cas, nous utilisons la méthode `charAt()` qui prend en argument une position et retourne le caractère correspondant, et nous envoyons le résultat comme un index dans notre table de symboles à la méthode `shiftcharmap()`. Si nous ne sommes plus dans la chaîne, nous complétons avec des symboles vides (des espaces). Notez que la méthode `charAt()` peut être remplacée par une syntaxe peut-être plus intelligible par certains : `text[i]` soit le caractère en position `i` dans l'objet `text`.

Il découle du fonctionnement de cette méthode plusieurs choses importantes :

- un texte plus petit que l'affichage sera aligné à gauche,
- un texte plus grand sera tronqué et l'affichage se limitera aux `NUM*4` premiers caractères.

Ceci peut être gênant, mais nous avons notre méthode bling-bling pour compenser. Celle-ci n'est pas très différente :

```
void DSP0401B::slidetext(String text,
     unsigned long pause) {
  clear();
  for(unsigned int i=0; i<text.length(); i++) {
    shiftcharmap(text.charAt(i));
    latch_pulse();
    delay(pause);
  }
  for(unsigned int i=0; i<NUM*4; i++) {
    shiftcharmap(' ');
    latch_pulse();
    delay(pause);
  }
}
```

Les méthodes utilisées sont plus ou moins les mêmes, mais nous avons un argument en plus. Cette fois, nous effaçons d'abord l'affichage puis nous bouclons sur la taille du texte reçu. Caractère par caractère, nous « poussons » la valeur du symbole correspondant et surtout, nous « latchons » à chaque fois. Enfin, nous complétons l'opération en envoyant et « latchant » `NUM*4` espaces. Le résultat est sympathique, le texte arrive petit à petit de droite et passe vers la gauche avant de disparaître. Le tout à la vitesse découlant de la longueur de pause en millisecondes entre chaque envoi de caractères. Longueur choisie par l'utilisateur en invoquant la méthode.

Et c'est fini ! Nous avons intégralement implémenté toutes les méthodes et notre classe est complète.

5. UTILISONS NOTRE BIBLIOTHÈQUE

Complétons rapidement le répertoire avec un fichier `keywords.txt` histoire d'ajouter un peu de couleurs :

```
DSP0401B        KEYWORD1
begin           KEYWORD2
brightness      KEYWORD2
clear           KEYWORD2
sendtext        KEYWORD2
slidetext       KEYWORD2
```

Pour coloriser les termes qui nous plaisent, nous n'avons qu'à les spécifier suivis d'un mot-clé :

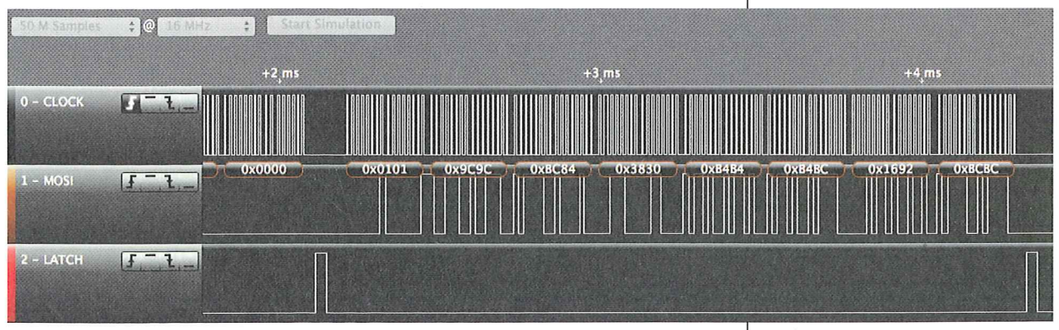
- `KEYWORD1` pour les types de données,
- `KEYWORD2` pour les méthodes et fonctions,
- `LITERAL1` pour les constantes.

Il ne nous reste plus qu'à placer tout cela dans le répertoire `libraries` de notre carnet de croquis (en fait on développe généralement directement la bibliothèque à cet endroit) et à écrire un petit croquis pour tester (là aussi, c'est un pieux mensonge, car on teste en même temps qu'on développe) :


```
#include <DSP0401B.h>

#define SDI_PIN 2
#define CLK_PIN 3
#define LE_PIN 4
#define OE_PIN 5

DSP0401B mondisp;

void setup() {
  mondisp.begin(2, SDI_PIN, CLK_PIN, LE_PIN, OE_PIN);
  mondisp.brightness(5);
}

void loop() {
  mondisp.sendtext("HACKABLE");
  delay(1000);
  mondisp.slidetext("LE MAGAZINE TROP COOL",150);
  delay(1000);
}
```

Et voilà. Ce fut beaucoup de travail, d'explications et de texte, mais le résultat est là. Nous n'avons plus besoin de jongler avec des caractères isolés et je vous laisse imaginer comme il devient facile d'écrire un croquis prenant du texte venant du moniteur série. En réalité, tout est énormément simplifié, car pour peu que l'on multiplie les afficheurs (ici à `2`) nous avons presque quelque chose d'aussi souple que `LiquidCrystal`. La fourniture du courant pour deux lignes de 16 caractères, soit 8 modules et donc potentiellement environ 4 ampères, est une autre affaire. Il devient extrêmement aisé d'ajouter à n'importe quel projet un affichage très voyant et composé de leds (je vous ai dit que tout le monde aime les leds ?) !

Lorsqu'on rencontre des problèmes lors du développement, il est relativement difficile de savoir si c'est son code qui pose problème ou une réaction du module connecté. On peut alors se tourner vers l'analyseur logique pour « écouter » les données qui circulent. On voit ici clairement les valeurs hexa des données envoyées aux TLC5926, récupérées par un Salae Logic (sans le chiffre derrière, c'est un ancien modèle). Dans notre cas, le problème était un problème de timing. Nous utilisions LAT trop tôt après l'envoi du dernier bit.



La suite, elle, relèvera de votre envie. Nous avons là une bibliothèque qu'en version 0.0.1 et il y a de la place pour de l'optimisation (SPI matériel), des améliorations et bien des ajouts de fonctionnalités et d'effets. Notre voyage qui a commencé avec un simple registre à décalage est devenu un projet prometteur et une belle opportunité de contribuer à l'écosystème Arduino. DB

ADAPTATEUR USB/SÉRIE : INDISPENSABLE DANS VOTRE TROUSSE À OUTILS

Denis Bodor

Travailler en ligne de commandes sur une Raspberry Pi sans écran, piloter un adaptateur Bluetooth, échanger des informations entre PC et Arduino, configurer un modem GSM/3G pour envoyer des SMS... Autant d'activités qui nécessitent d'une manière ou d'une autre d'avoir un adaptateur USB/série sous la main. Qu'est-ce que ce périphérique, comment et où en acheter, et comment l'utiliser ? Autant de points que nous allons découvrir ici...

Il fait partie de la boite à outils du bidouilleur au même titre que le multimètre et le fer à souder. Ce petit périphérique USB est sans le moindre doute LE petit bidule à avoir sur soi en toutes circonstances (on sait jamais). Nous allons vous faire découvrir ici cette petite merveille et le monde très animé dont il fait partie, mais avant cela, prenons le temps d'un petit détour au travers d'un peu de connaissances générales et d'histoire.

1. LE (VIEUX) PORT SÉRIE

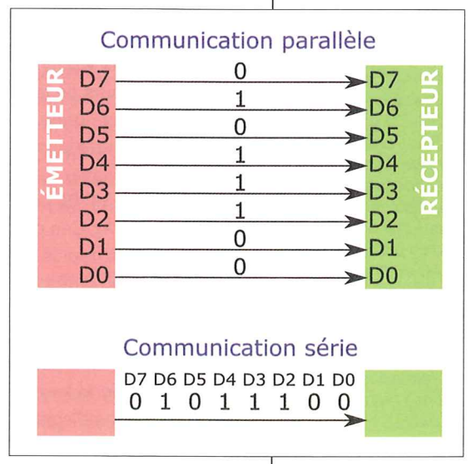
D'un point de vue théorique et général, une liaison série est une liaison où les bits sont envoyés les uns à la suite des autres. Par opposition, une liaison parallèle utilise de multiples connexions et envoie donc plusieurs bits simultanément, en parallèle. La largeur du bus parallèle détermine bien entendu le nombre de bits qu'il est possible d'envoyer en une seule fois.

Si on remonte un peu dans le temps, les PC illustrent parfaitement les deux types de liaison. Nous avons d'une part les ports parallèles au format DB-25 femelles composés de 25 connecteurs, initialement destinés aux imprimantes puis aux scanners, aux lecteurs Zip 100 (des disquettes de 100 Mo, waouh !), aux lecteurs CD externes, etc. Le port série de ces machines, quant à lui, se présentait sous la forme d'un connecteur DE-9 (faussement désigné DB-9) ou DB-25 mâles. Au regard de la vitesse des composants de ces ordinateurs, il était généralement admis que les échanges importants de données devaient forcément passer par un bus parallèle permettant un débit plus important.

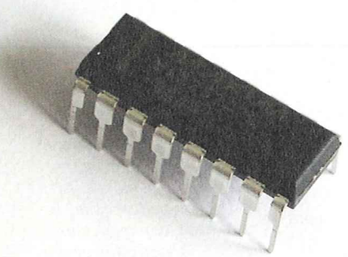
Cet état de fait est resté longtemps inchangé, mais la vitesse aidant et les débits autorisés augmentant de manière drastique, une grande partie de ce qui était la chasse gardée des bus parallèles est tombée sous le joug des liaisons série. Le port parallèle, standardisé sous le nom IEEE 1284 a totalement disparu ainsi que le port série tel qu'il était implémenté. Mais aujourd'hui, les successeurs de ce dernier que sont l'USB, le PCI Express ou encore Thunderbolt, FireWire ou SATA, sont tous des bus séries utilisés pour des applications qui historiquement reposaient sur des bus parallèles (PCI, ISA, IDE, etc.).

Et ce n'est pas tout, dans le monde de l'électronique aussi les bus séries sont une véritable invasion : i2c, SPI, EIA-485 (RS-485), CAN, DMX, MIDI... Pourquoi cette prédominance « soudaine » ? Les bus parallèles étaient privilégiés avant l'arrivée des circuits intégrés spécialisés dans les transmissions série à haute vitesse. Transférer des données en parallèle était alors bien plus simple que de les sérialiser. Les choses ont changé et la technologie aussi. Aujourd'hui, les avantages des bus parallèles ont perdu leur caractère exceptionnel et leurs problèmes sont devenus un véritable handicap. Un bus série nécessite peu de connexion et donc peu de câblage, ce à quoi s'ajoute une complexité bien inférieure sur le plan de la connectique.

Prenez le bus USB 2.0 : une alimentation, une masse et deux lignes de données D+/D- (c'est une signalisation différentielle, pas un bus parallèle de deux lignes). Imaginez maintenant une version parallèle 8 ou 16 bits



Comparaison entre une transmission parallèle (en haut) et série (en bas). Basiquement, plus de débit, mais plus de câbles pour l'un et moins de câbles, mais moins de débit pour l'autre. Avant l'arrivée de composants dédiés et performants, il était largement plus facile d'envoyer beaucoup d'informations via une liaison parallèle (IDE, ISA & PCI, SCSI, etc.).



Un classique du genre, le MAX232 de Maxim était, et est encore, un circuit spécialisé dans la conversion des signaux RS-232 vers TTL. Son travail, avec l'aide de quelques condensateurs, est de transformer le +12/-12 volts d'un port série de PC en 0/+5 volts utilisable avec un microcontrôleur (source : Wikipedia, puisque je n'ai pas retrouvé les miens dans mon vieux stock).

et un connecteur 8 ou 16 fois plus large et maintes fois plus fragile et complexe. Vous voyez le problème avec votre smartphone ?

Le port série DE-9 des PC d'il y a quelques années répondait à des critères particuliers. Comme le débit devait être relativement important pour arriver à quelque chose d'utilisable (on parle ici de quelques 19200 bps (bits par secondes), voire 57600), il ne fallait pas qu'une interférence électromagnétique perturbe la transmission. Ajoutez à cela que l'atténuation du signal est proportionnelle à la longueur du câble et vous voici obligé d'utiliser des tensions relativement « respectables » pour arriver à vos fins. Il est intéressant d'ailleurs de jeter un œil à la page Wikipédia consacrée à la norme RS-232. On y trouve un tableau résumant les relations entre débit et longueur de câble. À 9600 bps, nous sommes à 15 mètres maximum, mais à 38400 on chute à 3,7 mètres (c'est totalement linéaire). Ces problèmes existent toujours, bien entendu, mais depuis certaines technologies comme la signalisation différentielle sont bien plus facile à utiliser grâce à des circuits intégrés dédiés. L'USB, encore lui, est un bon exemple puisqu'il utilise cette technologie et une paire torsadée de câbles pour les données. La longueur maximale d'un câble USB 2.0 est de 5 mètres d'après la norme, à peine plus que nos 3,7 mètres, mais à un débit de 480 Mbit/s (*High Speed*). C'est 13000 fois plus rapide !

L'évolution touche aussi les tensions utilisées. Ainsi une liaison RS-232 repose sur des tensions entre +/-3V et +/-25V d'après la norme même si en pratique c'est +/-12V qui est utilisé. Vous vous demandiez pourquoi les alimentations de PC fournissent souvent -12V avec très peu de courant par rapport aux +12V, +5V et +3,3V ? Hé bien, c'est pour ça. L'USB lui (décidément), utilise des tensions relatives à 5V (+/- une tolérance) bien plus faciles à gérer par des composants et systèmes utilisant déjà typiquement des tensions de 5V ou 3,3V.

Il y a quelques années également, on avait donc l'habitude d'utiliser des convertisseurs spécialisés capables de transformer des signaux 0/+5V en +/-12V. Le plus connu de ces composants est sans le moindre doute le MAX232 de Maxim. Son travail consiste à simplement transformer un 0 logique en une tension entre +3V et +15V et un 1 logique en une tension entre -3V et -15V, et inversement. Nous avons d'un côté le monde TTL (*Transistor-Transistor Logic*) et l'autre le RS-232. La technologie TTL normalise l'alimentation en 5V et par la même occasion les niveaux logiques (haut entre 0 et 0,5V et bas entre 2,4V et 5V). Aujourd'hui, c'est la technologie CMOS (*Complementary Metal Oxide Semiconductor*) qui la remplace majoritairement avec des niveaux logiques basés sur la tension d'alimentation, VDD (bas entre 0 et 1/3 de VDD, et haut entre 2/3 de VDD et VDD). Initialement, le MAX232 et ses clones compatibles utilisaient une technologie TTL, mais le composant existe maintenant en version CMOS.

Pour les (vieux) Apple fanboys, sachez qu'en ce temps, alors que l'industrie PC embrassait joyeusement RS-232, Apple opta pour la norme RS-422 utilisant la signalisation différentielle avec une possibilité de configuration en RS-232, le tout sous la désignation LocalTalk. Le résultat fut l'utilisation de communication à 115200 bps et une architecture réseau propre à la marque. Ainsi, il faut reconnaître que même si

les surprenantes innovations se font rares aujourd'hui du côté de Cupertino (un écran 5K et un grand iPhone sont des évolutions et non des révolutions), la firme à la pomme a historiquement su souvent se risquer sur le terrain des technologies les plus avancées. [Note toute personnelle parce que je suis énervé : en même temps avec des clients (dont moi) capables de payer presque 1200€ un MacBook Pro 13" à core i5 (fin 2011) sur lequel on peut à peine jouer à WoW aujourd'hui même avec une résolution pathétique (Intel HD 3000), c'est facile d'innover. C'est tout de même la seule machine qui me permet de faire du multiboot Mac OS X - FreeBSD - GNU/Linux - Windows, et la batterie tient toujours 5 bonnes heures. Mais il n'est pas question que je paie 2500€ pour le seul MacBook Pro avec une GeForce.]

En avançant un peu dans l'histoire, les amateurs ayant l'habitude de bidouiller des circuits communiquant avec les PC ont eu à faire face à l'arrivée de l'USB. Petit à petit alors que cette nouveauté gagnait du terrain, le bon vieux port RS-232 se faisait de plus en plus rare. Heureusement, sont naturellement apparus des périphériques USB proposant un port RS-232. Nous nous retrouvions donc avec une chaîne USB vers RS-232 et RS-232 vers TTL/CMOS. Bien entendu, il n'a pas fallu longtemps avant de voir apparaître des circuits spécialisés se débarrassant de la conversion superflue en proposant directement de l'USB vers un port série TTL/CMOS. La transition se passa en douceur, à mesure que les connecteurs DE-9 disparaissaient des machines. Il n'y avait plus besoin de conserver une compatibilité tant les machines dépourvues de RS-232 commençaient à dominer le marché.

Le destin du port parallèle, lui, génériquement utilisable plus facilement par les hackers fut bien différent. Il resta plus durablement dans les configurations standards puis, lui aussi, fut remplacé par des adaptateurs USB. Malheureusement ces périphériques de conversion s'étaient spécialisés dans la connexion aux imprimantes et ce qui faisait le bonheur des bidouilleurs (contrôler les lignes de données individuellement comme des GPIO) devint de plus en plus difficile.
Le port parallèle disparut et ce type de bus perdit en popularité de manière générale au fil du temps. Si vous regardez aujourd'hui les interfaces présentes sur un PC ou un Mac, il n'y a plus guère de trace de bus parallèle accessible.

La transmission parallèle des informations retrouve cependant une nouvelle jeunesse dans un tout autre domaine qui est celui de la communication radio. Alors qu'il était d'usage d'utiliser des protocoles série, on voit actuellement de plus en plus apparaître des normes utilisant plusieurs canaux pour transmettre plusieurs bits en même temps. Et ça, c'est bien une liaison parallèle (même pas mort !).

2. LES ADAPTATEURS USB/SÉRIE

L'arrivée massive des adaptateurs USB/série TTL/CMOS va de pair avec l'omniprésence des ports série TTL/CMOS dans de nombreux équipements. En effet, quel que soit le système en œuvre, pour peu qu'il soit suffisamment « musclé » pour faire fonctionner un système d'exploitation (GNU/Linux ou autre), il faut un moyen pour que le développeur, l'ingénieur ou l'utilisateur curieux puisse prendre la main. C'est également le cas pour des appareils ne faisant pas fonctionner de système d'exploitation souhaitant offrir, pour des raisons de maintenance par exemple, un moyen de prendre le contrôle, de changer la configuration, de relever des données ou d'apporter des modifications.

Tous ces sympathiques équipements, petits ou grands, disposent généralement d'un port série et

d'une interface de gestion, mais en 0/+5V ou en 0/+3,3V, pas ou plus en RS-232 (j'ai ouï-dire cependant que dans certaines rames TER (ZGC ?), encore aujourd'hui, c'est un port RS-232 qui est utilisé pour gérer le système des toilettes embarquées). Ainsi, plutôt que de s'amuser à convertir les signaux de ce type en RS-232 (puis en USB) on utilise des petites « clés » USB faisant tout le travail.

Ce genre de périphériques repose généralement sur une puce unique assortie de quelques composants passifs (résistances et condensateurs) et éventuellement quelques leds et un oscillateur à quartz. Parmi la myriade de puces, quelques-unes se retrouvent plus souvent que d'autres :

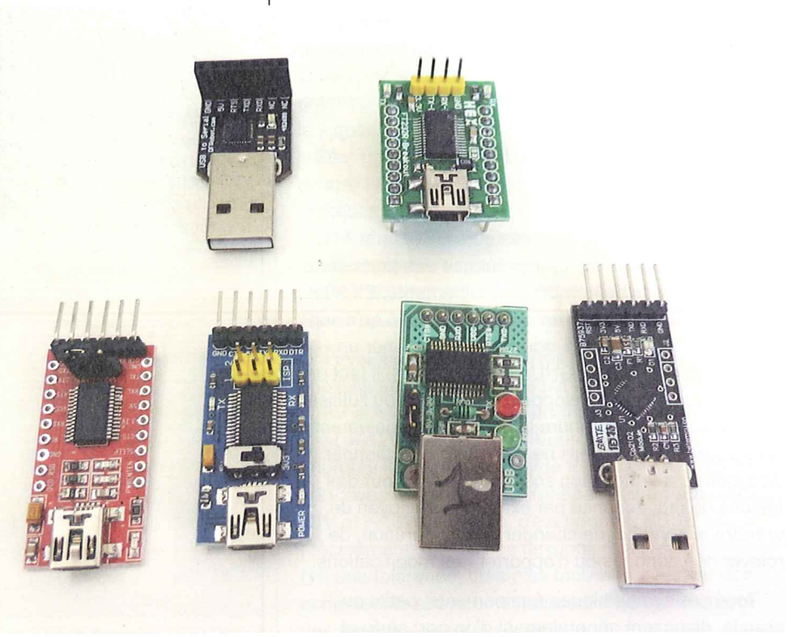
- Prolific PL2303,
- Silicon Labs CP210x,
- et la très populaire FTDI FT232RL.

Ce type de composants repose sur une logique simple : fournir d'un côté un périphérique USB CDC (classe port série ou *Communications Device Class*) et de l'autre une interface TTL/CMOS. Le circuit sur lequel est monté la puce se charge de toute l'architecture, de la mise à disposition des signaux logiques historiquement présents sur du RS-232 (RX, TX, RI, DCD, DTR, DSR, RTS et CTS), de l'alimentation via USB et souvent de la sélection de tension entre 5V et 3,3V.

Quel que soit le système d'exploitation utilisé sur la machine côté USB, le périphérique apparaît comme un nouveau port série. Tantôt il est nécessaire d'installer un pilote (Windows et Mac), tantôt celui-ci est directement soit présent dans le système (Linux), soit téléchargé automatiquement. Dans tous les cas, du point de vue logiciel, on dispose d'un port série standard utilisable en tant que tel. Ceci implique une notion très importante : tous les outils, programmes, codes sources, applications et logiciels qui manipulent et savent utiliser des ports séries « standards » RS-232, feront de même avec ces périphériques USB.

Vous ne l'avez peut-être pas remarqué, mais la plupart des cartes Arduino utilisent ce type de techniques et de puces. Initialement basées sur des puces FTDI, les nouvelles cartes Arduino font usage d'un microcontrôleur Atmel pour cette liaison. Mais l'AVR en question, un ATmega32U4 est connecté directement au port USB et fonctionne comme un périphérique CDC et donc un port série USB (sur une Leonardo le vrai port série interne au microcontrôleur (USART) est `Serial1`, le périphérique USB CDC est `Serial0`). La Uno R3 quant à elle s'est aussi débarrassée de la puce FTDI, mais là encore

Une petite collection de différents adaptateurs USB/série TTL collectés/achetés au fil du temps. Tous ne se valent pas et se différencient de par leur qualité générale, leurs fonctionnalités et la puce utilisée.

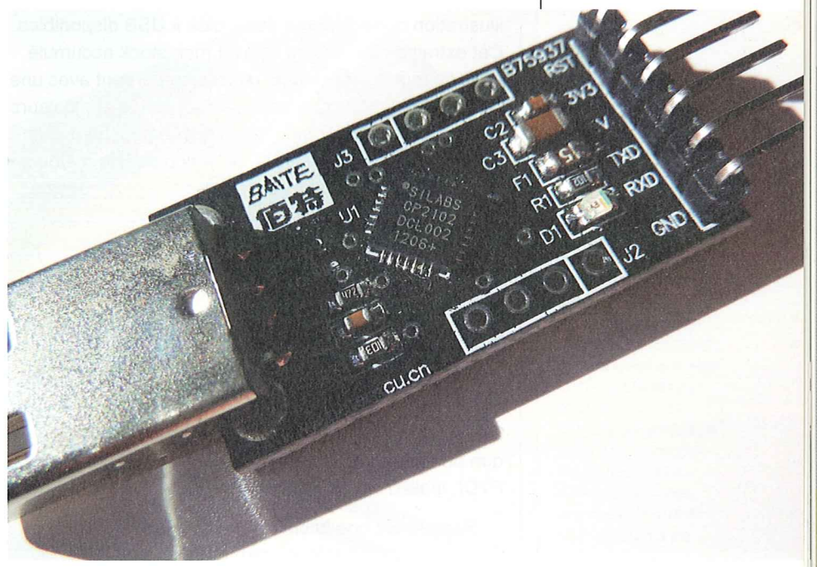


c'est un microcontrôleur AVR qui la remplace (différent de l'ATmega328 central) : un ATmega16U2 disposant également de fonctionnalités USB et fournissant un périphérique USB CDC (ce qui explique la présence de deux connecteurs ISP sur la carte, un pour chaque microcontrôleur). Idem sur la Due qui troque son AVR principal pour un AT91SAM3X8E (ARM Cortex-M3), mais utilise aussi un ATmega16U2 pour l'USB CDC et donc le port série « virtuel ».

Mais revenons aux petites « clés » USB et en particulier celles vendues de-ci de-là. Une simple recherche « USB serial » sur eBay vous retournera littéralement des centaines de produits allant de 2 euros port compris (oui, oui, 2€) à 10 ou 15 euros. Dans cette jungle, difficile de s'y retrouver et pour cause. Il semblerait qu'absolument tous les habitants de Hong Kong et Shenzhen se soient lancés dans la fabrication et la vente de ces périphériques. Pour arriver à faire un choix et minimiser le risque de se retrouver avec un produit douteux ou de mauvaise facture, voici à quoi il faut faire attention :

- Optez systématiquement pour des périphériques qui permettent de choisir la tension entre 5V et 3,3V. Préférez les produits utilisant un interrupteur pour la simple et bonne raison que ce genre d'ajout coûte cher et donc n'est généralement pas intégré sur des produits bas de gamme. Évitez comme la peste les options de sélection de tension nécessitant une soudure.
- Intéressez-vous principalement aux périphériques dont la description mentionne la puce utilisée et observez bien la photo pour vous assurer que cela correspond. On n'est pas à l'abri d'une tricherie quelconque, mais au moins on limite les risques. Si la photo masque d'une manière ou d'une autre, de manière évidente, la puce, fuyez ! (cf plus loin dans l'article, à propos des contrefaçons).
- Privilégiez des descriptions complètes et techniques, de préférence avec un lien vers une documentation (*datasheet*) en PDF, voire vers le schéma du circuit.
- Ouvrez un onglet de votre navigateur en même temps que vous choisissez votre produit et cherchez « Fake FTDI » ou « Fake PL2303 » dans Google Images. Vous aurez une bonne illustration des produits que d'autres utilisateurs ont désignés comme étant à base de puces contrefaites.
- Accordez de l'importance aux produits qui proposent plus de broches que les simples connexions GND, VCC, RX et TX. Si le fabricant s'est donné la peine d'ajouter des broches pour fournir tous les signaux offerts par la puce, c'est qu'il vise potentiellement des utilisateurs avertis et s'est donné du mal pour créer un produit plus complet.

Le CP2102 de Silicon labs, l'un des nombreux « single chip USB to UART bridge » disponible sur le marché. Mais le composant principal n'est qu'un élément décisif, l'autre étant le circuit utilisé par le fabricant. Ici, seuls quelques signaux (RT et TX) sont disponibles alors que le CP2102 propose également RI, DCD, DTR, DSR, RTS et CTS.



- Aimez les leds ! Ce sont des composants qui en plus d'être utiles pour littéralement voir qu'il se passe quelque chose dans la communication, sont coûteux. Là encore si le fournisseur augmente délibérément ses coûts de fabrication (à l'échelle chinoise), c'est qu'il ne cherche pas forcément à vendre le produit le plus économique et bas de gamme au monde.
- De manière toute pondérée, jugez sur l'image et l'impression générale. Certes, c'est un peu comme les sites de rencontre, la réalité n'est jamais vraiment tout à fait comme sur la photo de l'annonce (et ceci vaut pour les deux genres, ne venez pas m'accuser de sexisme). On voit généralement assez bien le soin apporté à la conception et la réalisation du périphérique. La description est également révélatrice. Si c'est en « chinglish », c'est relativement mauvais signe (mais pas toujours).

Soyons clairs, avant de recevoir le produit, et même ensuite, vous ne serez pas certain d'avoir acheté quelque chose de qualité ou même que vous oserez l'utiliser avec une carte ou un périphérique de valeur (de peur de l'endommager avec une tension non contrôlée). C'est le jeu.

Cet article présente en photo (voir page 62) une illustration de la diversité des « clés » USB disponibles. Cet extrait de 6 modèles issu de mon stock accumulé au fil du temps (ces petites choses se perdent avec une facilité troublante) couvre plusieurs qualités et plusieurs époques. Le premier modèle en haut à gauche accompagnait un produit et s'avère de bonne qualité même si le *formfactor* est un peu particulier. Mon préféré reste cependant le modèle en bas à gauche, marqué d'un « 1 », car il ne m'a jamais posé problème quel que soit la carte ou de système utilisé. Celui juste à sa droite est un modèle récent qu'on reconnaîtra à l'utilisation d'un *package* relativement moderne pour la puce (par rapport au SOIC). Enfin, je préciserai que le modèle en haut à droite, bien que proposant un grand nombre de signaux (tous ceux de la puce FTDI) est celui avec lequel j'ai rencontré le plus de problèmes. Il est plus que probable qu'il ne s'agisse pas d'une vraie puce FTDI, mais d'une contrefaçon de mauvaise qualité.

Remarquez que la connexion côté USB est également importante, on remarque dans la collection que certains utilisent du type A, du type B et du mini-B. Le plus pénible est sans doute le B, car ce genre de connecteur se fait relativement rare et est vraiment « massif ». Notez que le micro-B (comme sur les smartphones) commence à se populariser sur ce type de produits peu coûteux, mais je suppose qu'il faudra attendre que tous les autres modèles en stock en Chine soient vendus pour en trouver plus facilement.

3. FTDI GATE !

Il est temps à présent d'arriver au sujet qui fâche et qui a fâché une quantité incroyable d'utilisateurs, d'ingénieurs et d'industriels. Le déroulé des événements que nous allons présenter ainsi que les causes et explications découlent directement de nos observations techniques et des réactions à la fois des utilisateurs (réseaux sociaux et listes de diffusion) et du principal concerné (réseaux sociaux et site officiel). Une part de subjectivité est donc présente dans cette analyse, mais dans les faits il semblerait que, de l'avis général, FTDI a tout simplement réussi à ruiner sa réputation en quelques jours.

Commençons par le début. FTDI, pour *Future Technology Devices International* est une société écossaise produisant des semiconducteurs. L'un des produits les plus connus de la société se décline en une gamme de puces de conversion USB/série : les FT232. FTDI a su se tailler une réputation depuis sa création en

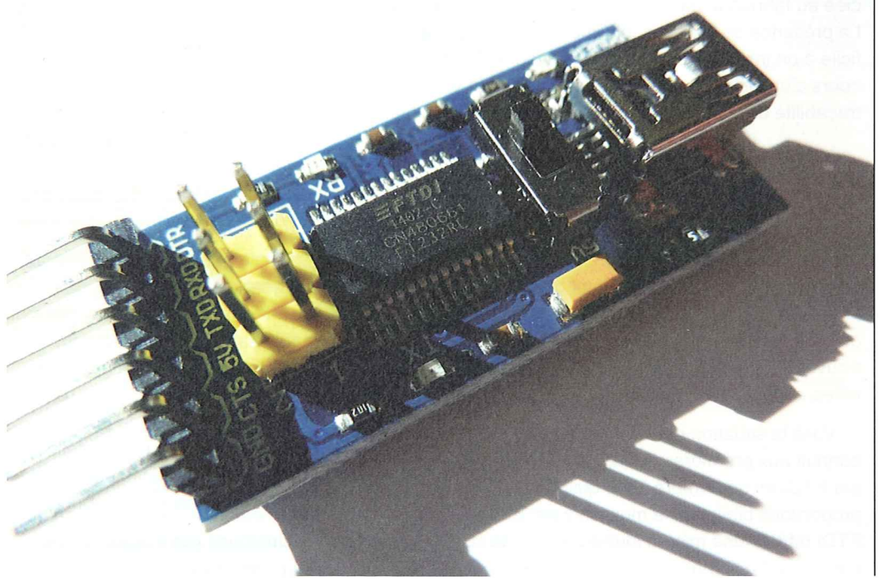
1992 en fournissant, par exemple IBM pour sa gamme PS/1. Au fil du temps le FT232 est devenu LA puce à utiliser pour l'USB/série et est presque devenu un standard de fait (ou du moins un choix réflexe). Fournissant également des bibliothèques permettant de prendre totalement le contrôle de la puce, le produit est également devenu un élément important pour les utilisateurs souhaitant personnaliser le composant (changement des ID USB, par exemple) ou encore l'utiliser pour faire office de contrôleur GPIO (*bitbanging*).

Le FT232 a donc très rapidement trouvé son chemin littéralement partout, des systèmes embarqués aux *appliances*, en passant par les routeurs et autres équipements réseau, les set top box ainsi que les cartes du type Arduino. Il est important de noter que cette diffusion massive a créé un parc d'utilisateurs absolument gigantesque aussi bien du côté utilisateurs que de celui des ingénieurs et constructeurs.

Victime de son succès, la puce FT232 a commencé à subir une concurrence toute légitime. Copier le fonctionnement d'un composant sans violer la propriété intellectuelle ou tout simplement produire une puce offrant les mêmes fonctionnalités n'est pas un problème, c'est même de bonne guerre et c'est ce qui pousse à l'innovation. C'est d'ailleurs là l'intérêt premier des brevets et de la protection de la propriété intellectuelle : on n'a pas le droit de copier à l'identique, c'est du vol. On est donc obligé de faire différemment et mieux pour se démarquer.

Mais la copie consistant à analyser le fonctionnement d'un composant et à le reproduire à l'identique est illégale. Pire encore, si on vend le produit en revendiquant qu'il s'agit du modèle original, c'est de la contrefaçon. Exactement comme une fausse Rolex vendue à la sauvette au coin d'une rue.

Le problème de la contrefaçon de composants n'est ni nouveau, ni facile à régler, en particulier lorsque le



Sans doute le plus connu et jusqu'à récemment le plus populaire des composants USB/série : le FT232RL. Le contrecoup d'une telle popularité est l'apparition massive de contrefaçons. Mais une mauvaise réaction à ce type de phénomène de la part d'un fabricant peut rapidement faire fuir les utilisateurs et ingénieurs, et détruire en un temps record la réputation du fabricant tout en ne portant pas vraiment préjudice aux fabricants de contrefaçons.

produit copié est déjà plus ou moins idéalement adapté au marché et donc très populaire. C'est un problème que doit gérer l'ensemble des entreprises produisant des semiconducteurs. C'est un fait, il existe des sociétés, principalement en Asie, analysant en détail le fonctionnement de certains composants et produisant en masse des clones libellés de la marque originale. Cette pratique ne date pas d'hier, les Soviétiques le faisaient déjà à l'époque avec des éléments et machines complètes copiées de l'industrie informatique américaine (mais l'URSS n'exportait pas, sauf à Cuba). Plus « amusant » encore (si on n'en est pas la victime), certaines sources douteuses n'hésitent pas à vendre des composants qui ne contiennent pas réellement de puce, juste le boîtier avec des pattes, mais rien dedans... Et tout ce joyeux monde se retrouve généralement sur des sites comme **alibaba.com**.

Voici pour le problème initial, mais il s'étend bien au-delà, car la fourniture de composants en quantité est un marché à part entière avec ses sources, ses intermédiaires, ses grossistes et ses distributeurs. Vous, moi et même des gens se lançant dans des projets ne peuvent acheter directement à l'usine du constructeur. En fonction de la quantité, on passe par un courtier, un distributeur ou un détaillant (tantôt une boutique en ligne est associée au fabricant, mais ce n'est pas un canal industriel). La présence de tous ces intermédiaires fait qu'il est difficile à un instant donné de suivre avec certitude le parcours d'un stock de composants. Il n'y a pas vraiment de traçabilité comme on en retrouve dans l'agroalimentaire.

Lorsque le composant dont vous avez besoin est en rupture chez un grossiste, vous devez assurer le planning de production et en trouver un autre. Ainsi, en cherchant des alternatives, il n'est pas impossible que vous tombiez sur un stock de contrefaçons, ou partiellement composé de contrefaçons, ayant déjà changé de main plusieurs fois. Et vous allez intégrer cela à votre produit ou projet sans le savoir. C'est ainsi que ces contrefaçons trouvent leur chemin dans des produits parfaitement honnêtes et originaux.

Voilà la situation mondiale « normale ». Ce qui nous conduit aux premières heures d'une manœuvre engagée par FTDI en septembre 2014 qui a subitement pris des proportions bibliques le mois suivant. En septembre, FTDI diffuse une mise à jour de son pilote pour Windows concernant les composants USB/série. Cette mise à jour intègre un code particulier testant la puce afin d'en déterminer l'originalité. Si la puce ainsi interrogée ne réagit pas comme prévu, elle est considérée comme une contrefaçon et sa mémoire EEPROM est réécrite de manière à lui attribuer un identifiant produit USB (*Product ID*) à `0x0000` en lieu et place de `0x6001` (pour le périphérique `0403:6001`). Suite à cette opération, le composant contrefait n'est plus reconnu par le pilote sur le système, mais également sur toutes les autres machines étant sous Windows ou n'importe quel système (puisque les pilotes utilisent l'ID vendeur et l'ID produit pour s'attacher au périphérique).

Les choses s'accélèrent mi-octobre avec l'arrivée de ce pilote mis à jour sur Windows Update. Subitement, l'ensemble des utilisateurs reçoit cette version très agressive avec les contrefaçons et les problèmes commencent à apparaître. Beaucoup d'utilisateurs se retrouvent avec des produits utilisant des FT232 contrefaits qui ne sont plus pris en charge immédiatement après leur connexion, et ne sont plus utilisables non plus sur d'autres systèmes par la suite.

Finalement, des développeurs analysent le pilote et diffusent les informations à propos des manipulations opérées sur les composants et la mise à zéro de l'ID produit. La nouvelle se répand comme une traînée de poudre : FTDI endommage (*brick*) délibérément tous les produits utilisant des contrefaçons de ses composants.

Quelques jours après, devant les protestations des utilisateurs

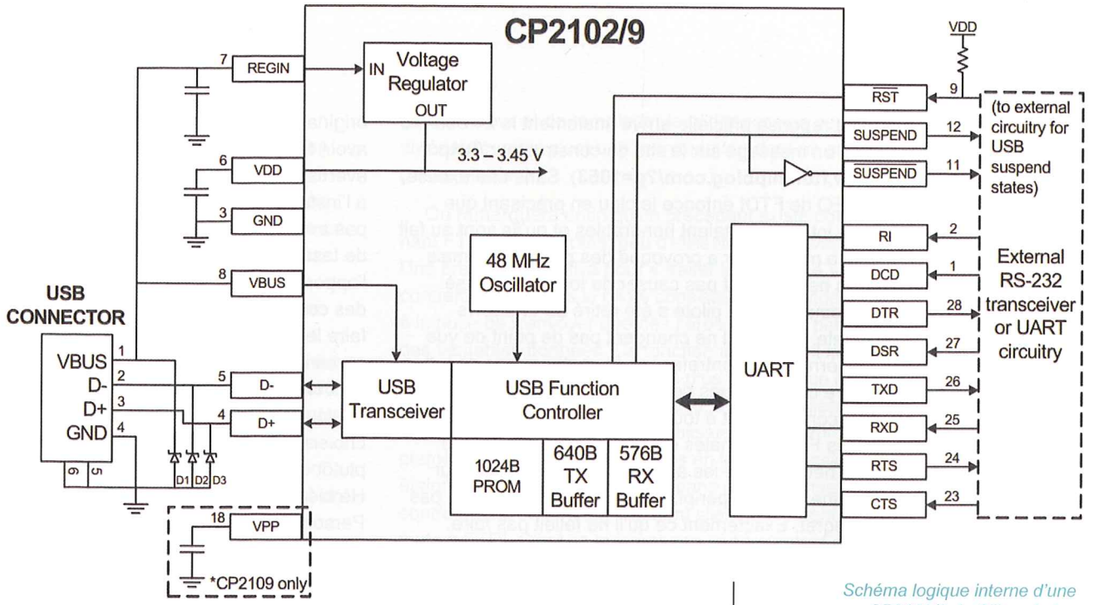


Schéma logique interne d'une puce CP2102/9 de Silicon Labs. Ce composant a la particularité de nécessiter peu de composants externes pour fonctionner. On remarque, entre autres, que l'oscillateur cadençant le composant est interne, nul besoin d'ajouter un quartz. Le CP2102 fonctionne en 3,3V via un régulateur interne qui adapte la tension à partir des 5V de l'USB, et fonctionnera aussi bien avec un périphérique 3,3V et 5V (5,8V max sur les ports). Le CP2102 est officiellement mon nouvel ami suite au scandale FTDIgate d'il y a quelques mois (source : datasheet Silicon Labs).

outrés, FTDI (ou Microsoft) retire le pilote, mais le mal est fait. Bon nombre de systèmes Windows possèdent ce pilote et surtout la réputation de FTDI vient d'en prendre un coup. La manœuvre était stupide, car s'en prendre aux clients n'est clairement pas une solution, en particulier lorsque ceux-ci ne sont même pas au courant qu'ils utilisent un composant contrefait. Le résultat était prévisible puisque cela donne clairement l'image d'une société qui n'hésite pas à pénaliser des utilisateurs ignorant totalement qu'un problème existe, afin de défendre agressivement et brutalement sa propriété intellectuelle.

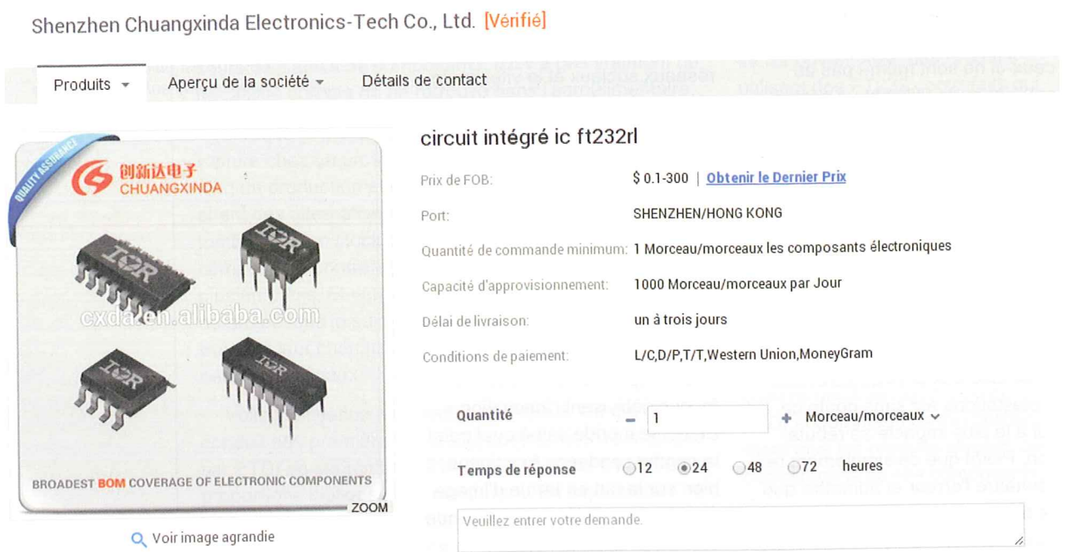
Pire encore, la réaction de FTDI face aux problèmes et aux protestations est sans doute ce qui a le plus impacté sa réputation. Plutôt que de simplement reconnaître l'erreur et admettre que ce n'était vraiment pas une bonne idée tout en présentant de plates excuses (et en mettant le responsable à la porte), FTDI est resté campé sur ses positions. Et ce en répondant de manière agressive aux utilisateurs et surtout, surtout, en tentant d'effacer ses traces ensuite (suppression des tweets). Bien entendu, c'est mal connaître le net d'aujourd'hui, les réseaux sociaux et la vitesse de propagation de ce genre de raz-de-marée. Les utilisateurs se sont empressés de faire des captures des tweets, se doutant de la suite, et ceci n'a fait que gonfler encore davantage la déferlante. Ces messages de FTDI sur Twitter étaient des réponses condescendantes du type « référez-vous à la licence du pilote », « utilisez des puces FTDI originales », « les contrefaçons détruisent l'innovation »... Et tout le monde sait à quel point la condescendance fonctionne bien sur le net en terme d'image de marque... Enfin, tout le monde sauf FTDI apparemment.

La réponse officielle arriva finalement le 24 octobre avec un message sur le site du constructeur (**http://www.ftdichipblog.com/?p=1053**). Sans une excuse, le CEO de FTDI enfonce le clou en précisant que leurs intentions étaient honorables et qu'ils sont au fait que la mise à jour a provoqué des problèmes, mais qu'ils ne voulaient pas causer de tort. Est précisé également que le pilote a été retiré de Windows Update, mais qu'il ne changent pas de point de vue concernant les contrefaçons et continuerons de lutter contre ces pratiques de manière non invasive. Enfin, ils recommandent à tous leurs clients d'acheter des puces FTDI originales et leur équipe d'assistance sera heureuse de les aider s'ils ont des doutes sur l'origine de leurs périphériques. Pas une excuse, pas un regret. Exactement ce qu'il ne fallait pas faire.

Le plus important est sans doute la vision des clients qu'on devine de la part de FTDI. Non, FTDI, vos clients gros ou petits ne sont pas des personnes prêtes à tout pour vous détrousser. Personne ne veut délibérément utiliser des contrefaçons et surtout pas les utilisateurs. Pourquoi ne pas innover et investir pour trouver un moyen pour vos clients de savoir avec certitude si leurs composants sont originaux ? Pourquoi ne pas avoir, tout simplement prévu un avertissement ou un message à l'installation ? Pourquoi ne pas avoir proposé un utilitaire de test laissant aux utilisateurs l'opportunité de vérifier l'origine des composants, et ainsi laisser faire le marché ? Parce que, très certainement, vous pensez qu'ils sont tous prêts à économiser quelques centimes d'euros en choisissant un clone douteux plutôt que vos produits de qualité. Hé bien, ce n'est pas le cas ! Personne n'aime les faux.

Est-ce rattrapable ? L'écosystème va-t-il finir par oublier ? Il va falloir du temps, beaucoup de temps, car aujourd'hui la réaction est la même chez un grand nombre d'utilisateurs et elle est parfaitement compréhensible. Les FT232 avaient

Typique Alibaba : des FT232RL à $0,1 contre 2,4€ HT chez Mouser. Chic alors, une affaire ! Heu... attendez voir... ce sont bien des composants originaux, pas des contrefaçons, n'est-ce pas ?



une excellente réputation et étaient généralement choisies par défaut, mais désormais cette histoire leur est attachée de manière indélébile. Même un constructeur aura des doutes quant à l'intégration de ces composants : « et si je tombe sur des contrefaçons ? », « et si le pilote est encore actif chez certains clients ? », etc. La décision de changer délibérément l'ID produit d'un périphérique était impensable et totalement stupide, et FTDI l'a fait. C'est un peu comme si Rolex ou Vuitton allaient directement chez les gens endommager les contrefaçons de leurs produits, après achat. La colère des utilisateurs ne serait naturellement pas dirigée vers les auteurs des contrefaçons, mais contre les marques. C'est l'évidence même ! Beaucoup parlent de suicide économique de la part de FTDI et le terme « FTDIgate », rapidement apparu, est une référence directe au scandale du Watergate qui a conduit Richard Nixon à sa démission de la présidence en 1974.

Personnellement, j'ai été déçu par FTDI. Comme d'autres j'avais le réflexe de me dire que c'était une valeur sûre. À présent, je me dis que ce fabricant ne nous connaît pas, ne nous respecte pas et qu'utiliser un FT232 signifie des doutes et problèmes à la clé. Et bien sûr, à chaque connexion d'une carte avec une puce FTDI, j'aurai invariablement des craintes qu'il puisse s'agir d'une contrefaçon et que l'ID produit soit automatiquement mis à zéro (il est possible de le remettre à la bonne valeur et de remplacer le pilote par une version antérieure, mais c'est une manipulation dont on se passerait bien).

On remarquera enfin qu'un précédent existe concernant FTDI et ses pilotes, peu d'utilisateurs le savent. Une première tentative pour « traiter le problème » des contrefaçons depuis le pilote consistait à demander à la puce de n'envoyer que des zéros en lieu et place des véritables données. Ce fonctionnement était implémenté dans les pilotes Windows plus récents que la version 2.08.14. Il est probable que, faute d'explications ou de message, de nombreux utilisateurs ont simplement considéré que les puces en leur possession étaient défectueuses, peut-être sans même penser aux contrefaçons et donc en imputant silencieusement la faute à FTDI. C'est ce qu'on gagne généralement en ne communiquant pas clairement avec ses clients. En février 2014, un blog russe spécialisé dans l'analyse de semiconducteurs, sur la base du comportement du pilote décida de se pencher sur une comparaison entre un vrai et un faux FT232RL, au niveau de la puce elle-même. Même si vous ne lisez pas l'anglais jetez-y un œil, ne serait-ce que pour les magnifiques images : **http://zeptobars.ru/en/read/FTDI-FT232RL-real-vs-fake-supereal** (au passage, le slogan du blog est totalement adorable : « *We love microchips - That's why we boil them in acid* »).

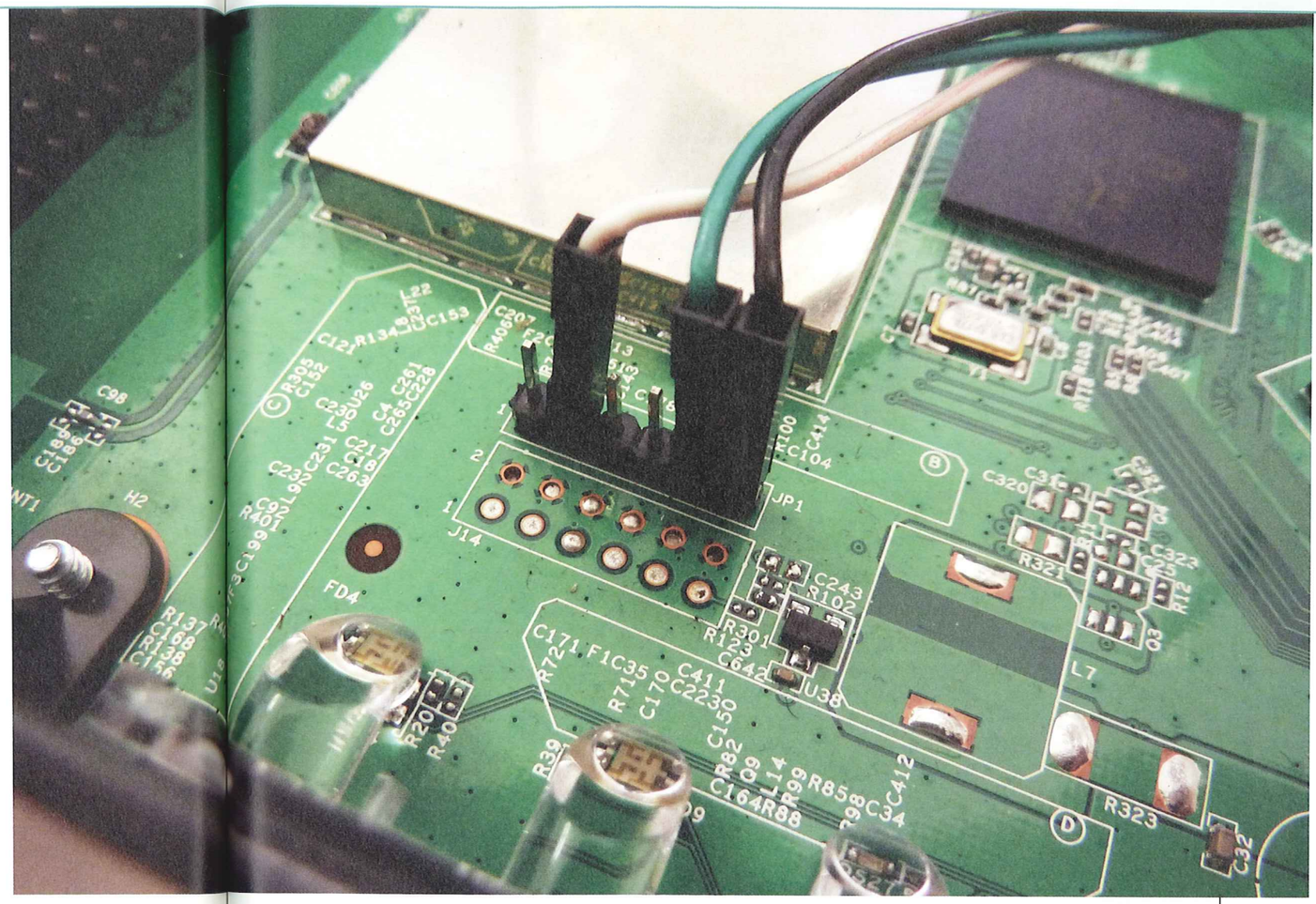
Les développeurs du noyau Linux se sont vus proposer un patch le 23 octobre 2014, en plein scandale, de la part de Russ Dill (apparemment ex-employé de TI) permettant de modifier le pilote pour détecter les contrefaçons. Difficile de savoir si la proposition était sérieuse (sans doute pas, c'est de l'humour de programmeur), mais la réaction des développeurs était, comme souvent, parfaite. Voici la réponse de Greg Kroah-Hartman (**https://lkml.org/lkml/2014/10/23/151**) : « *Funny patch, you should have saved it for April 1, otherwise people might have actually taken this seriously* » (« Patch amusant, vous auriez du le garder pour le 1er avril, autrement les gens pourraient vous avoir pris au sérieux »). Le patch proposé montre surtout la méthode utilisée pour détecter les fausses puces et la réaction des développeurs, dans le même temps, fut effectivement de changer le pilote, mais en lui faisant accepter les puces avec un ID produit mis à zéro afin d'en assurer le fonctionnement.

4. SE SERVIR D'UN CONVERTISSEUR USB/SÉRIE TTL EN QUELQUES MOTS

Pour un article de présentation et de description des adaptateurs USB/série, il est amusant de noter que cette partie sera sans doute la plus petite. En effet, l'utilisation d'un tel périphérique est d'une simplicité déconcertante. En résumé, vous reliez la masse et les broches d'envoi (TX) et de réception (RX) de données entre l'adaptateur et le périphérique, vous branchez l'adaptateur en USB, un port série est détecté par le système et vous utilisez un simple émulateur de terminal comme PuTTY, CoolTerm ou GNU Screen. Il suffira alors de choisir le bon port série (COM* dans Windows, `/dev/ttyUSB*` ou `/dev/ttyACM*` avec GNU/Linux, et `/dev/tty.usbserial-*` sous Mac OS X), régler la vitesse et les paramètres (cf article suivant) et le tour est joué.

Dès lors, tout ce que vous taperez dans l'utilitaire sera envoyé au périphérique cible et tout ce qu'il lui enverra se retrouvera à l'écran. Dans les grandes lignes, c'est aussi simple que cela et cela fonctionne exactement de la même manière que le moniteur série de l'environnement Arduino. Avec des logiciels spécialisés pour la communication par MODEM (cf article suivant) il est même possible de mettre en œuvre des protocoles de transfert de fichiers comme Xmodem, Ymodem, Zmodem ou Kermit. Ceci cependant est, sauf cas particulier, devenu un peu désuet, car la majeure partie du temps, la connexion série est utilisée pour contrôler un système distant, le configurer et très souvent procéder aux transferts via une liaison réseau performante.

Vous pouvez maintenant utiliser ces adaptateurs USB/série aussi bien avec des cartes comme Arduino et des nano-ordinateurs comme la Raspberry Pi ou la BeagleBone Black, mais également avec tous les périphériques intelligents modernes. En effet, dès qu'un système est embarqué (le plus souvent GNU/Linux), il est plus que probable que quelque part dans le produit existe une série de broches proposant un port série 5V ou 3,3V. Routeur, point d'accès Wifi, radio internet, téléviseur, box ADSL, téléphone VoIP, smartphone, liseuse électronique... autant de pistes à suivre pour explorer et hacker. En guise de conclusion, voici une méthodologie à suivre pour cette chasse au trésor :



Gros plan sur le port console série interne d'un routeur NetGear WNR3500U auquel est connecté un adaptateur USB/série. Grâce à cela, nous pouvons prendre la main sur le système GNU/Linux embarqué et en faire ce qui nous chante.

- démonter,
- repérer un connecteur de 3 ou 4 broches (ou plus) ou un emplacement à souder,
- utiliser un multimètre pour repérer la masse parmi ces broches en testant la continuité par rapport à une masse connue (alimentation, broche d'un composant connu, plan de masse, etc.),
- toujours avec le multimètre, mais en mesure de tension, repérer la broche Vcc d'alimentation et mesurer sa valeur, ceci indiquera la tension utilisée par le port et donc le réglage à adopter sur l'adaptateur,
- sur les broches restantes, par déduction connecter au hasard le RX de l'adaptateur,
- démarrer le périphérique pour tester avec un émulateur de terminal réglé à différentes vitesses couramment utilisées (115200, 19200, 9600, etc.),
- si rien ne se passe, tester une autre broche et recommencer. Chercher d'abord le TX permet d'éviter d'envoyer des données par inadvertance depuis le PC s'il ne s'agit pas d'un port série (mais d'un port JTAG par exemple),
- le TX de l'appareil trouvé, chercher le RX de la même manière.

À un moment ou un autre, s'il s'agit bien d'un port série, vous devriez finir par voir du texte et en particulier les messages de démarrage du système. La suite dépend du système en question. Tantôt on arrive directement sur un shell, tantôt c'est un login et parfois `root` en nom d'utilisateur et aucun mot de passe n'est suffisant pour prendre la main (symptomatique du « jamais personne ne va ouvrir notre produit et accéder au port console, pas besoin de mot de passe », ben si !).

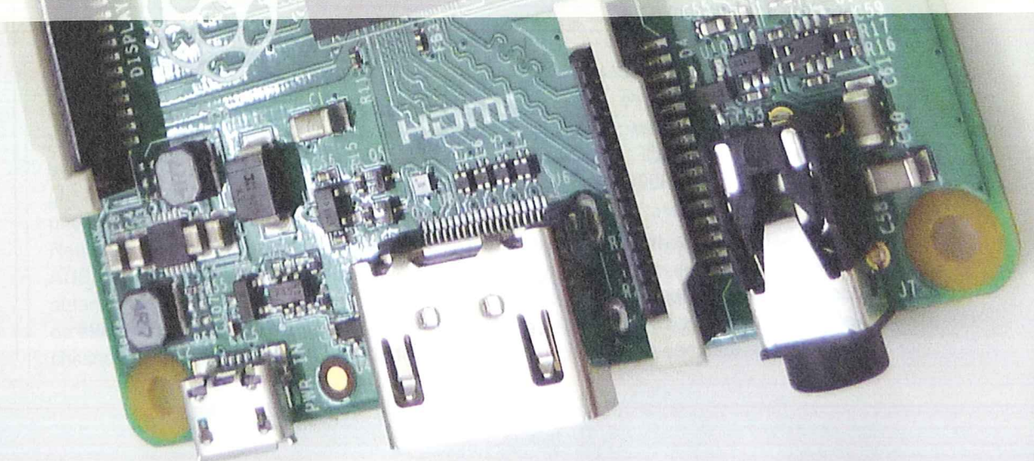
Bonne chasse ! DB

[1] **http://laruche.com/2014/04/05/lqt-connectique-ecran-pc-dvi-vga-hdmi-displayport-547957**

UTILISER UNE RASPBERRY PI SANS ÉCRAN

Denis Bodor

Ne vous est-il jamais arrivé de travailler sur un projet à base de Raspberry Pi sans pour autant, qu'à terme, un écran ou un clavier ne soit indispensable ? Avez-vous déjà trouvé très pénible de sans cesse devoir brancher/débrancher votre unique moniteur HDMI/DVI entre le PC et la Pi ? Avoir une sortie HDMI sur la Raspberry Pi est souvent un avantage, mais parfois aussi un gros problème. Pourtant, il est possible et relativement simple d'utiliser la carte sans écran, à l'aide d'un simple petit périphérique USB à quelques euros...



Il est amusant de remarquer que dans une installation complète d'une Raspberry Pi en tant que poste de travail léger, l'élément le plus coûteux n'est pas la carte Raspberry Pi, mais l'écran. Ceci vaut aussi bien pour un moniteur DVI/HDMI que pour un téléviseur moderne avec entrée HDMI et c'est tant mieux puisque cela nous permet d'avoir plein de Raspberry Pi pour plein d'usages différents. Passons sur le fait que certains moniteurs DVI, utilisés avec un câble HDMI/DVI peuvent poser problème étant donné que DVI et HDMI ne sont pas deux noms différents d'un même standard, mais bel et bien deux standards très proches, ayant chacun plusieurs versions (un excellent article en ligne sur les différences DVI/HDMI/DisplayPort est disponible ici **[1]**).

Pour autant, le trio écran/clavier/souris n'est pas forcément utile systématiquement, en particulier avec les modèles B et B+ qui grâce à leur connectivité Ethernet peuvent être utilisés comme des serveurs (de fichiers, d'impression, de VoIP, etc.). Même les modèles A et A+, équipés d'une clé Wifi se passeront très bien d'écran. Enfin, on peut aisément imaginer qu'un écran lui est bel et bien utile, mais que clavier et souris ne trouvent pas leur place dans un projet (système d'affichage quelconque).

Le cas des modèles A et A+ est intéressant et nous permet de rebondir sur une constatation et une interrogation : parmi les dizaines de milliers d'installations de Raspberry Pi dans le monde, combien sont réellement utilisées en mode graphique ? Car sans mode graphique, la souris est un périphérique qui perd rapidement un quelconque intérêt. Un duo écran/clavier, sans ajout de hub USB, implique presque automatiquement une utilisation en mode texte. Ce qui nous conduit donc à la constatation selon laquelle, le mode texte (ou console) est suffisant pour un grand nombre, sinon la majorité, des usages.

S'il existait un autre moyen d'accéder à ce mode texte, sans avoir recours à l'utilisation de la sortie HDMI, ce serait une économie fort intéressante et surtout une facilité importante pour toute la partie configuration, gestion et installation des applications et utilitaires. Et bien entendu, ce moyen existe. Il en existe même deux. L'un est utilisable au travers du réseau avec SSH ou Telnet (pour les plus téméraires), mais suppose donc, soit un modèle B/B+ avec Ethernet, soit un modèle A/A+ avec ajout d'une interface Wifi ou Ethernet en USB. L'autre moyen en revanche consiste à utiliser ce qu'on appelle une console série, c'est-à-dire une interface permettant de prendre la main sur la ligne de commandes à l'aide d'un adaptateur USB/série et d'une connexion à trois fils (RX, TX, GND) à la Raspberry Pi. C'est cette seconde possibilité qui nous intéresse ici.

1. LA CONSOLE SÉRIE : UN PEU D'HISTOIRE, MAIS PAS TROP

D'après les explications de l'article précédent sur la communication série, ses qualités, ses défauts et son histoire vous pouvez conclure que ce type de connexion était très bien adapté à la transmission de données sur des distances importantes, à une vitesse moyenne et avec une connectique relativement simple. Voilà qui s'adaptera parfaitement à un cahier des charges d'un équipement permettant d'afficher du texte sur un écran de 25 lignes de 80 caractères et d'envoyer des caractères provenant d'un clavier.

Il n'y a pas si longtemps que cela, la puissance de calcul et la mémoire étaient des ressources rares et surtout coûteuses. Il était presque impensable de mettre à la disposition de chaque utilisateur un ordinateur qui lui soit entièrement dédié. Les ressources étaient donc partagées entre les utilisateurs et résidaient à un seul endroit : l'ordinateur central. Ce monstre non seulement encombrant, mais coûteux à l'achat comme à l'entretien était utilisé de concert par des terminaux devant lesquels prenaient place les utilisateurs.

Un terminal IBM 3277 model 2 de 1972, un classique représentant de la gamme 3270 utilisé avec les mainframes IBM comme le classique System/360 (source par Gavin Eadie - licence CC BY-SA 1.0 via Wikimedia Commons).

Initialement composés d'un clavier et d'une imprimante, ces terminaux imprimaient le texte envoyé par l'ordinateur et renvoyait celui saisi au clavier. Cette première génération de terminaux fut appelée téléscripteurs, téléimprimeur ou encore télétype découlant du terme anglais *teletypewriter* qui fut rapidement abrégé TTY pour *TeleTYpewriter.*

L'affichage vidéo débarqua ensuite et remplaça le système d'impression dans les terminaux. Ces *glass TTY* ou VDU pour *Visual Display Units* ne disposaient pas de processeur, mais d'une logique câblée parfois sous la forme de circuits intégrés (LSI) relativement simples.

Ces terminaux étaient naturellement connectés à l'ordinateur central par une liaison série dans une quantité qui devait être supportée par l'architecture. Ceci resta de mise avec l'arrivée des terminaux intelligents comme l'IBM 3270 ou le DEC VT100 qui utilisaient des microprocesseurs, mais uniquement dans le but d'afficher et traiter les données envoyées et reçues. Les liaisons séries utilisées alors étaient soit totalement propriétaires, soit reposaient sur des standards comme RS-232, RS-422 ou RS-423.

La première génération de terminaux avec écran est appelée *dumb terminal* (terminal idiot) du fait du peu d'actions qu'ils étaient capables de traiter (retour à la ligne, etc.). La seconde génération, les *smart terminals* étaient capables de bien plus de choses du fait de la présence d'une certaine puissance de calcul embarquée, comme effacer l'écran, positionner le curseur à un endroit particulier, afficher des caractères avec certains attributs, etc.

C'est ainsi qu'historiquement, les systèmes multi-utilisateurs et multitâches comme UNIX et ses variantes ont toujours été capables de supporter des terminaux communiquant par des liaisons séries. Encore aujourd'hui, c'est une technique omniprésente avec des systèmes comme GNU/Linux et les différents BSD. Plus amusant encore, lorsque vous ouvrez une « fenêtre de terminal » cette notion est bel et bien présente, car l'application que vous utilisez est littéralement un émulateur de terminal. Dans le jardon UNIX, les terminaux, du point de vue du système sont des TTY. On parle également de « console » qui fait référence au terminal principal attaché à l'ordinateur central sur lequel apparaissent les informations de diagnostics (console de contrôle). Avec le temps, le terme console a fini par désigner n'importe quel terminal même si en principe c'est celui de diagnostic.

Les émulateurs de terminaux sont apparus avec l'arrivée des ordinateurs personnels. Pour que ces machines puissent également servir de terminaux, elles faisaient simplement fonctionner une application utilisant le port série pour communiquer avec l'ordinateur central plus puissant et traduisait les informations reçues sur l'écran à la manière d'un véritable terminal. Précisons au passage qu'en utilisant des équipements appelés MODEM, il était possible d'établir une communication avec un système distant au travers d'une ligne téléphonique. Les données séries étaient traduites par le MODEM en information analogique (audibles), passaient dans les lignes de téléphone, puis étaient traduites d'analogique en numérique par un autre MODEM pour arriver sous forme de données séries sur l'ordinateur appelé. Sachez que jusqu'à Windows XP, un émulateur de terminal pouvant communiquer via un port série était livré par défaut : l'HyperTerminal.

Précisons enfin qu'avant l'arrivée massive d'Internet chez le particulier, ce style de méthode de communication était vastement utilisé par les amateurs d'informatique sous la forme de BBS (*Bulletin Board System*). Il s'agissait d'ordinateurs équipés de MODEM reliés à plusieurs lignes téléphoniques faisant fonctionner un logiciel fournissant divers services (messagerie, téléchargement, discussion, annonces, etc.). Beaucoup de termes utilisés aujourd'hui comme *warez*, *gamez*, et toute une série d'acronymes proviennent directement de la culture BBS.

Les terminaux comme tous les périphériques série communiquent des données. Celles-ci doivent, d'un côté comme de l'autre de la communication, utiliser un format identique. Ce qui caractérise ces échanges au niveau logique se résume en plusieurs points :

- la vitesse en nombre de bits par seconde ou bps (300, 1200, 2400, 4800, 9600, 19200, 57600, 115200, etc.),
- la taille des données envoyées en lot les unes à la suite des autres, souvent 8 bits, parfois 7,
- la présence ou non d'un bit supplémentaire ajouté pour valider l'intégrité des bits de données. Cette parité peut être inexistante (N comme *none*), paire (E comme *even*) ou impaire (O comme *Odd*) et le calcul est simple : on compte tous les bits de données soit à 0 soit à 1 ; en fonction du nombre obtenu qui peut être pair ou impair, on ajoute un bit à 0 ou à 1. Si durant la transmission un bit est changé, le bit de parité ne correspond plus et les données sont écartées, car corrompues. Le problème bien sûr est que si plusieurs bits sont changés, l'erreur peut passer inaperçue,
- le nombre de bits de stop marquant la fin d'un paquet de données.

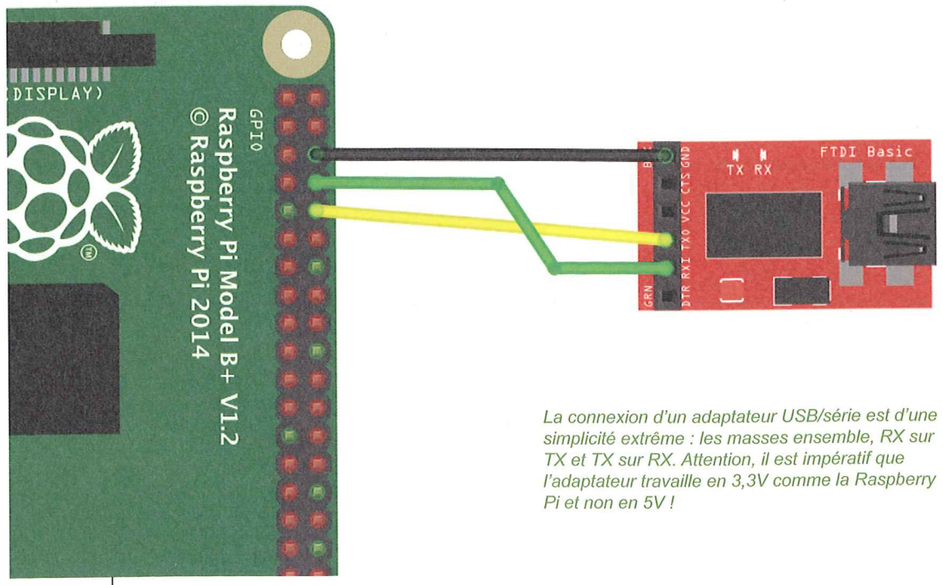
Ces caractéristiques se traduisent généralement sous la forme d'un nombre suivi de trois caractères. Exemple 115200 8N1 pour 115200 bps, 8 bits de données, pas de parité et un bit de stop.

Votre Raspberry Pi, comme toutes les machines capables de faire fonctionner GNU/Linux, est en mesure de permettre la connexion d'un terminal série. Les périphériques dédiés ont presque tous disparu, mais l'option consistant à utiliser un port série et un émulateur de terminal est non seulement réelle, mais aussi très courante.

2. INSTALLATION NOOBS SANS ÉCRAN/CLAVIER

Pour illustrer l'utilisation d'une Raspberry Pi, rien de mieux que de commencer par le début : une installation. En effet, il est parfaitement possible d'installer le système sur une carte sans le moindre écran/clavier connecté. On parle d'une utilisation *headless*. Je ne m'intéresse pas ici à une installation manuelle via une préparation d'image Raspbian sur SD ou microSD avec `dd` ou *Win32 Disk Imager*. Ce serait trop facile puisqu'il n'y a presque rien à faire, le système de base étant configuré pour supporter une console série. Non, je parle ici d'utiliser le système d'installation simplifié NOOBS que nous allons légèrement modifier pour provoquer une installation entièrement automatique. Non seulement *headless* (sans tête), mais aussi sans les mains (*handless* ?). Notez qu'on conserve tout de même l'avantage de l'installation simplifiée NOOBS en évitant de jouer avec des images disque.

Pour procéder à cette installation un peu spéciale, nous récupérons le fichier Zip de NOOBS 1.3.12 (en Torrent, car le téléchargement HTTP est totalement poussif) et décompressons le fichier `NOOBS_v1_3_12.zip` pour obtenir le répertoire `NOOBS_v1_3_12`.



La connexion d'un adaptateur USB/série est d'une simplicité extrême : les masses ensemble, RX sur TX et TX sur RX. Attention, il est impératif que l'adaptateur travaille en 3,3V comme la Raspberry Pi et non en 5V !

Les modifications à apporter ne sont pas nombreuses et visent à procéder à une installation automatisée sans intervention humaine. De ce fait un certain nombre de choix étant proposés dans le processus d'installation normal ne doivent pas exister pour éviter un blocage. La première étape consiste donc à regarder du côté du répertoire `os/` et à supprimer tout son contenu, sauf le répertoire `Raspbian/`. Fut un temps à cet endroit se trouvaient les différents systèmes (RiscOS, Arch, OpenELEC, etc.) pouvant être installés à partir de NOOBS, mais depuis la version 1.3.11, seul Raspbian est possible en l'absence de connectivité Internet lors de l'installation. Cette suppression s'avère donc particulièrement nécessaire si vous installez une ancienne version de NOOBS.

Mais Raspbian se décline en deux « parfums » (*flavours*) : normal ou avec démarrage automatique de l'environnement Scratch. C'est une option d'installation qui ne doit pas rester. Nous modifions donc le fichier `os/Raspbian/flavours.json` avec un éditeur de texte comme Vim, Emacs, Textedit ou Notepad. Avec NOOBS 1.3.12, le fichier ressemble à ceci :

```
{
  "flavours": [
    {
      "name": "Raspbian - Boot to Scratch",
      "description": "A version of Raspbian that boots straight into Scratch",
      "supported_hex_revisions": "2,3,4,5,6,7,8,9,d,e,f,10,11,12,14,1040,1041"
    },
    {
      "name": "Raspbian",
      "description": "A Debian wheezy port, optimised for the Raspberry Pi",
      "supported_hex_revisions": "2,3,4,5,6,7,8,9,d,e,f,10,11,12,14,1040,1041"
    }
  ]
}
```

C'est une syntaxe structurée JSON relativement facile à comprendre. Nous avons deux *flavours* nommés `"Raspbian - Boot to Scratch"` et `"Raspbian"`, énumérés entre `[` et `]` et définis par les lignes entres `{` et `}`. Tout ce que nous avons à faire est de retirer le parfum qui nous dérange et donc finir avec un fichier se présentant ainsi :


```
{
  "flavours": [
    {
      "name": "Raspbian",
      "description": "A Debian wheezy port, optimised for the Raspberry Pi",
      "supported_hex_revisions": "2,3,4,5,6,7,8,9,d,e,f,10,11,12,14,1040,1041"
    }
  ]
}
```

Et non, il ne faut pas retirer le `s` de `flavours`, la grammaire JSON n'a pas à respecter la grammaire des humains (veinarde !).

Nous avons éliminé les choix possibles et nous pouvons maintenant spécifier que l'installation doit se faire automatiquement au démarrage. Pour cela, c'est à la racine du répertoire `NOOBS_v1_3_12` que nous devons éditer le fichier `recovery.cmdline`. À la ligne :

```
runinstaller quiet vt.cur_default=1 elevator=deadline
```

nous ajoutons un espace et la directive `silentinstall`. Cette « installation silencieuse » sous-entend « pas de question ». Copiez le contenu du répertoire `NOOBS_v1_3_12` à la racine d'une carte SD (ou micro SD) fraîchement formatée. Éjectez la carte, placez-la dans la Raspberry Pi et assurez-vous que les connexions à l'adaptateur USB/série soient correctes (GND à la masse (broche 6), RX sur TX (broche 8) et TX sur RX (broche 10)).

Un adaptateur USB/série en 3,3V et trois malheureux câbles pour connecter le tout. Il n'en faut pas davantage pour pouvoir utiliser une Raspberry Pi A+ sans écran et sans réseau.

Côté PC ou Mac, branchez l'adaptateur USB et lancez un émulateur de terminal comme CoolTerm sur Mac OS X, PuTTY sous Windows ou GNU/Linux, ou encore GNU Screen sous GNU/Linux (commande `screen /dev/périphérique 115200`). Notez qu'il est parfaitement possible d'utiliser une autre Raspberry Pi comme émulateur de terminal, mais nous reviendrons sur le sujet.

Quel que soit le logiciel en œuvre, la configuration doit utiliser le bon port série et avec des réglages 115200 8N1, soit une vitesse de 115200 bps, 8 bits de données, pas de parité et un bit de stop.

Branchez l'alimentation de la Raspberry Pi et pour peu que vous n'ayez pas inversé RX et TX (parfois le marquage sur l'adaptateur USB désigne les signaux à l'autre bout de la connexion), ceci devrait apparaître :

```
Uncompressing Linux... done, booting the kernel.

Welcome to the rescue system
recovery login:
```

NE TOUCHEZ À RIEN ! Ce qui vient d'apparaître sur la liaison série n'est qu'une proposition de login et le processus d'installation s'est déjà mis en route. Rien d'autre ne s'affichera sur l'écran, mais nous pouvons voir que quelque chose se passe en observant l'activité des leds de la carte (led verte sur une A+ = activité microSD). La seule chose à faire c'est d'attendre et cela risque de durer entre 15min et une demi-heure selon les performances de la carte SD/microSD.

Une fois l'installation silencieuse terminée, la Raspberry Pi va automatiquement redémarrer sur le système qui s'est installé sur le support. Là, vous devez voir s'afficher une grande quantité de lignes témoignant du démarrage du noyau Linux puis des composants et services du système. Ceci se terminera sur des lignes comme :

```
[    2.532578] devtmpfs: mounted
[    2.538488] Freeing unused kernel memory: 340K (c07a7000 - c07fc000)
[    4.113055] udevd[158]: starting version 175
[    9.611046] EXT4-fs (mmcblk0p6): re-mounted. Opts: (null)
[   10.258892] EXT4-fs (mmcblk0p6): re-mounted. Opts: (null)
Raspbian GNU/Linux 7 raspberrypi ttyAMA0

raspberrypi login:
```

C'est l'invite de connexion sur le nouveau système avec comme nom d'utilisateur par défaut **pi** et comme mot de passe **raspberry**. Après login, le système vous rappelle :

```
NOTICE: the software on this Raspberry Pi
has not been fully configured.
Please run 'sudo raspi-config'
```

On ne voudrait pas fâcher notre carte et on se plie donc volontiers au conseil et on utilise la commande. Notez que l'interface est drastiquement plus simpliste que celle apparaissant en cas d'utilisation d'un écran HDMI.

```
┤ Raspberry Pi Software Configuration Tool (raspi-config) ├

1 Expand Filesystem          Ensures that all of the SD card s
2 Change User Password       Change password for the default u
3 Enable Boot to Desktop/Scratch Choose whether to boot into a des
4 Internationalisation Options  Set up language and regional sett
5 Enable Camera              Enable this Pi to work with the R
6 Add to Rastrack            Add this Pi to the online Raspber
7 Overclock                  Configure overclocking for your P
8 Advanced Options           Configure advanced settings
9 About raspi-config         Information about this configurat

                <Select>                <Finish>
```

L'aspect du script de configuration raspi-config s'adapte tout seul aux fonctionnalités disponibles pour l'affichage. C'est monochrome et simpliste, mais cela fonctionne exactement comme avec un écran HDMI et un clavier.

C'est pourtant là le même programme et le même script qui est utilisé. Un système complexe de gestion de types de terminaux est en place par défaut avec GNU/Linux tout comme avec la plupart des systèmes UNIX. Ainsi, en fonction des fonctionnalités disponibles sur le terminal, la console s'adapte et simplifie la présentation. Ici, par défaut, nous sommes en train d'émuler un terminal de type VT100 comme le montre le résultat de la commande :

```
pi@raspberrypi:~$ echo $TERM
vt100
```

Un VT100 est un terminal très populaire fabriqué par *Digital Equipment Corporation* (DEC) et commercialisé en 1978. Il communiquait via une liaison série avec l'ordinateur et permettait une gestion avancée de l'écran (pour l'époque) en utilisant des séquences d'échappement pour déplacer le curseur, définir des attributs de texte (caractères gras, inversés, soulignés, clignotants), etc. Il devint rapidement populaire et un standard de fait dans le domaine (puis un vrai standard et une norme, ANSI X3.64). On le retrouve donc naturellement comme terminal émulé par défaut.

Vous pouvez utiliser les menus de **raspi-config** exactement comme pour n'importe quelle installation de système Raspbian (voir le numéro précédent du magazine). L'idée est ici de faire le tour des options et d'ajuster tout ce qui est en rapport avec des ressources non utilisées. Je pense en particulier à la mémoire attribuée au GPU (processeur graphique) qui peut être réduite au minimum (16 Mo) laissant quelques 229 Mo de RAM pour le système.

L'ensemble, après redémarrage suite à la reconfiguration, est utilisable exactement de la même manière qu'avec un couple écran+clavier. La seule chose qu'il ne sera pas possible de faire, c'est lancer une interface graphique (**startx**). Dans l'absolu cependant, celle-ci démarrera effectivement, mais utilisera les ressources graphiques disponibles et donc le GPU et la sortie HDMI. Tout ce que vous verrez sur le terminal série, ce sont les messages de la commande (que généralement vous ne voyez pas).

Voici un vrai VT100 de la fin des années 70. Dans l'absolu, si vous disposez d'une telle pièce de collection, vous pouvez parfaitement l'utiliser comme terminal pour votre Raspberry Pi, mais à une vitesse maximum de 19200 bps, non 115200. (source : "DEC VT100 terminal" par Jason Scott - licence CC BY 2.0 via Wikimedia Commons)

3. CONFIGURATION DE LA LIAISON SÉRIE

La console série est gérée au niveau noyau et la notion de terminal au niveau système. Rappelez-vous que la console est en principe l'affichage de diagnostics du système. Sur votre Raspberry Pi, cette configuration trouve place dans les arguments passés au noyau au démarrage. C'est donc dans le fichier `/boot/cmdline.txt` (`/boot` est le point de montage de la partition FAT de la carte SD/microSD) que se trouvent ces informations (sur une ligne) :

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200
console=tty1 root=/dev/mmcblk0p6
rootfstype=ext4 elevator=deadline rootwait
```

C'est la mention `console=ttyAMA0,115200` qui précise au noyau qu'il doit afficher ces informations sur le périphérique `ttyAMA0` à la vitesse de 115200 bps. On notera également qu'une seconde mention de cette option est présente avec `console=tty1`. `tty1` est ici le terminal constitué du duo clavier/écran. On a donc ici deux consoles, une série et une sur la sortie HDMI. Remarquez au passage qu'il s'agit de deux terminaux séparés. Si vous branchez simplement un clavier à la Pi, vous n'aurez pas de retour sur le terminal série. Les terminaux ne se mélangent pas.

La gestion du terminal en tant que telle est l'affaire du système, et ce dès le démarrage. C'est à la fin du fichier `/etc/inittab` que vous trouverez sa configuration avec la ligne :

```
T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

Ceci signifie que sous l'identifiant `T0` et pour les niveaux d'exécution 2 et 3, nous lançons le programme `getty` au démarrage (`getty` pour *get teletype*). Si celui-ci se termine, il est relancé (`respawn`). Notre commande `getty` prend en argument le périphérique à utiliser (`ttyAMA0`, le premier port série de la Pi), la vitesse de communication et le type de terminal dont il s'agit. Hé oui, voilà notre bon vieux VT100 ! C'est ce programme `getty` qui vous invite à vous connecter juste après le démarrage et c'est également lui qui se charge de cela sur le terminal écran/clavier (`tty1`). Un peu plus haut dans le fichier on trouve en effet :

```
1:2345:respawn:/sbin/getty --noclear 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6
```

Six `getty` ? Vous ne le saviez peut-être pas, mais ce n'est pas un terminal qui est disponible avec le duo écran/clavier, mais six. Vous pouvez en effet basculer de l'un à l'autre en utilisant les touches [Alt] + [F1] à [Alt] + [F6]. Notez que si vous n'utilisez pas cette fonctionnalité, vous pouvez parfaitement commenter ces lignes (ou du moins 5 d'entre elles) en les faisant précéder d'un `#`. Vous économiserez ainsi un peu de précieuse mémoire vive.

Comme vous pouvez le constater, il n'y a rien à configurer puisque tout est déjà en place par défaut. Mais, comme en cuisine, il est toujours bon de savoir où se trouvent les choses dont on se sert régulièrement.

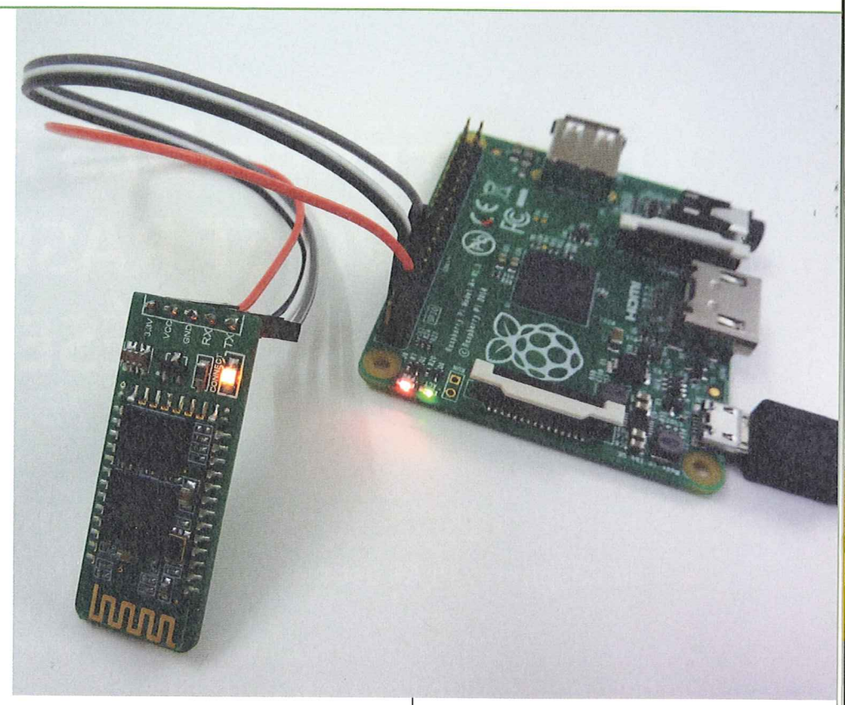
4. RÉCUPÉRER LE PORT SÉRIE POUR AUTRE CHOSE

Cet article présente l'utilisation du terminal série, mais c'est aussi l'occasion de détailler comment ne pas s'en servir. Ceci permettra de récupérer le port série de la Raspberry Pi pour un autre usage, comme connecter un périphérique série ou, pourquoi pas une autre Raspberry Pi.

Pour déconfigurer cette gestion, il nous suffit de modifier les deux fichiers dont nous venons de parler. Retirer toute mention de `ttyAMA0` dans `/boot/cmdline.txt` et commenter la ligne `T0` de `/etc/inittab`. Ceci nécessitera un redémarrage puisque les arguments passés au noyau changent.

Ceci rendra alors utilisable `/dev/ttyAMA0` pour tous vos futurs projets. Notez cependant qu'un petit problème existe avec la Raspberry Pi, car lors de l'ouverture du port une impulsion avec une tension négative apparaît sur TX durant 32 µs. Ce signal peut être interprété par le périphérique série que vous avez connecté comme une donnée et conduire à des comportements inattendus. C'est un problème matériel connu sur la plateforme et une manière de le contourner consiste à utiliser une broche GPIO comme signal RTS (*Request To Send*) qui est un signal de contrôle de flux hérité de RS-232. Ceci revient à demander au périphérique série (qui doit être compatible RTS/CTS) de ne pas prendre en compte les données sauf si RTS est activé.

Ceci est relativement délicat à mettre en place, technique et nécessaire pour contourner un bug pénible. Dans bien des cas, on choisira de conserver la configuration du terminal série dans l'état et de simplement ajouter un adaptateur USB/série à la Raspberry Pi. À cette date, faute de Raspberry Pi 2 sous la main pour l'instant, nous ne savons pas si ce problème est encore présent sur la nouvelle génération de carte (qui ne devait arriver en principe qu'en 2017, n'est-ce pas Eben ?). DB



En utilisant un module RFCOMM Bluetooth en lieu et place d'un adaptateur USR/série, il devient possible de rendre la console de la Raspberry Pi accessible depuis n'importe quelle machine ou smartphone équipé d'une connectivité Bluetooth. La partie délicate concerne uniquement la configuration du module qui est totalement indépendant de la configuration de la Pi. Nous aborderons cela dans un prochain numéro.

TRANSFÉRER DES FICHIERS ENTRE PC ET RASPBERRY PI SANS RÉSEAU

Denis Bodor

La Raspberry Pi A+ est très sympathique. Petite, peu gourmande en énergie et économique, elle peut être la base d'un projet ne nécessitant pas de connectivité réseau. En effet, point d'Ethernet sur la bête et le seul malheureux port USB ne possède pas de hub puisqu'il est directement relié au SoC Broadcom et donc utilisé prudemment. La question est : comment transférer des fichiers ?

Vous savez ce qui est absolument génial lorsqu'on est rédacteur en chef d'un magazine en dehors d'avoir le privilège de peu dormir et de rédiger plein, plein, plein d'articles ? On peut changer d'avis en ce qui concerne ce qu'on explique. Dans l'article sur les adaptateurs USB, je concluais en disant que les protocoles de transfert comme Zmodem étaient devenus désuets. Mais une petite voix dans ma tête me disais « Deniiiiis essaye » (sans doute la voix d'un des géants défunts sur les épaules desquels je me tiens humblement). Quelques minutes plus tard, quelques voisins inquiets se sont sans doute demandé qui était le fou qui venait de crier si tard le soir « mais c'est énorme ! ». Ainsi, voici un petit ajout complémentaire parce que... c'est juste énorme.

1. EN AVANT !

Précisons le contexte puisque dans mon excitation je n'ai pas pris la peine de tester avec d'autres outils que ce cher GNU Screen sous GNU/Linux. On parlera donc ici du fait d'utiliser GNU Screen pour établir une liaison série avec une Raspberry Pi, idéalement une A+ sans réseau. Dans cette situation, comment transférer un fichier de quelques 350 Ko du PC sur la Pi ? La réponse est Zmodem !

À ma grande surprise, le protocole Zmodem qui permet d'échanger des fichiers au travers d'une liaison série comme au bon vieux temps des BBS s'avère presque directement pris en charge par GNU Screen. Je l'ai appris en tombant sur le blog de Adam Monsen (**adammonsen.com**) et en particulier son billet sur ses recherches consistant à arriver à transférer des fichiers via Zmodem avec Screen, mais sur une connexion SSH et en ignorant délibérément `scp` (diantre, il y a des gens plus tordus que moi). Donc merci du fond du cœur Adam, vous avez illuminé ma soirée.

Côté PC, vous n'avez besoin de rien si ce n'est GNU Screen et éventuellement installer le paquet `lrzsz` s'il ne l'est pas par défaut. Côté Raspberry Pi en revanche, il nous manque une sacrée brique. Nous n'avons pas par défaut les outils nécessaires pour procéder

aux transferts. Comme nous n'avons pas de réseau, pas question d'installer un paquet Raspbian avec `apt-get`. Nous contournons le problème en retirant la microSD de la Pi et en la plaçant dans le PC. Nous pointons ensuite notre navigateur sur **http://mirrordirector.raspbian.org/raspbian/** et en particulier dans le sous-répertoire `pool/main/l/lrzsz/` pour télécharger le fichier `lrzsz_0.12.21-5_armhf.deb`. C'est le paquet qu'il nous faut installer. Nous enregistrons ce fichier directement sur la microSD dans la partition FAT (qui apparaît en `/boot` sur la Pi). Oui, nous venons en quelque sorte de transférer un fichier du PC vers la Pi, mais ce n'est pas le propos, nous ne voulons pas non plus éteindre la Pi à chaque transfert.

On replace la microSD dans la Raspberry Pi et on la démarre pour s'y connecter, donc tout naturellement avec `screen /dev/ttyUSB0 115200`. Une fois connecté sous l'utilisateur `pi`, nous devons installer le paquet, non pas avec `apt-get`, mais un autre utilitaire sous-jacent, `dpkg` :

```
$ sudo dpkg -i /boot/
lrzsz_0.12.21-5_armhf.deb
Sélection du paquet lrzsz
précédemment désélectionné.
[...]
Paramétrage de lrzsz (0.12.21-5) ...
Traitement des actions différées
(« triggers »)
pour « man-db »...
```

Nous disposons maintenant, entre autres, de l'outil `rz` comme *receive zmodem*. La magie peut opérer. Avant toutes choses, nous devons signifier à Screen qu'il doit gérer les « messages » Zmodem automatiquement. Il doit donc « attraper » des séquences de données typiques d'un transfert Zmodem. Pour cela, nous utilisons son mode de commande interactif avec le raccourci clavier [Ctrl] + [A] suivi de `:`. Une invite au bas de l'écran débutant par ce même caractère vous permet de taper des commandes. Nous saisissons `zmodem catch` et validons. GNU Screen nous confirme en affichant au bas de l'écran `zmodem mode is catch`.

Dès lors, dès qu'il verra un début de transaction Zmodem, GNU Screen s'en chargera et éventuellement nous demandera notre avis.

Comme nous avons la main sur le système de la Raspberry Pi, nous pouvons lancer directement l'utilitaire de réception avec la commande `rz` sans aucun argument. Une transaction Zmodem est engagée, GNU Screen réagit et nous demande le chemin et le nom du fichier à envoyer en nous affichant au bas de l'écran :

```
:!!! sz -vv -b
```

Ce n'est pas le résultat d'une éruption solaire, ni même un message dans une langue morte oubliée. GNU Screen se propose tout simplement d'utiliser la commande `sz` (*send zmodem*) avec les paramètres `-zz` et `-b` et nous invite à fournir un nom de fichier. Il nous suffit de le préciser et de valider. Et le transfert débute :

```
rz waiting to receive.**B0
Sending: 52903.jpg
Bytes Sent:  61440/ 362241  BPS:11877  ETA 00:25
```

puis se termine :

```
Transfer complete
```

Nous venons de transférer les 362241 octets de notre fichier qui se trouve à présent sur la Raspberry Pi. La preuve :

```
$ ls -l 52903.jpg
-rw-r--r-- 1 pi pi 362241 mai 3 2012 52903.jpg
```

Pour nous assurer que tout est en ordre et que les fichiers sont identiques (même si Zmodem dispose de mécanismes de vérification et de correction), nous n'oublions pas de vérifier la somme de contrôle MD5 avec :

```
$ md5sum 52903.jpg
80e914a1f77bb07d3d65d8202e8eac66  52903.jpg
```

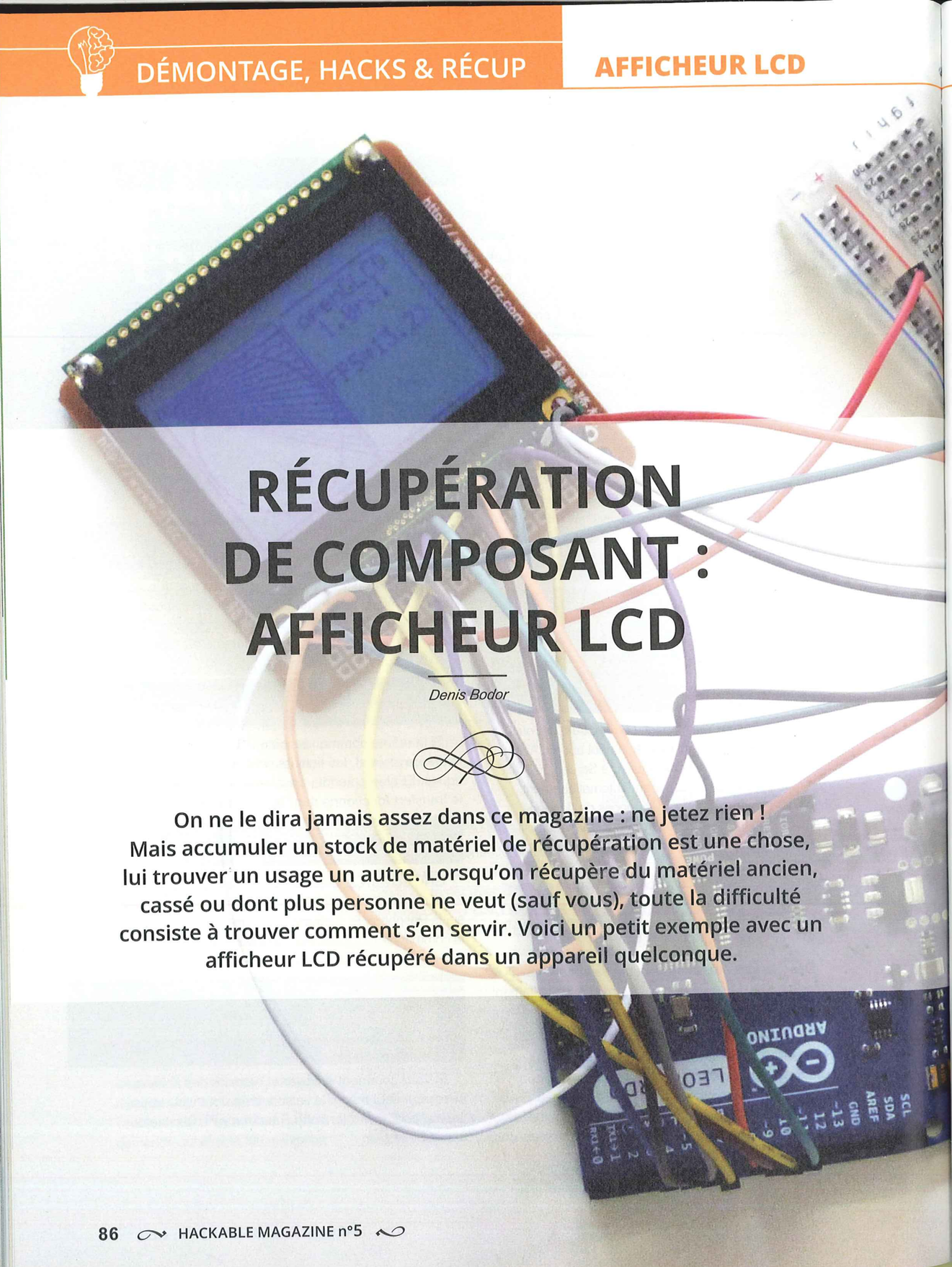
Si la même commande, côté PC, nous affiche le même résultat, les fichiers sont strictement identiques. Et bien entendu, c'est le cas. En toute logique, le transfert fonctionne dans les deux sens :

```
$ cp 52903.jpg toto.jpg
$ sz toto.jpg
```

GNU Screen est toujours en mode `zmodem catch` et le simple fait d'invoquer la commande `sz` (*send zmodem*) sur la Raspberry Pi déclenche une proposition d'utiliser la commande `rz` localement. En revanche, nous n'avons rien d'autre à faire que valider pour recevoir `toto.jpg` :

```
:!!! rz -vv -b -E

Receiving: toto.jpg
Bytes received:  154624/ 362241 BPS:11099  ETA 00:18
Transfer complete
```

Et voici comment envoyer et recevoir des fichiers avec pour seul média la liaison série dont vous vous servez pour contrôler votre Raspberry Pi. C'est juste énorme ! DB



RÉCUPÉRATION DE COMPOSANT : AFFICHEUR LCD

Denis Bodor

On ne le dira jamais assez dans ce magazine : ne jetez rien ! Mais accumuler un stock de matériel de récupération est une chose, lui trouver un usage un autre. Lorsqu'on récupère du matériel ancien, cassé ou dont plus personne ne veut (sauf vous), toute la difficulté consiste à trouver comment s'en servir. Voici un petit exemple avec un afficheur LCD récupéré dans un appareil quelconque.

Le principe est simple, dès qu'on trouve un matériel un tant soit peu électronique, on le conserve précieusement jusqu'au moment où on lui choisira un destin. Tantôt, le matériel peut être réparé, tantôt on peut le détourner de son usage et enfin, on peut le désosser pour en extraire ce qui pourrait être réutilisable de manière générique. Ceci comprend généralement tout un tas de composants passifs comme résistances ou condensateurs, mais également des éléments plus « travaillés » comme des connecteurs, des claviers, des imprimantes thermiques, ou des composants d'affichage.

Ce dernier point est l'un des plus intéressants, car ces petites choses se trouvent partout et peuvent s'avérer très coûteuses à l'achat. De plus, en dehors de cas très spécifiques comme les calculatrices, les horloges, les thermomètres et tout appareil très compact, les afficheurs utilisés sont standards. En effet, tout comme avec les afficheurs LCD alphanumériques de 2×16 caractères par exemple, une sorte de compatibilité se met naturellement en place : un constructeur domine le marché à un moment, son protocole devient presque incontournable, d'autres le copient et 80% des modules d'affichage de ce type sont donc majoritairement compatibles. C'est le cas du contrôleur HD44780 développé par Hitachi pour ce type d'écran « texte » dont le fonctionnement a joyeusement été repris par bon nombre de constructeurs (Sunplus SPLC780A1, Sitronix ST7066, Samsung KS0066), au point que l'on parle désormais de « compatibles HD44780 ».

L'élément de récupération qui nous occupe aujourd'hui est également un afficheur LCD, mais d'un format bien différent. Celui-ci a été récupéré et dessoudé sur ce qui semblait être à l'évidence un appareil bancaire de paiement par carte (on trouve de drôles de choses pour 5€ aux marchés aux puces). Dans l'appareil, il y avait non seulement cet écran, mais également une imprimante thermique, un support pour lecture de cartes à puce, un pad numérique, une circuiterie d'alimentation... Une partie des composants ne sont pas directement utilisables ou demanderaient un travail conséquent de recherche à la réussite peu probable. Le microcontrôleur ou du moins le composant « intelligent » de la machine n'est pas un modèle courant, ce qui n'est en rien étonnant pour ce type de périphérique. Si d'aventure l'idée de jouer au petit malin avec ce type d'appareil vous est passée par l'esprit, empressez-vous de la jeter rapidement aux orties. Non seulement un ensemble de mécanismes efface les données importantes au démontage (dispositif *anti-tamper*) et en plus, en toute franchise, s'il y a bien un secteur dans lequel il ne faut pas jouer au plus malin c'est bien celui de la banque (entre autres). L'idée n'est pas de chercher les petits secrets de l'appareil, mais bien de réutiliser les éléments qui s'y trouvent.

L'objet de cet article est de vous fournir quelques conseils à propos de ce qui peut s'apparenter à un jeu de piste ou une chasse au trésor. Nous prenons ici l'exemple de cet écran LCD de récupération, mais ceci s'appliquera de la même manière à n'importe quel composant, puce ou module extrait sauvagement d'un appareil soi-disant bon à jeter.

1. PHASE 0 : COLLECTE ET EXTRACTION

Ceci n'est pas, dans le cadre de cet article, à proprement parler une étape. En effet, en ce qui me concerne, dès acquisition d'un équipement d'occasion ou cassé (voire neuf, « *vous voulez une extension de garantie ?* », « *non, je vais le démonter* » a toujours un effet hilarant sur un vendeur Fnac), je m'empresse généralement de l'ouvrir et de l'inspecter à la recherche de composants recyclables. Si l'objet n'est clairement pas utilisable dans l'état et ne nécessite donc pas de planification pour une réparation, il est directement désossé. Les éléments pouvant « servir un jour » sont dessoudés, triés et conservés. Les choses qui répondent au syndrome « je ne sais pas pourquoi, mais je ne peux pas le jeter » finissent dans la « boîte à trucs pour plus tard ». Il s'agit non pas d'une boîte, mais de plusieurs bacs accueillant les circuits, les câbles, les vis, les parties mécaniques... qui trônent à la cave ou au grenier. Ils sont, en quelque

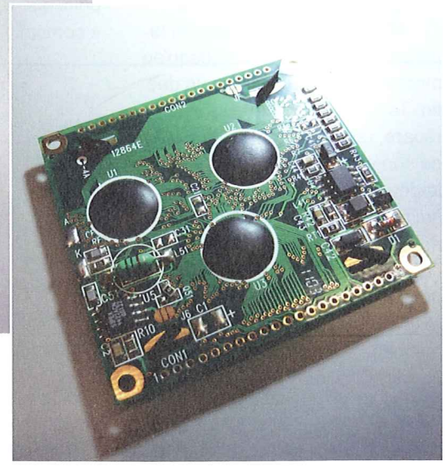
AFFICHEUR LCD

Après extraction du module d'affichage, il est prêt pour un petit nettoyage.

sorte, en quarantaine jusqu'au moment où une expédition s'avère nécessaire parce que « je crois me souvenir que j'ai un truc du même genre quelque part ».

Les éléments de valeur, quant à eux, le plus souvent des composants génériques, prometteurs et facilement identifiables passent alors à l'étape du dessoudage propre et du nettoyage. L'opération a pour but de les débarrasser de l'éventuelle crasse/poussière qui les surcharge,

Le module LCD nettoyé avec son connecteur maintenant facilement utilisable. En dehors de quelques rayures, il est presque comme neuf. Notez l'étiquette qui s'est décollée pendant l'opération, surtout ne pas la perdre (ou faire une photo).



L'arrière du module LCD. On distingue trois gros points noirs : ce sont les circuits intégrés chargés de l'interfaçage et du pilotage de l'écran. Sous la résine noire se trouvent directement les puces. Cette technique est appelée Chip-on-Board ou CoB.

et de surtout bien nettoyer les pattes et connecteurs, généralement à l'aide de tresse à dessouder, de flux et de l'indispensable fer à souder.

En ce qui concerne les connecteurs, en particulier si le module n'était pas simplement enfiché, il s'agit de retirer suffisamment d'étain de manière à pouvoir souder soit un connecteur, soit directement des fils. Généralement, après dessoudage, de l'étain obstrue les trous et empêche d'y glisser un connecteur (mâle ou femelle) correctement. La tresse à dessouder, imbibée de flux, s'avère très efficace. Le but est de chauffer à la fois la tresse et le connecteur de manière à ce que l'étain fonde et soit aspiré par capillarité dans la tresse.

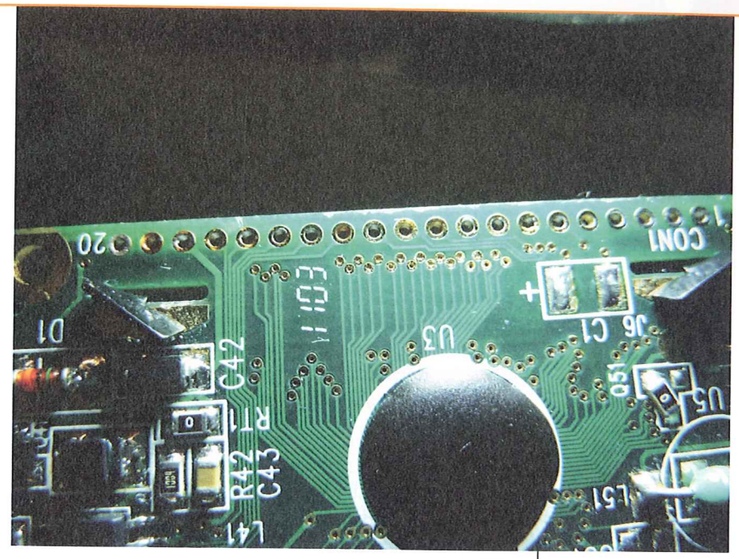
L'utilisation de la pompe à dessouder, permettant d'aspirer d'un coup l'étain fondu fonctionne

également, mais je dois avouer que je préfère réserver cette technique pour les composants aux connecteurs plus importants (bornes de connexion 230V, transformateurs, relais, etc.). Le dessoudage est une activité complète et il existe bien des techniques, des outils et des produits permettant de faciliter ce type de manipulation. À chacun de trouver la formule qui lui convient le mieux en fonction de sa dextérité et de ses moyens.

2. PHASE 1 : OBSERVATION, IDENTIFICATION ET RECHERCHE

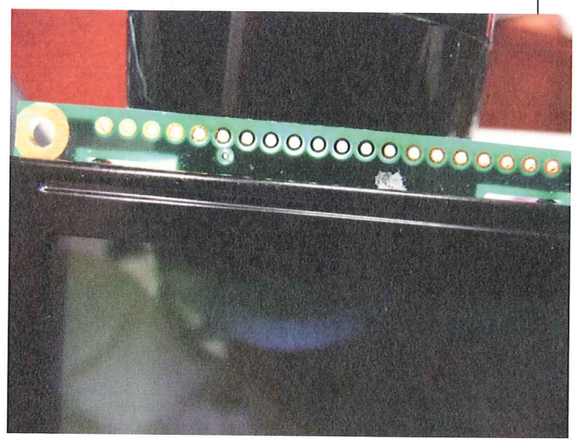
À présent que tout est propre, nous pouvons passer aux choses sérieuses et tenter de répondre à la question la plus capitale : qu'avons-nous entre les mains ?

De par son aspect et l'appareil d'où il provient, il s'agit clairement d'un afficheur LCD graphique. Une inspection de l'arrière du module nous montre qu'il est très certainement rétroéclairé, car deux connecteurs sur le côté sont soudés entre l'écran et le circuit, ils sont marqués « A » et « K », comme « anode » et « cathode ». Une grosse résistance et une puce marquée « SIPEX 4422ACM », couplée à une petite recherche sur le web, nous apprennent que le rétroéclairage est électroluminescent. Cette puce permet l'alimentation du rétroéclairage. C'est une bonne et une mauvaise nouvelle.



Les 20 broches du connecteur CON1 sont reliées à des pistes qui « sautent » d'un côté à l'autre du circuit imprimé par des « vias » et arrivent sur U3. C'est clairement une interface pour piloter le module.

La mauvaise est que ce type de système est relativement ancien. C'était une technique permettant d'obtenir un éclairage uniforme avant que les leds et les matériaux de diffusion de la lumière ne gagnent en efficacité (souvenez-vous du PalmIIIxe). La bonne nouvelle c'est que ce type de technique nécessite une alimentation par un courant alternatif et des tensions relativement importantes, mais que ceci est justement pris en charge par la puce Sipex. Nous n'aurons donc pas à nous en soucier. On notera que la puce en question est également marquée « 0122 » qui est très certainement un code pour la date de fabrication : 2001, 22ème semaine. Les « A » et « K » s'expliquent également malgré l'éclairage électroluminescent, il doit exister une déclinaison à leds du produit.



Le second connecteur ne semble pas utilisé pour d'autres raisons que purement mécaniques.

Le module dispose de deux séries de connecteurs libellés CON1 et CON2. Le premier est certainement l'interface de connexion pour piloter le module. Nous avons 20 broches, ce qui n'est pas sans rappeler les afficheurs LCD alphanumériques compatibles HD44870. Il est peu probable que la liaison se fasse de manière sérielle. Autant de connecteurs suggèrent une liaison parallèle, ça va être la fête aux câbles...

CON2 n'est relié à rien. Cependant, on ne peut se satisfaire d'un simple coup d'œil, car nous ne savons pas combien de couches composent réellement le circuit imprimé. Une bonne technique pour cela consiste à placer une source de lumière sur la tranche du circuit. Il s'en trouve éclairé de l'intérieur et on pourra alors réellement s'assurer que le connecteur ne dispose d'aucune piste reliée.

Maintenant, nous savons que nous devrons nous interfacer à 20 broches pour piloter la bête et nous avons une chance qu'il s'agisse d'une connectique et d'un protocole connu. Il est temps de faire nos devoirs et de mener l'enquête sur le web.

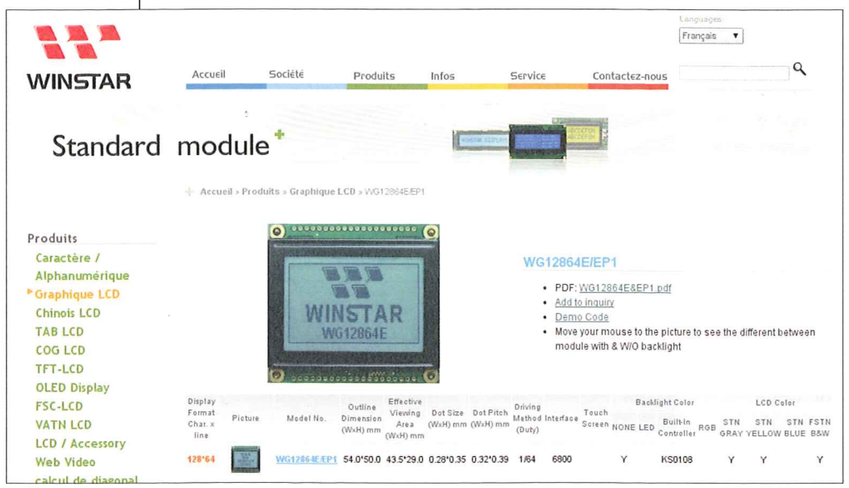
3. PHASE 2 : DOCUMENTATION

Nous avons de la chance, une étiquette était placée sur le module et nous fourni une piste de valeur : « WINSTAR WG12864E-BGI-V*D » (avec une date de fabrication au 14 août 2003 semble-t-il). Une recherche d'une partie de ce texte nous emmène directement sur le site du constructeur : **http://www.winstar.com.tw**, et en particulier sur la page de présentation du composant.

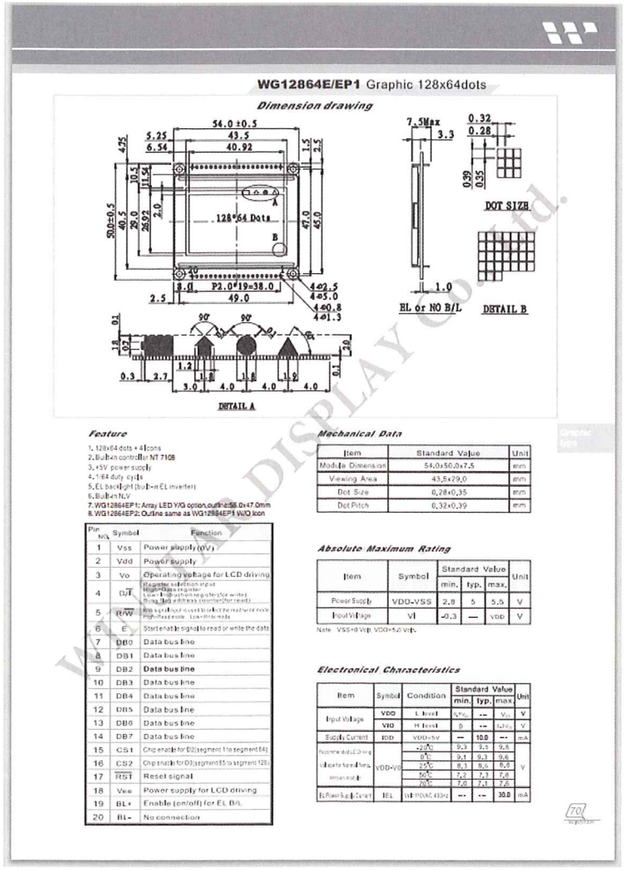
Nous y apprenons beaucoup de choses et surtout nous avons accès à la documentation, le fameux (ou la fameuse) *datasheet* : le descriptif technique, mécanique et électrique du composant. Nous avons là une grande partie des informations absolument indispensables pour pouvoir alimenter et faire fonctionner l'écran.

Sur la seule page qui constitue la documentation nous apprenons ainsi que :

- l'écran affiche 128×64 pixels (plus 4 icônes),
- le contrôleur embarqué est un NT 7108,
- le module s'alimente en 5V,
- le rétroéclairage est inclus comme on s'en doutait,



Une chance, l'inscription sur l'étiquette nous conduit directement sur le site du fabricant. Hé oui, c'est bel et bien notre module que nous avons là en photo.



Le document du constructeur est succinct, mais nous donnera des éléments indispensables.

- il n'est pas nécessaire de fournir une tension négative (*built-in N.V*) pour le rétroéclairage,
- le module utilise 10mA en 5V,
- le rétroéclairage a besoin de 30mA.

Mais, plus important encore, nous trouvons la nomenclature pour les broches de CON1. Alimentation, masse, réglage du contraste, bus de données, lignes de contrôle, rétroéclairage... tout y est.

En poursuivant les recherches sur le web, on tombe sur la documentation de quelques 25 pages du contrôleur NT 7108 de *Neotec semiconductor*, sur le site d'un autre fabricant, Crystalfontz. Cette documentation détaille tout le protocole de communication avec le contrôleur ou, en d'autres termes, la façon de lui parler pour afficher ce qu'on veut sur l'écran.

À ce stade, il est intéressant de remarquer et surtout de vous conseiller d'avoir pris suffisamment de café avant de partir en chasse de ce genre d'informations. Ainsi, si j'avais été moi-même plus attentif, j'aurai très certainement économisé un certain temps. En effet, le NT 7108 ne semble pas très connu et c'est via des recherches passablement alambiquées que je suis finalement tombé sur un message du 12 mai 2011, d'un certain « om0tm », sur les forums Arduino précisant : « *Definitely is NT7108 fully compatible with KS0108* » (« NT7108 est clairement compatible KS0108 »), en plus de la confirmation du fait qu'il ait réussi à utiliser ce contrôleur avec une carte Arduino Mega1280.

Cette voie détournée fut une perte sèche de temps car, même si la documentation PDF de Winstar précise le NT 7108, la page de description du produit, elle, signale « *Built-in Controller : KS0108* ».

Pourquoi ce détail est-il important ? C'est simple, le KS0108 est l'équivalent pour les afficheurs LCD graphiques du HD44780 pour les écrans alphanumériques. Vastement utilisé et copié, le KS0108 est un standard généralement pris en charge de longue date tant les modules l'utilisant sont nombreux. La chance est avec nous, la probabilité de réussir à faire fonctionner ce module est de plus en plus importante.

POURQUOI LA COMPATIBILITÉ EXISTE-T-ELLE ?

La question est légitime, car il est tentant de penser qu'en tant que constructeur, il est préférable de se distinguer et, implicitement, inciter ses clients à toujours utiliser ses produits. Mais les clients ne sont pas idiots. Lorsqu'on parle de fabrication industrielle, il ne s'agit pas simplement de produire une centaine d'appareils. Non seulement le plus souvent cela se chiffre en dizaine de milliers d'exemplaires, mais il faut également prévoir des évolutions, des réparations et des remplacements.

En choisissant, par exemple, un module d'affichage LCD compatible KS0108 ou HD44780, le client pourra le remplacer par un autre modèle, également compatible. Ceci se fait pour réduire les coûts ou simplement parce que les quantités immédiatement disponibles auprès d'un fournisseur ne sont pas suffisantes.

En prenant soin de se baser sur un élément standard avec plusieurs fournisseurs « compatibles », le développement du code, son évaluation et sa validation ne sont à faire qu'une seule fois. Pour peu que les fournisseurs assurent une réelle compatibilité, le client n'aura pas à recréer tout son produit pour une simple rupture de stock.

4. PHASE 3 : À LA RECHERCHE DE CODE

Deux mots d'ordre doivent toujours être gardés à l'esprit lorsqu'il s'agit de produire un code, non seulement pour Arduino, mais pour n'importe quel autre système ou machine :

- ne pas réinventer la roue,
- reposer sur l'épaule des géants.

En d'autres termes, avant de s'attacher à la lecture et au décodage d'une documentation comme celle du NT 7108 (et encore, 25 pages c'est peu) et à une longue série de boucles essais/échecs, il est important de voir si quelqu'un, quelque part, n'aurait pas déjà fait ce travail et diffusé le résultat sous une licence open source. Le pilotage à proprement parler d'un contrôleur compatible KS0108 consiste en grande partie à lui envoyer des données et des commandes via les huit lignes D0 à D7. Le tout en manipulant les autres signaux de manière adéquate.

Nous partons donc à la recherche d'une bibliothèque pour ce faire et la première étape nous mène sur l'Arduino *Playground* et en particulier sur la page présentant la *KS0108 Graphics LCD library* (**http://playground.arduino.cc/Code/GLCDks0108**).

Fini ? Certainement pas. Le réflexe ici n'est surtout pas de se contenter de la première trouvaille venue. Il faut faire le tour des options qui s'offrent à nous et n'aviser qu'ensuite. Réagir trop promptement c'est prendre le risque de suivre une documentation, procéder à une installation, étudier la connexion Arduino/module... bloquer sur la compilation du croquis exemple, déverminer le croquis, puis la bibliothèque, le tout avant de se rendre compte que le code n'est plus maintenu depuis des lustres, que son auteur n'a jamais considéré son œuvre autrement que comme un simple essai et qu'une réalisation équivalente, mais finie existe et est largement utilisée avec succès.

La page du site Arduino détaille la mise en œuvre de la bibliothèque tout en précisant que la version 3 date de juin 2012, mais que l'auteur principal a débuté un autre projet, OpenGLCD, offrant davantage de fonctionnalités et une compatibilité avec plus d'afficheurs.

La page OpenGLCD (**https://bitbucket.org/bperrybap/openglcd**) nous apprend alors que le développeur en question, Bill Perry (alias *bperrybap*), n'a pas chômé et a fait évoluer sa création jusqu'à la version 1.0rc1 (« rc » pour *Release Candidate*, une pré-finale en somme) diffusée depuis le 23 novembre 2014 (c'est tout frais).

Touchons un mot au passage à propos des numéros de versions. Utilisateur de longue date (bientôt 20 ans) de logiciels et systèmes en logiciel libre, je suis très attaché à l'usage raisonné des numéros de versions. Je vois généralement d'un très mauvais œil des successions de type V1, V2, V3... sur des durées de quelques semaines.

Board Type	D_I	R_W	EN	D0	D1	D2	D3	D4	D5	D6	D7	CSEL1	CSEL2	CSEL3*
Uno	A3	A2	A4	8	9	10	11	4	5	6	7	A0	A1	3*
Leonardo	A3	A2	A4	8	9	10	11	4	5	6	7	A0	A1	3*
Mega	36	35	37	22	23	24	25	26	27	28	29	33	34	32*
Teensy 2.0	5	6	9	0	1	2	3	13	14	15	4	7	8	(no default)*
Teensy ++ 2.0	9	8	7	10	11	12	13	14	15	16	17	18	19	(no default)*
Teensy 3.0/3.1	9*	8*	7*	16*	17*	18*	19*	20*	21*	22*	23*	3*	2*	(no default)*
Mighty1284p	27	26	28	0	1	2	3	4	5	6	7	24	25	29*
Sanguino	27	26	28	0	1	2	3	4	5	6	7	24	25	29*
Bobuino	A3	A2	A4	8	9	10	11	4	5	6	7	A0	A1	3*
SleepingBeauty	A3	A2	A4	8	9	10	11	4	5	6	7	A0	A1	3*
chipKIT	A3	A2	A4	8	9	10	11	4	5	6	7	A0	A1	3*
DUE (NOT SUPPORTED)	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Arduino pin to KS0108 Pin Function mapping Table

La documentation de la bibliothèque OpenGLCD détaille les connexions avec plusieurs modèles de cartes Arduino et compatibles.

Les programmes open source (ou pas) sont comme le vin ou le fromage, il faut du temps, des tests et beaucoup d'efforts avant d'avoir une réalisation « consommable » légitimement numérotée 1.0. Un programme ou une bibliothèque commence donc en version 0.1 (voire 0.0.1). À ce stade, il est déjà en mesure de faire ce pour quoi il est conçu, mais l'auteur sait parfaitement qu'il reste du travail avant de considérer le résultat comme stable. C'est cette humilité qui transparaît dans une numérotation de versions, et avec elle, la sagesse et l'expérience du développeur. Ainsi, comme moi, méfiez-vous des versions autoproclamées V1 ou 1.0 sans aucun passé.

La bibliothèque OpenGLCD supporte actuellement trois types de contrôleurs LCD (et compatibles) : hd44102, sed1520 et KS0108. Étant donné le nombre de broches utilisées, un brochage par défaut existe pour plusieurs cartes Arduino et compatibles. Un coup d'œil à la documentation de la bibliothèque (`openGLCD/doc/html/page_ks0108_family.html`) nous apprend ainsi que les modèles Uno, Leonardo et Mega sont supportés ainsi que des cartes compatibles comme Sanguino, Bobuino ou encore chipKIT.

Mais le plus important réside dans les fonctionnalités de la bibliothèque. OpenGLCD n'est pas le seul morceau de code à prendre en charge un écran compatible KS0108, mais il offre bien plus que le simple fait d'allumer ou éteindre un pixel sur les 8192 disponibles. OpenGLCD fournit des primitives graphiques ou, en d'autres termes, des fonctions permettant de :

- tracer des formes (lignes, courbes, rectangles, triangles, boites arrondies, cercles, etc.),
- dessiner des *bargraphs* et des jauges,
- afficher des dessins (*bitmap*),
- écrire du texte,
- définir des zones pour contenir le texte,
- formater des chaînes de caractères...

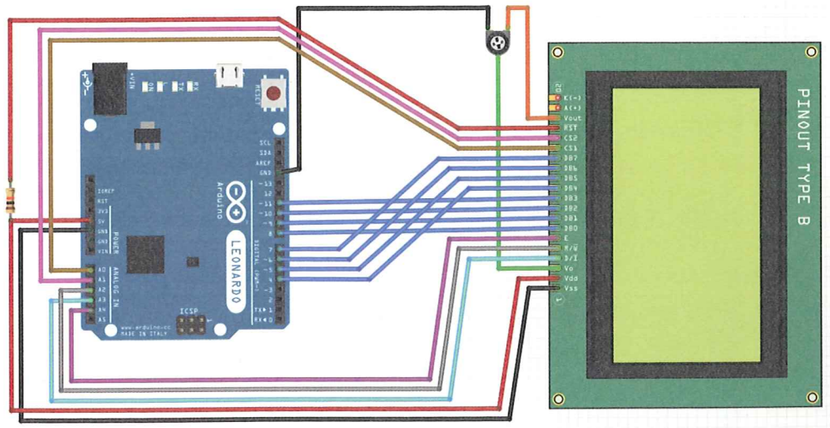
Le tout parfaitement documenté. Et ce n'est pas tout, comme Bill est un développeur vraiment sympa, il fournit également des outils permettant de générer des images utilisables avec sa bibliothèque dans vos croquis. Outils qu'il décline en version Java, Processing et C++ pour une version en ligne de commandes (si vous avez une erreur de compilation concernant `opterr`, `getopt`, etc., suite au `make`, incluez `unistd.h` dans `bmp2glcd.cpp`).

Maturité, documentation, diversité et humilité... Nous avons clairement la bibliothèque de choix pour mettre en œuvre notre écran de récupération.

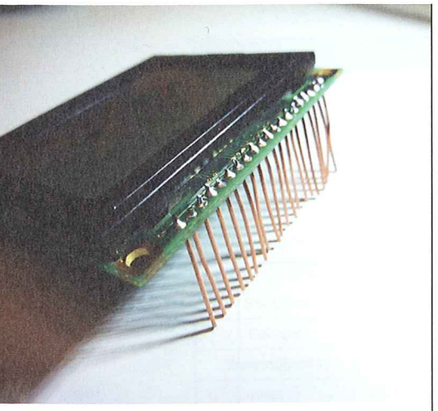
5. PHASE 4 : MISE EN PRATIQUE

Nous avons maintenant tous les éléments qui vont nous permettre de piloter cet écran et nous commençons donc par connecter l'ensemble. L'Arduino choisi, le seul pas déjà en cours d'utilisation pour d'autres projets, est une carte Leonardo. En suivant la nomenclature de la documentation Winstar ainsi que celle d'OpenGLCD, nous branchons :

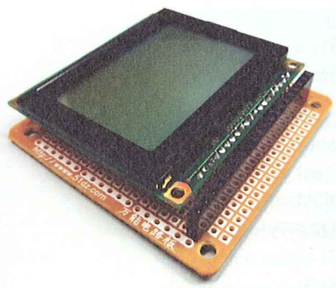
- (1) Vss à la masse,
- (2) Vdd à l'alimentation, c'est-à-dire le +5V fourni par l'Arduino,
- (3) Vo nécessite une liaison particulière et l'usage d'un potentiomètre 10K tout comme avec un afficheur 16*2 par exemple. Mais le potentiomètre sera relié entre la masse et la broche Vee avec la patte du curseur sur Vo (avec un HD44780, Vo est branché de la même manière, mais les autres broches du potentiomètre sont reliées à la masse et à l'alimentation (Vdd)),
- (4) D/I à la broche A3 : c'est un signal de sélection donnée/commande,
- (5) R/W sur A2 : cette ligne permet de basculer entre lecture et écriture,
- (6) E sur A4 : permet d'activer l'afficheur,
- (7) DB0 sur 8,
- (8) DB1 sur 9,
- (9) DB2 sur 10,
- (10) DB3 sur 11,
- (11) DB4 sur 4,
- (12) DB5 sur 5,
- (13) DB6 sur 6,
- (14) DB7 sur 7 : ces 8 dernières lignes permettent de transmettre 8 bits en parallèle à l'afficheur,
- (15) CS1 relié à A0,
- (16) CS2 sur A1 : il y a deux puces pour gérer deux moitiés d'écran (64×64 + 64×64). CS1 et CS2 permettent de choisir à quelle puce on s'adresse,
- (17) RST est relié via une résistance de rappel de 10 Kohms au +5V. C'est la broche de réinitialisation de l'afficheur, active à l'état bas (reset si à la masse),



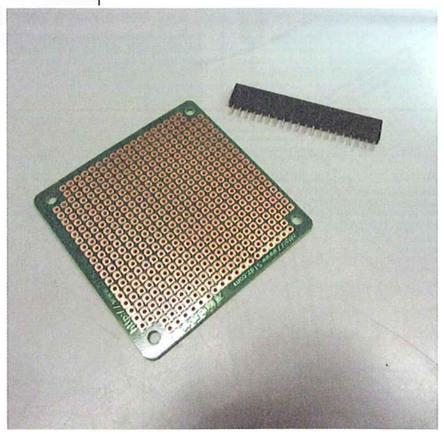
La connexion de l'Arduino Leonardo au module LCD utilise une importante quantité de câbles et il ne nous reste pas beaucoup de broches pour d'autres usages. Dans le cas d'un projet complet bien réel, il faudra alors ruser pour exploiter au mieux les entrées/sorties qui restent.



Première étape : ajouter une série de pattes au module. Si vous voulez savoir comment on soude après minuit alors qu'on était persuadé de vite ajouter un connecteur avant d'aller se coucher... hé bien, ça donne ça.



Le module en place sur la plaque pastillée, nous disposons maintenant d'un connecteur femelle au pas de 2,54 mm pour la connexion à la carte Arduino.



Une bonne amie du bricoleur en électronique, la plaque pastillée. Celle-ci est de triste facture (pas plate, trous non alignés, vernis douteux) mais, si mes souvenirs sont bons, j'en ai eu 30 comme celle-ci pour quelques euros.

- (18) Vee est relié à l'une des broches du potentiomètre qui règle le contraste (c'est la broche d'alimentation de l'écran LCD à proprement parler).

Vous vous en doutez, cela fait un paquet de connexions et littéralement une jolie salade de câbles. Notez que l'utilisation de connecteurs ou d'une platine à essais est largement sous-optimale. Ces connexions pas vraiment franches peuvent être la source de certains problèmes. Nous prenons cependant le risque, mais tâchons de faire cela proprement.

« Proprement » signifie que nous n'allons pas utiliser des fils électriques de récupération (les câbles de téléphones analogiques sont un vrai bonheur pour ça), dénudés et étamés de manière à s'enficher dans une platine à essais. Nous allons, au contraire, souder un connecteur sur le...
Oh ! Voilà une mauvaise surprise, le connecteur sur l'écran n'est pas au pas de 2,54 mm. Sur la largeur totale des 20 broches, il y a bien un bon centimètre d'écart. C'est un pas de 2 mm.

Et c'est reparti pour l'atelier soudure en commençant par la présentation d'une amie fidèle et tantôt très économique : la plaque pastillée. Il s'agit d'un morceau de circuit imprimé criblé d'une matrice de trous. Chacun d'eux est une broche, mais elles ne sont pas reliées entre elles (sauf quelques-unes en bordure). Objectif : relier le module à la plaque et ajouter un connecteur.

Le module sera connecté via des petites pattes (fils de cuivre rigides ou pattes de composants (rien jeter, j'ai dit)) en éventail sur la plaque pastillée.

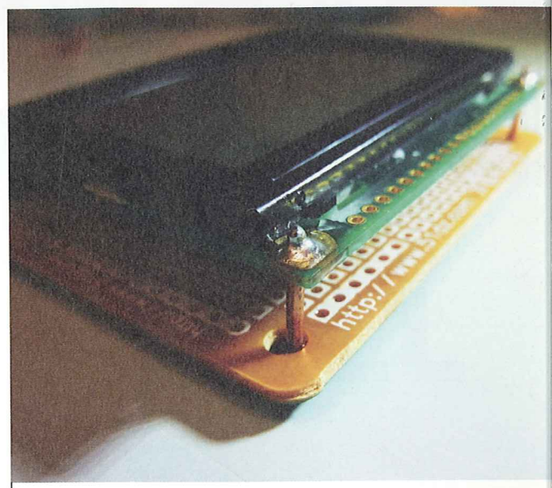
Le module est ensuite placé côté composant de la plaque et les pattes passées au travers d'une ligne de 20 trous. Une chance, c'est exactement la taille de la plaque, les deux trous restants étant connectés avec d'autres (*rails*). On soude, on ajoute le connecteur femelle et on procède aux liaisons côté cuivre.

Petit souci, le module ne tient que par la série de 20 pattes soudées. Un problème courant de ce type de plaques économiques est la qualité plus que douteuse des pastilles de cuivre, du vernis et de la colle utilisés. Laisser le montage dans l'état risque de créer des tensions qui peuvent décoller les pastilles et provoquer une rupture des connexions. Nous allons donc renforcer tant bien que mal la fixation en utilisant les deux supports de l'autre côté de l'écran. Un peu de fil de cuivre un peu plus épais, de l'étain et le tour est joué.

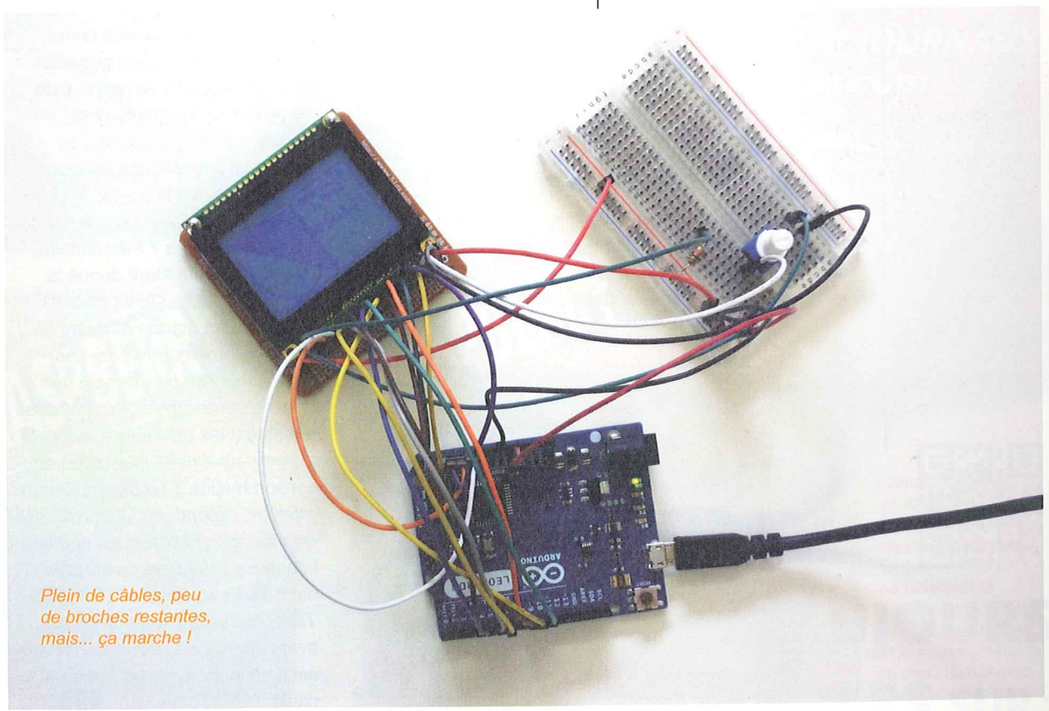
6. PHASE 5 : CROQUIS ET ESSAI

Il ne reste plus qu'à tout relier et pointer son navigateur web sur **https://bitbucket.org/bperrybap/openglcd**. Un lien « Downloads » permet de récupérer la dernière version sous la forme du fichier `openGLCD-v1.0rc1.zip`. On passe ensuite dans l'environnement Arduino (1.0.6) via le menu ***Croquis > Importer bibliothèque*** et ***Add Library*** (??) pour ouvrir une fenêtre de sélection de fichiers. On choisit le Zip fraîchement téléchargé et la bibliothèque est installée (oui, vous pouvez aussi désarchiver le Zip et copier le contenu dans le dossier `libraries` de votre carnet de croquis).

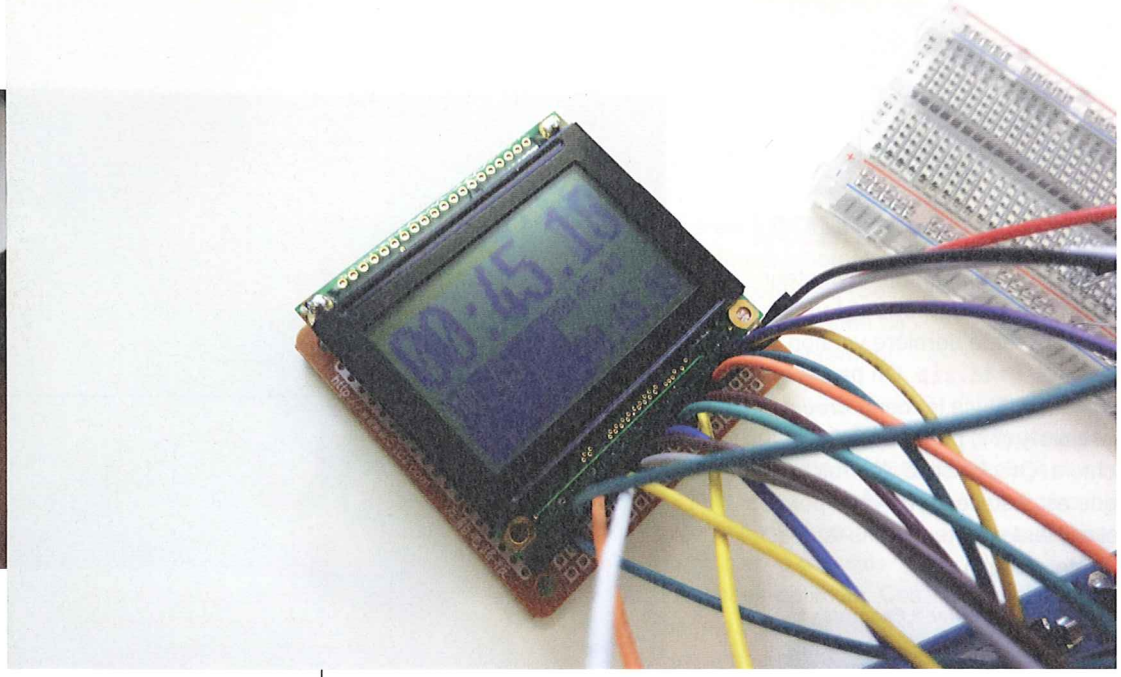
On passe ensuite par les menus ***Fichier > Exemples > OpenGLCD > GLCDv3demo > openGLCD > GLCDdemo*** pour charger un gros croquis, on compile et on charge le tout dans la carte Leonardo en croisant les doigts. Cet exemple occupe une grosse part de la mémoire flash disponible (19374 octets sur 28672), mais la récompense est bel et bien là : on voit apparaître différentes démonstrations sur l'écran LCD (du premier coup en plus) !



On tente de renforcer le montage en bricolant un support à l'arrière de l'écran. C'est assez brouillon, mais cela évite que l'écran ne flotte dans l'air, uniquement tenu par les connexions sur CON1.



Plein de câbles, peu de broches restantes, mais... ça marche !



L'un des écrans de démonstration d'OpenGLCD, bien plus sexy qu'un écran 2×16 caractères. Pas si mal pour quelque chose qui devait partir à la poubelle ou traîner sur l'étal de marchés aux puces encore des années sans mon intervention...

CONCLUSION

Le nombre important d'exemples permet de rapidement arriver à un résultat intéressant, mais le pauvre ATmega328 du Leonardo et ses 32Ko de flash peinent à accepter des programmes très complets. Il faut savoir, en effet, que nous avons là une simple matrice de pixels et que les polices pour le texte ainsi que les images sont stockées dans la flash en compagnie du code. L'une des démonstrations proposées, `GLCD_BigDemo`, ne tient pas dans un Leonardo ou une carte Uno, il faudra basculer sur un Arduino Mega (128Ko de flash) pour en profiter.

Le but ici était davantage de relever le défi en réutilisant un composant et d'en profiter pour apprendre à explorer et à voir les « déchets électroniques » d'une autre manière. Dans le précédent numéro, nous avons entr'aperçu les possibilités offertes par une nouvelle génération d'écrans, couleur, sur bus SPI, avec une intelligence embarquée et disponible pour quelques euros. Cependant, ne perdons pas de vue une chose importante : le module que nous avons utilisé aujourd'hui ne date pas d'hier et a certainement un grand nombre d'heures au compteur. Il fonctionne toujours. Pourra-t-on dire la même chose de ces nouveaux écrans SPI d'ici quelques années ? Personnellement, j'en doute étant donné la qualité générale. On vit maintenant dans un monde différent où les choses ne durent pas. Les fabricants n'ont plus besoin de prévoir, par exemple, des batteries remplaçables par l'utilisateur, car malheureusement, la « durée de consommation » de ces nouveaux produits, répondant à la mode plus qu'à de réels besoins, est souvent inférieure à la durée de vie d'une batterie. Et souvent le MTTF (*Mean Time To Failure*), le temps moyen avant une panne, des composants est calqué sur celui de l'élément le plus faible, la batterie... DB

Si déjà on se donne du mal et on se couche tard, autant en profiter un peu pour faire le tour et l'expérience des fonctionnalités amusantes. Et hop, un logo Hackable converti en code et affiché !