N°27 | NOVEMBRE - DÉCEMBRE 2018

HACKABLE MAGAZINE

DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

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

SOLAIRE / SERVO

Construisez un traqueur solaire pour optimiser le rendement de vos capteurs

p. 80

E-PAPER / COULEURS

Découvrez le papier électronique multicolore pour afficher textes et images

p. 08

MESURE / PI

Mesurez tension, courant et consommation simultanément avec votre Raspberry Pi

p. 72

NEOPIXEL / 3 VOLTS

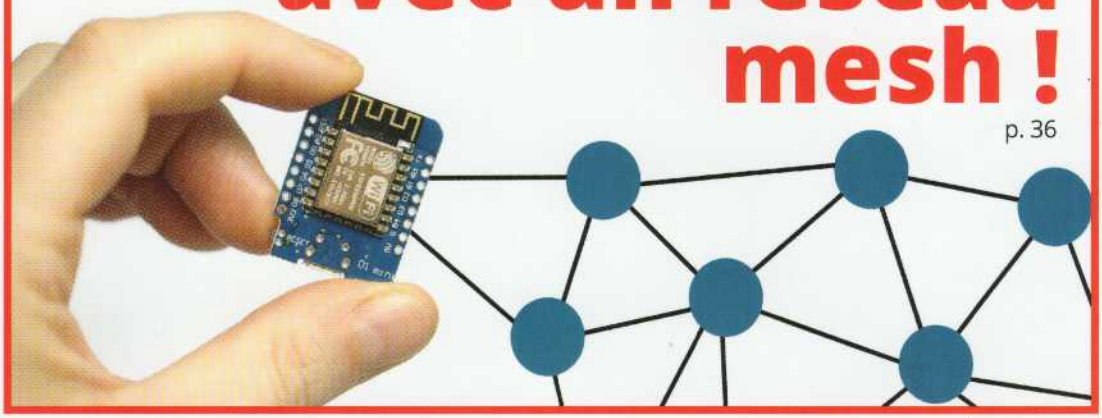
4 solutions pour piloter des leds adressables 5 volts depuis une carte 3,3 volts

p. 56

ESP8266 / Portée du WiFi / Réseau maillé :

Étendez votre WiFi avec un réseau mesh !

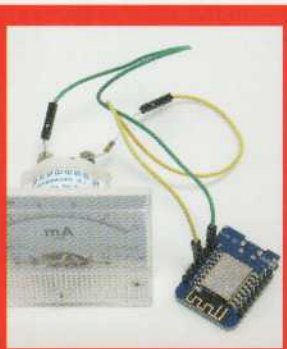
p. 36



MQTT / JAUGE

Utilisez un ESP8266 pour calibrer et piloter des cadrans analogiques via MQTT

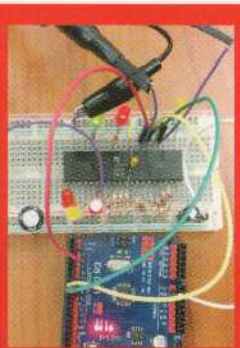
p. 26



Z80 / REBORN

Révision de l'architecture et retour aux sources pour notre ordinateur 8 bits

p. 64



L 19338 - 27 - F: 7,90 € - RD

digital.security

COnnected DEvices EXploitation

Formations de hacking hardware et software des objets connectés

CODEX01

4 JOURS
FORMATION EN ANGLAIS

- S'interfacer et communiquer avec un objet connecté
- Déterminer la surface d'attaque d'un objet connecté
- Extraire les micro-logiciels de différentes façons
- Analyser les micro-logiciels
- Identifier des vulnérabilités dans un micro-logiciel et les exploiter
- Conserver un accès résistant aux mises à jour de l'objet compromis

CODEX02 AVANCÉ

5 JOURS
FORMATION EN ANGLAIS

- Contourner les protections contre l'extraction de micro-logiciel
- Obtenir un accès privilégié au système d'un objet connecté
- Analyser un micro-logiciel d'un système ne reposant pas sur un système d'exploitation
- Identifier et exploiter des vulnérabilités applicatives sur architecture ARM
- Réaliser des attaques par canaux auxiliaires afin de contourner des restrictions
- Identifier, analyser et exploiter de multiples protocoles de communications

Les formations seront dispensées par **Damien Cauquil** (**@virtualabs**), expert sécurité spécialisé dans la sécurité des objets connectés depuis plus de 5 ans. Il est par ailleurs l'auteur de plusieurs outils de référence publiés lors de conférences comme DEF CON ou le Chaos Communicaton Congress et a trouvé et communiqué diverses vulnérabilités dans des objets connectés ainsi que dans plusieurs protocoles de communication.

- 10 % pour les 10 premiers inscrits avec le code 7mq9s3

Inscriptions par mail à formations@digital.security

ÉDITO

Mais que se passe-t-il donc dans le monde de l'open source et du logiciel libre ?

Le 16 septembre dernier, Linus Torvalds annonçait qu'après une phase d'introspection liée au fait qu'il ait raté (involontairement, mais bénéfiquement selon lui) pour la première fois le *Kernel Summit* (réunion annuelle des développeurs Linux), il juge son comportement déplacé et problématique pour les développeurs, et qu'il va prendre du recul tout en ayant une « aide » pour comprendre les émotions des personnes et répondre de façon appropriée **[1]**.

Ceci intervient quelques heures après l'ajout, la veille, d'un « code de conduite » dans la documentation de Linux **[2]**. Code massivement inspiré du texte présent sur **www.contributor-covenant.org**, produit par Coraline Ada Ehmke qui s'est, au passage, gratifié d'un commentaire (ou d'une boutade) sur Twitter se réjouissant de la future exode en masse, à présent que Linux est infiltré par les *social justice warriors* (« *I can't wait for the mass exodus from Linux now that it's been infiltrated by SJWs. Hahahah* ») et dans un autre tweet se féliciter que 40000 projets open source utilisent son code de conduite en précisant « Vous pouvez faire en sorte que je passe une mauvaise journée, mais ça ne change pas le fait que nous avons gagné et que vous avez perdu » (« *40,000 open source projects, including Linux, Rails, Golang, and everything OSS produced by Google, Microsoft, and Apple have adopted my code of conduct. You can make me have a bad day, but it doesn't change the fact that we have won and you have lost.* »). À qui font référence « on » et « vous » est une question à laquelle je n'ai pas trouvé de réelle réponse...

Cette personne est aussi à l'origine du manifeste post-méritocratie, prônant l'abandon du principe méritocratique (du latin *mereo* : être digne, obtenir, et du grec *krátos* : pouvoir, autorité) qui est à la base du développement open source car, je cite, celui-ci « a toujours profité principalement aux privilégiés, à l'exclusion des personnes sous-représentées dans la technologie ». Cette page **[3]**, signée entre autres par Patricia Torvalds **[4]**, la fille de Linus, liste une série de valeurs que d'aucuns trouveront peut-être étonnantes :

- « Nous avons l'obligation d'utiliser nos positions de privilège, même si elles sont précaires, pour améliorer la vie des autres. »

suite page 4...

Hackable Magazine
est édité par Les Éditions Diamond

Les Éditions Diamond
STIMULANT DE MATIÈRE GRISE

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

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

FSC www.fsc.org MIXTE Papier issu de sources responsables FSC® C015136

Suivez-nous sur Twitter
@hackablemag

Alsace

À PROPOS DE HACKABLE...

HACKS, HACKERS & HACKABLE

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

SOMMAIRE

- « Nous devons faire de la place aux personnes qui ne sont pas comme nous pour entrer dans notre domaine et y réussir. »
- « Nous reconnaissons que la valeur des contributeurs non techniques est égale à la valeur des contributeurs techniques. »
- « L'impact négatif des personnes toxiques sur le lieu de travail ou dans la communauté n'est pas compensé par leurs contributions techniques. »
- « Nous refusons de rabaisser les autres en raison de leurs choix d'outils, de techniques ou de langues. »

Je pourrai faire un commentaire sarcastique pour chacune de ces affirmations mais, en toute franchise, à quoi bon...

La page originale anglaise du code de conduite « *Contributor Covenant* » exprime les choses différemment de la traduction française qui explique : « Dans l'intérêt de favoriser un environnement ouvert et accueillant, nous nous engageons, en tant que contributeurs et responsables de ce projet, à faire de la participation une expérience exempte de harcèlement pour tout le monde, quel que soit le niveau d'expérience, le sexe, l'identité ou l'expression de genre, l'orientation sexuelle, le handicap, l'apparence personnelle, la taille physique, l'origine ethnique, l'âge, la religion ou la nationalité. ». Ce qui aurait bien du mal à convenir à qui que ce soit de raisonnable, bien que cela reste assez flou.

Mais ceci est très différent, je trouve, de la version originale : « Mais les projets de logiciels libres et open source souffrent d'un surprenant manque de diversité, avec une représentation dramatiquement faible de femmes, de gens de couleur et d'autres populations marginalisées. » (« *But free, libre, and open source projects suffer from a startling lack of diversity, with dramatically low representation by women, people of color, and other marginalized populations.* »). Le texte d'origine étant en anglais, on est en droit de considérer qu'il exprime plus fidèlement l'objectif visé par ce code.

Le texte poursuit en précisant que les cultures qui valorisent le mérite conduisent à de grandes inégalités, car non seulement les personnes avec du mérite sont souvent excusées pour leur comportement en raison de leur contribution technique, mais l'égalité des chances n'est pas de mise parce que tout le monde n'a pas les mêmes ressources, le même temps libre et la même expérience de la vie. Tout ceci faisant que tous n'osent pas contribuer aux projets open source. Quelque chose ici m'échappe, parce que, si mes capacités m'empêchent de jouer d'un autre instrument que du triangle (et encore), que je n'ai pas de temps à ma disposition pour apprendre ou que je n'ai pas les moyens de m'acheter un instrument, pourquoi donc un groupe devrait-il me laisser jouer ?

Même si ce qui se passe en ce moment parmi les développeurs Linux fait beaucoup de bruit, et certains avancent même l'idée de forker le code pour poursuivre un développement de leur côté, voire envisagent légalement de retirer leur code du projet mettant à mal le développement, le phénomène n'est pas nouveau. Des raisons « communautaires » et/ou sociopolitiques ont par le passé déjà provoqué le départ de certains leaders ou contributeurs importants.

En mai 2018, Rafael Avila de Espindola, 5ème plus gros contributeur de LLVM quitte le projet : « Les raisons pour lesquelles je pars sont les changements dans la communauté. Les discussions sur le changement de licence me rappellent malheureusement la politique de la FSF lorsque je travaillai sur GCC.[...] Le changement dans la communauté que je ne peux pas encaisser concerne la façon dont le mouvement d'injustice sociale l'a imprégné (infiltré ?). Lorsque j'ai rejoins LLVM personne n'a demandé ou n'en avait quelque chose à faire de ma religion, ou de mes opinions politiques. Nous semblions tous juste dévoués à l'écriture d'un bon framework de compilateur » (« *The reason for me leaving are the changes in the community. The current license change discussions unfortunately bring to memory the fsf politics when I was working on gcc.[...] The community change I cannot take is how the social injustice movement has permeated it. When I joined llvm no one asked or cared about my religion or political view. We all seemed committed to just writing a good compiler framework.* ») **[5]**.

Plus récemment, en juillet, Guido van Rossum créateur de Python et « Benevolent Dictator For Life » (BDFL, « dictateur bienveillant à vie ») jette l'éponge : « Mais grossièrement je quitte de façon permanente le poste de BDFL et vous devrez vous débrouiller seuls. Après tout, ceci devait finalement arriver quoi qu'il en soit — il y a toujours eu cette éventualité m'attendant au tournant, et je ne rajeunis pas... [...]. Je suis fatigué et j'ai besoin d'une très grande pause. » (« *But I'm basically giving myself a permanent vacation from being BDFL, and you all will be on your own. After all that's eventually going to happen regardless — there's still that bus lurking around the corner, and I'm not getting younger... [...] I'm tired, and need a very long break.* ») **[6]**. Peut-être est-ce en rapport avec ce mouvement de fond où le politiquement correct tend à prendre une place inquiétante dans les développements open source et où les termes maître/esclave (*master/slave*) sont désormais jugés blessants pour certaines personnes, au point qu'ils doivent être remplacés, dans la documentation Python, en quelque chose de plus anodin comme parent/travailleur (*parent/worker*) **[7]**.

Le phénomène d'interaction entre le développement des projets et des considérations politiques, sociales ou morales, n'a donc clairement rien de nouveau. Mais les choses dernièrement prennent des proportions importantes. En effet, Brendan Eich inventeur du langage JavaScript et architecte de la Mozilla Foundation laisse tomber le projet après avoir été accusé d'homophobie pour avoir fait un don à une association contre le mariage homosexuel, provoquant entre autres une campagne visant à ne plus utiliser le navigateur Firefox et une pétition demandant sa démission **[8]**. Il n'en avait pas fait la promotion, son nom figurait simplement dans la liste des donateurs sur un site. À savoir quel est le rapport entre les opinions politiques d'un responsable d'un projet open source et le travail qu'il accomplit, je ne saurai répondre. Mais l'histoire nous montre qu'il y a des précédents. D'autres exemples, aux effets moins dramatiques, existent dans les communautés de développeurs d'autres projets comme PHP **[9]**, Ruby **[10]**, Go **[11]** ou encore Django **[12]**.

Un certain nombre de personnes semblent voir dans le fonctionnement méritocratique et, oui, dans le comportement excusé de certains contributeurs très talentueux, une forme d'adhésion à des principes qu'ils jugent mauvais, comme « le capitalisme rampant » ou une relation dominant/dominé, voire exploitant/exploité. Ceux-là mêmes qui n'hésitent pas à proclamer que le logiciel libre et l'open source sont intrinsèquement politiquement motivés et que ce mouvement a débuté comme une campagne pour la liberté sociale et digitale, et non comme une simple volonté de rendre le code disponible, utilisable et modifiable par tous, ou plus exactement tous ceux qui en ont l'envie et la capacité.

Il est relativement aisé, en n'ayant pas vécu les premières heures de la croissance en popularité du logiciel libre et de GNU/Linux, de tout simplement s'arrêter à ce qui existe aujourd'hui, à l'acceptation sans retenue des principes du logiciel libre ou de l'open source et des fruits que ces principes apportent. Mais avant qu'il n'existe un tel abondant verger, il n'y avait rien ou, au mieux, des ronces qu'il fallait défricher et une terre brute qu'il fallait transformer. Pour continuer dans l'allégorie, ce sont des jardiniers acharnés, parfois bourrus et « abrasifs », mais toujours talentueux et dévoués à la tâche, qui ont eu la ténacité nécessaire pour faire d'un terrain vague une oasis. Aujourd'hui, alors que le plus gros du travail est fait, il est facile d'arriver avec sa chaise longue et de faire des commentaires sur la façon d'utiliser un râteau et de critiquer le fait de déranger l'herbe lorsqu'on ratisse des feuilles mortes, alors même que ni l'herbe, ni les arbres ne seraient là sans les efforts de ceux-là mêmes dont on critique le ratissage ou l'entretien du verger.

Mais le rejet même du principe méritocratique presque tribal, à la base même du développement qu'il soit open source ou non **[13]**, rend vain tout commentaire du type « Commencez par contribuer au projet et commentez ensuite ». Lorsqu'une personne pense qu'elle mérite une place de par ce qu'elle est et non en raison de ses contributions ou ses actions, ces arguments ne peuvent tout simplement pas être entendus. C'est une vision totalement différente du monde où le « je suis, que le monde s'adapte mes besoins » est de mise, et non le « je suis ce que je fais ».

Pire encore, avec ce genre de démarches, ce n'est jamais l'intérêt du projet qui est prioritaire. En contribuant à un projet, l'objectif premier est de l'améliorer, tout en gagnant en réputation et en expérience par effet de bord. La communauté se forme autour d'un but commun, d'un objectif que tous partagent et auquel tous contribuent. De cet agrégat naissent des relations entre contributeurs, et parfois des conflits, mais le code reste ce liant entre toutes les personnes et est ce qui, dans les situations difficiles, fait office de base d'entente commune. C'est aussi simple que de se poser simplement la question : « pourquoi je veux contribuer et participer à ce projet ? ».

Bien entendu, la notion même « d'aider un projet » peut être interprétée. Certains pensent qu'une « aide » c'est apporter ses propres valeurs morales, qui n'ont rien à voir avec le projet ou son développement, et uniquement cela sans autres contributions. Un peu comme si, dans votre verger vous ayant demandé tant d'efforts pour naître du chaos et rester vivace, quelqu'un venait vous dire que vous devez préférer le vélo à la voiture pour vous y rendre. Non seulement la manière dont vous (et les autres personnes qui participent), vous rendez sur place n'a rien à voir avec le lieu lui-même et son entretien mais, en plus, la personne à l'origine du commentaire le fait lors de sa première visite, sans jamais avoir tenu un râteau et sans jamais avoir créé un verger tel que le vôtre. Et cela, alors même que tous ceux qui participent ont leurs propres préférences quant à leur moyen de déplacement et que ceci n'a jamais eu le moindre impact, positif ou négatif, sur les relations entre les personnes ou le verger lui-même.

En ce qui me concerne, je pense que le mécanisme méritocratique tribal est précisément ce qui efface les différences entre les personnes. Peu importe que vous soyez grand, petit, blanc, noir, vert, bleu, un homme, une femme, de droite, de gauche, religieux, antithéiste... si votre code est bon, votre code vaut quelque chose pour le projet et, de ce fait, vous (ou le personnage que vous projetez) valez quelque chose pour le projet. Votre code ne vaudra pas moins parce que vous êtes ceci ou cela en dehors de votre contribution, mais il ne vaudra pas plus non davantage. Ce code, cette contribution, est votre richesse, votre monnaie, vos points de valeur, dans un cadre borné où le reste est sans importance, car hors sujet. C'est toute la beauté de la chose, cette abstraction est précisément ce qui efface les différences, n'en déplaise à certains. Vous savez, « jugez les gens par rapport à ce qu'ils font, non par rapport à ce qu'ils sont », tout simplement parce que nos actions sont factuelles et que ce que nous sommes est avant tout une affaire de perception. La vôtre bien sûr, mais aussi celle des autres.

« Soyez sympa et contribuez du bon code » est un principe de conduite que tout le monde peut adopter et quelque chose qui a toujours été implicitement présent parmi les développeurs. Oui, c'est vrai, ceci permet d'être très antipathique dès lors qu'on fournit un code de qualité, mais ceci permet aussi de faire de petites contributions modestes, et d'apprendre par la même occasion, en se montrant humble et aimable. Si vous êtes avenant et compétent, vous serez apprécié et respecté, mais si vous êtes hostile et improductif, vous ne pouvez pas raisonnablement espérer une bonne réception. Le reste est un équilibre.

Force est de constater qu'aujourd'hui, maintenant que les vergers sont vivaces et productifs, le politiquement correct prend de plus en plus de place dans les communautés de développement des projets open source et, en parallèle, certains acteurs historiques prennent proportionnellement du recul. Corrélation ou causalité ? Difficile à dire puisqu'il semble y avoir énormément de non-dits, mais le fait est que ces changements ne seront pas sans conséquence, soyez-en certains.

En attendant, pour honteusement paraphraser et détourner les propos du regretté Pierre Desproges, je dirai ceci tout en étant le plus inclusif possible : plus je connais l'espèce humaine, plus j'aime le code et les machines...

Denis Bodor

[1] https://lkml.org/lkml/2018/9/16/167

[2] https://github.com/torvalds/linux/commit/8a104f8b5867c682d994ffa7a74093c54469c11f

[3] https://postmeritocracy.org/

[4] https://opensource.com/life/15/8/patricia-torvalds-interview

[5] https://lists.llvm.org/pipermail/llvm-dev/2018-May/122922.html

[6] https://mail.python.org/pipermail/python-committers/2018-July/005664.html

[7] https://bugs.python.org/issue34605

[8] https://fr.wikipedia.org/wiki/Brendan_Eich#Mozilla

[9] http://news.php.net/php.internals/90728

[10] https://bugs.ruby-lang.org/issues/12004

[11] https://groups.google.com/forum/#!msg/golang-nuts/sy-YcVPADjg/bcO6LAr29EIJ

[12] http://voxday.blogspot.com/2015/10/exposing-true-face-of-sjw.html

[13] Voir « La Tribu informatique » de Philippe Breton chez Métailié 12/1990 ISBN: 2-86424-086-6

FACILITEZ-VOUS LA VEILLE TECHNO

Un accès à + de 7000 articles

Disponibles pour :

MISC · LINUX MAGAZINE / FRANCE · LINUX PRATIQUE · HACKABLE MAGAZINE

Consultez les numéros d'hier et ceux d'aujourd'hui

Un outil de recherche

Un affichage par numéro paru

Les magazines standards et leurs hors-séries

Des articles triés par domaines

connect.ed-diamond.com

ET L'ACCÈS À LA DOCUMENTATION !

Sécurité informatique, Open Source, Linux, Électronique, Embarqué...

c'est la solution pour vous et votre équipe !

À découvrir sur :
connect.ed-diamond.com

Pour vous abonner :
www.ed-diamond.com

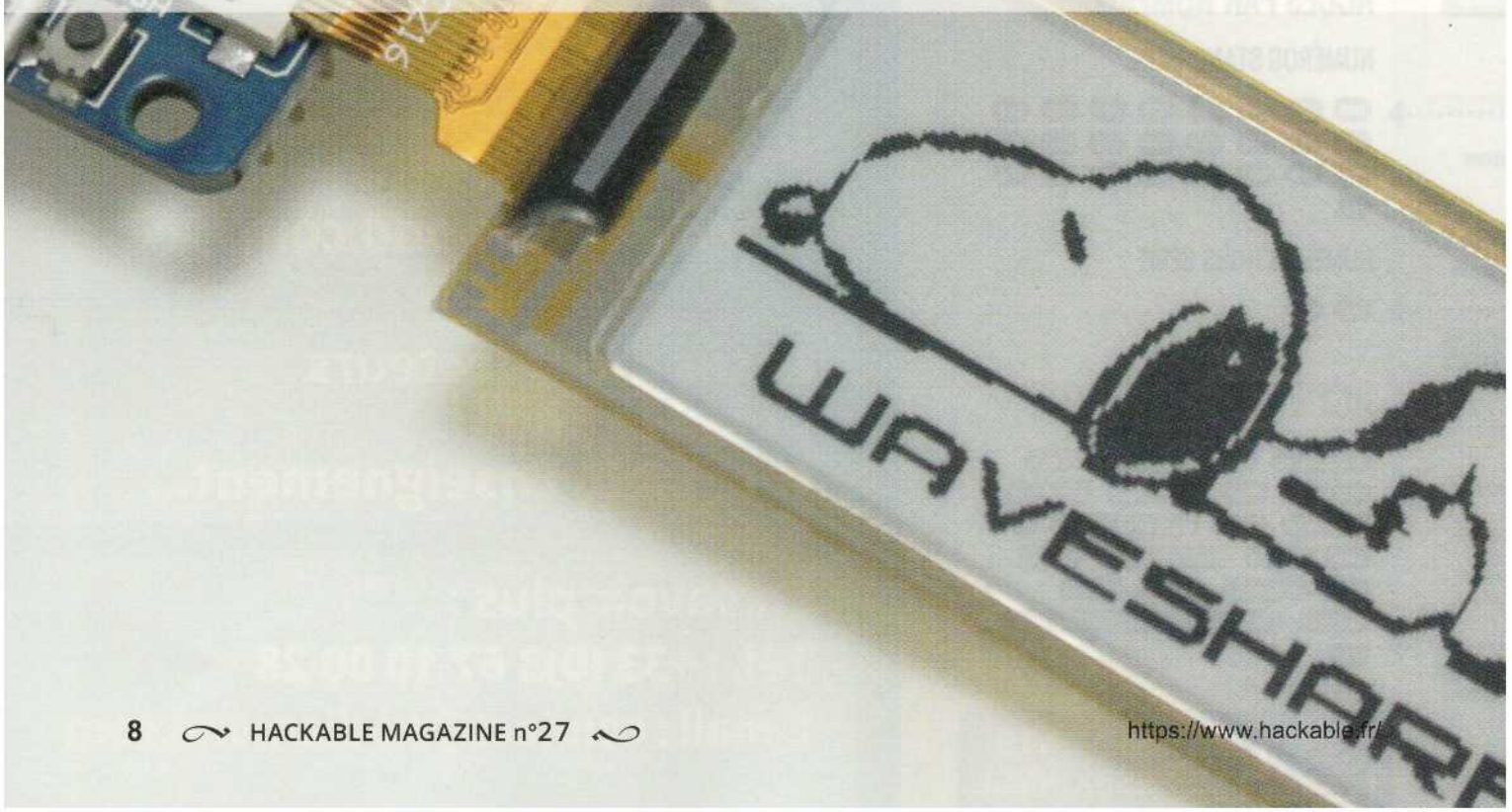
Accès multi-lecteurs possible pour :
Pro, R&D, Enseignement...

En savoir plus :
Tél. : +33 (0)3 67 10 00 28
E-mail : connect@ed-diamond.com

PILOTEZ DES ÉCRANS E-PAPER À TROIS COULEURS !

Denis Bodor

Les écrans e-paper ou e-ink tels ceux utilisés par les liseuses électroniques ou encore les systèmes d'affichage de prix en supermarchés ne datent pas d'hier. Outre le fait de se démocratiser massivement pour certaines applications, ceux-ci évoluent aussi d'un point de vue technologique. Plus fiables, plus rapides, plus économiques... ils n'ont jamais été aussi accessibles et faciles à utiliser qu'actuellement. Mais la grande nouveauté pour le hobbyiste, c'est le fait de pouvoir afficher des images en couleurs !



Attention, lorsque je dis « en couleurs », ceci n'a rien de comparable avec un écran LCD TFT proposant 16, 256, 65K ou 16 millions de couleurs. On parle ici de trois couleurs, dont une est celle du fond (blanc). Historiquement, ces écrans n'étaient capables d'afficher qu'en noir et blanc. Une troisième couleur spécifique au modèle d'afficheur s'ajoute à présent pour les modules pouvant être achetés au détail en ligne : généralement du rouge ou du jaune.

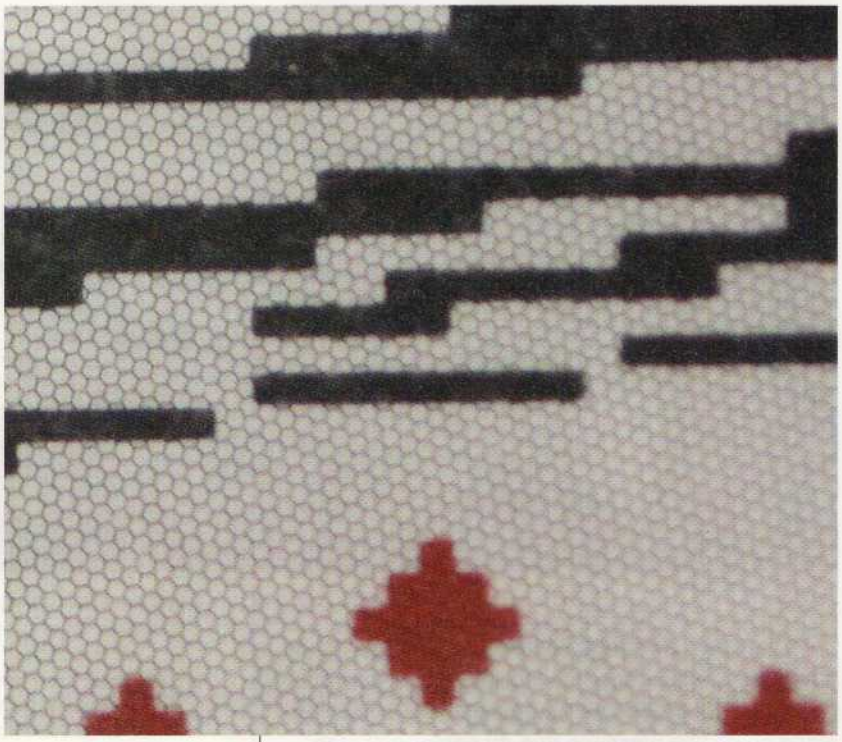
Cette pauvreté en termes de fonctionnalités peut paraître surprenante sachant ce qu'il est possible de faire avec d'autres systèmes d'affichage. Il faut cependant comprendre que la technologie est très différente. Alors qu'un écran LCD/TFT classique, ou même Oled, repose sur l'utilisation de cristaux liquides rétro-éclairés ou de leds formant une matrice, un écran e-paper ou EPD (pour *E-Paper Display*) repose sur un fonctionnement qu'on pourrait qualifier de mécanique, l'électrophorèse. L'écran est constitué d'une myriade de capsules contenant de fines particules d'un diamètre de quelques microns qui sont en suspension dans un liquide. Ces particules sont colorées et chargées électriquement puis contenues dans ces réceptacles étanches pris en sandwich dans un substrat. En appliquant une charge électrostatique de façon précise, on déplace les particules de manière à les rendre visibles à l'utilisateur.

Le papier électronique ou encore EPD ou e-paper est aujourd'hui une technologie relativement courante, en particulier depuis sa généralisation en supermarché, pour l'affichage des prix. Ce qui est plus rare en revanche c'est d'y voir de la couleur. C'est pourtant ce que propose cet écran à environ 10€.

Le principal avantage de cette technologie est de présenter un affichage très similaire à un support imprimé, plus proche de l'impression sur un papier glacé ou un film plastique que du papier mat standard. Il en résulte différentes caractéristiques intéressantes dont la lisibilité sous différents angles, un aspect très « naturel » ou encore le fait qu'une fois l'image affichée, il n'est plus nécessaire d'utiliser du courant pour la maintenir. Un écran hors tension, ou même totalement déconnecté de son contrôleur conservera l'affichage dans son état pendant une période très importante (mais pas infinie).

Parmi les points négatifs, on trouve la nécessité d'une source de lumière (comme pour le papier) et un délai relativement important pour mettre à jour l'image affichée. En effet, le mouvement des particules n'est pas parfait et il est nécessaire de réitérer plusieurs fois l'opération pour obtenir un affichage net et contrasté. La plupart du temps, même pour mettre à jour une petite partie de l'image, il est nécessaire de basculer plusieurs fois tout l'écran pour « secouer » les particules avant l'affichage définitif. Cette procédure pouvant prendre jusqu'à plusieurs secondes, il est très difficile d'afficher des animations, du moins avec les écrans les moins onéreux.

Il faut savoir cependant que depuis les premières commercialisations de liseuses, les procédés de fabrication ont grandement impacté les performances de ces écrans. On trouve désormais des liseuses capables de faire défiler du texte relativement rapidement ou de permettre l'utilisation d'interfaces, certes



La technologie utilisée par l'e-paper repose sur l'utilisation de pigment magnétisable, en suspension dans des capsules. Plus les capsules sont petites, plus la densité est grande et plus la résolution est importante... et plus l'écran est cher.

simples, mais assez proches de ce qu'on peut trouver sur une tablette ou un écran de PC. Le prix est toutefois directement proportionnel aux performances pour ce type de matériel, ainsi qu'à la taille de l'écran. Les quelques liseuses A4 disponibles actuellement, par exemple, avoisinent les 800€.

Ce qui nous intéresse ici, cependant n'est pas la performance ou la taille de l'écran, mais ces caractéristiques et fonctionnalités proposées. En particulier, le fait de permettre un affichage en noir et avec une troisième couleur (jaune ou rouge), et de rendre la gestion de cet affichage très facile (connexion SPI et électronique de puissance embarquée). Voyons donc ici qui fabrique ces petites merveilles et comment les utiliser au mieux.

1. LES ÉCRANS E-PAPER WAVESHARE

Waveshare est un fabricant proposant une très vaste gamme de matériels, d'écrans, de modules d'affichage et, bien entendu, d'écrans e-paper. Le matériel dont nous allons parler fait partie d'une gamme spécifique composée de trois types d'éléments :

- Les écrans eux-mêmes se déclinant en plusieurs tailles de 4,3 pouces à 1,54 pouce de diagonale et en version blanc/noir, blanc/noir/jaune et blanc/noir/rouge. Il existe également, pour certaines tailles, des versions souples. Ces écrans sont décrits dans la documentation (wiki Waveshare) comme des panneaux d'affichage bruts (*E-Ink display raw panel*).
- Les interfaces de gestion prenant en charge la génération des tensions nécessaires au pilotage des écrans. Ceux-ci peuvent être proposés sous la forme d'un circuit électronique que la documentation désigne sous le terme HAT (*E-Paper Driver HAT*) bien qu'il ne s'agisse pas de quelque chose de spécifique aux Raspberry Pi. Cette électronique d'interfaçage peut également être intégrée avec un écran, formant ainsi un module pilotable par une carte Arduino ou ESP8266. Enfin, Waveshare propose également un shield Arduino normalement réservé aux cartes Arduino en 3,3V ou aux cartes Nucleo de STMicroelectronics. Notez que ces produits sont prévus pour une utilisation en 3,3V même si le wiki du constructeur précise, tout en le décourageant fortement, qu'une utilisation en 5V peut être « tentée » (littéralement dans la documentation).
- Une carte intégrant un ESP8266 et l'électronique de contrôle. C'est le seul produit de la gamme à fournir un microcontrôleur et donc un élément

programmable. Celui-ci se présente sous la forme d'une carte d'environ 5 cm sur 3 cm assez similaire à un module Wemos D1 mini ou tout autre petite carte/module ESP8266. Cette carte dispose d'un connecteur *flatflex* permettant la connexion directe d'un panneau d'affichage brut.

C'est sur cette dernière solution que j'ai concentré mes efforts puisque cela me paraissait être celle la plus rentable et la plus simple à mettre en œuvre, tout en conservant une certaine souplesse d'utilisation. Cet achat (14,87€) s'est complété de trois écrans bruts de 2,13 pouces de diagonale : un blanc/noir souple, un blanc/noir/jaune et un blanc/noir/rouge. Le tout acheté sur eBay auprès de Wavesharé, avec le port offert, respectivement aux prix de 11,06€, 10,17€ et 10,96€.

Voici une liste non exhaustive des modèles en vente sur eBay, accompagnée des déclinaisons (rien=blanc/noir, B=blanc/noir/rouge, C=blanc/noir/jaune, D=blanc/noir souple), des tailles de la surface d'affichage, de la résolution et des prix actuels :

- 7,5 pouces e-Paper HAT : 163,2 mm (H) par 97,92 mm (V), 640 x 384 pixels, ~38€ à 50€ pour le panneau seul, 47€ pour le HAT et 58€ pour les versions B et C.
- 5,83 pouces e-Paper HAT : 118,80 mm (H) par 88.26 mm (V), 600 x 448 pixels, 40€ à 43€ pour le panneau seul, 47€ pour le module et 52€ pour les versions B et C.
- 4,2 pouces e-Paper Module : 84,8 mm (H) par 63,6 mm (V), 400 x 300 pixels, 20€ à 24€ pour le panneau seul, 28€ pour le module et 21€ pour les versions B et C.
- 2,9 pouces e-Paper Module : 29,05 mm (H) par 66,89 mm (V), 296 x 128 pixels, 12€ à 13€ pour le panneau seul, 18€ pour le module et 20€ pour les versions B et C.
- 2,7 pouces e-Paper HAT : 57,288 mm (H) par 38,192 mm (V), 264 x 176 pixels, 11€ à 12€ pour le panneau seul, 20€ pour le HAT ou sa version B (pas de version blanc/noir/jaune).
- 2,13 pouces e-Paper HAT : 23,71 mm (H) par 48,55 mm (V), 250 x 122 pixels (ou 212 x 104 modèle B/C/D), 8€ à 11€ pour le panneau seul, 16€ pour le HAT et 18€ pour les versions B et C. La version D, souple, existe en brute (11€) et en module (20€).
- 1,54 pouces e-Paper Module : 27,6 mm (H) par 27,6 mm (V), 200 x 200 pixels, 9€ à 10€ pour le panneau seul, 14€ pour le module et 17€ pour les versions B et C.
- *e-Paper Driver HAT* seul (circuit de contrôle et d'alimentation) : 12€ avec rallonge flatlex et circuit de connexion/raccord.
- *e-Paper Shield* pour Arduino (3,3V) ou STM32 Nucleo, 11€.
- *e-Paper ESP8266 Driver Board* dont il est question ici, 15€.

Vous remarquerez que la surface de l'écran n'est pas le seul facteur impactant le prix. La version 7,5 pouces est moins chère que la 5,83 pouces alors que la surface est moins importante, mais la résolution est toute autre. Et qui dit « résolution », dit finesse du tracé des images ou du texte, ce qui peut être assez important pour du e-paper en raison de la « sensation » de lecture qui est proche de celle du papier classique.

Le choix d'une taille ou d'une autre est fonction de vos préférences, de votre budget, mais également de la mise en pratique de ce type de matériel. Je dois dire que j'ai longuement réfléchi à la question avant d'opter pour du 2,13 pouces, car j'imagine parfaitement toutes sortes d'applications pour ces dimensions (sonnette, étiquette, personnalisation d'un clavier,

fond d'ampèremètre analogique, disque de stationnement intelligent (?!)), mais bien plus difficilement pour quelque chose de proche d'un format A6 (feuille de papier pliée en 4). Une liste de courses dynamique peut-être, mais à ce prix-là autant hacker une liseuse d'occasion (Kobo touch) avec, en prime, un système GNU/Linux complet sur microSD (voir **https://github.com/kobolabs/Kobo-Reader**).

Concernant la carte ESP8266 de Waveshare, celle-ci n'est pas très différente d'autres modèles sans rapport avec les écrans e-paper, si ce n'est par le fait qu'elle intègre un circuit d'alimentation spécifique pour l'écran et un connecteur adapté. Ce dernier permet la connexion directe d'un écran brut tout aussi bien que via le câble flatflex de quelques 20 cm livré avec l'écran. Ceci permet soit d'attacher l'écran à proximité de la carte ou de façon déportée pour des utilisations spécifiques.

En plus de ce connecteur et du classique micro-USB, d'une led, et de deux boutons poussoir (reset et flash), un interrupteur à deux positions est présent. Celui-ci permet de sélectionner le type d'écran connecté :

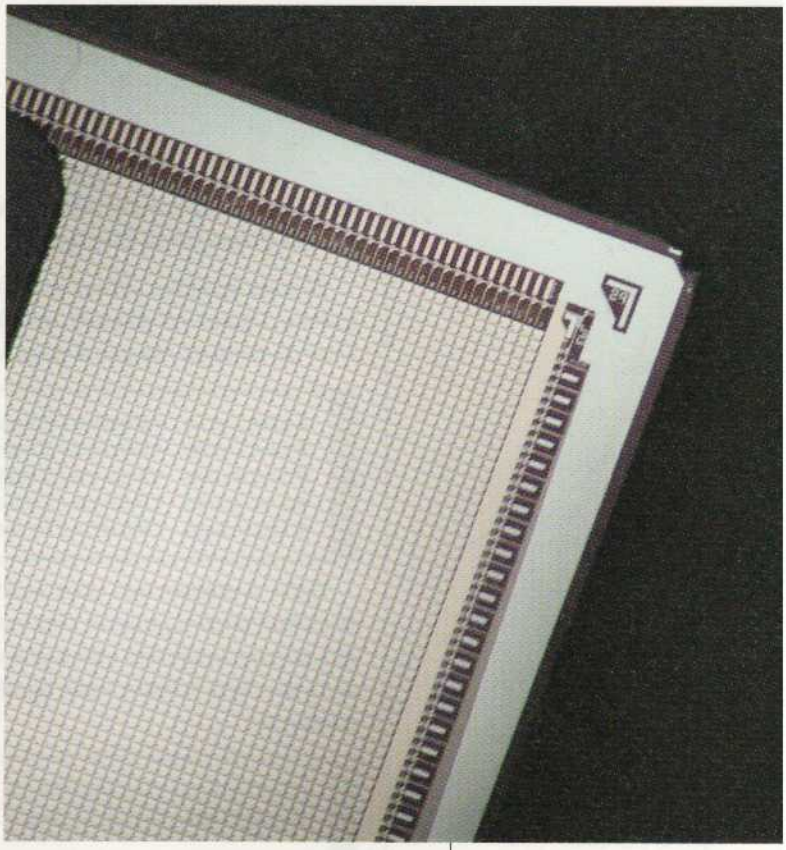
- position A : écrans 1,54 pouces, 2,13 pouces et 2,9 pouces en version blanc/noir.
- position B : écrans 1,54 pouces (B), 2,13 pouces (B), 2,7 pouces, 2,7 pouces (B), 4,2 pouces, 4,2 pouces (B), 7,5 pouces, 7,5 pouces (B). La documentation en ligne ne parle pas des versions C, mais les essais ont montré que B et C sont strictement équivalents, du moins pour la version 2,13 pouces.

En plus des schémas, documentations techniques et informations sur le wiki, Waveshare met à disposition un croquis Arduino de démonstration, accompagné d'un utilitaire Windows permettant de convertir des images à destination de l'écran. Comme souvent pour ce type de produits ou de composants, le code du constructeur n'est pas très évolué et assez ancien. De plus, la simple idée d'utiliser un logiciel dont l'origine n'est pas claire sur une machine Windows me dispense personnellement de tout essai. Je préfère ne pas jouer avec le feu et immédiatement chercher si un développeur n'aurait pas travaillé sur sa propre version du support, sous la forme d'une bibliothèque Arduino régulièrement mise à jour. Et c'est précisément le cas...

2. PILOTER SON ÉCRAN WAVESHARE

Pour mettre en œuvre notre ESP8266 équipé de son écran 2,13 pouces, nous n'allons pas utiliser le code, assez pauvre, fourni par le constructeur, mais préférer la bibliothèque *GxEPD* de J-M Zingg, disponible sur **https://github.com/ZinggJM/GxEPD**. Cette bibliothèque supporte non seulement les écrans e-paper que nous avons listés précédemment, mais également un certain nombre d'autres modèles de Dalian Good Display, qui semblent identiques.

Je ne saurai donner mon avis sur ce support n'ayant pas le matériel à ma disposition, mais on peut supposer qu'il est d'aussi bonne qualité que celui des périphériques Waveshare. Je remarquerai cependant au passage que le développeur a choisi de ne pas utiliser le mécanisme de contribution proposé par GitHub sous la forme de *Pull Requests* et de *Merge*, préférant des interactions, des questions, des suggestions et des contributions via le forum Arduino. Il précise d'ailleurs qu'en général il ne fusionne pas de suggestions/requêtes (*Pull Requests*) et qu'il les supprime à volonté, limitant délibérément leur intérêt à la simple information concernant d'éventuelles évolutions de son code. Bien que je comprenne ses choix, c'est un point de vue que je ne partage pas forcément et, comme nous le verrons dans la suite, mes modifications sur son code ne feront donc pas l'objet d'un *Pull Requests* inutile, pas plus que je vais passer par le forum Arduino pour proposer ma contribution. Et ça, c'est mon choix à moi.



À l'arrière des écrans testés pour cet article, on distingue clairement la matrice utilisée pour contrôler chaque pixel.

La bibliothèque *GxEPD* s'installera manuellement dans le sous-répertoire **libraries/** du répertoire de vos croquis, soit par téléchargement sous forme de fichier ZIP, soit par clonage du dépôt Git. Il vous faudra également installer la bibliothèque *Adafruit_GFX* fournissant les primitives graphiques et la gestion de polices de caractères. Celle-ci est disponible directement dans le gestionnaire de téléchargement. À la date où est écrit cet article, la dernière version de la bibliothèque *GxEPD* est la 3.0.0. Celle-ci marque un changement important dans la structure du code et en particulier dans la manière d'inclure la bibliothèque dans vos croquis. Le développeur a, en effet, souhaité se rapprocher au mieux de la structure standard d'une bibliothèque Arduino par rapport aux versions précédentes incluant directement des fichiers **.cpp**.

L'inclusion des fichiers entête est un point important pour le support du matériel, car tous les modèles d'écran ne s'initialisent et ne se gèrent pas de la même façon. L'inclusion se fait donc en plusieurs temps. Nous avons d'une part un fichier générique qui accompagne les inclusions classiques (du moins sur ESP8266) :

```
#include <ESP8266WiFi.h>
#include <GxEPD.h>
```

Puis nous avons le fichier d'entête propre au modèle d'écran :

```
//#include <GxGDEP015OC1/GxGDEP015OC1.h>    // 1.54" b/w
//#include <GxGDEW0154Z04/GxGDEW0154Z04.h>  // 1.54" b/w/r 200x200
//#include <GxGDEW0154Z17/GxGDEW0154Z17.h>  // 1.54" b/w/r 152x152
//#include <GxGDE0213B1/GxGDE0213B1.h>      // 2.13" b/w
#include <GxGDEW0213Z16/GxGDEW0213Z16.h>    // 2.13" b/w/r
//#include <GxGDEH029A1/GxGDEH029A1.h>      // 2.9" b/w
//#include <GxGDEW029Z10/GxGDEW029Z10.h>    // 2.9" b/w/r
//#include <GxGDEW027C44/GxGDEW027C44.h>    // 2.7" b/w/r
//#include <GxGDEW027W3/GxGDEW027W3.h>      // 2.7" b/w
//#include <GxGDEW042T2/GxGDEW042T2.h>      // 4.2" b/w
//#include <GxGDEW042Z15/GxGDEW042Z15.h>    // 4.2" b/w/r
//#include <GxGDEW0583T7/GxGDEW0583T7.h>    // 5.83" b/w
//#include <GxGDEW075T8/GxGDEW075T8.h>      // 7.5" b/w
//#include <GxGDEW075Z09/GxGDEW075Z09.h>    // 7.5" b/w/r
```

Les « b/w » et « b/w/r » accompagnant les tailles en pouces font référence aux couleurs utilisées, avec « b », « w » et « r » signifiant respectivement « noir », « blanc » et « rouge » (le jaune n'étant qu'une déclinaison de la version rouge). Le modèle utilisé pour les tests de cet article correspond au fichier **GxGDEW0213Z16.h** et pourra être utilisé avec l'une ou

l'autre version tricolore. Notez que ces fichiers sont communs aux modèles Waveshare et Dalian Good Display.

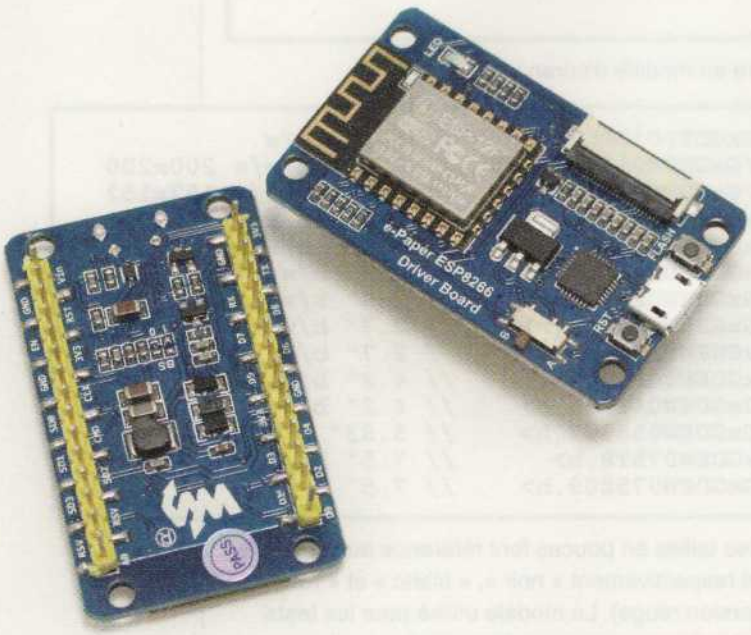
Si vous comptez utiliser du texte, vous devrez également inclure les fichiers de polices. L'écran ne dispose, en effet, pas de générateur de caractères intégré et dessiner du texte se résume à générer des groupes de pixels de façon logicielle :

```
#include <Fonts/FreeMono9pt7b.h>
#include <Fonts/FreeMonoBold9pt7b.h>
#include <Fonts/FreeMonoBold12pt7b.h>
#include <Fonts/FreeMonoBold18pt7b.h>
#include <Fonts/FreeMonoBold24pt7b.h>
```

Enfin, à présent que nous avons la base et le code spécifique à l'écran utilisé, il nous faut encore préciser comment est réalisée l'interface. La très grande majorité des écrans supportés utilise une connexion SPI et nous devons non seulement inclure les bons fichiers, mais également déclarer les objets correspondants de façon adéquate :

```
#include <GxIO/GxIO_SPI/GxIO_SPI.h>
#include <GxIO/GxIO.h>

GxIO_Class io(SPI, SS, 4, 5);
GxEPD_Class display(io, 5, 16);
```

Nous déclarons un objet `io` représentant la connexion avec comme argument du constructeur le type de liaison, la ligne utilisée pour le signal /CS (*chip Select*), le signal data/commande et le reset de l'écran. Les valeurs utilisées ici correspondent à la connexion physique, invariable, entre la carte ESP8266 de Waveshare et l'écran brut. Si vous décidez d'utiliser une autre carte, comme un ESP8266 générique type Wemos D1 mini, et des câbles pour connecter un module écran, vous devrez adapter cela en fonction de vos liaisons. Notez qu'ici les numéros de GPIO de l'ESP8266 sont utilisés et non les valeurs sérigraphiées sur un module ESP8266 comme une carte Wemos (SS=15=D8, 4=D2, 5=D1, 16=D0).



Waveshare, en plus des écrans déclinés en de nombreuses tailles, propose également une carte à base du très connu ESP8266. Celle-ci intègre non seulement le microcontrôleur, mais également l'électronique indispensable pour contrôler un écran e-paper.

Le second objet déclaré réutilise l'objet `io` et précise, en argument, les signaux reset (5) et *busy* (16) permettant au microcontrôleur, et donc au croquis, de savoir s'il peut s'adresser à l'écran. Rappelez-vous, l'affichage sur un support e-paper est bien plus lent que sur un écran TFT ou même un module LCD alphanumérique HD44780. Lorsque l'écran travaille et rafraîchit l'affichage, il n'est pas possible de lui intimer des ordres. Dans le reste du croquis, `display` est l'objet utilisé pour piloter l'affichage.

Nous pouvons directement enchaîner sur un premier essai en nous attaquant à la fonction `setup()`, qui commence très classiquement :

```
void setup()
{
  // Pas de Wifi ici
  WiFi.disconnect();
  WiFi.mode(WIFI_OFF);

  display.init(115200);

  Serial.println("setup fait");
```

Une première chose doit ici titiller votre curiosité : nous utilisons la méthode `println` de `Serial` alors que nous n'avons pas initialisé avec `begin()` en spécifiant une vitesse. Nous n'en avons pas besoin du fait de l'utilisation de `display.init(115200)` qui, lorsqu'un argument lui est passé, le fait à notre place à la vitesse indiquée (ici 115200 bps). Le fait de spécifier une vitesse nous permet de provoquer l'affichage de messages de mise au point sur le moniteur série, directement à partir de la bibliothèque.

Ceci n'est pas une approche technique qui m'enchante personnellement et j'aurais préféré voir `init()` prendre en argument optionnel directement l'objet représentant la connexion série, préalablement initialisée par mes soins. Cela me paraît plus « propre » tout en me laissant la liberté d'utiliser les liaisons comme je l'entends. Ce n'est pas le seul point qui me dérange dans le code de J-M Zingg, mais j'y reviendrais par la suite. Nous pouvons continuer en réglant quelques paramètres dans le but d'afficher du texte :

```
  // orientation de l'écran
  display.setRotation(3);
  // choix de police
  display.setFont(&FreeMonoBold9pt7b);
  // remplissage de l'écran
  display.fillScreen(GxEPD_WHITE);
  // placement du curseur
  display.setCursor(0, 12);
```

L'orientation de l'écran, définie avec `setRotation()`, prend en argument une valeur entre 0 et 3 correspondant à des rotations de 0°, 90°, 180°, 270°. Cet ajustement est purement esthétique et opéré dans le but d'aligner correctement l'affichage avec la position physique de l'écran. Avec l'afficheur en 2,13 pouces, cette rotation de trois quarts de tour nous permet d'afficher le futur texte avec un point de départ situé en haut à droite alors que l'écran est placé horizontalement avec son connecteur sur la gauche.

Le choix de la police d'affichage est dépendant des fichiers que vous aurez inclus en début de croquis et appartenant à la bibliothèque *Adafruit_GFX*. Une liste complète est présente dans le sous-répertoire `Fonts/` de la bibliothèque, mais ne vous réjouissez pas trop. En effet, les symboles qui s'y trouvent, sous la forme de tableaux de glyphes, ne couvrent que le jeu de caractères ASCII de base et les caractères accentués, ainsi que certains symboles, sont absents. Il vous est possible de générer de nouveaux fichiers d'entête à partir de fichiers TTF grâce à l'outil `fontconvert` livré avec les sources de *Adafruit_GFX*, mais ceci ne règle pas le problème de jeu de caractères UTF8 et il vous faudra bricoler, ou plonger votre nez dans les fonctions avancées de la bibliothèque.

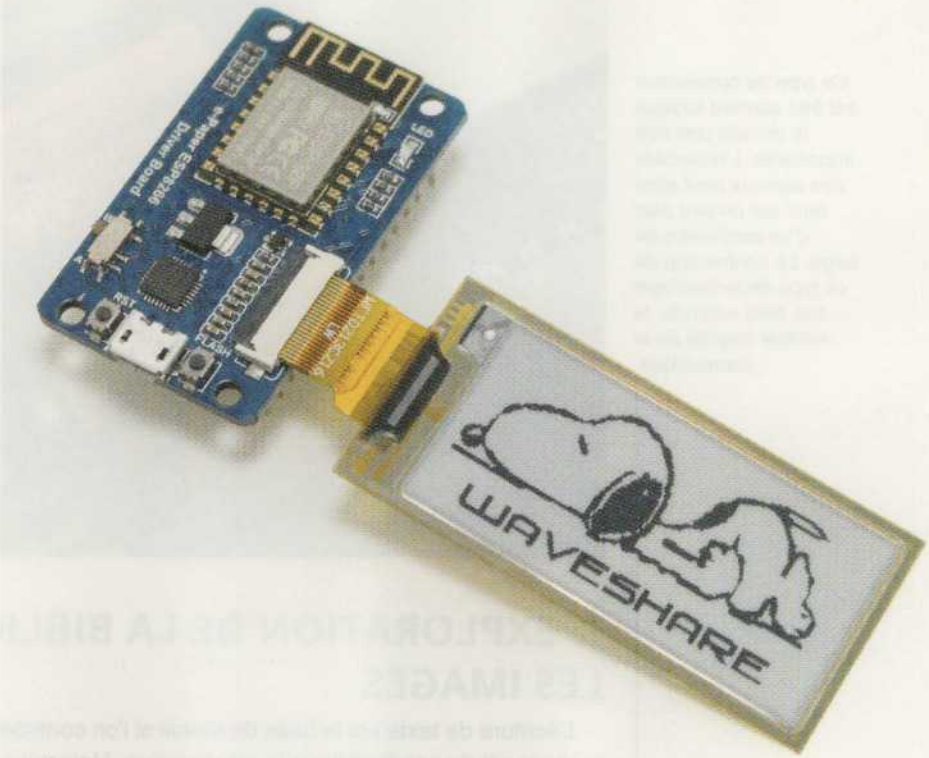
`fillScreen()` nous permet de remplir l'écran d'une couleur de notre choix entre `GxEPD_WHITE`, `GxEPD_BLACK` et `GxEPD_RED` (qui est jaune avec l'écran correspondant). Il est ici utile de préciser que ce type d'action n'est pas immédiat. Nous ne travaillons pas directement sur l'écran, mais dans une mémoire représentant son contenu (un *framebuffer*). Ce n'est que lorsque nous déciderons de « pousser » ces changements avec `update()` qu'ils seront répercutés sur l'écran.

Enfin, `setCursor()` prend en argument une position X et Y où la prochaine écriture de caractères aura lieu. Cette position est relative à l'orientation de l'écran et le système de coordonnées débutera toujours en haut à gauche. Il est important de remarquer ici que le placement de ce curseur virtuel fait correspondre le **bas** de celui-ci avec une position à l'écran. En d'autres termes, si vous utilisez `display.setCursor(0, 0)` et écrivez des caractères, vous ne verrez apparaître du texte, que la dernière ligne d'un pixel de haut et non le texte dans toute sa hauteur. C'est pourquoi ici, avec une police de 9 points, le curseur est placé à la position verticale 12 et non 0. Ce genre de choses est également à prendre en considération si vous faites un saut de ligne dans une police puis en changez pour écrire une nouvelle ligne. Si la nouvelle police est plus grande, le texte peut déborder sur la ligne supérieure et si la police est plus petite, le texte se trouvera à une distance trop importante.

À propos d'écrire du texte, justement, nous pouvons y passer :

```
  // Choix couleur
  display.setTextColor(GxEPD_BLACK);
  // écriture
  display.println("COUCOU Tralala");
  display.println();

  // Choix police et couleur
  display.setFont(&FreeMonoBold18pt7b);
  display.setTextColor(GxEPD_RED);
  // écriture
  display.println("HACKABLE");
  display.println("Num. 27");

  // Affichage effectif
  display.update();

  Serial.println("Affichage fait.");
]
```

Comme nous le ferions pour un tableau blanc, nous choisissons une couleur avec `setTextColor()` et nous nous plions d'un simple `println()` pour écrire. Ces méthodes reposant sur la bibliothèque *Adafruit_GFX*, elles en héritent le comportement. Nous avons ainsi la possibilité d'utiliser aussi bien `println()` et `print()` mais, de plus, le texte généré « chasse » à la ligne automatiquement s'il ne rentre pas sur la largeur de l'écran. Encore une fois ceci ne se fait pas réellement sur l'écran, mais sur un morceau de mémoire représentant son contenu. C'est pourquoi nous concluons notre fonction `setup()` avec l'appel à la méthode `update()` pour appliquer ces changements. À noter au passage que ce mode de fonctionnement nécessite entre 30 Ko et 60 Ko (trois couleurs) de mémoire (SRAM), chose aisée sur ESP8266, ESP32, STM32 et Arduino Due, mais plus problématique avec les cartes à base de microcontrôleur AVR, qui doivent alors fonctionner avec des pages de mémoire (gérées par la bibliothèque, mais non testées ici).

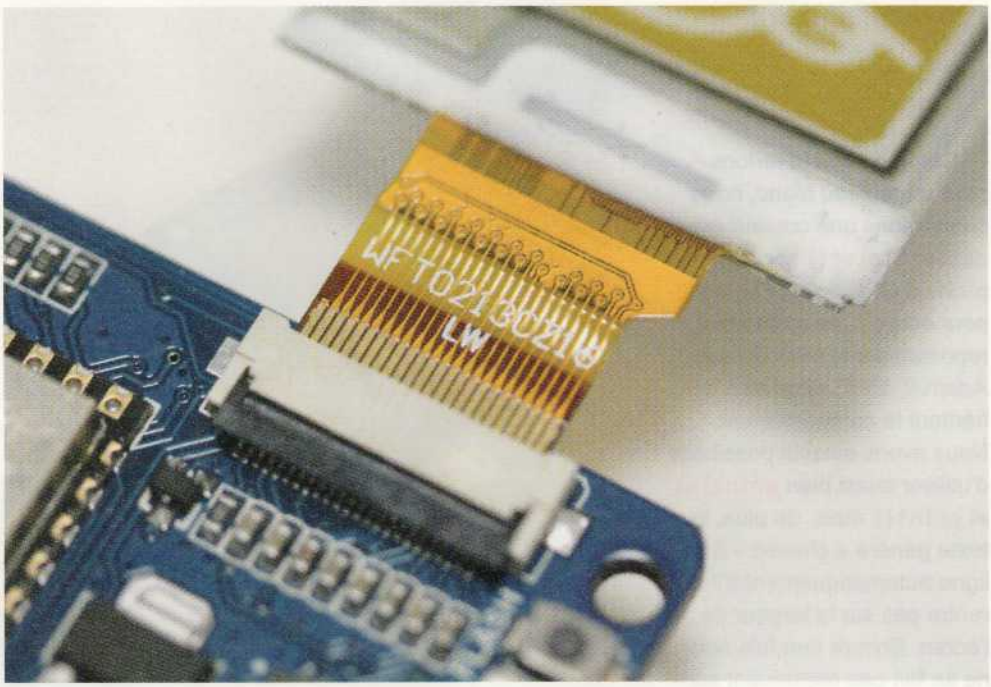


Le connecteur présent sur la carte ESP8266 permet de brancher directement un écran « brut ». Aucune interface supplémentaire n'est nécessaire comme ce serait le cas avec un module ESP8266 standard ou une carte Arduino (l'image affichée ici est celle présente sur l'écran lors de sa réception, la notion de droits d'auteur semble totalement échapper à Waveshare).

Ce premier croquis se conclut avec une fonction `loop()` toute aussi indispensable qu'inutile :

```
void loop()
{
  delay(1000);
}
```

Une fois le croquis chargé dans la mémoire de l'ESP8266 et celui-ci redémarré automatiquement, vous devriez voir l'écran s'agiter durant une bonne quinzaine de secondes. L'écran « flashe » tout d'abord pour s'effacer et revenir à un canevas vierge. Le texte en noir pulse un temps avant que l'écran ne s'occupe de la partie colorée (rouge ou jaune) en faisant apparaître doucement la couleur. Ce processus et sa durée, bien que conduisant à un résultat que je trouve extraordinaire et captivant, vous montrent clairement les limitations de ce type d'afficheur. Si l'information doit être présentée rapidement ou à intervalles courts, l'e-paper tricolore n'est sans doute pas la technologie à utiliser.



Ce type de connecteur est très courant lorsque la densité doit être importante. L'ensemble des signaux peut ainsi tenir sur un peu plus d'un centimètre de large. Le contrecoup de ce type de technologie est, bien entendu, la relative fragilité de la connectique.

3. EXPLORATION DE LA BIBLIOTHÈQUE : LES IMAGES

L'écriture de texte est la base de travail si l'on considère que l'objet même d'un écran e-paper est de servir d'étiquette électronique. Mais pour des applications un peu plus amusantes, il devient nécessaire d'utiliser des choses plus... graphiques. Il faut cependant distinguer plusieurs types de graphismes, que je classerai arbitrairement en trois catégories :

- Les tracés simples que l'on qualifie généralement de primitives graphiques : des cercles, des disques, des lignes, des boites, des triangles, des rectangles pleins, etc.
- Les graphismes simples de petite taille, comme des glyphes ou des icônes.
- Les images emplissant généralement tout l'écran, composées sur PC ou Mac à l'aide d'un logiciel de dessin ou de retouche d'images, puis converties en tableaux de données pour être utilisées dans le code.

Le premier type est relativement abordable puisqu'il s'agit, tout simplement, d'utiliser les fonctions mises à disposition par la bibliothèque et plus exactement par les méthodes héritées de *Adafruit_GFX*. Voici un court exemple :

```
display.setRotation(3);
display.fillScreen(GxEPD_WHITE);

display.fillCircle(GxGDEW0213Z16_HEIGHT/2-1,
  GxGDEW0213Z16_WIDTH/2-1, 20, GxEPD_BLACK);
display.writeLine(0, 0, GxGDEW0213Z16_HEIGHT-1,
  GxGDEW0213Z16_WIDTH-1, GxEPD_RED);
display.writeLine(GxGDEW0213Z16_HEIGHT-1, 0, 0,
  GxGDEW0213Z16_WIDTH-1, GxEPD_RED);

display.update();
```

Nous faisons abstraction ici de tout le reste du croquis qui ne change pas (déclaration, initialisation, etc.). Ces fonctions sont presque toutes construites de la même façon et on les retrouve sous diverses formes dans toutes les bibliothèques graphiques, que ce soit pour les microcontrôleurs ou en programmation sur PC. Les arguments précisent généralement des coordonnées de départ avec *0,0* situés en haut à gauche de l'écran, ainsi que d'autres paramètres variables comme le diamètre pour les cercles et disques, les coordonnées d'un coin opposé pour un rectangle et tantôt une taille horizontale et verticale.

Ici, nous nous contentons de tracer un disque noir en plein centre de l'écran, ainsi que deux lignes diagonales rouges (ou jaune), afin de confirmer le bon fonctionnement du système de coordonnées (le centre du disque doit correspondre au point d'intersection des deux lignes). Notez l'utilisation des macros `GxGDEW0213Z16_HEIGHT` et `GxGDEW0213Z16_HEIGHT` définies dans `GxGDEW0213Z16.h` et faisant référence aux tailles horizontale et verticale de l'écran en question. Bien entendu, comme nous avons tourné l'écran de 270°, les deux dimensions sont inversées.

Les graphismes de petite taille sont gérés sous la forme de blocs de données stockés en tableaux en flash. Le développeur de la bibliothèque en intègre une importante quantité (163) dans son code, dans le sous-répertoire `src/imglib/`. On y trouve de tout, du cœur au sablier en passant par le dossier, la cloche, le nuage ou encore la roue dentée. Chacune de ces mini-images de 24 pixels de côté, qu'on pourrait qualifier d'icônes, est stockée dans un fichier distinct. Exemple :

```
#if defined(ESP8266) || defined(ESP32)
#include <pgmspace.h>
#else
#include <avr/pgmspace.h>
#endif
// 24 x 24 gridicons_bell
const unsigned char gridicons_bell[] PROGMEM = {
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFE, 0x1F, 0xFF, 0xF8, 0x07, 0xFF, 0xF0,
0x03, 0xFF, 0xF0, 0x01, 0xFF, 0xE0, 0x00, 0xFF,
0xE0, 0x00, 0x7F, 0xE0, 0x00, 0x07, 0xE0, 0x00,
0x0F, 0xF0, 0x00, 0x1F, 0xF0, 0x00, 0x3F, 0xF8,
0x00, 0x7F, 0xFC, 0x00, 0xFF, 0xFE, 0x01, 0xBF,
0xFF, 0x03, 0x3F, 0xFF, 0x86, 0x3F, 0xFF, 0x8F,
0xFF, 0xFF, 0x9F, 0xFF, 0xFF, 0xBF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
```

L'image faisant 24×24 pixels, soit 576, nous retrouvons là 576/8=72 octets. Pour utiliser ce type d'images gentiment fournies, tout ce que nous avons à faire est d'inclure le fichier en question :

```
#include <imglib/gridicons_bell.h>
```

Puis de placer l'image avec :

```
display.drawBitmap(gridicons_bell, 10, 6, 24, 24, GxEPD_RED);
```

Les écrans bruts vendus sur eBay par Waveshare sont systématiquement livrés avec une « rallonge » flatflex de 20 cm et un adaptateur permettant de facilement déporter l'affichage pour des mises en œuvre spécifiques.

Les arguments de **drawBitmap()** sont respectivement, la variable à utiliser, la position en x et en y, la taille en largeur et hauteur et la couleur à utiliser pour le dessin. Un argument supplémentaire peut être utilisé pour spécifier un mode impactant la façon dont les données de l'image sont ajoutées à la mémoire qui représente l'écran. On peut ainsi utiliser **GxEPD::bm_invert** pour un négatif, **GxEPD::bm_r90**, **GxEPD::bm_r180** ou **GxEPD::bm_r270** pour une rotation, **GxEPD::bm_flip_x** ou **GxEPD::bm_flip_y** pour une inversion horizontale ou verticale, ou encore **GxEPD::bm_transparent** pour un effet de transparence où les pixels blancs de l'image laisseront voir les pixels noirs déjà présents.

Ce type d'images, en particulier avec la collection fournie par le développeur, est très utile en tant qu'icônes, accompagnant, par exemple du texte :

```
  display.fillScreen(GxEPD_WHITE);
  display.setFont(&FreeMonoBold9pt7b);
  display.drawBitmap(gridicons_bell,     10, 6, 24, 24, GxEPD_RED);
  display.setCursor(3+10+24, 6+24-8);
  display.setTextColor(GxEPD_RED);
  display.println("Alerte");
  display.drawBitmap(gridicons_offline, 10, 6+((24+10)*1), 24, 24, GxEPD_
BLACK, GxEPD::bm_invert);
  display.setCursor(3+10+24, 6+(10*1+24*2)-8);
  display.setTextColor(GxEPD_BLACK);
  display.println("Hackable 27");
  display.drawBitmap(gridicons_history, 10, 6+((24+10)*2), 24, 24, GxEPD_
RED, GxEPD::bm_invert);
  display.setCursor(3+10+24, 6+(10*2+24*3)-8);
  display.setTextColor(GxEPD_RED);
  display.println("En attente...");
  display.update();
```

La seule partie réellement difficile est l'alignement des éléments puisque le texte est écrit à partir d'une position inférieure du curseur, mais les images sont dessinées à partir de leur premier pixel, dans le coin supérieur gauche. En combinant cela avec les primitives

graphiques, il est possible de très simplement composer une image complète, et donc un écran, relativement sympathique à l'œil.

Enfin, nous arrivons à la partie qui m'aura sans doute donné le plus de fil à retordre, l'affichage de graphismes couvrant tout l'écran, dont les données sont tirées d'images composées pour l'occasion sur PC. La bibliothèque fournit un certain nombre d'exemples, mais ne précise aucun format spécifique, ni aucune méthode pour la conversion. Au mieux, nous avons l'exemple `GxEPD_SD_Example.ino`, mais celui-ci utilise un format BMP avec des images stockées sur un support SD. Et bien entendu, la taille des BMP fournis ne correspond pas à la résolution de l'écran de 2,13 pouces et le code est assez dense.

En tentant d'utiliser des données issues d'une image au format XBM, l'affichage ne donne pas les résultats espérés. En effet, l'ordre des bits dans les données ne correspond pas à celui attendu par l'écran. Il en résulte une inversion horizontale des pixels par paquet de huit. La solution la plus simple est de modifier la bibliothèque utilisée.

En éliminant la nécessité du support externe de stockage pour les images (une carte SD), on se retrouve en toute logique avec ces données devant être placées en flash. Sur cette base, le fait que les images soient dans un format standard simple et non compressé comme BMP ou sous la forme de données brutes, déjà prêtes à l'emploi, ne fait pas grande différence. De plus, l'aspect pratique d'un stockage sous la forme de fichiers PC standard m'échappe un peu. L'objectif d'un module e-paper de ce genre n'est pas de servir de cadre photo numérique où les images sont fréquemment mises à jour. Non, l'idée est ici d'avoir une bibliothèque/collection d'images préparées qu'on affichera en fonction des besoins, exactement comme pour les icônes.

Mon approche sera donc la suivante : composer une ou plusieurs images à la dimension de l'écran, puis, avec un outil de retouche comme The Gimp, exporter l'image dans un format qui sera simple à utiliser dans le croquis. Le choix de ce format est parfaitement logique, dès lors qu'on sait qu'il a été originellement créé spécifiquement pour intégrer des images dans du code, c'est XBM ou X BitMap. Faites quelques essais de conversion et vous constaterez que le contenu d'un fichier au format XBM n'est rien d'autre qu'une déclaration de variable en C, accompagnée de quelques macros. Exactement ce qu'il nous faut et tout à fait similaire au contenu des fichiers présents dans le répertoire `src/imglib/` de la bibliothèque.

Le format XBM permet de stocker des images monochromes. Comment alors pourrons-nous utiliser l'affichage en trois couleurs ? La réponse est simple, nous avons un fichier, ou bloc de données, pour le noir sur fond blanc et un autre pour le rouge/jaune sur fond blanc. Les deux seront alors combinés lors de l'appel à la fonction/méthode adéquate qu'il faudra aller chercher dans le code de la bibliothèque : `drawPicture()`.

L'opération de conversion est simple : prenez une image, adaptez sa taille à celle de l'écran, passez-la en noir et blanc (attention, pas en niveaux de gris), puis enregistrez-la au format XBM. Vous obtiendrez alors quelque chose comme ceci :

```
#define weeping_width 104
#define weeping_height 212
static char weeping_bits[] = {
  0x95, 0x54, 0x03, 0x60, 0xF8, 0x4F,
  0x80, 0x11, 0x06, 0x02, 0x60, 0xF8,
  0x1C, 0xC0, 0x53, 0xA6, 0x0A, 0x60,
  0xFF, 0x1C, 0xC0, 0x52, 0x21, 0x00,
[...]
  0x80, 0x00, 0x07, 0x04, 0x10, 0x1C, };
```

Transformez alors cela en fichier d'entête (`.h`) adapté au « langage » Arduino en modifiant quelques lignes :

```
#ifndef _monimage_
#define _monimage_

#include <pgmspace.h>

#define weeping_width 104
#define weeping_height 212
const unsigned char weeping_bits[] PROGMEM = {
  0x95, 0x54, 0x03, 0x60, 0xF8, 0x4F,
  0x80, 0x11, 0x06, 0x02, 0x60, 0xF8,
  0x1C, 0xC0, 0x53, 0xA6, 0x0A, 0x60,
  0xFF, 0x1C, 0xC0, 0x52, 0x21, 0x00,
[...]
  0x80, 0x00, 0x07, 0x04, 0x10, 0x1C, };
#endif
```

Notez le passage de `static` en `const` et l'ajout de `PROGMEM` pour un stockage en flash. Faites de même avec l'image contenant les données pour la partie rouge/jaune et vous vous retrouverez avec deux fichiers contenant vos données, ici `weeping.h` et `weepingj.h`, regroupant les déclarations des variables `weeping_bits[]` et `weepingj_bits[]`, qu'il vous suffira d'ajouter à votre projet dans l'IDE Arduino.

Jusqu'ici, tout va bien, mais en utilisant la méthode avec ces données :

```
display.drawPicture(weeping_bits, weepingj_bits,
  sizeof(weeping_bits), sizeof(weepingj_bits),
  GxEPD::bm_invert_red|GxEPD::bm_invert);
```

quelque chose de très problématique arrive : l'image est complètement chaotique et les pixels semblent mélangés. Que se passe-t-il ? Le problème ne vient pas du format XBM ou de la fonction utilisée, mais de la combinaison des deux et plus exactement de la façon dont les bits sont agencés dans le fichier par rapport à l'ordre dans lequel ils sont attendus par l'écran. En observant attentivement l'écran, on remarque, en effet, qu'il y a un certain ordre dans le chaos et que ce sont des groupes horizontaux de 8 pixels qui sont permutés.

Avant de régler ce problème, précisons les arguments à utiliser pour `drawPicture()` et, avant toutes choses, qu'il n'est pas nécessaire d'utiliser `update()` pour appliquer les changements. Ceci est directement pris en charge par `drawPicture()`. On passe en argument,

dans l'ordre, le tableau des données pour le noir, la même chose pour le rouge/jaune, la taille des données pour le noir, la même chose pour le rouge/jaune et enfin un mode, comme précédemment. Ici, nous utilisons `GxEPD::bm_invert_red` pour inverser l'image de la partie rouge/jaune, et `GxEPD::bm_invert` pour faire de même avec le noir. De cette manière l'image apparaît exactement comme sa version d'origine, sur PC.

Ces options de modes sont précisément là où nous allons intervenir car, plutôt que de modifier sauvagement le code de la bibliothèque, nous pouvons tout simplement ajouter un mode, arbitrairement nommé `GxEPD::bm_xbm`. Lorsqu'il sera activé, l'ordre des octets sera changé de façon à correspondre à celui attendu par le matériel

4. MODIFICATION DE LA BIBLIOTHÈQUE

Les différents modes utilisables sont déclarés dans le fichier `src/GxEPD.h` sous la forme d'une énumération :

```
class GxEPD : public GxFont_GFX
{
  public:
    enum bm_mode
    {
      bm_normal = 0,
      bm_default = 1,
      // these potentially can be combined
      bm_invert = (1 << 1),
      bm_flip_x = (1 << 2),
      bm_flip_y = (1 << 3),
      bm_r90 = (1 << 4),
      bm_r180 = (1 << 5),
      bm_r270 = bm_r90 | bm_r180,
      bm_partial_update = (1 << 6),
      bm_invert_red = (1 << 7),
      bm_transparent = (1 << 8)
    };
```

Chacun de ces modes correspond donc à une valeur qui est un exposant de 2, listé de 0 à 256. Remarquez que `bm_r270` est en réalité la combinaison de `bm_r90` et `bm_r180`, soit 16 OU 32, 00010000 OU 00100000, soit 00110000, ou encore 48. Cette énumération est ensuite utilisée dans la partie du code qui gère effectivement l'affichage, qui est propre à chaque modèle d'écran supporté. C'est donc tout naturellement, pour le modèle 2,13 pouces tricolore, dans le fichier `src/GxGDEW0213Z16/GxGDEW0213Z16.cpp` que nous pourrons voir de quoi il retourne.

Après l'ajout d'une fonctionnalité spécifique au support du format XBM, la bibliothèque GxEPD est en mesure d'afficher notre image à la perfection et celle-ci apparaît dans toute sa splendeur. Don't blink !

L'une des utilisations envisagées pour ces écrans dans l'avenir consistera à les intégrer dans des ampèremètres analogiques. Il sera alors possible d'afficher un cadran totalement personnalisable capable de présenter n'importe quel type d'information sous une forme des plus amusantes.

En « désemberlificotant » le code et les appels entre les méthodes, deux fonctions sont directement concernées, `GxGDEW0213Z16::drawPicture` permettant l'affichage d'une image sur tout l'écran (sans `update()`) et `GxGDEW0213Z16::drawBitmap`, utilisé pour l'affichage des « icônes ». En principe, nous n'avons pas besoin de toucher à cette dernière, mais autant ajouter notre fonctionnalité partout où elle peut être utile.

Pour trouver l'endroit précis où il serait le plus judicieux de faire notre modification, nous n'avons qu'à nous servir d'un autre mode. Or justement, `bm_invert` est utilisé pour éventuellement inverser les bits présents dans les données pour obtenir une image négative :

```
if (mode & bm_invert) data = ~data;
```

Tout ce que nous avons à faire est donc d'ajouter, juste après les différentes occurrences de cette ligne, un petit bout de code modifiant l'ordre des bits. C'est un problème classique en programmation et il est relativement simple de reposer sur l'expérience et le savoir commun accumulé sur Internet. Notre bout de code sera donc une méthode concise, éprouvée et efficace pour une telle opération :

```
if (mode & bm_xbm) {
  data = ((data * 0x0802LU & 0x22110LU) |
          (data * 0x8020LU & 0x88440LU))
         * 0x10101LU >> 16;
}
```

Si le nouveau mode `bm_xbm` doit être utilisé, `data` contiendra alors la version réordonnée des données. Exactement de la même façon qu'avec la ligne qui teste `bm_xbm` et inverse

les bits. Ce code devra être ajouté à trois endroits différents : deux fois dans la portée de `GxGDEW0213Z16::drawPicture` (une pour le noir et une pour le rouge/jaune) et une fois dans `GxGDEW0213Z16::drawBitmap`.

Ceci fait, nous n'aurons plus qu'à modifier `src/GxEPD.h` pour ajouter notre `bm_xbm` :

```
[...]
      bm_partial_update = (1 << 6),
      bm_invert_red = (1 << 7),
      bm_transparent = (1 << 8),
      bm_xbm = (1 << 9)
    };
```

Enfin, nous pouvons revoir notre appel à `drawPicture()` ainsi :

```
display.drawPicture(weeping_bits, weepingj_bits,
  sizeof(weeping_bits), sizeof(weepingj_bits),
  GxEPD::bm_invert_red|GxEPD::bm_invert|GxEPD::bm_xbm);
```

Et, bien entendu, l'image s'affiche alors parfaitement !

POUR FINIR

Si vous n'avez pas envie de modifier la bibliothèque *GxEPD* vous-même, sachez que cette version modifiée est disponible dans un de mes dépôts GitHub (**https://github.com/Lefinnois/GxEPD**). La modification a été apportée à la majorité du code supportant les écrans Waveshare mais, bien entendu, ceci n'a été testé qu'avec les modèles en ma possession (2,13 pouces tricolores et souples). Je ne saurai garantir le fonctionnement sur du matériel que je ne possède pas. Accessoirement, j'aurai bien proposé cette évolution au développeur original mais, comme dit précédemment, celui-ci ne semble pas vouloir reposer sur les mécanismes de *Pull Requests* de GitHub et, pour ma part, je n'ai pas envie d'utiliser un forum que je considère comme n'étant pas vraiment fait pour cela.

En dehors de cet aspect « social », l'association du matériel avec le fruit du travail de *J-M Zingg* est très bénéfique. Je n'ai pas regardé en détail le code proposé par Waveshare, mais celui-ci me paraît bien loin d'égaler *GxEPD*. Cette bibliothèque cependant pourrait être grandement allégée car, si vous y jetez un œil, vous constaterez qu'un certain nombre de fonctions sont inutiles (`drawExamplePicture()`) pour un usage courant et que l'ensemble pourrait être plus modulaire.

En termes d'utilisations pratiques de ces afficheurs, les idées ne manquent pas. Personnellement, la prochaine étape consistera sans doute à travailler sur une modification d'un ampèremètre analogique (tels ceux évoqués dans un autre article de ce numéro). En remplaçant le fond métallique de ce type de matériel par un écran e-papier, il devient possible d'afficher, de façon originale, absolument ce qu'on veut. On rentre ici, je trouve, dans le domaine parfait pour ces petits écrans e-paper puisqu'il ne s'agit pas de fréquemment mettre à jour l'affichage, mais qu'on souhaite toutefois conserver une grande souplesse de personnalisation. En réunissant cette technologie avec un cadran à aiguille et MQTT (oui, encore), on arrive littéralement au niveau de ce que peut faire Grafana, mais « en dur ».

Je ne doute pas, cependant, que vous aurez d'autres idées et, oui, l'idée du disque de stationnement qui change d'heure tout seul m'a traversé l'esprit et, non, je ne me risquerai pas à en faire un article... DB

UNE JAUGE ANALOGIQUE MQTT POUR AFFICHER VOS MESURES ET VALEURS

Denis Bodor

Comme nous l'avons vu dans le précédent numéro, MQTT est un véritable bonheur, en particulier lorsqu'il s'agit de collecter, communiquer et afficher des valeurs provenant de capteurs. Par habitude ou par convention, on utilise généralement des applicatifs comme Grafana pour produire une interface graphique attrayante présentant ces valeurs, avec des jauges et des courbes. Une jauge toute flashy dans un navigateur, c'est beau et pratique, mais une jauge bien rétro et bien physique, c'est encore mieux !

Dans le, maintenant lointain, numéro *Hackable 10*, nous avions couvert la réalisation d'une horloge faisant usage d'ampèremètre analogique afin de présenter de façon originale les heures et les minutes courantes. Avant toutes choses, précisons que le terme « ampèremètre » ne désigne pas ici un appareil de mesure similaire à votre multimètre digital, mais un objet relativement simpliste, constitué d'une simple bobine montée sur un axe rotatif, entourée d'un aimant et retenue par un ressort. Le fait de faire passer un courant dans la bobine crée un champ magnétique qui s'aligne sur les pôles de l'aimant, proportionnellement au courant qui circule, et l'aiguille montée sur l'axe indique le courant en question en ampères.

Une carte Arduino classique comme la UNO (mais pas la DUE par exemple), un ESP8266 ou encore un ESP32 n'est pas en mesure de fournir un courant, ou une tension, variable sur l'une de ses sorties. Il n'est donc, en principe, pas possible d'ajuster le courant et donc la position de l'aiguille de l'ampèremètre. Ce qu'il est possible de faire cependant, c'est de moduler l'état d'une sortie de façon à la passer à l'état haut pendant un temps suffisant pour positionner l'aiguille. Ce type de sortie pseudo-analogique est précisément celle qu'on obtient avec la fonction `analogWrite()`, en utilisant la PWM (*Pulse Width Modulation* ou modulation de largeur d'impulsions en français) pour jouer sur l'intensité lumineuse d'une led.

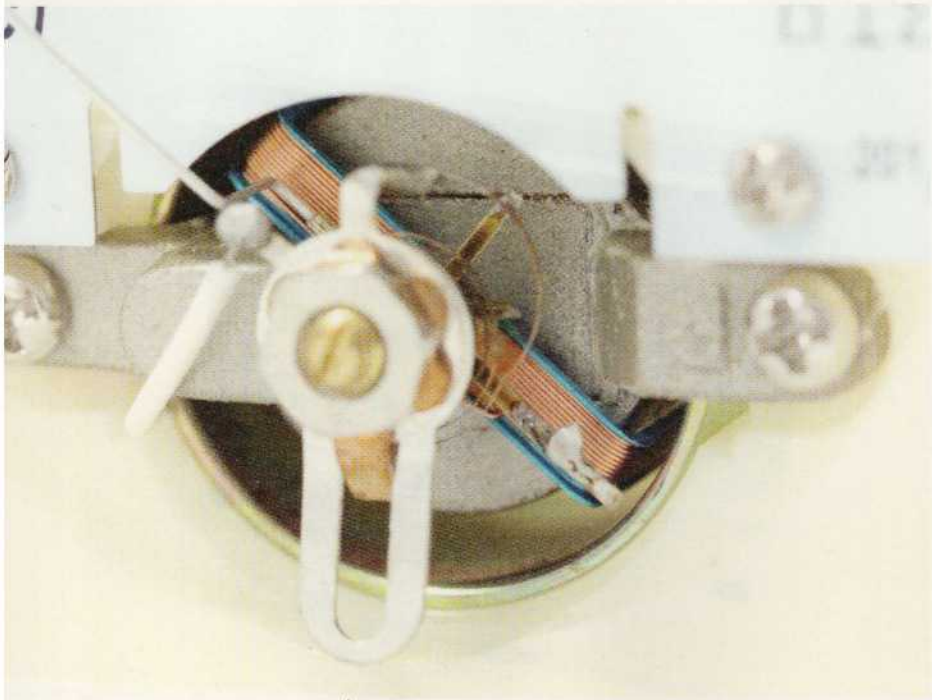
Voici typiquement le genre d'ampèremètre qu'on trouve en ligne entre 3 et 4 euros. Il existe d'autres déclinaisons à un prix sensiblement supérieur avec un aspect un peu plus ancien, un boîtier métal, un cadran de grande taille ou encore avec une forme ronde. Il y en a pour tous les goûts...

C'est un peu comme conduire un véhicule n'ayant que deux niveaux d'accélération : l'accélérateur non utilisé et le pied au plancher. En appuyant régulièrement sur l'accélérateur ainsi, durant une période plus ou moins importante, on peut théoriquement atteindre la vitesse de son choix. Plus longues sont les pressions sur l'accélérateur, plus le véhicule se rapprochera de sa vitesse maximum. Pour désigner la proportionnalité entre les périodes de non-accélération et d'accélération (état haut et bas), on parle de rapport cyclique, généralement exprimé sous la forme d'un pourcentage.

Il est donc possible, avec la PWM et `analogWrite()`, de contrôler la position de l'aiguille de l'ampèremètre. Tout ce que nous avons à faire est d'ajouter une résistance limitant le courant maximum pouvant circuler dans la bobine.

1. PILOTONS NOTRE AMPÈREMÈTRE

Un ampèremètre analogique comme ceux qui nous intéressent ici se trouve très facilement sur des sites comme eBay pour moins de quatre euros pièce. Ils se déclinent généralement en plusieurs modèles



Les différentes parties composant l'ampèremètre sont peu nombreuses : une bobine montée sur un axe, un aimant permanent, un ressort et quelques fils pour connecter le tout.

dépendants de la gamme de valeurs de courant mesuré : 0-1mA, 0-10mA; 0-50mA, 0-100mA, 0-1A, etc. Ceci est important pour deux raisons, d'une part le courant maximum mesuré impacte la résistance à utiliser pour atteindre la position maximum et, ensuite, le fait que le courant qui circule dans la bobine impactera forcément l'autonomie d'un projet sur batterie ou accu.

Personnellement, j'ai tendance à préférer les modèles avec un maximum à 10mA. Bien que ceux-ci consomment un peu plus de courant qu'un équivalent à 1mA, la souplesse d'utilisation est grandement améliorée, ne serait-ce que par la possibilité d'utiliser une résistance de 220 ohms, 330 ohms ou 470 ohms, en fonction de la tension utilisée par la sortie de la carte Arduino ou ESP8266.

Calculer la valeur de la résistance est très simple et repose sur la loi d'Ohm : U = R x I. Si notre sortie a un état haut à 3,3 volts, nous avons :

```
U = R x I
3,3 V = R x 0,01 A
3,3 / 0,01 = R = 330 ohms
```

Il nous suffit donc de relier une sortie de l'ESP8266 via une résistance de 330 ohms, puis de passer cette sortie à l'état haut pour catapulter l'aiguille sur la position maximum de 10mA. En utilisant alors la fonction `analogWrite()` pour fixer un rapport cyclique de 50%, l'aiguille devrait arriver à 50% de son parcours, et donc sur 5mA... en théorie.

Il y a un autre composant à ajouter dans cet assemblage car, même si cela pourrait parfaitement convenir pour un fonctionnement « normal » d'un tel ampèremètre, il ne faut pas perdre de vue que nous l'utiliserons avec la sortie d'un microcontrôleur. Or justement, s'il y a bien quelque chose que ces sorties détestent, c'est qu'on leur applique une tension en entrée. Une telle erreur, au mieux réduira la durée de vie du composant et, au pire, détruira simplement la sortie. « Pourquoi diable appliquerions-nous une tension sur une sortie ? » me direz-vous. La réponse est simple : pas nous, la bobine.

Avec un ampèremètre analogique, comme avec n'importe quel élément comportant une bobine, comme un relais, un phénomène spécifique apparaît. Lorsqu'on fait circuler un courant dans une bobine, un champ magnétique est créé. Celui-là même qui fait cliquer un relais ou déplacer l'aiguille sur le cadran. Ce champ magnétique existe tant que le courant circule, mais dès lors qu'on coupe l'alimentation, il s'effondre. L'énergie ainsi stockée doit aller quelque part et un courant circule alors dans la bobine dans le sens opposé : une tension apparaît entre ses bornes et est donc appliquée entre la masse et la sortie du microcontrôleur !

Pour éviter ce phénomène et surtout ces conséquences fâcheuses, il faut fournir un autre

chemin au courant qui apparaît à ce moment critique : on place alors une diode dite « de roue libre » entre les bornes de la bobine, avec la cathode de la diode reliée à la sortie du microcontrôleur et l'anode à la masse. Ainsi lorsque la sortie est à l'état haut, la diode ne conduit pas et le courant passe dans la bobine. Mais lorsqu'on passe la sortie à l'état bas, le champ magnétique qui s'effondre fait circuler un courant, via la diode et celui-ci est alors dissipé. N'importe quelle diode courante fera l'affaire, comme la très classique 1N4148.

2. PROBLÈME DE PRÉCISION ET SOLUTION

Après cette description matérielle somme toute relativement simpliste, dans les grandes lignes, côté logiciel, piloter la position de l'aiguille de l'ampèremètre n'est pas plus compliqué. En effet, ceci se limite à l'utilisation de la fonction `analogWrite()` après avoir passé la broche concernée en sortie. Mais là encore... en théorie.

L'essai est relativement simple à faire, par exemple, avec un ESP8266. La fonction `analogWrite()` prend en argument la broche à utiliser et une valeur entre 0 et 1023 correspondant à un rapport cyclique entre 0 et 100%. Un tout petit croquis à compiler et charger sur l'ESP8266 permet de rapidement tester quelques valeurs et de vérifier la position de l'aiguille :

```
#define AMP_PIN D2

void setup() {
  pinMode(AMP_PIN, OUTPUT);

  int valeur = 512;

  analogWrite(AMP_PIN,
valeur);
}

void loop() {
}
```

Ce que vous constaterez sans doute dans un premier temps sera le fait que l'aiguille n'arrive pas exactement sur la dernière position du cadran lorsque le rapport cyclique est à 100% (`1023`). Bien entendu, le cadran dispose d'un réglage en façade permettant d'ajuster la position de l'aiguille, mais ce faisant, dès lors que la bobine n'est plus alimentée, l'aiguille ne marquera plus la position zéro. Avec d'autres essais, vous constaterez également que le rapport cyclique n'est pas exactement proportionnel à ce que montre le cadran, et ce de façon non linéaire. Par exemple, à 20% nous sommes légèrement au-dessus de 2mA, mais à 80% nous sommes juste un peu en dessous.

Une première tentative de résolution du problème serait de réduire la valeur de la résistance utilisée. Ainsi, à 100% de rapport cyclique, nous sommes au-delà de 10mA et il nous suffirait alors de déterminer que le maximum n'est pas une valeur de 1023 passée à `analogWrite()`, mais, par exemple, 797. Ceci règle le problème de l'aiguille n'atteignant pas la marque « 10 » sur le cadran, tout en ajustant manuellement la position 0. Il serait alors tentant de simplement utiliser la fonction `map()` fournie par les bibliothèques standards Arduino...

Cette fonction est relativement simple à utiliser et entre en jeu lorsqu'il s'agit de *mapper* une plage de valeurs sur une autre (comme un produit en croix). Sa syntaxe est `map(valeur, depuis_début, depuis_fin, vers_début, vers_fin)` et la valeur retournée correspond à celle ajustée d'une plage vers l'autre. Pour mapper une valeur en pourcentage sur la plage de valeur par défaut de `analogWrite()` sur ESP8266 (0-1023), nous pouvons utiliser `nouv_valeur = map(valeur, 0, 100, 0, 1023)`, voire directement utiliser `map()` avec `analogWrite()` avec `analogWrite(AMP_PIN,map(valeur, 0, 100, 0, 1023))`. Dans notre cas, en remplaçant `1023` par `797`, et avec une résistance de 220 ohms à la place de celle de 330 ohms, nous obtiendrions l'effet souhaité... ou pas.

Cette approche règle le problème de plage, mais pas celui de la non-linéarité des variations de décalage entre le rapport cyclique et la position de l'aiguille. Nous pouvons faire mieux : établir une série de valeurs pour `analogWrite()` pour un certain nombre de positions de l'aiguille et se baser sur ces valeurs pour mathématiquement déduire celles qui manquent. L'opération est relativement simple, utilisez le croquis précédent pour tâtonner afin de trouver les bonnes valeurs de 0 à 10mA par incrément de 1mA.

Dans mon cas, et avec une résistance de 220 ohms sur ESP8266 (3,3v), les valeurs sont les suivantes : 0, 83, 166, 250, 330, 410, 490, 570, 645, 722 et 797. Ce qui, transformé en un tableau d'entiers donne :

```
unsigned int calibration[11] = {
  0, 83, 166, 250, 330, 410,
  490, 570, 645, 722, 797
};
```

Cette table de calibration peut à présent nous servir de base pour une nouvelle fonction, plus ou moins équivalente à `map()`, mais combinant à la fois la fonction de mapping et l'utilisation de `analogWrite()` :

```
void setanalog(short val) {
  if (val >= 100) {
    analogWrite(AMP_PIN, calibration[10]);
    return;
  }
  if (val < 0) {
    analogWrite(AMP_PIN, calibration[0]);
    return;
  }
  analogWrite(AMP_PIN, val * calibration[val / 10 + 1] / ((val / 10 + 1) * 10));
  return;
}
```

Notre tableau possède une taille de 11 valeurs. Même si la présence d'une valeur 0 à la position 0 peut paraître surprenante, non seulement ceci permettrait de fixer une valeur minimum autre, mais ceci simplifie également la fonction et le calcul de la valeur à passer à `analogWrite()`. La logique est la suivante :

- la fonction prend en argument un pourcentage et donc un entier entre 0 et 100 ;
- en cas de maximum ou supérieur, ou de minimum ou inférieur, on utilise les valeurs correspondantes ;
- l'argument passé à la fonction, divisé par 10 est ensuite utilisé comme index du tableau, incrémenté de un, pour récupérer une valeur ;
- on fait alors un produit en croix pour déduire la valeur à passer à `analogWrite()` sur la base de l'argument et de la valeur immédiatement supérieure.

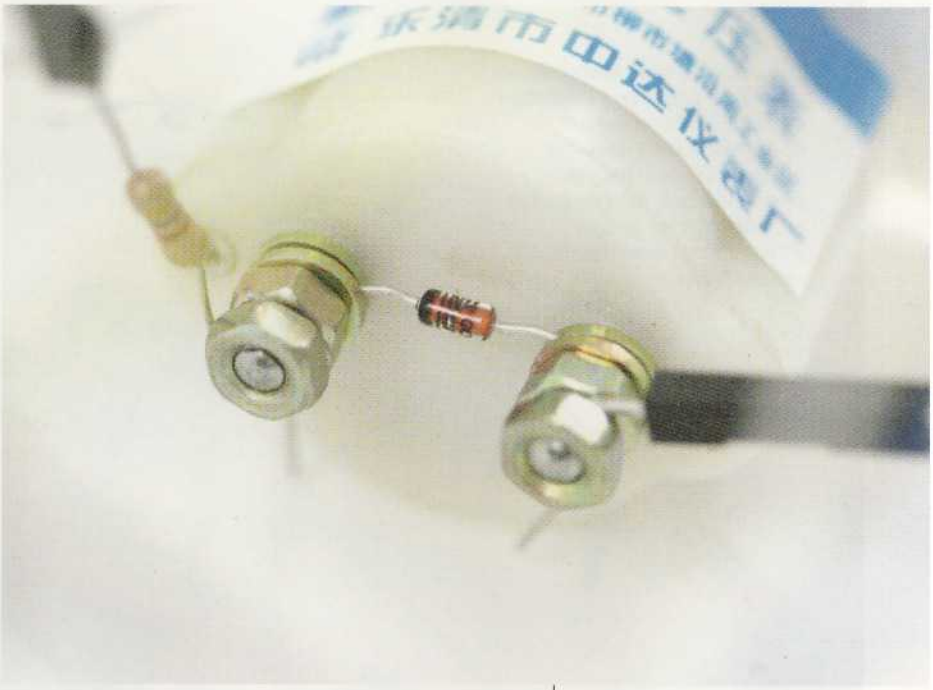
Dans notre tableau, par exemple, nous n'avons pas défini de valeur de calibration pour 17%. Comme on travaille avec des entiers, cette fonction va donc calculer 17/10=1, ajouter 1 et utiliser la valeur de `calibration[1]` qui vaut 166. Enfin, 166 va être multiplié pas 17 (`val`) pour devenir 2822, avant d'être divisé par 20 (`((17/10)+1)*10 = (1+1)*10 = 20`) pour devenir 141. C'est la valeur à passer à `analogWrite()` pour obtenir un rapport cyclique correspondant, plaçant l'aiguille à 17% de son parcours, et donc sur 1,7mA.

Cette fonction n'est pas parfaite, mais c'est une concession nécessaire pour conserver une certaine simplicité. En effet, en lui passant en argument une valeur qui correspond à une entrée du tableau, il est possible de se retrouver avec une valeur autre. Exemple, avec 50%, on utilise la valeur correspondant à 60, soit 490 et le calcul nous donne 50×490/60=408 et non 410 qui est la valeur définie arbitrairement pour la position 5.

Une approche possible aurait été de tester cette correspondance ou encore d'utiliser l'écart de valeurs entre les positions supérieures et inférieures et d'appliquer un décalage proportionnel. Mais tout ceci reviendrait à grossir notre fonction pour un effet relativement minime en termes de gain de précision. Le fait que la tension du ressort sur la bobine puisse changer en fonction de la température ou que le simple positionnement différent de l'ampèremètre (couché ou debout) impacte la position de l'aiguille, est pire que l'effet de notre approximation. Relativement parlant, ce ne sont donc pas des écarts de 2/1023 qui posent un véritable problème de précision au regard de la simplicité de la fonction.

3. UNE VRAIE JAUGE MQTT

À présent que nous sommes en mesure d'afficher précisément ce que nous voulons, pour une raison ou une autre, nous pouvons l'appliquer à ce qui nous chante. Non seulement nous pouvons recréer le montage du numéro 10 et ainsi créer une horloge plus précise que la précédente, mais aussi et surtout, nous pouvons mélanger cela avec nos connaissances acquises dans le précédent numéro.



Afin de se prémunir que tout problème, la première chose à faire lorsqu'on travaille avec un microcontrôleur et un composant reposant sur une bobine (relais, actuateur, etc.) est d'ajouter une diode de roue libre à ses bornes. Celle-ci permet d'éviter qu'une tension ne soit appliquée à la sortie du microcontrôleur lorsque le champ magnétique de la bobine s'effondre.

Je parle, bien entendu, de MQTT et donc de l'opportunité délicieuse de réunir l'aspect très vintage de l'ampèremètre analogique avec la modernité digitale et connectée d'une plateforme et d'un protocole contemporains. De l'extérieur, ce qui paraîtra être un simple cadran, indiquant par exemple une température, un taux d'hygrométrie ou une pression atmosphérique, sera en réalité un objet connecté rapportant des informations provenant de, littéralement, n'importe où.

Je ne reviendrai pas ici sur le principe de fonctionnement de MQTT, ni sur l'architecture à mettre en place ou encore la structure complète du croquis ESP8266. Le croquis complet, adapté à cet usage sera disponible dans le dépôt GitHub du magazine et je rappellerai simplement la notion de fonction par *callback*. Lorsqu'un message MQTT arrive, car l'ESP8266 s'est abonné au *topic*, une fonction est automatiquement exécutée, avec ce message en argument. À notre charge de rédiger cette fonction de façon à extraire les informations qui nous intéressent et les transformer pour en faire usage.

Les ampèremètres économiques de ce type présentent l'avantage de pouvoir être facilement démontés. Il suffit de retirer quelques vis pour pouvoir personnaliser le fond du cadran et ainsi l'adapter au type d'informations que l'aiguille est censée indiquer.

Par rapport aux croquis du numéro précédent, donc, la seule chose que nous avons à changer est précisément cette fonction de callback :

```
void callback(char* topic, byte* payload, unsigned int length) {
  String strPayload = "";
  Serial.print("Message [");
  Serial.print(topic);
  Serial.print("] ");
  // construction de la chaîne
  for (int i = 0; i < length; i++) {
    strPayload += (char)payload[i];
  }
  Serial.print("payload: ");
  Serial.print(strPayload);
  // conversion en int et "affichage" analogique
  setanalog(strPayload.toInt());
  Serial.println();
}
```

Le contenu du message (*payload*), tel qu'utilisé ici (et avec, par exemple, `mosquitto_pub` sur Raspberry Pi) est une série de caractères. Il ne s'agit pas d'une chaîne au sens Arduino (`String`) ou même C du terme (série de caractères `char` se terminant par `\0`). Nous devons donc forcément jongler un peu et la façon la plus simple, mais certainement pas la plus effi-

cace, est de passer par la création d'un objet de type `String`. La variable `strPayload` que nous déclarons contient une chaîne vide et nous concaténons chaque caractère un à un sous la forme d'une boucle `for` utilisant la valeur contenue dans `length` (le nombre de caractères reçus). Une fois cette chaîne composée, il nous suffit d'utiliser la méthode `toInt()` pour obtenir un entier que nous nous empressons de passer à notre fonction `setanalog()`.

4. ET POURQUOI PAS DES FLOAT ?

Nous partons ici du principe qu'il s'agit d'une jauge et de ce fait que l'entier contenu dans le message ait une valeur entre 0 et 100%. Cette approche nous permet de simplifier la fonction `setanalog()`, son utilisation et celle de précisément 11 valeurs de calibration (0 à 10). Cependant, si l'on utilise par exemple des mesures provenant de capteurs de température, un pourcentage n'est sans doute pas l'unité la plus pratique.

Pour éviter de devoir réviser le code des capteurs, qui n'ont absolument aucune raison de publier des valeurs en pourcentage dans ce cas, nous pouvons étoffer un peu notre présent croquis, à la fois pour prendre en compte une valeur en virgule flottante, mais également prendre en charge une notion de marge. En effet, si vos capteurs de températures sont dispersés dans un habitat, la plage de températures mesurées peut être très variable. Un lieu de vie pourrait raisonnablement varier de 17°C à 35°C, mais un garage pourra fluctuer, selon la saison entre 0°C et 40°C. Pire encore, avec un capteur extérieur, exposé au soleil, les températures pourront s'étaler entre -20°C et 50°C tout au long de l'année. Tous les cadrans ne sont donc pas logés à la même enseigne.

L'idée est donc ici de prévoir une fonction capable de prendre la température qu'on lui donne et d'en déduire un pourcentage de progression sur toute la course de l'aiguille, en fonction du minima et du maxima également passés en argument. Inutile ici de réinventer la roue, une simple règle de trois et un appel à `setanalog()` (qui ne changera pas), fera l'affaire :

```
void setanalogf(float val, float minval, float maxval) {
  if(val<minval) val = minval;
  if(val>maxval) val = maxval;
  if(maxval-minval<=0) return;
  setanalog((val-minval)*100/(maxval-minval));
  return;
}
```

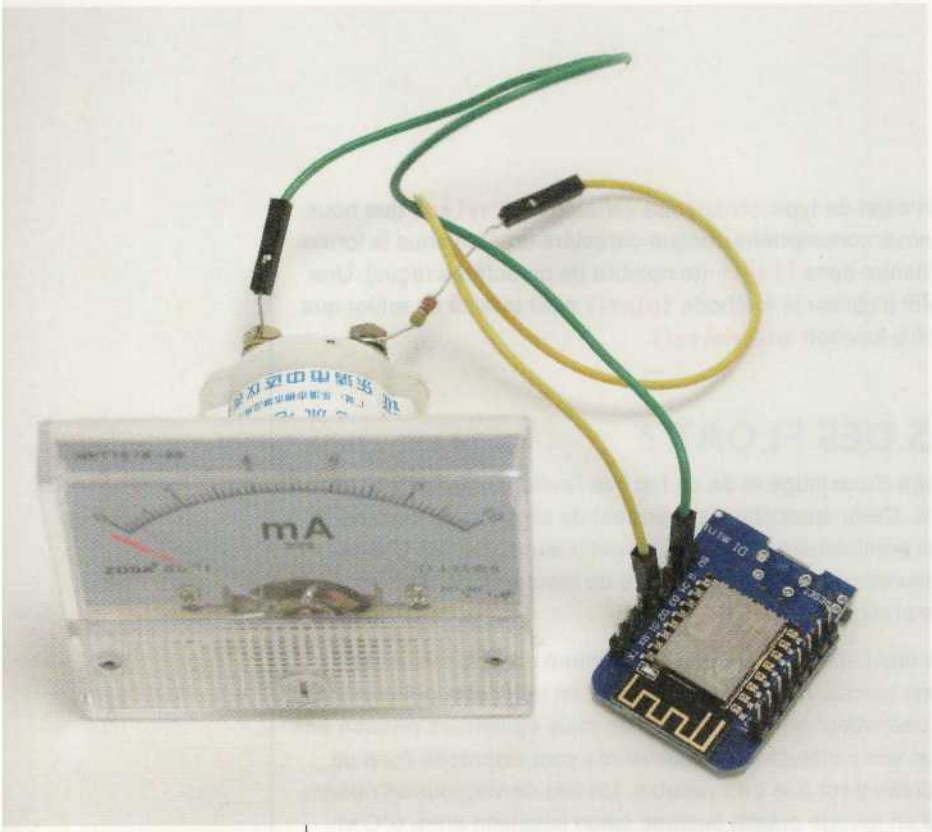
On pourra alors sensiblement modifier notre fonction callback en remplaçant simplement :

```
setanalog(strPayload.toInt());
```

en

```
setanalogf(strPayload.toFloat(),10.0,30.0);
```

Ce faisant, dès qu'un message sera publié sous le topic concerné en MQTT, par une sonde ESP8266 telle que celles que nous avons étudiées dans le numéro précédent, notre cadran/jauge analogique affichera la valeur. Il ne nous restera plus alors qu'à faire un peu de bricolage pour adapter visuellement l'ampèremètre à la plage de valeurs en imprimant un nouveau cadran sur, par exemple, du papier autocollant de type « étiquettes imprimables ».



Le montage dans son ensemble est excessivement simple puisqu'il ne nécessite, en plus d'une carte ou d'un module microcontrôleur Arduino ou ESP8266, qu'une résistance, une diode et deux malheureux câbles de connexion.

CONCLUSION ET PERSPECTIVES

Dans les grandes lignes, en comprenant le fonctionnement de MQTT et avec cette nouvelle fonction de calibration pour un ajustement plus précis de la position de l'aiguille, il devient très facile d'afficher peu ou prou n'importe quoi. Mieux encore, nous n'utilisons ici qu'une seule sortie de la plateforme ESP8266 et il est parfaitement possible de connecter plusieurs ampèremètres pour se créer un *bashboard* exactement comme nous le ferions avec Grafana. Avec un peu d'huile de coude et quelques notions de bricolage, un panneau industriel/vintage est parfaitement à la portée de n'importe qui. Bien entendu, cette mise en œuvre n'empêche en rien l'utilisation des techniques que nous avons déjà vues et donc d'avoir un affichage matériel pour l'aspect temps réel et, en même temps, Telegraf/Grafana pour le stockage des mesures et l'affichage de graphiques sur de longues périodes.

Côté croquis, une amélioration possible permettrait de rendre le montage générique. En effet, en oubliant le bricolage du cadran, si ce n'est pour le rendre indépendant des valeurs (absence de marquage textuel), il est parfaitement possible de faire en sorte que la position de l'aiguille dépende d'une plage paramétrable. En prévoyant que le croquis puisse prendre en compte, par exemple, un topic "param", on pourra alors formater un message (*payload*) sous la forme "numéro_cadran:valeur_min:valeur_max" avec comme exemple, la chaîne `1:15.0:45.0`. Il suffira alors au croquis de s'abonner au topic pour interpréter le message et ajuster sa configuration en conséquence, et de ce fait la position relative de l'aiguille. Il est même possible d'envisager un fonctionnement dynamique, en tirant les valeurs de minima et de maxima des messages précédents.

Au-delà de l'aspect vintage, il est également possible d'ajouter un peu de couleur. La plupart des modèles d'ampèremètre de ce type qu'on trouve sur eBay en provenance de Chine pour quelques euros possèdent une façade avant en plastique transparent. Intégrer des leds, qu'il s'agisse de leds standards, RGB ou des WS2812b (alias NeoPixels) peut être fait relativement facilement. Il serait alors possible, en plus de la position de l'aiguille, d'ajuster l'éclairage du cadran afin d'apporter une information supplémentaire : état « standard » en vert, « requiert de l'attention » en jaune/orange et « problème » en rouge. Un simple coup d'œil pourrait alors suffire pour s'assurer de la bonne marche du montage, qu'il s'agisse de températures, de pressions, d'hygrométrie ou de toute autre donnée potentiellement sensible ou importante.

Comme souvent dans les sujets abordés dans nos pages, les perspectives ne manquent pas et je ne doute pas un instant qu'il en existe une myriade d'autres auxquelles vous aurez pensé avant moi... **DB**

RÉSEAU MESH : ÉTENDRE FACILEMENT SON RÉSEAU SANS FIL POUR SES MONTAGES

Denis Bodor

Vous connaissez certainement la problématique wifi consistant à tenter de couvrir au mieux une surface donnée afin que vos différents montages puissent aisément communiquer, aussi bien dans votre salon qu'au fond du jardin. La plupart du temps, ceci sera solutionné par l'ajout d'un, voire plusieurs, points d'accès wifi, mais cela suppose un accès au réseau filaire. Il existe pourtant une autre solution : un réseau maillé...

Lorsqu'on souhaite déployer des montages communiquant sur une zone plus ou moins vaste, bien des options s'offrent à nous. Certaines sont plus économiques que d'autres, certaines utilisent des technologies plus communes que d'autres et certaines sont plus faciles à utiliser que d'autres. Dans notre domaine d'activité et avec les moyens facilement mis à notre disposition, si l'on cherche à obtenir quelque chose de relativement économique, on peut directement éliminer les technologies spécialisées et exotiques, comme ZigBee et consorts.

Reste alors l'adaptation des technologies réseau courantes qui se divisent en deux catégories : avec et sans fil. En d'autres termes, pour le commun des mortels, Ethernet et wifi. Bien entendu, la première catégorie est totalement dépendante de la présence de câbles réseau, mais aussi de la disponibilité de cette interface avec les plateformes, cartes ou modules utilisés. Pour une Raspberry Pi, le problème ne se pose pas, mais cet ordinateur mono-carte est, dans bien des cas, totalement surdimensionné par rapport aux tâches à accomplir (mesure, capteur, activation de relais, etc.). Un shield Ethernet et une carte Arduino sont envisageables, mais le coût est presque le même que celui d'une Pi. L'approche sera donc naturellement celle reposant sur le sans fil.

1. LE RÉSEAU MESH

Le wifi est naturellement l'option qui combine tous les avantages : pas de câbles à installer, universellement utilisé et, surtout, économique si l'on jette son dévolu sur ces cartes/modules à base d'ESP8266, comme des clones Wemos D1 mini chinois disponibles, en lot, pour quelques 30€ les 10.

Il reste cependant un problème, et de taille : le wifi c'est bien, mais pour couvrir une zone relativement importante, ou peuplée d'obstacles réticents à laisser passer les ondes radio à ~2,4Ghz (typiquement les murs bien massifs), il faut multiplier les points d'accès et donc, le plus souvent, utiliser tout de même le réseau filaire. Ceci est d'autant plus vrai avec des modules ESP8266 économiques, généralement équipés d'antenne PCB avec un gain très modeste. Selon la nature de la surface à couvrir, ceci peut être un véritable handicap en particulier dans les bâtisses anciennes (mur de pierre, granit, grès, briques, béton, etc.) ou encore les zones extérieures.

Il est, bien entendu, possible d'utiliser certaines astuces comme opter pour une antenne à haut gain pour votre point d'accès ou encore faire usage de répéteurs wifi (*wifi extender*). Ces derniers ne nécessitent qu'une alimentation et se contentent, comme leur nom l'indique, de répéter tout signal qui leur parvient en le réémettant. Mais ce genre d'équipement a un coût qui est bien supérieur à ce que l'on peut faire avec une autre solution : un réseau mesh.

L'architecture réseau la plus utilisée actuellement, où la topologie, est celle dite « en étoile ». Cette définition de la façon dont les équipements sont interconnectés repose sur l'utilisation d'un point central. Dans le cas des réseaux filaires Ethernet avec paires torsadées, ce point est le concentrateur ou hub. Dans le cas du wifi, c'est le point d'accès. Un autre type qui était utilisé pour Ethernet il y a quelques années (10base-2), utilisait une topologie en bus, où un câble coaxial unique, parcourant tout le réseau, permettait aux équipements de se connecter. Ce type de réseau présentait l'avantage de n'avoir qu'un câble à « tirer », mais toutes les données pour et depuis toutes les machines étaient transportées sur cet unique média, impliquant donc une vitesse relativement réduite. De plus, une rupture du câble rendait alors inaccessible une partie des équipements, tout en perturbant l'ensemble du réseau (problème de terminaison).

Le répéteur ou répétiteur wifi est l'option la plus économique pour étendre un réseau wifi. Son travail est de se connecter à votre point d'accès et de répéter le signal en l'amplifiant.

Cette topologie est toujours utilisée aujourd'hui, par exemple pour les bus i2c ou CAN (très présents dans l'automobile).

Il existe d'autres topologies, moins courantes, mais une seule en particulier nous intéresse ici : celle du réseau maillé, également appelée topologie mesh dans le domaine du wifi. Dans ce cas de figure, chaque entité du réseau est connectée pair à pair avec plusieurs autres et a pour tâche non seulement d'envoyer et de recevoir des données pour son propre compte, mais également de relayer les communications des entités avec lesquelles elle est connectée. Est alors formé un réseau en forme de filet où chaque entité est un nœud ou *node* en anglais.

Cette topologie permet non seulement de pallier à d'éventuels problèmes de disponibilité de certains nœuds puisqu'une autre route est potentiellement utilisable, mais ceci permet également de très facilement étendre un tel réseau. Il suffit, en effet, d'ajouter des nœuds en périphérie et automatiquement d'autres nœuds pourront être placés dans leur portée et ainsi de suite.

En ce qui concerne la technologie wifi, il existe des standards spécifiques pour les réseaux mesh comme 802.11s et des protocoles comme OLSR permettant de dynamiquement former une telle topologie. Bien entendu, il s'agit là de spécifications qui doivent normalement être implémentées directement dans les matériels ou dans leur firmware et ce ne sont pas des choses qu'on peut « bricoler » sur une plateforme comme ESP8266.

Espressif Systems, le constructeur des ESP8266, met à disposition des ressources orientées mesh sur le Web, mais ceci ne concerne malheureusement que l'ESP32, le grand frère de l'ESP8266 dont nous avons parlé dans *Hackable n°22*. Si nous voulons utiliser ce type de topologie avec un ESP8266, il faudra donc prendre un chemin de traverse. Là, bonne nouvelle, quelqu'un a déjà défriché ledit chemin avec une approche originale, mais parfaitement viable.

Ce quelqu'un est Germán Martín qui a créé la bibliothèque *painlessMesh*, elle-même dérivée de la bibliothèque *easyMesh* d'un certain *Coopdis*. *painlessMesh* utilise une idée relativement simple et, plutôt que de reposer sur une couche réseau inférieure ou supérieure (comme un protocole mesh au niveau TCP/IP), utilise les mécanismes de point d'accès et de client wifi standards. Ainsi, dans le réseau mesh composé d'ESP8266, chaque nœud est à la fois un point d'accès potentiel et un client pour un autre

ESP8266. On peut littéralement voir cela comme une flottille de points d'accès interconnectés.

La bibliothèque *painlessMesh* est directement installable depuis le gestionnaire de bibliothèques et fonctionnera aussi bien avec les ESP8266 que les ESP32, si vous disposez du support matériel dans votre environnement de développement Arduino. Notez au passage que depuis le numéro 22, la procédure pour installer le support ESP32 s'est grandement simplifiée. En effet, il suffit à présent d'ajouter une entrée dans « URL de gestionnaire de cartes supplémentaires », dans les préférences de l'IDE Arduino, **https://dl.espressif.com/dl/package_esp32_index.json**, et le support peut ensuite être ajouté, comme pour l'ESP8266, via le gestionnaire de cartes.

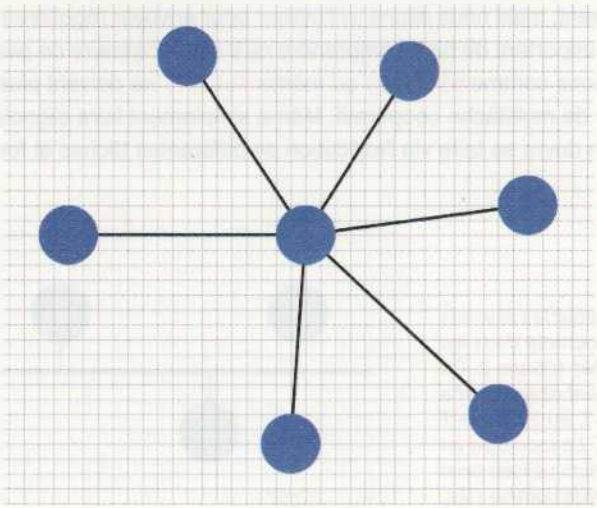
2. CONSTRUIRE UN RÉSEAU MESH AVEC DES ESP8266

La bibliothèque *painlessMesh* n'est pas la seule à devoir être installée, car, bien qu'elle se charge du travail de gestion de la topologie mesh, elle repose sur d'autres bibliothèques qu'il vous faudra donc également installer via le gestionnaire : *ArduinoJson*, *TaskScheduler* et *ESPAsyncTCP*. Ceci fait, vous pourrez alors procéder à vos tests avec un premier croquis.

La gestion d'une topologie mesh n'est pas chose facile en soi puisque chaque nœud doit jouer plusieurs rôles tant pour la structuration du réseau que dans le fait de véhiculer des informations propres à l'application pratique à mettre en œuvre. Si nous voulons que les nœuds relèvent une température, par exemple, cette opération vient en complément de l'activité même de maintenir la cohésion du réseau. Devant autant de choses à faire, le développeur de *painlessMesh* a fait le choix de reposer sur un système de gestion de tâches. Ce travail est supporté par la bibliothèque *TaskScheduler*, fournissant un ordonnanceur.

Un ordonnanceur est un élément fondamental d'un système d'exploitation moderne. Son travail consiste à gérer des processus ou tâches et de choisir l'ordre d'exécution de celles-ci. Le fait que le processeur dispose ou non de beaucoup de ressources, ou soit ou non performant, n'entre pas en ligne de compte, il ne s'agit que d'un bout de code décidant quel autre bout de code doit être exécuté à un instant donné. La bibliothèque *TaskScheduler* se propose de fournir ce type de mécanismes pour les cartes Arduino, c'est un ordonnanceur léger permettant un fonctionnement multitâche coopératif : sommairement, c'est à la charge de chaque tâche de « rendre la main » pour que l'ordonnanceur puisse en exécuter d'autres.

Sur un PC ou même une Raspberry Pi, le multitâche est préemptif, l'ordonnanceur décide quand une tâche est exécutée, mise en pause et stoppée, indépendamment de ce que souhaite la tâche en question. Encore une fois, cette description des approches de systèmes multitâches



La topologie réseau en étoile est sans le moindre doute celle qui aujourd'hui est la plus courante, tant au niveau filaire qu'en wifi. Deux problèmes accompagnent cette architecture : il y a une distance maximum du point central au-delà de laquelle la connexion n'est plus possible et si ce point a un problème (hub Ethernet ou point d'accès wifi), tout le réseau s'effondre.

est délibérément sommaire. Dans la réalité, l'ambivalence coopératif/préemptif n'est plus aussi nette, ne serait-ce qu'en raison de la notion de priorité, maintenant communément utilisée avec les systèmes d'exploitation courants.

En plus de *TaskScheduler*, *painlessMesh* utilise les bibliothèques *ESPAsyncTCP* pour la communication asynchrone entre les nœuds et *ArduinoJson* pour le formatage des messages de gestion de la topologie mesh, au format structuré JSON (*JavaScript Object Notation*) conforme à la RFC 4627.

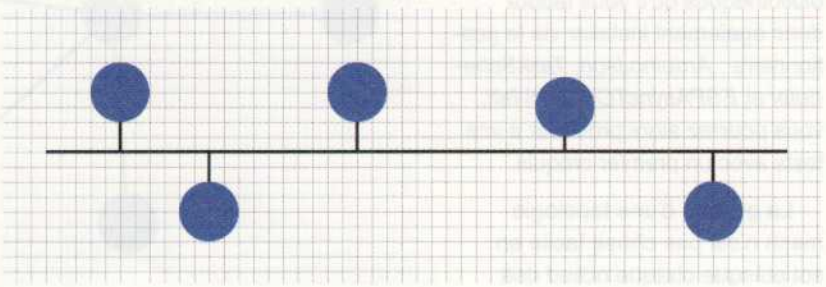
La théorie c'est bien beau, mais la pratique c'est mieux. Sans plus attendre donc, mettons-nous au travail pour mieux appréhender tout cela. Le croquis dont nous allons parler est honteusement tiré des exemples accompagnant la bibliothèque et est destiné à réaliser un premier test simple en utilisant plusieurs (au moins deux) ESP8266.

Nous commençons classiquement par l'inclusion des bibliothèques, la définition de macros et la déclaration des variables (et objets) globales :

```
#include <painlessMesh.h>

#define MESH_PREFIX     "CeQuiVousPlait"
#define MESH_PASSWORD   "QuelqueChoseDeSecret"
#define MESH_PORT       5555

Scheduler userScheduler;

painlessMesh  mesh;
```

`MESH_PREFIX` et `MESH_PASSWORD` seront utilisés pour créer un point d'accès afin que l'ESP8266 puisse accepter les connexions d'autres nœuds. Il s'agit donc respectivement du SSID et de la phrase de passe de ce point d'accès. Le port, identifié par `MESH_PORT` servira aux communications de gestion entre les nœuds à un autre niveau (TCP/IP).

`userScheduler` est un objet de type `Scheduler` fourni par la bibliothèque *TaskScheduler*. Il représente l'ordonnanceur chargé d'orchestrer l'exécution des différentes tâches que nous allons mettre en place. *TaskScheduler* est utilisé en interne par *painlessMesh* pour gérer le réseau dynamiquement, mais nous allons ici l'utiliser pour envoyer régulièrement des messages aux autres nœuds. Il est important de bien comprendre qu'il est nécessaire de passer par ce mécanisme, car d'autres fonctions comme `millis()` ou même `delay()` ne doivent plus être utilisées dans ce contexte particulier de multitâche coopératif. Même simplement faire clignoter une led avec une certaine temporisation devra se faire via l'exécution de différentes tâches et fonctions de rappel (*callback*).



Une topologie en bus pour les réseaux informatiques était très courante dans les années 90 avec l'Ethernet 10base-2. Ce standard est maintenant obsolète et a laissé place au 10base-T, au 100base-T et au 1000base-T. La topologie en bus est toutefois toujours utilisée pour d'autres standards comme l'i2c.

Pour cet exemple, notre tâche utilisateur consistera à envoyer un simple message à tous les nœuds via une fonction `sendMessage()` écrite à cet effet. Celle-ci, en plus de cette action, changera la temporisation séparant les exécutions. On se retrouve alors dans une situation particulière où pour créer la tâche, nous avons besoin de référencer la fonction, mais où la fonction elle-même aura besoin d'utiliser une méthode sur l'objet représentant la tâche. Pour arriver à faire cela, il faut donc que la fonction « existe » avant même qu'elle ne soit écrite. En d'autres termes, nous devons déclarer la fonction séparément de sa définition :

```
// Prototype de fonction associée à la tâche
void sendMessage();

// Tâche
Task taskSendMessage(TASK_SECOND*1, TASK_FOREVER, &sendMessage);

// fonction associée à la tâche
void sendMessage() {
  String msg = "Coucou du node ";
  // ajout du numéro de node
  msg += mesh.getNodeId();
  // envoi à tous
  mesh.sendBroadcast(msg);
  // changement de l'intervalle de temps
  taskSendMessage.setInterval(random(TASK_SECOND*1, TASK_SECOND*5));
}
```

Comme vous pouvez le voir, nous avons là ce qui pourrait sembler être deux déclarations de `sendMessage()`, mais il s'agit en réalité de la déclaration de la fonction, et plus bas, de sa définition : « on dit qu'elle existe » et plus loin « on décrit en quoi elle consiste ». Ainsi, nous pouvons utiliser un pointeur vers celle-ci (`&sendMessage`) pour le passer en argument de `taskSendMessage()`, alors même que nous utilisons `taskSendMessage` dans la définition de `sendMessage()`. Le compilateur, grâce à ces informations, retrouvera ses petits sans le moindre problème. En réalité, sans vous en rendre compte, lorsque vous définissez une fonction, vous la déclarez implicitement. Notez également que si, dans la déclaration, vous spécifiez le type des arguments, on parle de prototype de fonction et non de simple déclaration.

Quoi qu'il en soit, nous avons ici la création d'une nouvelle tâche, exécutée toutes les secondes (`TASK_SECOND` fois 1), qui se répète à perpétuité (`TASK_FOREVER`) et appelant la fonction de callback `sendMessage()`. Celle-ci émet un message composé d'une chaîne de caractères complétée du numéro de nœud de l'émetteur. Ce message est transmis pour tous les autres nœuds (*broadcast*) et sera donc automatiquement répété et diffusé à travers tout le réseau mesh. Après l'envoi du message, nous modifions l'intervalle de temps entre les exécutions, de façon aléatoire. Ceci ne présente ici pas d'autres intérêts que le simple fait de montrer, dans l'exemple proposé, qu'il est possible pour une tâche de modifier ses propres paramètres depuis la fonction callback.

Nous pouvons à présent passer à la définition des fonctions callback utilisables avec *painlessMesh*. Fonction dont l'appel sera déclenché automatiquement par la bibliothèque lors de l'apparition de certains événements que nous spécifierons plus loin. Ceci n'est pas un impératif, mais permettra, dans un premier temps de voir, sur le moniteur série, l'activité du nœud et son interaction avec le réseau :

```
// Callbacks
void receivedCallback( uint32_t from, String &msg ) {
  Serial.printf("Message de %u = %s\n", from, msg.c_str());
}

void newConnectionCallback(uint32_t nodeId) {
  Serial.printf("Nouvelle connexion, node = %u\n", nodeId);
}

void changedConnectionCallback() {
  Serial.printf("Changement de connexions %s\n",
    mesh.subConnectionJson().c_str());
}

void nodeTimeAdjustedCallback(int32_t offset) {
  Serial.printf("Ajustement du temps %u. Offset = %d\n",
    mesh.getNodeTime(),offset);
}
```

Ce qui nous amène enfin à la fonction **setup()** :

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

  mesh.setDebugMsgTypes(ERROR|STARTUP);

  // Initialisation
  mesh.init(MESH_PREFIX, MESH_PASSWORD, &userScheduler, MESH_PORT);
  // en cas de réception
  mesh.onReceive(&receivedCallback);
  // en cas de nouvelle connexion
  mesh.onNewConnection(&newConnectionCallback);
  // en cas de changement de connexion
  mesh.onChangedConnections(&changedConnectionCallback);
  // en cas d'ajustement de la synchronisation du temps
  mesh.onNodeTimeAdjusted(&nodeTimeAdjustedCallback);

  // On ajoute la tâche à l'ordonnanceur
  userScheduler.addTask(taskSendMessage);
  // Activation de la tâche
  taskSendMessage.enable();
}
```

Après avoir configuré le moniteur série, nous spécifions quel type de message de mise au point (*debug*) nous souhaitons voir apparaître. Ici, nous choisissons uniquement les éventuelles erreurs et messages de démarrage, mais il est possible d'utiliser également **MESH_STATUS**, **CONNECTION**, **SYNC**, **COMMUNICATION**, **GENERAL**, **MSG_TYPES** et **REMOTE**, séparés par un **|** correspondant à une opération logique OU. Bien entendu, en activant tout cela, vous verrez rapidement l'écran du moniteur se remplir d'une myriade impressionnante de messages pas nécessairement importants.

Ce choix du type de message de débogage intervient avant l'initialisation du mesh, nous permettant ainsi de voir toute la phase de démarrage. Juste après, nous précisons les fonctions callback à utiliser via des méthodes spécifiques de l'objet **mesh**. En ajoutant les méthodes **onDroppedConnection** et **onNodeDelayReceived**, cette liste de callback définissable serait exhaustive.

Enfin, il ne nous reste plus qu'à ajouter notre tâche utilisateur (l'envoi de messages dans le mesh) à l'ordonnanceur et à l'activer. Tout le reste se passera dans la fonction `loop()` :

```
void loop() {
  // tâche utilisateur et tâche mesh
  userScheduler.execute();
  mesh.update();
}
```

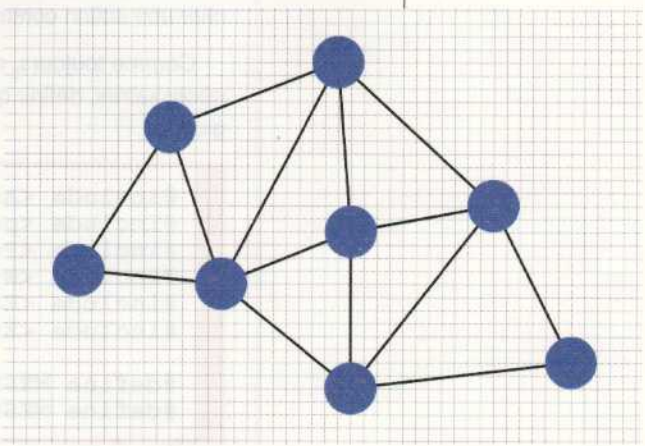
On retrouve les deux fonctions donnant réellement vie à la fois à notre ordonnanceur et au mécanisme de gestion du mesh. Comme avec beaucoup d'autres bibliothèques gérant des connexions réseau ou des événements ponctuels, l'appel constant de ces fonctions est impératif et il est hors de question de perdre du temps à exécuter une quelconque autre opération à ce niveau. Ne surchargez jamais une fonction `loop()` dans ce genre de situation, car le code risque de rater des événements.

3. INTERCONNECTER LES RÉSEAUX : EXEMPLE MQTT

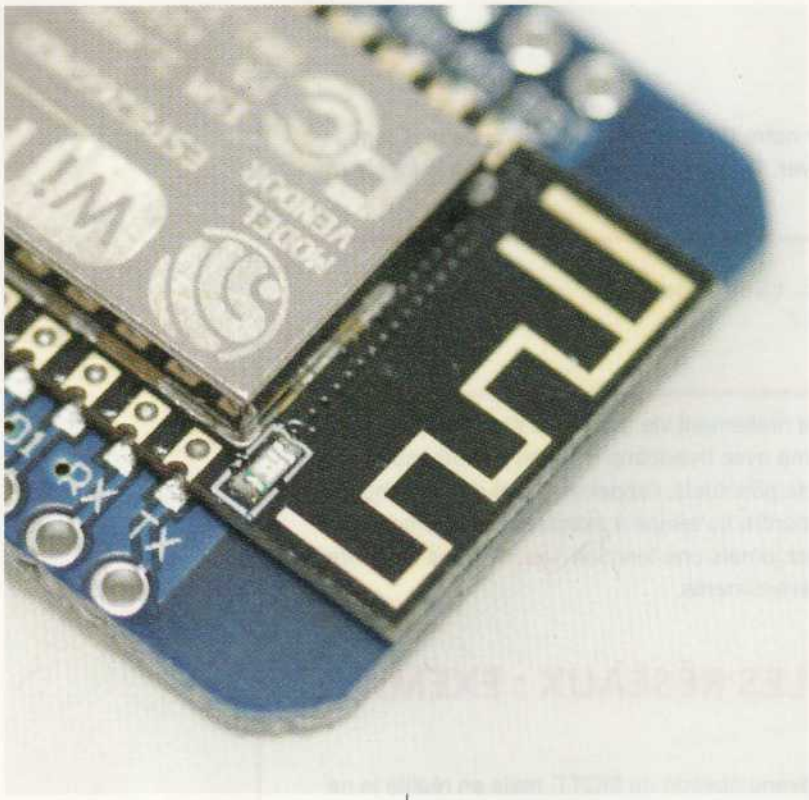
Vous allez finir par croire que je suis devenu obsédé de MQTT, mais en réalité je ne fais qu'optimiser un projet débuté dans *Hackable n°8*. Utilisant initialement des modules ESP-01 et des capteurs 1-wire DS18b20 pour relever des températures en HTTP, ceux-ci ont été mis à jour matériellement par remplacement de flash SPI (pour les mises à jour OTA), puis ont basculé de HTTP à MQTT et enfin, aujourd'hui, les voici en train de passer d'un réseau en étoile à un réseau mesh. Toute une aventure, dieu sait dans quelle direction partira le projet ensuite...

Quoi qu'il en soit, oui, effectivement, un réseau mesh tout autant que MQTT est une aubaine pour un réseau de capteurs. Judicieusement disposés, les ESP2866 peuvent étendre le réseau bien au-delà de ce qu'il serait possible de faire à l'aide d'un unique point d'accès wifi, mais en plus, ils sont autant de points de mesure utilisables. Il devient alors possible d'étendre très facilement le réseau, aussi bien horizontalement (pièces, annexes, garage, jardin) que verticalement (étages et même sous-sols).

Le réseau mesh ou réseau maillé est une architecture très particulière sans point névralgique central. Chaque élément du réseau, ou nœud est connecté en pair-à-pair avec ses voisins. Cette topologie est particulièrement intéressante dans le cas d'un réseau sans fil, comme le wifi.



Que les nœuds d'un réseau mesh échangent des données et tissent une toile sans point névralgique est une chose très intéressante, en particulier si chacun de ces nœuds est, en plus, équipé d'un ou plusieurs capteurs ou contrôlent un ou plusieurs relais. Mais le réseau formé dynamiquement doit cependant pouvoir être connecté au reste de votre installation domestique. Pour ce faire, un nœud spécifique doit faire la liaison entre les deux topologies réseau ou, en d'autres termes, servir de pont de l'un à l'autre (en anglais un *bridge*).



L'antenne PCB des modules ESP8266 n'est pas un modèle de performance et ceci impacte la distance maximum d'éloignement du point d'accès, même équipé d'une antenne avec un gain important. Dans le cas d'un réseau mesh, ceci n'a que peu d'importance puisqu'il suffit de placer autant d'ESP8266 que nécessaire pour arriver à la distance souhaitée.

Nous pouvons donc construire un pont entre le mesh et le réseau standard sous la forme d'un ESP8266 faisant fonctionner un croquis adapté. Le code que je vous propose d'étudier à présent est massivement inspiré de l'exemple fourni par la bibliothèque *painlessMesh*, adapté à notre configuration du précédent numéro (authentification auprès du broker MQTT et utilisation de SSL/TLS) et révisé sur certains points (conversion des chaînes de caractères). Ce croquis permet de capter les messages circulant sur le réseau mesh et les publie en MQTT sous le *topic* "painlessMesh/from/" complété de l'identifiant (*NodeId*) du nœud dans le mesh. Celui-ci est un numéro unique extrait à partir de l'identifiant de puce Espressif obtenu via la fonction `system_get_chip_id()`.

Ce croquis publie également des informations sur le *topic* "painlessMesh/from/gateway" qu'il s'agisse de son état ou des réponses aux requêtes qu'on pourra lui faire. À ce propos justement, inversement, le croquis s'abonne à des *topics* en utilisant un caractère joker : "painlessMesh/to/#". Comme nous allons le voir, ceci permet de lui adresser directement une requête sur le *topic* "painlessMesh/to/gateway" avec un simple message "getNodes" ayant pour conséquence de lui faire publier la liste des nœuds au format JSON. Le *topic* "painlessMesh/to/" pourra également être complété d'un identifiant de nœud pour adresser un message à ce nœud en particulier. Le pont fera alors suivre le message depuis MQTT vers le réseau mesh pour, par exemple, déclencher une action comme l'activation d'un relais.

Comme toujours, nous commençons par inclure les bibliothèques nécessaires, définir quelques macros et déclarer nos variables et objets globaux :

```
#include <Arduino.h>
#include <painlessMesh.h>
#include <PubSubClient.h>
#include <WiFiClient.h>
#include <ESP8266mDNS.h>
#include <EEPROM.h>

#define MESH_PREFIX     "CeQuiVousPlait"
#define MESH_PASSWORD   "QuelqueChoseDeSecret"
```

```
#define MESH_PORT       5555
#define MQTT_PORT       8883
#define FINGERPRINT "10:B3:81:C7:8A:21:81:66:EC:1C:8A:2C:D4:34:14:93:36:CE:39:19"

// nom de la machine ayant le broker (mDNS)
const char* mqtt_server = "raspberrypiled.local";

// connexion au mesh
painlessMesh  mesh;
// objet pour la connexion
WiFiClientSecure espClient;
// connexion MQTT
PubSubClient client(espClient);

// structure pour la configuration
struct EEconf {
  char ssid[32];
  char password[64];
  char myhostname[32];
} readconf;
```

Nous reprenons ici une bonne partie des lignes à la fois présentes dans notre croquis précédent concernant la découverte de MQTT et celles provenant de notre première expérimentation des réseaux mesh. Notez que les informations concernant le point d'accès wifi sont stockées en EEPROM émulée par l'ESP8266, d'où l'utilisation d'une structure spécifique permettant de facilement stocker et récupérer les données.

Nous enchaînons ensuite sur la création d'un certain nombre de fonctions, à commencer par celle permettant la connexion au broker MQTT :

```
// connexion au broker
void reconnect() {
  // Connecté au broker ?
  while (!client.connected()) {
    // non. On se connecte.

    // connexion via "espClient" et non "client"
    if (!espClient.connect(mqtt_server, MQTT_PORT)) {
      Serial.println("Unable to TLS connect");
      delay(5000);
      continue;
    }
    // vérification du fingerprint du broker
    if (!espClient.verify(FINGERPRINT, mqtt_server)) {
      Serial.println("Fingerprint check failed");
      espClient.stop();
      delay(5000);
      continue;
    }
    // déconnexion
    espClient.stop();
```

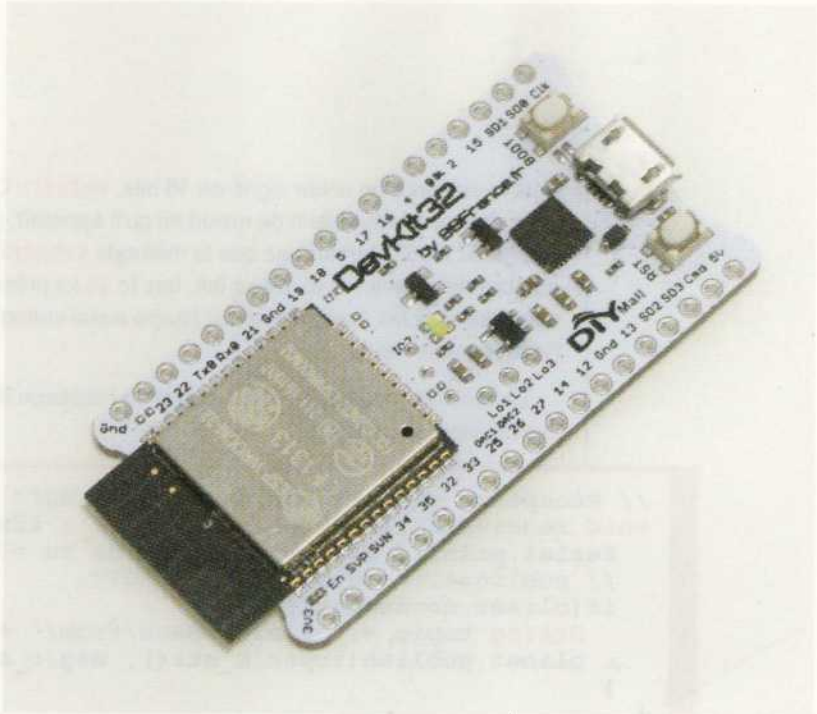
```
    // connexion MQTT via PubSubClient
    if (!client.connect(readconf.myhostname, "sondes", "mot2passe")) {
      Serial.print("Erreur connexion MQTT, rc=");
      Serial.println(client.state());
      delay(5000);
      continue;
    }
    Serial.println("Connexion serveur MQTT ok");
    // connecté.
    // on s'abonne au topic "analog"
    client.publish("painlessMesh/from/gateway","Ready!");
    client.subscribe("painlessMesh/to/#");
  }
}
```

Celle-ci est strictement identique à celle que nous avons utilisée dans le précédent numéro et intègre les mécanismes de sécurité jugés indispensables : authentification par mot de passe, chiffrement des communications SSL/TLS et vérification de l'empreinte du certificat du serveur MQTT Mosquito.

Il faut maintenant nous occuper de la communication depuis et vers le mesh en créant deux fonctions callback, l'une traitant les messages qui viennent en MQTT et l'autre de ceux provenant du réseau mesh. Commençons donc par la première et la plus étoffée :

```
// réception MQTT et retransmission mesh
void mqttCallback(char* topic, uint8_t* payload, unsigned int length) {
  String strPayload = "";
  // construction de la chaîne
  for (int i = 0; i < length; i++) {
    strPayload += (char)payload[i];
  }
  String targetStr = String(topic).substring(16);

  if(targetStr == "gateway") {
    if(strPayload == "getNodes") {
      client.publish("painlessMesh/from/gateway",
        mesh.subConnectionJson().c_str());
    }
  } else if(targetStr == "broadcast") {
    mesh.sendBroadcast(strPayload);
  } else {
    uint32_t target = strtoul(targetStr.c_str(), NULL, 10);
    if(mesh.isConnected(target)) {
      mesh.sendSingle(target, strPayload);
    } else {
      client.publish("painlessMesh/from/gateway", "Client absent!");
    }
  }
}
```

Un message arrivant en MQTT peut avoir trois significations particulières : une requête concernant le pont lui-même, un message à relayer à tous les nœuds du mesh et enfin, un message à envoyer à un nœud spécifique identifié par son numéro. Ici, la chaîne de caractères nous informant



du *topic* doit être analysée et non simplement le contenu du message. C'est pour cette raison qu'il est nécessaire de jongler avec ces chaînes et les convertir en `String` pour en faciliter la gestion. La création de `strPayload` repose sur le morceau de code que nous avons vu dans le numéro précédent, car, contrairement à `targetStr`, une conversion par la fonction `String()` n'est pas possible (nous n'avons pas le `\0` en fin de tableau). Une fois ces conversions faites, le reste de la fonction se contente d'une série de conditions `if` pour réagir en conséquence.

Nous parlons ici principalement de l'ESP8266 étant donné le coût minimum des modules et cartes qui en sont équipés. Sachez cependant que tout ceci est également applicable avec l'ESP32, le grand frère plus musclé de l'ESP8266.

Notez plusieurs points intéressants. La bibliothèque *painlessMesh* met à notre disposition des fonctions avancées nous facilitant grandement les opérations. Envoyer la liste des nœuds en réponse au message "getNodes" sur le *topic* "painlessMesh/to/gateway" se résume, par exemple, au simple appel de la méthode `subConnectionJson()`. Ce qui en résulte sera un message publié en MQTT sur le *topic* "painlessMesh/from/gateway", sous la forme, par exemple, de la chaîne "[{"nodeId":2786571136,"subs":[{"nodeId":3895826423,"subs":[]}]}]", représentant la structure du réseau mesh. En remettant en forme l'information au format JSON de façon plus lisible, nous obtenons :

```
[
    {
        "nodeId":2786571136,
        "subs":[
            {
                "nodeId":3895826423,
                "subs":[

                ]
            }
        ]
    }
]
```

On voit ici que le nœud 2786571136 est connecté au *bridge* MQTT et que ce nœud sert de point de connexion au nœud 3895826423. Nous avons donc quelque chose que nous pouvons représenter mentalement en `3895826423 -> 2786571136 -> pont -> Broker`.

Pour envoyer un message à un nœud, nous devons désassembler le *topic* pour retrouver l'identifiant en utilisant la méthode `substring()` pour supprimer "painlessMesh/to/" de la chaîne. L'identifiant, présent sous la forme d'une chaîne est ensuite converti en entier long (casté en `uint32_t`) avec la fonction C standard `strtoul()`. Il n'existe, en effet, pas de méthode sur un objet `String` pour obtenir de type de variable, comme

c'est le cas pour un entier signé de 16 bits, `toInt()`. Or, justement, 16 bits sont bien insuffisants pour stocker un identifiant de nœud tel qu'il apparaît, par exemple, dans le *topic* "painlessMesh/to/2786571136". Remarquez que la méthode `substring()` prend en argument une position en nombre de caractères et, de ce fait, que le `16` ici présent correspond à un *topic* spécifique. Si vous changez les *topics* utilisés, il faudra aussi changer cette valeur pour extraire correctement l'identifiant de nœud.

La fonction inverse, celle qui prend un message du réseau mesh et le publie en MQTT est beaucoup plus concise :

```
// Réception mesh et retransmission MQTT
void receivedCallback( const uint32_t &from, const String &msg ) {
  Serial.printf("bridge: Message de %u =%s\n", from, msg.c_str());
  // publication du message en MQTT
  if(client.connected()) {
    String topic = "painlessMesh/from/" + String(from);
    client.publish(topic.c_str(), msg.c_str());
  }
}
```

Ici, aucune difficulté, on s'assure simplement de la connexion au broker avant de publier le message qui est, de toute façon, déjà de type `String`. Voilà qui nous permet d'enchaîner directement sur les fonctions classiques Arduino, à commencer par `setup()` :

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

  EEPROM.begin(sizeof(readconf));
  EEPROM.get(0, readconf);

  mesh.setDebugMsgTypes(ERROR | STARTUP | CONNECTION);

  // Initialisation du mesh en utilisant le canal wifi 3
  // Ce canal doit être identique à celui utilisé par le point d'accès
  mesh.init(MESH_PREFIX, MESH_PASSWORD, MESH_PORT, WIFI_AP_STA, 3);
  mesh.onReceive(&receivedCallback);

  // Connexion wifi en mode client (station) manuelle
  mesh.stationManual(readconf.ssid, readconf.password);
  // Configuration du nom d'hôte
  mesh.setHostname(readconf.myhostname);

  // configuration broker
  client.setServer(mqtt_server, MQTT_PORT);
  // configuration callback
  client.setCallback(mqttCallback);
}
```

On retrouve dans le corps de cette fonction de nombreux éléments que nous connaissons déjà, de par notre utilisation précédente de MQTT et notre premier exemple de réseau mesh. Une petite nuance existe en revanche concernant le mode de fonctionnement de ce nœud/pont. En effet, celui-ci ne cherchera pas dynamiquement à se connecter à un autre nœud pour former le réseau mesh, mais utilisera une connexion manuelle à notre point d'accès

avec la méthode `stationManual()`. On précise ici les informations de connexion extraites de l'EEPROM et c'est la bibliothèque *painlessMesh* qui se charge de tout. Il est important, toutefois, de bien spécifier un canal Wifi dans l'initialisation du mesh. Celui-ci doit être le même pour le mesh et pour le point d'accès. Ici, mon point d'accès géré par HostAPd sur Raspberry Pi utilise le canal 3, alors que la valeur par défaut pour `init()` est 6 dans la bibliothèque.

Enfin, il ne reste plus qu'à nous occuper de `loop()` :

```
void loop() {
  // mesh
  mesh.update();

  // Sommes-nous connecté ?
  // wifi
  if (WiFi.status() == WL_CONNECTED) {
    // broker MQTT
    if (!client.connected()) {
      reconnect();
    }
  }
  // MQTT
  client.loop();
}
```

On se contente ici de passer le relais à la gestion MQTT et du mesh à chaque tour de boucle, tout en vérifiant que la connexion au broker est active. Le démarrage du croquis, incluant une connexion wifi, au broker et au mesh prend un certain temps, et cette procédure doit se faire dans un certain ordre. Sans Wifi, pas de connexion au broker, sans broker, pas d'appel à la fonction callback, etc.

Dès la compilation et le chargement du croquis dans l'ESP8266, et avec deux autres nœuds chargés avec le croquis du début d'article, nous pouvons voir le déroulé des opérations sur le moniteur série Arduino :

- Tout d'abord, l'initialisation :

```
setDebugTypes: ERROR | STARTUP | CONNECTION |
STARTUP: init(): 1
STARTUP: AP tcp server established on port 5555
CONNECTION: Event: 2786571136 Connected to AP Mode Station
CONNECTION: New AP connection incoming
CONNECTION: meshConnectedCb(): we are AP
```

- Très rapidement, un premier message d'un nœud du mesh arrive alors que toutes les connexions ne sont pas établies, la procédure se poursuit cependant normalement :

```
bridge: Message de 2786571136 =Coucou du node 2786571136
CONNECTION: Event: 2786571136 Disconnected from AP Mode Station
CONNECTION: stationScan(): SSIDmonAP
CONNECTION: scanComplete():-- > scan finished @ 13190470 < --
CONNECTION: scanComplete():-- > Cleared old aps.
CONNECTION: scanComplete(): num = 5
CONNECTION: found : SSIDmonAP, -50dBm
```

```
CONNECTION: Found 1 nodes
CONNECTION: findConnection(1806250640): did not find connection
CONNECTION: connectToAP(): Best AP is 1806250640<---
CONNECTION: connectToAP(): Trying to connect, scan rate set to 4*normal
CONNECTION: Event: Station Mode Auth Mode Change
CONNECTION: Event: Station Mode Connected
CONNECTION: Event: Station Mode Got IP (IP: 192.168.10.119
   Mask: 255.255.255.0  Gateway: 192.168.10.1)
CONNECTION: onDisconnect():
CONNECTION: onDisconnect(): dropping 0 now= 14606541
CONNECTION: MeshConnection::close().
CONNECTION: eraseClosedConnections():
CONNECTION: ~MeshConnection():
CONNECTION: MeshConnection::close() Done.
Connexion serveur MQTT ok
CONNECTION: closingTask():
```

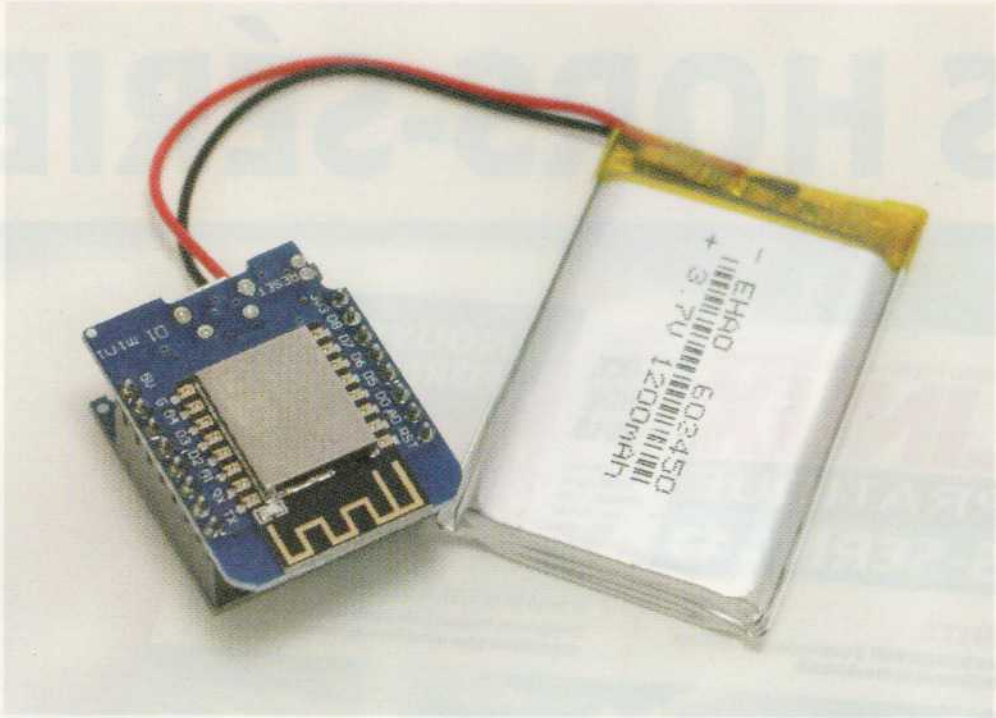
- Nous sommes à présent connectés au point d'accès, au mesh et au broker MQTT. Le fonctionnement normal se met en place :

```
CONNECTION: closingTask(): dropping 0 now= 14910827
CONNECTION: Event: 2786571136 Connected to AP Mode Station
CONNECTION: New AP connection incoming
CONNECTION: meshConnectedCb(): we are AP
CONNECTION: findConnection(2786571136): did not find connection
CONNECTION: newConnectionTask():
CONNECTION: newConnectionTask(): adding 2786571136 now= 17539549
bridge: Message de 2786571136 =Coucou du node 2786571136
bridge: Message de 3895826423 =Coucou du node 3895826423
bridge: Message de 3895826423 =Coucou du node 3895826423
bridge: Message de 3895826423 =Coucou du node 3895826423
bridge: Message de 3895826423 =Coucou du node 3895826423
bridge: Message de 2786571136 =Coucou du node 2786571136
bridge: Message de 3895826423 =Coucou du node 3895826423
bridge: Message de 3895826423 =Coucou du node 3895826423
```

En parallèle, sur la Pi du broker, nous pouvons nous abonner à l'ensemble des *topics* pour voir ce qui se passe avec `mosquitto_sub -v -h raspberrypiled.local -p 8883 --tls-version tlsv1 --cafile /etc/mosquitto/ca.crt -u "sondes" -P "mot2passe" -t "#"`, et nous voyons alors :

```
painlessMesh/from/3895826423 Coucou du node 3895826423
painlessMesh/from/2786571136 Coucou du node 2786571136
painlessMesh/from/2786571136 Coucou du node 2786571136
painlessMesh/from/2786571136 Coucou du node 2786571136
painlessMesh/from/3895826423 Coucou du node 3895826423
painlessMesh/from/3895826423 Coucou du node 3895826423
painlessMesh/from/2786571136 Coucou du node 2786571136
```

Dans une autre console sur la Pi, nous pouvons alors publier un message pour connaître l'architecture du réseau mesh avec `mosquitto_pub -h raspberrypiled.local -p 8883 --tls-version tlsv1 --cafile /etc/mosquitto/ca.crt -u "sondes" -P "mot2passe" -t "painlessMesh/to/gateway" -m "getNodes"`. Ceci devrait faire apparaître, sur la première console, la réponse :



En ajoutant un shield à une carte Wemos D2 mini et un accu LiPo/Li-Ion, il devient possible de dynamiquement et temporairement créer un réseau mesh. Il suffit alors de disséminer vos capteurs sur la zone à surveiller, même en extérieur, et de collecter vos données.

```
painlessMesh/to/gateway getNodes
painlessMesh/from/gateway [{"nodeId":2786571136,"subs":[{"nodeId":3895826423,"subs":[]}]}]
```

Enfin, pour envoyer un message à un nœud, nous pouvons le faire avec **`mosquitto_pub`** depuis la Pi via, par exemple, **`mosquitto_pub -h raspberrypiled.local -p 8883 --tls-version tlsv1 --cafile /etc/mosquitto/ca.crt -u "sondes" -P "mot2passe" -t "painlessMesh/to/3895826423" -m "Coucou aussi !"`**. Bien entendu, le croquis basique utilisé sur les nœuds ne fera rien de particulier avec ce message, mais vous pourrez voir, en connectant le moniteur série au nœud 3895826423, qu'il est effectivement réceptionné via le réseau mesh.

CONCLUSION

Comme nous venons de le voir, et même si un grand nombre de bibliothèques entrent en jeu, la création d'un réseau mesh composé d'ESP8266 ne présente pas de grandes difficultés. Le gain en revanche est énorme puisque le réseau pourra s'étendre à mesure que vous ajouterez des nœuds dans le réseau. En ajoutant le pont vers le réseau domestique et grâce à la souplesse offerte par MQTT, il devient possible d'imaginer toutes sortes de réalisations allant du contrôle domotique de l'éclairage aux réseaux de capteurs de toutes sortes.

Même en dehors de l'application pratique d'une telle architecture, le simple fait que *painlessMesh*, en plus de porter magnifiquement son nom, repose sur le travail d'autres développeurs est un élément très pédagogique. Je pense naturellement à la bibliothèque *TaskScheduler* qui apporte une vision radicalement différente de l'utilisation de cartes Arduino ou ESP8266 et vous « forcera » à réfléchir à une autre approche pour votre code. Reposer sur un code sain, étendre facilement son réseau et en même temps apprendre énormément de choses... Que demander de plus ? **DB**

CONTRÔLEZ VOS NEOPIXELS EN 3,3 VOLTS

Denis Bodor

Voici un problème des plus classiques : vous travaillez avec une carte utilisant des niveaux de tensions entre 0 et 3,3 volts et vous souhaitez piloter ces fameuses leds intelligentes qui s'alimentent en 5 volts et qui surtout attendent des signaux en 0/5 volts. Avec la popularisation sans cesse grandissante de cartes ou modules comme les ESP8266, Micro:bit, STM32 Nucléo, ESP32 ou tout simplement une Raspberry Pi et consorts, ce problème ne risque pas de disparaître, bien au contraire. Découvrons ensemble quelques techniques permettant de le régler...

Il faut se rendre à l'évidence, plus le temps passe, plus le monde des niveaux logiques 0/5V se réduit. Il ne reste aujourd'hui presque plus que les cartes Arduino classiques (UNO, etc.) à travailler avec ces niveaux de tensions. Toutes les « nouvelles » plateformes, qu'il s'agisse des ordinateurs mono-cartes comme la Raspberry Pi et ses « clones », des nouvelles plateformes comme l'ESP8266/ESP32, ou encore les cartes à base de microcontrôleur ARM Cortex-M, toutes utilisent 3,3 volts pour un niveau logique haut (1) et 0 volt pour le niveau logique bas. Pire encore, la plupart du temps, ces plateformes et cartes ne sont pas tolérantes au 5 volts et leur appliquer une telle tension pourra potentiellement les endommager, voire les rendre inutilisables à coup sûr.

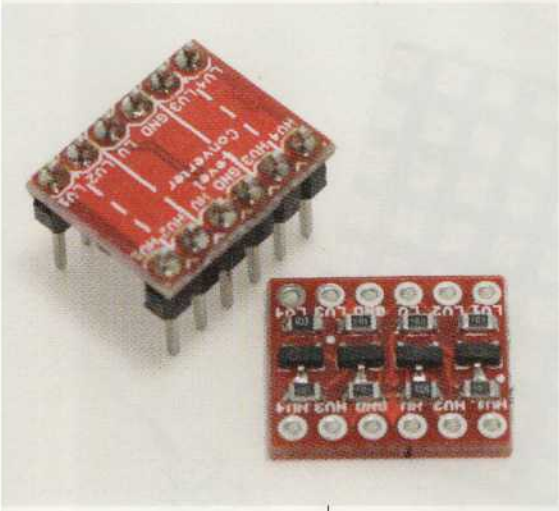
Pourtant, bon nombre de circuits, de composants, de modules et de périphériques attendent un beau et ferme 5 volts en guise de niveau haut. C'est vrai pour les afficheurs LCD alphanumériques à base de HD44780, les RTC comme le DS1302, mais aussi et surtout les leds RVB « intelligentes » (ou « adressables ») comme la WS2812b et ses consœurs, généralement désignées par le terme, bien malheureux, de « NeoPixel ». Il existe une jolie gamme de ce type de leds, des composants de surface seuls aux rubans, en passant par les panneaux, les anneaux de différentes tailles ou encore les déclinaisons au format led standard de 3, 5 ou 8 mm de diamètre. Même les plus récents de ces composants, ajoutant une quatrième led dans le boitier pour obtenir RGBY (rouge, vert leur, jaune) ou RGBW (rouge, vert, bleu, blanc), voire RGBWW (2 leds blanches en plus), fonctionnent toujours en 5V tant pour l'alimentation que pour les niveaux logiques.

Les leds tricolores adressables, généralement appelées NeoPixels ou WS2812b, existent en de nombreuses déclinaisons et sont vendues déjà montées sous différentes formes. Elles ont toutes cependant en commun la nécessité d'utiliser une alimentation et des signaux en 5 volts.

Vous l'aurez compris, le problème est là pour un bon bout de temps et mieux vaut alors savoir comment votre plateforme, uniquement capable de fournir 3,3 volts, pourra se faire « entendre » et comprendre par ces leds adressables si fascinantes, amusantes et utiles. Notre cas pratique consistera ici à utiliser un ESP8266 (clone de Wemos D1 mini) pour contrôler trois leds APA106 translucides de 8 mm.

1. MODULE DÉDIÉ

L'approche la plus simple pour régler le problème consiste à utiliser un module tout fait. Une petite recherche sur eBay de termes comme « *4 Channels Logic Level Converter BiDirectional* » vous retournera bon nombre



L'approche la plus simple pour piloter un circuit 5 volts à partir d'une carte n'utilisant que 3,3 volts en sortie est de passer par un adaptateur de tension comme ceux-ci. Mais encore faut-il en avoir sous la main au bon moment...

de résultats. Si vous activez la recherche pour le monde entier, vous constaterez rapidement que pour moins de 3€ vous pouvez obtenir 3 ou 4 pièces de ces « convertisseurs de niveaux ».

Leur construction est relativement simple et leur niveau de qualité de fabrication très variable, mais ils présentent l'avantage d'être maintenant très économiques et faciles à trouver. Il n'y a pas si longtemps, seuls des sites comme Adafruit et Pololu proposaient ce type de circuits, mais les vendeurs chinois ont rapidement réagi en clonant le design qui, à la base, était déjà connu de longue date et même publié dans une note d'application de NXP à propos du bus i2c (AN97055).

Ce montage, mettant en oeuvre uniquement des MOSFET et des résistances de rappel de 10K ohms, présente l'avantage d'être bidirectionnel. Ainsi, lorsque votre carte contrôleur en 3,3 volts présente sur l'une de ses sorties un état haut sur la grille du MOSFET, celui-ci devient conducteur et ainsi utilise l'alimentation 5 volts pour faire de même sur sa broche drain. Inversement, lorsque le circuit connecté en 5 volts change l'état de la ligne, c'est la diode intégrée dans le MOSFET qui se charge de baisser la tension. Nous obtenons donc un adaptateur de niveaux de tensions parfaitement adapté pour des bus comme l'i2c utilisant une seule ligne de données (SDA) bidirectionnelle. Ce type de circuits pourra donc parfaitement être utilisé pour une horloge temps réel en i2c comme la DS1301 de Dallas, afin, par exemple de la connecter à un ESP8266 ou une carte Raspberry Pi.

Dans notre cas, l'aspect bidirectionnel n'est pas utile étant donné que les leds WS2812b n'utilisent leur broche DOUT (*Data Out*) que pour chaîner une led à une autre, et qu'il n'y a pas de communication de la led vers le contrôleur. De plus, vous pouvez être certain que lorsque vous en aurez besoin dans l'urgence, vous ne retrouverez plus vos adaptateurs, quel que soit le nombre que vous aurez acheté (il est même tout à fait possible qu'ils soient tous au même endroit, dont vous vous rappellerez précisément au moment où vous n'en n'avez plus besoin, ou que vous cherchez autre chose en vain).

2. MOSFET

Si vous n'avez pas, ou plus, ces modules à portée de main, vous pouvez vous tourner vers la construction d'un circuit similaire à l'aide d'un seul MOSFET et de deux résistances. L'idée est séduisante, mais moins facile à mettre en oeuvre qu'il n'y paraît. En effet, le principe de fonctionnement du MOSFET (*Metal Oxide Semiconductor Field Effect Transistor* ou « transistor à effet de champ à grille isolée » en français), qui est un type de transistor, repose sur un certain nombre de caractéristiques qui doivent correspondre à notre application.

L'utilisation générale des MOSFET consiste à les utiliser comme des interrupteurs (en commutation) contrôlés par un montage intelligent. Le plus souvent, il s'agit de piloter le fonctionnement de systèmes utilisant une tension supérieure à celle du signal de contrôle, et ce avec un courant maximum. L'application typique de ce composant pourra être, par exemple, le contrôle par une carte Arduino d'un moteur ou encore d'un panneau de leds standards alimenté en 12 ou 24 volts.

Contrairement au transistor bipolaire, tel un 2N2222, où c'est le courant qui contrôle son comportement, avec un MOSFET c'est la tension entre la grille et la source (Vgs) qui détermine le courant qui traverse le composant entre le drain et la source. Il existe donc

une tension de seuil minimale qui déclenche le passage du courant pour chaque modèle de MOSFET. De plus, cette tension n'est pas celle permettant à un maximum de courant de passer, mais une simple limite basse. La plupart des MOSFET ont effectivement une tension de seuil compatible avec des signaux 5 volts, mais seuls quelques modèles sont réellement utilisables directement avec une carte Arduino. Un exemple typique est l'IRF520, très courant, qui ne vous sera pourtant d'aucune utilité pour piloter un moteur avec une carte Arduino UNO. Vous devrez préférer sa déclinaison dite « *logic level gate drive* », l'IRL520, avec un seuil (*Vgs Threshold*) entre 1 et 2 volts. En effet, celui-ci est conçu pour être pleinement conducteur (résistance entre le drain et la source ou Rds(on) à 0,27 ohm) avec une tension Vgs (entre la grille et la source) entre 4 et 5 volts.

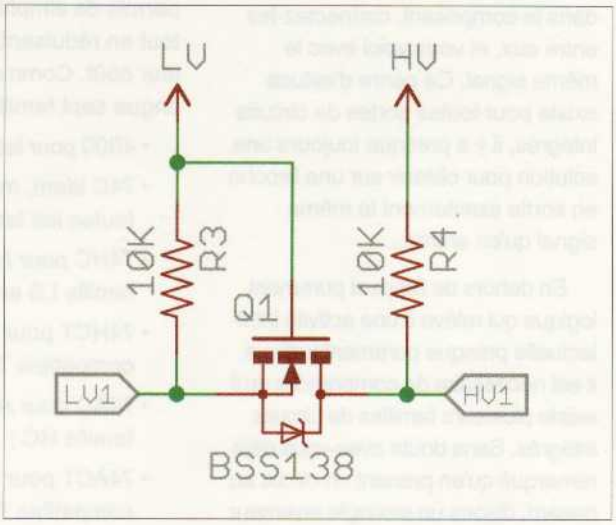
« Et 3,3 volts alors ? » me direz-vous ? Eh bien, si vous tentez d'utiliser un IRL520 avec une configuration similaire à celle utilisée dans les circuits adaptateurs vendus sur eBay, vous allez être déçu. En effet, la résistance entre le drain et la source est inversement proportionnelle à la tension entre la grille et la source. Comme vous serez, certes au-delà du seuil, mais en dessous de la tension nécessaire pour atteindre Rds(on), une résistance relativement conséquente existera. Celle-ci sera alors assez significative pour que vous vous retrouviez avec un diviseur de tension, constitué de la résistance interne du MOSFET et de celle que vous aurez utilisée comme résistance de rappel au +5 volts.

Vous vous retrouverez donc avec une tension en sortie quasiment identique à celle en entrée. Pire encore, à cause de la capacité interne du MOSFET et celle du circuit, non seulement vous n'aurez pas la tension souhaitée, mais en plus vous allez dégrader le signal, l'ensemble des résistances et des capacités en présence formant un circuit RC introduisant des délais et des transitions parasites.

Les modules proposés sur eBay, et ailleurs, utilisent des MOSFET spécifiques, comme les 2N7002P, qui sont aux 3,3 volts ce que les IRL520 sont aux 5 volts. Bien sûr, pour contourner le problème, vous pourrez toujours utiliser un transistor bipolaire pour piloter le MOSFET ou l'utiliser directement pour convertir les tensions. Mais encore faut-il en avoir un sous la main et, si vous êtes pressé, prendre le temps de faire tous les calculs nécessaires. Il y a plus simple...

3. CIRCUITS LOGIQUES

Bien plus simples à utiliser que des MOSFET ou des transistors, mais sensiblement plus coûteux, une autre approche implique de tout simplement reposer sur une technologie prévue pour supporter des écarts de tensions et les corriger. Si vous disposez de circuits logiques, récupérés ou achetés il y a bien longtemps en quantité plus que suffisante pour le projet du moment, il y a de fortes chances que vous puissiez vous en servir pour adapter les tensions entre votre carte en 3,3 volts et vos leds adressables.



Les modules permettant d'interfacer des circuits 3,3v et 5v sont souvent bidirectionnels et reposent sur la mise en oeuvre d'un MOSFET-N et de deux résistances. Mais ne vous y trompez pas, ce MOSFET n'est pas choisi par hasard et doit répondre à des caractéristiques bien précises pour ce type d'usage.

Une autre solution pour adapter facilement les niveaux de tensions consiste à utiliser des circuits logiques. Pour peu qu'il s'agisse de technologies TTL ou CMOS compatibles TTL, tout ce que vous aurez à faire c'est de trouver le moyen d'obtenir en sortie le même signal qu'en entrée.

Lorsque je parle de circuits logiques, ceci concerne toutes sortes de circuits intégrés : inverseurs, portes logiques, buffer, registres à décalage, etc. Peu importe leur usage d'origine, l'objectif ici est de retomber sur vos pieds ou, en d'autres termes, d'obtenir en sortie le même signal qu'en entrée, mais à la bonne tension. Prenons un sextuple inverseur comme un 74LS04. Celui-ci prend en entrée un signal et l'inverse. On lui présente un niveau haut sur une entrée et il présentera un niveau bas sur la sortie correspondante, et inversement. Utilisez deux des six inverseurs intégrés dans le composant, connectez-les entre eux, et vous voici avec le même signal. Ce genre d'astuce existe pour toutes sortes de circuits intégrés, il y a presque toujours une solution pour obtenir sur une broche en sortie exactement le même signal qu'en entrée...

En dehors de l'aspect purement logique qui relève d'une activité intellectuelle presque purement ludique, il est nécessaire de comprendre qu'il existe plusieurs familles de circuits intégrés. Sans doute avez-vous déjà remarqué qu'en prenant un circuit au hasard, disons un sextuple inverseur 7404, celui-ci peut se décliner en de nombreuses appellations : 74HC04, 74HCT04, 74LS04, etc. Ces lettres ne sont, bien entendu, pas là par hasard et renseignent sur la technologie utilisée pour la fabrication du composant. Ces techniques de fabrication se divisent en deux grandes technologies : TTL et CMOS.

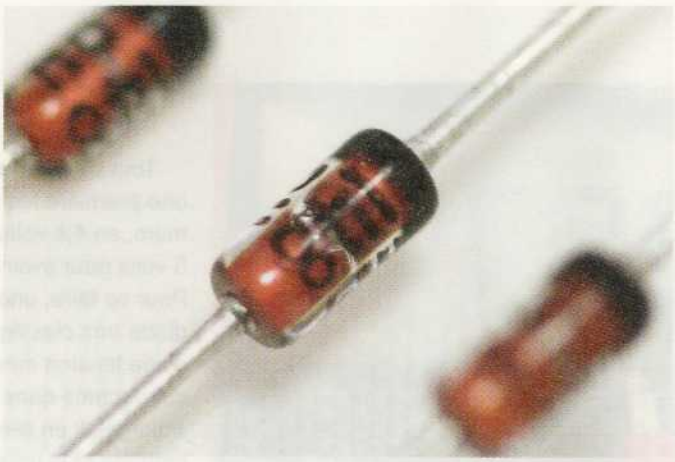
TTL, signifiant *Transistor Transistor Logic*, est la technologie la plus ancienne qui repose sur l'utilisation de transistors bipolaires NPN ou des transistors Schottky. Dans la technologie TTL, vous trouverez les familles de composants suivantes :

- 74xx pour les TTL standards,
- 74Lxx pour *Low power*, faible consommation,
- 74Sxx pour *Schottky*, utilisant des transistors Schottky (plus rapides),
- 74LSxx pour *Low power Schottky*, Schottky et faible consommation,
- 74ASxx pour *Advanced Schottky*, Schottky avec d'autres caractéristiques,
- 74ALSxx pour *Advanced Low power Schottky*, idem, mais faible consommation,
- 74Fxx pour *Fast*, standard, mais rapide, non Schottky.

Les circuits intégrés TTL sont, de base, prévus pour une alimentation 5 volts et des signaux 5 volts.

Quelques années après la naissance de la technologie TTL, la technologie CMOS pour *Complementary Metal Oxide Semiconductor*, a vu le jour. Celle-ci a permis de simplifier la fabrication des circuits intégrés tout en réduisant les dimensions des puces et donc leur coût. Comme pour la technologie TTL, on distingue sept familles :

- 4000 pour les CMOS standards (ou « classiques ») ;
- 74C idem, mais avec un brochage commun à toutes les technologies 74xx ;
- 74HC pour *High-speed CMOS*, aussi rapide que la famille LS en TTL ;
- 74HCT pour *High-speed CMOS TTL*, idem, mais compatible TTL ;
- 74AC pour *Advanced CMOS*, plus rapide que la famille HC ;
- 74ACT pour *Advanced CMOS TTL*, idem, mais compatible TTL ;



Les diodes 1N4148 permettent toutes sortes de montages très intéressants, parmi lesquels abaisser légèrement une tension. En bonus, lorsqu'on les observe vraiment tout près, elles sont tellement plus sexy que les 1n4001 tristement opaques et noires.

- 74LVT pour *low Voltage Threshold*, prévu pour fonctionner en 3,3 volts.

La notion de compatibilité entre CMOS et TTL est là où les choses deviennent très intéressantes pour nous. En effet, peut-être avez-vous déjà remarqué que les désignations des tensions d'alimentation positives et négatives ne sont pas toujours les mêmes. D'un côté, pour la technologie TTL, on parle respectivement de Vcc et de Vee, mais pour le CMOS, c'est Vdd et Vss. Ceci provient, entre autres, du fait que l'alimentation des circuits intégrés est très différente de l'une à l'autre. Les TTL fonctionnent en 5 volts, mais les CMOS peuvent aller de 2 volts à 10 volts, selon les modèles. Il en résulte que dans un cas, la tension correspondant à un état haut ou bas est fixe, mais dans l'autre, elle est dépendante de la tension d'alimentation.

À alimentation égale, en 5 volts, les composants TTL et CMOS (non compatibles TTL) se comportent de façons différentes :

- TTL : niveau logique 1 à 2v minimum et niveau logique 0 à 0,8v maximum ;
- CMOS : niveau logique 1 à 3,5v minimum et niveau logique 0 à 1,5v maximum.

Les circuits TTL et CMOS compatibles TTL peuvent donc accepter des signaux en 0/3,3v en entrée, mais produiront des signaux en sortie de 0/5v. Ce qui signifie donc qu'en utilisant un circuit logique comme un 74LS04, un 74HCT04 ou un 74ACT04, et en connectant deux inverseurs du composant ensemble, nous pouvons nous en servir comme adaptateur 3,3v/5v et donc piloter nos leds WS2812b (ou APA106) avec un ESP8266. Tout ce que vous avez à faire est d'alimenter le circuit logique et les WS2812b en 5 volts, de connecter la sortie de l'ESP8266 à l'entrée d'un inverseur, la sortie de ce dernier à un autre inverseur et enfin, la sortie de celui-ci à la broche DIN de la première led adressable.

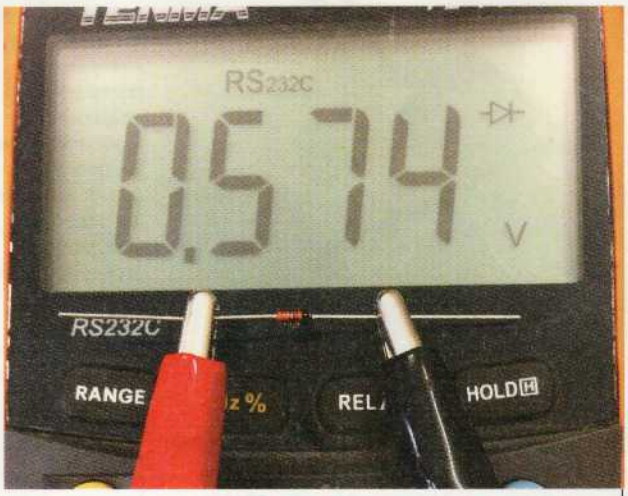
4. LE SACRIFICE RITUEL

Le fait qu'un circuit intégré précise une tension minimum pour un état logique haut qui soit différente de sa tension d'alimentation est fort pratique et surtout quelque chose de parfaitement standard. Savez-vous ce qui se trouve au cœur de nos fameuses leds adressables type WD2812b ou APA16 ? Un circuit intégré !

Celles-ci ne peuvent bien sûr pas être contrôlées avec des signaux 0/3,3 volts mais, pour autant, un état logique haut ne doit pas nécessairement être strictement égal à leur tension d'alimentation. En effet, la documentation précise qu'un niveau logique haut doit être au minimum de 0,7 fois la tension d'alimentation qui, selon les versions, se situe entre 3,5 et 5,3 volts. En alimentant vos leds en 5 volts, vous devez donc fournir un signal de 3,5 volts minimum pour vous faire obéir.

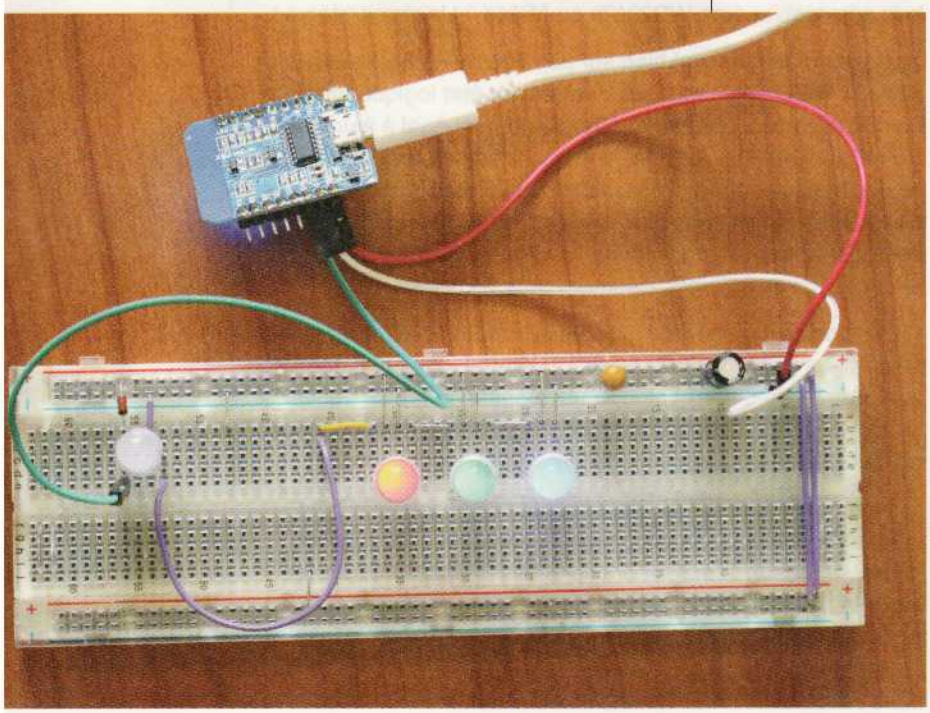
Par contre, en abaissant la tension d'alimentation à, disons, environ 4,4 volts, nous pouvons fournir des signaux à une tension d'au minimum 3,1 volts. Seuls problèmes, il faut trouver un moyen de générer cette tension et alimenter tous vos leds de cette façon. Vraiment ?

Non, car la tension sur les broches DOUT des leds sera toujours à 4,4 volts. Seule la première d'entre elles recevra effectivement un signal en 3,3 volts de l'ESP8266. Et donc, seule la première led est concernée. Ceci signifie que si la seconde led de la série est alimentée avec la tension typique de 5 volts, le signal qu'elle va recevoir sera en 4,4 volts et donc au-dessus des 5*0,7=3,5 volts minimum.



Mesurer la tension de seuil d'une diode est un jeu d'enfant. Tout ce que vous avez à faire est d'utiliser le bon mode avec votre multimètre et tester la diode dans le bon sens. Cette 1N4148 provoquera la chute de tension de 0,574 volts dont nous avons besoin.

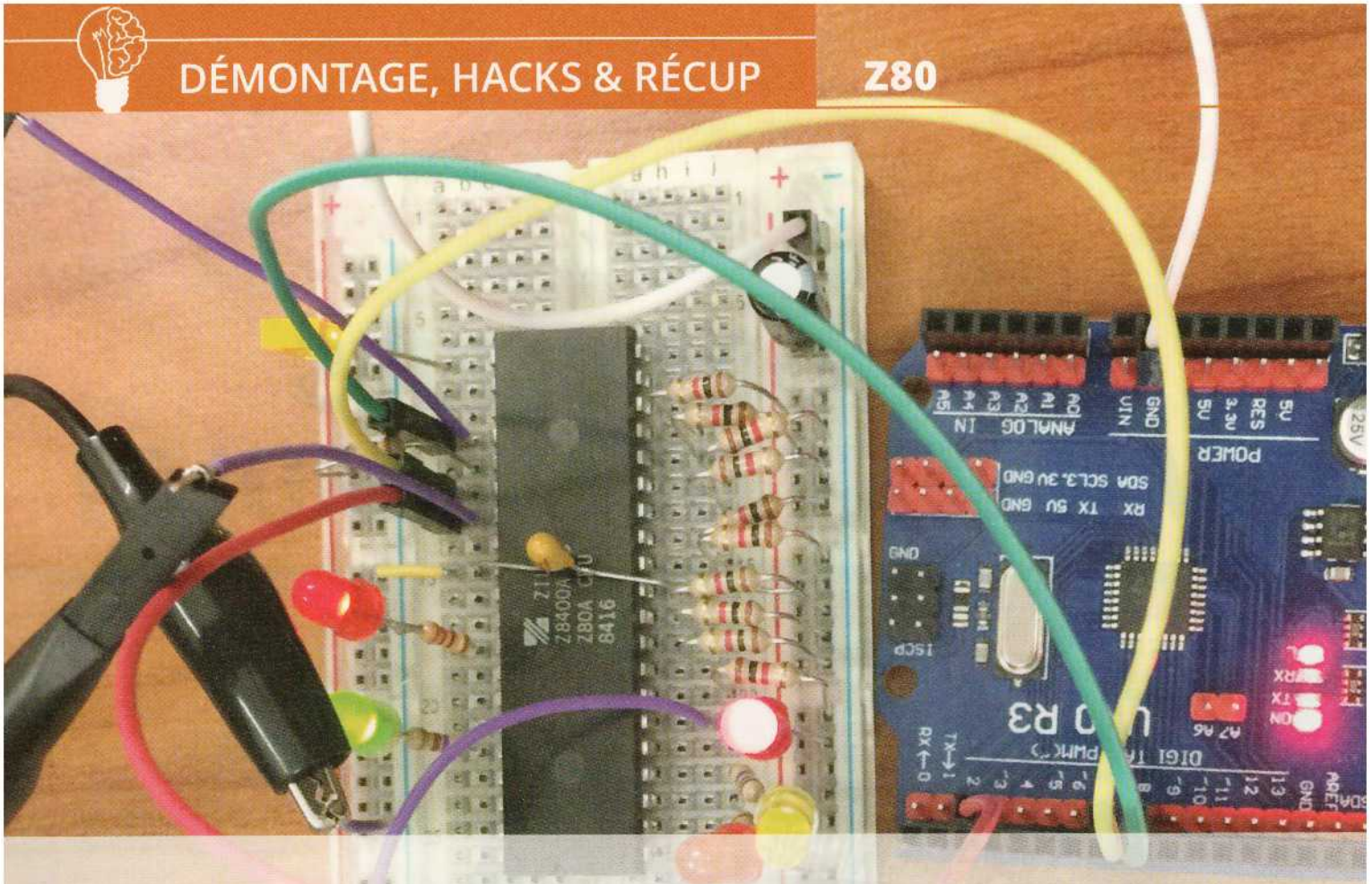
Voici notre montage mis en oeuvre. La led adressable à l'extrême gauche n'est là que pour ajuster les tensions. Elle utilise en entrée un signal en 3,3 volts provenant de l'ESP8266, parfaitement valide puisqu'elle est alimentée en 4,4 volts via la diode. De ce fait, son signal DOUT en sortie, à cette même tension, sera lui-même totalement valide pour le DIN de la led suivante alimentée, elle, en 5 volts.



Tout ce que nous avons donc à faire est d'alimenter une première led au plus proche de la tension minimum, en 4,4 volts par exemple, et toutes les autres en 5 volts pour avoir une intensité lumineuse optimale. Pour ce faire, une simple diode suffit. En effet, une diode très classique comme une 1N4148, a besoin d'une tension minimum, ou tension de seuil, pour être conductrice dans le sens direct ou passant (*Forward Voltage*). Il en résulte une chute de tension entre la cathode et la masse, dépendante du courant qui circule. Un simple test avec un multimètre montre qu'une 1N4148 a une tension de seuil tout à fait typique d'une diode au silicium, soit environ 0,6 volts. En plaçant la diode entre l'alimentation 5 volts et la broche Vdd de la première led, nous l'alimentons donc effectivement en 4,4 volts.

Cette technique permettant d'interfacer des leds WS8212b avec un ESP8266, ou n'importe quelle carte en 3,3 volts, n'utilisera rien d'autre qu'une WS2812b et une simple diode standard très courante. Certes, vous allez sacrifier une led qui sera sensiblement moins lumineuse que les autres, et vous devrez revoir votre code en conséquence, mais cette approche est sans nul doute celle qui sera la plus simple à rapidement mettre en oeuvre. Si le fait de sacrifier une led de votre projet de la sorte vous dérange, sachant qu'elle coûtera sans doute plus cher qu'un circuit logique ou un module d'adaptation de tension, rien ne vous empêche de l'inclure à votre concept, par exemple comme témoin d'état ou de notification.

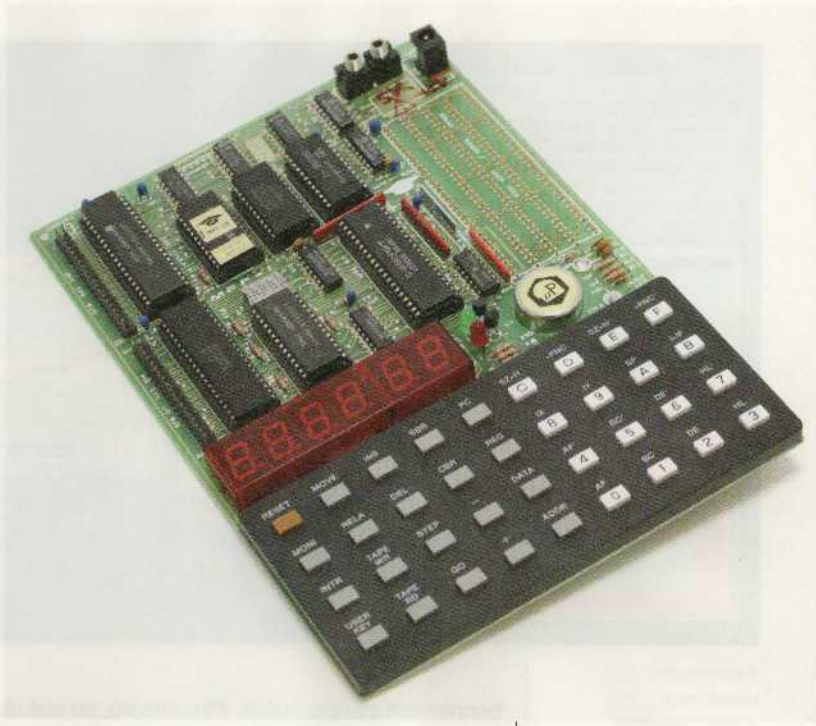
Notez que cette solution n'est pas nouvelle et grandement discutée en ligne de longue date, tant sur son principe que sur l'aspect économique. Gardons à l'esprit que ceci reste avant tout un hack et une solution parmi tant d'autres. Le choix d'une approche ou une autre n'est pas seulement une question de principe, mais aussi, et surtout de tout simplement faire avec ce qu'on a, que ce soit du temps ou des composants... **DB**



ORDINATEUR 8 BITS Z80 : ON PREND LES MÊMES ET ON RECOMMENCE

Denis Bodor

Dans la vie, comme pour les projets, il arrive que des décisions plus ou moins arbitraires prises initialement soient, soit des fantastiques idées fructueuses en devenir, soit de futures catastrophes inéluctables. Dans ce second cas, au mieux, il s'agit d'un handicap auquel le projet devra s'accoutumer tant bien que mal, et au pire, c'est l'élément qui obligera à faire un choix : s'arrêter dans l'impasse et jeter l'éponge ou faire marche arrière et revoir sa copie.



Voici mon exemplaire, légèrement modifié, du vénérable Multitech Micro-Professor MPF-1B. Presque 40 ans d'âge, toujours en parfait état de marche et, surtout, toujours capable de vous apprendre énormément de choses !

Comme je l'avais évoqué dans l'édito du précédent numéro, la vie d'un projet, quel qu'il soit, n'est jamais vraiment un long fleuve tranquille. Si tel était le cas, il n'y aura absolument aucun intérêt à réaliser des montages, découvrir de nouvelles choses et apprendre à résoudre des problèmes. Lorsqu'on est dans une démarche où l'on se fixe un objectif avec un cahier des charges, des ressources finies et un planning de réalisation, il ne s'agit pas d'une approche pédagogique ou d'une quête de savoirs. Dans ce cas-là, on est tout simplement dans une démarche classique de création d'un produit. Or, justement, nous ne sommes pas là pour créer la brosse à dents électrique du futur, la bouilloire connectée ultime et encore moins l'ordinateur familial 8 bits qui va révolutionner la perception de la technologie pour les 15 ans à venir.

De ce fait, vos projets, mes projets, demandent un certain espace de liberté qui en font des éternels « travaux en cours », voire des entités « vivantes » ayant leur propre progression. C'est ainsi qu'un réseau de capteurs, une installation domotique « maison » ou encore un système de notification ou d'éclairage, se voient être améliorés au fil des week-ends et des soirées. Et les choses sont très bien ainsi...

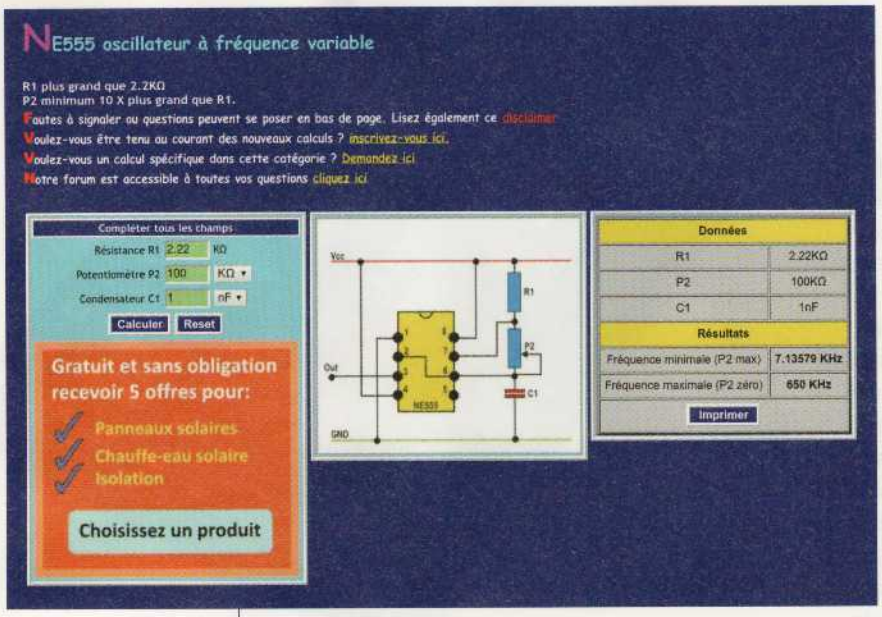
Le dangereux problème qui peut apparaître, cependant, est de se retrouver bloqué dans cette possibilité d'évolution, faute d'avoir pu prévoir les besoins ou les caractéristiques nécessaires à long terme. Voilà précisément où nous en sommes avec notre projet d'ordinateur 8 bits à base de processeur Zilog Z80.

1. ANALYSE DE LA SITUATION

Dans le précédent article sur le sujet (*Hackable n°24*), nous étions arrivés au stade le plus avancé où la carte Arduino émulait la mémoire vive et cadençait l'exécution des instructions par le processeur Z80. Celles-ci étaient produites à partir d'un code en C, avec quelques parties en assembleur, permettant de contrôler notre premier périphérique, un UART *Universal Asynchronous Receiver/Transmitter* 16550 nous fournissant un port série permettant d'afficher du texte et la valeur d'un compteur sur un terminal série (comme le moniteur Arduino).

En ayant suivi les différents articles nous ayant conduits à cette réalisation, vous savez que ce résultat, bien que pouvant paraître désuet au profane, est le fruit d'une collection de connaissances, de concepts et de réalisations, exécutés étape par étape. De la prise en main du Z80 sur platine à essais avec quelques résistances de rappel à l'exécution de ce code, nous avons dû explorer chaque mécanisme formant la base de l'informatique : architecture des ordinateurs, code machine, assemblage, gestion de la pile, compilation, contrôle des bus...

Mais c'est en se posant la question « Quelle est la prochaine étape ? » que le problème se fait jour. Même en poussant à l'extrême le croquis Arduino animant ce montage, nous avons un problème de performance évident. Si vous avez fait l'essai avec votre propre matériel et même en apportant vos modifications expérimentales, le débit de caractères est tout



Il existe de nombreux « simulateurs » de NE555 en ligne. En voici un qui fonctionne à merveille, proposé sur une page perso de telenet.be avec tous les éléments qu'on aime sur les sites d'électroniciens : couleurs flashy, HTML basique, compteur de visites... sans oublier l'incontournable police Comic Sans.

bonnement insupportable. Pire encore, on voit clairement, avant l'affichage de la valeur du compteur, que le nombre de cycles nécessaires aux différentes manipulations des données, impacte ce qui est un affichage à peine passable. Il nous faut plus de puissance !

Plus de puissance ici signifie tout simplement une fréquence d'horloge plus importante. Un processeur Zilog Z80A est en mesure de fonctionner jusqu'à 4 Mhz, mais la carte Arduino, elle, aura bien du mal à émuler les opérations de lecture et d'écriture au-delà de quelques dizaines de kilohertz. Nous nous trouvons donc dans une situation paradoxale où nous pouvons effectivement décharger la carte Arduino de la génération du signal d'horloge, mais devrions tout de même nous adapter à la fréquence de ce nouveau signal, avec des ressources d'émulation limitées.

Comprenez bien ici qu'il est impératif que la carte Arduino et le Z80 soient synchronisés d'une façon ou d'une autre. C'est le seul moyen pour que l'émulation fonctionne. Cette émulation pose un autre problème : l'espace à notre disposition. En effet, bien que le microcontrôleur Atmel ATmega328 d'une carte Arduino UNO dispose de 32Ko de mémoire flash en tout, nous n'avons que 2Ko de SRAM à notre disposition. Ainsi, quelle que soit la façon dont on approche le problème, nous n'aurons jamais autant de mémoire que le Z80 peut en utiliser. Dans le meilleur des cas, en émulant ROM et RAM, nous pourrions placer une partie des instructions pour le Z80 en flash et la pile, ainsi que les autres segments de mémoire accédés en écriture, en RAM. Mais ceci reviendrait à commettre une seconde fois l'erreur consistant à sous-estimer les besoins futurs.

Enfin, dernier problème et non des moindres, la quantité d'entrées/sorties d'une carte comme la UNO est excessivement limitée. Dans la version précédente du montage, nous avons reposé sur deux 74LS165 afin de pouvoir lire le bus d'adresse en n'utilisant que trois broches de la carte Arduino. Ce faisant, nous avons cependant tout de même utilisé toutes les broches disponibles (tout en conservant le moniteur série). Avec l'approche que nous allons voir sous peu, nous devrons prendre en compte deux nouveaux signaux et le fait de se débarrasser de la génération du signal d'horloge n'est pas suffisant.

Une solution possible serait de basculer tout le projet vers la carte Arduino Mega 2560, proposant quelques 54 broches d'E/S. En plus d'imposer l'acquisition de cette plateforme bien moins courante que la UNO (et plus chère), ceci ne règle pas vraiment le problème de mémoire (256Ko de flash, mais seulement 4Ko de SRAM), ni celui de la vitesse d'émulation.

2. LA NOUVELLE APPROCHE

Avant que quelqu'un ne trouve amusant de construire un ordinateur à base de Z80 contrôlé par une carte Arduino, les plateformes d'expérimentation et d'enseignement pour ce processeur existaient déjà. La plupart d'entre elles étaient commercialisées avant même que le « quelqu'un » en question n'ait la moindre idée de ce qu'est un processeur.

Il y a quelques 37 ans, par exemple, la société Multitech (devenue ensuite Acer) commercialisait le Micro-Professor MPF-1, un système d'apprentissage construit autour du Z80, pour apprendre à programmer... un Z80. Cet ordinateur n'avait rien de commun avec des machines « familiales » comme le Commodore 64 ou le ZX81 arrivées peu après. Il s'agissait d'un circuit imprimé peuplé de différents composants, équipé d'un clavier de 36 touches façon calculatrice et d'un afficheur led 7 segments de 6 chiffres. Le tout placé dans un boîtier plastique thermoformé ayant l'aspect d'un livre.

Le MPF-I et sa déclinaison intégrant le langage BASIC, le MPF-IB, ne disposaient pas d'un circuit secondaire de gestion, mais prenaient en charge le contrôle de l'afficheur et du clavier, et donc l'interface avec l'utilisateur, directement sous la forme d'un code en ROM exécuté par le Z80. Pour programmer l'ordinateur, on remplissait tout simplement, via le clavier, la RAM avec une série d'instructions en langage machine, généralement assemblé à la main, sur papier.

Le MPF-I intégrait également un haut-parleur, un circuit d'alimentation, deux circuits d'interface programmables (un Z80 PIO et un 8255), un quadruple timer programmable (Z80 CTC), deux connecteurs jack pour un lecteur de cassettes, ainsi qu'une zone du circuit permettant d'intégrer ses propres composants. Originellement livré avec une ROM de 2Ko et autant de RAM, il était toutefois possible d'ajouter une ROM supplémentaire ou de la RAM via un emplacement DIP. Enfin, des connecteurs d'extension proposant l'accès aux bus (adresse, données, contrôle, PIO, CTC) permettaient aux utilisateurs d'expérimenter et faire évoluer leur machine.

Au début des années 80, le MPF-1B était vendu 1395 francs. Ce qui, converti en euros et ajusté pour l'inflation, nous donne quelques 470€ de nos jours. Du fait que cet ordinateur ait été vendu en quantité importante à l'époque, il est encore tantôt possible d'en trouver des exemplaires relativement bien conservés pour moins de 150 euros (le plastique thermoformé vieillit très mal et part en morceaux).

Le MPF-1 est toujours une plateforme très intéressante pour apprendre et comprendre les principes qui régissent toujours l'informatique moderne. Mais le point important ici est de voir que ce qui était possible à l'époque, sans aucune utilisation de microcontrôleur satellite, est toujours possible aujourd'hui et que les principes architecturaux utilisés sont toujours valables. Inutile pour autant de réinventer la roue, même s'il est parfaitement envisageable de reconstruire de toutes pièces une telle machine, voire d'adapter pour un assembleur moderne les quelques 46 pages de listing assembleur (2659 lignes) relativement faciles à trouver en ligne en version scannée (sinon ce n'est pas amusant).

L'idée consiste plutôt à fusionner les deux approches en conférant davantage d'autonomie au système Z80 tout en conservant la carte Arduino pour inspecter le fonctionnement et éventuellement, par la suite, impacter la mémoire, comme le fait le moniteur du MPF-1, mais sans consommer d'espace en ROM ou demander un développement spécifique. Le principe est que le système Z80 peut être totalement autonome si besoin et la carte Arduino peut être vue comme un outil de développement et de mise au point, un superviseur de bus en somme. C'est ainsi que nous allons approcher les choses désormais.

Remarquez que tout ce que nous avons appris précédemment n'est en rien perdu. Seuls la réalisation elle-même et les composants à utiliser demanderont une révision, étape par étape. Nos connaissances acquises sur le fonctionnement du Z80, le langage machine, la compilation ou encore la façon dont la mémoire est gérée, sont toujours d'actualité et il en va de même pour la majorité du code que nous avons écrit, aussi bien pour le Z80 que pour la plateforme Arduino.

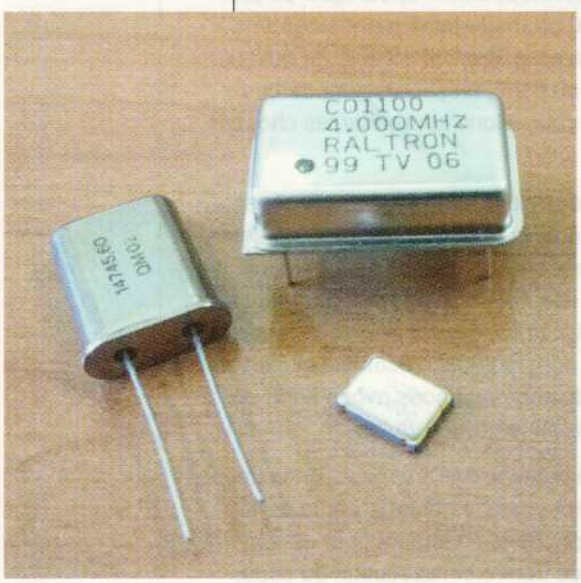
Mais avant de construire cette nouvelle version, nous devons ajouter les briques qui étaient prises en charge par la carte Arduino et prendre en compte ce changement pour découvrir un mécanisme permettant de contrôler l'exécution du code.

3. CIRCUIT D'HORLOGE

Depuis la toute première itération du projet, et jusqu'à notre dernier article, c'était la carte Arduino qui cadençait l'exécution des instructions par le Z80 via la fonction `doClock()` appelée depuis `loop()`. La fréquence de ce signal pouvait être arbitrairement choisie avec une simple définition de macro (`UCLKDELAY` ou `CLKDELAY`). La génération d'un signal sous la forme d'une succession d'états haut et bas sur l'une des broches de la carte Arduino doit être remplacée par un circuit indépendant.

Plusieurs approches sont possibles et tout dépend de ce que nous souhaitons obtenir. Nous pourrions prendre exemple sur des architectures existantes comme le ZX Spectrum mais, ici comme dans bien des cas, le signal d'horloge est généré par un circuit dédié. Dans le cas du ZX Spectrum, c'est l'ULA qui génère ce signal (et d'autres). Nous avons besoin d'une solution plus simple.

En regardant du côté des circuits d'horloge proposés sur le Web concernant les réalisations « maison », le très classique NE555 revient souvent. Il présente l'avantage de permettre, à l'aide d'un condensateur, d'une résistance et d'un potentiomètre, d'ajuster la fréquence et donc d'obtenir un oscillateur à fréquence variable. J'avais déjà partiellement couvert l'utilisation du NE555 en configuration astable dans un article du numéro 7, et cette solution semble séduisante.



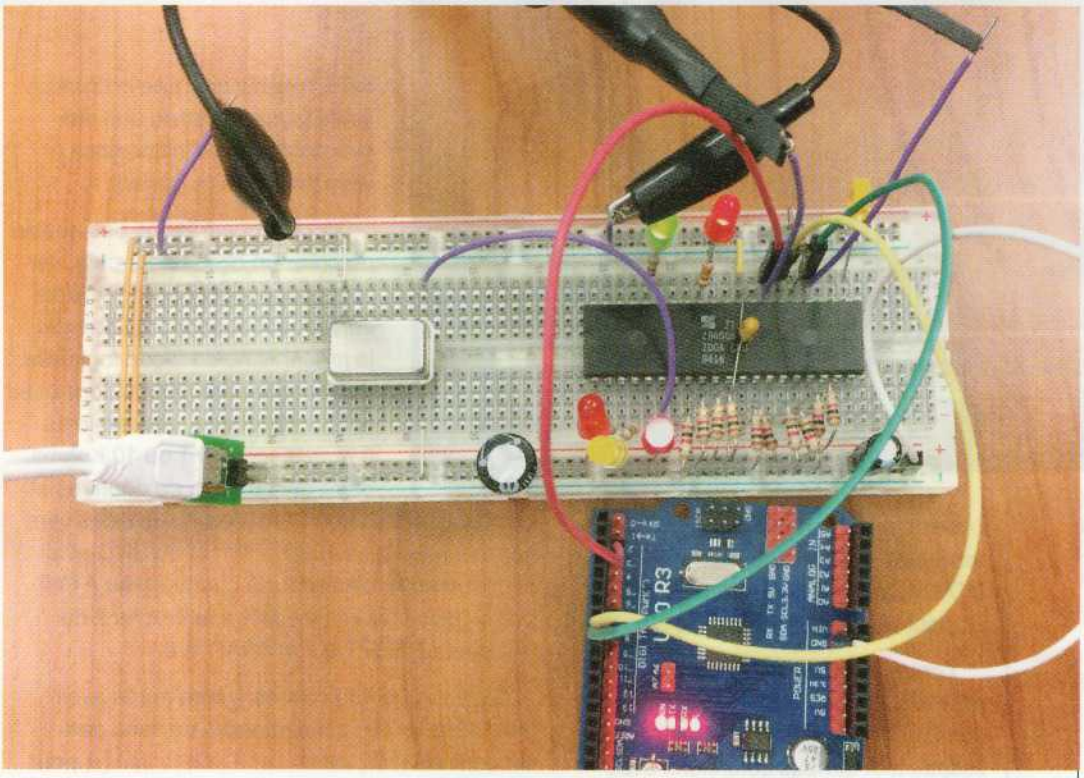
Deux composants à ne surtout pas confondre, à gauche un quartz et à droite deux formats d'oscillateurs à quartz (DIP et SMD). Le premier est un cauchemar à utiliser et les seconds un jeu d'enfant.

Nous devons cependant penser aux limitations et en particulier à la fréquence maximum qu'il est possible d'obtenir avec un NE555. Dans le cas d'une version CMOS en mode astable, cette fréquence est de 2 Mhz dans de bonnes conditions (pas sur une platine à essais donc). Ceci est deux fois moins que ce qu'un Z80A peut utiliser. De plus, une fréquence variable n'aura que peu d'importance pour notre application, car nous allons utiliser un mécanisme dédié pour permettre à la carte Arduino de suivre l'exécution des instructions par le Z80.

Une autre solution consiste à utiliser un 4060, mais là encore la fréquence maximum n'atteint pas les 4 Mhz. Il est par contre possible d'utiliser un quartz pour obtenir une fréquence plus stable. Chose également valable pour un circuit utilisant un CD4521, mais ceci ouvre en réalité les portes vers toute une gamme de circuits. Il est, en effet, possible de mettre en œuvre un quartz avec presque n'importe quel circuit logique. C'est d'ailleurs précisément ce que fait, en interne un microcontrôleur comme l'ATmega328 d'une carte Arduino.

Lorsqu'on veut cependant construire un tel circuit en partant de zéro, on se heurte rapidement à des difficultés dans le choix des composants et surtout dans le calcul des valeurs à utiliser. Le plus connu de ces circuits est l'oscillateur Colpitts, du nom de son inventeur, Edwin H. Colpitts, mais un simple coup d'œil à la page dédiée sur Wikipédia vous fera comprendre que la tâche n'est pas si simple. Il n'y a pas de honte à avoir à se montrer réticent devant une telle tâche, car non seulement notre objectif n'est pas de construire un oscillateur de toutes pièces, mais cette difficulté en a effarouché plus d'un.

Pour preuve, l'industrie électronique a trouvé une solution permettant d'éviter tous ces tracas : l'oscillateur à quartz. Celui-ci ne doit pas être confondu avec le quartz lui-même qui devrait plutôt être



Les résistances sont de retour sur le bus de données, mais cette fois le processeur Zilog Z80A fonctionne à 4 Mhz. Il ne fait donc à nouveau rien, mais cette fois, il le fait vraiment beaucoup plus vite.

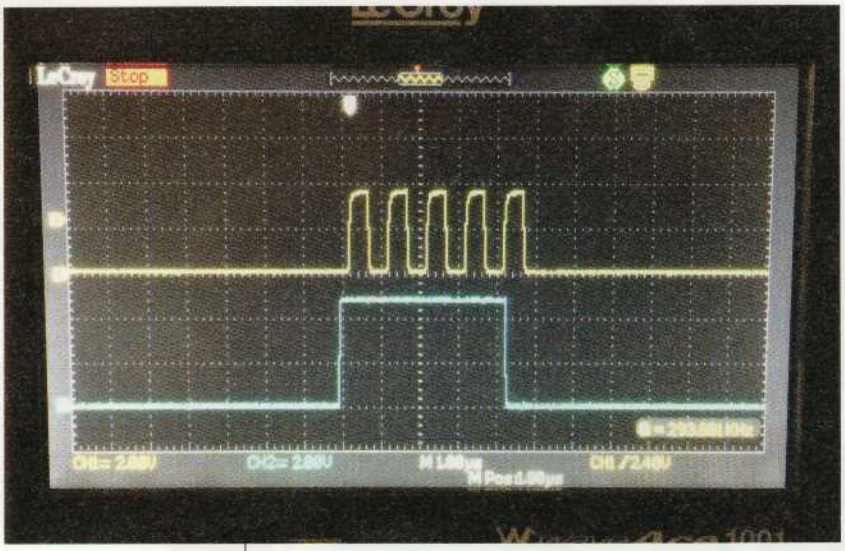
désigné par les termes de « résonateurs piézoélectriques ». Un quartz est un composant simple constitué d'un quartz taillé et l'oscillateur à quartz est un circuit complet, utilisant un quartz, dans un boîtier métallique. Ce type de composant est tantôt désigné par l'acronyme MCO pour *Multi-Component Semiconductors*.

Ces composants sont très simples d'utilisation et disposent de 4 broches : une masse, une alimentation, le signal en sortie et une patte non utilisée. Leur mise en œuvre est tout aussi simple puisqu'il suffit de l'alimenter correctement et nous obtenons un signal carré à la fréquence souhaitée sur la sortie. Deux caractéristiques sont importantes pour ces composants : la tension d'alimentation (+5V) et la fréquence. Acquérir un oscillateur à quartz à 1, 2 ou 4 Mhz pourra se faire en passant par les filières habituelles, mais il vous est également possible de tout simplement le récupérer sur du vieux matériel, comme une ancienne carte mère de PC par exemple.

4. EXÉCUTION PAS-À-PAS

Le Z80 présente un avantage certain du point de vue de la génération de la fréquence d'horloge par rapport à, par exemple, le MOS 6502 : il n'a pas besoin d'utiliser une fréquence d'horloge minimum. Ceci nous permettait donc d'abaisser cette fréquence à notre convenance et donc, initialement, d'utiliser un signal généré par la carte Arduino.

Cette facilité nous a permis d'exécuter nos codes suffisamment lentement pour suivre le déroulement des opérations à une vitesse « humaine ». Cela signifie-t-il donc pour autant qu'une telle chose est impossible avec un MOS 6502, un Motorola 6809 ou encore un Intel 8080, du fait qu'ils utilisent de la mémoire dynamique pour leurs registres ? Non, car il existe toujours une solution pour mettre le processeur en pause. Initialement, ce mécanisme et sa broche dédiée sont faits pour la gestion des mémoires lentes, incapables de répondre aussi vite que le processeur



En haut, le signal /M1 et en bas, le signal / WAIT. Vous voyez le problème ? Et encore, dans ce cas précis, c'est « PORTB &= ~_BV(PB0) » qui est utilisé et non « digitalWrite(B_WAIT, LOW) »...

l'attend. Avec le Motorola 68010 par exemple, cette broche est nommée DTACK, pour le MOS 6502 c'est RDY et, pour le Z80, c'est /WAIT.

Lorsque la broche /WAIT (24) est mise à la masse, le processeur passe dans un état d'attente et l'exécution est suspendue. Si nous contrôlons avec précision ce signal, nous pouvons donc cadencer l'exécution du code à un rythme arbitraire, indépendamment de la fréquence d'horloge appliquée sur la broche PHI (6). Bien entendu, ceci ne peut pas être fait à n'importe quel moment puisque, comme nous l'avons vu dans les précédents articles, certaines instructions peuvent prendre plus d'une dizaine de cycles d'horloge pour être traitées.

Heureusement pour nous, le Z80 est en mesure de nous donner une indication concernant l'étape du traitement des instructions. Le signal /M1 est actif (passe à la masse) au moment de la récupération de l'instruction (*opcode fetch*). Nous pouvons donc stopper l'exécution à ce moment précis en contrôlant /WAIT. Une implémentation matérielle de ce mécanisme met en œuvre une bascule (D Flip-flop), un bouton poussoir avec un circuit anti-rebond et deux NE555 afin de permettre, au choix, une exécution standard, l'exécution d'une instruction seule (*single-step*) ou une exécution continue d'instructions à vitesse réduite (*autostep*).

Nous, nous n'avons pas besoin de créer un circuit pour cela, car nous pouvons reposer sur notre carte Arduino pour implémenter le mécanisme de façon logicielle. Retour à la case départ donc avec le Z80 sur platine, une série de 8 résistances de 1 ou 10 Kohms, entre les broches de données et la masse (instruction **NOP**), des résistances vers +5V pour les lignes /INT, /NMI et BUSRQ et la sortie de notre oscillateur à quartz reliée à PHI. Enfin, nous connectons /M1 à la broche 2 de l'Arduino, /RESET à 7 et /WAIT à 8.

Tout ce que nous avons à faire après avoir réinitialisé le Z80 est d'attendre un flanc descendant sur /M1, via **attachInterrupt()**, pour passer /WAIT à l'état bas. Après une brève temporisation arbitrairement choisie, on repasse /WAIT à l'état haut en attendant le prochain flanc descendant sur /M1, etc. :

```
#define RSTDELAY  100

#define B_RESET     7
#define B_M1        2
#define B_WAIT      8

volatile int actm1=0;

// réinitialisation
void doReset() {
  digitalWrite(B_RESET, LOW);
  delay(RSTDELAY);
  digitalWrite(B_RESET, HIGH);
}

// routine d'interruption
void m1ISR() {
  if(!actm1) {
    // dodo Z80
    digitalWrite(B_WAIT, LOW);
    actm1=1;
  }
}
```

```
void setup() {
  // configuration des E/S
  pinMode(B_RESET, OUTPUT);
  pinMode(B_M1, INPUT);
  digitalWrite(B_WAIT, HIGH);
  pinMode(B_WAIT, OUTPUT);

  doReset();

  // Appel routine sur changement d'état de la broche 2
  attachInterrupt(digitalPinToInterrupt(B_M1), m1ISR, FALLING);
}

void loop() {
  // /M1 bas ?
  if(actm1) {
    // pause
    delay(1);
    // réinitialisation
    actm1=0;
    // "réveil" du Z80
    digitalWrite(B_WAIT, HIGH);
  }
}
```

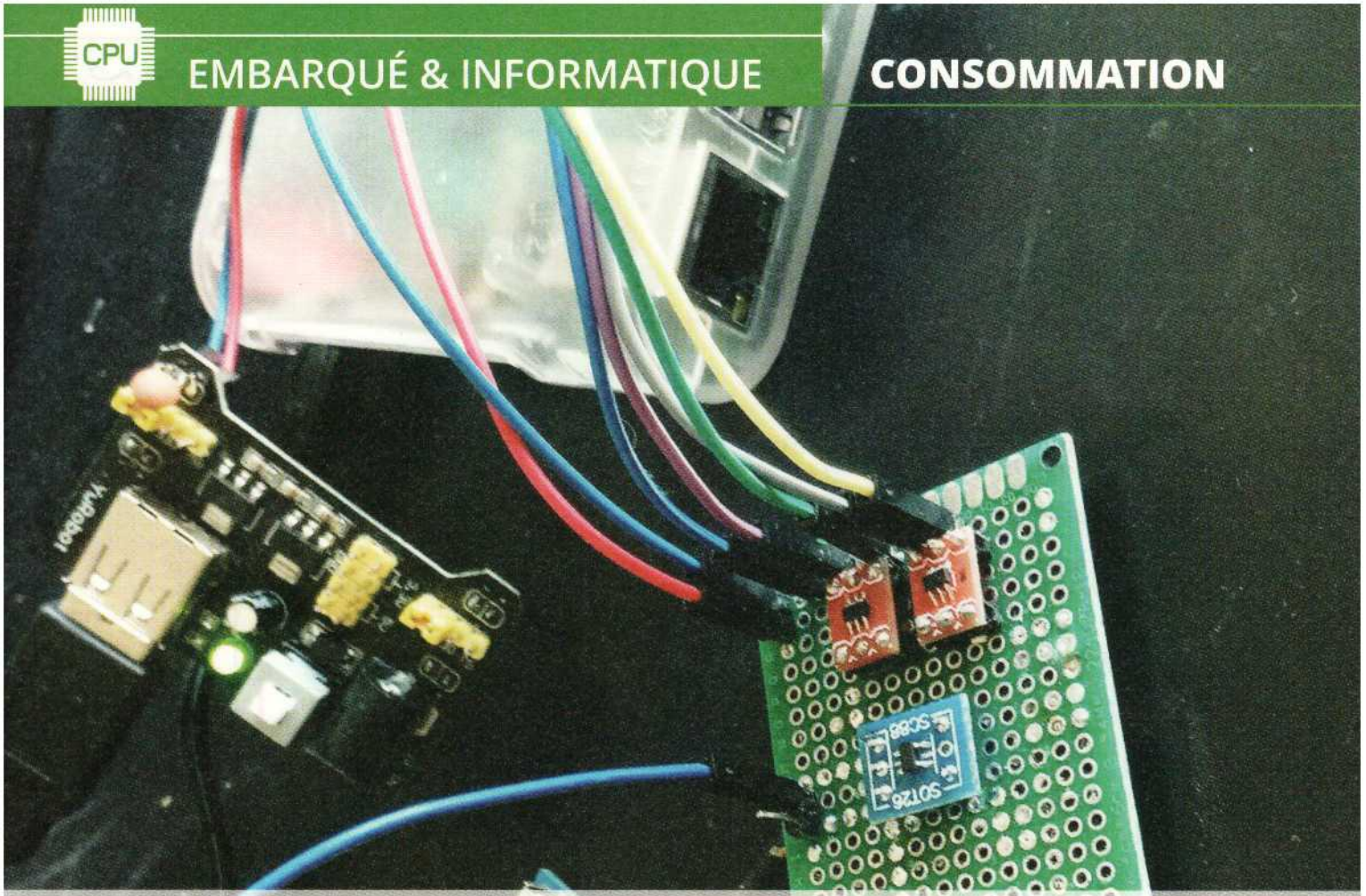
Pour surveiller l'exécution de façon simple, nous n'avons qu'à connecter quelques leds et leurs résistances entre les broches du bus d'adresse et la masse. Nous devrions alors voir les adresses défiler à mesure que le Z80 exécute les instructions `NOP` qui peuplent virtuellement toute la mémoire.

LA PROCHAINE ÉTAPE

Nous sommes presque littéralement revenus à la case départ de notre projet, mais il sera relativement facile et rapide de réintégrer les précédents travaux selon cette nouvelle optique. Un gros changement cependant va intervenir, car en lieu et place de l'émulation de la mémoire, nous allons devoir utiliser de véritables composants : une EPROM (ou EEPROM) et de la mémoire statique. Les composants choisis pour cela sont respectivement un Atmel AT28C256-15P (32 Ko) et un Cypress CY62256NLL-70 (32 Ko). Nous aurons besoin de circuits logiques (74HC04 et 74HC32) pour diviser l'espace d'adressage entre ROM et RAM.

Enfin, il nous faudra une petite collection de 74HC165 et de 74HC595 afin d'interfacer notre carte Arduino aux bus du Z80, à la fois pour lire et écrire. Trois 74HC165 pourraient être suffisants pour simplement lire les bus d'adresse et de données, mais nous en profiterons certainement pour nous pencher sur la réalisation d'un programmeur d'EEPROM. Bien entendu, la solution de facilité serait d'utiliser un programmeur économique comme celui que nous avons traité dans *Hackable n°17*. Mais cet investissement ne saurait être obligatoire si vous n'en avez pas l'usage par ailleurs.

Mais avant d'en arriver là, nous devons régler un problème, un gros problème, qui vient tout juste de se faire jour. Pourrez-vous le trouver ? Deux indices : quelque chose n'est pas normal dans le comportement du montage lorsqu'on change la fréquence de l'oscillateur et un oscilloscope permet de clairement voir le problème. Si vous trouvez, vous serez sans doute amusé de voir que les 40 ans qui séparent Z80 et Arduino n'impliquent finalement pas grand-chose... **DB**



ROBOTIQUE ET ÉLECTRONS : MESURER UNE CONSOMMATION AVEC LE RPI

Laura Bécognée - **hackaday.io/Aqueuse**

Où on continue d'avancer vaille que vaille vers les bases d'une robotique sensible et adaptative.

Cet article est le second d'une trilogie commencée dans *Hackable n°24*, visant à créer un banc de test pour moteurs universels. Nous avions créé ensemble une carte dotée d'un convertisseur analogique-numérique permettant de récupérer dans un Raspberry Pi la tension fournie à un moteur quelconque. Nous devions maintenant dans ce second article améliorer ce banc de test avec un CSA (*Current Sense Amplifier*) et un deuxième CAN, afin qu'il soit aussi capable de mesurer l'intensité fournie à ce moteur, pour avancer peu à peu vers un contrôle total de sa consommation.

1. INTRODUCTION

Afin de rester le plus simple possible, j'ai tenté de construire un CSA avec des transistors et des résistances en suivant le schéma en Figure 1.

Malheureusement, après de longues et coûteuses recherches et des semaines d'échecs successifs, il m'a été totalement impossible de faire fonctionner ce CSA maison de façon satisfaisante et je me suis donc repliée vers un CSA en circuit intégré. Après avoir acheté par erreur un amplificateur de mesure de courant (INA286) suite à de mauvais conseils et perdu encore plusieurs semaines suite à diverses péripéties, j'ai enfin trouvé et acquis la perle rare : un authentique CSA de Texas Instrument, le INA195, parfaitement adapté à nos conditions d'utilisation.

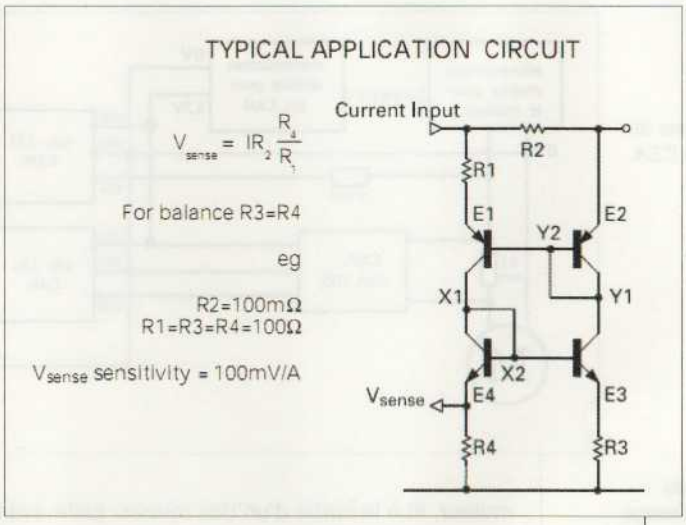


Fig. 1 : Schéma du CSA théorique (à vous de relever le défi ;-)).

Grâce à un ami, j'ai pu le souder sur un adaptateur qui m'a permis de l'intégrer au circuit exactement comme je l'avais fait pour les CAN (Convertisseurs Analogiques-Numériques) qui communiquent avec le RPI. Nous voilà donc repartis d'un bon pied, et après une longue absence nous pouvons continuer à améliorer le banc de test.

2. CSA : CURRENT SENSE AMPLIFIER

Comme le CSA cité précédemment, son fonctionnement se base sur un double miroir de courant couplé à une résistance de shunt, c'est-à-dire une résistance de très faible valeur qui sert seulement à récupérer la valeur du courant et doit donc très peu influencer le résultat (j'ai utilisé pour ma part une 0,1 Ohm) (Figure 2).

L'idée est la suivante : on crée un double miroir de courant. Comme son nom l'indique de façon un peu sibylline, il permet de dupliquer un courant, et comme son nom ne l'indique pas, on peut aussi avec un peu d'astuce l'utiliser pour transformer un courant en tension : on branche une résistance à ses bornes, ainsi que notre

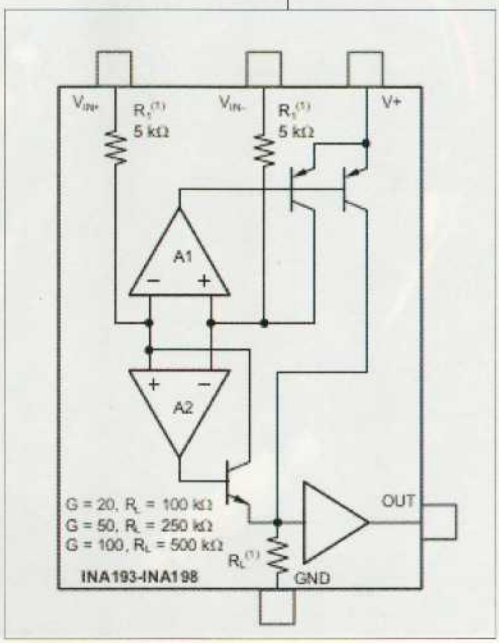


Fig. 2 : Schéma de l'intérieur du CSA INA 195, tiré de son datasheet.

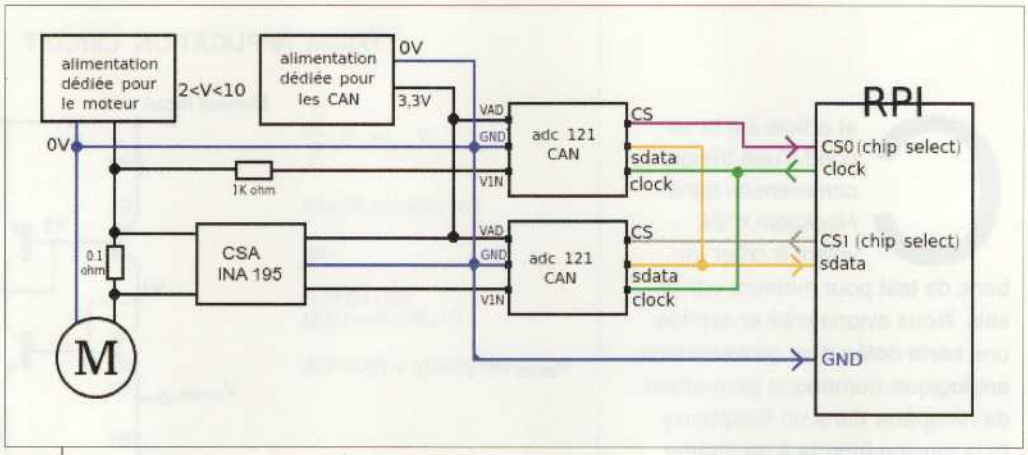
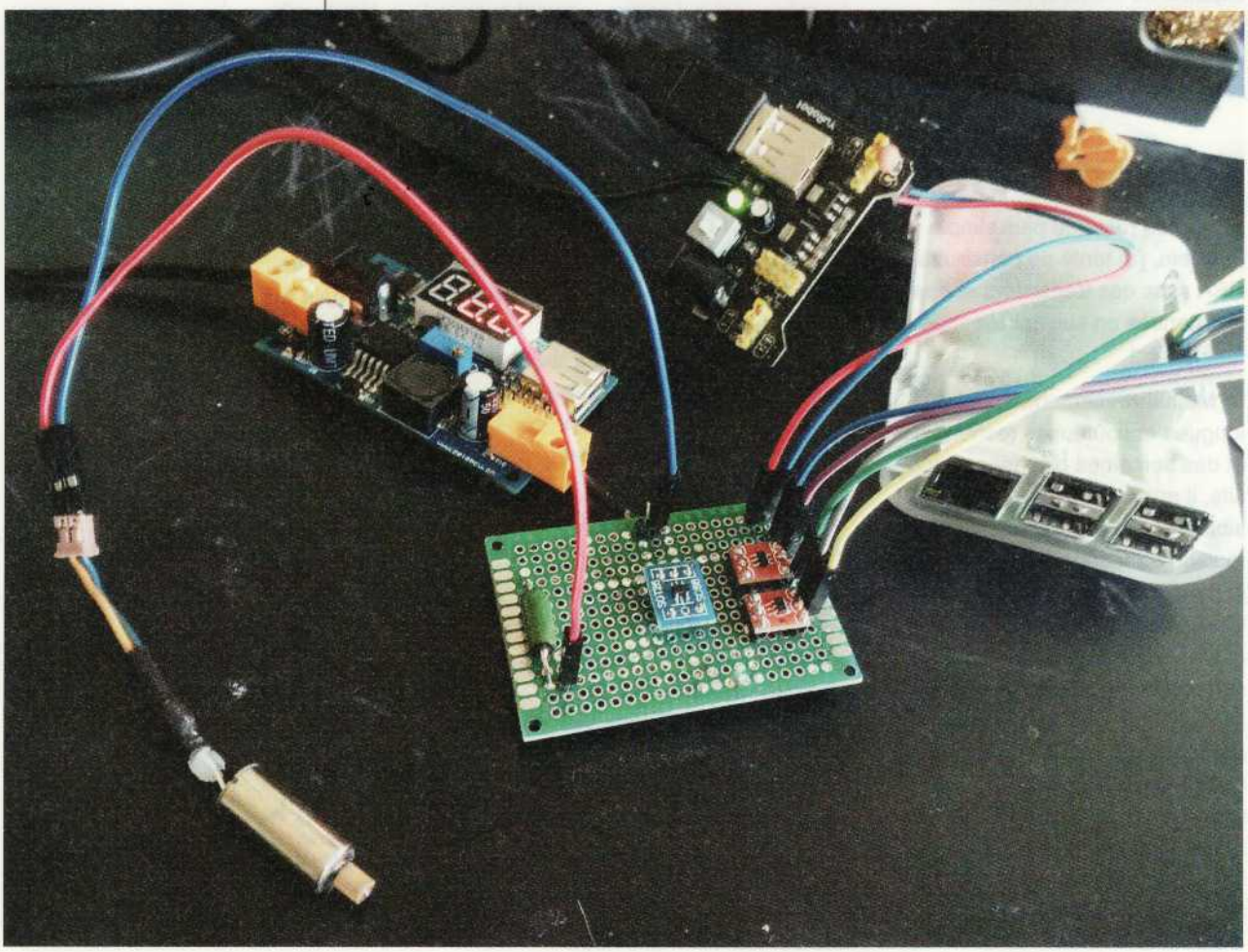


Fig. 3 : Schéma du circuit avec le CSA.

Fig. 4 : L'ensemble du banc de test prêt à l'usage. Notez l'alimentation séparée pour les moteurs et les composants de la carte, afin de ne pas perturber les mesures.

moteur, et à la sortie d'un des miroirs, juste avant une des résistances, on branche notre CSA. Normalement, on aurait courant - tension - courant, et on ne serait pas plus avancé, mais grâce à ce repiquage anticipé, on a courant - tension.

Futé, n'est-ce pas ?



3. ET LA CONSOMMATION DANS TOUT ÇA ?

On est désormais en possession de tous les éléments pour achever la première version complète du banc de test : un de nos CSA récupère la tension et l'autre récupère le courant consommé par notre moteur. Y a plus qu'à ! Modifions le code en python afin d'inclure notre deuxième CSA.

Rappelez-vous, on était arrivé à ce code :

```
#!/usr/bin/python
import spidev

# creation de l'objet SPI et initialisation
spi = spidev.SpiDev()
spi.open(0,0)

data = spi.xfer2([0,0], 8000000, 0, 8)

# Oups, la réponse arrive sur deux octets
msb = data[0]
lsb = data[1]
value_brute = (msb << 8) + lsb
value_mV = (value_brute * 3.3) / 4095

print (bin(data[0]) + " + " + bin(data[1]) + " = " + bin(value_brute))
print value_mV

spi.close()
```

On va ajouter un deuxième objet SPI : **spi1**, qui passera par les broches SPI1 du Raspberry Pi. Cet objet récupèrera les données du second CSA et nous les traduirons en décimal pour pouvoir les lire.

```
#!/usr/bin/python
import spidev

# creation des objets SPI et initialisation
spi0 = spidev.SpiDev()
spi1 = spidev.SpiDev()

spi0.open(0,1)
spi1.open(0,0)

# on affiche le resultat en forcant l'affichage en binaire
data0 = spi0.xfer2([0,0], 8000000, 0, 8)
data1 = spi1.xfer2([0,0], 8000000, 0, 8)

# CS0
msb0 = data0[0]
lsb0 = data0[1]
value_brute0 = (msb0 << 8) + lsb0
value_mA = (value_brute0 * 3.3) / 4095

# CS1
msb1 = data1[0]
lsb1 = data1[1]
value_brute1 = (msb1 << 8) + lsb1
```

```
value_mV = (value_brute1 * 3.3) / 4095

print value_mA
print value_mV

spi.close()
```

À quoi nous pouvons ajouter le calcul de la consommation en Watt par heure : P=U * I.

```
# calcul de la consommation du moteur
consommation = value_mV * value_mA

print consommation
```

4. ET PAF, ÇA FAIT DES DONNÉES !

Techniquement, on a réussi : on a pris un moteur, on l'a branché à un circuit capable de calculer la tension à laquelle il tourne et le surplus de courant qu'il occasionne et on en a déduit une consommation. On a les premières briques pour bricoler. Mais pour surveiller réellement la consommation du moteur dans le temps et analyser finement ses réactions au stress, il nous reste deux problèmes : le paramètre temps et la lisibilité. Il serait plus pratique à ce stade d'afficher les résultats de façon plus lisible et en échantillonnant à intervalle régulier, on pourrait voir apparaître des tendances, des pics, des creux et les interpréter !

4.1 Échantillonner

La solution la plus élégante que j'ai trouvée consiste à créer une liste par paramètre, chacun dans un fichier texte. Il faut d'abord créer une boucle pour répéter la lecture des CSA. Avec la librairie `time`, c'est facile : on crée une boucle toujours vraie, qu'on interrompra en quittant l'exécution du code avec CTRL+C et à l'intérieur de cette boucle on ajoute une pause d'une seconde avec `time.sleep(1)` afin de laisser notre Raspberry Pi respirer.

Voici maintenant le début de notre fichier, qui ne bougera plus. Attention à ne pas oublier d'indenter le code dans la boucle.

```
#!/usr/bin/python
import spidev
import time

# creation des objets SPI et initialisation
spi0 = spidev.SpiDev()
spi1 = spidev.SpiDev()

spi0.open(0,1)
spi1.open(0,0)

while True:
 # on affiche le resultat en forcant l'affichage en binaire
 data0 = spi0.xfer2([0,0], 8000000, 0, 8)
 data1 = spi1.xfer2([0,0], 8000000, 0, 8)

 time.sleep(1)
```

Il nous faut ensuite récupérer les résultats, à savoir les variables `value_mV`, `value_mA` et `consommation`. Au terme de petites recherches sur Internet, j'ai trouvé la solution la plus simple sur **apprendre-python.com [1]** : open. Il permet de créer, de lire et d'écrire dans des fichiers.

Voilà ce que ça va donner pour la tension :

```
tension_print = open("tension_ADC.txt", "a")
tension_print.write(str(value_mV)+"\n")
tension_print.close()
```

Attention à bien utiliser le mode `a` dans notre cas : on ajoute les données sous forme de liste verticale dans le fichier, ce qui sera beaucoup plus pratique à travailler par la suite. On n'oubliera donc pas le retour à la ligne avec `\n`. Subtilité du python : il faut transformer les variables en string pour pouvoir les écrire dans un fichier et donc les englober dans un `str()`.

Il ne reste plus qu'à créer les fichiers, puis exécuter le code. Pour ma part, j'ai créé un petit script shell afin de pouvoir rapidement vider les fichiers et recommencer la lecture depuis zéro.

```
rm tension_ADC.txt
touch tension_ADC.txt

rm intensite_ADC.txt
touch intensite_ADC.txt

rm consommation_ADC.txt
touch consommation_ADC.txt
```

À chaque fois que je voudrais tester un nouveau moteur, je n'aurai qu'à faire :

```
sh empty.sh && python intensite_adc_SPI.py
```

Je vous affiche le code final pour vous aider à y voir plus clair :

```
#!/usr/bin/python
import spidev
import time

# creation des objets SPI et initialisation
spi0 = spidev.SpiDev()
spi1 = spidev.SpiDev()

spi0.open(0,1)
spi1.open(0,0)

while True:
 # on affiche le resultat en forcant l'affichage en binaire
 data0 = spi0.xfer2([0,0], 8000000, 0, 8)
 data1 = spi1.xfer2([0,0], 8000000, 0, 8)

 time.sleep(1)

 # CS0
 msb0 = data0[0]
```

```
lsb0 = data0[1]
value_brute0 = (msb0 << 8) + lsb0
value_mA = (value_brute0 * 3.3) / 4095

# CS1
msb1 = data1[0]
lsb1 = data1[1]
value_brute1 = (msb1 << 8) + lsb1
value_mV = (value_brute1 * 3.3) / 4095

# calcul de la consommation du moteur
consommation = value_mV * value_mA

print value_mA
print value_mV
print consommation

tension_print = open("tension_ADC.txt", "a")
tension_print.write(str(value_mV)+"\n")
tension_print.close()

intensite_print = open("intensite_ADC.txt", "a")
intensite_print.write(str(value_mA)+"\n")
intensite_print.close()

consommation_print = open("consommation_ADC.txt", "a")
consommation_print.write(str(consommation)+"\n")
consommation_print.close()

spi0.close()
```

5. GNUPLOT

Il serait assez insensé de penser réussir à faire un tour complet de ce logiciel dans un si court article, d'autant que nous allons utiliser GNUplot de façon très sommaire. Il permet de faire beaucoup plus qu'un graphique en 2D avec trois variables. Je vous invite à consulter les tutoriels disponibles sur Internet et notamment celui-ci **[2]**, hébergé par mon université d'étude (cocorico !). Il est assez complet pour qu'on y trouve le nécessaire et assez sommaire pour ne pas se noyer inutilement dans les détails.

Après avoir installé GNUplot on va le lancer, mais en faisant bien attention à le faire dans le répertoire où seront hébergés les fichiers de données. Ce sera beaucoup plus pratique pour la suite. Vous allez voir.

```
Apt-get install gnuplot
cd resultats_banc_de_test/
gnuplot
gnuplot>
```

Il faut ensuite utiliser la fonction `plot`, en utilisant nos fichiers comme données d'entrée. Pour afficher l'évolution de la tension, on fera par exemple :

```
gnuplot > plot 'tension_ADC.txt'
```

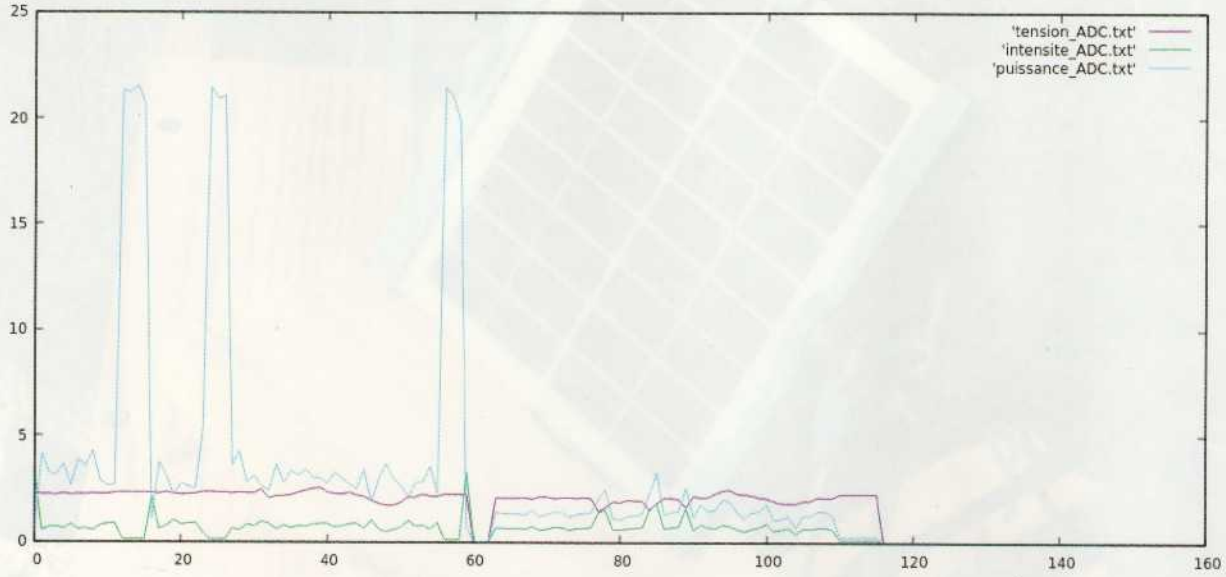


Fig. 5 : Le résultat graphique avec GNUplot.

Mais ce n'est pas tout, il faut aussi préciser à GNUplot que nous voulons afficher ces données, ou on obtiendra alors seulement une liste de points à relier et on ne sera pas beaucoup plus avancé niveau lisibilité. Au final, pour tout afficher proprement, voici la commande qu'il nous faut :

```
gnuplot> plot 'tension_ADC.txt' with lines,'intensite_ADC.txt' with
lines, 'consommation_ADC.txt' with lines
```

Je vous laisse deviner à quels moments j'ai branché le moteur, à quel moment j'ai fait varier (très légèrement) la tension et à quels moments j'ai perturbé la rotation du moteur. Les pics sont assez explicites et fascinants à étudier.

CONCLUSION

Cet outil se révèle de plus en plus pratique, même si je lui associerai bien le variateur de courant conçu par Whygee et présenté sur **Hackaday.io [2]** afin de contrôler absolument tous les paramètres d'entrée. Il nous faut maintenant découvrir le comportement de la tension, de l'intensité et de la consommation qui en résulte lors des changements de rotation du moteur et pourquoi pas tester celui des servomoteurs en ajoutant un driver de moteur. Ce sera l'objet du prochain et dernier article de cette série.

Dans une optique de gestion autonome des moteurs, il serait aussi intéressant de pouvoir contrôler la tension et l'intensité directement depuis le Rpi et d'y détecter la vitesse et le sens de rotation des moteurs. Ainsi, nous pourrions séquencer l'intégralité de leur fonctionnement avec du code python et plonger véritablement dans la gestion adaptative et la robotique.
Stay tuned ! **LB**

RÉFÉRENCES

[1] **http://apprendre-python.com/page-lire-ecrire-creer-fichier-python**

[2] **https://hackaday.io/project/159693-precision-current-generator**

ÉTUDE D'UN TRAQUEUR SOLAIRE

Jannick Paris

Dans un article précédent, nous avons vu la technologie des capteurs photovoltaïques. Connaître la théorie c'est bien, mais rien ne vaut un peu de pratique. On va s'intéresser au gain apporté par un traqueur solaire. Est-il rentable (énergétiquement parlant) d'ajouter des moteurs pour conserver la meilleure orientation ?

La technologie des capteurs photovoltaïques ne permet qu'un rendement théorique maximal de 30 %. Il est donc intéressant, voire nécessaire, de faire en sorte que les conditions idéales soient toujours réunies. Notamment l'orientation du capteur par rapport au soleil.

1. POURQUOI SUIVRE LA POSITION DU SOLEIL ?

1.1 Petits rappels de trigonométrie

Pour bien comprendre l'influence de ce paramètre, il va falloir faire un peu de trigonométrie. Mais comme écrit au dos du très célèbre guide du voyageur galactique : « Don't panic », nous n'allons pas nous lancer dans de longues et ennuyeuses démonstrations. Mais il est nécessaire pour la suite de faire quelques rappels.

Selon Wikipédia, la trigonométrie est une branche des mathématiques qui traite des relations entre distances et angles dans les triangles. Mais avant toutes choses, pour bien comprendre, il nous faut un repère (voir figure 1).

Afin de saisir l'importance de cette discipline dans le cas que nous étudions, regardons la figure 2. Dans ce cas, on remarque vite que quelque chose ne va pas. Le capteur a une longueur de 1 unité, cependant il ne capte la lumière que de 0 à 0,5. En termes mathématiques, on dit que : cos(60) = 0,5. Ce qui signifie simplement que le capteur sera deux fois moins efficace ... Si on note Φ le flux de lumière qui arrive sur Terre, Φ_{eff} le flux effectif qui arrive sur le capteur et α l'angle que forment les rayons lumineux avec la normale (droite perpendiculaire à la surface) du capteur, on peut généraliser en écrivant : $\Phi_{eff} = \Phi\cos(\alpha)$. Sachant que le cosinus (voir figure 3, page suivante) est une fonction qui varie, sur sa partie positive, entre 0 et 1, on comprend qu'il faut à tout prix éviter de s'approcher de 0, sinon le capteur sera tout bonnement inutile.

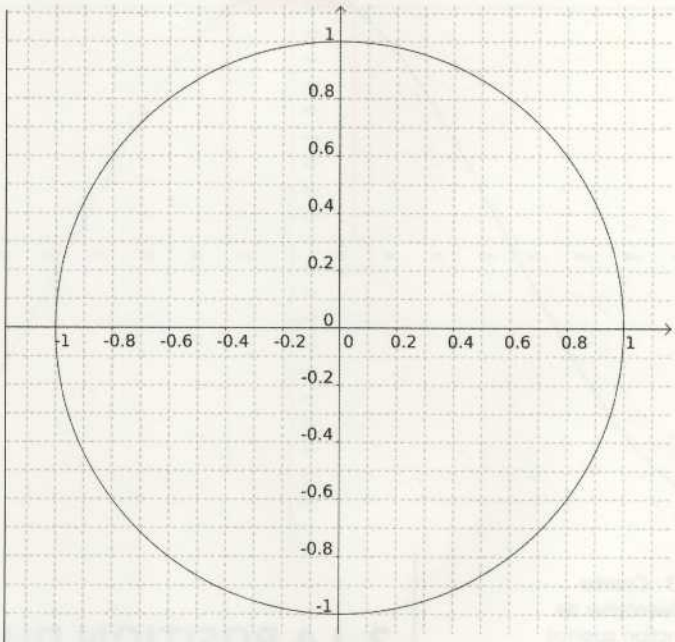
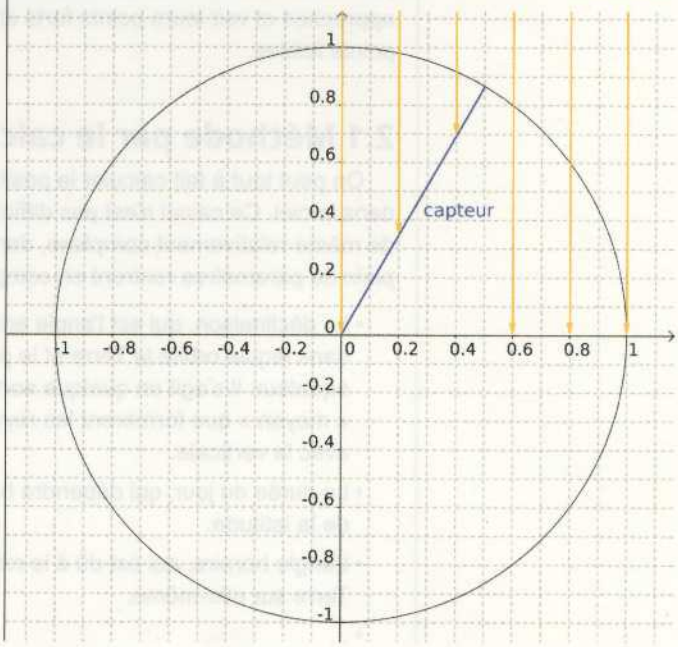


Fig. 1 : On utilise généralement un repère orthonormé, c'est-à-dire que les axes sont perpendiculaires et que les unités sur ces axes sont identiques. Quand on fait de la trigonométrie, il est d'usage de rajouter un cercle de rayon 1 centré sur l'origine.

Fig. 2 : Les rayons du soleil arrivent de façon verticale et le capteur fait un angle de 60° avec l'horizontale.



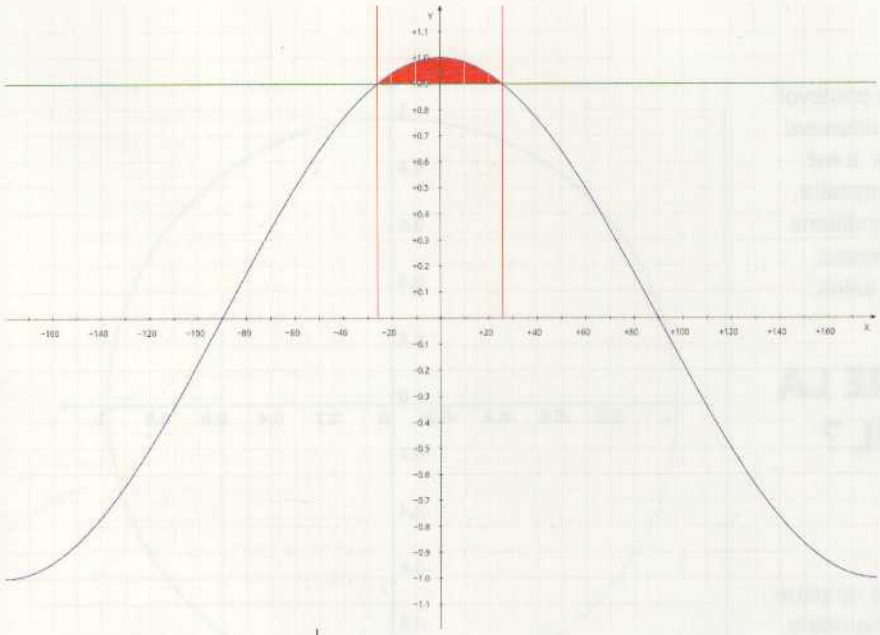


Fig. 3 : Courbe représentative de la fonction cosinus sur l'intervalle [-π ; π]. La zone en rouge correspond à $\Phi_{eff} > 0,9$. Cette zone est comprise entre -25,84° et +25,84°, ce qui est plus que jouable.

2. LA POSITION DU SOLEIL DANS LE CIEL

Pour pouvoir orienter correctement les capteurs, il va nous falloir absolument connaître la position du soleil dans le ciel. On peut le faire de deux manières différentes, par le calcul ou par l'observation. Nous allons détailler ces deux approches et voir leurs points forts ainsi que leurs points faibles.

2.1 Méthode par le calcul

On peut tout à fait calculer la position du soleil dans le ciel. Ce calcul n'est pas difficile, mais tout de même relativement complexe, dans le sens où plein de paramètres rentrent en compte :

- La déclinaison, qui est l'angle entre le plan dans lequel orbite la Terre et le plan de son équateur. Il s'agit en quelque sorte de l'angle « moyen » que formeront les rayons du soleil avec la verticale.
- La durée du jour, qui dépendra bien entendu de la latitude.
- L'angle horaire, qui est dû à la rotation de la Terre sur elle-même.
- ...

Je ne vais pas tout détailler ici, car il existe des documents très complets sur Internet comme par exemple :

https://perso.limsi.fr/bourdin/master/Calculs_astronomiques_simples.pdf

On peut même trouver un code déjà tout fait pour Arduino :

https://www.cerebralmeltdown.com/projects/arduino-sun-position-program/

Pour ma part, vu que je suis curieux et que j'essaye toujours de comprendre les choses avant de passer à la pratique, j'ai suivi le document PDF cité plus haut pour créer mes propres fonctions.

Le point fort de cette méthode est qu'elle ne nécessite pas de matériel supplémentaire. Cependant, elle possède plusieurs points faibles :

- la nécessité de connaître l'heure ;
- la nécessité de connaître sa position ;
- avoir correctement étalonné ses servomoteurs.

J'aurai pu rajouter que les calculs sont des approximations, mais je pense que les erreurs devraient mettre beaucoup de temps à se voir. Par contre, connaître l'heure est un peu plus délicat pour un montage basé sur une carte Arduino, il faut utiliser une horloge RTC, ou une connexion à Internet et récupérer l'heure grâce au protocole NTP, mais cela rajouterait du matériel, ce qui annulerait son point fort... Le fait de devoir connaître la

position rend le dispositif non transportable. De même pour l'étalonnage, la position 0 des servomoteurs doit correspondre au 0 dans les calculs, et ce sur les deux axes. Donc pas possible avec cette méthode de se confectionner un chargeur solaire efficace pour son téléphone à emmener en vacances.

2.2 Méthode par l'observation

Pour chercher la position du soleil dans le ciel, il va nous falloir des composants sensibles à la lumière. Le plus simple est d'utiliser des photorésistances, ce type de composant voit sa résistance varier en fonction de la luminosité. Théoriquement, trois de ces composants seraient suffisants pour pouvoir déterminer la position du soleil. Cependant, en ajouter un n'élèvera vraiment pas beaucoup le prix du dispositif, mais facilitera grandement la gestion au niveau du code.

Il faut maintenant concevoir un support qui va nous permettre de séparer convenablement les photorésistances de manière à pouvoir déterminer la position du soleil par rapport à notre dispositif.

Le point fort de cette méthode est que peu importe la position ou l'heure, le système sera capable de chercher tout seul la position réelle du soleil et de s'orienter correctement. Les points faibles sont :

- ce système nécessite un peu de matériel en plus ;
- qu'en est-il de la précision ?

2.3 Choix de la méthode

Cela va dépendre de l'application finale, dispositif fixe ou portatif, la précision demandée, etc. Les deux méthodes ont leurs avantages et leurs inconvénients. La meilleure solution pour les départager est simplement de les confronter. Je vais mettre en place la méthode par l'observation et comparer la position angulaire des servomoteurs avec les calculs théoriques. On pourra alors trancher et avoir une idée précise des domaines d'application des deux méthodes.

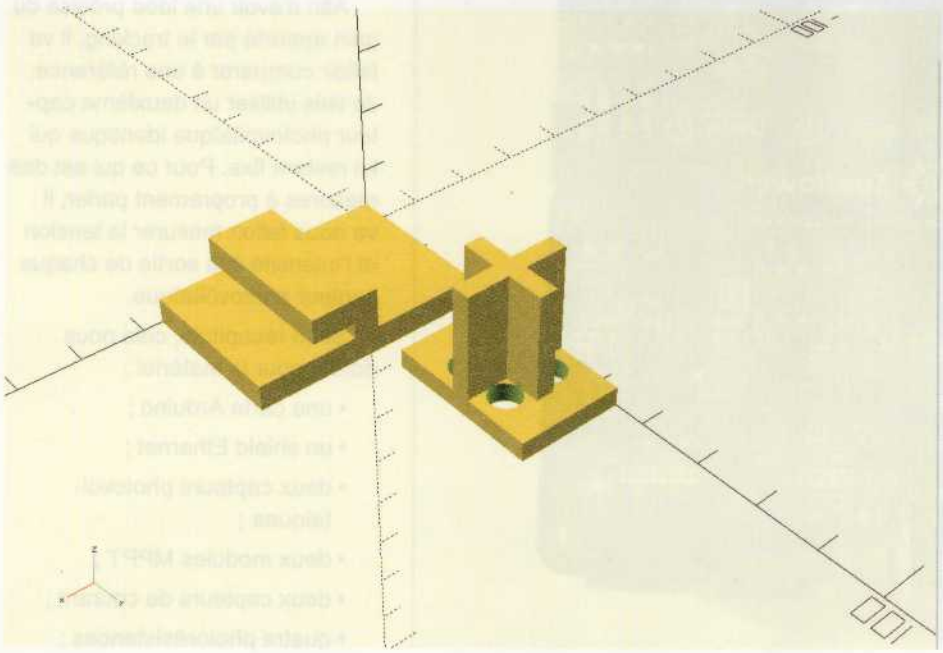


Fig. 4 : La croix permet de bien séparer les photorésistances. En comparant les deux du haut avec les deux du bas et en faisant de même avec droite/gauche, on trouve facilement comment tourner pour faire face au soleil. La seconde pièce est simplement un support pour pouvoir accrocher les photorésistances au capteur.

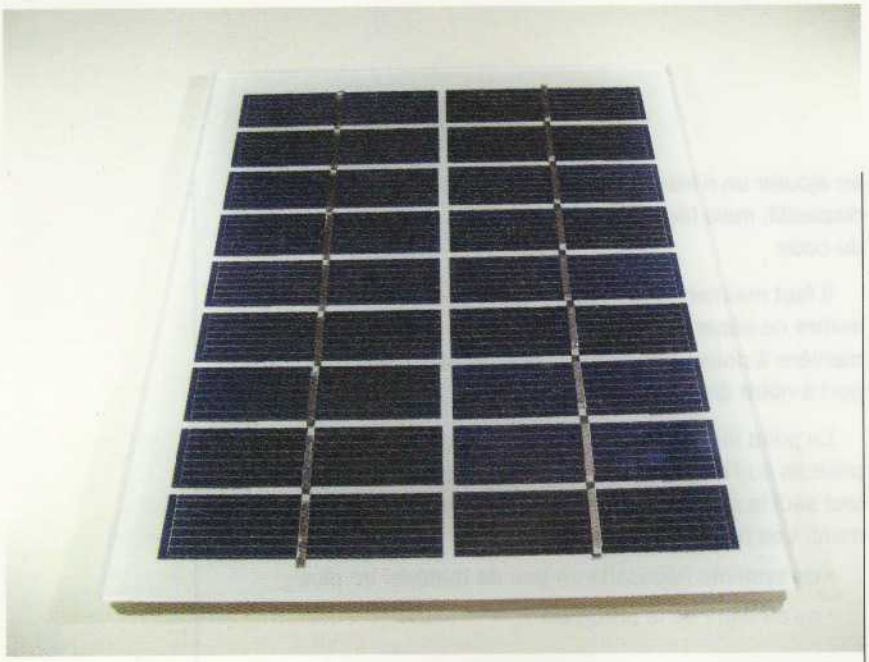
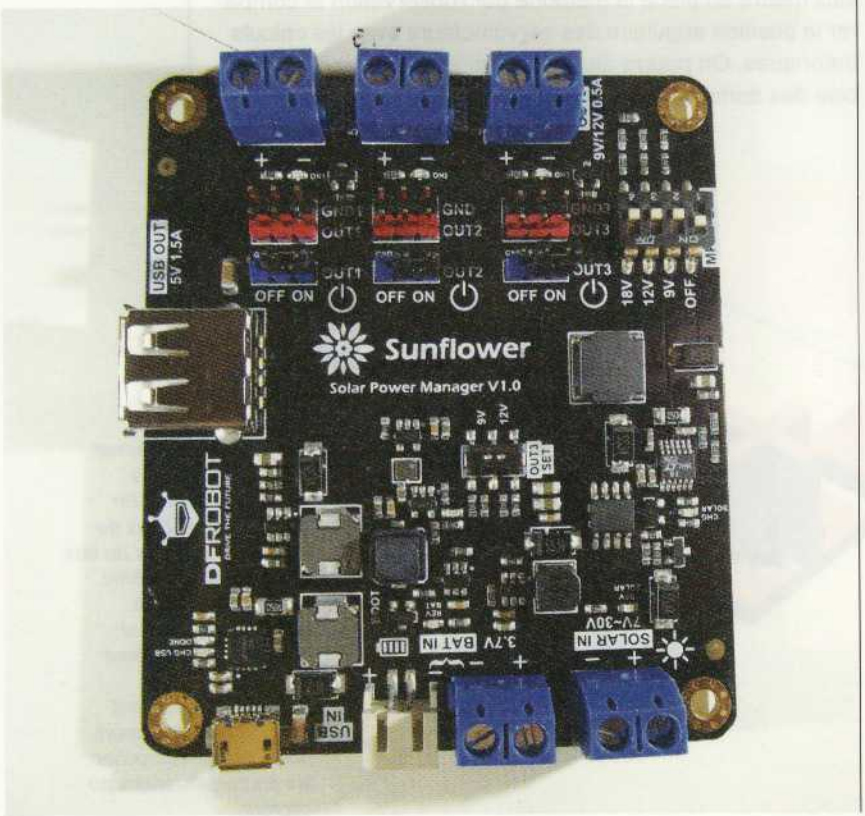


Fig. 5 : Voilà notre petit cobaye.

Fig. 6 : Sunflower de son petit nom, ce module prend en entrée (en bas à droite) un capteur photovoltaïque, charge une batterie (le bornier juste à gauche) et permet d'obtenir en sortie (les 3 borniers du haut) 5V, 3.3V et enfin du 9V ou du 12V au choix.



3. MATÉRIELS ET LOGICIELS NÉCESSAIRES

En tant que bon fainéant, je n'ai pas envie de passer mon temps à prendre des mesures. Je vais donc faire en sorte que les mesures se fassent toutes seules, que la comparaison des différents cas puisse être simple en faisant travailler le plus possible mon ordinateur.

Je vais donc mettre en place une base de données pour récupérer les mesures, ça sera bien rangé et le traitement informatique sera plus simple pour faire des comparaisons. MariaDB est un système de gestion de base de données équivalent à MySQL, mais en libre. Pour que l'Arduino puisse communiquer directement avec la base de données, il va lui falloir une connexion à mon réseau.

Afin d'avoir une idée précise du gain apporté par le tracking, il va falloir comparer à une référence. Je vais utiliser un deuxième capteur photovoltaïque identique qui lui restera fixe. Pour ce qui est des mesures à proprement parler, il va nous falloir mesurer la tension et l'intensité à la sortie de chaque capteur photovoltaïque.

Si on récapitule, cela nous donne pour le matériel :

- une carte Arduino ;
- un shield Ethernet ;
- deux capteurs photovoltaïques ;
- deux modules MPPT ;
- deux capteurs de courant ;
- quatre photorésistances ;

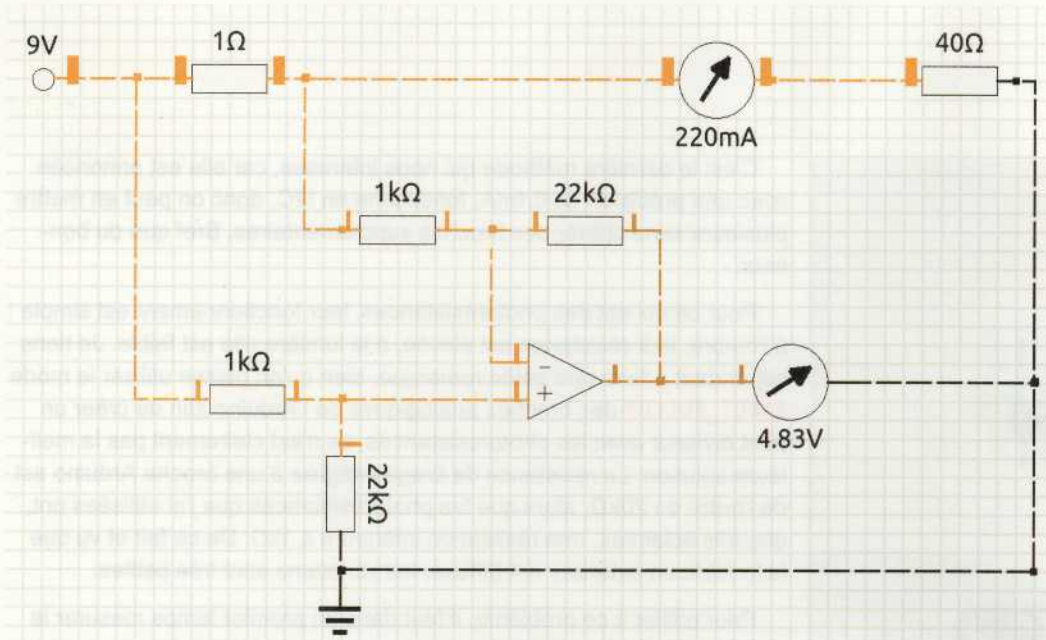


Fig. 7 : Simulation de l'AOP soustracteur dans le logiciel Ktechlab. On peut voir que pour une charge de 40Ω le circuit sera parcouru par un courant de 220mA et que l'AOP fournira une tension de 4,83V. On se trouve donc presque au maximum de précision que peut offrir une carte Arduino.

- deux servomoteurs ;
- une sonde de température DS18B20 (optionnelle) ;
- une DEL de puissance.

Pour un total d'environ 150€.

Pour les logiciels :

- une base de données (MariaDB) ;
- MySQL Connector ;
- Time ;
- DFRobot DFR0198 ;
- Arduino sun position calculations.

3.1 Détail du matériel

Les cartes Arduino ne sont plus à présenter, pareil pour le shield Ethernet, quant à la sonde de température elle a fait l'objet de plusieurs articles dans ce magazine. Attardons-nous un peu plus sur les capteurs photovoltaïques, les modules MPPT, capteurs de courant, ainsi que sur les photorésistances.

Pour ce test, je me suis procuré deux petits capteurs photovoltaïques d'une puissance d'environ 2W. La description du produit annonce 9V et 220mA sous un flux de 1000W/m² à une température de 25°C.

Afin d'obtenir des mesures correctes, il faut être sûr que les capteurs délivrent en permanence leurs puissances maximales. C'est le rôle du module MPPT qui signifie *Maximum Power Point Tracking*, littéralement : suiveur de point de puissance maximale.

La mesure des paramètres (tension et intensité) sera confiée aux modules INA219. À la base, je souhaitais utiliser les amplificateurs opérationnels (AOP) que j'ai sous la main dans un montage soustracteur comme sur le schéma.

Bien que théoriquement valable, la réalisation de ce montage se révèle très problématique. Après de nombreuses recherches infructueuses, c'est un ami qui m'a conseillé de regarder du côté des AOP spécialisés avec la mention « High side ». J'ai alors trouvé ce petit module, qui à première vue ne paie pas de mine, mais qui va grandement me simplifier la vie. La bibliothèque écrite par Adafruit est simple et efficace. C'est écrit sous forme d'objet et ils ont prévu une méthode pour chaque chose, mesure de la tension, de l'intensité, même une méthode qui retourne directement la puissance et la possibilité de changer de calibre. Pour cette dernière fonctionnalité, on a le choix entre :

```
void setCalibration_32V_2A(void);
void setCalibration_32V_1A(void);
void setCalibration_16V_400mA(void);
```

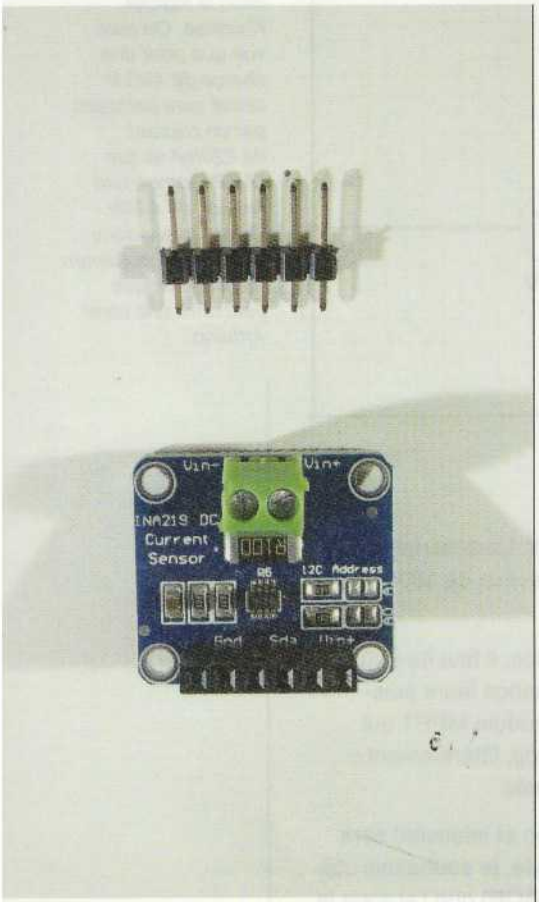


Fig. 8 : Malgré sa petite taille, ce module équipé d'un INA219 correspond parfaitement à ce dont j'ai besoin pour réaliser mes mesures.

C'est la dernière méthode qui nous intéresse, car elle est annoncée avec une précision de 0,1mA, fonctionne en I2C, donc on peut en mettre plusieurs sans utiliser des broches supplémentaires. Bref que du bonheur.

Pour ce qui est des photorésistances, leur fonctionnement est simple : à l'ombre, leur résistance est élevée, à la lumière elle est faible. Je tiens cependant à faire une petite remarque, bien qu'on puisse utiliser le mode `INPUT_PULLUP` des broches analogiques de l'Arduino afin de créer un pont diviseur avec les photorésistances, ce n'est clairement pas la meilleure solution. La résistance de tirage intégrée à une broche Arduino est de l'ordre de 20kΩ, alors que les photorésistances que j'ai utilisées ont, une fois éclairées, une résistance inférieure à 1kΩ. De ce fait et vu que la lumière en extérieur est diffuse, les variations sont très petites.

Pour pallier à ce problème, il faut dans un premier temps mesurer la valeur des photorésistances dans les conditions d'utilisations, puis ajouter des résistances de tirage externe de valeurs assez proches. Pour bien comprendre, je vous invite à lire la page Wikipédia sur les diviseurs de tension : **https://fr.wikipedia.org/wiki/Diviseur_de_tension**. Sur cette page, vous pourrez voir que si les résistances ont la même valeur, alors la tension de sortie est égale à la moitié de la tension d'entrée. On se trouve donc au milieu de l'échelle de valeurs et donc on profite du maximum de précision.

Bien sûr tout ça ne sert à rien si l'énergie produite n'est pas évacuée, il faut donc placer une charge à la sortie des MPPT. Pour cela, j'ai choisi d'utiliser une DEL de puissance. L'intérêt d'utiliser un tel composant est le fait que la courbe caractéristique d'une DEL est une exponentielle. De ce fait, même une légère augmentation de la tension de sortie engendrera une énorme augmentation de l'intensité et donc de la puissance évacuée. Le MPPT se trouvera donc face à un véritable mur et devra fournir toute la puissance à sa disposition pour essayer de le franchir. On s'assure donc qu'il drainera le maximum de puissance du capteur.

3.2 Détail des logiciels

Maintenant que l'on sait quel est le matériel qui sera utilisé et son fonctionnement on peut se pencher sur la partie logicielle pour gérer tout ça. Il n'y a rien de bien sorcier, mais il faut tout de même faire attention à certains détails. Voyons la librairie pour le module INA219 écrite par Adafruit.

On peut y trouver les méthodes suivantes :

```
float getBusVoltage_V(void);
float getShuntVoltage_mV(void);
float getCurrent_mA(void);
float getPower_mW(void);
```

Dans l'ordre, elles correspondent à :

- la tension à la sortie de la résistance de shunt ;
- la tension aux bornes de la résistance ;
- l'intensité traversant la résistance ;
- la puissance de la source branchée sur le module.

Toutes ces méthodes retournent des **FLOAT**.

Les fonctions calculant la position du soleil retourneront également des **FLOAT**. Par contre, la position des servomoteurs est quant à elle des entiers, donc de type **INT**.

Concernant la mesure de la température, la valeur sera de type **FLOAT**. On peut maintenant attaquer la base de données.

Ayant un serveur qui tourne déjà H24, je ne vais pas allouer un ordinateur exprès pour ce montage. Grâce à Docker, je peux créer un conteneur spécifiquement pour ça. Il ne contiendra qu'une base de données et rien d'autre.

Vu que j'ai déjà une image toute prête pour les bases de données (**ubuntu:bdd**), il me suffit de copier les dossiers concernant la base de données dans un dossier extérieur au conteneur pour pouvoir conserver les données (partie volumes) et d'ajouter ce petit paragraphe dans le fichier **docker-compose.yml** :

```
########## BASE DE DONNEES ARDUINO ##########
 arduino_bdd:
    image: ubuntu:bdd
    container_name: arduino_bdd
    volumes:
    - "/etc/localtime:/etc/localtime:ro"
    - "/mnt/donnees_serveur/docker/conf_conteneurs/arduino_bdd/var/lib/
mysql:/var/lib/mysql:rw"
    - "/mnt/donnees_serveur/docker/conf_conteneurs/arduino_bdd/etc/mysql:/
etc/mysql:ro"
    environment:
    - TERM=xterm
    ports:
    - "13377:3306"
```

Pour créer et démarrer le conteneur, il me suffit d'exécuter la commande :

```
# docker-compose up -d
```

Et voilà, ça roule. Vérifions ça en nous connectant au conteneur puis à la base de données qu'il contient :

```
# docker exec -ti arduino_bdd bash
root@ef8e00b08c83:/# mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 32
```

```
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Et nous voilà dans la base de données. J'en profite pour créer une base de données **arduino** :

```
MariaDB [(none)]> CREATE DATABASE arduino;
Query OK, 1 row affected (0.00 sec)
```

Ainsi qu'un utilisateur :

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON arduino.* TO 'arduino'@'%'
IDENTIFIED BY 'mot_de_passe' ;
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> FLUSH PRIVILEGES ;
Query OK, 0 rows affected (0.00 sec)
```

Maintenant penchons-nous un peu sur les champs que doit contenir la table. Bien que le module INA219 est capable de nous donner directement la puissance, je pense qu'il serait intéressant de faire des mesures complètes : tension, intensité et puissance. De cette manière, on est sûr de ne rien rater. On va également noter les positions des servomoteurs ainsi que la position calculée du soleil, histoire de pouvoir comparer les deux. Pour finir, on enregistrera également la température.

On notera les paramètres électriques tension, intensité et puissance respectivement : U, I et P. Pour le premier capteur, ça sera donc : **U1**, **I1** et **P1**, pour le deuxième : **U2**, **I2**, **P2** et la sortie du module MPPT (histoire de pouvoir calculer le rendement) sera notée **U_s**, **I_s** et **P_s** (oui avec un underscore, car Is est un nom réservé). L'orientation des deux servomoteurs sera notée **thêta** et **phi** (voir **https://fr.wikipedia.org/wiki/Coordonn%C3%A9es_sph%C3%A9riques**). La position calculée du soleil dans le ciel : **altitude** et **azimuth**. La température sera tout simplement notée : **temperature**. Pour finir, la date et l'heure de la mesure, cette dernière colonne sera générée directement par la base de données et sera du type **DATETIME**, que je nommerai **date_et_heure**. Il faut éviter de nommer cette colonne simplement 'date', car il s'agit d'un nom réservé.

Maintenant qu'on a la liste complète des paramètres, leur notation, ainsi que le type des variables, on va pouvoir créer la table. Voilà ce que ça nous donne :

```
CREATE TABLE mesure_capteur_solaire (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    U1 FLOAT, I1 FLOAT, P1 FLOAT,
    U2 FLOAT, I2 FLOAT, P2 FLOAT,
    U_s FLOAT, I_s FLOAT, P_s FLOAT,
    theta SMALLINT, phi SMALLINT,
    altitude FLOAT, azimuth FLOAT,
    temperature FLOAT,
    date_et_heure DATETIME,
    PRIMARY KEY (id)
);
```

Pour être le plus complet possible, il va falloir faire des mesures tout au long d'une journée. C'est pour cela que le champ `id`, qui va permettre d'identifier chaque ligne, est de type `INT UNSIGNED`, ça permettra de faire plus de 4 milliards de mesures avant d'avoir un problème, il y a donc de la marge.

3.3 Le code Arduino

Première chose, faisons la liste des tâches que devra accomplir l'Arduino :

- connexion à MariaDB (**https://www.arduinolibraries.info/libraries/my-sql-connector-arduino**) ;
- mesure des tensions, des intensités et des puissances ;
- mesure de la température (**http://www.dfrobot.com/image/data/DFR0198/DFRobot%20DFR0198.zip**) ;
- récupération de la date et de l'heure (**https://www.arduino.cc/en/Tutorial/UdpNtpClient**) ;
- calcul de la position du soleil (**https://www.cerebralmeltdown.com/projects/arduino-sun-position-program/**) ;
- orientation du capteur solaire.

Il va nous falloir « mixer » tout ça pour obtenir un code nous permettant d'atteindre tous ces objectifs. Pour que ça soit plus simple et plus lisible, on va créer différents fichiers, un par tâche, qui contiendront les fonctions nécessaires à l'accomplissement d'une tâche. Puis créer un fichier principal qui va appeler ces différentes fonctions. Tout mettre dans un seul fichier finirait par être une prise de tête est n'est clairement pas une bonne solution.

3.3.1 Gestion des requêtes pour la base de données

Commençons par la connexion à MariaDB, l'exemple `complex_insert.ino`, se trouvant dans la bibliothèque en lien, nous servira de base. Le principe est simple, il faut utiliser une chaîne de caractères correctement formatée. L'exemple donné est :

```
char INSERT_DATA[] = "INSERT INTO test_arduino.hello_sensor (message,
sensor_num, value) VALUES ('%s',%d,%s)";
```

Puis il faut écrire les valeurs dans cette chaîne grâce à la fonction `sprintf` :

```
sprintf(query, INSERT_DATA, "test sensor", 24, temperature);
```

Mais on va se retrouver face à un petit problème... La fonction `sprintf` de l'Arduino ne gère pas les `FLOAT` ... Si vous essayez :

```
float val_float = 42.1337;
char affichage[8], base[] = "%f";
sprintf(affichage, base, val_float);
Serial.println(affichage);
```

Vous obtiendrez la réponse suivante :

```
?
```

Il va donc falloir utiliser une fonction en plus qui va simplement convertir une variable de type **FLOAT** en une variable de type **CHAR**. C'est le rôle de la fonction **dtostrf** :

```
char * dtostrf(
        double __val,
        signed char __width,
        unsigned char __prec,
        char * __s)
```

Le premier paramètre est la valeur à convertir, le second détermine la taille minimale de la partie entière (signe et '.' compris), le troisième est le nombre de décimales qui seront affichées (arrondi si nécessaire) et le dernier est la chaîne de caractères dans laquelle vous allez récupérer votre nombre.

On peut maintenant utiliser le caractère spécial **%s** pour écrire la base de notre requête. Ce qui nous donne :

```
char base_requete[] = "INSERT INTO arduino.mesure_capteur_solaire
VALUES (NULL,%s,%s,%s,%s,%s,%s,%s,%s,%s,%d,%d,%s,%s,%s, NOW());";
```

Le premier paramètre est passé à **NULL** pour que ce soit la base de données elle-même qui gère le champ **id**, elle l'incrémentera automatiquement. Il en va de même pour le dernier paramètre, il est passé à **NOW()**, pour que la base de données inscrive elle-même la date et l'heure. Une fois les mesures et les conversions effectuées, on crée la requête en utilisant la fonction **sprintf**. Et enfin, on envoie la requête :

```
cur_mem->execute(requete) ;
```

3.3.2 Mesure des tensions, des intensités et des puissances

Cette partie est la plus facile, la bibliothèque écrite par Adafruit nous mâche bien le travail. Il faut juste cependant faire attention à un petit détail. La méthode :

```
float getBusVoltage_V(void);
```

Retourne la tension à la sortie de la résistance de shunt, il faut donc penser à additionner cette valeur à celle retournée par :

```
float getShuntVoltage_mV(void);
```

qui est la tension aux bornes de la résistance pour obtenir la tension de la source. Les autres méthodes pour l'intensité et la puissance retournent directement les bonnes valeurs.

3.3.3 Mesure de la température

Pas de difficulté majeure ici, le code fourni par DFrobot pour sa sonde de température est très simple. Il faut juste mettre le code en forme pour qu'il puisse être utilisé depuis un autre fichier.

3.3.4 Récupération de la date

Pour la date et l'heure, on va utiliser une petite astuce pour ne pas être obligé d'envoyer une requête à chaque fois qu'on veut connaître l'heure. On va utiliser trois variables :

```
unsigned long temps = 0, timestamp = 0, temps_execution = millis();
```

La première sera le temps actuel. La seconde, le timestamp récupéré lors de l'envoi de la requête NTP qui est pour rappel le nombre de secondes écoulées depuis le 01/01/1970. Enfin, la troisième, le nombre de millisecondes écoulées depuis le démarrage de la carte Arduino, elle nous servira de repère pour savoir depuis combien de temps on a lancé la requête. On peut donc actualiser le temps en faisant :

```
temps = timestamp + (millis() - temps_execution)/1000;
```

Sur une longue durée, on risque tout de même de constater un « glissement » du temps, c'est pour cela qu'il faut rajouter une condition afin de refaire une requête de temps en temps. Une toute les 6h sera largement suffisant.

3.3.5 Calcul de la position du soleil

Le code du calcul de la position du soleil doit être modifié pour prendre un timestamp en guise de paramètre. C'est une des raisons qui m'a poussé à écrire mes propres fonctions. Pour ce faire, l'appel à la bibliothèque <Time.h> sera d'une grande aide.

3.3.6 Orientation du capteur solaire

Là on entre dans le genre de cas qui est théoriquement très simple, mais qui, dans la pratique, se révèle bien problématique.

Dans la théorie, cela donne : on a quatre photorésistances identiques, bien séparées. On compare les valeurs des deux photorésistances de gauche avec les deux de droite. Si les valeurs de droite sont plus élevées, cela signifie qu'elles sont à l'ombre et qu'il faut tourner à gauche et inversement. Même chose concernant les valeurs haut/bas.

Dans la pratique : on a quatre photorésistances presque identiques, couplées à des résistances de tirages elles aussi presque identiques, dans un environnement qui diffuse la lumière, les ombres ne sont que partielles, il y a des reflets, les valeurs fluctuent, etc. Ce qui fait qu'on ne peut pas avoir exactement les mêmes valeurs pour les quatre photorésistances et même si c'était le cas on ne serait pas sûr d'être vraiment en face.

Quand on mesure une valeur analogique, il y a toujours du bruit, des fluctuations. Heureusement, ces fluctuations se font autour de la valeur moyenne. On va donc faire une somme de mesures afin de limiter l'impact du bruit sur la valeur.

```
for(int i = 0 ; i < nbr_mesure ; i++)
{
 somme_gauche += analogRead(haut_gauche) + analogRead(bas_gauche);
 somme_droite += analogRead(haut_droite) + analogRead(bas_droite);
 somme_haut += analogRead(haut_gauche) + analogRead(haut_droite);
 somme_bas += analogRead(bas_gauche) + analogRead(bas_droite);
}
```

Ici, pas besoin de s'embêter à faire une vraie moyenne, puisque ce n'est pas réellement ce qui nous intéresse. L'information pertinente est de savoir si un côté retourne en moyenne des valeurs plus grandes que l'autre.

Pour palier aux défauts du capteur, des composants pas tout à fait identiques, des reflets, etc., on utilise deux variables :

```
long correction_haut_bas = -90 * nbr_mesure, correction_gauche_
droite = -130 * nbr_mesure;
```

Notez bien que plus on fera de mesures, plus les défauts du capteur se feront sentir, il faut donc que les corrections soient proportionnelles au nombre de mesures. Mais même avec ces précautions, on ne pourra jamais avoir exactement les mêmes valeurs. On va donc utiliser une variable seuil.

```
delta_gauche_droite = somme_gauche - somme_droite + correction_
gauche_droite;
delta_haut_bas = somme_haut - somme_bas + correction_haut_bas;

if((delta_gauche_droite > seuil) || (delta_gauche_droite < -seuil))
{
 angle_theta += facteur * delta_gauche_droite;
}

if((delta_haut_bas > seuil) || (delta_haut_bas < -seuil))
{
 angle_phi -= facteur * delta_haut_bas;
}
```

L'utilisation des valeurs delta pour incrémenter l'angle permet de se tourner plus vite vers le soleil. Quand le capteur n'est pas orienté vers le soleil, les deltas sont grands, il va donc tourner rapidement. Mais au fur et à mesure qu'il se rapproche de la bonne orientation, la valeur de delta diminue et les variations angulaires se font de plus en plus petites. Ce qui fait qu'il se positionne précisément. Cependant, les deltas sont issus des sommes qui sont proportionnelles au nombre de mesures. De ce fait, plus on fera de mesures, plus les valeurs des deltas seront grandes, d'où la présence de la variable facteur qui elle est inversement proportionnelle au nombre de mesures. On obtient donc un produit du type : *valeur1 * nbr_mesure * valeur2 / nbr_mesure*. Ce qui nous assure d'avoir une variation angulaire indépendante du nombre de mesures.

Avec toutes ces précautions, on arrive enfin à un résultat intéressant. Le capteur s'aligne vite, de manière précise et oscille très peu autour de cette position.

Nous voilà à présent parés pour passer à la construction et aux tests.

4. CONSTRUCTION DU PROTOTYPE

On peut voir sur Internet des structures pour servomoteurs appelées *pan/tilt*. Par exemple : **https://www.thingiverse.com/thing:708819**. Elles sont très pratiques pour des petites applications, cependant le capteur solaire sera en extérieur, avec du vent. Il va s'en dire que ça risque

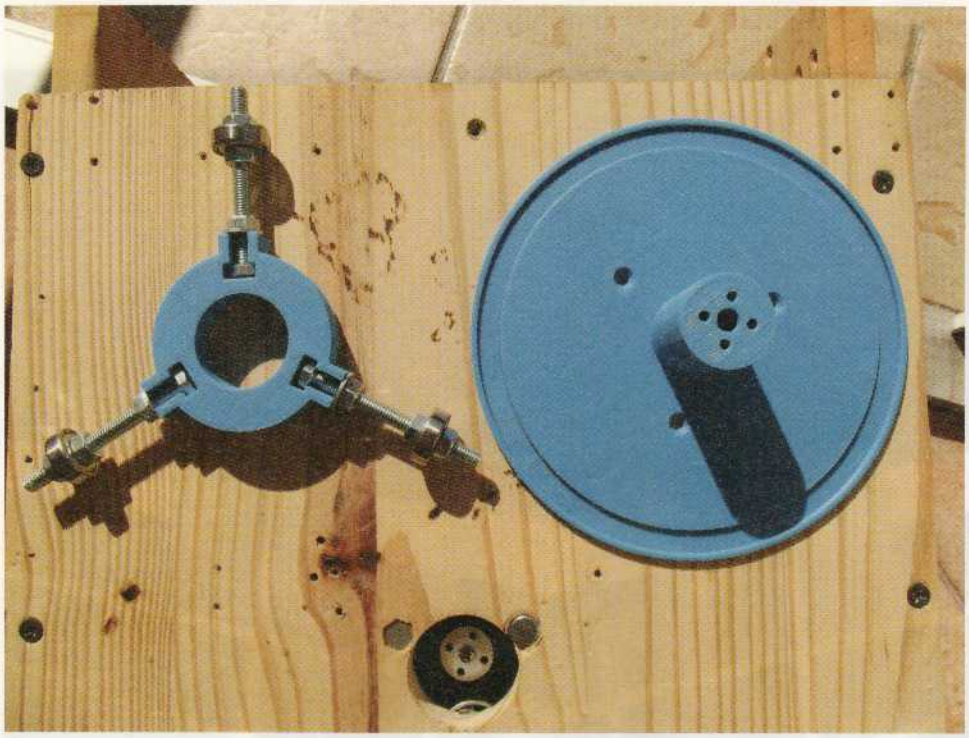


Fig. 9 : Un plateau large avec un axe et une gorge dans laquelle vient s'emboîter des roulements permettent de supporter de gros efforts. On peut également voir le servomoteur à travers le trou dans la plaque de bois.

de bouger pas mal, voire même de casser. Il faut donc une structure plus résistante. Pour ça, je me suis inspiré du mécanisme qui fait tourner les plateaux dans un four micro-ondes (voir figure 9).

Voilà pour la rotation suivant l'angle thêta. Pour l'angle phi, il nous faut un autre servo-moteur suivant un axe perpendiculaire au premier (voir figure 10).

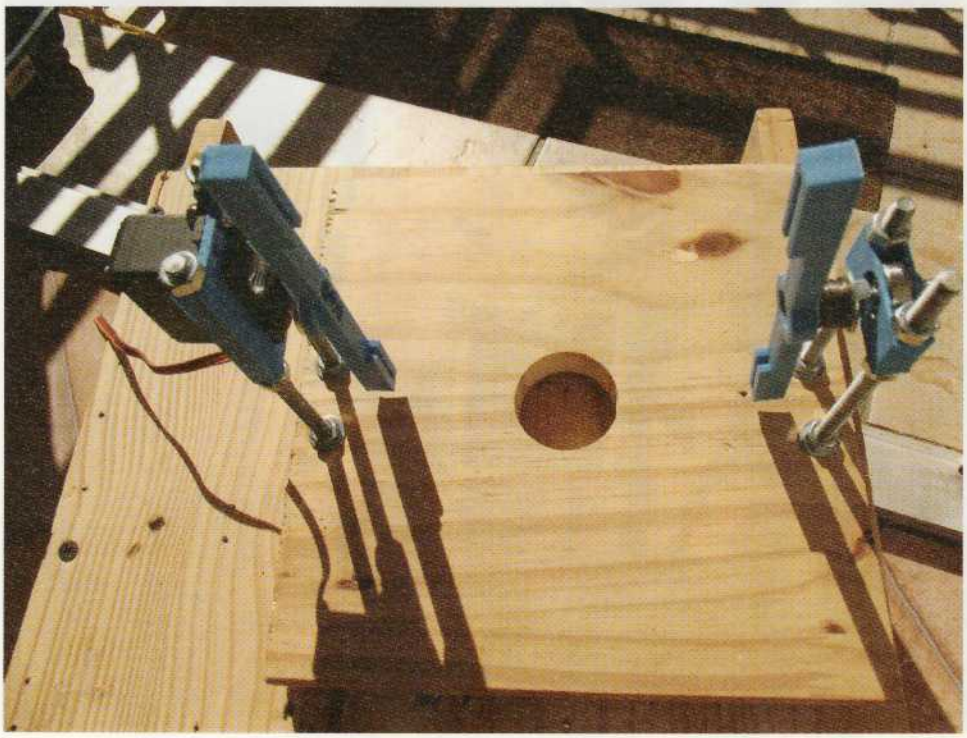


Fig. 10 : Voilà le deuxième servomoteur, monté sur des tiges filetées et muni d'une « barquette » pour accueillir le capteur. Ce dernier sera serré grâce à la barquette de droite qui est équipée d'un ressort.

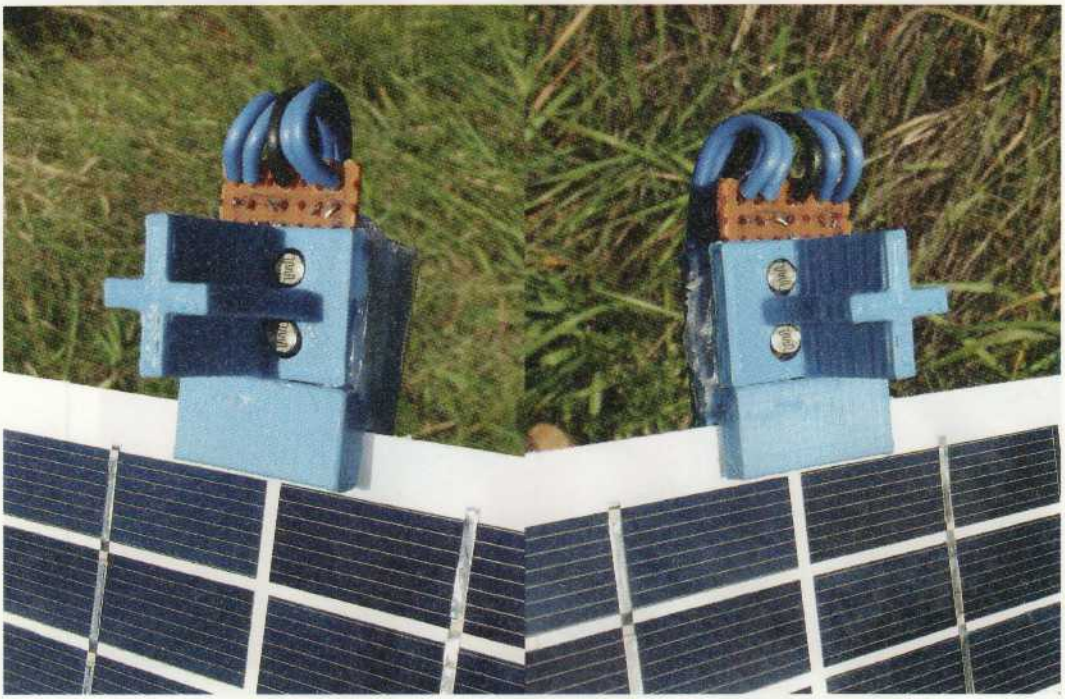


Fig. 11 : Les photorésistances en action. L'orientation n'est pas parfaite, mais l'ombre qu'on peut voir sur le côté droit ne fait que 1 à 2mm.

J'ai également imprimé quelques petits supports pour toutes les cartes électroniques, pour pouvoir les coller dans une boite en plastique à l'abri de la pluie et de l'humidité.

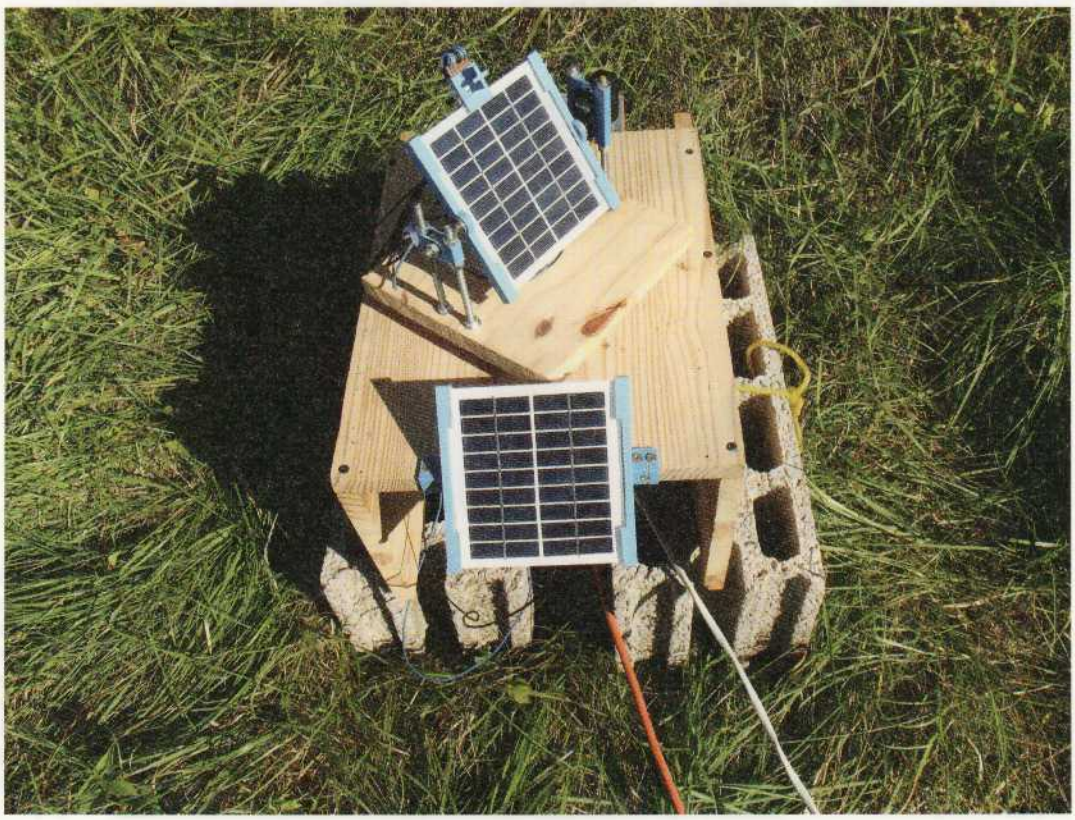


Fig. 12 : Maintenant tout est prêt, c'est à son tour de travailler.

5. RÉCUPÉRATION ET TRAITEMENT DES DONNÉES

Comme dit précédemment, en tant que fainéant je vais faire bosser mon ordinateur à ma place. Après quelques recherches, je suis tombé sur : **https://www.quennec.fr/trucs-astuces/syst%C3%A8mes/gnulinux/utilisation/bash-ex%C3%A9cuter-une-requ%C3%Aate-mysql-et-exploiter-le-r%C3%A9sultat**.

On va donc créer un script qui questionne la base de données, place les résultats dans un fichier et appelle Gnuplot pour nous afficher un joli graphique. Le script est très simple puisqu'il tient sur deux lignes :

```
echo "SELECT * FROM arduino.mesure_capteur_solaire;" | mysql -u arduino
-pmot_de_passe -h 192.168.1.15 -P 13377 > donnees_arduino.txt

gnuplot -e "set terminal png size 1024,748 ; set output 'test.png'; set
xdata time ; set timefmt '%Y-%m-%d %H:%M:%S'; set format x '%d %H:%M' ;
plot 'donnees_arduino.txt' using 13:1"
```

La première ligne est assez simple à comprendre. La deuxième quant à elle est un peu plus intéressante. L'option `-e` permet de créer une liste de commandes à exécuter par Gnuplot. Il suffit alors de les placer les unes à la suite des autres en les séparant par un `;`.

Dans l'ordre, on choisit le format de sortie ainsi que la taille de l'image. Puis le nom du fichier. On précise que l'axe des X contiendra des données temporelles. Puis on en détermine le format (année-mois-jour heure:minute:seconde). Choix du format de l'étiquette sur l'axe des X. Et enfin, pour finir, le tracé du graphique à proprement parler en précisant les colonnes à utiliser.

Voilà ce que ça donne :

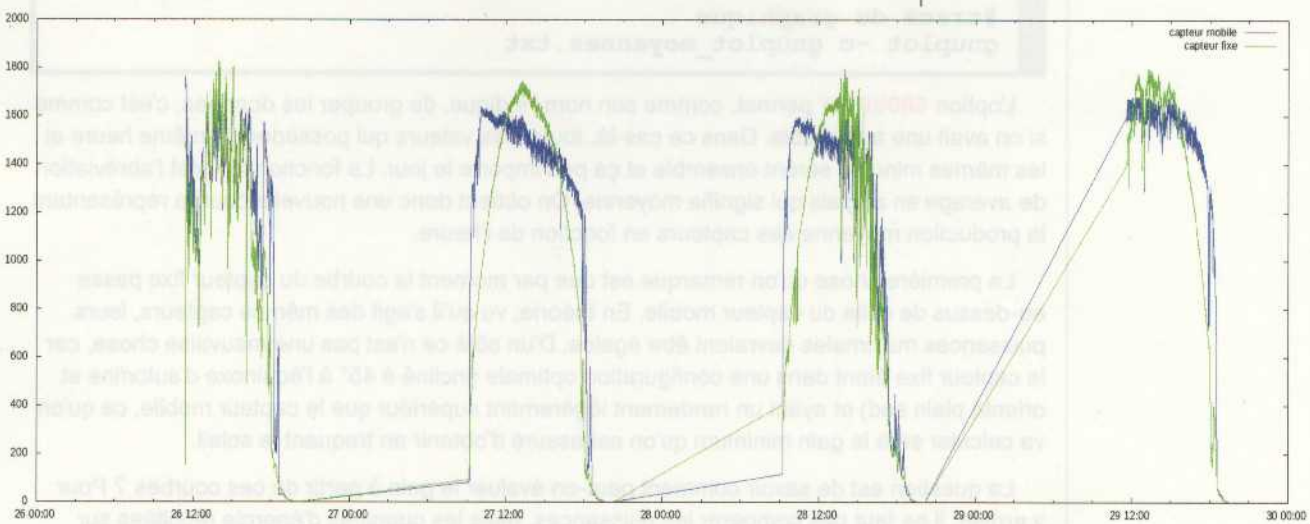


Fig. 13 : Voici les données brutes. La courbe bleue représente la puissance instantanée du capteur mobile, la courbe verte la puissance instantanée du capteur fixe. Lors du dernier jour de mesure, il y a eu un petit raté, d'où l'allure bizarre de la courbe.

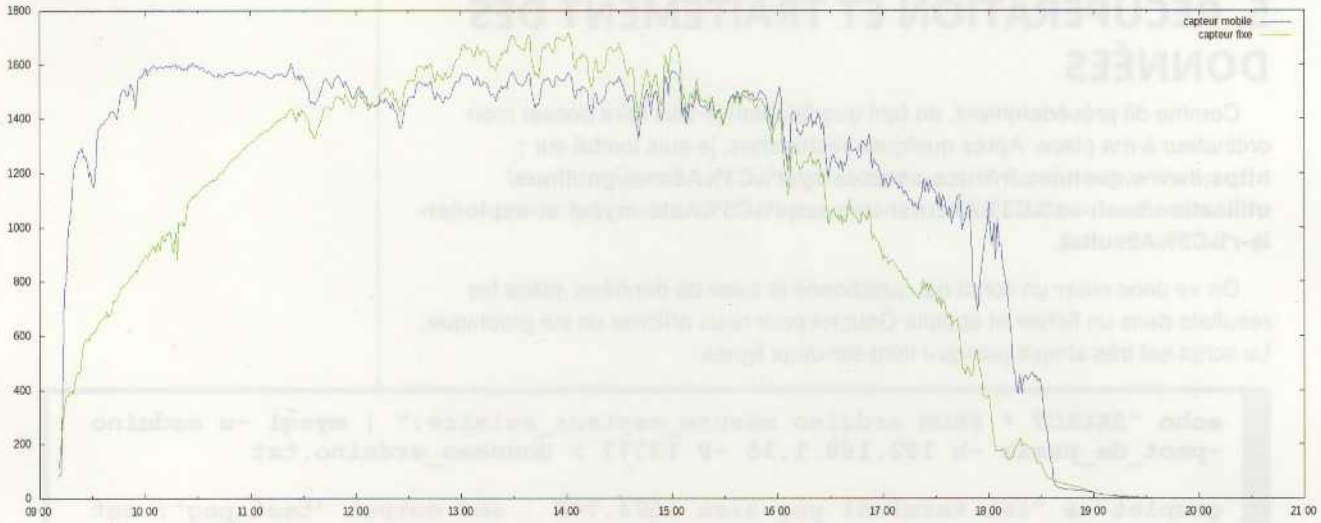


Fig. 14 : Avec les mêmes couleurs, voici la puissance instantanée moyenne des deux capteurs.

Pour rendre le script plus lisible, on peut utiliser l'option `-c` de `gnuplot`, qui permet de passer un fichier contenant toutes les commandes.

C'est sympa d'avoir une vue d'ensemble, mais l'intérêt de prendre autant de mesures est quand même de pouvoir faire de belles moyennes. Mais pour y arriver, il va falloir utiliser une autre requête.

```
#requete mysql
echo "SELECT DATE_FORMAT(date_et_heure, '%T') AS heure,
AVG(P1), AVG(P2) FROM arduino.mesure_capteur_solaire GROUP
BY HOUR(date_et_heure), MINUTE(date_et_heure);" \
| mysql -u arduino -p$mot_de_passe -h 192.168.1.15 -P 13377
> donnees_moyennes.txt

#tracé du graphique
gnuplot -c gnuplot_moyennes.txt
```

L'option `GROUP BY` permet, comme son nom l'indique, de grouper les données, c'est comme si on avait une autre table. Dans ce cas-là, toutes les valeurs qui possèdent la même heure et les mêmes minutes seront ensemble et ça peu importe le jour. La fonction `AVG` est l'abréviation de *average* en anglais qui signifie moyenne. On obtient donc une nouvelle courbe représentant la production moyenne des capteurs en fonction de l'heure.

La première chose qu'on remarque est que par moment la courbe du capteur fixe passe au-dessus de celle du capteur mobile. En théorie, vu qu'il s'agit des mêmes capteurs, leurs puissances maximales devraient être égales. D'un côté ce n'est pas une mauvaise chose, car le capteur fixe étant dans une configuration optimale (incliné à 45° à l'équinoxe d'automne et orienté plein sud) et ayant un rendement légèrement supérieur que le capteur mobile, ce qu'on va calculer sera le gain minimum qu'on est assuré d'obtenir en traquant le soleil.

La question est de savoir comment peut-on évaluer le gain à partir de ces courbes ? Pour y arriver, il ne faut pas comparer les puissances, mais les quantités d'énergie récoltées sur

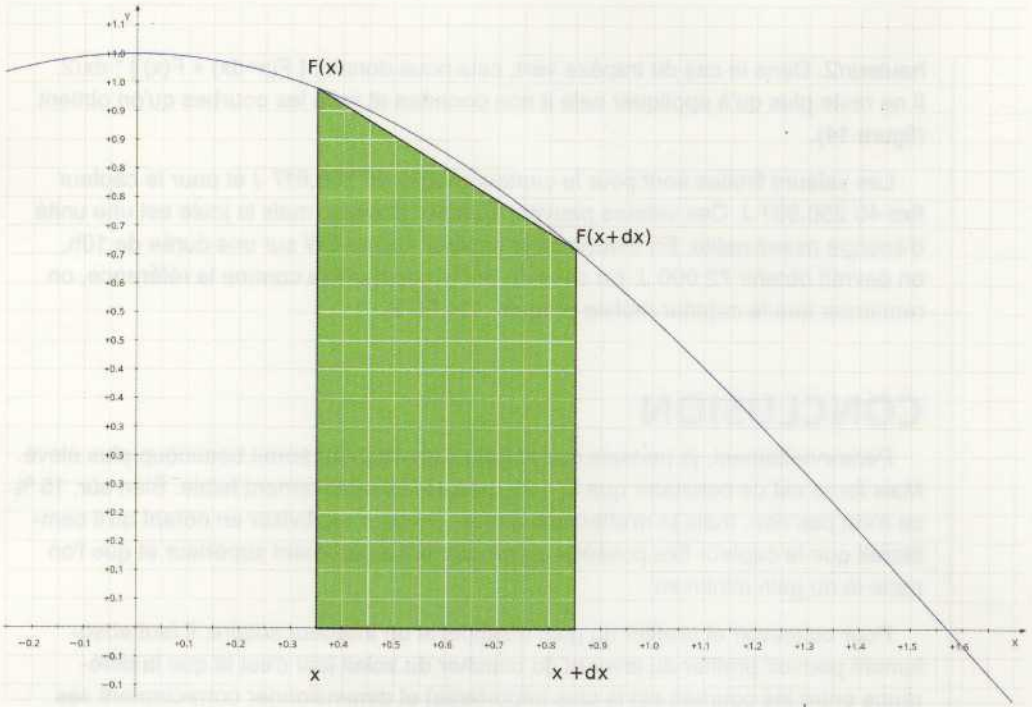


Fig. 15 : La méthode des trapèzes fait l'approximation que la surface sous la courbe est la somme des surfaces de trapèzes comme celui en vert. Plus le paramètre dx sera petit, plus les trapèzes seront proches de la courbe et l'intégrale sera précise.

la journée. Or par définition, une puissance est le transfert d'une certaine quantité d'énergie (en joules) par seconde (1W = 1J/s). Mais vu que la puissance varie tout au long de la journée, on ne peut pas simplement faire une règle de trois. Mathématiquement parlant, il faut faire ce qu'on appelle une intégrale. Cela revient à calculer la surface sous la courbe. Pour ce faire, on va utiliser la méthode des trapèzes. Pour bien comprendre, il faut regarder la figure 13.

La formule pour calculer la surface d'un trapèze est : (petite base + grande base) *

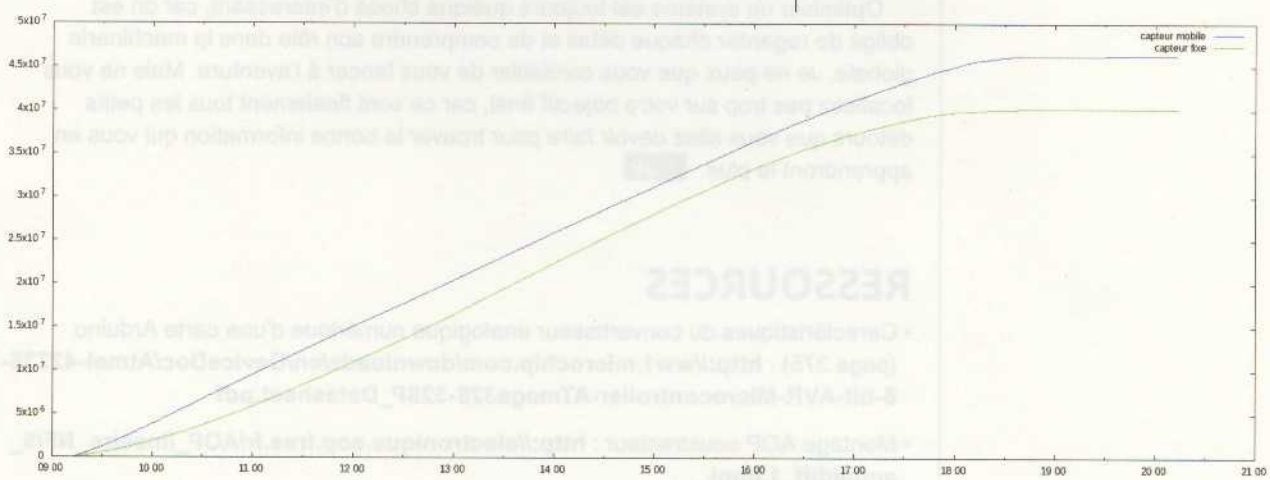


Fig. 16 : L'axe des ordonnées est en millijoules (mJ). Ici le gain est clairement visible, la courbe bleue est toujours au-dessus de la courbe verte. On remarque un écart de tendance surtout aux extrémités.

hauteur/2. Dans le cas du trapèze vert, cela nous donne : [F(x+dx) + F(x)] * dx/2. Il ne reste plus qu'à appliquer cela à nos données et voilà les courbes qu'on obtient (figure 14).

Les valeurs finales sont pour le capteur mobile 46 268,617 J et pour le capteur fixe 40 250,961 J. Ces valeurs peuvent sembler élevées, mais le joule est une unité d'énergie assez petite. En théorie, si le capteur délivre 2W sur une durée de 10h, on devrait obtenir 72 000 J. En considérant le capteur fixe comme la référence, on remarque que le capteur mobile produit : 114,95 %.

CONCLUSION

Personnellement, je pensais que le gain d'un traqueur serait beaucoup plus élevé. Mais force est de constater que le gain mesuré est relativement faible. Bien sûr, 15 % ce n'est pas rien, mais je m'attendais à plus. On peut relativiser en notant qu'il semblerait que le capteur fixe possède un rendement légèrement supérieur et que l'on parle là du gain minimum.

Pour concevoir et profiter du gain qu'apporte un traqueur solaire, il faut absolument pouvoir profiter du lever et du coucher du soleil (car c'est là que la différence entre les courbes est la plus importante) et dimensionner correctement ses moteurs. Les servomoteurs sont finalement à proscrire pour cette utilisation, car ils doivent consommer du courant pour maintenir leur position. Il est préférable d'utiliser un moteur à courant continu couplé à un engrenage du type roue + vis sans fin. Concernant les méthodes de positionnement (par l'observation et par le calcul), elles peuvent être utilisées conjointement pour déterminer la position du soleil. L'observation pour piloter les moteurs et le calcul pour déterminer l'écart entre la position visée et la position réelle. De ce fait, on peut déclencher la rotation du capteur quand l'écart dépasse un certain seuil. Les moteurs n'utilisent donc pas du courant en permanence. De plus, une fois la nuit tombée, on peut calculer la position du soleil lors du lever et orienter tout de suite le capteur pour profiter des premières lueurs du jour.

Optimiser un système est toujours quelque chose d'intéressant, car on est obligé de regarder chaque détail et de comprendre son rôle dans la machinerie globale. Je ne peux que vous conseiller de vous lancer à l'aventure. Mais ne vous focalisez pas trop sur votre objectif final, car ce sont finalement tous les petits détours que vous allez devoir faire pour trouver la bonne information qui vous en apprendront le plus. JP

RESSOURCES

- Caractéristiques du convertisseur analogique numérique d'une carte Arduino (page 375) : **http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf**
- Montage AOP soustracteur : **http://electronique.aop.free.fr/AOP_lineaire_NF/5_amplidiff_1.html**

LEROY MERLIN PRÉSENTE

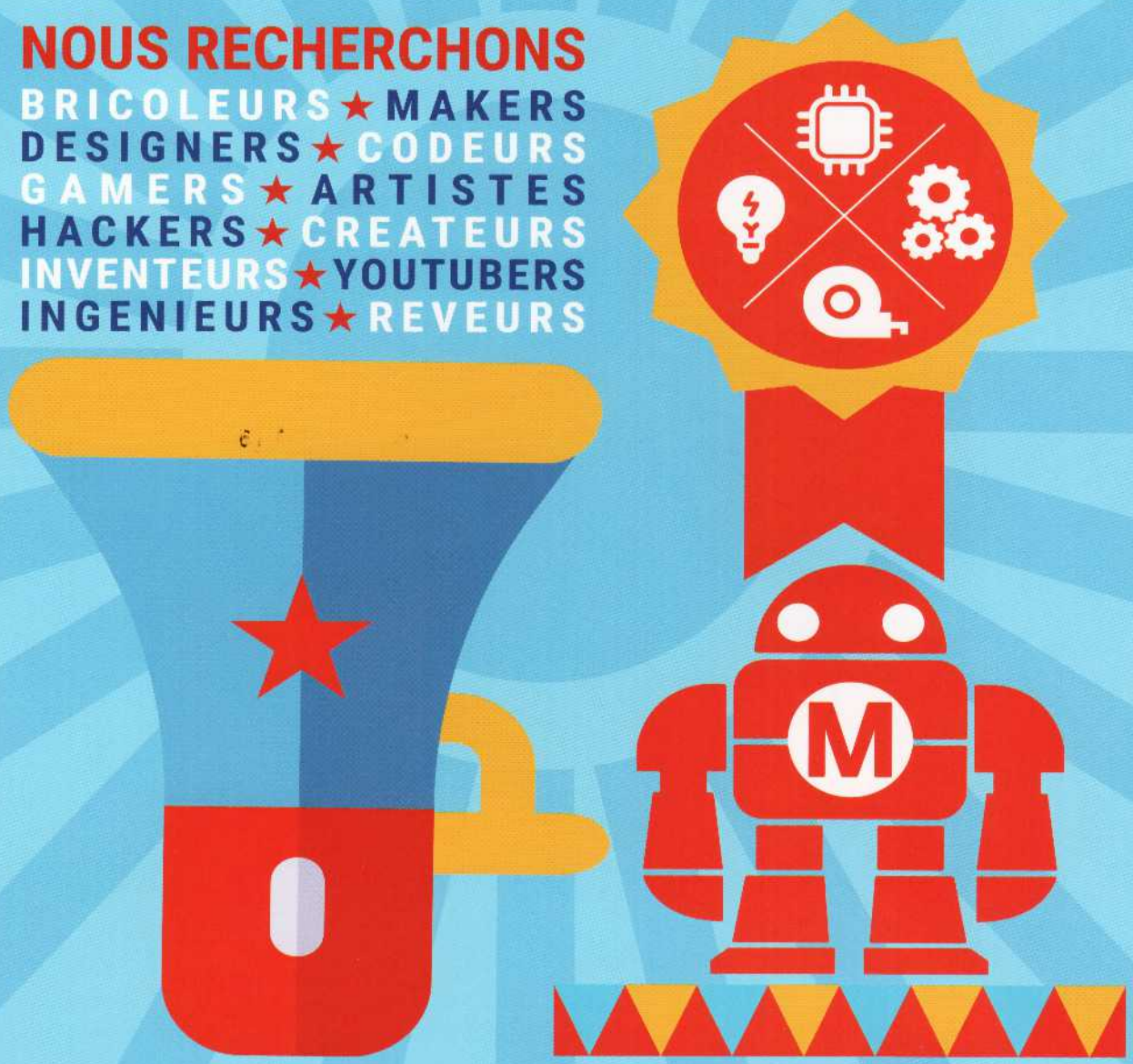
Maker Faire® Lille

DU 01 AU 03 MARS 2019 AU TRIPOSTAL

APPEL AUX MAKERS

NOUS RECHERCHONS

BRICOLEURS ★ MAKERS
DESIGNERS ★ CODEURS
GAMERS ★ ARTISTES
HACKERS ★ CREATEURS
INVENTEURS ★ YOUTUBERS
INGENIEURS ★ REVEURS



INSCRIPTIONS JUSQU'AU 30 NOVEMBRE 2018 SUR LE SITE

LILLE.MAKERFAIRE.COM