JANVIER / FÉVRIER / MARS 2012
N°5

Open Silicium

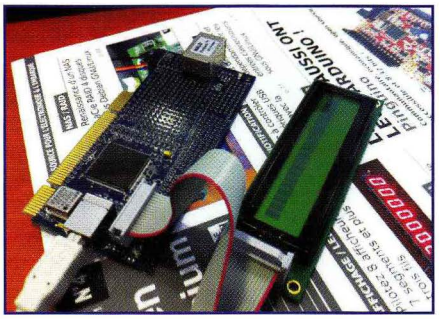
MAGAZINE

INFORMATIQUE
OPEN SOURCE
EMBARQUÉ
ÉLECTRONIQUE

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

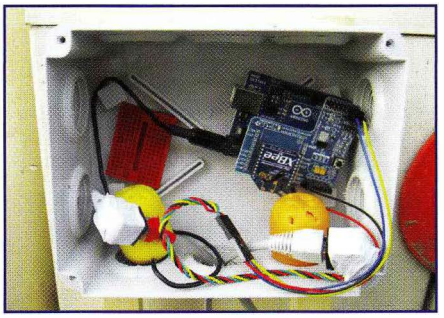
PCI / FPGA

Découverte, prise en main et exploitation de la carte Dragon PCI KNJN p.15



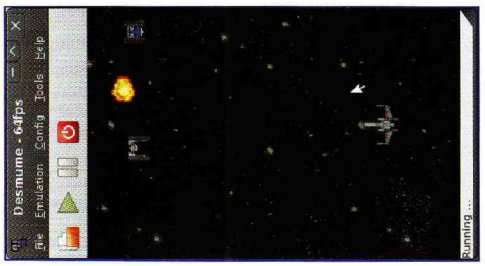
DOMO / ARDUINO

Créez un capteur domotique avec Arduino, Xbee et Domogik p.86



HOMEBREW / DS

Développez vos jeux « maison » pour Nintendo DS sous GNU/Linux p.69



AVR / CONSO

Apprenez à réduire la consommation de vos montages à base d'Atmel AVR p.10

LED / ST VALENTIN

Un cœur tout rose pour votre adoré(e) : dites-le avec des LED ! p.81

RÉSEAU / OUTILS

Découvrez Netcat, un couteau suisse pour vos réseaux TCP/IP p.96

SANS FIL

EXPLOITEZ LE BLUETOOTH ! p.24

- **Comprenez les principes et le fonctionnement des protocoles**
- **Utilisez les commandes GNU/Linux pour gérer le Bluetooth**
- **Intégrez le support BlueZ dans vos programmes C**
- **Ajoutez du Bluetooth à vos montages AVR/Arduino, Launchpad, Pic, ...**
- **Développez des applications Android pour piloter vos réalisations**

L 18310 - 5 - F: 9,00 € - RD

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

GÉREZ VOS SOURCES & PROJETS PROPREMENT !

LM 145
Actuellement en kiosque !

N°145 JANVIER 2012

L 19275 - 145 - F: 6,50 €

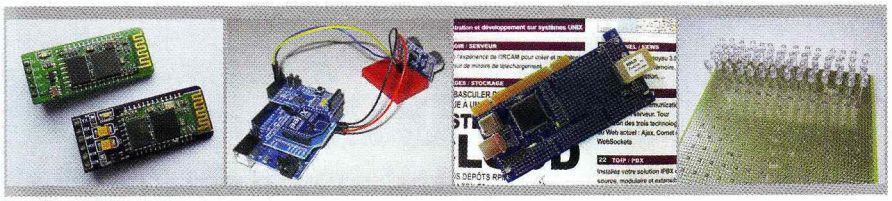
GNU LINUX MAGAZINE / FRANCE

Administration et développement sur systèmes UNIX

France Métro : 6,50 € / DOM : 7 € / TOM Surface : 950 XPF / POL. A : 1400 XPF / CH : 13,80 CHF / BEL,PORT.CONT : 7,50 € / CAN : 13 $CAD / TUNISIE : 8,80 TND / MAR : 75 MAD

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 27 JANVIER 2012 ET SUR :
www.ed-diamond.com

SOMMAIRE N°5



Open Silicium Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

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

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

Responsable publicité : Valérie Fréchard
Tél. : 03 67 10 00 27 / v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : VPM Druck Rastatt / Allemagne

Distribution France : (uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

Service des ventes : Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution. **N° ISSN :** 2116-3324
Commission paritaire : K90 839

Périodicité : Trimestriel
Prix de vente : 9 €

ÉDITO

J'espère que vous aimez le bleu !

Le bleu est l'une des trois couleurs primaires et c'est aussi une couleur généralement associée à plein de bonnes choses comme la paix, le calme, la volupté, la fraîcheur, ... « Il symbolise l'infini, le divin, le spirituel. Il invite au rêve et à l'évasion spirituelle. », nous dit Wikipédia. J'espère donc que vous aimez le bleu car vous risquez d'en avoir pour votre argent dans ce numéro.

Le bleu est quelque chose qui a longtemps été attendu par les amateurs d'électronique, même si, depuis quelque temps, lorsqu'on pense « interface » ou « truc qui pulse », c'est le blanc qui devient à la mode. À croire que chaque époque a sa couleur. Vous en doutez ? Voyons cela chronologiquement avec le domaine le plus à même de résumer les fantasmes technologiques des plus ambitieux : la science fiction. À la fin des année 70, nous avons eu droit à l'Alien de Ridley Scott et une dominante verte. Malheureusement, le peu d'interfaces apparaissant dans le film sont majoritairement « analogiques ». Ce premier volet est donc un peu hors-jeu.

Plus en arrière, alors que la technologie LED était encore inexistante, la modernité était l'orange des tubes Nixie et autres afficheurs et voyants à filament. Revisionnez un James Bond (un vrai, avec Sean Connery), vous verrez...

Le premier film, selon moi, à bien pousser la notion d'interface, est celui où apparaît un ex-barbare body-buildé aux cheveux longs qui n'espérait sans doute pas encore devenir gouvernator de Californie au moment de la sortie en salles. Le film, signé James Cameron, nous parle d'un robot méchant-méchant venu tuer la gentille-gentille Sarah Connor. Le Terminator de 1984 caractérise la modernité par le rouge. La superbe avance...

suite page 4

ÉDITO

technologique de Skynet prend la forme d'une interface presque monochrome intégralement rouge. Il en va de même pour *Predator*, sorti trois ans plus tard, avec le viseur laser rouge du canon d'épaule et le fameux compte à rebours « de quand le méchant il perd, mais sans laisser de preuves, histoire de faire passer le gentil pour un affabulateur psychotique ». Le rouge se retrouve également dans les séries comme Knight Rider (K2000) et sa sublime Pontiac Trans Am nommée KITT, autant dans le design intérieur que dans le mémorable *Larson effect* à l'avant du véhicule (effet déjà utilisé sur les Cylons de la série *Galactica*, du même créateur, d'où le nom de l'effet).

Le vert, quant à lui, n'a étrangement jamais fait de percée dans la science fiction, sauf peut-être pour *Alien*. Je pense qu'il y a principalement deux raisons à cela. Le vert est synonyme de « plante » et de « nature ». Il y a donc clairement une connotation biologique. Ensuite, les premiers écrans, type IBM PC, à tube cathodique, étaient généralement de cette teinte (et plus rarement ambre/orange). Difficile de donner un ton moderne en utilisant une caractéristique tout à fait contemporaine et courante.

Fin 1990, début 2000, nous avons eu droit à un changement. Le rouge étant devenu le symbole de la vieille science-fiction, le bleu arriva en masse. Étonnamment, cela correspond avec l'amélioration des performances des LED bleues. Pour rester dans le ton *Terminator*, c'est là, au troisième volet (2003) avec le fabuleux et terriblement sexy T-X, que le bleu en signe de modernité a fait brutalement son apparition. On a d'ailleurs droit à la formule cinématographique classique du désuet modèle T-800 qui, emprunt d'une subite humanité sentimentale, surpasse le T-X pourtant largement supérieur techniquement, mais froidement logique. En 2002, un autre film intéressant, *Resident Evil* (les suites sont, à mon avis, dignes d'être ignorées), présente également une caractéristique amusante. L'interface de la reine rouge, intelligence artificielle contrôlant le *hive*, présentée plusieurs fois dans le film, utilise une dominante bleue. Mais la projection holographique laser reste rouge, sans doute pour correspondre au nom de l'entité au détriment de la modernité du laser (n'importe qui le sait, les lasers rouges, tout le monde en a, les verts, c'est moderne et le bleu/violet, c'est la grande classe).

Enfin, pour finir, nous avons en 2008 le film réalisé par Jon Favreau tiré d'un célèbre comics et mettant en scène un riche dirigeant d'une fabrique d'armes interprété par Robert Downey Jr. : *Iron Man*. L'interface de l'ordinateur gérant la maison du héros, J.A.R.V.I.S., est principalement bleue tout comme l'interface intégrée à l'armure rouge et or du « justicier » ou encore son environnement de travail découvert au fil du film.

Voilà donc la démonstration faite que « le moderne » change de couleur et que nous sommes en pleine période bleue (tirant vers le blanc ces derniers temps). On remarquera avec amusement que la dernière version d'Android, 4.0, utilise un thème par défaut appelé Holo, tout à fait dans ces teintes. Cependant, vous l'aurez sans doute compris (étant donné le logo en couverture du magazine), le bleu de ce numéro n'a pas été choisi dans ce sens, mais en raison du sujet principalement traité : le Bluetooth. Je n'en dirai cependant pas plus, le menu est en couverture et dans le sommaire et les différents plats dans les pages qui suivent. Régalez-vous !

1 ARM et la fin des architectures x86

Voilà pour le sujet principal du numéro. Mais il y a d'autres points dont j'aimerais m'entretenir avec vous (enfin presque, puisqu'un éditorial est un peu un monologue). Il y a tout d'abord ces annonces qu'on peut voir de-ci de-là, du type « les jours de l'architecture x86 sont comptés ».

La mort de l'architecture x86, soyons clairs, c'est un peu comme les prophéties de fin du monde : on en entend parler régulièrement mais on ne voit jamais rien venir... Cependant, aujourd'hui, cela semble à certains un peu plus sérieux. Cela provient principalement de deux événements. D'une part, nous avons un commentaire, il y a quelques semaines, de Mike Silverman d'AMD : « *We will all need to let go of the old 'AMD versus Intel' mindset, because it won't be about that anymore* ». Les analyses vont bon train, de la reddition à retardement d'AMD dans la guerre contre Intel suite à l'arrivée du Core 5, au « jetage d'éponge », en passant par la stratégie commerciale la plus tordue possible. On peut aussi prendre l'affirmation pour ce qu'elle est : « la guerre AMD/Intel est un sujet obsolète ».

L'autre événement est, bien entendu, l'annonce du développement de Windows 8 sur architecture ARM. Ceci signe pour beaucoup l'arrêt de mort du duo Wintel ou, du moins, un signe annonciateur d'un changement profond. J'ai même lu quelque part l'expression « les rats quittent le navire ». Allez savoir qui est qui dans l'expression... Il faut ajouter à cela d'autres informations comme la rumeur persistante d'un nouveau Mac Air équipé d'un processeur A6 se voyant gratifié d'une autonomie doublée (rien que ça ?).

Mais soyons pragmatiques. Tout le monde se rend compte de deux choses vérifiées et tangibles :

- En l'espace de 3 ans, une partie des habitudes des utilisateurs ont radicalement changé. Tout aussi radicalement que l'usage qu'on a de son téléphone, son baladeur multimédia ou de sa console portable. Ceci ne signifie pas forcément que la majorité des utilisateurs délaisseront leurs ordinateurs personnels, mais cela donne une idée de la vitesse à laquelle le marché peut changer.
- ARM est présent partout et est arrivé sur le devant de la scène à une vitesse fulgurante. En faisant abstraction de l'époque *Acorn RISC Machine*, la véritable évolution ARM se donne en spectacle depuis 2005 avec de plus en plus de processeurs fabriqués... mais pas par ARM, et c'est là toute la clé de la réussite d'ARM.

Intel est un constructeur ou un « fondeur » de processeur. Le business d'ARM a toujours été de vendre des blocs IP (ou *IP cores*), du design, mais pas de matière. Il y avait un ARM d'Apple dans le Newton, un ARM dans l'Acorn Risc PC, mais également chez DEC avec le StrongARM et même chez Intel avec le XScale. Aujourd'hui, comme depuis le début, ARM vend un cœur, auquel les fabricants ajoutent des éléments pour produire leurs propres SoC, CPU et microcontôleurs. Ce n'est finalement pas à ARM qu'Intel doit faire face mais à Ti, Nivida, Qualcomn, Samsung, AMD, Apple, Atmel, STM et beaucoup d'autres. ARM, via les licences accordées, peut se gratifier indirectement de quelque 15 milliards de processeurs livrés, par l'intermédiaire de 600 licences cédées à plus de 200 fabricants (dont Intel).

ARM est au devant de la scène, incontestablement, et en particulier pour les tablettes et les smartphones. Dans ce domaine, l'architecture x86 a un rôle d'outsider avec une présence plus faible encore que MIPS (le MIPS XBurst équipe la première tablette sous Android 4.0, la chinoise Novo7). Mais soyons clairs, Intel ne fabrique pas que des processeurs x86 et un déclin de cette plateforme ne signifie pas pour autant le déclin de la firme. C'est un raccourci malheureux qui est trop souvent fait.

De plus, il est évident que le monde qui se dessine pour l'instant est divisé en deux univers, celui les périphériques mobiles où ARM domine et celui des machines de type PC où x86 est quasi-seul. Le déclin tant annoncé du x86 est donc indubitablement lié au déclin de la machine de bureau. Les équipements de ce type, utilisant de l'ARM, sont d'une rareté telle qu'on peut presque les qualifier d'inexistants (et Dieu sait comme j'aimerais posséder un laptop ARM). La vraie question est donc : les tablettes de future génération remplaceront-elles vraiment les PC ? À court ou moyen terme, j'en doute. Et ce principalement en raison de questions ergonomiques et des habitudes acquises depuis des années par les utilisateurs (dont je fais partie).

Bien sûr, les tâches ludiques se prêtent bien aux tablettes, fussent-elles de petite taille. Lire une vidéo, écouter de la musique, consulter et répondre à quelques mails, visiter des sites web, consulter l'état de son compte bancaire ou les horaires de train et même payer et acheter en ligne sont autant d'activités où l'écran tactile apporte beaucoup. Mais comment allons-nous écrire un long mail, rédiger un document, faire notre comptabilité personnelle, programmer, composer de la musique ou faire des montages vidéo poussés en glissant nos doigts sur un écran ? Attention, je ne dis pas que ces tâches sont irréalisables dans l'absolu avec ce matériel. Je dis que, pour l'instant, personne n'a encore trouvé comment remplacer la facilité donnée par un écran posé sur un bureau, face à un clavier et une souris. Pour l'instant...

Le déclin de l'ordinateur de bureau commencera effectivement le jour où il sera possible de travailler ou d'avoir une activité « sérieuse » avec une tablette. Que ceux qui seraient tentés d'imaginer étendre les fonctionnalités d'une tablette dans ce sens se retiennent. Il ne s'agit pas là d'innovation. Que serait une tablette de 23 pouces connectée à un clavier et une souris, si ce n'est un ordinateur de bureau ? Le seul gain serait l'élimination de l'unité centrale et une relative portabilité de l'équipement (relative parce qu'une tablette de 23 pouces, même pliée en deux, c'est un peu grand pour une poche ou un sac).

Il faut imaginer bien au-delà de la simple adaptation pour arriver à quelque chose d'utilisable. Peut-être que la projection sans surface dédiée serait une solution ou encore les écrans souples, mais nous n'en sommes pas là technologiquement. Peut-être d'ici quelques années. Mais d'ici là, ce sont toujours des ordinateurs de bureau, équipés de processeurs x86, bien compatibles avec les applicatifs et les systèmes existants qui équiperont les étudiants, les entreprises et les particuliers utilisant un ordinateur pour une activité créative, professionnelle ou équivalente. Nous avons le temps de voir venir et l'architecture x86 aussi... (oui, on risque de « trimbaler » le BIOS encore un bout de temps aussi).

ÉDITO

2 GNU/Linux a-t-il réussi ?

LinuxFr a dernièrement interviewé le professeur Andrew Tanenbaum, créateur de Minix et grand défenseur de l'architecture micro-noyau (par opposition à l'approche monolithique utilisée par le noyau Linux). Je connais les concepts et les principes des deux architectures ainsi que le franc-parler du professeur Tanenbaum. Ce n'est pas le sujet. Mais une remarque dans l'entretien publié en ligne m'a fait m'interroger : « Also, success [of Linux] is relative ».

J'utilise GNU/Linux depuis 17 ans et presque exclusivement depuis 14 ans. Autour de moi, je vois des personnes qui utilisent ce système souvent quotidiennement. J'échange des informations, des données et des commentaires dans un univers où tous, ou presque, ont des habitudes similaires aux miennes quant à l'outil informatique. Je suis rédacteur en chef d'une publication consacrée à GNU/Linux ainsi que d'une autre traitant également d'open source (celle que vous avez en main), éditées par une société reposant sur du logiciel libre. Je travaille dans une rédaction où le démarrage d'un Windows signifie tout simplement un test d'interopérabilité ou la préparation d'un article sur les alternatives en logiciels libres de certains logiciels ou systèmes propriétaires. Pour moi, GNU/Linux est un succès mais, effectivement, mon jugement est totalement relatif.

Andrew Tanenbaum détaille dans ses propos le fait qu'il gère un site web consacré à la politique que les gens ordinaires consultent (« ordinary people », c'est presque un *sheldonisme*). Il affirme que les statistiques de consultation du site montrent environ 5 % de visiteurs sous GNU/Linux, 30 % sous Mac et le reste sous Windows. « I don't think of 5 % as that big a success story. », conclut-il. Diantre ! Est-ce vrai ?

L'une des façons de vérifier est de consulter les statistiques d'un site et, pourquoi pas, l'un des nôtres et en particulier un dont je vous parlerai plus en détail par la suite. Les chiffres sont différents, avec plus de 50 % de Windows, 35 % de GNU/Linux et moins de 8 % de Mac. Côté navigateur, IE « gratte » quelque 9 malheureux pour cent contre 60 % pour Firefox et plus de 20 % pour Chrome/Chromium. Ma foi, 35 % c'est plutôt un succès.

Mais là encore, tout est relatif. Il s'agit de 35% des visites sur un site dont le contenu est clairement destiné à un public spécifique. Il faut alors se tourner vers d'autres sources, plus génériques, comme celles de StatCounter, et là, les choses sont bien différentes. L'affichage du graphe des systèmes utilisés par les visiteurs des sites participant à la collecte de données donne, pour la France et l'année écoulée, un taux d'utilisation de GNU/Linux de... 1,71 % !

Windows 7 décroche la palme avec 35 % suivi de près par 30 % de Windows XP, 22 % de Vista, et enfin, presque 10 % de Mac OS X. Le calcul est vite fait, les trois versions de Windows détectées occupe 87 % des ordinateurs pour StatCounter. Si on a le malheur de vouloir visualiser les mêmes données à l'échelle mondiale, la part de GNU/Linux, chute à 0,78 %. En réduisant aux seuls USA, ce système n'est plus affiché et est noyé dans les 1,77 % de « autres ». GNU/Linux est alors derrière les 1,47 % d'iOS, le système des iPhone, iPod et iPad. En parcourant les différents pays du monde, souvent GNU/Linux n'apparaît pas et tantôt, la grande majorité revient à Windows XP. Je ne pense pas que cela puisse être un problème de localisation puisque le support des langues est exemplaire sous GNU/Linux, contrairement aux systèmes propriétaires généralement traduits dans moins de langues et dialectes.

Oui, mais GNU/Linux gagne du terrain. Non ? Les statistiques publiquement consultables ne remontent pas avant 2008, mais sur cette période, pour la France, GNU/Linux a progressé de 1,43 % à 1,72 % en trois ans avec un maxima à 1,94 % en 2009. Durant la même période, l'usage de Windows 7 est monté en flèche dès sa sortie, compensant la lente descente de Windows XP.

Bien sûr, monsieur tout-le-monde ne choisit pas son système lors de l'achat d'un nouvel ordinateur (sauf à choisir une machine Apple) et de toutes façons, il souhaite généralement avoir la même chose en mieux (comprendre, « en plus récent » mais « pas trop différent »). Mais là, ce sont des statistiques d'utilisation, pas d'activation. Existe-t-il une masse cachée de machines, non connectées, sous GNU/Linux ? Un énorme collectif d'utilisateurs de GNU/Linux rebootant sous Windows pour surfer sur le Web ? Là encore, j'en doute.

Voilà les chiffres ! Vient alors le moment de les interpréter. Ceci peut-il être utilisé pour juger du succès d'un système d'exploitation ? Oui, si le succès est synonyme de popularité. Mais si l'on traduit *success* par « réussite », l'interprétation peut être tout autre. Un peu comme « réussir dans la vie » ne veut pas dire « réussir sa vie ».

Alors, qu'en est-il ? GNU/Linux a-t-il réussi ? Pour répondre à cette question, il suffit de s'interroger sur sa satisfaction personnelle dans l'utilisation du système. GNU/Linux me permet de travailler, de me divertir, de me tenir informé, de satisfaire ma passion et de vaquer aux tâches administratives courantes (courriers, impôts, comptabilité, etc.). Suis-je tantôt bloqué dans mes

démarches ? Rarement, et c'est généralement pour quelques raisons ou activités précises.

Le jeu, par exemple, est quelque chose de très peu présent sous GNU/Linux, mais je ne serais pas enclin à utiliser une machine Windows pour cette activité. Lorsque je souhaite jouer, je ne veux pas me battre avec un antivirus ou un pilote de carte graphique. Il y a d'autres plateformes, spécialisées, qui sont mieux adaptées. Les médias protégés par des mécanismes DRM me bloquent, comme du contenu vidéo ou audio lié à une plateforme ou encore des documents dans un format fermé qu'il n'est possible de lire qu'avec des applications précises. Là, il ne s'agit pas d'un problème technique mais éthique et cela relève de la pérennité. Lorsque j'achète un contenu, je veux pouvoir en profiter même si l'éditeur ou l'éditeur du logiciel n'existe plus (j'ai payé ce droit). Un bon vieux CD ou DVD arrivant par la poste fait alors très bien l'affaire, aussi bien sous Windows, GNU/Linux, Mac OS X ou un lecteur classique. Enfin, il y a les vraies limitations techniques. Lorsqu'un matériel n'est pas supporté par mon système, je dois faire face à un véritable blocage ou, au minimum, à une recherche intensive d'une solution adaptée à ce que je préfère utiliser. Tantôt cela débouche sur un échec cuisant, mais souvent, d'autres ont fait face à la même situation et un recoupement de toutes les informations et du code de chacun permet d'aboutir à un résultat (et souvent à un article).

Donc oui, GNU/Linux, pour moi, est une réussite. Une réussite qui n'est pas majoritairement partagée et un peu réservée à un public de spécialistes et de passionnés, certes. Est-ce un problème ? GNU/Linux serait-il un meilleur système s'il affichait 30 %, 50 % ou 90 % d'utilisation mondiale ? Certainement pas.

L'histoire nous a montrés qu'en bien des situations, la majorité ne fait pas la vérité et que le nombre ne fait pas la raison. Le choix d'un système d'exploitation est une affaire de préférences et de besoins personnels, même s'il n'est possible de choisir judicieusement que si l'on dispose d'une connaissance du sujet et de ses besoins. Chose qui généralement demande de l'énergie, de l'investissement et du temps, beaucoup de temps.

Je conclurai en précisant que le domaine exploré est ici celui du poste de travail dans une vision globale. La présence de GNU/Linux dans des secteurs précis, comme celui des serveurs web, par exemple, affiche des statistiques bien différentes. Mais cela, c'est la partie immergée de l'iceberg, et non le monde des gens ordinaires, comme dirait le professeur Tanenbaum. Parlant du personnage, je compte bien garder un œil sur le portage annoncé de Minix 3 sur ARM. Le projet vise l'embarqué et Andrew Tanenbaum marque un point en précisant qu'Android n'utilise que Linux comme base et avec un grand nombre d'ajouts et surtout de coupes franches dans le code. Google a fait le ménage dans Linux pour créer son Android. Preuve que tout n'a pas semblé bon à prendre. Nous verrons donc ce que Minix pourra apporter dans ce domaine.

3 Unix Garden

Un changement majeur a été opéré dernièrement sur le site du magazine *Open Silicium*, ainsi que sur l'ensemble des sites de nos différentes publications. Ils redirigent tous un autre site. Nous avons choisi de réunir l'ensemble de ces informations sur un site qui ne vous est sans doute pas inconnu : Unix Garden (www.unixgarden.com). Une refonte totale de l'ensemble des pages et de l'architecture de notre site de contenu font, à présent, d'Unix Garden notre plateforme principale pour toutes nos éditions. Beaucoup d'entre vous sont lecteurs de plus d'un de nos magazines ou sont susceptibles de l'être puisqu'ils reposent tous sur les mêmes bases : une passion pour la technique et le partage, pour l'open source et pour la publication d'articles utiles tant au niveau pratique que sur la réflexion à propos de l'usage des données, de leur sécurité et de leur caractère privé.

Il s'agit là de la troisième évolution majeure du site et toute la difficulté a été de « réunir sans mélanger ». Vous y trouverez donc une sélection d'articles (le compteur en affiche actuellement 1654) parus dans nos divers magazines, *Open Silicium* bien sûr, mais également *GNU/Linux Magazine France* (GLMF pour les intimes), *GNU/Linux Pratique*, *MISC*, *GNU/Linux Essentiel* ainsi que leur déclinaison en hors-série. Tout cela classé par publication et par thématique. Cette sélection d'articles se verra grossie et complétée au fil du temps. Vous trouverez également sur le site les sommaires et les annonces des nouvelles publications et bien d'autres choses, mais aussi (et je me rends compte que vous lisez peut-être ceci sur le site lui-même, oups) les différents éditoriaux, dont ceux de votre serviteur dévoué (ou dévoué à sa passion qui est également la votre, ce qui revient au même).

N'hésitez pas à explorer le site et nous faire part de vos critiques et améliorations souhaitées. Sur ce, je vous souhaite, pour ma part, une bonne lecture de ce qui suit et une bonne balade dans les articles sur le site. ■

Denis Bodor

CONCOURS LINUXEMBEDDED.FR : LES RÉSULTATS SE PRÉCISENT

Comme annoncé lors des Rencontres Mondiales du Logiciel Libre à Strasbourg en juillet dernier, le blog linuxembedded.fr a lancé un concours autour d'une plateforme embarquée que vous connaissez bien, puisqu'elle a fait l'objet de la couverture du premier numéro : la carte Mini2440 de FriendlyARM. La première phase du concours s'est terminée en octobre dernier avec des résultats très intéressants.

L'objectif du concours dont nous sommes partenaires est de construire un système qui se connecte au réseau et affiche une interface graphique le plus rapidement possible. La première phase s'est déroulée dans un environnement émulé par Qemu. Les participants sont divisés en deux catégories distinctes : d'une part les « vieux gnous » concourant individuellement et d'autre part les « gangs de poussins » réunissant des étudiants sous forme d'équipes de une à plusieurs personnes.

L'objectif, bien qu'il s'agisse d'un concours, a un but pédagogique et se déroule dans un esprit de partage et d'entraide, grâce au wiki à disposition et via IRC (vous savez une vraie plateforme de discussion en ligne, pas Facebook).

Pour cette première phase sur émulateur, un total de 16 projets ont été rendus et l'équipe du jury a remarqué une qualité globale très intéressante. Au point que ces informations ont permis d'améliorer le fork Qemu de Michel Pollet (alias BusError) supportant la carte mini2440. D'autres contributions et techniques d'optimisation très intéressantes ont également été observées. Nous restons en contact avec l'équipe d'organisation afin de trouver un moyen de partager ces informations dans nos pages après la clôture du concours, pour que les lecteurs ayant craqué pour la Mini2440 puissent en profiter.

Pendant la première phase, les participants devaient arriver à prouver leur capacité à démarrer un système GNU/Linux dans un simulateur de Mini2440. Le système devait démarrer une application graphique tactile qui affichait :

- l'adresse IP obtenue par DHCP ;
- un bouton déclenchant l'apparition du nom du participant.

Cette étape a été franchie avec succès par :

Pour les gangs de poussins :

- Gabriel Huau (Leirbag) : 0,4s ;
- Alexandre Aminot & Clément Leger (CLAX) : 1,4s ;
- Jean-Baptiste Théou (Anbreizh) : 1,7s ;
- Martin d'Allens (7trollan) : 2,2s ;
- Élie Bouttier & Franklin Delehelle (7totomatix) : 2,5s.

Pour les vieux gnous :

- Fabrice Jouhaud (Yargil) : 0,3s ;
- Julien Heyman (Bids) : 0,3s ;
- Cédric Roux (Sed) : 0,4s ;
- Julien Pichon (Quidox) : 1,7s ;
- Albéric Aublanc (Pouet) : 1,9s ;
- Laurent Navet (Mali) : 2,7s ;
- Thomas Bouffon : 3,5s ;
- Yann Le Doaré (Linuxconsole) : 3,6s.

Évidemment, ces temps ne représentent pas la réalité puisqu'ils ont été mesurés en émulation. Les gagnants de cette première étape ont eu ainsi le plaisir de recevoir une Mini2440 afin de poursuivre le concours. Bien entendu, c'est un prix et ils pourront donc conserver la carte quelle que soit l'issue de la finale.

La deuxième phase, la finale, se déroule sur carte. Elle commence dès que possible et se termine à la fin décembre. Il s'agit là de réduire encore le temps de démarrage en reposant sur les fonctionnalités propres au matériel. L'image flash ainsi que les codes sources du système devront être transmis et documentés (décrit). Un serveur FTP est mis à la disposition des participants dans ce but.

Les gagnants seront désignés au cours du mois de janvier. Ils devront présenter leurs techniques lors de l'événement de clôture et de remise des récompenses : tablettes multimédias, Wobes Plugnsurf, gadgets ARMWorks pour mini2440 et abonnements à *Open Silicium*. ■

GNU LINUX
DÉCOUVRIR, COMPRENDRE ET UTILISER LINUX
PRATIQUE
HORS-SÉRIE

ENVIE DE DÉVELOPPER VOS PROPRES OUTILS ?

LA PROGRAMMATION AVEC PYTHON

NIVEAU : DÉBUTANT ET INTERMÉDIAIRE

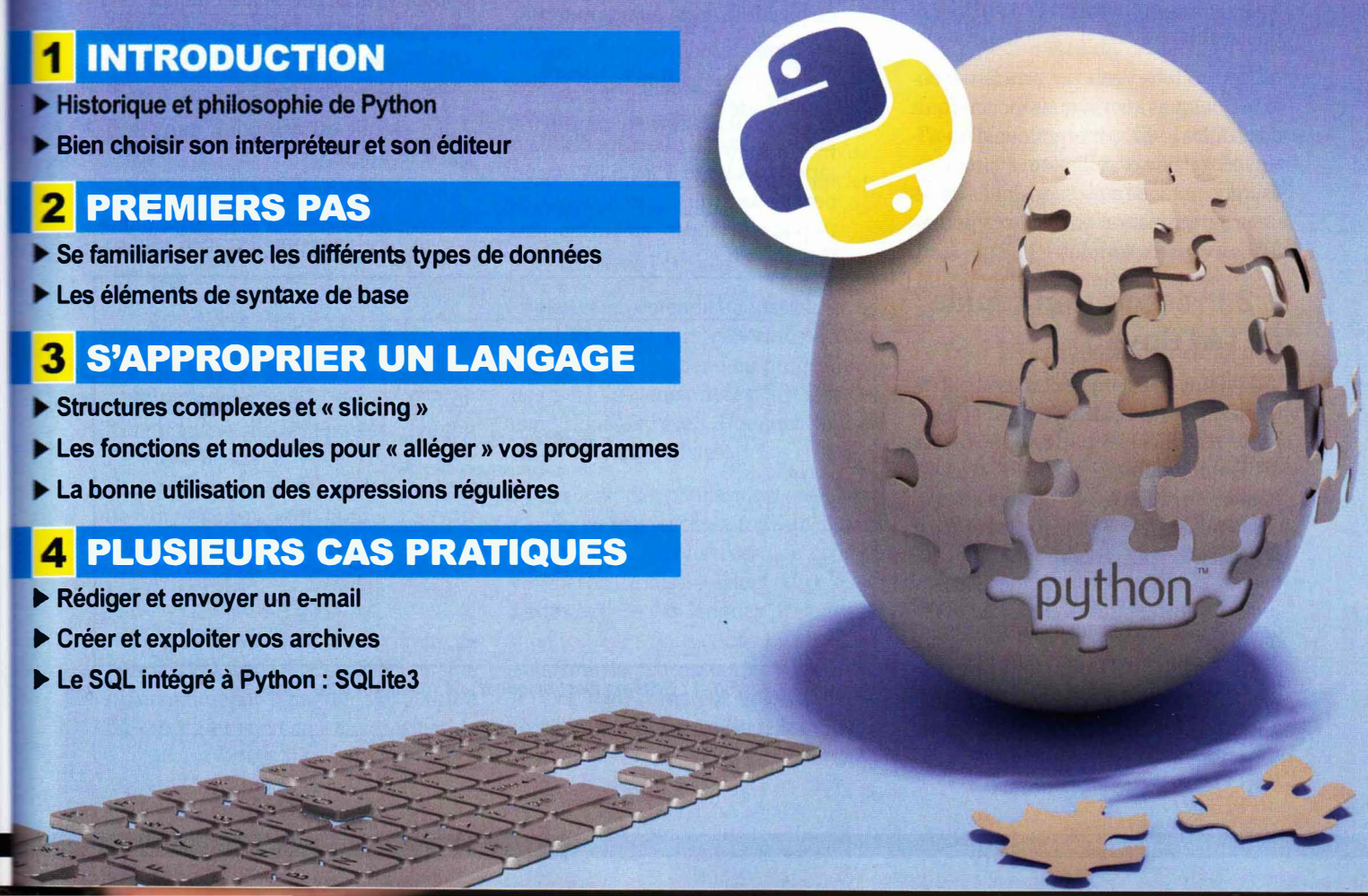
TOUTES LES BASES DU LANGAGE POUR CRÉER RAPIDEMENT VOS PREMIERS PROGRAMMES !

1 INTRODUCTION
- Historique et philosophie de Python
- Bien choisir son interpréteur et son éditeur

2 PREMIERS PAS
- Se familiariser avec les différents types de données
- Les éléments de syntaxe de base

3 S'APPROPRIER UN LANGAGE
- Structures complexes et « slicing »
- Les fonctions et modules pour « alléger » vos programmes
- La bonne utilisation des expressions régulières

4 PLUSIEURS CAS PRATIQUES
- Rédiger et envoyer un e-mail
- Créer et exploiter vos archives
- Le SQL intégré à Python : SQLite3



RÉDUIRE LA CONSOMMATION D'UN AVR

par Jérôme Labidurie

Qu'il soit alimenté par panneau solaire ou par piles, un montage autonome nécessite une faible consommation. Cela tombe bien car les AVR sont capables de nous impressionner dans ce domaine. Voyons ensemble comment...

1 Introduction

1.1 Pourquoi réduire la consommation ?

Dès que l'on veut créer des montages autonomes, la consommation devient très vite un facteur limitant. Dans un montage alimenté par piles, il peut être très contraignant (et cher) de devoir les changer souvent. Même si nous optons pour un panneau solaire, il sera nécessaire de consommer le moins possible si nous voulons éviter d'y adjoindre 10m².

Nous allons dans cet article découvrir quelques techniques permettant de réduire le coût énergétique d'un ATtiny45 à l'aide d'un montage simple. Il est recommandé d'avoir quelques bases de programmation en C de ces petites bêtes avant d'aller plus loin.

1.2 Présentation rapide de l'ATtiny45

L'ATtiny45 est un petit microcontrôleur 8bits de la société ATMEL. Il comporte seulement 8 pattes dont 6 sont utilisables comme entrée/sortie. Ce nombre se réduit à 5 si l'on veut conserver la fonction ISP[1] puisque le reset est nécessaire dans ce cas.

Sa petite taille ne l'empêche pas de disposer de nombreuses fonctions intégrées dignes des plus grands. 4KB de flash pour stocker le code, 256B d'eeprom et 256B de RAM permettent de développer des programmes conséquents. Parmi les fonctions qu'il offre, citons celles qui vont nous intéresser :

- Comparateur analogique qui permet de tester une tension par rapport à un seuil prédéfini. Celui-ci peut être présent sur une 2ème patte ou en interne (1,1V).
- Watchdog qui peut déclencher un reset ou une interruption.
- Oscillateur interne qui permet de ne pas utiliser de quartz externe.
- Fonctions de réduction de la consommation que nous étudierons plus en détail ci-dessous.
- Enfin, il possède une plage d'alimentation très large (1,8V-5,5V), ce qui permet de l'utiliser facilement sans trop se soucier des valeurs fournies.

Pour plus de détails, vous pouvez consulter la datasheet **[DS]** du composant.

1.3 Présentation d'une fonction simple servant d'exemple

Pour les besoins de cet article, nous allons utiliser un petit montage simple. La fonction à offrir sera de faire clignoter une LED pendant la nuit. Nous pouvons rattacher cela au monde réel en imaginant par exemple que nous créons une balise cardinale pour le balisage maritime.

Le circuit est reproduit dans la figure 1 ci-dessous.

Une photo-résistance montée en pont diviseur de tension est branchée sur une

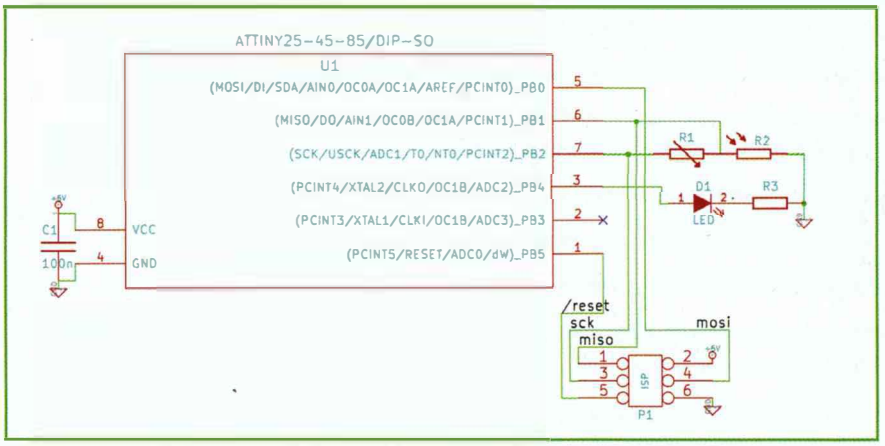


Fig. 1 : Schéma électronique d'une (pseudo)balise cardinale

[1]ISP : In System Programming : permet de programmer le µC directement dans le montage ?

entrée du comparateur analogique de notre Attiny. Elle nous permettra de détecter une baisse de luminosité. Quand la lumière diminue, la résistance du composant augmente, et selon la loi d'Ohm, la tension à ses bornes augmente. Le potentiomètre nous permet d'ajuster le point de détection par rapport à la référence interne.

Le reste du schéma se compose d'une simple LED associée à sa résistance pour réduire le courant la traversant.

Nous trouvons aussi l'habituel connecteur ISP pour reprogrammer l'ATtiny en situation.

Table 7-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes

	Active Clock Domains					Oscillators	Wake-up Sources					
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{PCK}	Main Clock Source Enabled	INT0 and Pin Change	SPM/EEPROM Ready	USI Start Condition	ADC	Other I/O	Watchdog Interrupt
Idle			X	X	X	X	X	X	X	X	X	X
ADC Noise Reduction				X		X	X(1)	X	X	X		X
Power-down							X(1)		X			X

Note: 1. For INT0, only level interrupt.

Fig. 2 : Extrait de la datasheet de l'ATtiny45 [DS] p35

2 Sleep modes des AVR

Il est bien connu que tout ordinateur passe son temps à attendre. Les µC n'échappent généralement pas à la règle. Pour réduire la consommation pendant ces temps d'attente, nous pouvons endormir le composant en ne gardant que le strict nécessaire pour le réveiller au moment opportun et lui faire exécuter sa tâche.

2.1 Les différents sleep modes

Notre Attiny45 propose 3 modes suivant les fonctions éteintes et les capacités de réveil offertes :

- *Idle* ;
- *ADC Noise Reduction* ;
- *Power-down*.

Notons aussi que d'autres AVR offrent d'autres modes appelés *Standby*, *Extended Standby* ou encore *Power-Save*. Vous pouvez vous référer à la datasheet de votre composant pour en savoir plus.

2.1.1 Idle

Ce mode conserve un maximum de périphériques en fonction, il se contente principalement d'arrêter l'exécution du programme en cours et permet de le réveiller de nombreuses manières. En contrepartie, il consommera plus d'énergie que ses camarades.

2.1.2 ADC Noise Reduction

Comme son nom l'indique, le but de ce mode est de réduire le bruit électrique lors de l'utilisation du convertisseur analogique/numérique. Il arrête les horloges de la flash, du CPU et des entrées/sorties.

2.1.3 Power-down

Enfin, le mode le plus économe. S'il ne consomme presque rien, peu de conditions peuvent l'en sortir. Principalement, un changement de niveau sur *INT0* (interruption) ou le déclenchement du watchdog.

2.2 Lequel choisir ?

Il est bien entendu impossible de répondre de manière générique à cette question. Tout dépend du programme, des périphériques nécessaires et des conditions de réveil. Donnons tout de même quelques pistes.

Il faut étudier précisément quels périphériques sont nécessaires à un instant donné pour ne conserver que ceux-ci. On cherchera aussi à éviter, outre les attentes actives, les longues périodes de `_delai_ms()`. Dans ce cas, on préférera endormir le processeur et se réveiller à l'aide d'un compteur (idle), d'une interruption extérieure (tous les modes) ou du watchdog si la durée n'a pas besoin d'être très précise (power-down).

On aura tout intérêt à varier les plaisirs en utilisant plusieurs modes au cours de notre programme. Quelques mA sont toujours bons à gagner.

Appliquons cela à notre exemple de balise.

Pour le comparateur analogique, la 1ère idée est de passer en Idle afin d'être réveillé par interruption quand le seuil est franchi. Mais la détection de nuit n'a pas besoin d'être immédiate, un délai de quelques secondes est largement acceptable. Nous pouvons donc rester en Power-down quasiment en permanence et nous faire réveiller régulièrement par le watchdog afin d'utiliser le comparateur analogique (sans interruption cette fois). Ainsi, en détournant le watchdog de sa fonction première, nous économisons beaucoup.

Pour faire clignoter notre LED, nous pouvons envisager la même solution. Mais elle présente tout de même quelques inconvénients : le watchdog ne propose que quelques valeurs de timeout (15ms, 30ms, 60ms, 120ms, 250ms, 500ms, 1s, 2s et aussi 4s et 8s sur l'ATtiny45) ; de plus, ces valeurs ne sont pas précises et augmentent quand la tension d'alimentation diminue. Notre clignotement devant être précis, il nous faut une autre solution. Nous pouvons cette fois envisager le mode Idle en nous faisant réveiller par un Timer/Counter.

2.3 Show us the code !

2.3.1 Watchdog

La bibliothèque avr-libc nous fournit un certain nombre de fonctions et macros nous facilitant la vie pour utiliser les sleep modes de nos AVR. De ce fait, un endormissement en mode Power-down du µC sera aussi simple que :

```
#include <avr/sleep.h>
...
   set_sleep_mode(SLEEP_MODE_PWR_DOWN);
   sleep_mode();
```

On s'assurera bien sur que les interruptions sont activées au préalable (via **sei()**) afin de pouvoir être réveillé. L'exécution continuera alors simplement à la ligne suivante. Il est aussi possible de découper le déclenchement en étapes élémentaires (via **sleep_enable()**, **sleep_cpu()**) afin d'éviter des conditions de blocage. Typiquement si l'interruption de réveil apparaît pendant la fonction **sleep_mode()**.

Nous pouvons donc écrire les fonctions suivantes pour gérer notre endormissement :

```
/** set system into the sleep state
 * system wakes up when watchdog is timed out
 *
 * @param duration watchdog timer (see wdt.h)
 */
void system_sleep(uint8_t duration)
{
   PORTB &= ~_BV(DRIVE); // Power off photoresistor
   ACSR |= _BV(ACD); // switch Analog Comparator OFF
   PRR |= _BV(PRADC); // switch off ADC clock
   wdt_enable ( duration );
   // set wdt to generate interrupt instead of reset (see p47)
   WDTCR |= _BV(WDIE);

   set_sleep_mode(SLEEP_MODE_PWR_DOWN); // sleep mode is set here
   sleep_mode();

   wdt_disable ();
   PORTB |= _BV(DRIVE); // Power on photoresistor
   PRR &= ~ _BV(PRADC); // switch on ADC clock
   ACSR &= ~ _BV(ACD);  // switch Analog Comparator ON
   // wait for stabilization of internale voltage ref (See p44)
   _delay_ms(1);
}

/** Interrupt handler for Watchdog
 */
ISR(WDT_vect)
{
   // this is set to 0 on each interrupt, so re-force it to 1
   // set wdt to generate interrupt instead of reset (see p47)
   WDTCR |= _BV(WDIE);
}
```

On désactive le comparateur analogique et l'horloge associée. On active alors le watchdog configuré pour utiliser une interruption. On désalimente aussi notre photo-résistance. Lorsqu'on est réveillé, on rétablit tout cela pour être en capacité d'utiliser notre photo-résistance. Notre boucle principale pourra alors ressembler à ceci :

```
while (1)
{
   if ( ! bit_is_set (ACSR,ACD) ) {
      // ACD is not set --> dark, switch on led
      PORTB |= _BV(LED);
      system_sleep(WDTO_8S);
   }
   else {
      // ACD is set --> light, switch off led
      PORTB &= ~ _BV(LED);
      system_sleep(WDTO_8S);
   }
}
```

Ainsi, le changement de luminosité sera détecté sous un délai maximum d'environ 8s et la LED sera allumée tant qu'il fera nuit. Vous trouverez le code complet dans **[CODE]** sous le nom **ex1.c**.

2.3.2 Timer0

Le Timer0 compte les cycles d'horloge sur 8bits. Son horloge peut être divisée par un prescaler. Il peut déclencher une interruption lorsqu'il déborde. Nous allons utiliser cette particularité en comptant le nombre de débordements nécessaires pour atteindre une durée voulue.

La période de débordement (overflow) du compteur (en secondes) peut être calculée ainsi : taille du compteur * prescaler / F_{CLK_IO}. Il nous suffit alors de diviser la durée recherchée par ce nombre pour obtenir le nombre de débordements nécessaires. Nous pouvons alors écrire les fonctions suivantes pour attendre une durée voulue en restant en mode Idle la plupart du temps.

```
volatile uint16_t timer0_overflows; // number of timer overflows

/** sleep by going idle
 * @param duration sleep time (ms)
 * @param prescal value of Timer0 prescaler
 */
void idle_sleep(uint16_t duration, uint16_t prescal)
{
   // compute number of overflows needed
   uint16_t overflows = ( ( duration * F_CPU ) / (0xFF * prescal) )
/ 1000;

   while (timer0_overflows < overflows)
   {
      set_sleep_mode(SLEEP_MODE_IDLE); // sleep mode is set here
      sleep_mode();
   }
   timer0_overflows = 0;
} // idle_sleep

/** Interrupt for Timer0
   */
ISR (TIMER0_OVF_vect) {
   timer0_overflows++;
}
```

Fonction que nous appellerons ainsi :

```
// set a timer0
PRR    &= ~ _BV(PRTIM0); // enable Timer0 module
TIMSK  |=   _BV(TOIE0);  // enable interrupt on overflow
TCCR0B |=   _BV(CS02);   // set prescaler to CLKio/256
idle_sleep (3000, 256); // sleep 3s
```

Nous pouvons alors faire clignoter notre LED à la fréquence voulue avec une précision relativement importante. On n'oubliera pas d'arrêter le comparateur analogique et la photo-résistance pendant chaque cycle de clignotement. Vous trouverez le code complet dans **[CODE]** sous le nom **ex2.c**.

3 Réduire la consommation en fonctionnement

Maintenant, attaquons-nous au moment où notre µC travaille. Voici quelques trucs pour réduire sa consommation en dehors des périodes de sommeil.

3.1 Horloge

La fréquence d'horloge a une forte influence sur le courant utilisé, il est donc inutile de fonctionner à de hautes fréquences si ce n'est pas nécessaire.

Avec la configuration usine, notre ATtiny45 fonctionne sur l'oscillateur interne à une fréquence de 8MHz. Celle-ci traverse une PLL qui la divise par 8 et offre donc une horloge système à 1MHz. Pour réduire la consommation générale, nous pouvons jouer sur cette configuration de deux manières. Configurer la PLL pour augmenter la division et ralentir l'horloge système ou changer la source d'origine. En restant sur une génération interne d'horloge, l'ATtiny45 nous offre aussi une source à 6,4MHz (mode de compatibilité avec l'ATtiny15) et une source de faible consommation à 128KHz. Avec cette dernière, nous réduisons de beaucoup notre puissance de calcul et nous perdons en précision (à titre d'exemple, sur mon Attiny à 3,9V, j'obtiens une fréquence CPU de 11,9KHz au lieu des 128/8=16KHz attendus).

Pour jouer sur ces paramètres, nous disposons des fusibles (fuses) pour sélectionner la source (CKSEL) et d'un fusible+registre pour la PLL (CKDIV8+CLKPR)

On sélectionne la clock à 128KHz en positionnant les poids faibles du fuse low. Ce qui donne avec avrdude et un programmeur de type stk500 :

```
$ /usr/bin/avrdude -V -p t45 -c stk500  -P /dev/ttyUSB0 -q -U
lfuse:w:0b01100100:m
```

Nous voici donc avec une horloge principale à 128KHz qui passe dans un diviseur par 8 pour avoir une horloge système à 16KHz.

Dans le code, on se bornera à changer la définition de **F_CPU** (utilisé par l'avr-libc). Pour une raison que je ne m'explique pas, j'ai aussi dû supprimer le prescaler du Timer0. Le comportement était erratique en le laissant à 256. Le code est disponible sous le nom **ex3.c** dans **[CODE]**.

Attention !

Attention, si vous changez la source d'horloge de votre ATtiny, il faut dès lors coordonner l'horloge de votre programmeur avec l'horloge système de votre ATtiny. Il est recommandé d'avoir une fréquence de programmation au moins égale à 1/5 de la fréquence de la cible. J'ai pour habitude de simplement mettre la même.

Avrdude attend une valeur en µs. Donc pour une fréquence à 1MHz (valeur usine), votre période doit être d'au moins 1/1MHz = 1µs. Pour une horloge cadencée à 16KHz (128/8), votre période devient 62,5µs.

Ce réglage peut se faire de 2 manières suivant le modèle :

- Avec un stk500, vous pouvez positionner le paramètre **sck** :

```
$ /usr/bin/avrdude -p t45 -c stk500  -P /dev/ttyUSB0 -tuF
avrdude> parms
>>> parms
Vtarget         : 0.0 V
SCK period      : 0.1 us
avrdude> sck 62
>>> sck 62
avrdude> q
>>> q
```

Vous aurez remarqué que les paramètres affichés par mon clone de stk500 ne sont pas valides.

- Si celui-ci n'est pas disponible, l'option **-B** en ligne de commandes sera votre amie :

```
$ avrdude -p t45 -B62 -c stk500  -P /dev/ttyUSB0 -U flash:w:ex3.hex
```

À noter aussi que mon programmeur n'est pas très stable à cette fréquence. Pour éviter de rater des programmations, on pourra flasher le programme à 1MHz et ne changer la fréquence (via le lfuse) qu'après.

3.2 Désactiver l'inutile

Les AVR nous offrent aussi un mécanisme simple pour désactiver les périphériques inutiles pour notre application. C'est très utile en fonctionnement, mais aussi dans certains sleep modes comme Idle. On pourra ainsi activer ceux-ci à la demande et pour le temps strictement nécessaire.

Cela se fait via le *Power Reduction Register* (PRR). Notre ATtiny45 nous permet de désactiver les Timers (0 et 1), l'USI, le convertisseur analogique/numérique. Nous pouvons aussi,

via d'autres registres, désactiver le comparateur analogique (ADCSRA), le watchdog (WDTCR) ou la tension de référence interne (en désactivant l'ADC et le comparateur analogique). Il est donc important d'étudier la datasheet du composant pour traquer et désactiver toutes les fonctions inutiles pour votre application.

```
// Configure Power Reduction Register (See p39)
// disable Timer1, USI
PRR |= _BV(PRTIM1) | _BV(PRUSI);
// switch Analog to Digitalconverter OFF
ADCSRA &= ~_BV(ADEN);
```

3.3 Consommations « parasites »

Même non connectées, les entrées/sorties inutilisées du composant peuvent consommer du courant. Pour éviter cela, on s'assurera :

- que les pins non connectées sont configurées en entrée avec la résistance de pull-up afin d'avoir un niveau bien défini.
- que les buffers d'entrée sont désactivés.

```
// enable input / Pull-Up on unconnected pins (See 10.2.6 p59)
 DDRB  &= ~ ( _BV(DDB0)   | _BV(DDB3)   | _BV(DDB5) );
 PORTB |= (  _BV(PORTB0) | _BV(PORTB3) | _BV(PORTB5) );
//disable all Digital Inputs (See p142, p125)
 DIDR0 &= ~ ( _BV(ADC0D) | _BV(ADC2D) | _BV(ADC3D) | _BV(ADC1D)
           | _BV(AIN1D) | _BV(AIN0D) );
```

Ces modifications sont accessibles dans **ex4.c**.

On fera aussi attention au schéma de notre montage. Prenons par exemple la figure suivante :

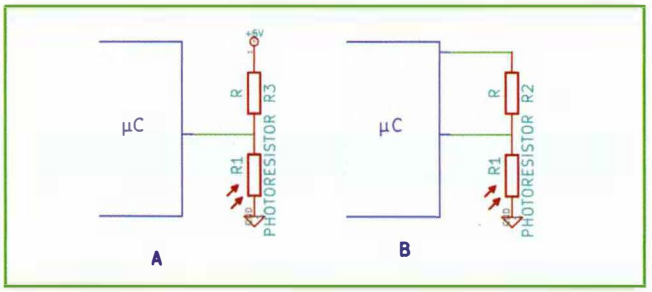


Fig. 3 : 2 schémas ayant la même fonction

Dans le schéma A, le pont de résistances consomme en permanence, quel que soit l'état du µC et le besoin de mesure. Dans le schéma B, l'ATtiny45 n'alimente la photo-résistance que lorsqu'une mesure est nécessaire, cela nous prend une patte de plus, mais réduit grandement la consommation totale. On prêtera toutefois attention à ce que nos résistances ne tirent pas plus que les 40mA que peut fournir notre ATtiny (en cas de besoin en courant supérieur à 40mA, on peut adjoindre un système de transistor piloté par l'ATtiny, qui bloquera ou non le courant fort pris sur l'alimentation selon l'état de la sortie, sans augmenter le courant fourni par l'ATtiny).

4 Consommations comparées

Quelques notes sur les mesures suivantes.

L'alimentation est régulée à 3,9V, soit l'équivalent de 3 piles AA rechargeables.

Pour l'exemple 1 : loop : fonctionnement en boucle infinie. **sys_sleep** : passage en mode power down après chaque action.

Pour l'exemple 2 : **delay_ms** correspond à l'utilisation de la fonction de la **libc _delay_ms()** pour les temps d'attente, contrairement à idle qui passe en mode idle pour ces mêmes attentes.

	Nuit LED on	Nuit LED off	Jour
ex1 **loop**	3,46 mA	Non applicable	1,39 mA
ex1 **sys_sleep**	3,33 mA	Non applicable	4,66 µA
ex2 **delay_ms**	3,46 mA	1,27 mA	4,66 µA
ex2 idle	3,41 mA	0,67 mA	4,66 µA
ex3 (128KHz)	3,35 mA	0,23 mA	4,66 µA
ex4 (128KHz+PRR)	3,33 mA	0,21 mA	4,64 µA

De ces quelques mesures, nous pouvons déduire que le mode power down est effectivement très intéressant puisqu'il permet de réduire la consommation de notre AVR à seulement quelques µA. On remarque aussi qu'endormir le µC plutôt que de faire une attente active est valable sur le long terme (jusqu'à 0,6mA de gagné pour ex2). De même, utiliser l'horloge cadencée à 128KHz vaut le coup si la précision n'est pas très importante. Enfin, lorsque l'on commande la LED, celle-ci consomme tellement qu'il n'est pas possible d'obtenir une réduction très significative.

En conclusion, un petit calcul de consommation en mAH nous permet d'estimer la durée de vie de nos piles. Dans le cas le plus consommateur, elles dureront ~66 jours et dans le meilleur des cas ~278 jours. Soit un gain d'un facteur >4. Cela vaut la peine, non ? ■

Références

[CODE] Code complet des exemples (Licence GPLv2). http://svn.tuxfamily.org/viewvc.cgi/scrippets_scripts/trunk/avr/OpenSilicium_01/

[DS] Datasheet de l'ATtiny45, http://www.atmel.com/dyn/resources/prod_documents/doc2586.pdf

[AT1] AVR1010: Minimizing the power consumption of Atmel AVR XMEGA devices, http://www.atmel.com/dyn/resources/prod_documents/doc8267.pdf

[AT2] Innovative Techniques for Extremely Low Power Consumption with 8-bit Microcontrollers, http://www.atmel.com/dyn/resources/prod_documents/doc7903.pdf

UTILISATION DE LA CARTE KNJN / DRAGON PCI

par Pierre Ficheux
[pierre.ficheux@openwide.fr]

Les FPGA (Field Programmable Gate Array) sont de plus en plus présents dans les systèmes embarqués modernes. Ils permettent de combiner les performances du matériel avec la souplesse de programmation du logiciel. Dans cet article, nous allons présenter un produit développé par la société californienne KNJN (http://www.knjn.com). Cette carte d'interface générique permet de construire des fonctions applicatives dédiées grâce au FPGA Xilinx Spartan 2 présent sur la carte. Au fur et à mesure du déroulement de l'article, nous verrons comment mettre en place un environnement de développement GNU/Linux alors que le constructeur fournit uniquement un support Windows.

1 Introduction

Dans le n° 4 d'*Open Silicium*, notre camarade Yann Guidon a présenté l'approche matérielle de l'utilisation d'un FPGA Actel/Microsemi. Il existe également des cartes de développement incluant à la fois un processeur ARM et un FPGA (cartes Armadeus).

La société KNJN (http://www.knjn.com) conçoit de nombreux produits utilisant des FPGA, les interfaces proposées partant du simple port RS232 (carte Pluto) jusqu'au bus PCI express, certaines cartes pouvant disposer de plusieurs interfaces. Ces cartes sont vendues uniquement sur Internet depuis les USA ou l'Europe, le stock européen étant localisé à Paris :-) Le fondateur de KNJN (Jean Nicolle) est français et c'est un spécialiste de longue date des FPGA.

Dans le cas présent, la carte KNJN Dragon PCI dispose de diverses interfaces standards :

- une interface PCI ;
- une interface USB fournie par un composant AN2131 (Cypress, EZ-USB) ;
- une interface Ethernet RJ45 (à installer) ;
- un connecteur permettant de relier un afficheur LCD (à installer) ;
- une interface I2C.

La figure 1, page suivante, montre la carte Dragon PCI équipée de l'adaptateur Ethernet (à gauche) et du connecteur LCD (en haut à droite).

Il existe de nombreux développements autour de cette carte sur le site http://ww.fpga4fun.com. Cependant, ces derniers sont en majorité basés sur l'utilisation de l'environnement Windows. KNJN fournit un pilote PCI Linux dans le kit de développement vendu, mais ce pilote est très ancien et ne fonctionne absolument pas avec les noyaux récents car prévu pour un noyau 2.6.12.

Suite à l'acquisition il y a quelques mois d'une carte Dragon PCI, nous avons mis en œuvre le développement de pilotes Linux pour plusieurs des interfaces disponibles, en l'occurrence :

- un pilote PCI en mode caractère ;
- un pilote PCI pour l'extension temps réel Xenomai du noyau Linux (RTDM pour *Real Time Driver Model*) ;
- un pilote USB en mode caractère permettant le pilotage des LED de la carte ainsi que de l'afficheur LCD ;
- une procédure de test de l'interface réseau.

Des pilotes réseau (network driver) pour les interfaces PCI et USB (en cours de développement).

Les exemples de pilotes RTDM et réseau ne seront pas traités dans le présent article car ils nécessitent des présentations détaillées qui seront abordées dans une future publication.

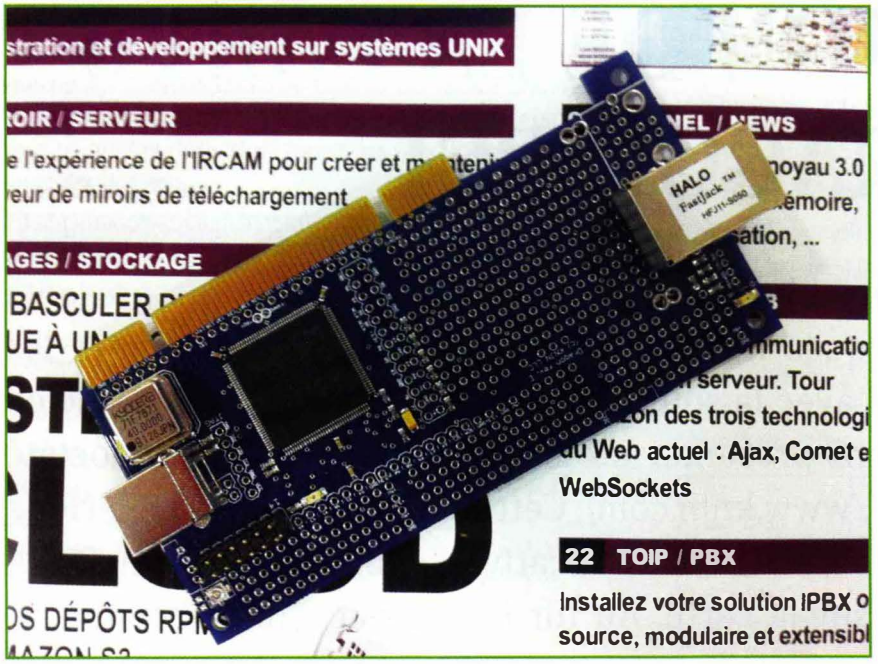


Figure 1 : La carte Dragon PCI

Suite à ces développements, le constructeur a mis en place sur son site un lien permettant d'accéder au dépôt des travaux réalisés, ces derniers étant disponibles sur le portail Github à l'URL https://github.com/pficheux/KNJN.

REMARQUE : Dans cet article, nous ferons référence soit au kit de développement fourni par KNJN (archive **DragonStartupKit.zip**), soit aux travaux réalisés par l'auteur et disponibles sur Github.

2 Principe de fonctionnement de la carte

La carte étant équipée d'un FPGA, la mise en place d'une fonction donnée passe par le développement de deux parties distinctes :

1. La partie matérielle ou *design* installée sur le FPGA. Cette partie est écrite en langage VHDL ou en Verilog. On produit ensuite un binaire **.bit** installé sur la carte par un programme spécialisé fourni.
2. Le pilote dépendant du système d'exploitation.

Le kit de développement fournit un certain nombre de designs déjà prêts (en binaire **.bit** et en Verilog **.v**) ainsi que des programmes de test dont la majorité - hormis le pilote PCI Linux - sont destinés à l'environnement Windows. Sachant que nous ne sommes pas (encore) des experts en VHDL ni en Verilog, nous avons légèrement adapté les designs à notre besoin, le gros du travail ayant été réalisé sur la partie système d'exploitation, en l'occurrence des pilotes et des programmes de test sous Linux.

La documentation de la carte est disponible sur le site KNJN ou dans le kit de développement sur le répertoire **DragonStartupKit/Documentation**.

REMARQUE : Le constructeur étant américain, les exemples sont fournis en Verilog (proche du C), langage de programmation plus répandu que le VHDL (proche de l'Ada) de ce côté de l'Atlantique. Les spécialistes pourront toujours convertir les sources en VHDL s'ils le souhaitent.

3 Un premier test d'utilisation

On peut facilement valider le fonctionnement de la carte en chargeant un exemple de design déjà compilé disponible dans le répertoire **DragonStartupKit/Sample Files – FPGA** du kit de développement. Ces exemples très simples permettent de piloter les trois LED présentes sur la carte. Attention, il faut choisir les fichiers adaptés au FPGA de la carte Dragon, disposant de 100000 portes logiques, comme **ledglow.100K.bit**.

La carte est alimentée par le connecteur USB et c'est cette même connexion qui permet de charger le fichier **.bit** en utilisant le programme (Windows) **FPGAconf.exe** livré avec le kit. Une solution simple est d'utiliser une image Windows virtuelle exécutée dans VirtualBox ou VMware player. Bien entendu, lors de la première connexion de la carte, il est nécessaire d'installer le pilote USB (Windows également) fourni dans le répertoire **DragonStartupKit/USB driver**.

Le programme de configuration précité permet de charger le **.bit** dans la mémoire vive de la carte ou de l'installer dans la PROM de démarrage du FPGA. Dans ce dernier cas, il faut débrancher/brancher la carte du bus USB pour constater le résultat. La figure 2, page suivante décrit l'installation triviale d'un programme de test avec l'outil *FPGA configurator*.

4 Moi je n'aime pas Windows :)

Pour les allergiques à Windows, il est possible de charger le fichier **.bit** en utilisant l'outil **dragon-ctrl**, développé par un utilisateur de la carte Dragon (Nicolas Noble). Les sources sont disponibles sur le dépôt Github, dans le répertoire **Dragon_PCI/USB/dragon-ctrl**. Avant de charger le fichier,

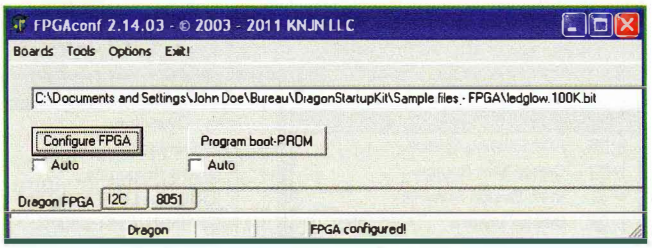


Figure 2 : Utilisation de l'outil de configuration

il est nécessaire d'exécuter un nouveau *firmware* (**firmware.hex**) fourni avec les sources de **dragon-ctrl**. On utilise donc les commandes suivantes :

```
# dragon-ctrl -i firmware.hex reboot
# dragon-ctrl -b ledglow.100K.bit program
```

Après exécution des commandes, on doit constater le clignotement des LED. Cependant, il n'est malheureusement pas possible d'écrire le fichier **.bit** dans la PROM de démarrage en utilisant cette commande et nous devrons utiliser pour cela la procédure Windows décrite précédemment.

5 Installation et utilisation de l'environnement ISE

Contrairement aux premiers exemples, les versions binaires des exemples donnés dans les répertoires **Projects** ne sont pas fournis. Pour chaque exemple, il est nécessaire d'utiliser l'outil ISE développé par Xilinx pour créer un projet et produire le fichier **.bit**. Cet outil graphique est disponible pour Windows, mais également pour Linux, sur le site de Xilinx. Pour la carte Dragon, équipée d'un FPGA un peu ancien, il est nécessaire d'installer ISE 10.1.03 dans sa version gratuite nommée *WebPack*. L'installation nécessite d'obtenir une clé (gratuite) auprès de Xilinx et la procédure est assez lourde bien qu'assez bien documentée sur le site de Xilinx (voir bibliographie).

ISE fournit un grand nombre de fonctions pour la création du design (édition simulation, ...). L'article étant focalisé sur la partie logicielle, nous nous bornerons à décrire rapidement la procédure de création du projet et de compilation du design, cette procédure étant applicable à tous les designs d'exemple. Les projets utilisés pour les développements sont dans le répertoire **ISE** du dépôt Git.

À titre d'exemple, nous pouvons créer un projet à partir du contenu du répertoire **DragonStartupKit/Projects - USB/IO8**, soit **USB_IO8.v** et **USB_IO8.ucf**. Le fichier **.v** correspond au fichier source Verilog du design et le fichier **.ucf** correspond à l'affectation physique des entrées/sorties (bornes ou pins) utilisées.

Cet exemple permet simplement de copier des données 8 bits du bus USB vers le FPGA. Après une légère adaptation du code Verilog, on peut piloter les trois LED en associant l'état de chaque LED aux bits de poids faible de l'octet reçu du bus USB.

```
assign LED[0] = USBdata[0];
assign LED[1] = USBdata[1];
assign LED[2] = USBdata[2];
```

Nous avons ajouté les trois lignes suivantes au fichier **.ucf** afin d'indiquer la prise en compte des LED dans le fichier Verilog.

```
NET "LED<0>"  LOC = "P27"  ;
NET "LED<1>"  LOC = "P51"  ;
NET "LED<2>"  LOC = "P75"  ;
```

L'extrait présenté indique que le contenu du tableau LED (3 éléments) du fichier Verilog correspond respectivement aux bornes 27, 51 et 75 décrites au paragraphe 3.1 de la documentation de la carte et repérables sur le schéma mécanique et le schéma électronique.

En premier lieu, on doit créer un nouveau projet en utilisant l'option ***File*** puis ***New Project***. À l'apparition de la boîte de dialogue, on donne le nom du projet, soit *usbio8* dans notre cas.

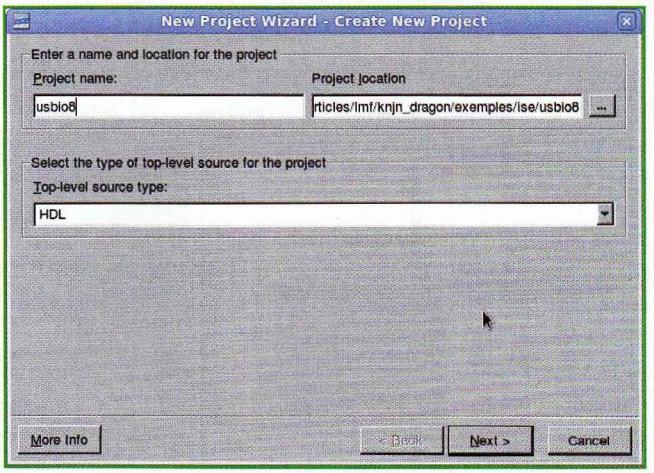


Figure 3 : Création du projet dans ISE

On clique sur ***Next*** jusqu'à obtenir la fenêtre de résumé décrite ci-dessous. On note le type de FPGA utilisé (xc2s100).

Une fois le projet créé, on doit y ajouter les deux fichiers source **.v** et **.ucf**. Pour cela, on utilise la fonction ***Add Existing Source*** disponible dans la fenêtre de gauche (***Processes***). Lorsque les deux fichiers sont sélectionnés, l'arborescence apparaît dans la fenêtre de droite (***Sources***). Les différentes actions possibles apparaissent dans la fenêtre ***Processes***.

On peut alors lancer la compilation un cliquant avec le bouton droit la fonction ***Generate Programming File*** dans la

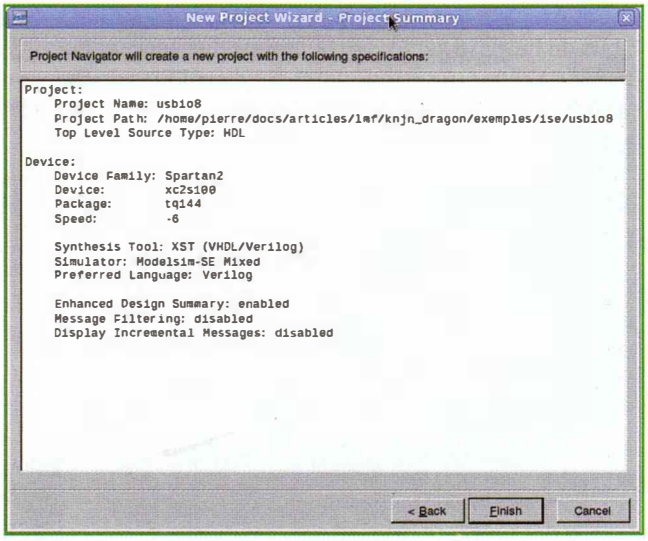


Figure 4 : Résumé du projet

fenêtre de droite, ce qui fait apparaître un menu contextuel proposant la fonction ***Run*** qui déroule les différentes étapes de la compilation.

Si la compilation est correcte, les trois étapes ***Synthesize – XST, Implement Design*** et ***Generate Programming File*** doivent être cochées en vert à la fin de la compilation, comme décrit sur la figure ci-dessous. Le fichier **`USB_IO8.bit`** est alors disponible sur le répertoire du projet.

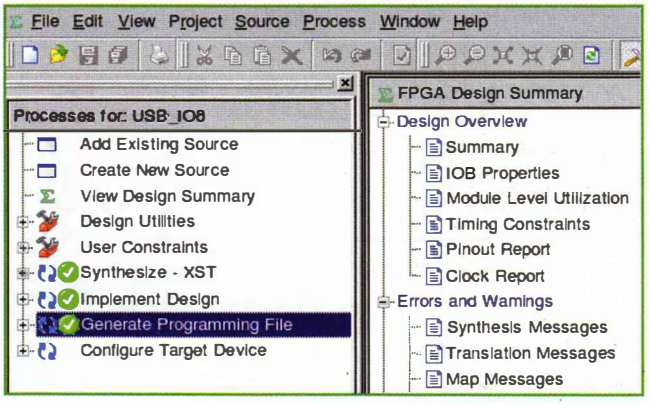


Figure 6 : Fin de la génération du design

REMARQUE : Ce projet correspond en fait au répertoire **`ISE/myusb`** dans le dépôt Github.

6 Développement d'un pilote USB

Comme nous l'avons vu, le constructeur fournit dans le répertoire **`DragonStartupKit/Projects - USB`** des exemples de designs permettant d'exploiter l'interface USB. Les exemples permettent de piloter les LED ou un afficheur LCD (non fourni). Le principe de l'interface USB est décrit

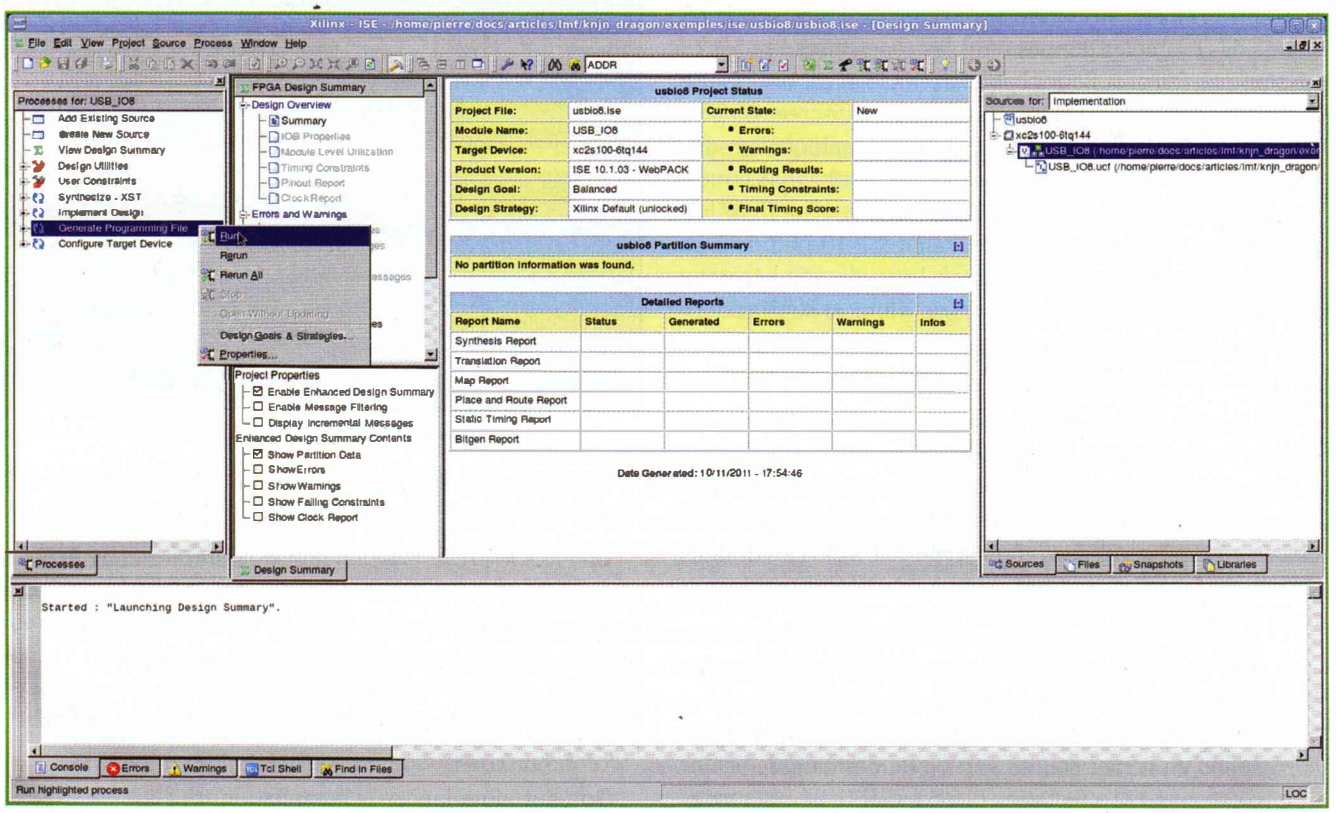


Figure 5 : Lancement de la génération du design

au paragraphe 4.2 de la documentation de la carte, en particulier le protocole de communication entre le contrôleur USB Cypress AN2131 et le FPGA.

6.1 Pilotage des LED

Le design du répertoire **IO8** permet d'accéder à l'état des LED en envoyant un simple octet sur le bus USB. Les 3 LED disponibles (LED1, LED2, LED3) correspondant aux 3 bits de poids faible de l'octet envoyé.

Le design du répertoire **USB_reg_banks** permet de créer 3 registres de 16 bits dans le FPGA. L'accès aux registres s'effectue par l'envoi d'un message USB correspondant à 2 octets d'adresse de registre puis 2 octets de données (état des LED). Les adresses définies dans le design sont 0x0000, 0x0002, 0x1000.

6.2 Pilotage d'un afficheur LCD

Le design du répertoire **LCD_Text_HDL** permet de piloter un LCD compatible avec le standard HD44780. Il est possible d'acheter le LCD sur le site KNJN, mais il est tout à fait standard et disponible chez n'importe quel revendeur d'électronique. Le branchement à la carte est décrit sur la figure ci-dessous en utilisant une simple nappe. Le LCD choisi utilise les 16 broches du HD44780, mais le connecteur de la carte n'en fournit que 14 car les broches 15 et 16 concernant le rétro-éclairage ne sont pas présentes. Il est donc important de brancher les 14 premières broches de la nappe. Sur la photo, le fil rouge (broche 1) correspond à la masse du LCD à relier à la broche 1 du connecteur de la carte (côté USB). Lorsque l'on branche le LCD, il doit donc être alimenté si la connexion est correcte (affichage des pavés).

Un programme C pour Windows est fourni dans **LCD_Text_C**, mais nous fournirons au paragraphe suivant la fonction équivalente en utilisant un pilote Linux. Il est cependant possible de compiler les exemples Windows fournis par KNJN en utilisant un compilateur croisé Linux/Windows tel que MinGW32 (http://www.mingw.org). Ce compilateur existe sous forme de paquetage pré-compilé dans toutes les distributions Linux récentes.

```
$ rpm -qf /usr/bin/i686-pc-mingw32-gcc
mingw32-gcc-4.4.2-2.fc13.i686
```

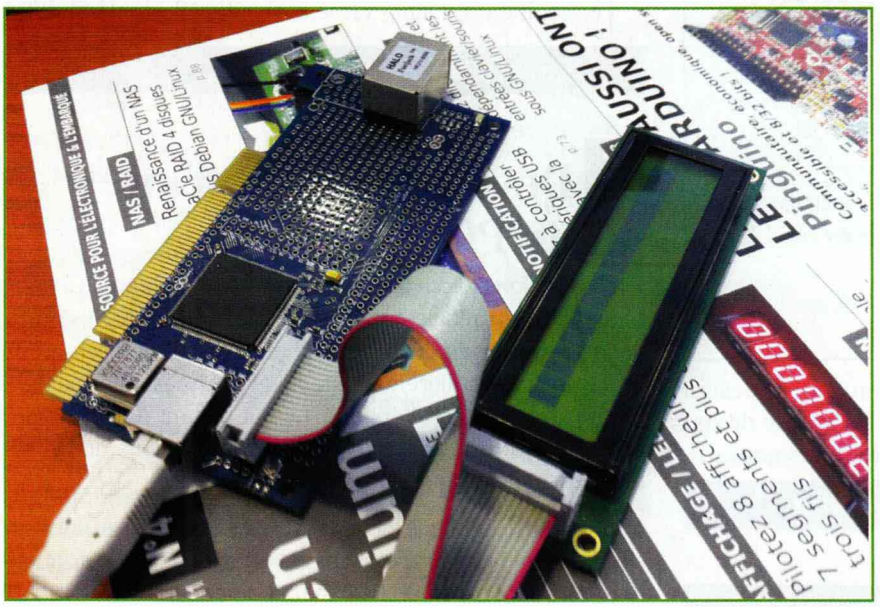


Figure 7 : Branchement du LCD

L'exemple de **Makefile** ci-dessous permet de compiler les exemples et il est bien entendu adaptable.

```
CC=  i686-pc-mingw32-gcc

PROGS= USB_IOB.exe USB_regs.exe LCD_Text.exe

all: $(PROGS)

clean:
  rm -rf *~ $(PROGS)

%.exe : %.c
  $(CC) -o $@ $<
```

6.3 Développement et utilisation du pilote

La compréhension du paragraphe nécessite une connaissance du fonctionnement du bus USB. Pour cela, on pourra se rapporter à la bibliographie et à l'article du même auteur décrivant le développement d'un pilote USB paru dans le magazine *GLMF* n° 101.

Le dialogue avec le FPGA par le bus USB utilise le *endpoint* numéro 2. Les messages qui transitent sont de type bulk. Le pilote développé est compatible avec les trois designs USB proposés dans le kit de développement à la fois pour les LED et l'afficheur LCD.

REMARQUE : comme l'a très justement fait remarquer Denis Bodor dans le n° 4 d'*Open Silicium*, l'utilisation de la libusb et non d'un pilote noyau Linux a quelques avantages tant au niveau de la facilité de programmation que de la portabilité. Malheureusement, et pour une raison non élucidée (indépendante de la mauvaise foi de l'auteur), nous n'avons pas pu faire fonctionner correctement un exemple à base de libusb alors que le module noyau fonctionne parfaitement.

Le pilote noyau développé suit les règles classiques du genre et nous renvoyons le lecteur à l'étude du code source disponible sur le répertoire **Dragon_PCI/USB/driver/dragon_usb** du dépôt Github.

Une fois le design installé PUIS le module noyau chargé (par **insmod**), on

peut communiquer avec la carte en utilisant un fichier virtuel **/sys/bus/usb/drivers/dragon_usb/*/ledmask** ou l'entrée **/dev/dragon_usb0** tous deux créés par le pilote. Il suffit d'écrire le code hexadécimal de l'octet à envoyer au FPGA sous la forme :

```
# echo -n 61 > /dev/dragon_usb0   # 4 octets maximum
```

ou :

```
# echo -n \'a > /dev/dragon_usb0  # 1 seul caractère à la fois !
```

Ce qui dans les deux cas envoie le code de la lettre « a » au FPGA. Dans le cas du code hexadécimal, on peut envoyer au maximum 4 octets. Le pilote définit une structure de donnée privée très simple pour le périphérique sachant que le seul paramètre est le masque d'état des LED sur 4 octets.

```
struct usb_dragon_usb {
  struct usb_device *udev;
  int ledmask; // we can send up to 32 bits
};
```

La fonction ci-après effectue l'envoi des données en utilisant la fonction **usb_bulk_msg()**.

```
int dragon_set_ledmask (struct usb_dragon_usb *mydev, const char *buf)
{
  int ret = 0, l, n;
  char c;

  // Remove \n if any
  l = strlen(buf);
  l = (*(buf+l-1) == '\n' ? l-1 : l);

  // Get value from buf
  if (*buf == '\'') {
    // 'a => 0x61
    sscanf (buf+1, "%c", &c);
    mydev->ledmask = c;
    l--;
  }
  else
    sscanf (buf, "%x", &(mydev->ledmask));

  // Handle x instead of 0x
  n = (l >= 2 ? l/2 : l);

  // Send bulk message to EP 2
  ret = usb_bulk_msg  (mydev->udev, usb_sndctrlpipe(mydev->udev, 2),
  &(mydev->ledmask), n, &l, 2 * HZ);

  return ret;
}
```

L'utilisation est un peu différente suivant le design chargé dans le FPGA. Pour allumer la LED3 avec le design **USB_IO8.bit**, on utilisera la commande shell suivante :

```
# echo -n 4 > /sys/bus/usb/drivers/dragon_usb/6-1:1.0/ledmask
```

ou :

```
# echo -n 4 > /dev/dragon_usb0
```

La même action avec le design **USB_reg_banks.bit** s'effectuera avec la commande suivante. Le premier mot long correspond au masque, le deuxième à l'adresse du registre.

```
# echo -n 80001000 > /sys/bus/usb/drivers/dragon_usb/6-1:1.0/ledmask
```

ou :

```
# echo -n 80001000 > /dev/dragon_usb0
```

Dans le cas du design **LCD_Text.bit**, on doit envoyer les commandes de pilotage du LCD au format HD44780 vers le fichier virtuel ou l'entrée dans **/dev**. Le script présenté ci-après permet d'afficher une chaîne de caractères sur le LCD en redirigeant la sortie vers **/dev/dragon_usb0**.

```
#!/bin/sh
#
# Send a string to Dragon board through USB Linux driver
#
if [ "$1" != "" ]; then
    STR="$1"
else
    STR="Hello Linux !"
fi

# Init LCD
echo -n 3800
echo -n 0f00
echo -n 0100
sleep 2

# Send string
L=${#STR}
i=0
while [ $i -lt $L ]; do
    echo -n "'${STR:$i:1}"
    i=$(expr $i + 1)
done
```

Le résultat est visible sur le LCD suite à la commande :

```
# ./lcd_test.sh > /dev/dragon_usb0
```

7 Développement d'un pilote PCI

Nous avions décrit le développement d'un pilote PCI pour Linux dans un article publié dans *GLMF* en 2002 (!), donc à l'époque du noyau 2.4. L'API PCI du noyau Linux est - pour une fois - particulièrement stable et n'a quasiment pas été modifiée depuis cette époque. Le but n'étant pas ici de décrire la structure complète d'un pilote PCI, nous nous bornerons à évoquer quelques rappels.

REMARQUE : Même lors de l'utilisation de la carte dans un slot PCI, celle-ci doit toujours être alimentée par le connecteur USB, sauf si l'on réalise une modification matérielle décrite au paragraphe 6.3 de la documentation KNJN.

7.1 Rappels sur le PCI

Le bus PCI a été introduit par Intel en 1992. Depuis, il a connu de nombreuses évolutions comme le *Mini PCI* ou le *PCI Express*. L'espace mémoire de configuration d'un périphérique PCI (256 octets) est organisé de manière standard pour les 64 premiers octets. La figure 8 décrit la répartition des registres de configuration.

Les registres les plus utiles concernent :

1. le *Vendor ID* et *Device ID* ;
2. les *Base Address Registers* 0 à 5 (BAR) ;
3. les registres *IRQ Line* et *IRQ Pin*.

Les premiers concernent l'identification PCI du périphérique. La détection de la présence du périphérique correspondant est automatique et la méthode d'initialisation **`probe()`** est appelée si la carte est détectée sur le bus. Si la carte n'est plus présente ou désactivée, la méthode **`remove()`** est appelée. Ce principe a été repris par l'API USB avec les méthodes **`probe()`** et **`disconnect()`**.

Les BAR0 à BAR5 correspondent à des pointeurs vers des zones d'adressage mémoire de la carte, ces zones pouvant être ou non accessibles depuis le système d'exploitation. Côté Linux, la BAR est caractérisée par un flag défini dans le fichier **`linux/ioport.h`**. Les plus importants sont :

- **`IORESOURCE_IO`** pour un accès PMIO (*Port Mapped I/O*) ;
- **`IORESOURCE_MEM`** pour un accès MMIO (*Memory Mapped I/O*).

La principale différence est l'utilisation d'instructions spéciales **`IN`** et **`OUT`** du CPU – généralement Intel - dans le cas d'un port PMIO (**`inb/w/l`** et **`outb/w/l`**).

Le niveau d'interruption utilisé par le périphérique PCI est donné dans le registre IRQ Line si le registre IRQ Pin est différent de 0, une valeur nulle signifiant que le périphérique ne traite pas d'interruption. Les valeurs de BAR et du niveau d'interruption son dynamiques et affectées à l'initialisation par le BIOS PCI.

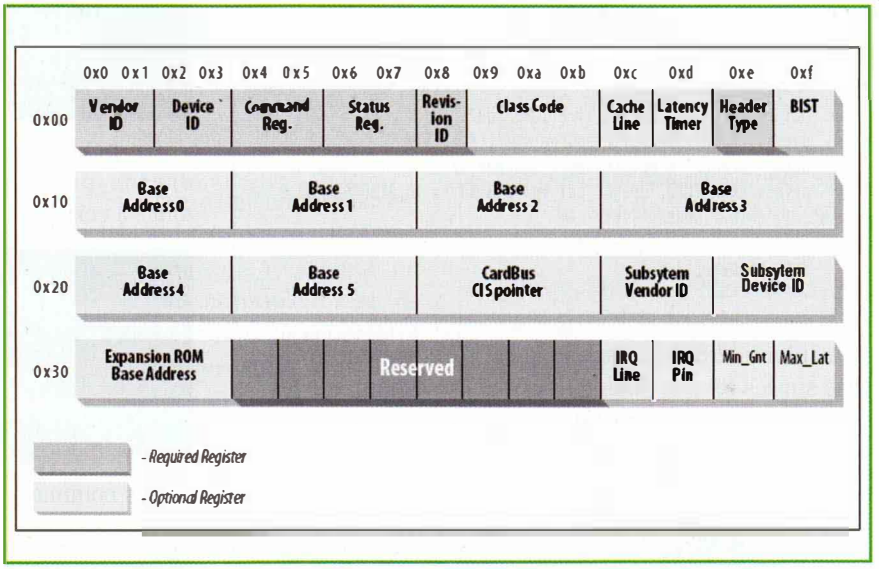


Figure 8 : Registres de configuration PCI

7.2 Développement et utilisation du pilote

Le design d'exemple **`DragonStartupKit/Projects - PCI P&P/HDL/PCI_PnP.100K.bit`** définit le BAR0 comme une zone d'accès PMIO et le BAR1 comme une zone d'accès MMIO. Les sources des pilotes Linux développés sont disponibles dans les répertoires **`Dragon_PCI/PCI/IO`** et **`Dragon_PCI/PCI/MEM`** de l'arborescence Git. Ces pilotes sont de type caractère et permettent simplement d'échanger des données entre Linux et les BAR0/BAR1 par des appels système **`read()`** et **`write()`**.

La principale différence entre les deux pilotes est la nécessité d'utiliser la fonction **`ioremap()`** dans le cas de l'accès MMIO lors de la détection par la méthode **`probe()`**. Cette fonction permet d'obtenir une adresse dans l'espace du noyau à partir de l'adresse physique sur le bus PCI.

```
if (pci_resource_flags(dev, i) & IORESOURCE_MEM) {
  if (pci_resource_flags(dev, i) & IORESOURCE_CACHEABLE) {
    printk (KERN_INFO "cacheable ! \n");
    data->mmio[i] = ioremap(pci_resource_start(dev, i), pci_resource_len(dev, i));
  }
  else {
    printk (KERN_INFO "NOT cacheable ! \n");
    data->mmio[i] = ioremap_nocache(pci_resource_start(dev, i), pci_resource_len(dev, i));
  }

  if (data->mmio[i] == NULL) {
    printk(KERN_WARNING "dragon_pci_mem: unable to remap I/O memory\n");

    ret = -ENOMEM;
    goto cleanup_ioremap;
  }

  data->mmio_len[i] = pci_resource_len(dev, i);

  printk(KERN_INFO "dragon_pci_mem: I/O memory has been remaped at %#08x\n", (u32)data->mmio[i]);
 } else {
  data->mmio[i] = 0;
 }
}
```

Dans le cas de l'accès PMIO, on peut stocker directement la valeur de l'adresse.

```
  if (pci_resource_flags(dev, i) & IORESOURCE_IO) {
      data->iobase[i] = pci_resource_start(dev, i);
      data->iolen[i] = pci_resource_len(dev, i);
      printk(KERN_INFO "dragon_pci_io: BAR %d is IO_RESOURCE_IO @
%x!\n", i, data->iobase[i]);
    } else {
      data->iobase[i] = 0;
    }
```

Au niveau des accès lecture et écriture, le mode PMIO est basé sur les fonctions **inl()** et **outl()**, sachant que les accès à la carte sont en 32 bits. Voici ci-après un extrait de la méthode **write()** en PMIO.

```
    if (copy_from_user(kbuf, buf, real))
      return -EFAULT;

    port = data->iobase[i] + *ppos;

    for (j = 0 ; j < real ; j += sizeof(long))
      outl(*((unsigned long *)(kbuf + j)), port + j);
```

Dans le cas de l'accès MMIO, on utilise un simple pointeur.

```
  if (real)
    if (copy_from_user((void*)data->mmio[bank] + (int)*ppos, (void
__user *)buf , real))
      return -EFAULT;

  *ppos += real;
```

L'installation de la fonction de traitement d'interruption est identique pour les deux modes d'accès, en testant la présence de l'interruption avec le registre **PCI_INTERRUPT_PIN**.

```
 pci_read_config_byte (dev, PCI_INTERRUPT_PIN, &mypin);
  if (mypin) {
    ret = request_irq(dev->irq, dragon_pci_mem_irq_handler, IRQF_
SHARED, "dragon_pci_mem", data);

    ...
  }
```

L'insertion de l'un ou l'autre des modules (mais pas les deux à la fois !) indique la détection de la carte dans les traces du noyau.

```
dragon_pci_mem: found 100:0
dragon_pci_mem: using major 254 and minor 0 for this device
dragon_pci_mem: BAR 0 (0x00ec00-0x00ec3f), len=64, flags=0x020101
dragon_pci_mem: BAR 1 (0xd6000000-0xd600ffff), len=65536,
flags=0x020200
pci: I/O memory has been remaped at 0xcf8a0000
```

Le majeur dynamique détecté correspond ici à 254, on peut donc créer l'entrée dans **/dev** avec la commande **mknod**.

```
# grep dragon_pci /proc/devices
254 dragon_pci_io
# mknod /dev/dragon c 254 0
```

On peut ensuite tester les accès avec le programme **dragon_pci_test** fourni, en précisant le nombre de mots longs à écrire.

```
# dragon_pci_test /dev/dragon 2
buf[0] = 0x00000000
buf[1] = 0x11111111
Wrote 8 chars
Read 8 chars
buf[0] = 0x00000000
buf[1] = 0x11111111
```

Les commandes **hexdump** ou **dd** permettent de vérifier le résultat.

```
# hexdump -C /dev/dragon  | more
00000000  00 00 00 00 11 11 11 11  22 22 22 22 33 33 33 33
|........"""3333|
...

# od -x /dev/dragon  | more
0000000 0000 0000 1111 1111 2222 2222 3333 3333
0000020 4444 4444 5555 5555 6666 6666 7777 7777
```

On peut également copier le contenu d'un fichier sur la BAR1 en utilisant la commande **cat** et un fichier de données.

```
# cat meduse.txt > /dev/dragon
```

Lors de la lecture, on constate qu'à partir de l'adresse 0x80, les données sont répétées. Si l'on écrit plus de 128 octets, les premières données sont écrasées.

```
# hexdump -C /dev/dragon

00000000 4e 6f 6e 2c 20 63 65 20 6e 27 65 74 61 69 74 20 |Non, ce n'etait |
00000010 70 61 73 20 6c 65 20 72 61 64 65 61 75 20 44 65 |pas le radeau De|
00000020 20 6c 61 20 4d 65 64 75 73 65 2c 20 63 65 20 62 | la Meduse, ce b|
00000030 61 74 65 61 75 20 71 75 27 6f 6e 20 73 65 20 6c |ateau qu'on se l|
00000040 65 20 64 69 73 65 20 61 75 20 66 6f 6e 64 20 64 |e dise au fond d|
00000050 65 73 20 70 6f 72 74 73 20 64 69 73 65 20 61 75 |es ports dise au|
00000060 20 66 6f 6e 64 20 64 65 73 20 70 6f 72 74 73 2e | fond des ports.|
00000070 20 49 6c 20 6e 61 76 69 67 75 61 69 74 20 65 6e | Il naviguait en|
00000080 4e 6f 6e 2c 20 63 65 20 6e 27 65 74 61 69 74 20 |Non, ce n'etait |
00000090 70 61 73 20 6c 65 20 72 61 64 65 61 75 20 44 65 |pas le radeau De|
000000a0 20 6c 61 20 4d 65 64 75 73 65 2c 20 63 65 20 62 | la Meduse, ce b|
000000b0 61 74 65 61 75 20 71 75 27 6f 6e 20 73 65 20 6c |ateau qu'on se l|
...
```

Le problème est dû à la conception du design d'exemple dans lequel on utilise uniquement 128 octets (32 mots longs) dans la mémoire du FPGA (block RAM). Pour modifier cela, nous nous sommes inspirés de l'article disponible sur http://danstrother.com/2010/09/11/inferring-rams-in-fpgas. Le nouveau design définit une constante **ADDR**, le nombre de mots longs alloués étant égal à 2^ADDR. La nouvelle valeur

de la constante est 8, soit 256 mots longs (1024 octets). Le nouveau fichier Verilog est disponible sur le dépôt Github dans le répertoire **Dragon_PCI/ISE/mypci**.

8 Test Ethernet

Moyennant la modique somme de 8,95€, il est possible d'obtenir l'extension Ethernet pour la carte Dragon PCI (le connecteur RJ45 et l'oscillateur 40 Mhz). Lors de l'installation de l'oscillateur, on doit prendre garde à l'orientation – point noir en haut à droite - sous peine de le rendre inutilisable. Le kit de développement fournit un exemple de design Ethernet dans le répertoire **DragonStartupKit/Projects\ -\ Ethernet/HDL**. Le design définit de manière statique les adresses IP et MAC de la carte Dragon PCI et du PC de test. Il émet une trame UDP environ toutes les deux secondes. Si le PC envoie une trame UDP vers la carte, celle-ci devient la trame envoyée régulièrement par la carte. Le port UDP 1024 est utilisé par défaut.

Les répertoires **UDP_receive** et **UDP_send** correspondent aux exemples (Windows) d'émission et de réception de trames UDP. Dans le cas d'un test sous Linux, il n'est pas nécessaire d'écrire un programme et nous pouvons utiliser la commande **nc** (netcat).

En premier lieu, il faut ajouter l'adresse de la carte à la table ARP du PC (Linux) de test.

```
# arp -s 192.168.3.40 16:FD:22:04:B1:61
# arp -a
...
? (192.168.3.40) at 16:fd:22:04:b1:61 [ether] PERM on eth0
```

On peut alors envoyer une trame via **nc** par la commande :

```
$ echo 'hello dragon' | nc -u  192.168.3.40 1024
```

La carte doit alors renvoyer le même message toutes les deux secondes, ce que l'on peut vérifier avec la commande **tcpdump** ou l'application graphique Wireshark.

```
# tcpdump -i eth0 -A

17:05:10.492766 IP opti760pf > 192.168.3.40: ICMP opti760pf udp port 1024 unreachable, length 54
E..J....@."....m...(..q.....E.............(...m........hello dragon
.....
17:05:11.331437 IP 192.168.3.40.1024 > opti760pf.1024: UDP, length 18
E.............(...m........hello dragon
```

Conclusion et perspective

Les cartes KNJN nous ont permis de mettre en place une initiation à l'utilisation des FPGA (programmation Verilog et VHDL) ainsi qu'au développement des pilotes associés. Malgré l'absence de support Linux officiel, nous avons toujours eu d'excellents rapports avec la société KNJN, avec des réponses par courrier électronique quasi immédiates modulo le décalage horaire.

À l'heure actuelle, nous travaillons sur un projet de construction d'une maquette de carte Ethernet ainsi que la réalisation du pilote Linux network device associé. Ce travail peut être également réalisé sur la carte Xylo qui est moins onéreuse si on se contente de l'interface USB. ■

Bibliographie

- Le site de la société KNJN sur http://www.knjn.com
- Les pilotes Linux développés par l'auteur sur https://github.com/pficheux/KNJN
- Manipulation d'un block RAM en Verilog ou VHDL sur http://dans-trother.com/2010/09/11/inferring-rams-in-fpgas
- Outil ISE version WebPack de Xilininx sur http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm
- Prise en main de l'outil ISE sur http://www.fpga4fun.com/ISE-QuickStart.html
- Patrice Nouel, grand spécialiste VHDL à l'ENSEIRB sur http://uuu.enseirb.fr/~nouel
- Tutoriel Verilog sur http://www.asic-world.com/verilog/veritut.html
- Développement de pilote PCI sous Linux (2.4) sur http://pficheux.free.fr/articles/lmf/pci
- Linux Device Drivers version 3 (PCI) sur http://lwn.net/images/pdf/LDD3/ch12.pdf
- Linux Device Drivers version 3 (USB) sur http://lwn.net/images/pdf/LDD3/ch13.pdf
- Essential Linux Device Drivers sur http://elinuxdd.com
- Le site de la société Armadeus sur http://www.armadeus.com
- Exemple d'afficheur LCD HD44780 utilisable avec la carte Dragon PCI sur http://www.selectronic.fr/afficheur-2x20-caracteres.html
- Article sur l'utilisation du LCD HD44780 sur http://www.icehw.net/article.php?id=104
- Utilisation d'un afficheur LCD avec un carte Dragon sur http://www.knjn.com/FPGA-PCI-LCD.html

LE BLUETOOTH POUR VOS MONTAGES ET VOS SYSTÈMES EMBARQUÉS

par Denis Bodor

Il existe bon nombre de systèmes de communication sans fil permettant d'asservir un montage, de communiquer avec des sondes ou d'établir un lien avec un système embarqué. Celui qui nous intéresse aujourd'hui est le Bluetooth car, en ce qui concerne la communication à courte distance, c'est la technique la plus universelle et la plus simple à mettre en œuvre. Comme vous allez le constater, la disponibilité du matériel permet à chacun de créer des systèmes communiquant à peu de frais tout en conservant une certaine souplesse d'utilisation.

1 Pourquoi le Bluetooth ?

Lorsqu'on s'intéresse au domaine ou que l'on cherche un moyen de faire communiquer de manière non filaire un montage, à base de microcontrôleur, par exemple, plusieurs options se présentent à nous :

- Wi-Fi : Généralement utilisé pour les ordinateurs ou systèmes équivalents, le principal désavantage de ce standard tient dans sa complexité à laquelle il faut ajouter la faible autonomie due à une consommation énergétique importante. Le Wi-Fi n'est pas conçu, de base, pour les systèmes fonctionnant sur des sources d'alimentation autonomes. Bien qu'il soit parfaitement possible d'ajouter un module Wi-Fi à vos montages, les besoins mémoire (>1 Mb) imposent l'utilisation de ressources importantes. Il existe bien entendu des modules complets accessibles en SPI, par exemple, mais cela ne fait que déplacer la complexité dans un module gourmand tout en laissant le choix au développeur d'implémenter, généralement, une pile TCP/IP quasi-complète. Notez que dans cette optique, votre smartphone EST équivalent à un ordinateur tant en termes de puissance CPU que de mémoire et d'architecture. Il est donc légitime qu'il dispose de Wi-Fi.

- Zigbee : Voici un système de communication dédié à l'embarqué et aux microcontrôleurs. Conçu dès le départ comme une telle solution, et basé sur la norme IEEE 802.15.4, le Zigbee est une norme dont les spécifications sont publiques depuis 2005. Proposant un débit bien inférieur au Wi-Fi (250Kb/s contre 11Mb/s à 330Mb/s pour le Wi-Fi), il est largement moins gourmand en ressources avec entre 4 et 32Kb pour une implémentation. L'autonomie théorique sur batterie se chiffre en années, contre moins de deux jours pour le Wi-Fi. Il semble donc tout indiqué pour notre type d'utilisation puisque, de plus, bon nombre de modules sont disponibles à la vente à un prix raisonnable. Enfin, pour en finir avec les nombreux avantages du Zigbee, il permet de mettre en œuvre un réseau maillé (*mesh*) où la fiabilité est assez élevée (en contrepartie d'un rayon d'action et d'une vitesse réduite). Pourquoi ne pas choisir le Zigbee dans ce cas ? La réponse tient en une autre question : votre [smartphone|PC|Mac|Arduino] est-il équipé en Zigbee ? Ainsi, pour développer une solution complète, cette norme est idéale. Mais lorsqu'il s'agit de faire communiquer des systèmes hétérogènes (GNU/Linux, Windows, Mac, Android, microcontrôleurs, etc.), Zigbee impose l'ajout de matériel. Si nous prenons le cas du smartphone, ceci est à la limite de l'ergonomiquement faisable (module Bluetooth ou Wi-Fi comme passerelle Zigbee ?).

- Liaison non standardisée : Il existe un grand nombre de solutions faisant usage d'un système de communication sans fil reposant sur une vaste gamme de fréquences. Financièrement très accessibles et souvent faciles à mettre en œuvre, même sur de très longues distances (jusqu'à 3 Km pour quelque 65 euros pour deux *transceivers* TTL), ces solutions présentent les mêmes limitations que le Zigbee sans offrir les mêmes avantages. L'utilisation de ce type de solutions sera alors très spécifique et dédiée à des projets devant répondre à des contraintes particulières, comme un rayon d'action élevé avec une vitesse très réduite (<1200bps).

- Bluetooth : Finalement le meilleur compromis (dans notre cas), le Bluetooth équipe déjà un grand nombre de matériel et, pour une implémentation reposant sur un besoin mémoire de quelque 250Kb et une autonomie théorique de l'ordre d'une semaine, il se présente comme le juste milieu. Le rayon d'action limité à 10-100 mètres peut être étendu à 300 mètres via l'utilisation de techniques spécifiques (antennes et matériel particulier). Enfin, côté mise en œuvre pour vos montages et sondes, le Bluetooth prendra la forme de modules communiquant via SPI ou liaison série TTL dont l'utilisation est très accessible. La majorité des microcontrôleurs modernes disposent nativement d'interfaces ou d'UART permettant ce type de communication et, dans le cas d'un montage « esclave », le code à développer se limitera à la communication et au traitement des données sans avoir à intervenir au niveau du protocole Bluetooth (association, connexion, etc.).

Le Bluetooth est donc un choix raisonnable car il est tout aussi présent que le Wi-Fi, mais plus simple et plus économique à mettre en œuvre. Les quelques articles qui vont suivre vous permettront de comprendre les éléments clés de la technologie Bluetooth, de faire connaissance avec le support Linux sous la forme de la pile Bluez, d'utiliser les outils Bluetooth en ligne de commandes, d'utiliser les périphériques Bluetooth parmi lesquels les modules Bluetooth/série et divers accessoires, de programmer vos outils et applications en C et de développer autour du support Bluetooth d'Android. Parmi les choses qui ne seront pas couvertes par ces articles, vous avez, tout particulièrement, ce qui relève de l'utilisation et de la programmation du support Bluetooth sous Windows et Mac OS X. Dans la ligne rédactionnelle du présent magazine, nous partons en effet du principe que l'ouverture est une condition importante dans le développement de solutions. En conséquence, même si la création d'un applicatif Windows aurait été intéressante du point de vue technique, les outils de développement ainsi que le support dans le système ne nous permettent pas réellement d'appréhender tout le plaisir du développement ouvert. D'autre part, si vous avez dans l'idée de développer un client propriétaire-et-fermé pour votre produit propriétaire-et-fermé et ainsi de construire un *business model* reposant sur la captivité de vos clients, vous n'avez certainement pas besoin de nous. Et dans ce cas, le présent magazine n'est peut-être pas un choix très pertinent.

Pour info

Lorsque vous utilisez une technologie sans fil à fréquence, vous reposez sur les épaules des géants. Les premières transmissions sans fil ont été réalisées par le mythique et mystérieux Nikola Tesla, juste avant Guglielmo Marconi (à qui on a attribué, à tort, l'invention de la radio avant de rendre à Tesla la paternité de l'invention (et le brevet)). Ceci est valable pour le Wi-Fi et le Bluetooth, mais également pour la télécommande de votre porte de garage et bien d'autres choses. Merci les pionniers et merci Tesla !

Maintenant que le décor est planté et que le menu est annoncé, commençons par préciser exactement de quoi nous parlons.

2 Qu'est-ce que le Bluetooth ?

Avant toutes choses et pour votre culture personnelle, sachez que le nom « Bluetooth » est inspiré du roi Harald 1er du Danemark, dit Harald à la dent bleue. Celui-ci, né en 910, fut roi à partir de 958 et mourut en 986. On suppose qu'il fut surnommé ainsi soit en raison de sa dentition gâtée (bleu signifiant à l'époque « sombre » en danois), soit pour les effets buccaux de la surconsommation de myrtilles dont il était très friand. Le logo du Bluetooth est formé de deux lettres de l'alphabet Futhark moderne, Hagall et Bjarkan, qui correspondent aux initiales du roi Harald. Voilà de quoi glisser une phrase démontrant l'étendu de votre connaissance de Wikipédia autour d'une tarte composée de petits fruits bleus/violets. Mais passons aux choses sérieuses...

Le Bluetooth est un protocole en couches qui se divise en deux grandes parties : d'une part les couches matérielles et d'autre part les couches logicielles, plus intéressantes de notre point de vue. Matériellement, deux éléments importants sont pris en charge. Nous avons ainsi la couche radio chargée de gérer les fréquences et les canaux utilisés. La bande de fréquences utilisée est divisée en 79 canaux séparés de 1Mhz, le premier (0) utilisant 2402 MHz. Le choix de la modulation de fréquence utilisée ainsi que de l'encodage des données a été fait de manière à simplifier et alléger l'implémentation du protocole. Les périphériques Bluetooth se divisent en trois classes définies en fonction de la puissance utilisée :

- Classe 1 : 100mW avec une portée théorique de 100 mètres ;
- Classe 2 : 2,5mW avec une portée théorique de 10 ou 20 mètres ;
- Classe 3 : 1mW avec une portée théorique de 1 à 3 mètres.

Le choix de votre matériel (module ou adaptateur) doit être motivé par le rapport prix/puissance. Il existe aujourd'hui peu de matériel de classe 3, mais certains revendeurs n'hésitent pas à vendre des classes 2 en annonçant des portées totalement fantaisistes. Lisez systématiquement les spécifications techniques des produits avant achat pour vous assurer de la puissance effective du matériel. Bien entendu, la puissance ne fait pas tout et la qualité générale du produit se définira également en fonction de la présence d'une antenne amovible, par exemple. Bon nombre d'adaptateurs USB de classe 1 intègrent une antenne se résumant à une simple piste sur le circuit. De par mon expérience, je vous conseille d'opter pour un matériel de marque plutôt qu'un *ersatz* chinois d'entrée de gamme. Si vos moyens vous le permettent, visez les adaptateurs Bluetooth USB industriels (30 à 50 euros).

L'autre aspect matériel du protocole se résume en la couche immédiatement au-dessus de la couche radio : la bande de base (*baseband*). Cette couche définit les adresses matérielles des périphériques, appelées BD_ADDR pour *Bluetooth Device ADDRess*. Le principe est similaire aux adresses MAC des interfaces Ethernet et elles sont codées sur 48 bits attribuées et gérées par l'*IEEE Registration Authority*. Il n'est donc pas possible, en principe, que deux périphériques possèdent la même adresse. C'est également la couche baseband qui gère les liaisons au niveau matériel, qu'elles soient synchrones et orientées connexion (SCO) ou asynchrones et sans connexion (ACL).

Les couches LC (*Link Control*) et LM (*Link Management*) complètent l'implémentation matérielle. Au dessus, se place L2CAP (*Logical Link Control and Adaptation Protocol*), qui s'assure du désassemblage/réassemblage des paquets, du multiplexage et de la QoS. Il est possible d'utiliser directement la couche L2CAP pour vos applications. Le protocole est orienté connexion, mais il est possible de gérer cette fonctionnalité en réglant le délai de retransmission. Ainsi, L2CAP peut être comparé à de l'UDP, dans un mode *best effort*, avec un court délai de retransmission. Il est également possible, avec un délai de retransmission fixé à infini, d'obtenir une communication orientée connexion mais basée sur des datagrammes (sorte de TCP batard).

Au-dessus de L2CAP se placent divers protocoles dont deux doivent être connus. Le premier est RFCOMM, orienté connexion et flux (*stream*), il s'agit principalement d'une émulation série RS-232. C'est ce protocole qui est généralement utilisé pour les modems Bluetooth, il est jugé fiable (*reliable*) car il n'y a pas de perte de données. La grande majorité des applications que nous allons devoir rencontrer dans notre domaine relèvent de la communication via RFCOMM. Notez que le protocole OBEX (*OBject EXchange*) permettant l'envoi et la réception, entre autres, de fichiers, repose sur RFCOMM. Pour compléter l'analogie, on peut dire que RFCOMM est le TCP du Bluetooth.

L'autre protocole basé sur L2CAP est SDP pour *Service Discovery Protocol*. Il permet à un appareil de s'informer sur les services proposés par un périphérique Bluetooth. SDP est relativement complexe en soi mais permet de régler un problème majeur du Bluetooth. En effet, il existe une vaste gamme d'applications possibles d'une telle connexion sans fil à courte distance. Un appareil doit pouvoir déterminer avec exactitude la manière dont il va devoir communiquer des données au périphérique. Il y a donc une phase de recherche/découverte de services qui doit aboutir sur un profil applicatif ou, du moins, un format de données à utiliser. Un périphérique peut proposer plusieurs services, comme nous allons le voir dans les exemples pratiques qui vont suivre. Ces services sont identifiés par un UUID (*Universally Unique IDentifier*) avec, pour les UUID standards, des profils associés. SDP est, il faut le savoir, également la source de la plupart des problèmes d'implémentation et d'utilisation des liaisons Bluetooth. Nous allons également en faire la démonstration, en particulier dans la partie consacrée au développement Android.

D'autres protocoles font partie de la pile Bluetooth, comme BNEP (*Bluetooth Network Encapsulation Protocol*) pour le réseau, TCP (*Telephony Control Protocol*), AVDTP (*Audio/Video Data Transport Protocol*) pour les casques sans fil, etc. Ici, nous nous concentrerons sur l'utilisation de RFCOMM, et par nécessité pratique, sur SDP.

3 Le matériel Bluetooth

Il existe une vaste gamme de périphériques Bluetooth. Comme avec bien d'autres protocoles, les rôles dans une communication se divisent en deux catégories : les esclaves et les maîtres. Sans entrer dans le détail et se basant sur les produits qui nous entourent, nous avons déjà une vaste sélection de produits à tester. Smartphone et ordinateurs (via interface Bluetooth intégrée ou ajoutée en USB, par exemple) forment la majorité des cas où le Bluetooth est utilisé côté maître. Rares sont les matériels de ce type qui n'en sont pas équipés et, même si techniquement les interfaces peuvent être utilisées comme esclave (en attente de connexion donc), c'est généralement le PC/Mac ou le smartphone qui recherche un périphérique, se renseigne sur les services qu'il propose et initie le dialogue.

Côté périphérique esclave, nous avons toutes sortes de matériels : claviers, souris, modems (souvent le smartphone en esclave), kits mains-libres, lecteurs de codes-barres, *gamepads*, gadgets divers, appareils médicaux, de soin et d'aide à la personne, etc. Parmi les périphériques qu'on est tantôt étonné puis ravi de découvrir comme utilisant le Bluetooth, on trouve les accessoires pour Sony PS3 ou encore les Wiimote de la Nintendo Wii. Dans certains cas cependant, le protocole utilisé au-dessus de la liaison Bluetooth reste propriétaire ou parfois seulement proche du standard.

FOCUS : Petit lexique du Bluetooth

- Classe : Valeur entre 1 et 3 définissant la puissance et la portée d'un périphérique ou d'une interface Bluetooth. 1 est la classe la plus intéressante (100 mètres / 100 mW 20 dBm).

- Pairage / Liaison / Association : Pour que deux appareils Bluetooth communiquent, ils doivent généralement établir une relation de confiance en partageant un secret (*shared secret*) : une clé de liaison ou *link key*. Le fait de négocier cette clé entre deux appareils est appelé pairage (*paring* ou *bonding*) ou association. Dans la négociation, c'est un code PIN qui est utilisé pour partager la clé. Certains périphériques permettent de choisir librement le code et ils doivent, tout simplement, être identiques pour les deux appareils. D'autres produits, ne disposant pas de fonctionnalités de saisie, utilisent un code PIN fixe (souvent 0000 ou 1234) que l'autre périphérique doit utiliser. Si un des périphériques (ou les deux) supprime(nt) la clé de liaison, l'association n'existe plus et une nouvelle négociation devra avoir lieu.

- SPP : *Serial Port Profile* est l'un des profils Bluetooth les plus intéressants et les plus simples à utiliser. Ce profil définit un sous-ensemble de normes permettant l'émulation d'une liaison filaire RS-232. Ce profil est directement basé sur RFCOMM. Attention de ne pas confondre avec le SPP du port parallèle (ce port est mort et enterré depuis longtemps maintenant).

- LMP : le *Link Management Protocol* est utilisé pour la gestion du lien radio entre deux périphériques Bluetooth. Ce protocole existe en plusieurs versions et est implémenté directement au niveau du contrôleur. La version du LMP est liée à la version des spécifications Bluetooth. Celles-ci sont compatibles et, de ce fait, un périphérique Bluetooth 2.1 pourra communiquer avec un périphérique 2.0 ou 1.2 dans la limite des fonctionnalités de ces versions.

- EDR : Pour *Enhanced Data Rate*, EDR apparaît pour la première fois dans la désignation « Bluetooth v2.0 + EDR ». Il n'y a pas de spécifications « Bluetooth v2.0 » en tant que telles. « + EDR » est là pour signaler une fonctionnalité supplémentaire optionnelle : le gain en vitesse. Un périphérique supportant EDR permettra un taux de transfert de l'ordre de 3 Mbit/s au niveau liaison, 2.1 Mbit/s dans la pratique. Peu de périphériques supportent uniquement les spécifications 2.0 (ou 2.1) sans EDR.

- 2.1 et SSP : Les spécifications 2.1 du standard Bluetooth apportent une fonctionnalité appelée SSP pour *Secure Simple Pairing* étendant la sécurité du pairage de périphériques tout en le simplifiant. Le mécanisme de pairage disponible avec 2.0 et inférieur est désigné sous le terme *legacy pairing*. SSP introduit une notion de cryptographie asymétrique (à clé publique) et plusieurs mécanismes d'authentification. L'un d'eux est très intéressant, l'*Out Of Band* (OOB) permet l'utilisation d'un canal de communication supplémentaire afin de procéder à l'échange d'informations pour le processus de pairage. Le canal généralement cité en exemple est le support NFC (*Near Field Communication*). Ne confondez pas SSP et SPP.

- Bandes ISM : Des plages de fréquences radio utilisables sans demande d'autorisation auprès des autorités pour les utilisation industrielles, scientifiques et médicales. Un exemple connu est la bande 26,957 à 27,283 MHz utilisée pour les radio-commandes et pendant longtemps par les téléphones sans fil. La bande 433 MHz (433,05 à 434,79 MHz) est également largement utilisée pour les applications domestiques. Le Bluetooth, tout comme le Wi-Fi, le Zigbee et d'autres normes et standards, utilise la bande 2,4 GHz. Notez cependant que ceci concerne uniquement l'utilisation dans le respect de la réglementation et de l'homologation du matériel utilisé. Hors de question de bidouiller votre périphérique Bluetooth « maison » de 10 Watts, même chez vous !

- HCI : pour *Host/Controller Interface*, est le standard de communication, dans le cas du Bluetooth, entre l'hôte (PC, smartphone, etc.) et le contrôleur (puce Bluetooth). Ne pas confondre avec la *Host Controller Interface* de l'USB (OHCI, EHCI, UHCI, XHCI) ou du Firewire (OHCI).

- IAC : signifiant *Inquiry Access Code* est le premier niveau de filtrage dans la recherche de périphériques Bluetooth. Pour l'heure, cependant, on peut raisonnablement dire que ce code ne sert à rien pour qui veut rester dans la norme. Deux IAC existent, le GIAC (pour *General*) et le LIAC (pour *Limited*). Les DIAC (pour *Devices*) devraient être proposés par le Bluetooth SIG (*Special Interest Group*). Vous n'êtes pas censé utiliser une autre valeur que `0x9e8b33` (GIAC) ou `0x9e8b00` (LIAC), les autres étant *reserved* dans les spécifications. Cependant, l'utilisation d'un DIAC permet, dès la recherche de périphériques, de procéder à une sélection afin de n'obtenir qu'une gamme précise de périphériques à porté.

FOCUS : Wi-Fi et Bluetooth

Le lecteur curieux aura sans doute remarqué, s'il a consulté les différentes pages Wikipédia sur les plages de fréquences ISM, certains... télescopages hertziens. En regardant attentivement les fréquences et leurs correspondances avec les canaux Bluetooth et Wi-Fi, on se rend compte d'un réel problème.

Le Bluetooth définit 79 canaux sur la bande 2,4 GHz avec un taux de saut de fréquence à fréquence de 1600 sauts/seconde. Le Bluetooth n'utilise pas une fréquence ou un canal particulier mais une technique appelée *Frequency Hopping Spread Spectrum* (FHSS).

Le Wi-Fi utilise, par contre, 14 canaux de 22Mhz de large, espacés de 5 Mhz. Il n'y a pas de saut car le Wi-Fi n'utilise pas FHSS mais le *Direct Sequence Spread Spectrum* (DSSS). Les plages de fréquences des canaux Wi-Fi se superposent donc les unes aux autres, ne laissant que trois canaux pouvant être utilisés en même temps sans interférences. Ceci explique pourquoi, à certaines manifestations (salons informatiques), la profusion de points d'accès est contre-productive, car trois systèmes Wi-Fi maximum peuvent fonctionner correctement à proximité l'un de l'autre.

Lorsque Wi-Fi et Bluetooth sont utilisés dans le même espace (à portée l'un de l'autre), le canal utilisé par le Wi-Fi va occuper une place de fréquences correspondant à une vingtaines de canaux Bluetooth. Bien entendu, du fait même du fonctionnement du Bluetooth, des sauts dans cette plage de fréquences seront faits et donneront lieu à des interférences.

Le Bluetooth va réagir à cette situation et changer de canal très rapidement, évitant les situations à interférences. Cependant, il y aura soit des pertes de données (ACL, sans connexion), soit une dégradation de la vitesse (SCO, avec connexion et retransmission). Bluetooth 1.2 améliore la situation en utilisant l'*Adaptive Frequency Hopping* (AFH) afin de trouver et éviter les canaux inutilisables. Toutefois, dans un environnement très « bruité », ce mécanisme peut conduire à la désactivation complète de la communication Bluetooth, les canaux ayant été « blacklistés » jusqu'au dernier.

Je suis sûr que vous vous demandez alors par quel procédé votre ordinateur portable ou votre smartphone arrivent miraculeusement à utiliser ces interfaces Wi-Fi et Bluetooth en même temps. La réponse est simple : ils se parlent et évitent d'utiliser les canaux qui pourraient interférer. Ce *channel skipping* est un mécanisme préventif de désactivation des canaux Bluetooth. Enfin, le multiplexage est également une solution pour un système disposant des deux interfaces, le Wi-Fi signalant une activité et le Bluetooth catégorisant et définissant des priorités pour ses canaux.

Sachez cependant que le problème reste entier en utilisant de multiples systèmes Wi-Fi et Bluetooth dans une maison ou des locaux d'entreprise. Il en va bien sûr de même pour la multiplication des AP Wi-Fi alors que l'AFH permettra généralement à plusieurs communications Bluetooth de fonctionner correctement.

Enfin, fermons cette parenthèse avec quelque chose de plus réjouissant : les similarités dans les bandes de fréquences utilisées par le Bluetooth et le Wi-Fi impliquent une certaine compatibilité. Ainsi, une antenne passive omnidirectionnelle, par exemple, généralement vendue pour une utilisation avec un appareil Wi-Fi, pourra s'adapter sur une interface Bluetooth afin d'en augmenter la portée théorique. Rares sont les adaptateurs Bluetooth disposant d'une connectique d'antenne, mais si vous disposez d'un tel matériel (avec un connecteur RP-SMA, par exemple), rien ne vous empêchera d'utiliser une antenne plus performante que celle livrée en standard. Dans le cas d'un adaptateur de plus bas de gamme, il vous restera toujours la possibilité, armé de votre fer à souder, d'accéder au circuit pour adapter, de force, un connecteur adéquate. Vous trouverez, via une simple recherche sur le Web, quelques pages intéressantes d'utilisateurs ayant procédé à de telles modifications avec succès.

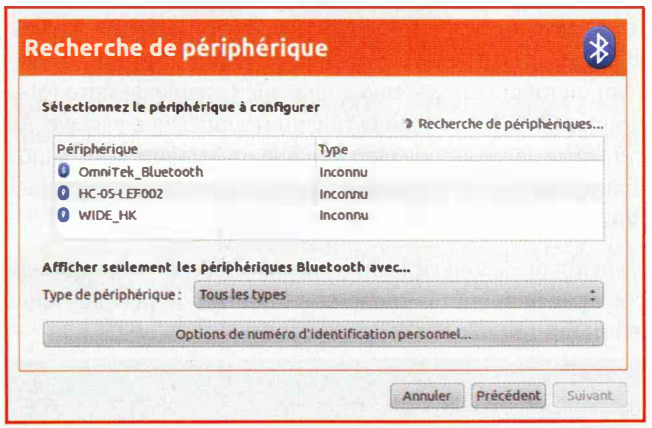
Dans cet article et ceux qui le suivent, nous nous intéresserons principalement à l'utilisation de modules Bluetooth et de périphériques destinés à être explorés. Je pense en particulier à la MetaWatch de Ti. Notre terrain de jeu se composera donc de plusieurs matériels :

- Des smartphones et en particulier un Samsung Naos et un Google Nexus S (à défaut d'avoir eu mon Galaxy Nexus à temps).
- Divers adaptateurs USB de différentes marques, qualités et modèles avec une attention toute particulière portée sur une clé USB industrielle SENA utilisable non pas en guise de simple interface Bluetooth, mais comme une passerelle USB/série fournissant l'accès à une interface Bluetooth pilotée via un jeu spécifique de commandes.
- La montre MetaWatch digitale de Ti.
- Des modules série/Bluetooth utilisables à la fois en esclave pour fournir une connectivité Bluetooth à des montages électroniques, mais également en maître, soit pour un système comme un PC ou un ARM, soit pour les montages souhaitant accéder aux périphériques Bluetooth.

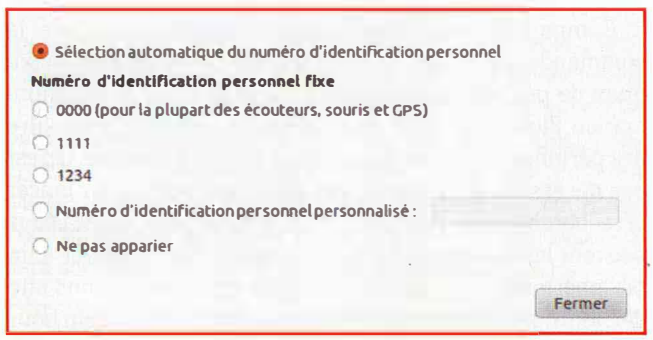
4 Monsieur ? C'est quand qu'on fait du Bluetooth ?

Il est temps à présent de nous pencher sur l'utilisation réelle du Bluetooth. Nous allons ici principalement nous concentrer sur la ligne de commandes GNU/Linux et les mécanismes de configuration et d'utilisation du Bluetooth. Des systèmes comme Windows, Mac OS X ou encore GNU/Linux via l'utilisation de l'interface graphique simplifient grandement la mise en œuvre de périphériques Bluetooth pour l'utilisateur final.

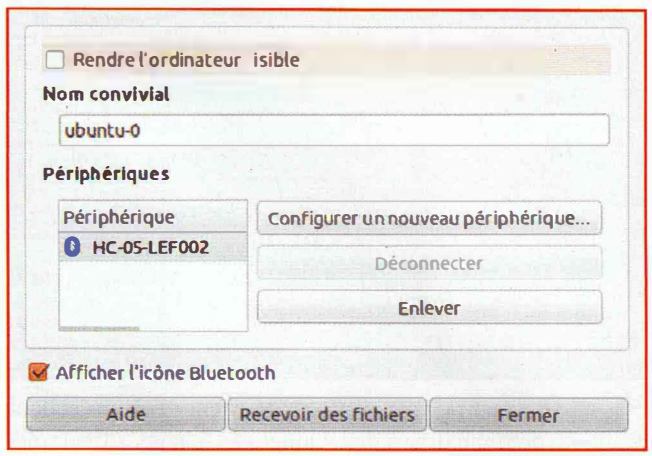
Ainsi, si nous prenons l'exemple d'Ubuntu, tout comme d'autres systèmes du même type (masquant la technicité à l'utilisateur, soi-disant pour son bien), il nous suffit de cliquer sur le petit logo dans la barre de menu pour demander une recherche de périphériques Bluetooth, qui apparaît alors ainsi :



Une tentative d'association, si le périphérique est inconnu (pas de *link key*), sera matérialisée à l'écran via une demande de code PIN :



Des valeurs par défaut (**0000**, **1111**, **1234**) les plus couramment utilisées sont proposées, ainsi que la saisie d'un code PIN personnalisé. Une fois le périphérique appairé, celui-ci figurera dans la liste pour qu'il puisse être supprimé, par exemple :



Ce mécanisme de demande de code PIN est visuellement très simple. Sous le capot, cependant, les choses sont quelque peu différentes. Le système d'authentification utilise le bus logiciel D-Bus qui offre des facilités non négligeables d'un point de vue environnement de bureau. Le support Bluetooth dans GNU/Linux, appelé Bluez, utilise énormément D-Bus. L'authentification des intervenants selon le processus d'association passe donc par ce bus logiciel permettant au démon **bluetoothd** de demander à l'utilisateur la saisie du code PIN. L'application ou l'utilitaire permettant l'intervention humaine est un agent à l'écoute du bus qui répondra à l'appel du système. Ce mécanisme, pendant longtemps, a posé un gros problème aux utilisateurs de la ligne de commandes souhaitant eux aussi profiter de la pile Bluetooth Bluez. Il n'est pas vraiment possible, en effet, comme sur un bureau graphique, de faire apparaître une fenêtre popup et ainsi interrompre l'utilisation du shell et/ou du terminal en cours.

Aujourd'hui, fort heureusement, un agent en ligne de commandes peut être utilisé alors qu'initialement, tous les développements étant orientés vers le bureau graphique, il était nécessaire de littéralement bricoler un agent à partir d'un code de test présent dans les sources de Bluez ou d'aller tripoter les éléments de configuration dans **/var/lib** afin de spécifier un code PIN par défaut. Notez que ce fait historique regrettable découle directement d'une problématique du Bluetooth : la nécessité de l'intervention humaine dans le fait de renseigner un code PIN. Cela dépendra de votre configuration globale mais sachez que si vous comptez créer un ensemble (montage, système embarqué, solution autonome, etc.) susceptible de se connecter à des périphériques Bluetooth inconnus/nouveaux, il vous faudra d'une manière ou d'une autre permettre à un humain (ou équivalent) de saisir et valider un code PIN. Bien entendu, dans une optique d'*appliance-like*, il y a de fortes chances que l'ajout de périphériques ne soit pas du ressort de l'utilisateur final, mais d'un administrateur ou d'un technicien. Ceci suppose un accès à une interface différente, comme un shell distant ou encore le passage du système dans son ensemble dans un mode permettant la configuration et l'ajout de périphériques.

L'utilisation du Bluetooth sur une machine de bureau ou un ordinateur portable nécessite la présence d'une interface Bluetooth. Alors que cela sera totalement transparent pour l'utilisation d'un environnement de bureau, pour nous utilisateurs avancés et initiés, cette interface devra être visible, accessible et configurable. À l'instar de commandes concernant l'Ethernet (**ifconfig**) ou le Wi-Fi (**iwconfig**), il existe une commande de gestion des interfaces Bluetooth :

```
% sudo hciconfig
hci0:   Type: USB
        BD Address: 00:23:4D:E5:E1:3B ACL MTU: 1017:8 SCO MTU: 64:8
```

```
        UP RUNNING PSCAN
        RX bytes:4225 acl:0 sco:0 events:156 errors:0
        TX bytes:446 acl:0 sco:0 commands:37 errors:0
```

Nous obtenons ainsi des informations basiques sur le ou les adaptateurs en présence. L'option **-a** nous en apprendra davantage :

```
% sudo hciconfig -a
hci0:   Type: USB
        BD Address: 00:23:4D:E5:E1:3B ACL MTU: 1017:8 SCO MTU: 64:8
        UP RUNNING PSCAN
        RX bytes:4225 acl:0 sco:0 events:156 errors:0
        TX bytes:446 acl:0 sco:0 commands:37 errors:0
        Features: 0xff 0xff 0x8f 0xfe 0x9b 0xf9 0x00 0x80
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'x61-0'
        Class: 0x4a010c
        Service Classes: Networking, Capturing, Telephony
        Device Class: Computer, Laptop
        HCI Ver: 2.0 (0x3) HCI Rev: 0x212b LMP Ver: 2.0 (0x3) LMP Subver: 0x41d3
        Manufacturer: Broadcom Corporation (15)
```

Une partie des informations concernent le matériel lui-même, comme l'adresse ou le fabricant. D'autres éléments sont purement logiciels (nom, classe, etc.) et sont configurables via les fichiers **/etc/bluetooth/*.conf**. Chacun des fichiers de configuration permet le paramétrage d'un certain nombre de fonctionnalités et de services. Dans le cadre de ce magazine, nous ne nous intéresserons pas aux aspects audio ou réseau afin de nous concentrer sur RFCOMM et la configuration générale. Notez dans la sortie de la commande **hciconfig** la mention de **UP RUNNING PSCAN** indiquant que l'interface est *up* (active) et qu'elle répondra aux tentatives de connexion.

L'initialisation d'une connexion entre deux interfaces Bluetooth passe grossièrement par les procédures de recherche/demande (*Inquiry Procedure*) puis de connexion effective (*Paging Procedure*). Pour qu'une interface réponde à ce qu'on appelle simplement des *scans*, elle doit accepter les *inquiry scans* (**ISCAN**) et elle sera ainsi « découvrable ». Pour qu'elle réponde aux demandes de connexions directes et d'association, elle doit accepter les *page scans* (**PSCAN**).

La méthode la plus radicale et directe pour changer l'état de prise en charge de ces procédures est d'utiliser **hciconfig** suivie du nom de l'interface concernée (généralement **hci0**) et des commandes **pscan**, **iscan**, **piscan** ou **noscan** pour respectivement activer uniquement le *page scan*, activer uniquement le *inquiry scan*, activer les deux ou désactiver les deux. Notez qu'il est également possible de rendre votre interface Bluetooth « scannable » (*inquiry*) en envoyant un message D-Bus avec **dbus-send** si la configuration système le permet (cf. **/etc/bluetooth/main.conf**).

4.1 Recherche et informations

Nous avons sommairement découvert la prise en charge de l'interface Bluetooth telle que présente dans votre système GNU/Linux. Dans la majorité des cas et étant donné le principe même du fonctionnement du Bluetooth, il est plus probable que vous utilisiez GNU/Linux pour vous connecter à un périphérique esclave que l'inverse. Il est rare, en effet, qu'un « gros système » comme une machine de bureau ou même un système embarqué capable de faire fonctionner GNU/Linux joue le rôle du périphérique esclave. Au contraire, la situation selon laquelle un système GNU/Linux doit rechercher et connecter un ou plusieurs périphériques Bluetooth sera bien plus fréquente.

Avant de découvrir le mécanisme d'association, commençons par le début : la recherche (*inquiry*) de périphériques avec l'outil **hcitool** :

```
% hcitool scan
Scanning ...
  A8:F2:74:8B:ED:15  GT-C5130
  D0:37:61:C3:57:AE  MetaWatch Digital WDS112
  00:19:5D:EE:A4:24  WIDE_HK
  00:10:10:16:01:46  HC-05-LEF002
```

Comme vous le devinez, il s'agit là d'un *inquiry scan* et la commande, après une certaine période ou un nombre maximum de périphériques détectés (cf. article sur la programmation Bluetooth en C), nous affichera une liste sommaire des périphériques à portée. Comme vous le constatez, il n'est pas nécessaire de disposer des privilèges **root** pour lancer la recherche mais d'autres actions déclenchées par **hcitool** peuvent les demander (dès qu'il s'agit de se connecter à un périphérique). Nous pouvons utiliser quelques options afin d'obtenir plus d'informations sur les périphériques qui nous entourent :

```
% hcitool scan --flush --info --class
Scanning ...

BD Address:     00:19:5D:EE:A4:24 [mode 1, clkoffset 0x36bf]
Device name:    WIDE_HK [cached]
Device class:   Invalid (0x001f1f)

BD Address:     D0:37:61:C3:57:AE [mode 1, clkoffset 0x6420]
Device name:    MetaWatch Digital WDS112 [cached]
Device class:   Invalid (0x001f00)

BD Address:     00:15:83:18:A0:D4 [mode 1, clkoffset 0x17af]
Device name:    morgane-0 [cached]
Device class:   Computer, Desktop workstation (0x4a0104)

BD Address:     00:10:10:16:01:46 [mode 1, clkoffset 0x3097]
Device name:    HC-05-LEF002 [cached]
Device class:   Peripheral, Sensing device (0x800510)
```

--flush nous permet d'effacer le cache d'un précédent scan afin de repartir sur des données « fraîches ». Ceci n'est

bien entendu nécessaire qu'en phase de test et mieux vaut, en temps normal, reposer sur les mécanismes mis en place par les développeurs de la pile Bluetooth du système. Comme vous pouvez le constater, le cache est encore maintenu pour les noms de périphériques. Vous pouvez utiliser l'option **--refresh** pour forcer une relecture.

Les options **--info** et **--class** nous apporteront respectivement des informations complémentaires sur les périphériques (*clock offset*) et la classe de périphérique/service (*Class of Device/Service* ou CoD) de chaque équipement trouvé. La valeur affichée pour la CoD est composée de trois données fondamentales :

- Le majeur de classe de service permet de renseigner sur le type d'utilité que peut avoir le périphérique dans le sens « service » (capture, téléphonie, transfert de données, ...).
- Le majeur de classe de périphérique indique une famille générale de matériel (périphérique, téléphone, ordinateur, ...).
- Le mineur de classe de périphérique précise la sous-famille (appareil de mesure, sans catégorie, gamepad, etc.).

Le format des données de classe est codifié par un champ spécifique. Pour l'heure, seule la valeur **00** est valable, précisant un premier format standard composé des données listées ci-avant. Notez que le mineur de la classe de périphérique possède un sens variable en fonction du majeur. Ainsi, pour un majeur décrivant un périphérique LAN comme un point d'accès, les données du mineur précisent la charge courante. Pour un majeur spécifiant un téléphone, le mineur déterminera son type (cellulaire, sans fil, smartphone, etc.).

Il est utile de savoir un minimum de choses sur les CoD car la préparation d'une connexion à un périphérique Bluetooth, dans la pratique, se passe en trois phases préliminaires : le scan, la lecture de la classe et la recherche de services via SDP. J'ai personnellement fait l'expérience de certaines versions de logiciels (dont les premières éditions d'Android supportant le Bluetooth) qui, dans la liste des périphériques « à portée », refusaient d'afficher certains produits car leur classe était non initialisée (**0x000000**). De plus, le problème n'était pas visible dans l'interface utilisateur, le périphérique en question n'apparaissant tout simplement pas, mais était signalé via un **logcat**.

Si vous cherchez à composer une valeur complète de CoD, je vous recommande chaudement ce site : http://bluetooth-pentest.narod.ru/software/bluetooth_class_of_device-service_generator.html. Vous pouvez également, si vous en avez le courage, lire les spécifications de la dernière version stable du Bluetooth.

Dans cette dernière sortie en ligne de commandes, vous voyez également apparaître **clkoffset**, signifiant *clock offset* ou décalage d'horloge. En effet, pour que le maître et l'esclave puissent dialoguer de manière efficace, il faut que leurs horloges soient synchronisées. Le maître va donc demander quel est ce décalage dans les toutes premières phases d'échange (FHS pour *Frequency Hopping Synchronization*) avec un esclave. N'oubliez pas que le Bluetooth utilise le *channel hopping* et que, de ce fait, la synchronisation est un élément critique.

Une fois la liste des périphériques répondant à notre scan obtenu, nous pouvons nous adresser à un périphérique en particulier pour obtenir encore plus d'informations :

```
% hcitool info 00:19:5D:EE:A4:24
Requesting information ...
Can't create connection: Operation not permitted
```

Comme vous le voyez, ici des droits super-utilisateur sont indispensables :

```
% sudo hcitool info 00:19:5D:EE:A4:24
Requesting information ...
  BD Address:  00:19:5D:EE:A4:24
  Device Name: WIDE_HK
  LMP Version: 2.1 (0x4) LMP Subversion: 0x1735
  Manufacturer: Cambridge Silicon Radio (10)
  Features: 0xff 0xff 0x8f 0xfe 0x9b 0xff 0x59 0x83
    <3-slot packets> <5-slot packets> <encryption> <slot offset>
    <timing accuracy> <role switch> <hold mode> <sniff mode>
    <park state> <RSSI> <channel quality> <SCO link> <HV2 packets>
    <HV3 packets> <u-law log> <A-law log> <CVSD> <paging scheme>
    <power control> <transparent SCO> <broadcast encrypt>
    <EDR ACL 2 Mbps> <EDR ACL 3 Mbps> <enhanced iscan>
    <interlaced iscan> <interlaced pscan> <inquiry with RSSI>
    <extended SCO> <EV4 packets> <EV5 packets> <AFH cap. slave>
    <AFH class. slave> <3-slot EDR ACL> <5-slot EDR ACL>
    <sniff subrating> <pause encryption> <AFH cap. master>
    <AFH class. master> <EDR eSCO 2 Mbps> <EDR eSCO 3 Mbps>
    <3-slot EDR eSCO> <extended inquiry> <simple pairing>
    <encapsulated PDU> <non-flush flag> <LSTO> <inquiry TX power>
    <extended features>
```

Bien plus d'éléments sont retournées par cette commande, Nous obtenons ainsi la version du LMP, le fabricant du microcontrôleur gérant le Bluetooth du périphérique distant et une liste importante de fonctionnalités supportées. On remarquera que le SSP (**<simple pairing>**) est ici pris en charge, tout comme l'EDR (**<EDR ACL 3 Mbps>**) ou la gestion de l'alimentation (**<power control>**). **<encryption>** est également important, car il nous informe de la possibilité de chiffrer la communication (au niveau service ou lien) afin d'éviter les attaques du type « homme du milieu » (*Man-In-The-Middle* ou MITM en abrégé).

Nous obtenons des informations sur le matériel, mais rien en dehors de ce que permet de supposer la classe (CoD) du périphérique ne nous permet de déterminer exactement les services proposés. Nous ne savons pas ce qu'est effectivement le périphérique distant et ce qu'il nous permet de faire.

Un protocole particulier est dédié à l'obtention de ces informations, c'est SDP (*Service discovery protocol*). Un outil en ligne de commandes, **sdptool**, nous permet de récupérer et d'afficher ces informations. Ici, notre périphérique **WIDE_HK** (qui est un adaptateur Bluetooth/série qui fait l'objet d'un article dédié dans ce numéro) ne nous permet pas de lister (*to browse*) directement les services proposés. Ainsi, la commande suivante est peu loquace :

```
% sdptool browse 00:19:5D:EE:A4:24
Browsing 00:19:5D:EE:A4:24 ...
```

En revanche, nous pouvons, via **sdptool**, provoquer une recherche des périphériques proposant un service particulier, ici **SP** pour *serial port* :

```
% sdptool search SP
Inquiring ...
Searching for SP on 00:19:5D:EE:A4:24 ...
Service Name: Dev B
Service RecHandle: 0x10000
Service Class ID List:
  "Serial Port" (0x1101)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
```

Nous retrouvons donc bien notre périphérique. La commande **sdptool browse** suivie de l'adresse d'un smartphone, par exemple notre Nexus S, nous donnera bien plus d'informations. Tant qu'il est nécessaire de les filtrer avec **egrep** :

```
% sdptool browse 04:18:0F:41:11:2B | \
  egrep "Service Name|Channel: "
Service Name: Audio Source
Service Name: AVRCP TG
Service Name: Voice Gateway
    Channel: 10
Service Name: Voice Gateway
    Channel: 11
Service Name: OBEX Object Push
    Channel: 12
Service Name: OBEX Phonebook
              Access Server
    Channel: 19
```

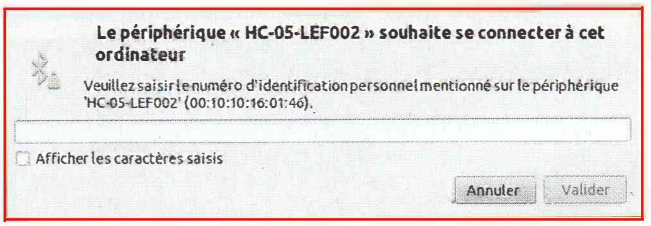
Remarquez que le Nexus n'apparaît pas avec **sdptool search SP**, pas plus qu'avec **sdptool browse** sans spécifier d'adresse de périphérique. En effet, les périphériques Android ne sont pas visibles (scannable) par défaut (mais ils sont associables). Or justement, lorsque nous ne précisons pas d'adresse, **sdptool** procède à une recherche de périphériques puis interroge un par un ceux qu'il aura trouvés.

4.2 Connectons-nous !

Notre module **WIDE_HK** présent à proximité propose un service de liaison série (SSP/RFCOMM). Nous pouvons donc, en principe, nous y connecter simplement en utilisant la commande **rfcomm** et... :

```
% rfcomm connect 0 00:19:5D:EE:A4:24
Can't connect RFCOMM socket: Connection refused
```

... ça ne fonctionne pas ! La connexion est refusée, et pour cause : les périphériques ne sont pas associés. Si vous avez utilisé cette commande dans un terminal lancé depuis une abomination^W^Wun environnement de bureau comme GNOME 3 ou Unity, vous avez sans doute vu apparaître une fenêtre comme :



C'est un message D-Bus qui provoque cela. En l'absence d'une *link key* existante entre le maître et l'esclave, le message sur le bus logiciel demande une intervention humaine pour une demande de code PIN. Dans un environnement de bureau, un agent est généralement en attente de ce type de message et, dès réception, il fait apparaître à l'utilisateur une fenêtre permettant d'entrer l'information nécessaire à l'association (code PIN). Il en ira de même pour des environnements plus acceptables et légers comme XFCE ou e17, incluant eux aussi un agent Bluetooth.

Dans un cadre d'utilisation pure de la ligne de commandes, ou avec un environnement graphique très léger n'incluant pas d'agent à l'écoute sur le bus logiciel, il est nécessaire d'utiliser un programme en ligne de commandes. Par défaut, celui-ci s'appelle **bluetooth-agent** et il pourra parfaitement être utilisé ponctuellement, lorsqu'il sera nécessaire d'associer un périphérique inconnu. Le démon **bluetoothd**, une fois la phase d'association via un code PIN terminée, n'aura plus besoin de l'assistance d'un agent.

Pour associer notre périphérique **00:19:5D:EE:A4:24**, nous disposons de deux solutions avec **bluetooth-agent**. Soit nous spécifions un code PIN par défaut, valable pour tous nouveaux périphériques en passant simplement ce code pour seul argument de la commande :

Dans un premier terminal :

```
% bluetooth-agent "1234"
```

L'agent reste en attente, prêt à répondre à la demande de code PIN sur le bus logiciel. Dans un autre terminal, nous procédons à une tentative de connexion :

```
% rfcomm connect 0 00:19:5D:EE:A4:24
```

Dans le premier terminal, nous voyons ainsi apparaître la réception et le traitement de la demande via D-Bus :

```
Pincode request for device
  /org/bluez/8664/hci0/dev_00_19_5D_EE_A4_24
```

Et, à nouveau sur le second terminal, la connexion s'établit :

```
Connected /dev/rfcomm0 to 00:19:5D:EE:A4:24 on channel 1
Press CTRL-C for hangup
```

Dès lors, le périphérique distant est accessible via **/dev/rfcomm0** comme s'il s'agissait d'un port série **/dev/ttyS*** local. En arrêtant l'exécution de la commande **rfcomm** avec CTRL+C, l'agent quitte également. Une nouvelle tentative d'établissement de la connexion réussit sans l'intervention de ce dernier. En effet, la link key existe car les deux systèmes sont d'ores et déjà associés. En jetant un œil dans **/var/lib/bluetooth**, vous trouverez un répertoire nommé d'après l'adresse de votre interface Bluetooth locale. Dans ce répertoire, un certain nombre de fichiers regroupent les informations sur les périphériques utilisés ou vus par l'adaptateur en question :

- **classes** : la CoD de chaque périphérique ;
- **config** : les informations de configuration maintenues par le démon Bluetooth pour cet adaptateur local ;
- **features** : le cache des fonctionnalités reportées par les périphériques détectés,
- **lastseen** : la date et l'heure de la dernière détection pour chaque périphérique (au format humainement lisible) ;
- **lastused** : idem pour la dernière utilisation ;
- **manufacturers** : le code désignant le fabricant de chaque périphérique et la version LMP (version/sous-version) ;
- **names** : les noms annoncés par chaque périphérique ;
- **linkkeys** : et enfin, la fameuse clé utilisée pour l'association.

L'autre solution consiste à ne pas utiliser de code PIN par défaut mais d'en spécifier un spécialement pour un périphérique. Il suffit pour cela de faire suivre le code PIN par l'adresse du périphérique concerné :

```
% bluetooth-agent "1234" 00:19:5D:EE:A4:24
Pincode request for device /org/bluez/8432/hci0/dev_00_19_5D_EE_A4_24
Agent has been released
```

Inutile ici de tenter d'initier une connexion en parallèle puisque la commande elle-même va utiliser le code PIN pour établir cette liaison et procéder à l'association. **bluetooth-agent** rend ainsi la main relativement rapidement et on retrouvera le même type d'informations dans le fichier **linkkeys**.

Vous vous demandez certainement comment il est possible de dissocier un périphérique Bluetooth pour une interface. La réponse que nous avons trouvée n'est malheureusement pas très élégante avec les outils installés par défaut : éditer ou supprimer le fichier **linkkeys** puis redémarrer le démon **bluetoothd**. Heureusement, il existe une autre solution via l'utilisation des outils du projet **bluez-tools** (installables sous Debian GNU/Linux via le paquet du même nom). Jugez plutôt...

Recherche de périphériques :

```
% bt-adapter -d
Searching...

[00:19:5D:EE:A4:24]
  Name: WIDE_HK
  Alias: WIDE_HK
  Address: 00:19:5D:EE:A4:24
  Icon: (null)
  Class: 0x1f1f
  LegacyPairing: 1
  Paired: 1
  RSSI: -70

[00:10:10:16:01:46]
  Name: HC-05-LEF002
  Alias: HC-05-LEF002
  Address: 00:10:10:16:01:46
  Icon: (null)
  Class: 0x800510
  LegacyPairing: 1
  Paired: 0
  RSSI: -51

[D0:37:61:C3:57:AE]
  Name: MetaWatch Digital WDS112
  Alias: MetaWatch Digital WDS112
  Address: D0:37:61:C3:57:AE
  Icon: (null)
  Class: 0x1f00
  LegacyPairing: 1
  Paired: 0
  RSSI: -78

Done
```

Information sur les services d'un périphérique :

```
% bt-device -s 00:19:5D:EE:A4:24
Discovering services...

[RECORD:65540]
SrvClassIDList: "SP"
```

```
ProtocolDescList:
    "L2CAP"
    "RFCOMM", Channel: 1
SrvName: "Dev B"

Done
```

Lancement de l'agent interactif :

```
% bt-agent
Agent registered
[...]
```

Connexion **rfcomm** sur un autre terminal et saisie/utilisation du code PIN dans le terminal de l'agent interactif :

```
[...]
Device: WIDE_HK (00:19:5D:EE:A4:24)
Enter PIN code: 1234
```

Listage des périphériques associés et des informations sur l'un d'eux (via son nom en guise d'alias) :

```
% bt-device -l
Added devices:
WIDE_HK (00:19:5D:EE:A4:24)

% bt-device -i WIDE_HK
[00:19:5D:EE:A4:24]
  Name: WIDE_HK
  Alias: WIDE_HK [rw]
  Address: 00:19:5D:EE:A4:24
  Icon: undefined
  Class: 0x1f1f
  Paired: 1
  Trusted: 0 [rw]
  Blocked: 0 [rw]
  Connected: 0
  UUIDs: []
```

Suppression du périphérique (dissociation) :

```
% bt-device -r WIDE_HK
Done

% bt-device -l
No devices found
```

Vous reconnaîtrez que l'ensemble est cohérent et bien plus agréable à utiliser que les outils par défaut de la pile Bluez. Si cet ensemble d'utilitaires n'est pas disponible pour votre distribution (pour l'embarqué ou de bureau), vous pourrez utiliser directement les sources disponibles sur http://code.google.com/p/bluez-tools/.

Petite note au passage : lors de nos expérimentations, nous avons pu remarquer que l'interrogation des services disponibles pour les module Bluetooth/série ne renvoyait parfois pas d'UUID. Dans la sortie précédente, on voit en effet que le champ est vide alors qu'une autre occurrence de la commande nous retourne :

```
[...]
  Class: 0x1f1f
  Paired: 1
  Trusted: 0 [rw]
  Blocked: 0 [rw]
  Connected: 0
  UUIDs: [SerialPort]
```

Ceci pourrait expliquer les problèmes de connexion répétés rencontrés par moi-même dans le développement d'une application Android permettant de communiquer avec un tel module. Mais le problème ne semble pas se limiter à ces modules car, comme vous le verrez dans l'article dédié, il existe une solution (tenant plus du hack que d'un correctif). Celle-ci consiste à utiliser une méthode qui n'est pas disponible dans l'API publique d'Android. Or cette méthode est bien connue et utilisée dans quasiment toutes les applications Android utilisant RFCOMM et ce incluant, par exemple, le support MetaWatch de Ti pour Android. Pourquoi l'UUID n'est-il pas toujours obtenu ? Mystère. Mais le fait est que le problème souvent rencontré par les développeurs Android existe également sur d'autres systèmes et qu'il n'est pas dû au périphériques Bluetooth en présence.

L'UUID (*Universal Unique IDentifier*), comme son nom l'indique, est une valeur unique permettant d'identifier un service. Un certain nombre d'UUID sont génériques, comme le bien connu **00001101-0000-1000-8000-00805F9B34FB** pour les ports séries. Le protocole voulant qu'une phase de découverte de service précède la connexion, l'API Android utilise cet UUID comme validateur avant la connexion. Si le périphérique ne propose pas ce service, la connexion échoue car elle n'est même pas tentée. La problème ne se pose pas avec les outils GNU/Linux en ligne de commandes comme **rfcomm**, car c'est à la charge de l'utilisateur de vérifier la disponibilité du service (avec **sdptool**, par exemple) avant la tentative.

Conclusion

Nous venons de voir ici les bases de la recherche de périphériques et de la connexion RFCOMM. Nous n'avons cependant pas exploré l'ensemble des fonctionnalités offertes par le Bluetooth car ceci ne se résume, bien entendu, pas à l'émulation de ports série. Cependant, dans l'optique du magazine, les services audio, la gestion de profils ou encore le transfert d'objets (OBEX) sont des fonctionnalités beaucoup moins intéressantes car moins universelles qu'une communication RFCOMM très simple à implémenter et à utiliser avec tous types de systèmes (jusqu'aux microcontrôleurs). Dans le prochain article, nous ferons connaissance avec un module d'entrée de gamme (ou pas) capable de fournir une connectivité Bluetooth à n'importe quel matériel déjà capable de communiquer via une liaison série TTL (5V). Il sera même possible d'établir une connexion de manière totalement transparente entre deux de ces systèmes. ■

Abonnez-vous !

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

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

Économisez plus de 20%*

* Sur le prix de vente unitaire France Métropolitaine

4 Numéros de Open Silicium

Les 3 bonnes raisons de vous abonner :

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

4 façons de commander facilement :

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

par ABONNEMENT : 27€*

au lieu de 36,00* en kiosque

Économie : 9,00 €*

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

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

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

Open Silicium

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

Vos remarques :

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

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

Tournez SVP pour découvrir toutes les offres d'abonnement

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

➡ Voici une sélection des offres de couplage avec Open Silicium

offre 13 — Open Silicium Magazine (4 nos)

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

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

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

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

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

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

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

Vous pouvez également vous abonner sur : www.ed-diamond.com ou par Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 2

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

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

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

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

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

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

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

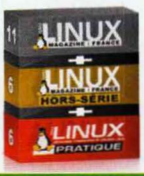
offre 5 — + GNU/Linux Magazine (11 nos) + MISC (6 nos)

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

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

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

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



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

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

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

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

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

➡ Nos Tarifs s'entendent TTC et en euros

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

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

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

Mes choix :

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

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

Je choisis de régler par :

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

Expire le : ⌊_⌊_⌊_⌊_⌋

Cryptogramme visuel : ⌊_⌊_⌊_⌋

Date et signature obligatoire

UTILISATION DU MODULE CSR BLUECORE4 DE WIDE.HK

par Denis Bodor

Transformer n'importe quel montage électronique « intelligent » en périphérique Bluetooth. Ajouter des fonctionnalités Bluetooth à un système ne disposant que d'une connectivité série. Créer automatiquement une liaison série sans fil, over Bluetooth, entre deux systèmes sans avoir à implémenter le moindre code de gestion de part et d'autre. Voilà autant de réalisations possibles en mettant en œuvre ce qu'il est commun d'appeler un UART Bluetooth.

Avant toutes choses, il est important ici de bien définir de quoi nous parlons. Un UART (*Universal Asynchronous Receiver Transmitter*) Bluetooth se résume en peu de choses : un contrôleur (ou système) Bluetooth et une interface série. Ce type de module est très courant et, c'est là que c'est un peu délicat, les différents modèles sont matériellement tous, à très peu de choses près, totalement identiques. Une simple recherche sur eBay ou les boutiques en ligne de composants/modules vous permettra d'en décompter plus d'une douzaine aux désignations différentes, et parfois, aux *datasheets*, manuels et commandes/configurations différents.

Les photos insérées dans cet article, si vous les comparez à celles obtenues en cherchant « bluetooth uart » sur le Web, vous paraîtront étrangement proches. Il ne semble y avoir matériellement que deux déclinaisons ou, plus exactement, un module nu systématiquement utilisé et une *breakout board* incluant plus ou moins de composants. Le module de base est construit autour d'un circuit intégré dédié, le BC417143B. Cette puce BlueCore4-External de chez CSR (Cambridge Silicon Radio) intègre les couches matérielles de la pile Bluetooth en version 2.1+EDR, tout en ajoutant divers éléments comme des interfaces séries, USB, et SPI, de la RAM, un DSP ou encore des GPIO. La désignation « External » du BlueCore4 (BC4) fait référence à l'utilisation d'une mémoire flash de 8Mbit externe accessible par le composant. Cette mémoire est destinée au stockage d'un firmware spécifique, normalement développé avec la *stack* CSR Bluetooth.

Ceci vous expliquera pourquoi un même module disposera de commandes et de procédures de mise en œuvre différentes en fonction du vendeur. Les fabricants personnalisent le module en y chargeant le firmware puis en le soudant sur une platine incluant un régulateur de tension, une résistance, une LED et éventuellement un micro-switch permettant le choix d'un mode de fonctionnement.

Pour ce type de composant, je me fournis généralement auprès de deux entités qui sont SURE electronics et Wide.hk, via la plateforme de vente eBay. Bien entendu, ce type de module peut se trouver dans des boutiques web, mais entre un module comme le « Bluetooth Modem - BlueSMiRF Silver » de SparkFun à 29 euros (+ port) ou la même chose à moins de 12 euros en provenance de Hong Kong, je préfère attendre un peu plus longtemps, en particulier lorsqu'on me fait une remise ou qu'on m'ajoute quelques exemplaires car j'en ai commandé une douzaine (eh oui, c'est aussi cela la philosophie chinoise de la vente par correspondance).

J'ai choisi le module de Wide.hk car il intègre un firmware qui semble largement utilisé, même s'il existe en plusieurs versions et que je ne suis pas arrivé à mettre la main sur les sources ou un éventuel SDK. De plus, la dernière version avec breakout board s'est vue équipée d'un micro-switch permettant de basculer le module en mode configuration (mode commandes AT) par opposition au simple mode *transparent data*. Précédemment ou avec d'autres modules, il était nécessaire d'appliquer un niveau haut sur la broche 34/PIO11 du module pour le passer en mode AT. Avec des broches de 1 mm de large espacées de 0,5 mm, je vous laisse imaginer le calme et la paix intérieure nécessaire. Ainsi, la combinaison du faible coût avec la popularité du firmware et l'interrupteur de changement de mode font pour moi de ce module une version très acceptable pour s'amuser un peu.

Mon nom est...

Voici une petite sélection de noms et références d'un seul et même produit, utilisant le même firmware mais décliné sous bien des désignations : Emartee CuteDigi BMX, HHW-SPP embedded Bluetooth serial module, BTM400_6b CSRBC4 module, Lanwind BT-1800-1, HC-05 et bien d'autres. Dans tous les cas, si vous voulez savoir si le produit qui vous est présenté est le même que celui décrit ici, récupérez le manuel et vérifiez la correspondance des commandes.

1 Tour du propriétaire

Vous l'aurez compris, le module se compose de deux éléments :

- Un circuit composé de la puce CSR BlueCore4-External, de la mémoire flash et de quelques composants passifs (résistances, capa, etc.). Le module est équipé de 34 connecteurs sur la tranche du circuit : 13 à droite, 13 à gauche et 8 en bas. Il s'agit d'une connectique permettant un montage (soudure) en surface et le choix d'une version avec breakout board est dans ce cas plus que légitime. La partie supérieure du circuit intègre l'antenne qui n'est autre qu'une piste de cuivre aux caractéristiques adéquates. Un point de test à proximité pourrait peut-être servir pour l'adaptation d'un connecteur d'antenne RP-SMA, mais je dois vous avouer que je n'ai pas eu le cœur à tenter l'opération.
- La breakout board intégre un convertisseur/régulateur de tension XC6206P332 (selon la documentation) 5V vers 3,3V, quelques condensateurs de filtrage, un couple LED/résistance relié à la broche 31/PIO8 et des vias permettant de souder un connecteur au pas de 0,25. Le dessous du circuit comprend un micro interrupteur marqué « AT MODE » directement relié à la broche 31/PIO11 du module de base permettant, au choix, de passer cette ligne à la masse ou à l'état haut.

Une précédente version du module en ma possession, également de Wide.hk, est quasi identique à l'exception du verni violet du circuit (contre le classique vert avec la nouvelle version) et de l'absence d'interrupteur. Ah oui, l'ancienne version intégrait une LED bleue. La nouvelle est orange et bien moins captivante.

Deux modes de fonctionnement sont possibles :

- Avec l'interrupteur positionné vers le « E » de la sérigraphie « AT MODE », le module est en mode série transparent. Dans ce mode, toutes les données arrivant via la liaison série (RX) iront directement au périphérique de l'autre côté de la liaison RFCOMM Bluetooth. Inversement, les données reçues seront retransmises sur la ligne TX sans plus de fioriture. La configuration par défaut du module, normalement, définit un fonctionnement en tant qu'esclave, détectable (*inquiry*), apairable, proposant une liaison à 9600 bps et utilisant un code PIN « 1234 ».
- Avec l'interrupteur côté « AT » et après un reset (rupture et rétablissement d'alimentation), le module passe en mode commandes. Les utilisateurs ayant fait leurs armes à l'époque des modems analogiques retrouveront leurs marques puisque la configuration du module se fait, dans ce mode, via des commandes débutant par **AT** et retournant, en cas de réussite, **OK** tout comme un modem.

Notez que le module de base, si vous n'avez pas peur de souder ces toutes petites choses, vous coûtera moins de 6 euros pièce (port inclus). Vous devrez cependant vous fabriquer votre propre breakout board, comme l'a fait le créateur du blog Elastic Sheep en utilisant les services d'Olimex. De plus, il faudra alimenter le module en 3,3V et donc prévoir l'électronique adéquate, la LED, etc.

Côté mise en œuvre, on utilisera des convertisseurs USB/série TTL qu'on connectera directement VCC/VCC, GND/GND, RX/TX et TX/RX. Bien entendu, n'importe quelle solution permettant une communication série en utilisant les mêmes tensions est également possible, ceci incluant les convertisseurs RS-232/TTL ou le connecteur 4 broches présent sur un Arduino Duemilanove, par exemple.

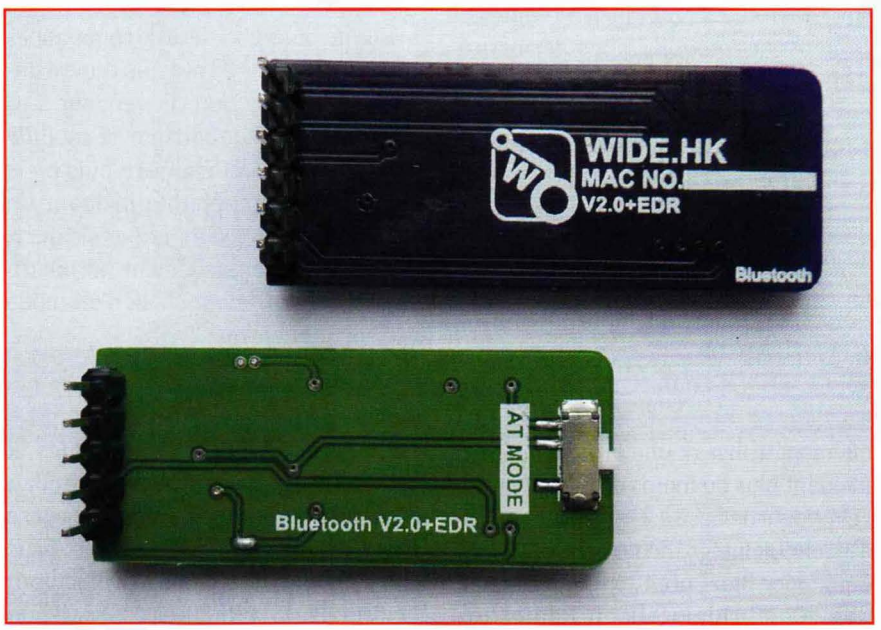


Figure 1

2 Mode AT

Le module Wide.hk peut être configuré facilement. Le matériel mis sous tension une fois le micro-switch placé sur le mode AT, il suffit de se connecter en 38400 8N1, soit 38400 bps, 8 bits de données, pas de parité et 1 bit de stop. Le choix de l'utilitaire de communication est une affaire de préférences personnelles. Pour ma part, et pour en avoir testé un grand nombre, je vous recommande, sous GNU/Linux (et éventuellement Mac OS X en mode terminal), GNU Screen, qui pourra être lancé avec **`screen /dev/ttyUSB0 38400`** où **`/dev/ttyUSB0`** est le premier adaptateur USB/série détecté par le système. Adaptez en fonction de votre configuration. Minicom, sous GNU/Linux, peut également être un choix possible si l'on supporte l'interface. Il existe également des outils en mode graphique.

Sous Mac OS X, en mode graphique, CoolTerm est un excellent choix. Notez que ce système n'intègre pas nativement, contrairement à Linux, de pilotes pour les adaptateurs FTDI ou autres. Il vous faudra installer un driver préalablement téléchargé sur le site FTDI : le VCP ou *Virtual Comm Port*. De plus, par défaut, les permissions sur l'entrée **`/dev`** correspondant au port série supplémentaire ne permettent pas un usage par un utilisateur « standard » (groupe **`wheel`**). Pour corriger cela, vous devrez, dans le cas d'un adaptateur utilisant une puce FTDI :

- Ouvrir un terminal.
- Devenir **`root`** avec **`sudo -s`**.
- Vous placer dans le répertoire des extensions noyau avec **`cd /system/library/extensions`**.
- Changer le groupe auquel appartient l'extension correspondant au pilote FTDI avec **`chgrp -R wheel FTDIUSBSerialDriver.kext`**.
- Redémarrer le système.

Une procédure équivalente sera sans doute nécessaire pour d'autres pilotes, comme les adaptateurs à base de PL2303.

Côté Windows, là aussi, un pilote devra être installé pour un adaptateur USB/série, mais c'est quelque chose que je n'ai pas testé, n'étant pas très amateur de ce système. Il semblerait que l'application RealTerm fasse office d'outil appréciable pour la gestion d'un terminal série.

Dans tous les cas, le point important de la configuration, en dehors du 38400 8N1, réside dans la configuration du caractère marquant les fins de lignes (EOL pour *End Of Line*). Avec les modules Bluetooth Wide.hk, le marqueur est CR+LF, soit un retour chariot **`\r`** suivi d'un saut de ligne **`\n`**. Pour rappel, les équivalences sont :

- CR = **`\r`**, **`0x0D`** en hexa, **`13`** en décimal et **`15`** en octal ;
- LF = **`\n`**, **`0x0A`** en hexa, **`10`** en décimal et **`12`** en octal.

Attention, le marqueur EOL est complètement dépendant du module que vous utilisez et ceci ne se limite pas au Bluetooth. Dès qu'il s'agit d'utiliser une interface de communication non binaire et donc orientée « texte », les constructeurs font un peu comme ils en ont envie. Tantôt CR, tantôt LF, parfois CR+LF, et j'ai même vu quelques cas de LF+CR (si, si). Quelle que soit votre application préférée, veillez à bien configurer EOL. Dans le cas de GNU Screen, la solution consiste à ajouter un raccourci clavier pour mapper la touche F11 sur **`\r\n`** dans votre **`~/.screenrc`** :

```
bindkey -k F1 stuff "\015\012"
```

Dès lors, il faudra valider les commandes avec F11 et non la touche entrée (donnant un **`\n`**) provoquant, étrangement, la répétition non-stop du message de validation ou d'erreur de la commande qui vient d'être validée.

3 Configuration du module

J'ai commandé chez Wide.hk une quantité non négligeable de modules et force est de constater qu'ils ne sont pas tous configurés de la même manière. Voilà donc l'occasion de faire un tour de la configuration et des différentes options permettant l'utilisation la plus courante : le mode esclave simple.

À la mise sous tension avec le mode AT activé, la LED intégrée clignote de façon lente (1 seconde allumée, 1 seconde éteinte) signifiant ainsi l'état actuel en mode configuration. Nous pouvons alors utiliser nos premières commandes :

```
AT
OK

AT+ORGL
OK

AT+NAME?
+NAME:HHW-SPP-1800-2
OK

AT+VERSION?
+VERSION:1.0-20090818
OK
```

Toutes les commandes commencent par **`AT`** et sont complétées par un **`+`** suivi d'une directive. Les commandes pouvant prendre en argument un ou plusieurs paramètres sont normalement suivies de **`=`** puis des paramètres, éventuellement séparés par des virgules. Pour consulter l'état d'un paramètre, la même commande est utilisée, sans **`=`**, mais avec pour seule suite un point d'interrogation (**`?`**). Ici, nous utilisons tout d'abord la commande **`AT`** seule qui est un simple test. Si tout est correctement configuré, le module doit répondre **`OK`** sur une ligne. En cas d'erreur de syntaxe, le message est **`ERROR:(x)`**, où **`x`** est le code d'erreur.

Notre seconde commande, qui ne prend pas d'argument, demande un retour à la configuration d'usine. Les valeurs de cette configuration sont, normalement :

- Nom du périphérique Bluetooth : **`HHW-SPP-1800-2`**.
- Classe (CoD) : **`0x000000`**.
- IAC : **`0x009e8b33`**.
- Mode esclave.
- Association en SPP.

- Port série en 38400 8N1. Attention : ceci correspond à la configuration en fonctionnement normal. En mode AT, la configuration reste 34000 8N1 même si ce paramètre est changé.
- Code PIN **1234**.

J'ai bien dit « normalement », certains des modules reçus, par exemple, utilisaient un nom différent, comme **WIDE.HK**.

La troisième commande affiche le nom du périphérique, et la dernière, la version du firmware en flash. Ici, il s'agit de la même version que celle donnée dans la documentation référencée par Wide.hk mais produite, semble-t-il, par EU-ASIA. Là encore, j'ai constaté qu'il existait plusieurs « variations » autour de la documentation (couleur, mise en page, format) sans effet notable sur la syntaxe des différentes commandes ou le comportement du module.

Commençons par personnaliser le module non sans avoir auparavant affiché et noté dans un coin l'adresse matérielle du périphérique :

```
AT+ADDR?
+ADDR:19:5d:eea412
OK

AT+NAME=OpenSilicium_5
OK

AT+NAME?
+NAME:OpenSilicium_5
OK
```

L'adresse de notre périphérique est **00:19:5D:EE:A4:12**. La notation utilisée correspond à **NAP:UAP:LAP**, soit :

- NAP (*Non-significant Address Portion*) d'une taille 16 bits qui n'est pas préfixée de **00** dans la sortie ;
- UAP (*Upper Address Portion*) de 8 bits ;
- LAP (*Lower Address Portion*) de 24 bits.

Après avoir changé le nom du périphérique (max 32 caractères), nous nous assurons de la prise en compte effective de la commande, malgré le rassurant **OK** affiché. Je vous conseille de toujours

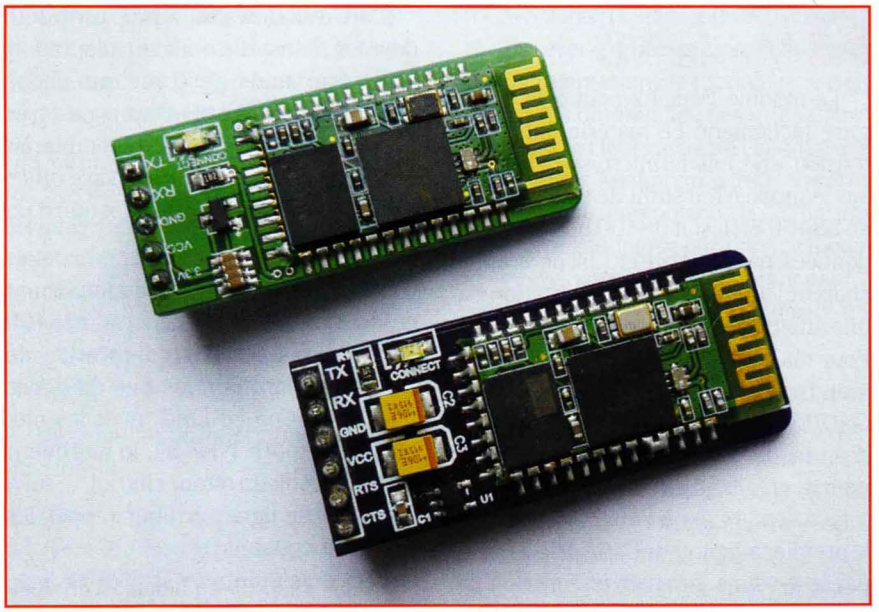


Figure 2

procéder ainsi, même si je n'ai jamais eu de problème particulier. On est jamais trop prudent. Il n'est, bien entendu, pas possible de spécifier l'adresse matérielle via la configuration.

Un autre élément que vous pouvez souhaiter changer est la classe de périphérique et de service (CoD, cf. le premier article du dossier). Celui-ci détermine l'objet du périphérique, son utilité et son usage, donnant ainsi une idée des services qu'il propose. La CoD peut être utilisée pour filtrer les périphériques que vous souhaitez asservir depuis votre application côté maître. Par défaut, sa valeur est **0**. Pour mon usage personnel, j'ai l'habitude d'utiliser la classe **0×800510** : Information / périphérique / non-clavier / détecteur/sensor.

```
AT+CLASS=800510
OK

AT+CLASS?
+CLASS:800510
OK
```

L'IAC peut être changé. Il est, par défaut, défini sur **0x9e8b33**, soit le GIAC (*General Inquire Access Code*). Bien que la documentation précise qu'il est possible de définir un code autre que le GIAC ou le LIAC, il n'est généralement pas recommandé de faire cela pour des raisons évidentes de respect des standards.

Le dernier point demandant une attention particulière de la configuration est celui de l'UART, c'est-à-dire de la liaison série. Notez que la configuration que vous allez spécifier est celle du fonctionnement normal et non du mode AT, qui reste toujours en 38400 8N1.

```
AT+UART?
+UART:38400,0,0
OK

AT+UART=9600,0,0
OK

AT+UART?
+UART:9600,0,0
OK
```

Les paramètres sont au nombre de trois, séparés comme il se doit par des virgules. Le premier détermine la vitesse entre 4800 et 1382400 avec des paliers à 9600, 19200, 38400, 57600, 115200, 230400, 460800, 912600. Le choix doit être fait en fonction de la vitesse de la liaison série du montage ou du système auquel est attaché le module. Généralement, 9600 est une valeur acceptable. 115200 peut être intéressante si vous reliez le module à un port console d'un

système embarqué, par exemple, afin de fournir une connectivité sans fil au système embarqué. Pour une communication entre deux modules, tâchez d'utiliser une vitesse de l'UART identique de chaque côté afin d'éviter tout risque de débordement. Le second paramètre détermine le nombre de bits de stop, **0** pour 1 bit, **1** pour deux (attention !). Enfin, le dernier paramètre précise la parité avec **0** pour aucune, **1** pour paire et **2** pour impaire. Il n'est pas possible de choisir la taille des données (7 ou 8 bits).

Le dernier paramètre concerne la sécurité avec la définition du code PIN de votre périphérique :

```
AT+PSWD=0000
OK

AT+PSWD?
+PSWD:0000
OK
```

Il est possible de définir un code PIN non numérique, mais il s'agit généralement d'un code à 4 ou 6 chiffres. Le maximum est de 16 octets. Ce dernier réglage clôt l'étape de configuration de base. En coupant l'alimentation du module et en basculant le micro-switch sur l'autre position avant de rétablir le courant, la LED intégrée devrait clignoter rapidement, signifiant que le module est en attente de connexion. Il devrait être détectable en procédant à une recherche et associable avec le code PIN que vous aurez spécifié. Une fois la connexion établie, par exemple via **rfcomm** sous GNU/Linux, tout ce que vous enverrez au module en Bluetooth sera retransmis sur la ligne TX. Tout ce que vous enverrez sur RX traversera les airs dans l'autre sens. À l'aide d'un PC équipé de Bluetooth, il est donc relativement simple de vérifier la connectivité exactement dans le même contexte de connexion que pour la configuration : ce qu'envoie le maître Bluetooth doit apparaître dans le terminal série et inversement.

Lorsqu'il est accédé (connecté en Bluetooth), le module fait brièvement clignoter deux fois la LED intégrée toutes les secondes.

4 Histoire de rôle

La configuration par défaut du module, via quelques paramètres basiques, permet l'usage courant en esclave. Cependant, le module est capable de bien plus. C'est par la définition du rôle du module que ce comportement peut être changé. Par défaut, nous avons :

```
AT+ROLE?
+ROLE:0
OK
```

0 correspond ici au rôle esclave où le module est censé être accédé par un périphérique maître initiant la connexion. Le rôle **2** est destiné aux tests. Le module est également en attente de connexions, mais n'utilisera pas les lignes RX/TX. Il se contentera de renvoyer n'importe quel caractère (octet) reçu à l'expéditeur. Ce rôle est appelé *slave-loop*. Enfin, le rôle **1** est plus intéressant puisque le module passe alors en mode maître et est capable d'initier des connexions avec d'autres périphériques.

Le principe est le suivant : vous configurez le module de manière à initier une communication avec un périphérique Bluetooth distant en mode AT puis, lors de la mise sous tension en mode standard, le module utilisera vos paramètres pour tenter la connexion. Bien entendu, une partie de la communication va se faire dans le mode AT puisque les deux périphériques vont devoir être associés. Un certain nombre de paramètres de configuration sont identiques au rôle esclave vu précédemment, mais ils ne possèdent pas le même sens.

Nous allons commencer par utiliser le module en mode AT comme hôte pour rechercher les périphériques à proximité :

```
AT+INIT
OK

AT+STATE?
+STATE:INITIALIZED
OK

AT+IAC=9e8b33
OK

AT+CLASS=0
OK

AT+INQM=1,9,48
OK

AT+INQ
+INQ:19:5D:EEA3F0,1F1F,FFA8
+INQ:19:5D:EEA3F0,1F1F,FFA9
+INQ:19:5D:EEA3F0,1F1F,FFA7
+INQ:19:5D:EEA3F0,1F1F,FFA8
+INQ:19:5D:EEA3F0,1F1F,FFA8
+INQ:19:5D:EEA3F0,1F1F,FFA6
+INQ:19:5D:EEA3F0,1F1F,FFA7
+INQ:19:5D:EEA3F0,1F1F,FFA6
+INQ:19:5D:EEA3F0,1F1F,FFA6
OK
```

Nous procédons à l'initialisation de la « bibliothèque de profils SPP » (dixit la documentation) puis vérifions l'état du module. Le réglage de l'IAC et de la classe dans le rôle de maître ne définit plus ce qui concerne le module lui-même, mais les paramètres du ou des périphériques que nous cherchons. Ainsi, utiliser une autre valeur d'IAC aura pour effet, lors de la recherche, de ne lister que les périphériques utilisant l'IAC en question. Il en va de même pour la classe, avec une exception pour la valeur **0**, permettant de rechercher tous les périphériques quelle que soit leur classe.

AT+INQM définit les paramètres de recherche (*inquiry mode*). Les trois arguments à préciser sont :

- Le mode de recherche. Soit standard avec **0**, soit **1** pour une indication de l'intensité du signal (RSSI pour *Received Signal Strength Indicator*). Il n'existe pas de notation normalisée du RSSI et, malheureusement, la documentation du module ne spécifie pas le rapport entre la valeur affichée et l'intensité effective du signal.
- Le nombre maximum de périphériques à détecter. Cette valeur, si elle est atteinte, arrêtera la recherche même si la valeur du dernier paramètre n'est pas atteinte.
- Le temps maximum de la recherche selon la formule valeur*1,28 secondes. Dans notre exemple, la valeur 48 nous donne 48*1,28=61,44 secondes.

Si le nombre de périphériques détectés est atteint avant que ce délai ne soit écoulé, la recherche s'arrête.

Les périphériques détectés, ici un seul, voient leur adresse matérielle s'afficher au format **NAP:UAP:LAP**. Celui que nous avons détecté a donc pour adresse **00:19:5D:EE:A3:F0**. Nous pouvons en profiter pour demander son nom avec :

```
AT+RNAME?0019,5D,EEA3F0
+RNAME:WIOE_HK
OK
```

Oui, il s'agit d'un autre module Wide.hk. Remarquez que nous sommes là très loin des possibilités offertes par des outils en ligne de commandes comme ceux existant sous GNU/Linux (**hcitool** ou **bluez-tools**). Il ne nous est pas possible d'obtenir d'autres informations sur le ou les périphériques distants, par exemple. Notez cependant que la commande **AT+INQ** nous retourne non seulement l'adresse, mais également la classe (ici **0x001f1f**) et le RSSI en dernière position.

Nous devons ensuite spécifier par **AT+PSWD=** le code PIN à utiliser pour l'association. Bien entendu, celui-ci doit correspondre au code utilisé par le périphérique distant. Nous pouvons alors procéder à l'association avec :

```
AT+PAIR=19,5D,EEA3F0,10
[...attente]
OK
```

La syntaxe est similaire à celle de toutes les adresses spécifiées par ailleurs avec, en plus, un délai maximum pour l'opération (ici 10 secondes). Si la manipulation réussit, la commande retourne **OK**, dans le cas contraire, ce sera **FAIL**. Un **OK** signifie donc l'association et le partage d'une *link key*. Il n'existe cependant pas de commande permettant de lister les périphériques pour lesquels il existe une association. Tout ce que nous pouvons faire, c'est afficher l'adresse du périphérique associé (authentifié) dernièrement utilisé :

```
AT+MRAD?
+MRAD:19:5d:eea3f0
OK
```

Nous voyons ici qu'il s'agit du périphérique que nous venons d'ajouter. À l'utilisation, il semblerait que la documentation fasse légèrement erreur. Il s'agirait plutôt de l'adresse la plus en haut de la pile FIFO des périphériques associés. Une autre commande permet de s'assurer qu'un périphérique, désigné par son adresse, est présent dans la liste des associations :

```
AT+FSAD=0019,5D,EEA3F0
OK
```

Là encore, un **OK** vaut « oui » et un **FAIL** vaut « non ». Enfin, la commande suivante permet de connaître le nombre d'entrées dans la liste de périphériques associés :

```
AT+ADCN?
+ADCN:6
OK
```

Une commande est décrite dans la documentation, permettant de vider la liste : **AT+RMSAD**. Celle-ci ne prendrait pas d'argument et devrait retourner **OK**. Les essais ont cependant montré qu'une telle commande n'existe pas dans le firmware que nous avons utilisé. Pour vider la liste, la procédure est donc la suivante :

- Afficher le nombre de périphériques présents dans la liste avec **AT+ADCN?**.
- Afficher le dernier périphérique entré dans la liste avec **AT+MRAD?**.
- Utiliser l'adresse affichée pour supprimer le périphérique avec **AT+RMSAD=** suivi de l'adresse en utilisant la syntaxe **NAP,UAP,LAP** (virgules et non double-points, tant pis pour le copier/coller).
- Boucler sur ces trois étapes jusqu'à obtenir :

```
AT+ADCN?
+ADCN:0
OK

AT+MRAD?
+MRAD:0:0:0
OK
```

Ceci revient à, tout simplement, dépiler la liste des périphériques associés. La documentation ne précise pas le nombre maximum de périphériques qu'il est possible de stocker dans la liste. On préférera alors la vider intégralement avant de nous lancer dans une nouvelle association (notez que **AT+ORGL** ne vide pas la fameuse liste, alors que cela aurait été souhaitable). Suite à l'association, le code retour doit être **OK**. Si vous obtenez **FAIL**, allongez le délai, vérifiez l'adresse et le code PIN, ainsi que la distance séparant les deux périphériques. Le module est de classe 2 avec une portée théorique de 10 mètres et une antenne très basique. Ce type de circuit est très sensible aux interférences. Tentez également une commande **AT+RNAME?** pour vous assurer de la bonne communication.

Une fois l'association effectuée, vous devez retrouver les informations suivantes :

```
AT+STATE?
+STATE:PAIRED
OK

AT+ADCN?
+ADCN:1
OK

AT+MRAD?
+MRAD:19:5d:eea3f0
OK
```

À ce stade, vos deux périphériques sont associés. vous pouvez alors utiliser les trois dernières commandes de configuration pour ce rôle :

```
AT+BIND=19,5D,EEA3F0
OK

AT+BIND?
+BIND:19:5d:eea3f0
OK

AT+CMODE=1
OK

AT+ROLE=1
OK
```

Dans l'ordre, nous spécifions une adresse de périphérique pour la connexion automatique en mode maître selon la syntaxe classique, puis nous définissons le mode de connexion. Vous avez le choix entre **0** pour une connexion à n'importe quel périphérique à portée ou

1 pour la connexion à l'adresse spécifiée par **AT+BIND**. Enfin, nous spécifions le nouveau rôle du module : maître.

Assurez-vous une dernière fois que les paramètres de l'UART soient identiques pour les deux modules, puis coupez l'alimentation, repassez en mode standard pour enfin rétablir l'alimentation.

Automatiquement, vous devez voir la connexion s'établir en quelques secondes et les deux modules passer au statut connecté (LED clignotant deux fois toutes les secondes). Si vous utilisez un terminal série de chaque côté, ce que vous tapez dans l'un apparaîtra dans l'autre et inversement. Vous venez d'établir une connexion RFCOMM/SSP automatique entre deux modules et vous n'avez plus qu'à vous soucier de la gestion de la liaison série.

5 Autres commandes et fonctionnalités

La documentation référence d'autres commandes intéressantes et en particulier celle permettant de sécuriser la communication. Malheureusement, le détail n'est pas au rendez-vous et nous devons faire avec le peu d'informations à notre disposition. La commande **AT+SENM** nous permet de chiffrer la communication :

```
AT+SENM?
ERROR:(1)

AT+SENM=3,1
OK

AT+SENM?
+SENM:3,1
OK
```

Notez que l'erreur à la première commande n'en est pas vraiment une. Le code d'erreur **1** correspond aux « valeurs par défaut ». La commande prend deux arguments, le premier est le mode de sécurité avec au choix :

- **0** : désactivé ;
- **1** : non sécurisé ;

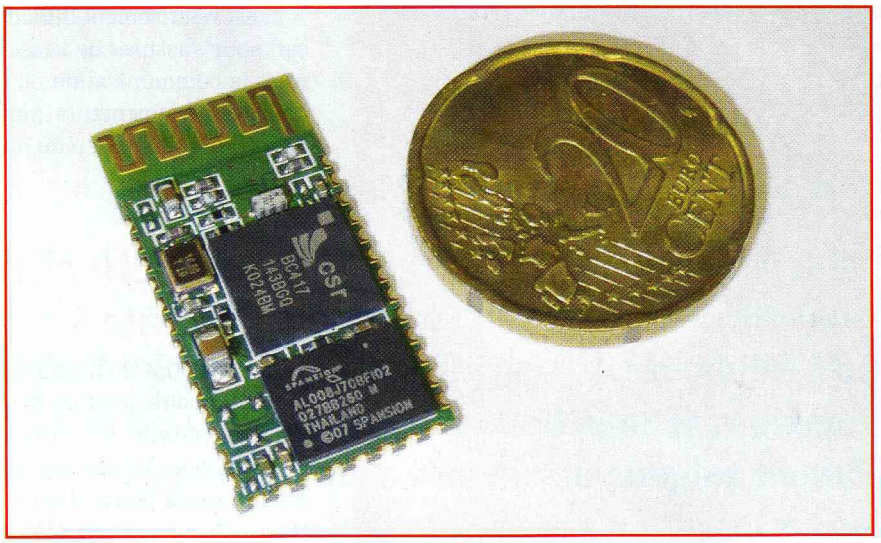


Figure 3

- **2** : niveau service ;
- **3** : niveau lien.

Le second argument définit le chiffrement :

- **0** : désactivé ;
- **1** : point à point ;
- **2** : point à point et *broadcast*.

Pour comprendre de quoi il s'agit, c'est à une documentation NIST appelée *Guide to Bluetooth Security* (référence SP-800-121) qu'il est préférable de se reporter (la lecture du document de 43 pages est, par ailleurs, très intéressante). Le premier argument reprend tout simplement les numéros des modes de sécurité de la norme :

- Niveau 1 : Toutes les fonctionnalités d'authentification et de chiffrement sont désactivées. Techniquement, le périphérique dans ce mode se trouve dans un état similaire du mode *promiscuous* des interfaces Ethernet. Ce mode n'est disponible que pour les périphériques Bluetooth 2.0+EDR et inférieurs.
- Niveau 2 : Les mécanismes de sécurisation prennent place entre LMP et les canaux L2CAP. Ces mécanismes permettent de gérer une politique d'accès aux services. Il sont implémentés au niveau LMP (sous L2CAP) exactement comme le niveau trois, mais ce sont bien les services qui sont « protégés ». Un périphérique Bluetooth 2.1+EDR ne supporte normalement ce mode que dans un souci de compatibilité avec 2.0+EDR.
- Niveau 3 : Ici, les mécanismes de sécurisation entrent en jeu avant qu'une liaison physique ne soit pleinement établie. Ce niveau suppose un chiffrement et une authentification et permet de protéger au mieux la communication entre les périphériques.

Le mode niveau 4 ajoutant le chiffrement par courbes elliptiques n'est pas proposé par la configuration car introduit avec Bluetooth 2.1+EDR. Il est censé remplacer le mode 2.

Le second argument, lui aussi, reprend les trois possibilités offertes par la norme et décrites dans la documentation NIST :

- Mode 1 : Pas de chiffrement du tout quel que soit le type de trafic.
- Mode 2 : Seules les informations échangées de périphérique à périphérique sont chiffrées, avec une clé dérivée de la link key.
- Mode 3 : L'ensemble du trafic est chiffré, y compris celui partagé par l'ensemble des périphériques présents (*broadcast*).

ANNEXE : Les codes d'erreurs

- 0 : Commande AT inconnue.
- 1 : Le résultat de la commande correspond à la valeur par défaut (ou pas).
- 2 : Erreur d'écriture de la *PSKEY* (link key).
- 3 : Nom de périphérique trop grand. Maximum 32 caractères.
- 4 : Le nom du périphérique a une taille nulle.
- 5 : Composante NAP de l'adresse trop grande (max 16 bits).
- 6 : Composante UAP de l'adresse trop grande (max 8 bits).
- 7 : Composante LAP de l'adresse trop grande (max 24 bits).
- 8 : Le masque du port E/S est 0.
- 9 : Port E/S invalide.
- A : La classe du périphérique (CoD) a une taille nulle.
- B : Classe de périphérique de taille nulle.
- C : L'IAC (*Inquire Access Code*) a une taille nulle.
- D : L'IAC est trop grand.
- E : IAC invalide.
- F : Le code PIN a une taille nulle.
- 10 : Le code PIN est trop grand. Max 16 octets.
- 11 : Le rôle spécifié est invalide.
- 12 : La vitesse (UART) est invalide.
- 13 : La configuration du bit de stop est invalide.
- 14 : La parité spécifiée est invalide.
- 15 : Aucun périphérique associé présent dans la liste.
- 16 : SPP (*Simple Pairing Protocol*) non initialisé.
- 17 : SPP déjà initialisé.
- 18 : Mode de recherche invalide.
- 19 : Temps d'attente dépassé pour la recherche.
- 1A : L'adresse est 0.
- 1B : Le mode de sécurité est invalide.
- 1C : Le mode de chiffrement est invalide.

Il est relativement difficile de dire ce que fait exactement le module car, pour s'assurer de la sécurisation effective, il faudrait tenter de pirater la communication ou, du moins, de l'espionner. On ne peut que croire la documentation sur parole car, à l'utilisation simple, `AT+SENM` ou pas ne change rien du tout (et c'est bien normal).

Conclusion, critiques et espérances

Nous venons de faire le tour de ce qu'il est possible de faire avec ce type de module pour un firmware donné. Comme souvent avec du matériel de ce type, le rapport prix/fonctionnalité est très bas, mais les informations d'utilisation ne sont pas toujours très claires ou fiables. Ainsi, dans la phase d'expérimentations ayant permis la rédaction de cet article, un certain nombre de points sont restés sans réponse.

Le périphérique, celui-ci ou un modèle similaire, est vendu comme répondant à la norme Bluetooth 2.0+EDR, mais d'un autre côté, il semble également intégrer des éléments de la version 2.1+EDR. Ceci correspond avec les documentations du composant BlueCore4-external qui est à la base du module qui lui est donné comme étant compatible 2.1+EDR.

La documentation est généralement le point faible de ce type de produit. Ainsi, le choix d'annoncer `ERROR(1)` pour signaler une configuration identique aux valeurs par défaut est documenté mais pas toujours respecté. Dans mon lot, j'ai reçu deux modules identiques, affichant la même version (`1.0-20090818`), l'un répondant `ERROR(1)` à la commande `AT+SENM?` et l'autre `+SENM:0,0` (qui correspond pourtant à la valeur par défaut dans la documentation). Notons au passage que l'utilisation du terme « error » n'est, de plus, pas très pertinente.

Dans le même registre, la commande `AT+RMSAD`, sans argument, n'existe tout simplement pas. Il peut s'agir d'une faute de frappe dans la documentation, mais après avoir essayé quelques variations et vérifié d'autres documentations, je n'ai pas trouvé le nom de la commande permettant effectivement de purger la liste des périphériques associés.

D'autres reproches peuvent être faits à cette gamme de modules, mais nous ne devons pas oublier leur coût relativement réduit. On regrettera surtout l'aspect nébuleux entourant ce matériel. Qui fabrique réellement ces modules (sans breakout board) ? Qui a écrit le firmware qui est utilisé partout ? Existe-t-il une manière de le mettre à jour ? Les sources sont-elles disponibles quelque part ?

Enfin, pour conclure, puisqu'il est temps de le faire, notez que SeeedStudio, un autre diffuseur de modules électroniques, dispose également d'un produit similaire (WLS123A1M), mais équipé d'un firmware totalement différent. Le module qui est censé se voir soudé sur un *shield* Arduino (vendu séparément) est alimenté en 3,3V et vendu quelque $21 (contre <$13 pour le module Wide.hk avec breakout).

Mais finalement, le plus important est au rendez-vous : cela fonctionne. Et ce, aussi bien entre deux modules qu'avec un adaptateur USB Bluetooth sur PC ou encore avec un smartphone Android (cf. article quelques pages plus loin). ■

PRISE EN MAIN DE L'ADAPTATEUR SENA PARANI-SD1000U

par Denis Bodor

Nous l'avons vu dans l'article d'introduction, la mise en œuvre d'une pile de protocoles Bluetooth peut s'avérer très lourde et gourmande en ressources pour un système embarqué de faible capacité. D'autre part, des modules à base de puce BlueCore4 comme celui de Wide.hk permettent d'ajouter rapidement une connectivité Bluetooth à un système, mais restent de simples modules. Et s'il était possible de réunir les deux avec un équipement intégrant toute la gestion du Bluetooth, mais se présentant comme une simple clé USB/série ? Voilà exactement ce que propose le module SD1000U de SENA.

SENA Technologies conçoit, développe et produit des équipements industriels M2M (*machine-to-machine*) dans le domaine du wireless et en particulier le Bluetooth et ZigBee. Parmi les différents produits de la société se trouve toute une gamme d'adaptateurs Bluetooth intelligents, la plupart interfacés en RS-232. Un modèle en particulier a retenu notre attention : le Parani SD1000U. Il s'agit d'une clé USB toute rouge fort intéressante. C'est un adaptateur série supportant Bluetooth 2.0+EDR, SPP (*Serial Port Profile*) et le FHSS (*Frequency Hopping Spread Spectrum*). Son principal avantage est de ne nécessiter aucun pilote particulier sur le système auquel il est connecté, si ce n'est le support FTDI pour le port série virtuel, chose que tous les systèmes d'exploitation, et en particulier GNU/Linux, intègrent nativement ou sous la forme d'un pilote supplémentaire générique.

Une autre caractéristique intéressante du produit est la présence d'une antenne amovible via un connecteur RP-SMA très standard. L'antenne équipant le périphérique dans sa version par défaut dispose d'un gain de 3 dBi. En choisissant une autre antenne disposant d'un gain plus important, il devient possible d'atteindre des portées bien au-delà de la norme Bluetooth, de l'ordre de 600 mètres voire 1000 mètres, avec une antenne directionnelle !

Côté matériel, la clé dispose de plusieurs éléments de configuration et de monitoring :

- un bouton de reset aux valeurs d'usine en cas de mauvaise configuration ;
- un bouton d'association (voir plus loin dans l'article) ;
- trois micro-switch permettant la configuration de la vitesse de la liaison série ;
- un micro-switch pour la configuration du contrôle de flux ;
- et une série de trois LED permettant de visualiser l'état du système.

La configuration des switch est donnée ci-contre. Notez qu'une configuration spécifique des micro-switch dédiée à la sélection de la vitesse permet de basculer le mode de configuration en version logicielle par l'intermédiaire de commandes AT (voir plus loin dans l'article).

1 Modes de fonctionnement

La clé SD1000U dispose de plusieurs modes de fonctionnement qu'il est possible de visualiser rapidement via la LED « Mode » en façade :

- Mode 0 : la LED reste allumée de manière constante ;

source : documentation/manuel SENA Technologies

Table 3-4 Baud rate Settings by Dipswitches

Baud rate	2400	4800	9600	19.2K	38.4K	57.6K	115.2K	S/W Config

Table 3-5 Hardware Flow Control Settings by Dipswitches

Hardware Flow Control Handshaking	No Use	Use

documentation/manuel SENA Technologies

- Mode 1 : un clignotement toutes les secondes ;
- Mode 2 : deux clignotements toutes les secondes ;
- Mode 3 : trois clignotements toutes les secondes.

Lorsque la clé est connectée avec un périphérique distant, c'est la LED « Connect » qui clignote toutes les secondes.

Le mode 0 est le mode par défaut de la clé. C'est le mode permettant la configuration du périphérique via les commandes AT. Il n'y a pas de connexion avec des périphériques Bluetooth dans ce mode, ni de communication transparente via la liaison série.

En mode 1, la SD1000U essaie de se connecter au dernier périphérique Bluetooth utilisé. La clé se comporte alors comme maître dans la connexion et utilise l'adresse mémorisée du dernier périphérique connecté. Par défaut, il n'y a pas d'adresse mémorisée, il en va de même suite à un reset aux valeurs d'usine via le bouton dédié (appui d'une seconde minimum). Le passage au mode 1 ne peut se faire que si la clé s'est connectée avec succès à un périphérique distant. Une fois passé en mode 1, la simple connexion de la clé à un port USB provoque la tentative de connexion. Remarquez qu'il n'y a pas de procédure de recherche de périphériques dans ce mode et qu'il n'est pas possible manuellement de connecter un périphérique distant.

Le mode 2 de la clé est dédié à l'attente de connexion de la part du dernier périphérique distant utilisé. Le rôle du SD1000U est celui d'esclave, mais la connexion n'est possible QUE depuis le dernier périphérique utilisé exactement comme dans le mode 1. L'adresse du périphérique en question est stockée dans la mémoire de la clé. Là encore, la configuration, la découverte et la connexion à un périphérique arbitrairement désigné n'est pas possible. La clé ne sera pas non plus détectable (iscan) ou connectable (pscan). À la mise sous tension, connexion USB, la clé passera automatiquement en attente de connexion.

Enfin, dans le mode 3, la clé SD1000U attend les connexions de n'importe quel périphérique distant. Elle est donc détectable (iscan ou *inquiry scan*) et connectable (pscan ou *page scan*) par n'importe quel appareil Bluetooth maître.

2 Configuration sans ordinateur

Je précise d'entrée de jeu que ce qui suit n'a pas été testé car je ne dispose que d'une seule clé SD1000U. Il est possible d'auto-configurer la clé sans avoir recours à un ordinateur et donc une connexion USB via le bouton « Pairing ». La procédure est la suivante (dixit la documentation) :

- Brancher et allumer (via une alimentation USB, je suppose) les deux clés et procéder à un reset (1 seconde minimum).
- Appuyer sur le bouton « Pairing » et maintenir pendant 2 secondes jusqu'à ce que la LED « Mode » clignote trois fois toutes les trois secondes, sur la première clé.
- Procéder de même sur la seconde clé puis appuyer encore jusqu'à un clignotement toutes les secondes.
- Attendre que la connexion s'établisse entre les deux clés. Ceci sera indiqué par un clignotement toutes les secondes sur les deux SD1000U. Cela peut prendre du temps dans un environnement peuplé de nombreux périphériques Bluetooth.
- Déconnecter et reconnecter la première clé qui doit alors clignoter deux fois toutes les 3 secondes.
- Déconnecter et reconnecter la seconde clé qui doit clignoter toutes les secondes.
- La liaison est configurée et la connexion USB des deux clés doit alors provoquer une liaison automatique.

Ce processus de configuration peut être intéressant pour deux systèmes devant établir une connexion sans avoir d'interface permettant de travailler avec des commandes AT. Il s'agit cependant d'un cas très particulier nécessitant, bien entendu, l'usage de deux SD1000U.

3 Commandes AT et configuration

Le jeu de commandes AT, également appelées commandes Hayes, forme un langage qui trouve ses origines dans la communication par modem. Ce type de communication, maintenant un peu désuet, était le moyen courant de faire communiquer des systèmes (ordinateur) via des modems connectés à des lignes téléphoniques. Pour tout dire, la découverte et l'apprentissage des commandes AT a constitué mon entrée dans le monde de l'informatique professionnel et l'usage, encore aujourd'hui, de ces commandes n'est pas sans laisser une douce impression de nostalgie. À l'époque, les équipements chargés de convertir les données en modulation capables de circuler sur des lignes téléphonique analogique, les modems, répondaient à un jeu de commandes initialement implémentées dans les produits de la marque Hayes, et en particulier le Smartmodem. Les autres fabricants calquaient alors le mode de contrôle de leurs équipements sur ce même jeu de commandes.

Le principe est fort simple. Contrôler un modem se faisait en dialoguant avec l'appareil via la connexion série. À la mise sous tension de l'appareil, il fallait alors configurer les paramètres de connexion (modulation, gestion du contrôle de flux, vitesse, parité, décrochage de la ligne, etc.) avec des commandes débutant toutes par `AT` et se terminant par le marqueur de fin de ligne CR+LF. Les commandes étaient soit sans paramètres, soit accompagnées d'arguments. Cette dernière catégorie se divisait en deux :

- Les commandes prenant en argument des données « ponctuelles » comme un numéro de téléphone. La commande était alors suivie directement du paramètre.
- Les commandes permettant de configurer une fonctionnalité et dont l'argument restait en mémoire. Ces commandes utilisaient alors la syntaxe `ATxxx=argument`.

Ainsi, il était possible de tout faire :

- Composer un numéro de téléphone.
- Commander le raccordement du modem à la ligne.
- Connaître l'état de la ligne.
- Configurer le type de transmission et le protocole à utiliser.
- Régler le volume sonore du haut-parleur intégré.
- Afficher certaines informations concernant le modem.

Le modèle de commandes AT était tellement bien pensé qu'on le retrouve encore aujourd'hui, et en particulier avec les équipements offrant des fonctionnalités identiques aux modems. Ainsi, la clé SD1000U reprend ce « langage » pour sa configuration et la gestion des connexions Bluetooth. Bien entendu, ce produit n'est pas le seul, il en va de même pour les autres équipements du fabricant, mais également les périphériques de bien d'autres constructeurs.

Faisons un petit tour des commandes avec, tout d'abord, quelques opérations de base :

```
AT
OK

AT+BTVER?
SD1000Uv2.0.1
OK

AT+BTLAST?
000000000000
OK

AT+BTINFO?
0001950CA617,SD1000Uv2.0.1-
0CA617,MODE0,STANDBY,0,0,NoFC
OK
```

La commande **AT** seule n'a pour but que de vérifier le bon fonctionnement de la configuration et de la connexion entre le PC et la clé. Ici, notre connexion en 9600 8N1 via **screen /dev/ttyUSB0 9600** fonctionne sans problème. La commande suivante permet de vérifier le numéro de version du firmware embarqué. Celui-ci peut être mis à jour via un outil Windows appelé ParaniUpdater. Je n'ai pas connaissance d'un équivalent pour GNU/Linux ou encore Mac OS X.

AT+BTLAST? nous permet d'afficher l'adresse du dernier périphérique connecté avec succès. Bien entendu, avec une clé neuve ou suite à un reset aux valeurs d'usine, cette mémorisation est inexistante. Enfin, **AT+BTINFO?** est la commande nous permettant d'obtenir les éléments les plus utiles concernant l'état du périphérique :

- **0001950CA617** : l'adresse de la clé SENA ;
- **SD1000Uv2.0.1-0CA617** : le numéro de version complet du firmware (avec les 6 derniers chiffres de l'adresse) ;
- **MODE0** : le mode actuel ;
- **STANDBY** : l'état entre **STANDBY**/**PENDING**/**CONNECT** ;
- **0** : authentification activée (**1**) ou non (**0**) ;
- **0** : le chiffrement activé (**1**) ou non (**0**) ;
- **NoFC** : le contrôle de flux matériel (**HwFC**) ou aucun (**NoFC**).

Passons maintenant aux choses sérieuses avec une recherche de périphériques :

```
AT+BTINQ?
001010160024,HC-05-LEF001,800510
D03761C357AE,MetaWatch Digital WDS112,001F00
OK
```

Nous avons trouvé deux équipements Bluetooth répondant à un *inquiry scan*. Le premier est un module Wide.hk et nous pouvons alors spécifier un code PIN puis tenter une connexion :

```
AT+BTKEY=0000
OK

ATD001010160024
OK
CONNECT 001010160024
G
G ok
Z
Z ok
+++
OK
```

Cela se passe exactement comme pour un modem analogique (de la belle époque). **ATD** est utilisé pour « numéroter » et la commande est confirmée par **OK**. Nous attendons ensuite que l'étape de connexion soit passée (authentification et association) et nous obtenons **CONNECT**. Dès lors, nous parlons directement avec notre montage à l'autre bout de la liaison RFCOMM/SPP. Ici, nous envoyons les ordres **G** et **Z** comme dans tous les exemples utilisés dans ce numéro. Pour revenir au mode AT, nous utilisons le code d'échappement **+++** (comme avec un modem) et obtenons confirmation de l'opération avec un **OK**.

Nous pouvons alors vérifier quelques informations :

```
AT+BTLAST?
001010160024
OK

AT+BTINFO?
0001950CA617,SD1000Uv2.0.1-
0CA617,MODE0,CONNECT,0,0,NoFC
OK

ATO
[dialogue avec le module]
+++
OK

ATH
OK
DISCONNECT
```

Nous pouvons constater que l'adresse du dernier périphérique connecté avec succès est bien celle de notre module. La commande **AT+BTINFO?** nous montre qu'une connexion est établie. Nous pouvons alors utiliser **ATO** (O comme Oscar) pour rebasculer sur la liaison série avant d'utiliser à nouveau **+++** pour reprendre le contrôle, et enfin, nous servir de **ATH** pour « raccrocher » et couper la connexion. À ce stade et à présent qu'une entrée dans la liste de connexion est mémorisée, utiliser simplement **ATD** sans argument nous reconnectera au module.

Après plusieurs connexions à des périphériques différents, nous pouvons obtenir la liste des périphériques associés avec :

```
AT+BTSD?
001010160024
00195DEEA3FF
OK
```

Une fois ces premières manipulations effectuées, nous pouvons choisir un mode avec la commande **AT+BTMODE,n** où **n** est le numéro du mode choisi. Ainsi, en passant au mode 1 via **AT+BTMODE,1**, automatiquement, la clé se connectera dès la mise sous tension au dernier périphérique utilisé.

Notez au passage que la description de la commande **AT+BTSD?** n'est pas totalement juste dans la documentation : « *Display a list of Bluetooth devices sharing the same pin code* ». Il n'est, en effet, normalement plus question de code PIN

à ce stade, mais uniquement de *link key*. Nos deux périphériques listés utilisent deux codes PIN différents. La valeur de **AT+BTKEY** n'a aucune importance une fois un périphérique associé. **AT+BTCS**, vidant la liste des périphériques associés, est décrite de la même manière dans la documentation.

Nous utilisons ici un terminal série pour dialoguer avec la clé. SENA met également à disposition une application, ParaniWIN, permettant de simplifier la communication en remplaçant ses échanges par une interface graphique.

4 Connexions multiples

Un autre gros avantage de la clé SD1000U est sa capacité à gérer de multiples connexions simultanées. Deux modes de fonctionnement de ce type sont utilisables. Le premier mode appelé *Multi-Drop Mode* permet à la clé maîtresse de la connexion d'envoyer et recevoir des données jusqu'à quatre périphériques esclaves en même temps. Notez qu'elle peut aussi recevoir des données et qu'il est à la charge de l'application que vous développez de faire la différence entre la provenance de ces données. C'est là un mode de fonctionnement qui peut être intéressant si vous mettez en œuvre un ensemble de capteurs connectés en Bluetooth.

L'autre mode de connexion multiple est appelé *Node Switching Mode*. Dans ce cas, la clé maintient une communication avec jusqu'à quatre périphériques et vous choisissez avec lequel vous dialoguez. Voici un exemple, nous commençons par passer dans ce mode :

```
AT+MULTI,2
TASK1 OK
TASK2 OK
TASK3 OK
TASK4 OK
```

Dès lors, nous sommes en mesure de nous connecter à deux périphériques :

```
ATD001010160024
OK
CONNECT 001010160024
+++
OK

ATD00195DEEA3FF
OK
CONNECT 00195DEEA3FF
```

Notez que la seconde connexion ne nous met pas directement en liaison. Nous pouvons alors lister les connexions :

```
AT+MLIST?
CURRENT MODE: NODE SWITCHING MODE
TASK1 - 001010160024
TASK2 - 00195DEEA3FF
TASK3 - DISCONNECT
TASK4 - DISCONNECT
```

Puis choisir l'une ou l'autre :

```
ATD1
[dialogue avec 001010160024]
+++
OK

ATD2
[dialogue avec 00195DEEA3FF]
+++
OK
```

En utilisant le code d'échappement et en revenant au mode de commande AT, nous pouvons nous déconnecter d'un des périphériques :

```
ATH1
DISCONNECT 001010160024

AT+MLIST?
CURRENT MODE: NODE SWITCHING MODE
TASK1 - DISCONNECT
TASK2 - 00195DEEA3FF
TASK3 - DISCONNECT
TASK4 - DISCONNECT

ATO1
ERROR

ATH2
DISCONNECT 00195DEEA3FF
```

Notez que l'utilisation de **ATO** (ou **ATH**) sur un périphérique non connecté provoque l'affichage de **ERROR**. Pour clore l'exemple, nous revenons au mode standard mono-connexion :

```
AT+MULTI,0
OK

AT+MLIST?
CURRENT MODE: SINGLE CONNECTION MODE
```

Les commandes **ATH** et **ATO** peuvent également se voir complétées de l'adresse du périphérique désigné plutôt que son numéro. Une autre méthode pour la multi-connexion passe par les registres de configuration de la clé, visibles comme sur un modem via la commande :

```
AT&V
S0: 0; S1: 1; S2: 0; S3: 0; S4: 1; S5: 0;
S6: 0; S7: 0; S8: 0; S9: 0; S10: 1; S11: 1;
S12: 1; S13: 1; S14: 1; S15: 0; S16: 0;
S17: 0; S18: 0; S19: 3; S20: 0; S21: 0;
S22: 3; S23: 0; S24: 15; S25: 0; S26: 0;
S27: 0; S28: 43; S29: 0; S30: 300; S31: 20;
S32: 600; S33: 30; S34: 5; S35: 39; S36: 4353;
S37: 5; S38: 0; S39: 1024; S40: 72; S41: 1024;
S42: 128; S43: 001F00; S44: 000000; S45: 9e8b33;
S46: 001010160024; S47: 0; S48: 5000; S49: 4500;
S50: 4; S51: 4; S52: 5; S53: 0;
S54: 00195DEEA3FF; S55: 000000000000;
S56: 000000000000; S57: 3; S58: 0; S59: 2;
S60: 5; S61: 0; S62: 1
```

Pour ce type d'utilisation et l'usage des registres de configuration, consultez l'annexe B de la documentation officielle. Décrire les quelque 32 registres et leur utilisation reviendrait, en effet, à paraphraser le manuel qui, sur ce point, est relativement explicite.

Conclusion

La clé SD1000U est une solution complète permettant de grandement faciliter la mise en œuvre du Bluetooth sur un système ne disposant d'absolument aucun support de ce type. Pour environ 100 euros TTC, vous pourrez donc ajouter du Bluetooth à votre système sans avoir à intégrer, dans le cas d'un GNU/Linux, une pile BlueZ et tous les outils qui l'accompagnent. À vous ensuite de faire le calcul et ainsi déterminer si le coût de ce périphérique vaut ou non le travail d'intégration d'un support Bluetooth dans le système.

Nous n'avons ici traité que le cas de la clé USB, mais sachez que des déclinaisons existent pour RS232 ou une liaison série TTL. La mise en place de ce genre de configuration sera alors sans doute plus évidente car, dans de nombreux cas, un système implémentant déjà un support USB hôte a de fortes chances d'être en mesure de supporter BlueZ.

Une fois encore, c'est à vous qu'il revient de juger de la pertinence d'une telle solution en fonction de votre cahier des charges et de l'énergie et du temps à votre disposition pour votre projet. Personnellement, ma clé SD1000U équipe une carte, récupérée gratuitement, au format PC104 à base de processeur MIPS. L'intégration d'une pile Bluetooth sous GNU/Linux était presque hors de question étant données les ressources disponibles et l'état du système intégré (noyau 2.4). Ainsi, le coût du système terminé était simplement celui de la clé SENA tout en étant modulaire et adaptable à une évolution possible du projet. ■

COMMUNICATION SÉRIE/BLUETOOTH POUR AVR/ARDUINO

Les essais et implémentations clientes d'utilisation du Bluetooth dans ce numéro, qu'il s'agisse de passerelle Bluetooth/série, de C ou de Java/android, reposent tous sur un montage décrit dans le précédent numéro. Nous n'allons pas vous faire l'insulte ici de redétailler ce montage relativement simpliste. Ce très bref article, en revanche, a pour but de servir soit de piqûre de rappel, soit de point de départ pour la lecture d'une documentation plus poussée concernant les microcontrôleurs AVR ou une platine Arduino.

Le montage utilisé comme périphérique est très simple. Il se sert du périphérique série intégré à la plupart des AVR d'Atmel pour recevoir une ligne via le port série et la décoder. Selon l'ordre reçu, le programme du microcontrôleur va alors afficher, sur une série de 8 afficheurs LED 7 segments, une valeur numérique (ou un motif quelconque). Comme base, nous avons deux ordres simples, **G\r** qui affiche « 12345678 » et **Z\r** pour afficher « 00000000 ». La manière de décoder ces ordres importe peu ici, nous nous contenterons de parler de la liaison série.

1 Arduino

Le code qui va suivre n'est qu'un simple squelette minimaliste d'une communication série avec Arduino.

```
void setup()
{
  Serial.begin(9600);
  Serial.flush();
  Serial.println("Coucou serial");
}

void loop (){

  int i=0;
  char buffer[100];

  if(Serial.available()){
     delay(100);
    while( Serial.available() && i< 99) {
       buffer[i++] = Serial.read();
    }
    buffer[i++]='\0';
  }

  if(i>0)
     Serial.println((char*)commandbuffer);
}
```

Comme tous programmes Arduino, il se compose de deux fonctions : **setup()** permettant de définir les éléments lors de la mise sous tension du montage, et **loop()**, appelé indéfiniment en boucle jusqu'à la rupture de l'alimentation ou un reset. L'utilisation du port série d'un Arduino est ici grandement simplifié par les bibliothèques Arduino qui se chargent de tout configurer à votre place (mais qui occupent un espace non négligeable en mémoire flash). Un simple **Serial.begin(9600)** suffit alors à configurer les lignes RX et TX pour une communication en 9600 bps. On notera l'utilisation de **Serial.flush()** permettant de vider le buffer interne.

La méthode **Serial.println("texte")** permet d'envoyer la chaîne de caractères spécifiée via la liaison série. La lecture nécessite un peu plus de code puisque les fonctionnalités fournies ne s'occupent que de signaler la présence de caractères à lire et de dépiler les caractères un par un. **Serial.available()** retourne 1 si au minimum un caractère est en attente. On utilise alors **Serial.read()** pour récupérer cet unique octet et le placer dans un tableau. Nous ne testons pas ici la présence d'un marqueur de fin de ligne et remplissons le tableau jusqu'à obtenir le compte d'octets attendu. Notez qu'il faut penser à laisser de la place pour la fin de chaîne, le fameux **\0**.

2 AVR et C

Avec un AVR, la bibliothèque AVR-Libc et son cerveau, les choses sont un peu plus complexes, mais également plus économes en mémoire flash utilisée (j'irai jusqu'à dire qu'il s'agit de vraie programmation). Il faut tout d'abord configurer l'USART de l'AVR (ici un Attiny2313) :

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

Il faut ensuite écrire le code de l'interruption déclenchée à la réception d'un caractère :

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

Nous utilisons une fonction **putchr()** pour afficher un echo de ce qui est reçu et envoyer un caractère à l'USART. La voici :

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

Enfin, comme notre ISR remplit elle aussi un tableau signalé comme disponible/utilisable avec une variable, nous avons besoin d'une fonction pour indiquer lorsque nous avons fini de traiter une chaîne disponible :

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

Je comprends tout à fait que ceci puisse paraître obscur, et si vous n'avez pas lu les numéros 2 (communication série avec les AVR) et 4 (afficheur 8 fois 7 segments), je vous en recommande la lecture ou, plus simplement, la visite de notre site unixgarden.com. ■

PROGRAMMER LE BLUETOOTH EN C SOUS GNU/LINUX

par Denis Bodor

Le titre est un peu maladroit, je le reconnais volontiers, mais allez résumer en une phrase courte le fait de développer vos programmes en C en utilisant les bibliothèques et donc la partie userland de l'implémentation de la pile Bluetooth, appelée BlueZ, telle qu'elle est développée pour le système GNU/Linux... Ouf ! Ça fait un sacré titre. Car c'est bien de cela qu'il s'agira de traiter dans le présent article. Du C, du Bluetooth et un système open source en logiciel libre qui fait le bonheur des développeurs et bidouilleurs en tous genres.

Pourquoi le C alors qu'il existe des *bindings* pour les bibliothèques BlueZ pour presque tous les langages (Perl, Python, Ruby, etc.) ? tout simplement, parce qu'on traite ici de systèmes embarqués et d'architecture aux ressources relativement réduites. Or les dépendances de BlueZ sont déjà relativement conséquentes si l'on compte mettre en œuvre une implémentation Bluetooth complète. Jugez plutôt :

```
% ldd /usr/sbin/bluetoothd
linux-gate.so.1 =>  (0xf7731000)
libbluetooth.so.3 => /usr/lib/i386-linux-gnu/libbluetooth.so.3 (0xf764b000)
libglib-2.0.so.0 => /lib/libglib-2.0.so.0 (0xf7572000)
libdbus-1.so.3 => /lib/libdbus-1.so.3 (0xf7534000)
libpthread.so.0 => /lib/i386-linux-gnu/i686/cmov/libpthread.so.0 (0xf751b000)
libcap-ng.so.0 => /usr/lib/libcap-ng.so.0 (0xf7516000)
libdl.so.2 => /lib/i386-linux-gnu/i686/cmov/libdl.so.2 (0xf7512000)
librt.so.1 => /lib/i386-linux-gnu/i686/cmov/librt.so.1 (0xf7509000)
libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xf73ae000)
libpcre.so.3 => /lib/libpcre.so.3 (0xf7371000)
/lib/ld-linux.so.2 (0xf7732000)
```

D-Bus, Glib, et PCRE (expressions régulières façon Perl) donnent déjà le ton. Le fait d'ajouter un interpréteur, fut-il « calibré » pour l'embarqué, ne peut être justifié que par deux choses (dont une mauvaise) : la profusion de ressources et une utilité autre dudit interpréteur. Une autre raison pour laquelle le C est une bonne solution est la suivante : c'est fun (traduisez librement en « amusant », « intéressant », « pédagogique », etc.). Nous allons donc explorer l'API BlueZ mise à notre disposition sous la forme de la **`libbluetooth.so.3`**. Si vous êtes utilisateur de Debian GNU/Linux, vous devrez installer le paquet **`libbluetooth-dev`** afin d'avoir à disposition les fichiers d'en-têtes vous permettant de développer des programmes.

Pas d'inquiétude sur les dépendances (D-Bus, Glib et autres). Ceci n'est pas l'affaire du développeur d'applications. C'est l'infrastructure BlueZ qui repose et fait usage de ces composants logiciels. Nous nous ne reposerons que sur cette bonne vieille bibliothèque GNU C et la **`libbluetooth`**.

1 En avant !

Cet article va se composer de deux codes distincts. Le premier que nous allons voir dans un instant sera une mise en bouche nous permettant de découvrir l'API BlueZ via la recherche de périphériques et l'obtention de quelques informations de base sur les périphériques à portée. Le second code consistera en une approche plus directe puisqu'il s'agira d'établir une connexion avec un périphérique RFCOMM afin de lui envoyer des données (exactement les mêmes périphérique et processus de test qui seront mis en œuvre dans l'article consacré au développement Android).

La bibliothèque BlueZ est relativement facile a utiliser. Tout repose sur la compréhension des différents *handles* et descripteurs, et ce à quoi ils servent, ainsi que sur la connaissance des différentes structures à utiliser.

Nous souhaitons procéder à un *scan* afin de découvrir les périphériques à portée. Pour cela, nous utiliserons la fonction **hci_inquiry()**. Nous n'allons cependant pas tout faire dans **main()** mais créer une fonction **doscan()** plus facile à réutiliser par la suite. La fonction principale de notre programme se résumera donc à :

```
int main(int argc, char *argv[]) {
  int dev_id, dd;

  dev_id = hci_get_route(NULL);
  dd = hci_open_dev(dev_id);

  if (dev_id < 0 || dd < 0) {
    perror("opening socket");
    exit(EXIT_FAILURE);
  }

  doscan(dd, dev_id);

  hci_close_dev(dd);
  return(EXIT_SUCCESS);
}
```

Nous avons là déjà quelques éléments de BlueZ puisque nous passons en argument de notre fonction deux arguments qui sont respectivement l'identifiant de ressource de notre adaptateur Bluetooth (HCI) local et une socket pour l'accès à ce matériel. Il s'agit d'un accès au contrôleur présent localement sur le système et non d'un périphérique distant.

Il existe plusieurs solutions permettant d'obtenir un accès à l'adaptateur local. **hci_get_route(NULL)** est la solution la plus simple et la plus efficace puisque cette fonction permet d'obtenir automatiquement l'identifiant du premier adaptateur détecté. Si vous disposez de plusieurs adaptateurs dans le système, il peut être souhaitable de désigner explicitement celui que vous souhaitez utiliser avec, par exemple, **hci_devid("00:15:83:18:A0:D4")**. Il conviendra bien entendu de ne pas coder « en dur » l'adresse en question, mais de la lire dans un fichier de configuration ou en ligne de commandes. Cette fonction retourne un identifiant, tout comme **hci_get_route()**. Il est également possible de passer une **bdaddr_t** à **hci_get_route()**. Il s'agit d'une représentation interne d'une adresse de périphérique Bluetooth. Deux fonctions utilitaires permettent la conversion vers et depuis une chaîne de caractères de type **"12:34:56:78:90:AB"** :

```
int ba2str( const bdaddr_t *ba, char *str );
int str2ba( const char *str, bdaddr_t *ba );
```

Une fois le numéro de ressource ainsi récupéré, on peut utiliser **hci_open_dev()** pour obtenir une socket vers l'adaptateur. Nous n'avons plus qu'à nous pencher sur notre fonction **doscan()**.

Nous souhaitons lancer une recherche de périphériques et cela tient en une seule fonction :

```
num_rsp = hci_inquiry(dev_id, len,
          max_rsp, NULL, &ii, flags);
```

num_rsp est un simple **int** destiné à recevoir le nombre effectif de périphériques qui ont été détectés. La fonction **hci_inquiry** prend en argument l'identifiant de ressource que nous avons récupéré précédemment. Comme le précise la documentation BlueZ, il s'agit là de l'une des très rares fonctions à utiliser cette information, les autres utilisant la socket. Nous fournissons également d'autres informations permettant le scan :

- **(int)len** est un multiplicateur d'un délai de 1,28 secondes permettant de spécifier un temps maximum (*timeout*) pour le scan. Ici, nous obéissons sagement aux recommandations de la documentation avec 8*1,27 secondes, soit un délai, décrit comme standard, de 10,24 secondes.
- **(int)max_rsp** est le nombre maximum de périphériques que nous souhaitons détecter. Si cette valeur (ici 255) est atteinte avant l'écoulement du timeout, le scan s'arrête.
- **(const uint8_t *)NULL** est normalement utilisé pour préciser le LAP permettant de faire un scan avec une IAC spécifique (autre que le GIAC). Cette utilisation est très peu courante puisque la quasi-totalité des périphériques utilisent le GIAC. **NULL** laisse le scan être fait « classiquement ».
- **(inquiry_info **)ii** est la destination ou plus exactement un pointeur sur la destination où vont être stockés les résultats du scan. Il faut prévoir un emplacement suffisamment gros pour éviter un débordement (cf. ci-après).
- **(long)flags** regroupe les drapeaux permettant de spécifier des options pour le scan. Pour l'heure, le seul drapeau utilisable est **IREQ_CACHE_FLUSH** (défini à **0x0001** dans **hci.h**), permettant de vider les informations en cache avant la recherche afin d'obtenir que des informations « fraîches » et récentes.

ii pour *Inquiry Info* est un pointeur obtenu via **malloc** avant le scan :

```
inquiry_info *ii = NULL;

ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));
```

La zone allouée est directement calculée avec le nombre maximal de périphériques que nous souhaitons obtenir

(**max_rsp**). Une fois le scan effectué, qui durera au maximum **len*1.28** secondes, nous devons obtenir (ou pas) un lot de **inquiry_info**. Il nous suffit alors de boucler pour analyse jusqu'à arriver à **num_rsp**.

```
for (i = 0; i < num_rsp; i++) {
}
```

Une partie des informations sur les périphériques se trouve dans **inquiry_info** :

```
typedef struct {
  bdaddr_t     bdaddr;
  uint8_t      pscan_rep_mode;
  uint8_t      pscan_period_mode;
  uint8_t      pscan_mode;
  uint8_t      dev_class[3];
  uint16_t     clock_offset;
} __attribute__ ((packed)) inquiry_info;
```

La première chose que nous allons afficher est un lot d'indications basiques : adresse, nom et classe (CoD) des périphériques. Les adresses et les classes des périphériques à portée ont été stockées dans **ii** à l'étape de recherche. En revanche, le nom des périphériques distants doivent faire individuellement l'objet d'une recherche :

```
if (hci_read_remote_name(dd, &(ii+i)->bdaddr,
  sizeof(name), name, 0) < 0)
    strcpy(name, "[unknown]");

printf("%s  %s (class 0x%02X%02X%02X)\n",
  addr, name, ii[i].dev_class[2],
  ii[i].dev_class[1], ii[i].dev_class[0]);
```

La fonction **hci_read_remote_name()** fait le travail souhaité mais ne retourne pas autre chose qu'un code signifiant la réussite ou l'échec. En revanche, elle prend en argument un pointeur sur un espace pour stocker ce nom. Nous avons ici utilisé une variable :

```
char name[248] = {0};
```

C'est largement suffisant car bon nombre de périphériques n'utilisent pas un nom dépassant 32 caractères. De plus, la taille est précisée (**sizeof(name)**). Notez qu'en cas d'erreur, le **name[]** se voit rempli avec une chaîne de caractères permettant de signaler le problème à l'utilisateur. Nous n'arrêtons cependant pas le programme car il est parfaitement possible qu'entre la recherche et la connexion pour récupérer le nom, le périphérique ne soit plus à portée. On peut donc supposer que, dans la suite du déroulement du programme, il puisse réapparaître et répondre à nos demandes. Notez également que pour récupérer le nom d'un périphérique distant, nous utilisons la socket **dd** et l'adresse cible sous forme de **bdaddr_t** récupérée dans la structure de type **inquiry_info**. Nous pouvons, si nous le souhaitons, utiliser **hci_read_remote_name** indépendamment d'une recherche, en fournissant les arguments manuellement.

Pour clore cette première étape, nous affichons ensuite joyeusement toutes ces informations avec un simple **printf()**. Nous ne nous arrêterons pas là, car d'autres éléments peuvent être intéressants dans une recherche de périphériques. C'est le cas, par exemple, de la version LMP de chaque matériel détecté. Pour obtenir ces informations, nous devons allez un cran au-delà dans notre rapprochement avec chaque périphérique. Maintenant, nous devons établir une connexion. Ceci n'a rien à voir avec un socket RFCOMM ou la communication avec un service, c'est une liaison directe au périphérique. Il n'y a donc pas d'association qui soit nécessaire. Pour se connecter à un périphérique de la sorte, nous utilisons la fonction :

```
if (hci_create_connection(dd, &(ii+i)->bdaddr,
      htobs(HCI_DM1 | HCI_DH1),
      0, 0, &handle, 25000) < 0) {
  perror("hci_create_connection: Can't create connection");
}
```

Le prototype est le suivant :

```
int hci_create_connection(int dd, const bdaddr_t *bdaddr,
uint16_t ptype, uint16_t clkoffset, uint8_t rswitch,
uint16_t *handle, int to);
```

avec comme arguments :

- **dd** est notre socket ;
- ***bdaddr**, l'adresse du périphérique distant au format **bdaddr_t** ;
- **ptype** : le ou les types de paquets autorisés (**hci.h**) ;
- **clkoffset** est le décalage d'horloge, généralement défini à 0 ;
- **rswitch** pour l'activation ou non de la capacité d'inversion de rôle (souvent 0) ;
- ***handle** est un pointeur sur un handle permettant de manipuler la connexion ;
- **to** est une valeur de timeout en millisecondes (les gars, vous auriez pu mettre **timeout** dans le prototype).

La documentation de Bluez est très maigre sinon inexistante pour la plupart des fonctions. Les fichiers d'en-tête ne fournissent pas grand chose d'autre que les prototypes de fonctions et les sources des bibliothèques elles-mêmes sont très pauvres en commentaires. Étrangement, c'est souvent dans le *binding* Java, JBlueZ, qu'on trouvera la plupart des réponses aux questions qu'on pourrait se poser.

Vous remarquerez l'utilisation de **htobs(HCI_DM1 | HCI_DH1)** dans notre appel de la fonction. Il s'agit de l'une des quatre fonctions utilitaires permettant la conversion des valeurs sur plusieurs octets. N'oubliez pas, le Bluetooth est en mesure de faire communiquer des machines aux architectures et systèmes très hétérogènes. Se pose alors un problème d'*endianness* dès lors qu'on dépasse 8 bits. La norme

Bluetooth définit l'ordre des octets (*endian*) comme étant du *little-endian* alors que le programme risque de fonctionner sur du *big-endian*, mais également du little-endian. Nous avons donc des fonctions de conversion du système hôte vers Bluetooth et Bluetooth vers hôte pour les **short** (16b) et les **int** (32b), respectivement :

```
unsigned short int htobs( unsigned short int num );
unsigned short int btohs( unsigned short int num );
unsigned int htobl( unsigned int num );
unsigned int btohl( unsigned int num );
```

Nous utilisons **htobs** afin de passer une valeur 16 bits définie en combinant les définitions précisées dans **hci.h**. Ici, nous spécifions DM1 et DH1 qui correspondent à des paquets de données sur le lien ACL, respectivement en vitesse moyenne (*Medium-rate*) et en haute vitesse (*High-rate*). Pour en savoir plus sur les types de paquets existants (au nombre de 13), je vous recommande la consultation de la page http://www.palowireless.com/infotooth/glossary.asp. Ici, nous nous en tenons au strict minimum étant donné ce dont nous avons besoin.

Une fois la connexion (HCI) établie avec le périphérique distant, nous pouvons utiliser la fonction **hci_read_remote_version()** pour obtenir enfin ce que nous cherchons :

```
struct hci_version version;

if (hci_read_remote_version(dd, handle, &version, 20000) == 0) {
  printf(" Manufacturer: %s (%d)\n",
    bt_compidtostr(version.manufacturer), version.manufacturer);
  printf(" LMP Version: %s (0x%x) LMP Subversion: 0x%x\n",
    lmp_vertostr(version.lmp_ver), version.lmp_ver,
    version.lmp_subver);
  hci_disconnect(dd, handle, HCI_OE_USER_ENDED_CONNECTION, 25000);
}
```

Notez cependant que la structure **hci_version** contient un peu plus de choses que de simples numéros de versions :

```
struct hci_version {
        uint16_t manufacturer;
        uint8_t  hci_ver;
        uint16_t hci_rev;
        uint8_t  lmp_ver;
        uint16_t lmp_subver;
};
```

Ce qui nous intéresse ici se limite à la couche LMP (*Link Management Protocol*) et au nom du fabricant (du contrôleur Bluetooth), qui peut être un élément utile dans l'identification d'un périphérique. Nous disposons, là encore, de fonctions utilitaires nous permettant de représenter ces informations sur la forme de chaînes :

- **bt_compidtostr(version.manufacturer)** pour le fabricant ;
- **lmp_vertostr(version.lmp_ver)** pour la version LMP.

Ces informations affichées, nous coupons la connexion avec **hci_disconnect()** en précisant une raison. Ici, **HCI_OE_USER_ENDED_CONNECTION**, équivalent à **0x13**, est un code erreur standard (voir **hci.h** pour les 55 autres) qui est presque une valeur passe-partout.

Ceci termine la boucle principale de notre fonction qu'on finalisera sans oublier un petit **free(ii)** avant le **return(0)**. Le programme développé pour cette première moitié d'article intègre également une fonction **printclassinfo(uint32_t bitmask)** permettant d'afficher une version décodée de la classe de périphérique. Se résumant à quelque 300 lignes de C principalement composées de **if** et de **switch**/**case**, j'estime qu'il n'y a pas grand intérêt à détailler tout cela.

Penchons-nous plutôt sur le **Makefile** :

```
TARGET  := main
WARN    := -Wall
CFLAGS  := -O2 ${WARN} `pkg-config bluez --cflags`
LDFLAGS := `pkg-config bluez --libs`
CC      := gcc

C_SRCS    = $(wildcard *.c)
OBJ_FILES = $(C_SRCS:.c=.o)

%.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@

$(TARGET): $(OBJ_FILES)
        $(CC) $(LDFLAGS) -o $@ $(OBJ_FILES)

all: ${TARGET}

clean:
        rm -rf *.o ${TARGET}
```

Et surtout un exemple d'utilisation du programme en question :

```
% sudo ./main
Start scanning...
Please wait 10.24 second(s) or 255 device(s) to be found...
done.

BC:B1:F3:7E:6E:FF  Galaxy Nexus (class 0x5A020C)
Manufacturer: Broadcom Corporation (15)
LMP Version: 4.0 (0x6) LMP Subversion: 0x4103
    Major Service Class : 0x5A0000
        Networking (LAN, Ad hoc, ...)
        Capturing (Scanner, Microphone, ...)
        Object Transfer (v-Inbox, v-Folder, ...)
        Telephony (Cordless telephony, Modem, Headset service, ...)
    Major Device Class : 0x000200|0x00000C
        Phone (cellular, cordless, payphone, modem...)
            Smart phone

00:10:10:16:00:24  HC-05-LEF001 (class 0x800510)
Manufacturer: Cambridge Silicon Radio (10)
LMP Version: 2.1 (0x4) LMP Subversion: 0x1735
    Major Service Class : 0x800000
        Information (WEB-server, WAP-server, ...)
    Major Device Class : 0x000500|0x000010
        Peripheral (mouse, joystick, keyboards...)
            keyboard/pointing device field : Not Keyboard / Not Pointing Device
            Peripheral Major Class : Sensing device
```

2 Et si on susurrait quelques mots doux à un périphérique ?

Le soleil descend à l'horizon, la lumière se fait moins vive, le ciel se change en un dégradé rouge-orangé magnifique. Vous êtes là, profitant de cet instant unique et éphémère, vous et votre périphérique, si proches l'un de l'autre (en fonction de sa classe). C'est le moment de trouver les mots qui conviennent...

Nous avons déjà fait la majorité du travail et connaissons quelques-unes des fonctions utiles. Le début de notre programme sera donc relativement standard :

```
int main(int argc, char *argv[]) {
  int dev_id, dd;
  char dest[18] = "00:10:10:16:00:24";
  struct sockaddr_rc destaddr = {0};
  int s, status, bytes_read;
  bdaddr_t  *bdaddr;
  char name[248] = {0};
  char buf[8] = {0};

  memset(name, 0, sizeof(name));

  bdaddr = malloc(sizeof(bdaddr_t));

  dev_id = hci_get_route(NULL);
  dd = hci_open_dev(dev_id);

  if (dev_id < 0 || dd < 0) {
    perror("opening socket");
    exit(EXIT_FAILURE);
  }
  if (str2ba(dest, bdaddr) < 0) {
    fprintf(stderr, "Can't convert address to bdaddr_t\n");
  } else {
    if (hci_read_remote_name(dd, bdaddr, sizeof(name), name, 0) < 0) {
      fprintf(stderr, "Can't get device's name\n");
      strcpy(name, "[unknown]");
    }
    printf("Remote device : %s", name);
  }
```

Cette première phase montre comment, à partir d'une adresse Bluetooth arbitrairement définie, nous pouvons obtenir une connexion simple dans le but de récupérer le nom du périphérique distant. Il ne s'agit que d'une autre version de ce que nous venons de voir dans la partie précédente. Notez que **bdaddr** et les fonctions qui s'y rapportent (**malloc()**) ne sont présentes que dans ce but.

En effet, pour établir la connexion RFCOMM, nous avons besoin d'utiliser une structure **sockaddr_rc**, définie dans **rfcomm.h** :

```
struct sockaddr_rc {
        sa_family_t     rc_family;
        bdaddr_t        rc_bdaddr;
        uint8_t         rc_channel;
};
```

Ceci vous rappelle peut-être quelque chose. En réalité, RFCOMM et TCP sont si proches sémantiquement que l'appel à la fonction **socket(int domain, int type, int protocol)** diffère très peu entre TCP et RFCOMM :

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

contre

```
s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

Tout ce que nous avons à faire est donc de créer une socket et de renseigner notre structure **sockaddr_rc** :

```
  printf("\nConnecting to remote device : %s\n", dest);
  s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
  destaddr.rc_family = AF_BLUETOOTH;
  destaddr.rc_channel = (uint8_t) 1;
  str2ba(dest, &destaddr.rc_bdaddr);
```

Comme dit précédemment, **bdaddr** ne nous est plus utile car **str2ba** utilise directement comme cible le champ **rc_bdaddr**. Nous pouvons ensuite utiliser **connect()** en fournissant un **sockfd**, une structure (*cast*) **sockaddr** et la taille des données. La valeur retournée par la fonction est zéro en cas de succès et **-1** en cas d'erreur, c'est classique :

```
  printf("connecting\n");
  status = connect(s, (struct sockaddr *)&destaddr,
    sizeof(destaddr));
  printf("status : %d\n", status);
  if(status == 0) {
    printf("Connected\n");
    write(s, "G\r", 2);
    sleep(2);
    write(s, "Z\r", 2);
  }
```

Il s'agit exactement de la même technique que celle utilisée pour l'écriture d'un client TCP. **write** est ici également utilisé pour écrire des données sur le descripteur de fichier **s**. Celles-ci seront alors envoyées au périphérique distant. Ici, nous procédons exactement de la même manière que pour le code de test sous Android (cf. article concerné) : nous envoyons l'ordre **G** affichant **12345678** puis, deux secondes après, l'ordre **Z** pour une remise à zéro de l'affichage.

Notre code s'arrête là. Nous terminons donc le programme sans oublier de fermer le descripteur de fichiers, couper l'accès à l'adaptateur et rendre la main :

```
  close(s);

  hci_close_dev(dd);
  return(EXIT_SUCCESS);
}
```

La lecture est sensiblement plus compliquée car, si lire des octets sur une socket est quelque chose de relativement facile, une connexion Bluetooth RFCOMM vers un montage électronique se fera souvent avec un protocole orienté

« lignes ». Il faudra donc traiter les lots de caractères reçus sous la forme de lignes. Une fonction classique pour ce type d'utilisation peut être :

```
ssize_t readLine(int fd, void *buffer, size_t n) {
  ssize_t numRead;
  size_t totRead =  0 ;
  char *buf;
  char ch;

  if (n <= 0 || buffer == NULL) {
    errno = EINVAL;
    return -1;
  }

  buf = buffer;

  for (;;) {
    numRead = read(fd, &ch, 1);
    printf("READ -> got a byte\n");
    if (numRead == -1) {
      if (errno == EINTR)
        continue;
      else
        return -1;
    } else if (numRead == 0) {
      if (totRead == 0)
        return 0;
      else
        break;
    } else {
      if (totRead < n-1) {
        if (ch == '\r') {
          printf("READ -> got EOL\n");
          break;
        } else {
          printf("READ -> got char=%u\n", ch);
          printf("READ ->  totRead=%u\n", totRead);
          totRead++;
          *buf++ = ch;
        }
      } else {
        break;
      }
    }
  }

  *buf = '\0';
  printf("READ -> returning %u\n", totRead);
  return totRead;
}
```

Il nous suffit alors d'utiliser quelque chose comme ceci pour lire une ligne envoyée par notre périphérique distant :

```
  printf("start reading...\n");
  bytes_read = 0;
  bytes_read = readLine(s, buf, sizeof(buf));
  if (bytes_read > 0) {
    printf("%u bytes readed\n", bytes_read);
    printf("string readed : \"%s\"\n", buf);
  }
```

Il existe bien d'autres approches à cette problématique puisque le principal avantage de cette utilisation du Bluetooth est la similarité avec la programmation des sockets INET qui, elle-même, permet d'utiliser des appels **`read()`** et **`write()`** exactement comme sur un fichier (philosophie UNIX : tout est fichier). Le problème revient donc à lire des lignes dans un fichier et les traiter.

Pour finir...

Nous venons de le voir rapidement, BlueZ met à notre disposition des fonctions nous permettant de très facilement communiquer avec un périphérique RFCOMM. Nous n'avons ici couvert que ce type de communication mais, vous vous en doutez, les bibliothèques BlueZ permettent de faire bien plus. Ainsi, SDP et la découverte de services sont implémentés. BlueZ n'a finalement que comme principal défaut son manque de documentation. C'est là un problème véritablement bloquant car certaines fonctions comme **`hci_read_rssi()`**, par exemple, ne sont tout simplement pas documentées. Pour tout dire, il manque un site, une doc ou n'importe quoi qui pourrait servir de référence à l'API.

Ce sont finalement les bindings Python et Java, ainsi qu'en partie la documentation développeur Android, qui permettent au développeur C de s'en sortir. Ceci, et la lecture du code d'autres développeurs ayant pris le temps de fouiller les sources ou ayant suivi le développement de BlueZ au fil du temps.

La documentation est un élément important, de nombreux projets s'en rendent compte à un moment ou un autre. Malheureusement, plus un projet fait face à cette nécessité tardivement, plus la masse de travail est énorme. Dans le cas de BlueZ, le document de référence est « An Introduction to Bluetooth Programming », un tutoriel écrit par Albert Huang et disponible à l'adresse http://people.csail.mit.edu/albert/bluez-intro/index.html. Comme le précise lui-même l'auteur, ce document a évolué avec le temps et s'est transformé en un livre édité par *Cambridge University Press* couvrant le développement sous GNU/Linux, mais également Windows, Symbian ou encore Mac OS X, et ce, avec différents langages. Le tutoriel pour GNU/Linux est toujours disponible mais, pour qui veut véritablement explorer le développement d'application gérant le Bluetooth, l'ouvrage est sans doute indispensable. Cependant, si vous êtes comme moi un peu allergique aux PDF avec DRM type *Adobe Digital Editions* (tout bonnement illisibles sous GNU/Linux) et réticent à l'idée d'attendre une version papier venant d'outre-Manche, il faudra vous contenter des exemples/code du livre disponibles sur http://www.btessentials.com/examples/examples.html.

Donc oui, l'absence de documentation pour BlueZ est un problème, mais au moins nous avons les sources. Le fait que *Cambridge University Press* choisisse de vendre un ouvrage traitant d'open source et de GNU/Linux dans un format électronique qu'il n'est pas possible de lire avec un applicatif open source est un tout autre problème... plus grave encore, à mon avis, car parfaitement incohérent. ■

ANDROID ET BLUETOOTH

par Denis Bodor

Nous venons de voir au fil des précédents articles que la prise en charge du Bluetooth est relativement aisée et permet toutes les réalisations. Qu'il s'agisse de la ligne de commandes ou du développement en C, il est ainsi possible de construire un ensemble permettant à un ordinateur de bureau, un serveur ou un système embarqué, de piloter et dialoguer avec un montage qui peut être très simple ou très complet. La dernière étape de notre voyage va nous amener dans un environnement bien différent mais tout aussi captivant : celui des smartphones et tablettes Android.

Avant de débuter la découverte du support Android pour le Bluetooth, procédons à quelques rappels et listons par la même occasion les prérequis pour appréhender cet article. Nous n'allons pas couvrir dans les paragraphes qui suivent l'installation du SDK Android, ni les manipulations de base permettant d'installer et retirer une application du système pour périphériques mobiles de Google. Je vous recommande, pour se faire, la lecture du guide d'introduction au développement Android présent sur le site officiel : http://developer.android.com/sdk/index.html.

Le développement Android est majoritairement dominé par le langage Java. Bien qu'il soit possible de développer des composants natifs en C via le NDK, Java reste un élément qu'il faut connaître. En effet, le développement C ne sert qu'à ajouter des fonctionnalités via JNI pour les applications et les services écris en Java. C'est là quelque chose qu'il faut accepter, même si comme moi, votre langage de prédilection reste le C.

Java est un langage objet où il est question de classes, de méthodes, d'instanciations, etc. Tout un patois spécifique, sinon une philosophie, accompagne donc le développement Android. Aux principes de base de Java doivent également s'ajouter ceux d'Android mais, curieusement, il est plus facile d'aborder ce langage lorsqu'on ne le connaît pas en utilisant un framework et une API claire plutôt que de se risquer dans les habituels tutoriels. En effet, il est universellement établi que l'apprentissage est plus facile lorsqu'il est motivé par le fait d'atteindre un objectif. Découvrir un langage comme Java en manipulant des classes **`vehicule`**, **`voiture`**, **`moto`** instanciant des objets du même type et disposant de méthodes **`demarrer`**, **`avancer`** ou **`stopper`** est bien trop abstrait et dénué de sens pratique. Si vous souhaitez démarrer l'apprentissage de Java et du développement Android, je ne saurais que trop vous conseiller de le faire « à la dure », en vous fixant un but et en passant par toutes les étapes nécessaires, quitte à développer en parallèle quelques applications de tests et d'expérimentation. Je vous conseillerais éventuellement de suivre les guides du site officiel Android, pour le développement de l'activité **`HelloAndroid`** et des différentes applications « Hello, Views » permettant de se familiariser avec les concepts de l'interface graphique d'Android. Il vous faudra cependant faire preuve de volonté car, même si les guides sont très bien conçus, l'absence d'objectif personnel réel rend cela terriblement ennuyeux (et puis vous finirez par les étudier par la force des choses) en parcourant le contenu du répertoire **`sample`** du SDK.

Plusieurs éléments doivent être assimilés pour le développement Android et, par conséquent, pour la compréhension de cet article. La notion d'activité (*activity*) est au cœur du système Android. Une activité est un écran de l'interface utilisateur. Une application est généralement composée de plusieurs activités et une seule à la fois est présentée à l'écran, et donc, en fonctionnement. Une activité est créée en implémentant une sous-classe de **`Activity`**. Il existe plusieurs sous-classes utilisables en fonction du type d'écran que vous souhaitez présenter à l'utilisateur.

Un service est un élément qui ne possède pas d'interface graphique et qui est implémenté dans le but de procéder à un traitement pouvant durer un certain temps. Il faut bien comprendre, en effet, qu'une activité dont le traitement durera longtemps ne peut être acceptable car elle bloquera l'interface et laissera l'utilisateur patienter sans indication particulière. Dès lors qu'il s'agit de calculer, opérer sur des données ou simplement communiquer avec un périphérique,

il faudra se frotter à la programmation multi-threadée et aux principes qui vont avec. Android n'est pas tendre avec celui qui découvre le développement sur la plateforme car, comme vous allez le voir, le simple fait d'afficher un message d'attente pendant que la connexion est établie avec un périphérique Bluetooth nécessite le lancement d'un thread qui devra alors communiquer avec l'activité en cours d'exécution pour signaler son état d'avancement. Nous avons là du classique *multithreading* avec des *IPC-like* à la sauce Java/Android.

Une autre notion importante d'Android se résume en l'utilisation d'*intents*. Il s'agit de messages asynchrones traversant le système et permettant toutes sortes d'actions. Un exemple simple consiste au lancement d'une application. Il s'agit d'un ordre qu'elle reçoit et auquel elle doit répondre. Le lancement d'une activité depuis une autre est provoqué par l'émission d'un tel ordre ou message, un intent. Il en existe pour toutes sortes d'actions.

Ensuite, nous avons le fichier **Manifest** qui décrit une application et les activités qu'elle contient. Ce manifeste comprend divers éléments, comme les permissions que l'application nécessite, les niveaux d'API qu'elle demande ou encore les fonctionnalités qui lui sont nécessaires. Ces informations, tout comme la description des interfaces, sont au format XML. Qu'on aime ou pas, c'est comme ça et il faudra vous y plier si vous n'avez pas envie de créer des applications où l'intégralité des interfaces est décrite dans le code (et donc pénible à gérer).

Enfin, comme vous le savez peut-être, je suis un utilisateur de Vim pur et dur. Cependant, force est de constater que l'utilisation de l'environnement de développement (IDE) Eclipse s'avère presque indispensable tant il facilite le développement. Même s'il reste possible de développer des applications avec un éditeur de grande qualité comme Vim et les outils en ligne de commandes mis à disposition par le SDK, la tâche devient rapidement improductive avec des développements conséquents. Le simple fait de renommer une activité dans une application relève parfois du parcours du combattant tant les données peuvent être disséminées dans l'arborescence d'un projet. Le plugin ADT (*Android Development Tools*) pour Eclipse permet d'intégrer très efficacement le SDK dans l'IDE au point de vous fournir, en une seule interface, tout ce qu'il vous faut pour développer, tester, débugger et gérer vos applications, et ce sur un périphérique réel ou dans l'émulateur Android. Enfin, et c'est l'une des raisons pour laquelle j'ai supporté de lâcher mon cher Vim, Eclipse est un IDE open source multiplateforme (écrit en Java).

Tout cela, si vous êtes un développeur UNIX, changera radicalement vos habitudes de programmation et, comme bien d'autres codeurs, le passage du C à Java, du procédural à l'objet, de Vim/Emacs à Eclipse... vous donnera le sentiment de revenir à vos débuts en programmation. Vous savez, ce sentiment de découvrir, les yeux écarquillés, de nouvelles contrées sauvages et exotiques. Mais c'est également ce qui vous fera douter car si certains programmeurs (qui n'en sont pas vraiment) peuvent se satisfaire du « juste ça marche », il n'en va pas de même pour le développeur, le vrai (avec des poils luisants la nuit), qui se demandera invariablement si la méthode, la classe ou le principe utilisé est celui qui convient. Là, malheureusement, il n'y a pas de réponses toutes faites, si ce n'est de lire encore et encore des codes d'autres programmeurs, plus expérimentés, et de perdre, au passage, quelques heures ou nuits de sommeil. Sachez cependant que d'autres sont dans cette situation et j'ai appris dernièrement que beaucoup de développeurs hésitent à diffuser les sources de leurs applications Android car peu convaincus qu'elles soient vraiment « propres ». Pourtant, comme le dit ESR : RERO, *Release early, release often*. C'est le meilleur moyen d'améliorer votre code, quitte à recevoir des leçons (peut-être désagréables) des autres.

1 Et si on commençait ?

Notre application exemple, qui n'a pas pour but de servir effectivement à quoi que ce soit, mais constitue simplement une exploration de l'API Android et en particulier celle de la gestion Bluetooth, se composera de deux activités. Nous partons ici du principe que nous souhaitons tester le bon fonctionnement de la liaison Bluetooth avec un montage présenté par ailleurs. Le montage en question se compose d'un AVR ATtiny2313 équipé d'un module Bluetooth **Wide.hk** lui servant d'interface via l'USART. L'AVR attend sur son port série un message lui demandant d'afficher quelque chose sur un dispositif composé de 8 afficheurs 7 segments. L'ordre **G** affichera **12345678** et l'ordre **Z** remettra l'affichage à zéro (**00000000**).

Côté architecture de notre applicatif, la première activité se chargera de présenter tous les périphériques Bluetooth associés sous la forme d'une liste. Un « clic » sur l'un de ces éléments listés lancera une autre activité affichant quelques informations de base avant de créer et exécuter un thread chargé de la communication avec le périphérique désigné. Un message d'attente assorti d'une barre de progression fera patienter l'utilisateur. Une fois le traitement terminé, l'activité restera affichée. L'utilisateur pourra alors revenir en arrière et choisir un autre périphérique pour procéder au même test.

Plusieurs éléments entrent en ligne de compte :

- Nous ne nous occupons pas de la recherche et de l'association de périphériques, mais déléguons cette tâche au système. Nous proposerons à l'utilisateur d'accéder directement à l'activité de gestion Bluetooth qui est accessible via les paramètres système.
- Nous listons tous les périphériques associés quelle que soit leur classe (CoD). Filtrer cette liste constituera une évolution possible de l'application.

- Nous tâcherons de faire les choses proprement en implémentant tout le mécanisme de connexion et de communication (RFCOMM) avec le périphérique de manière indépendante de la seconde activité afin de ne pas bloquer l'interface. C'est également le seul moyen pour afficher un message d'attente puisque dans le cas contraire, la gestion de la communication empêcherait le rafraîchissement de l'interface et donc l'affichage effectif du message.
- Il ne s'agit que d'un squelette destiné à vous mettre l'eau à la bouche et susciter d'autres idées d'applications potentielles pour vous motiver dans la découverte d'Android et de Java.
- Le code est loin d'être optimisé ou élégant, son principal objectif est didactique et démonstratif (ceci dit, je ne doute pas un instant qu'il doit y avoir des choses bien pires sur le Market Android, mais personne ne le sais, leurs sources n'étant pas disponibles).

Entrons sans plus attendre dans le vif du sujet. Une activité est composée de l'implémentation d'une sous-classe de la classe **Activity**. Ici, notre première activité sera une **ListActivity** :

```
public class BlueTalkActivity extends ListActivity {
}
```

Une activité peut se trouver dans quatre états différents : démarrée, en fonctionnement, tuée ou arrêtée (détruite). Comme vous le savez peut-être, on ne quitte pas une application Android. Celle-ci est soit en cours d'utilisation, soit inutilisée. Dans ce dernier cas, elle est en pause et, si d'autres applications nécessitent des ressources, le système détruira le processus. On ne quitte pas une application Android, on l'oublie.

Plusieurs méthodes *callback* sont appelées lorsqu'une activité change d'état. Deux sont très importantes : **onCreate()** est appelée lors de la création de l'activité, au moment où elle arrive en avant-plan et **onResume()** lorsqu'elle est présentée à l'utilisateur. Attention, **onResume()** arrive juste après **onCreate()** lors du premier lancement de l'application, mais également lorsque l'activité est passée en arrière-plan puis revient au devant de la scène. Lorsque cela arrive, nous ne savons pas ce que l'utilisateur a pu faire. Un exemple simple qui nous concerne est le fait de savoir si le Bluetooth est actif ou non. Lorsque l'utilisateur se sert d'une autre application et revient à notre liste, il a très bien pu désactiver le Bluetooth de son smartphone. Nous devrons alors tester la fonctionnalité dans le callback **onResume()** et non dans **onCreate()**.

D'autres callbacks existent, comme **onPause()** ou **onStart()**. Consultez la page http://developer.android.com/reference/android/app/Activity.html de la documentation développeur. Celle-ci comporte un diagramme très utile du cycle de vie d'une activité.

Nous implémenterons donc les méthodes suivantes :

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
}

@Override
public void onResume() {
  super.onResume();
}
```

Le **super** est présent afin de faire appel au constructeur de la classe parente. En d'autres termes, moins académiques, comme notre **ListActivity** est une sous-classe de **Activity**, nous cherchons les morceaux qui nous manquent dans la classe parente pour construire nos méthodes en utilisant ce qui existe déjà (là les « vrais » développeurs Java ont normalement envie de me massacrer pour oser faire un tel raccourci). Le **@Override** est présent afin de signaler que nous allons réécrire des méthodes existantes.

Une **ListActivity** est une activité intégrant une **ListView**, c'est à dire une liste verticale d'éléments *scrollables*. Pour composer une telle liste de données et la rattacher à la **ListView**, il faut utiliser un *adapter* et plus exactement un **ListAdapter**.

Dans un premier temps, nous devons, dans les ressources de notre projet, définir cette liste sous la forme d'un fichier XML **res/layout/bluetalk_myrow.xml** :

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="48px"
        android:layout_height="48px"
        android:layout_marginLeft="10px"
        android:layout_marginRight="10px"
        android:layout_marginTop="10px"
        android:src="@drawable/icon" >
    </ImageView>

    <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@+id/label"
        android:paddingLeft="5dp"
        android:paddingRight="5dp"
        android:paddingTop="5dp"
        android:paddingBottom="5dp"
        android:textSize="16sp" >
        <!--
        android:textSize="20px"
        -->
    </TextView>
</LinearLayout>
```

Il s'agit là d'une **ListView** personnalisée où chaque ligne (*item*) est composée d'une icône sur la gauche et d'un **TextView** sur la droite, occupant le reste de l'espace disponible. Nous y placerons le nom du périphérique Bluetooth, son adresse et la valeur de son CoD (classe de périphérique/service). L'icône sera fixe et intégrée au projet sous la forme d'un fichier **icon.png**, la même icône que celle de l'application. Dans Eclipse, le code XML nous donnera un aperçu de notre liste :

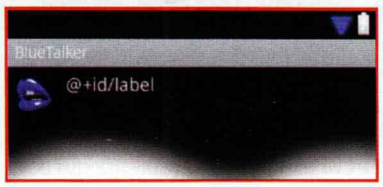


Figure 1

Dans notre callback **onCreate()**, notre première tâche sera donc d'utiliser cette ressource et de la connecter avec des données. Pour ce faire, nous allons instancier un **ArrayAdapter** en spécifiant le contexte, la ressource utilisée et le **TextView** qu'il faudra peupler avec nos données :

```
mArrayAdapter = new ArrayAdapter<String> (this,
  R.layout.bluetalk_myrow, R.id.label);
```

R est une classe représentant les ressources dynamiquement créées à partir de nos fichiers XML. Cette classe est maintenue à jour par le SDK au moment de la construction de l'application et les fichiers **.java** générés sont placés dans le sous-répertoire **gen/** de votre projet (pas touche !). La ressource ici est donc notre description de liste et **label** identifie le **TextView** à manipuler. La seconde étape consiste à désigner le **ArrayAdapter** instancié comme le pont entre la **ListView** et nos données :

```
setListAdapter(mArrayAdapter);
```

À ce stade, nous avons une **ListView** utilisable, ou presque. Nous prenons les devants en récupérant notre **ListView** sous la forme de l'objet **lv** et en profitons pour activer le filtrage du texte saisi :

```
ListView lv = getListView();
lv.setTextFilterEnabled(true);
```

Nous avons besoin de **lv** pour, plus tard, nous permettre de lancer une autre activité en fonction de l'élément choisi par l'utilisateur dans la liste. Le filtrage, quant à lui, permet, si l'utilisateur tape du texte, de n'afficher que les éléments dont le label commence par les caractères saisis. C'est purement cosmétique.

2 Le Bluetooth pour Android

Il est temps de faire connaissance avec le Bluetooth sous Android. La première chose à faire à la « création » de notre activité principale est de nous assurer que le périphérique utilisé dispose effectivement de fonctionnalités Bluetooth. Nous tentons donc de récupérer un objet **BluetoothAdapter** :

```
mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

if (mBluetoothAdapter == null) {
Toast.makeText(this,
"Device don't have Bluetooth",
Toast.LENGTH_SHORT).show();
finish();
return;
}
```

Notez qu'il s'agit ici de simplement prendre la main sur l'adaptateur Bluetooth par défaut. Ceci n'implique pas que le Bluetooth soit effectivement activé, mais simplement que c'est une fonctionnalité existante dans le smartphone ou la tablette. Si l'objet obtenu n'existe pas (**null**), inutile de poursuivre. Nous affichons un message via un *toast* (bulle d'information) et stoppons les réjouissances. Notez que l'émulateur Android ne dispose pas de fonctionnalité d'émulation du Bluetooth. Il n'est donc pas possible de « jouer » avec cette application sans un vrai smartphone Android.

Le reste se passera (jusqu'à nouvel ordre) dans le callback **onResume()**. Appelé à la fois juste après **onCreate()**, mais également au moment où l'utilisateur revient à notre activité après l'avoir mise de côté, ce callback consistera en une grosse boucle destinée à récupérer la liste des périphériques déjà associés pour composer notre liste **mArrayAdapter**, élément par élément.

Mais avant cela, nous devons procéder à quelques tests. Normalement, la capacité Bluetooth de l'appareil n'est pas censée pouvoir disparaître « en live ». Cependant, notre premier test consistera à nous assurer que **mBluetoothAdapter** existe, puis si c'est le cas, qu'il est bel et bien activé :

```
if (mBluetoothAdapter != null) {
if (!mBluetoothAdapter.isEnabled()) {
```

S'il ne l'est pas, plutôt que de simplement le signaler à l'utilisateur, nous allons l'activer (avec son consentement). Nous allons donc composer un intent afin de demander au système de réagir.

```
Intent enableBtIntent =
   new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
```

Notre intent, **enableBtIntent**, est créé puis utilisé via **startActivityForResult**. C'est là une facilité mise à notre disposition permettant de lancer une activité dans le but d'obtenir une réponse. **REQUEST_ENABLE_BT** est ici arbitrairement défini à **3**. **startActivityForResult** prend en argument l'intent à utiliser ainsi qu'un code (un *request code*). Ce code, si l'activité à lancer est présente, nous est retourné ainsi qu'un code de résultat via l'appel à un callback, **onActivityResult()**, qui doit être implémenté dans notre activité :

```
public void onActivityResult(int requestCode,
  int resultCode, Intent data) {
switch (requestCode) {
case REQUEST_ENABLE_BT:
if (resultCode == Activity.RESULT_OK) {
        Toast.makeText(this,
           "BT enabled", Toast.LENGTH_SHORT).show();
}
if (resultCode == Activity.RESULT_CANCELED) {
Toast.makeText(this, "Please Activate Bluetooth",
                Toast.LENGTH_SHORT).show();
        finish();
        return;
}
break;
}
}
```

Nous utilisons un **switch/case** et non un **if** car, plus tard, d'autres lancements d'activités seront faits de la sorte et nous devrons les différencier via le code utilisé au lancement. Pour notre **REQUEST_ENABLE_BT**, nous testons simplement le code de retour qui désignera l'activation du Bluetooth ou l'abandon par l'utilisateur. Dans ce cas précis, nous ne pouvons plus rien faire d'autre que de rendre la main au système. Si l'utilisateur active le Bluetooth, notre activité reviendra au premier plan et le callback **onResume()** sera utilisé. Dans ce cas, le Bluetooth sera activé et nous passerons à la suite.

Figure 2

Figure 3

Cette suite consiste à récupérer un *set* (ensemble) d'objets représentant les périphériques Bluetooth associés via la méthode **getBondedDevices()** de notre objet **mBluetoothAdapter**. Nous composons ensuite deux listes si le set obtenu contient au minimum un objet :

```
} else {
Set<BluetoothDevice> pairedDevices =
   mBluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
mArrayAdapter.clear();
mDeviceArray.clear();
    for (BluetoothDevice device : pairedDevices) {
    mBluetoothClass = device.getBluetoothClass();
        mArrayAdapter.add(device.getName() +
          "\n  " + device.getAddress() +
          "\n  Device Class: 0x" + String.format("%06X",
          mBluetoothClass.getDeviceClass()));
        mDeviceArray.add(device.getAddress());
    }
}}}}
```

Vous reconnaîtrez là notre **mArrayAdapter** servant de lien avec notre **ListView**. Nous avons en parallèle un array **mDeviceArray** destiné à contenir l'adresse matérielle des périphériques Bluetooth obtenus. Pourquoi deux listes ? Je vous parlais en début d'article du sentiment de ne pas forcément faire les choses au mieux, voilà un exemple typique. **mArrayAdapter**, après vidage via la méthode **clear()**, permet de peupler notre **ListView** avec des éléments qui seront des chaînes de caractères utilisées comme labels. Visuellement, nous voulons le nom, l'adresse et la CoD. Cependant, nous aurons également besoin de lancer une nouvelle activité en passant un argument qui se trouve être l'adresse matérielle seule du périphérique à contacter. Je n'ai trouvé que deux solutions. La première consiste à utiliser le label comme argument sans avoir à utiliser une technique spécifique (*bunble*) pour le passage de paramètre. Ceci impliquait de retrouver l'adresse dans la chaîne ou de faire une concession sur le contenu du label... Pas très élégant. L'autre solution, finalement choisie, était de passer l'adresse en guise d'argument via un bundle et donc de tirer cette information de quelque part : une liste parallèle des périphériques associés. C'est **mDeviceArray**. Cette liste est également vidée avant de boucler dans le set de périphériques et les éléments sont respectivement ajoutés avec les méthodes **add** de **mArrayAdapter** et **mDeviceArray**.

Notre callback **onResume()** s'arrête là. Notre liste est composée. Dans l'absolu, notre application est déjà utilisable mais se limite au listage des périphériques associés (voir la figure 4 ci-contre).

Il nous faut cependant spécifier dans le manifeste de l'application deux permissions importantes :

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

Notre application doit en effet pouvoir accéder aux fonctionnalités Bluetooth, mais également disposer de la possibilité de provoquer l'activation du Bluetooth. Ajoutez simplement cela, soit sous la forme XML dans le **AndroidManifest.xml**, soit via l'interface graphique du plugin ADT pour Eclipse, onglet ***Permissions***, bouton ***Add***, élément ***Uses Permission***.

Figure 4

3 Choisir un périphérique associé

Pour l'heure, notre application ne fait pas grand chose si ce n'est afficher une liste de périphériques. Pour continuer le développement, nous devons provoquer une action dépendante du choix de l'utilisateur dans la liste. Il nous faut donc réagir à l'équivalent d'un clic sur l'un des éléments. Pour ce faire, nous devons définir un *listener*, c'est-à-dire un callback qui sera appelé au clic :

```
ListView lv = getListView();
lv.setTextFilterEnabled(true);

lv.setOnItemClickListener(mClickedHandler);
```

Nous devons implémenter une classe **OnItemClickListener** disposant d'une méthode **onItemClick**. Nous pourrions le faire directement dans la portée des arguments de **setOnItemClickListener** et ainsi nous éviter de nommer un objet, mais je trouve cette syntaxe plus claire. Imbriquer une implémentation dans une liste d'arguments me semble trop fouillis, ce qui n'est sans doute pas sans rapport avec mon bagage « C » (ou simplement parce que j'ai l'habitude de ranger mes affaires) :

```
private OnItemClickListener mClickedHandler = new OnItemClickListener() {
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

Bundle monBundle = new Bundle();
monBundle.putString("lacleflaclef", mDeviceArray.get(position));

Intent i = new Intent(BlueTalkActivity.this, BlueTalkDevice.class);
i.putExtras(monBundle);

startActivity(i);
}
};
```

Notre méthode **onItemClick** reçoit en argument, entre autres choses, **position** qui correspond à l'index de notre texte dans la liste d'éléments. Ceci correspond naturellement à l'adresse que nous avons stockée pour ce périphérique dans **mDeviceArray**, la liste parallèle.

Pour passer une valeur à une activité que nous lançons avec un intent, nous utilisons un *bundle* qui est une sorte de conteneur avec une relation clé/valeur. À **monBundle** se voit donc ajouté une chaîne correspondant à l'adresse du périphérique sous la clé (pardon) **lacleflaclef**. L'intent **i** (encore pardon) est instancié en passant au constructeur le contexte et la classe correspondant à notre prochaine activité chargée de présenter des données en rapport avec le périphérique choisi. Nous attachons à notre intent le bundle **monBundle** avec la méthode **putExtras()**. Enfin, nous lançons notre seconde activité, **BlueTalkDevice**.

4 Parler à un périphérique aka tu veux pas mon UUID ?

La première partie de notre application est terminée. Lorsque l'utilisateur tapotera sur un élément de la liste, nous lancerons une nouvelle activité destiné à traiter le périphérique désigné par une adresse Bluetooth. Cette nouvelle activité sera composée d'un simple **TextView** :

```
public class BlueTalkDevice extends Activity {
}
```

Comme précédemment, nous allons traiter les deux callbacks **onCreate()** et **onResume()**. Le premier se chargera de l'interface et de l'initialisation des objets. Nous commençons donc par créer un **TextView** avant de récupérer la fameuse adresse :

```
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

tv = new TextView(this);

Bundle iciBundle  = this.getIntent().getExtras();

if (iciBundle != null && iciBundle.containsKey("lacleflaclef")) {
mBlueToothAddress = iciBundle.getString("lacleflaclef");
tv.setText(mBlueToothAddress + " from intent\n");
}
```

Nous récupérons notre bundle sous la forme d'un objet **iciBundle**, et après nous être assurés qu'il était valide et contenait effectivement une chaîne pour la clé donnée, nous plaçons l'adresse dans **mBlueToothAddress** puis en profitons pour l'afficher. Précisons de suite que l'objectif n'est pas le design d'une belle

activité, mais surtout de parler « Bluetooth » (donc non, c'est pas beau). Nous définissons la vue comme étant notre **TextView** puis récupérons notre adaptateur Bluetooth local afin de nous en servir plus tard. Normalement, il n'est pas censé avoir disparu, mais testons tout de même (encore un tic du C ?) :

```
setContentView(tv);

mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
Toast.makeText(this, "No Bluetooth", Toast.LENGTH_SHORT).show();
finish();
return;
}}
```

À ce stade, nous avons tout ce qu'il nous faut pour continuer et le reste de l'aventure est du ressort de **onResume**. En effet, rien ne nous dit qu'une fois cette activité à l'écran, l'utilisateur ne va pas partir et faire autre chose. Lorsqu'il reviendra à cette activité, nous considérons qu'aucune opération n'a été effectuée, et donc, recommencerons nos manipulations : dialoguer avec le périphérique.

Ce dialogue risque de prendre du temps s'il fonctionne et il peut également échouer puisque le périphérique n'est pas forcément à portée. Nous devons donc faire patienter l'utilisateur. Plusieurs solutions sont envisageables. Ici, le choix se portera sur une barre de progression s'affichant par-dessus notre interface, un **ProgressDialog**. Nous allons instancier l'objet **dialog** et le paramétrer :

```
public void onResume() {
super.onResume();

dialog = new ProgressDialog(this);
dialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
dialog.setMessage("Talking to bluetooh device...\n");
dialog.setProgress(0);
dialog.setMax(5);
dialog.show();
```

Le voici réglé sur un maximum de **5** pas avec une valeur actuelle de **0**. **dialog** est immédiatement affiché à l'écran. Encore une fois, nous optons pour la sécurité en testant si le Bluetooth est activé. J'insiste, mais rappelez-vous que nous sommes dans **onResume()** et il est possible que l'utilisateur ait, entre autres choses, désactivé le Bluetooth. Nous ne réactivons pas cependant la fonctionnalité ici, mais abandonnons de manière à ce que notre précédente activité prenne le relais. Elle-même procède au test et si l'utilisateur n'active pas le Bluetooth, l'activité sera stoppée.

```
if (mBluetoothAdapter != null) {
if (!mBluetoothAdapter.isEnabled()) {
finish();
return;
} else {
```

Comme le préconise la documentation d'Android, il est impératif de vérifier la syntaxe d'une adresse d'un périphérique Bluetooth avant d'en tenter l'utilisation. Ceci se fait très simplement avec :

```
if(BluetoothAdapter.checkBluetoothAddress(mBlueToothAddress)) {
BluetoothDevice mBluetoothDevice =
mBluetoothAdapter.getRemoteDevice(mBlueToothAddress);
```

Ce n'est que si cette adresse est valide que nous poursuivons. Nous commençons par instancier un objet représentant notre périphérique cible. Attention, il ne s'agit que d'un point d'accès, aucune communication n'est tentée à ce stade. Continuons ensuite avec quelque chose d'un peu plus ardu : le lancement d'un thread. C'est là quelque chose de relativement sans pitié pour qui fait ses premiers pas en Java, mais il n'est pas acceptable de bloquer l'interface utilisateur, et donc notre barre de progression pendant que les échanges sont faits avec le périphérique.

Il nous faut instancier un thread puis le démarrer :

```
try {
mConnectThread = new ConnectThread(mBluetoothDevice, handler);
} catch (Exception e) {
Log.d(TAG, "ConnectThread constructor Exception");
}
mConnectThread.start();
```

Avant de nous lancer dans de folles explications, terminons l'implémentation de **onResume()** en réagissant à une possible mauvaise syntaxe de l'adresse testée plus haut :

```
} else {
finish();
return;
}}}}
```

La création de notre thread est entourée de **try** et **catch**. Il s'agit là du mécanisme de gestion/traitement des erreurs de Java. Le C utilise généralement les valeurs retournées par des fonctions pour déterminer si une erreur est survenue (**-1**) ou que l'appel a été couronné de succès (**0** ou **>0**). Java utilise un mécanisme consistant à essayer (bloc **try**) et intercepter des exceptions en cas d'erreur. Le bloc **catch** est un *handler* d'exceptions permettant de réagir à l'erreur survenue. On peut aussi décider de ne pas traiter le problème et remonter l'exception à un bloc de plus haut niveau avec **throw**. L'exception elle-même est un objet représenté ici par **e**.

Remarquez que le constructeur de notre thread prend en argument deux paramètres. Le premier est un handler offrant un moyen de communication entre notre thread et notre activité. Ceci permet de mettre à jour la valeur de la

barre de progression que nous affichons à l'utilisateur. Nous y reviendrons plus tard. L'autre paramètre est l'objet représentant notre périphérique cible.

Penchons-nous maintenant sur notre classe **ConnectThread** étendant la classe **thread** :

```
private class ConnectThread extends Thread {
private final BluetoothSocket mmSocket;
OutputStream tmpOut;
Handler mHandler;

public ConnectThread(BluetoothDevice device, Handler h) throws
Exception {
mHandler = h;
Method m = device.getClass().getMethod("createRfcommSocket",
new Class[] { int.class });
mBluetoothAdapter.cancelDiscovery();
mmSocket = (BluetoothSocket)m.invoke(device, Integer.valueOf(1));
}
```

La méthode portant le même nom que la classe est notre constructeur. Nous y récupérons tout d'abord notre handler permettant la communication avec notre activité. Ensuite, cela devient plus ésotérique. **m** est une méthode récupérée « de force » depuis l'API Android. Nous ne sommes pas censés l'utiliser, mais nous le pouvons via le mécanisme Java de réflexion. La documentation Android propose d'utiliser la méthode **createRfcommSocketToServiceRecord(UUID uuid)** d'un objet **BluetoothDevice** afin d'obtenir une socket pour nous connecter au périphérique. Si cette opération ne pose généralement pas de problème, la connexion effective avec la méthode **connect()** sur l'objet socket obtenu donne des résultats qu'on peut librement qualifier d'aléatoires.

L'UUID est censé décrire le service qu'on souhaite utiliser sur le périphérique. Dans le cas du service SSP ou SP (*serial port*), c'est le bien connu UUID **00001101-0000-1000-8000-00805F9B34FB** qui est utilisé. Cependant, comme nous l'avons vu par ailleurs dans ce dossier, la réponse ne semble pas toujours au rendez-vous. L'API Bluetooth d'Android propose cette méthode afin de s'assurer que nous accédons à un périphérique disposant bien du service attendu. Or il arrive très fréquemment, question de timing sans doute, que le périphérique ciblé ne semble pas proposer le service et la connexion échoue (tantôt cela fonctionne). C'est un problème connu au point que les applications utilisant ce type de connexions intègrent naturellement cette astuce consistant à utiliser la réflexion et donc l'utilisation d'une méthode non publique. On retrouve par exemple cela dans l'application Android pour la MetaWatch de Ti, mais également dans toutes les applications permettant la communication avec des Arduino via Bluetooth.

Sachez cependant que, comme cette solution ne repose pas sur l'API publique, il n'est absolument pas garanti qu'elle soit viable dans le temps. Elle nous permet temporairement de faire fonctionner notre application en obtenant une socket sans spécifier d'UUID (et donc sans vérification de service). C'est ce que nous faisons pour obtenir **mmSocket**.

La connexion et le dialogue à proprement parler ne se déroulent pas à la création de notre thread, mais lors de son exécution. C'est la méthode **run()** qui doit être implémentée :

```
public void run() {

try {
mmSocket.connect();
} catch (IOException e) {
Log.d(TAG, "++++ ERROR !!! mmSocket.connect IOException");
dialog.dismiss();
return;
}
```

Encore une fois, nous faisons usage du traitement classique des exceptions de Java en tentant la connexion via la méthode **connect()** de notre socket. En cas d'erreur, nous signalons le problème, désactivons l'affichage de la barre de progression et abandonnons. Dans le cas contraire, nous utilisons notre handler pour passer des valeurs à notre activité :

```
mHandler.sendMessage(mHandler.obtainMessage(0, 1, 0, "connected"));
```

Un handler est un objet que nous demandons au système via **obtainMessage()**, nous ne le créons pas. Nous pouvons l'initialiser avec différentes informations. Ici, nous utilisons **arg1** pour la valeur de la progression de la barre et **obj** pour passer une **String**. Nous verrons le traitement du handler par la suite. La phase 1 sur 5 est accomplie, passons à la suite en obtenant un *stream* nous permettant d'envoyer des données au périphérique :

```
try {
tmpOut = mmSocket.getOutputStream();
} catch (IOException e) {
Log.d(TAG, "++++ ERROR !!! mmSocket.getOutputStream IOException");
cancel();
}
mHandler.sendMessage(mHandler.obtainMessage(0, 2, 0, "have output
stream"));
```

Encore une fois, la gestion des exceptions nous permet de traiter un éventuel problème. Cette fois, nous n'abandonnons pas simplement mais appelons la méthode **cancel()** de notre classe **ConnectThread**. Nous avons obtenu une socket, en cas de problème, il convient de la refermer. Si tout se passe bien, la phase 2 est achevée et nous en informons l'utilisateur comme précédemment. Il ne nous reste plus qu'à envoyer des données :

```
try {
tmpOut.write('G');
tmpOut.write('\r');
} catch (IOException e) {
Log.d(TAG, "++++ ERROR !!! write G
IOException");
cancel();
}
mHandler.sendMessage(mHandler.
obtainMessage(0, 3, 0, "G writed"));

try {
Thread.sleep(5000);
} catch (InterruptedException e1) {
e1.printStackTrace(); }

try {
tmpOut.write('Z');
tmpOut.write('\r');
} catch (IOException e) {
Log.d(TAG, "++++ ERROR !!! write Z
IOException");
cancel();
}
mHandler.sendMessage(mHandler.
obtainMessage(0, 4, 0, "Z writed"));
```

La méthode **write()** permet d'envoyer un octet à la fois. Comme notre protocole de communication est simpliste, nous n'avons pas besoin de créer quelque chose de plus complexe que deux occurrences de **write()** par commande. Bien entendu, en cas de communication plus évoluée, nous devrons être en mesure de traiter une chaîne complète et la découper en autant d'octets que nécessaire. Notez que les deux envois de commandes **G** et **Z** sont séparés d'une temporisation de 5 secondes. Nous mettons notre thread en sommeil durant ce temps tout en gérant d'éventuelles exceptions quant à l'opération. Enfin, nous fermons la socket et terminons le traitement :

```
try {
mmSocket.close();
} catch (IOException e) {
Log.d(TAG, "++++ mmSocket.close IOException");
cancel();
}
mHandler.sendMessage(mHandler.obtainMessage(0,
5, 0, "Socket closed"));
}
```

Il nous reste à implémenter la méthode permettant de nous sortir proprement d'une situation problématique :

```
public void cancel() {
dialog.dismiss();
try {
mmSocket.close();
} catch (IOException e) {
Log.d(TAG, "++++ mmSocket.close IOException");
}
}}
```

Rien de bien exceptionnel ici, il s'agit simplement de fermer la socket après avoir retiré la barre de progression. Nous venons de nous connecter à un périphérique Bluetooth et de lui envoyer des données. *Enjoy* !

5 Faire attendre l'utilisateur

Tout au long de la communication, nous avons utilisé des handlers afin de faire part de la progression à l'utilisateur. Notre handler est un objet devant implémenter une méthode **handleMessage** :

```
final Handler handler = new Handler() {
    public void handleMessage(Message msg) {
    int progress = msg.arg1;
    tv.append(Integer.toString(progress) + " : " + (String)msg.obj + "\n");
    dialog.setProgress(progress);
    dialog.setMessage("Talking to bluetooh device...\n(" + (String)msg.obj + ")");
    if(progress >= 5) {
    dialog.dismiss();
    }
    }
};
```

Nous récupérons les éléments qui nous intéressent de chaque message reçu, **msg.arg1** et **msg.obj** afin de faire progresser la barre affichée via la méthode **setProgress()** de notre **dialog** et d'afficher les **String** avec **append()** sur notre objet **TextView**. Un simple test nous permet de supprimer la barre lorsque la valeur atteint 5.

Figure 5

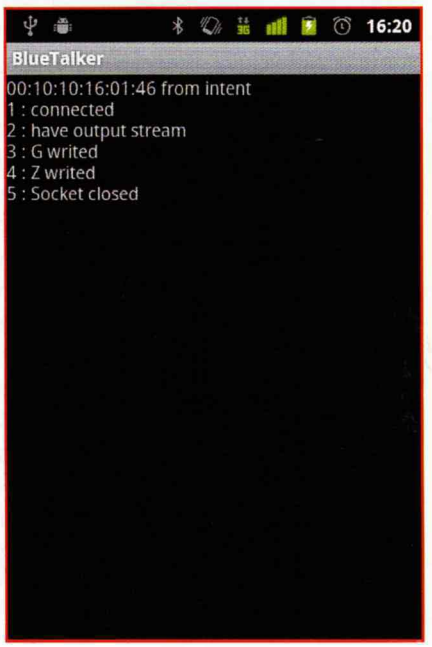


Figure 6

L'application nous permet d'obtenir le résultat escompté en début d'article. Lorsque nous choisissons dans la liste le bon périphérique, le lancement de la seconde activité provoque l'affichage et la progression de la barre : voir Figure 5.

En fin de processus, le thread créé disparaît, ainsi que la barre de progression, pour laisser voir à l'utilisateur les différents messages qui sont parvenus à l'activité (l'affichage se fait simultanément mais il est caché par la barre) : voir Figure 6.

En cas de tentative de connexion découlant sur une erreur, la barre disparaît simplement sans plus de fioriture.

Conclusion, perspectives et devoirs de vacances

Cette application est loin d'être parfaite, et surtout, elle ne sert pas à grand chose en dehors de la démonstration et de l'expérimentation en tant que telles. En d'autres termes, elle n'est tout simplement pas destinée à un utilisateur « normal » mais nous a permis d'explorer l'API Bluetooth d'Android, ses spécificités et ses problèmes. Il reste beaucoup à ajouter pour en faire une application pouvant servir à piloter un montage dans la vraie vie.

N'a été traité ici que le strict minimum. L'application test inclut également un menu permettant d'appeler l'activité intégrée au système et permettant de gérer les périphérique Bluetooth. Ceci, je pense, ne changera pas, mais c'est un choix tout personnel : j'estime que l'utilisateur peut se débrouiller pour associer les périphériques de son choix pour ensuite les choisir dans la `ListView`. Bien entendu, personne ne vous interdit d'intégrer directement une activité permettant l'ajout de périphériques.

Figure 7

Figure 8

L'application affiche également TOUS les périphériques associés, ce qui n'est pas forcément souhaitable. Il serait ainsi judicieux de filtrer cette liste en fonction de la classe (CoD) des périphériques ou leur nom. Il peut également être envisagé de tester la connectivité de chaque périphérique afin d'ajouter un élément visuel dans la liste pour ceux qui sont à portée (changement de couleur de l'icône, par exemple). Ceci implique le lancement d'un nouveau thread depuis la première activité puisque c'est une tâche de longue haleine.

Enfin, et c'est sans doute plus grave, l'utilisateur n'est absolument pas prévenu d'un éventuel problème. Ceci devrait être simple à ajouter avec les éléments décrits ici. L'activité propre à la communication avec le périphérique est sans doute l'endroit où il est possible d'inventer toutes sortes de choses. Si nous restons dans le cadre du module d'affichage distant, permettre à l'utilisateur de saisir une valeur à afficher peut être un bon début.

Je n'ai pas traité non plus la lecture depuis le périphérique distant et si l'implémentation est relativement simple, l'étendue des possibilités est énorme. Vous avez totalement le choix des armes puisque vous contrôlez à la fois la partie cliente, mais également la réalisation du périphérique Bluetooth. Lire et écrire ouvrent d'intéressantes perspectives : capteur de mouvement, de température, contrôle d'éclairage d'une pièce, notification diverses, robotique, écriture de Widgets à placer sur le « bureau » Android, ...

Le code source de l'application est à votre disposition sur le site du magazine sous la forme d'une archive GNU/Tar créée via la fonction d'exportation d'Eclipse. Soyez indulgent si vous y trouvez des horreurs dues au fait que l'application ait le même âge que mes connaissances de Java, soit environ deux semaines. N'hésitez pas cependant à me faire part de vos remarques, critiques et conseils. ■

CRITIQUE LIVRE : LINUX EMBARQUÉ DE GILLES BLANC

Les ouvrages en français sur l'embarqué sont rares. Lorsqu'il s'agit de traiter de GNU/Linux, cette rareté devient un chiffre : 2. Jusqu'à présent, ce chiffre était divisé par deux mais c'était sans compter la ténacité de Gilles Blanc, auteur de « Linux embarqué : comprendre, développer, réussir » paru il y a quelques jours chez Pearson.

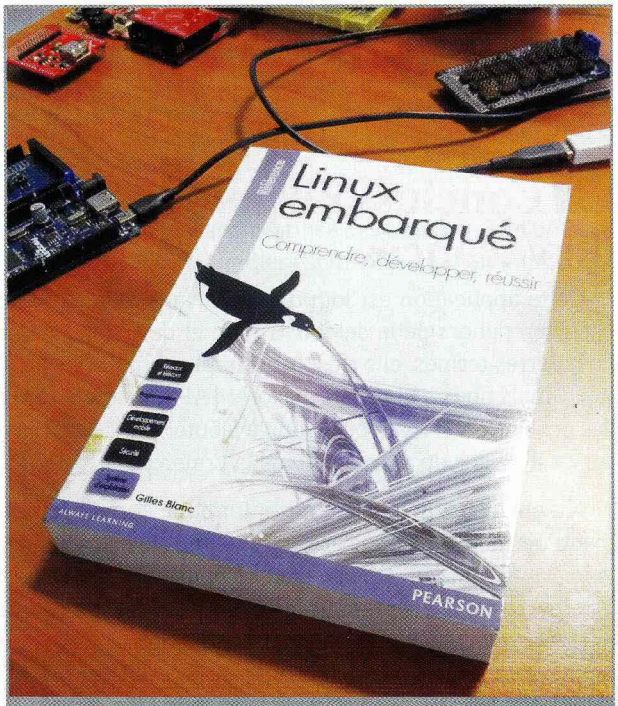
Jusqu'à présent, le seul livre en français sur le sujet qui nous est cher était l'excellent ouvrage de Pierre Ficheux chez Eyrolles. Pearson, cependant, réussit le tour de force de publier un ouvrage non pas concurrent mais complémentaire sur un sujet similaire. L'approche de Gilles est beaucoup plus orientée gestion de projet et stratégie de développement mais ne délaisse pas pour autant la technique. L'accent est ainsi mis très nettement sur la philosophie, les pratiques et les implications liées à l'utilisation de licences open source.

Ainsi, vous trouverez dans les quelque 500 pages du livre, bien sûr, des informations techniques sur l'organisation des sources de Linux, les différentes distributions pour l'embarqué ou encore la mise en place d'un environnement de développement croisé, mais également une étude du marché de « Linux embarqué », une description des travaux préparatoires à planifier avant le développement ainsi que des indications sur la manière de rédiger et composer votre cahier des charges.

Un autre point notable du livre concerne le processus d'intégration et la valeur ajoutée de votre projet. Tout l'aspect collaboratif, comme la relation avec les développeurs est également traitée. On regrettera que ce point ne soit pas davantage exploré en détaillant, par exemple, une démarche d'ouverture sous forme d'étapes et de ressources à mettre en place pour une meilleure communication vers l'extérieur. Mais ceci sera peut-être un ajout possible à une future seconde édition.

Table des matières :

- I L'embarqué et le libre : des éléments de choix stratégiques
- 1. Le libre en général et dans l'informatique industrielle
- 2. Mais pourquoi le libre ?
- 3. Comprendre GNU/Linux et les distributions
- 4. Osons parler du droit
- II Rédiger les spécifications
- 1. Le marché de Linux embarqué
- 2. Comment choisir une solution adaptée
- 3. Modélisation
- III Réalisation
- 1. Organisation générale du système
- 2. Le kernel et les modules
- 3. Mettre en place l'environnement de développement croisé
- 4. Mise en pratique
- IV Intégration et valeur ajoutée
- 1. Interactions avec le développeur
- 2. Démarrage du système
- 3. Finalisation du système
- 4. Intégrer sa valeur ajoutée. ■



À propos de l'ouvrage :
- Auteur : Gilles Blanc
- Broché: 550 pages
- Éditeur : Pearson Education
- Collection : Référence
- Langue : Français
- ISBN-10: 2744025208
- ISBN-13: 978-2744025204

Complétez votre collection des anciens numéros de...

Les 4 façons de commander !

Par courrier
En nous renvoyant ce bon de commande.

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

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

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

Bon de commande Open Silicium		
Réf.	**Désignation**	**Prix / N°s**
OS N°3	ARM + FPGA = ROBOTIQUE	9,00 €
OS N°4	Les PIC aussi ont leurs Arduino !	9,00 €
LMHS N°51	GNU/Linux Magazine Hors-Série ÉLÉCTRONIQUE, EMBARQUÉ ET HACKS	6,50 €
LMHS N°47	GNU/Linux Magazine Hors-Série VOYAGE AU CENTRE DE L'EMBARQUÉ	6,50 €
LMHS N°43	GNU/Linux Magazine Hors-Série ÉLÉCTRONIQUE, EMBARQUÉ ET HACKS	3,25 €
LMHS N°39	GNU/Linux Magazine Hors-Série CARTES À PUCE	3,25 €
LMHS N°38	GNU/Linux Magazine Hors-Série ÉLÉCTRONIQUE, EMBARQUÉ ET DOMOTIQUE	3,25 €
LMHS N°27	GNU/Linux Magazine Hors-Série ÉLECTRONIQUE ET LINUX	3,25 €
LMHS N°25	GNU/Linux Magazine Hors-Série LINUX EMBARQUÉ 2	3,25 €
LMHS N°24	GNU/Linux Magazine Hors-Série LINUX EMBARQUÉ	3,25 €
LMHS N°23	GNU/Linux Magazine Hors-Série ÉLECTRONIQUE ET LINUX	3,25 €

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

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

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

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

Je choisis de régler par :

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

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

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

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

❑ Je souhaite recevoir des infos des Édtions Diamond

❑ Je souhaite recevoir des infos des partenaires des Éditions Diamond

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

Open Silicium

DÉVELOPPEZ VOS JEUX POUR NINTENDO DS SOUS LINUX

par Yann Morère

Je suis assez nostalgique des bornes arcades des années 80 : Xgalaga, 1942, Arkanoid [1]... autant de « Shoot Them Up » qui ont eu raison de mes petites économies d'adolescent. Ensuite est venu le temps du Commodore 64, de l'Amiga 500. Des machines mythiques (je n'ai pas dit ordinosaures) qui restaient inaccessibles pour moi à l'époque. Ensuite, les consoles portables ont fait leur apparition, je me souviens, ému, de ma première Sega GameGear. Maintenant, nous sommes à l'air de la 3D, mais, mis à part l'aspect graphique et réseau (ou encore les jeux d'énigmes [2]), le cœur des jeux n'a guère évolué depuis Space Invader ou SWIV [3] : il faut toujours détruire l'autre :-). Serait-ce pour cela que je m'en serais détourné et utilisé tout ce temps libéré à la découverte et l'utilisation de Linux... mais là je m'égare...

1 Introduction

Comme ne le dit pas le chapeau de l'article (terme rédactionnel consacré), nous allons ici nous intéresser au développement d'application pour la petite console portable de Nintendo : DS, DS Lite et Dsi. Cette petite console double écran (DS pour « Dual Screen ») a été créée par Nintendo et a vu le jour en Europe en 2005 (cf. Figure 1). Elle est équipée de plusieurs fonctions très intéressantes **[4]** :

- deux écrans rétro-éclairés dont un écran tactile ;
- un microphone ;
- deux ports cartouche (un pour les jeux DS, un autre pour les cartouches de jeu Game Boy Advance et les accessoires) ;
- deux haut-parleurs compatibles « surround » ;

(Source Wikimedia http://commons.wikimedia.org/wiki/File:Nintendo-DS-Lite-w-stylus.png?uselang=fr)

Figure 1

- Wi-Fi intégré, d'une portée de 10 à 30 mètres en LAN, permettant de connecter seize consoles entre elles, et de se connecter au « Nintendo WiFi Connection » pour jouer en ligne.

Son architecture est basée sur 2 processeurs de type Arm, pour la DSLite, un processeur principal ARM946E-S (66 MHz) et un sous processeur ARM7TDMI (33 MHz), et pour la Dsi, un processeur principal ARM9 (133 MHz) et un sous-processeur ARM7 (33 MHz), une mémoire de 4Mo pour la DS Lite et 32Mo pour la Dsi. En ce qui concerne l'affichage graphique, le moteur 2D gère 4 couches (arrière-plan, « background ») et 128 sprites sur chaque écran, en 3D, 120 000 polygones/seconde et 30 millions de pixels/seconde.

Cette console étant très diffusée (fin 2009, la Nintendo DS compte 8,4 millions d'exemplaires vendus), les hackers ont développés des accessoires qui permettent l'utilisation de logiciels développés à la maison (les fameux « Homebrews » : brassé à la maison) et l'utilisation de copie de sauvegarde de jeux officiels : on les appelle des « Linkers ».

Les Linkers les plus répandus utilisent le « Slot 1 », c'est-à-dire le slot des cartouches de jeux. Il possède la plupart du temps un emplacement microSD pour le stockage des roms. La figure 2 présente un linker très connu : le R4i

Figure 2

2 Ai-je le droit d'acheter et d'utiliser un « linker »

Clairement NON depuis le 26 septembre 2011 **[5]**. La vente de Linker est devenue illicite en France comme dans d'autres pays européens comme l'Allemagne, la Belgique, la Grande-Bretagne, l'Italie et les Pays-Bas. La justice a donné raison à Nintendo en appel, alors qu'en première instance (décembre 2009), les sociétés qui distribuaient ces dispositifs avaient eu gain de cause. Les avocats avaient alors défendu le droit à la copie de sauvegarde et la possibilité d'utiliser des applications tierces, les « homebrews ».

Ces derniers permettent de pallier certains manques de la console : un bon exemple est le lecteur de fichiers audio et vidéo Moonshell **[6]**, sans parler du portage de Linux sur cette console qui fera l'objet d'un paragraphe particulier.

Voici une liste non exhaustive de sites partageant des logiciels « homebrews » pour la console Nintendo :

- http://en.wikipedia.org/wiki/Nintendo_DS_homebrew ;
- http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html ;
- http://www.docteeboh.net/blog/?page_id=346 ;
- http://www.ndshb.com/cgi-bin/cfiles/cfiles.cgi?2,0,0,0,1 ;
- http://wiki.akkit.org/Downloadable_DS_Demos ;
- http://nintendo-ds.dcemu.co.uk/NintendoDS-HomebrewGames.php ;
- http://www.dev-scene.com/NDS/Homebrew_Catalog.

Cependant, la partie n'est pas encore terminée car les sociétés revendeuses peuvent parfaitement se pourvoir en cassation.

Pour éviter toute mauvaise posture vis-à-vis de la loi, dans la suite de cet article, nous utiliserons un émulateur qui nous permettra de tester les développements réalisés.

3 Les outils pour développer

Avant de commencer, il est nécessaire de connaître les bases du langage C pour pouvoir développer à l'aide des outils que je vais présenter. Si vous ne maîtrisez pas ce langage, il est préférable de vous former un minimum. Le Web regorge de très bon cours d'initiation au langage C.

La principale bibliothèque permettant de développer pour DS est la libnds **[7]**. Anciennement NDSLIB, c'est une bibliothèque créée par Michael Noland et Jason Rogerset maintenue par Dave Murphy. C'est une alternative open source au SDK commercial de Nintendo pour les consoles portables. Cependant, la prise en main de cette bibliothèque n'est pas aisée et la documentation est quasi inexistante. Les plus motivés pourront jeter un œil du côté de **[8]**.

Nous utiliserons la PALib, *Programmer's Arsenal Library* **[9]**. C'est une bibliothèque qui utilise libnds (bibliothèque de bas niveau), mais qui possède des fonctions de plus haut niveau et est donc plus simple à utiliser. Elle vous offre toutes les fonctions nécessaires pour développer un jeu ou un programme sur NDS sans vous soucier des problèmes matériels. Cependant, elle ne vous permettra pas d'exploiter pleinement la puissance de votre console. Elle peut être téléchargée à l'adresse **[9]**.

Remarque

PALib ne semble plus maintenue depuis 2010, cependant elle reste tout à fait utilisable. On pourra aussi utiliser la version du site **[10]** qui semble avoir été mise à jour en 2009.

PALib n'est qu'une bibliothèque et il faut pouvoir réaliser des exécutables pour les processeurs ARM en faisant de la cross-compilation. Pour cela, nous utiliserons Devkitpro qui propose un ensemble d'outils, dont un compilateur pour ARM et la bibliothèque libnds permettant de créer nos propres binaires pour Nintendo DS.

Mes principales sources documentaires pour l'écriture de cet article ont été les pages **[11]** et **[12]** ainsi que **[13]** pour la documentation de la PALib. Ces pages vous permettront d'aller plus loin dans l'utilisation de cette bibliothèque.

4 Installation des outils

4.1 Outils de compilation

PALib, pour pouvoir fonctionner, nécessite les 6 outils suivants : devKitARM, LibNDS, maxmodnds, dsWifi, libfilesystem, libFAT. Ils sont téléchargeables à l'adresse **[14]**. PALib, elle, est téléchargeable à l'adresse **[9]**.

L'installation est réalisée en désarchivant l'ensemble des composants dans le répertoire devkitPro que l'on doit créer au préalable. L'arborescence finale ressemble à ceci :

```
$ tree -d -L 2 devkitPro/
devkitPro/
├── devkitARM
│   ├── arm-eabi
│   ├── bin
│   ├── include
│   ├── lib
│   ├── libexec
│   └── share
├── libnds
│   ├── include
│   └── lib
└── PAlib
    ├── cpptemplate
    ├── docs
    ├── emulators
    ├── examples
    ├── include
    ├── lib
    ├── source
    ├── template
    ├── tools
    └── vctemplate
```

Le script disponible à l'adresse **[14]** (prendre la nouvelle version du script mise à jour par votre serviteur) réalise cela pour vous en une seule fois. Il a été mis à jour avec les dernières versions des bibliothèques. Faites un copier/coller des commandes dans votre éditeur préféré et enregistrez le fichier sous « install_stuff.sh ». Puis donnez-lui les droits en exécution et finalement lancez-le.

```
$ chmod 755 install_stuff.sh
$ ./install_stuff.sh
```

La partie importante du script concerne la mise à jour des variables d'environnement :

```
export DEVKITPRO=~/devkitPro
export DEVKITARM=$DEVKITPRO/devkitARM
export PAPATH=$DEVKITPRO/PAlib/lib
```

Ces exportations de variables permettent au compilateur de trouver les différentes bibliothèques nécessaires à la génération du binaire NDS. Les commandes restantes réalisent le téléchargement et le désarchivage des différents composants.

Les fichiers binaires générés possèdent l'extension « .nds » (ex : Text.nds) : il s'agit du format binaire utilisé par les jeux officiels et la plupart des émulateurs. Il permet d'embarquer un petit logo, une brève description du jeu. Il existe d'autres formats et je vous renvoie à la page **[15]** pour plus de détails.

Remarque

Si, au moment de votre installation, les versions de devkitARM et des bibliothèques ont évolué, vous aurez des messages d'erreur sur des fichiers non trouvés. Il faut alors éditer manuellement le fichier de script et remplacer par les noms des fichiers des nouvelles versions que vous trouverez à l'adresse **[16]**.

Nous allons ensuite tester le bon fonctionnement des outils installés. Pour cela, rendez vous, par exemple, dans le répertoire « ~/devkitPro/PAlib/examples/Text/Normal/Text » et lancez la commande « make ».

```
$ cd ~/devkitPro/PAlib/examples/Text/Normal/Text
$ make
Build process start for project "Text"...
main.c
Linking...
Built: Text.nds
```

Si vous n'avez aucun message d'erreur, un fichier « Text.nds » doit être créé dans le répertoire. Ce fichier représente l'image d'un programme binaire pour votre console.

Si vous obtenez le message d'erreur suivant :

```
Makefile:46: /PAlib/lib/PA_Makefile: Aucun fichier ou dossier de ce type
make: *** Pas de règle pour fabriquer la cible " /PAlib/lib/PA_Makefile ".
Arrêt.
```

il est fort probable que le fichier **PA_Makefile** soit inaccessible. La plupart du temps, cela vient du fait que les variables d'environnement ne sont pas initialisées. Si vous avez modifié le fichier « .bashrc » comme demandé par le script d'installation, sourcez-le et les erreurs devraient disparaître :

```
$ source ~/.bashrc
```

Vous pouvez aussi relancer manuellement les commandes d'export de variables **DEVKITPRO**, **DEVKITARM** et **PAPATH**.

Tous les outils sont prêts pour commencer le développement d'application pour votre Nintendo. Avant de poursuivre, nous allons installer un émulateur, pour pouvoir tester notre programme fraîchement compilé.

4.2 DeSmuMe : Emulateur NDS

Nous allons utiliser DeSmuMe **[17]** en le recompilant à partir des sources. Il est aussi possible d'utiliser « Ideas Emulator » qui est livré avec la PALib dans le répertoire « ~/devkitPro/PAlib/emulators/ideas-linux ».

On commence par télécharger l'archive de DeSmuMe et on le recompile :

```
$ wget http://downloads.sourceforge.net/project/desmume/desmume/0.9.7/
desmume-0.9.7.tar.gz
$ tar xzf desmume-0.9.7.tar.gz
$ cd desmume-0.9.7
$ ./configure # on ignore le message d'erreur sur  po/Makefile.in.in was
not created by intltoolize
$ make
```

Cela crée 2 versions de l'émulateur : une version CLI (« command line interface ») dans le répertoire « src/desmume-cli », et une version gtk dans le répertoire « src/gtk ». Nous utiliserons dans la suite la version GTK.

Afin d'éviter d'installer des programmes non « packagés », nous allons utiliser desmume directement depuis son répertoire de compilation. On pourra créer un simple lien depuis le répertoire « /usr/local/bin », par exemple :

```
$ sudo ln -s /repertoire/absolu/de/compilation/src/gtk/desmume /usr/
local/bin/desmume
```

Ensuite, pour tester notre programme, il suffit de lancer « desmume » avec comme paramètre une image de binaire NDS valide (ici « Text.nds ») :

```
$ desmume Text.nds
addonsChangePak
Failed to set format: Argument invalide
Microphone init failed.
DeSmuME 0.9.7 svnr0
SoftRast Initialized
File doesn't appear to have a secure area.
ROM crc: 429A7323
ROM serial: Homebrew
ROM internal name: .
ROM game code: ####
DEBUG_reset: 00414980
DeSmuME .dsv save file not found. Trying to load an old raw .sav file.
Missing save file /home/yann/.config/desmume/Text.dsv
```

Et vous obtenez la figure 3. En réalisant un cliquer/glisser sur la partie inférieure de l'écran, vous devriez voir apparaître l'évolution de la position à côté de « Stylus Position ».

Avant de poursuivre par la création d'un petit projet personnel, nous allons voir que votre console peut faire tourner votre système d'exploitation préféré.

5 Linux sur NDS

Le projet DSLinux **[18]** a porté le système d'exploitation Linux sur Nintendo DS et DSLite (les modèles plus récents ne sont pas supportés). Cependant, le projet n'est plus actif, mais reste fonctionnel et les documentations sont très complètes pour qui voudrait reprendre le développement.

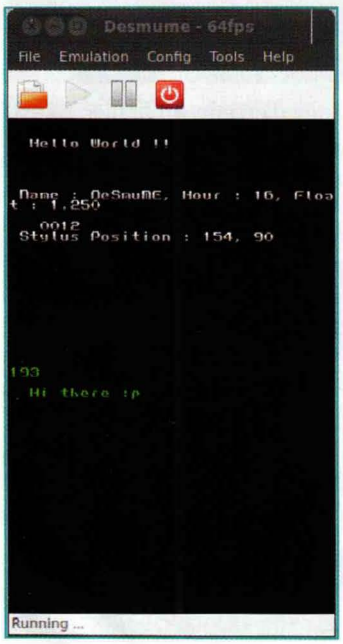


Figure 3

Figure 4

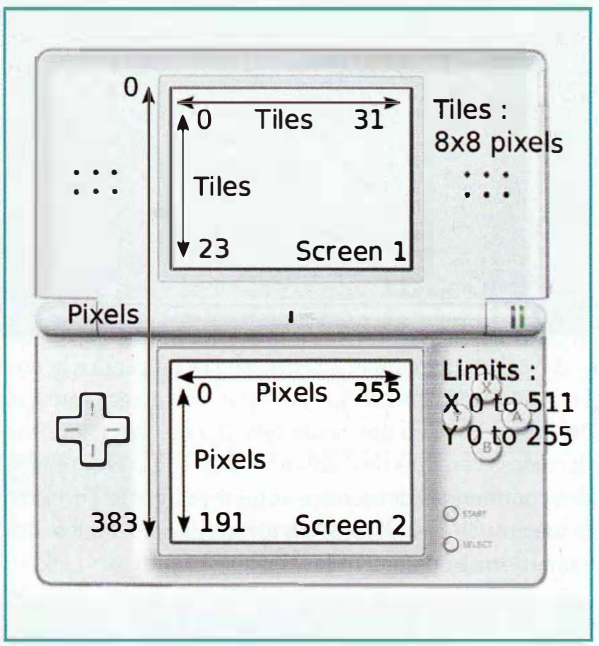


Figure 5

DSLinux est basé sur le projet µClinux **[19]** lui-même dérivé d'un noyau Linux 2.0 (au départ) pour les microcontrôleurs sans unité de gestion mémoire (MMU pour Memory Management Unit) utilisée pour la protection de plages mémoire. En effet, notre console ne possède pas d'unité de gestion mémoire matérielle, ni logicielle **[20]**.

Ici, nous allons juste tester le binaire Linux dans notre émulateur. Bien sûr, il ne sera pas possible de profiter pleinement du système de fichiers et du Wi-Fi dans cette configuration.

Une utilisation plus approfondie de ce projet a été réalisée par J.M. Friedt et G. Goavec-Merou dans *GLMF* HS 43 : « Interfaces matérielles et OS libres pour Nintendo DS : DSLinux et RTEMS » **[21]**.

On télécharge une image linux de binaire NDS à l'adresse **[22]**. Ensuite, on teste notre distribution préférée en émulation avec la commande :

```
$ desmume dslinux.nds
```

On se connecte sur le système à l'aide du login « root » et du mot de passe « uClinux » (attention à la casse des caractères et au pilotage du clavier à la souris). On obtient alors la figure 4.

6 projet « Shoot Them Up » StarShooter

Avant de démarrer notre petit projet de « Shoot Them Up » StarShooter, nous allons écrire notre premier programme pour NDS : un fameux « Hello World ».

6.1 Screen, Sprite et Background

Avant de commencer à programmer, il faut connaître et comprendre quelques concepts relatifs à la programmation de jeux et à la console.

Tout d'abord, voyons les limites physiques de notre console en image (figure 5). Ce schéma vous indique les dimensions d'affichage en « pixels » et « tiles » (tuiles).

Les deux écrans peuvent être gérés indépendamment ou considérés comme les 2 parties d'un écran unique plus grand. Ils peuvent être utilisés aussi en faisant pivoter la DS de 90 degrés, un peu comme un livre ouvert.

Voyons ensuite quelques particularités sur le calcul des images.

Si le rendu des images du jeu est réalisé pendant que les écrans sont redessinés, l'utilisateur verra des images partiellement mises à jour donnant lieu à des artefacts visuels du plus mauvais effet. La solution est de modifier le contenu de l'écran entre deux rafraîchissements d'écran, ou alors d'utiliser le « Page Flipping » (changement de page) **[23]**.

La première approche peut être implémentée en attendant l'apparition de l'interruption VBI (Vertical Blank Interrupt). Cette interruption (VBI) est activée lorsqu'un écran complet a été redessiné. Les 2 écrans sont rafraîchis à 60 Hz, la période entre deux VBI est de 16,7 ms. On peut donc effectuer le rendu pendant ce temps qui est relativement court.

Si votre calcul est plus long, il faudra utiliser le « page flipping ». Cette méthode consiste à réaliser le rendu dans un buffer écran pendant que le matériel affiche un autre « buffer » précédemment calculé. À chaque VBI, les buffers sont échangés de telle sorte que chaque tâche peut continuer. Cette technique permet de disposer de plus de temps pour le rendu que dans le cas de l'utilisation de l'interruption VBI.

Dans notre cas, nous utiliserons l'interruption VBI.

La console possède 2 systèmes de rendu 2D, un par écran. Nous n'utiliserons pas le rendu 3D dans notre petit projet. Les banques de mémoire VRAM (*Video Random Access Memory*) sont dédiées aux deux types de composants graphiques que l'on utilise :

- la mémoire d'arrière-plan (*background*) qui va contenir les images de décors, interfaces, cartes. La plupart du temps, ils sont composés d'un grand nombre de tuiles (*tiled map*).
- la mémoire sprite qui contient les images de sprite (élément actif du jeu composé d'une image).

Un sprite est un petit objet graphique qui peut être transformé indépendamment des autres objets. Il s'agit en général de personnages, d'objets en mouvement (véhicules, vaisseaux, etc.), mais ils peuvent être n'importe quelle entité avec laquelle on pourra avoir une interaction **[24]**.

Les sprites sont gérés nativement par la console. Différents effets peuvent être appliqués à un sprite. Il peut être animé, redimensionné, pivoté, retourné (horizontalement ou verticalement), rendu translucide (canal alpha) ou pixelisé.

Notre console permet d'afficher jusqu'à 128 sprites par écran. Cependant, il faut faire attention à leurs dimensions car le matériel ne gère que les dimensions suivantes (largeur par hauteur) :

8x8	8x16	8x32	
16x8	16x16	16x32	
32x8	32x16	32x32	32x64
		64x32	64x64

Si votre sprite ne fait pas exactement une de ces tailles, il faudra opter pour la taille supérieure la plus proche. De même, si votre sprite est plus grand que 64x64, vous devrez utiliser plusieurs sprites.

On peut considérer les « Backgrounds » (arrière-plans) comme les opposés des sprites. Ils sont utilisés pour représenter les décors et l'environnement souvent statique.

Il en existe 2 types sur cette console : les arrière-plans construits à l'aide de tuiles (les « tiled map ») et les arrière-plans dessinables (« drawable »).

La mémoire vidéo de la console nous permet d'utiliser 4 arrière-plans de type « Tiled Map » en même temps ou un arrière-plan de type « Drawable » accompagné de 2 autres de type « Tiled Map ».

Le « Tiled Map » est le type d'arrière-plan le plus courant. Il est rapide à afficher et à modifier. Le « Drawable » a l'avantage de pouvoir être utilisé en mode 16 bits. On peut ainsi profiter de l'utilisation directe d'images GIF/JPG pour l'affichage de l'arrière-plan. Il est aussi possible de dessiner directement dessus.

Vous avez maintenant quelques bases concernant l'affichage graphique sur cette console. Nous allons passer à la suite, notre premier programme pour Nintendo DS.

6.2 Hello World

Un projet utilisant la PALib possède une structure propre. Par exemple, notre « Shoot Them Up » aura la structure arborescente suivante :

```
├── audio
├── build
├── gfx
│   └── bin
├── source
```

Le répertoire « audio » recevra les fichiers son, le répertoire « build » est le répertoire de travail du projet, on y retrouve tous les fichiers objets issus de la compilation et les fichiers intermédiaires. Le répertoire « gfx » contiendra les sprites et les fonds d'écran, le répertoire « source » contiendra le code source de notre projet.

On commence par créer l'arborescence nécessaire à la compilation d'un programme pour PALib en reprenant celui contenu dans les exemples de la bibliothèque :

```
$ mkdir -p progs_nds/BonjourMonde/source
$ mkdir -p progs_nds/BonjourMonde/build
$ cd progs_nds/BonjourMonde
$ cp ~/devkitPro/PAlib/examples/Text/Normal/HelloWorld/Makefile .
$ cp ~/devkitPro/PAlib/examples/Text/Normal/HelloWorld/source/main.c source/
$ geany source/main.c
```

Le code source est le suivant :

```
#include <PA9.h>

int main(){
  // Initialize PAlib
  PA_Init();

  // Load the default text font
  PA_LoadDefaultText(1, // Top screen
                     1); // Background #2

  // Write the text "Hello World"
  PA_OutputSimpleText(1, // Top screen
                      1, // X position 1*8 = 8 X coordinate in TILES (0-31) where to begin writing the text
                      1, // Y position 1*8 = 8 Y coordinate in TILES (0-19) where to begin writing the text
                      "Bonjour le Monde");
  // Infinite loop to keep the program running
  while(true){
    // Wait until the next frame.
    // The OS runs at 60 frames per second.     PA_WaitForVBL();
  }
}
```

Je vous propose quelques explications concernant ce code source. La ligne d'« include » permet d'utiliser notre bibliothèque PALib. La commande « PA_Init() » réalise les opérations d'initialisation de la bibliothèque. On charge ensuite la police de texte par défaut sur l'écran supérieur et dans l'arrière-plan numéro 2. Puis on affiche un texte sur l'écran supérieur à une position qui représente un multiple de 8 pixels (la largeur d'une tuile représentant un caractère). Ensuite, la boucle infinie permet d'attendre chaque rafraîchissement d'écran (**PA_WaitForVBL**). Pour plus d'informations sur les fonctions utilisées, je vous invite à regarder la page de manuel de l'adresse **[9]**.

Une fois compilé à l'aide de la commande « make », on obtient un fichier binaire nds que l'on peut tester à l'aide de desmume et on obtient la figure 6.

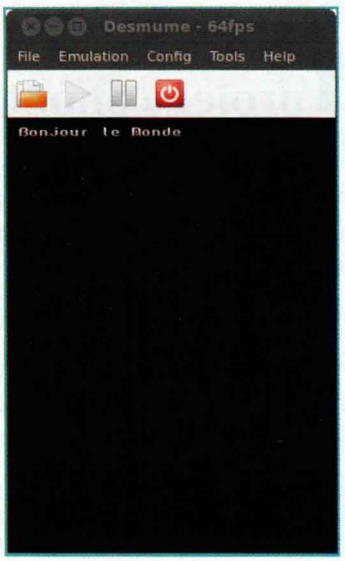


Figure 6

Hourra, vous venez de produire votre premier programme pour Nintendo DS. Nous allons passer à la suite en affichant notre premier sprite.

6.3 Création de notre premier sprite

Notre petit projet consiste à réaliser un début de jeu « Shoot Them Up » à la manière de Xgalaga, mais en ajoutant les déplacements verticaux à notre vaisseau.

Avant de préparer notre sprite, nous allons voir quelques notions importantes concernant les formats d'images à utiliser pour la création de sprite.

Les sprites disposent de 3 modes de couleurs : palette de 16 couleurs (trop limitée), palette de 256 couleurs (très utilisée) et les sprites 16bits sans palette, mais qui consomment trop de mémoire. Nous utiliserons ici les sprites à palette 256 couleurs.

Un sprite étant une image de forme rectangulaire, il est nécessaire de ne pas afficher la couleur d'arrière-plan de l'image. Pour cela, on choisit une couleur particulière comme couleur de transparence. La couleur magenta (rouge : 255, vert : 0, bleu : 255) est souvent utilisée pour cela.

Pour résumer, il nous faut utiliser une image en 256 couleurs, qui utilise le Magenta comme couleur de fond.

Nous avons vu que cette image doit être de petite taille et respecter certaines conditions de ratio (voir le paragraphe sur les sprites). Lors de mes essais, j'ai trouvé les sprites de dimensions 32x32 un peu petits, et les 64x64 limitent l'amplitude des déplacements du vaisseau dans l'espace de l'écran.

Pour pallier ce problème, on conserve des sprites de dimensions 64x64, mais le vaisseau, lui, aura une taille de 48x48 et sera centré dans cette tuile.

N'étant pas un artiste pour la réalisation des sprites, j'ai utilisé mon expérience LEGO (cf. LP63) et Blender (LP64). Le tout est résumé sur la figure 7.

J'ai utilisé une de mes créations Lego sous Blender, puis positionné convenablement une caméra pour avoir une vue de dessus. Ensuite, on règle les dimensions de l'image et de la caméra. On opère le rendu avec Blender, ensuite à l'aide de Gimp on ajoute un calque de fond de couleur magenta. On prendra garde à bien réduire le nombre de couleurs pour l'utilisation ultérieure. Dans l'optique d'animer mon vaisseau lors des déplacements, j'ai également

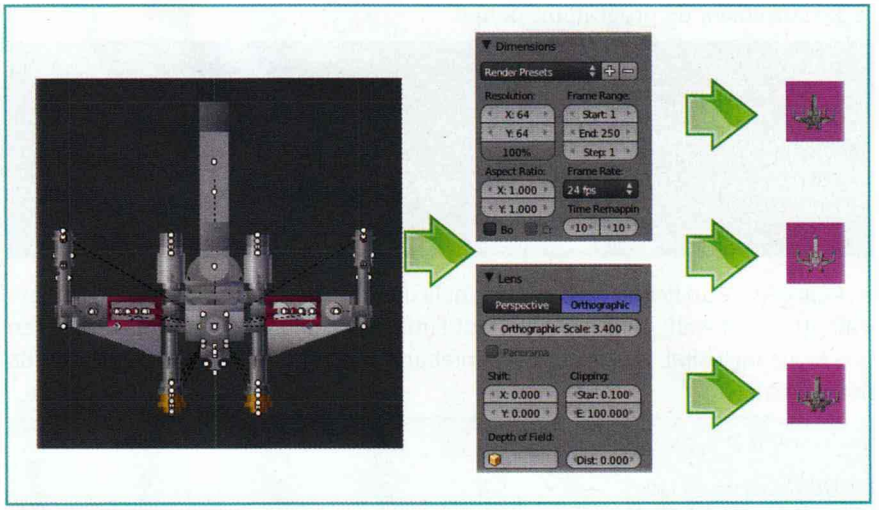


Figure 7

ajouté deux autres caméras et effectuer 2 autres rendus (modification de l'angle de vue) pour simuler la rotation du vaisseau sur son axe longitudinal lors des déplacements gauche-droite (cf. figure 6).

Notre premier sprite est prêt, cependant, il n'est pas directement utilisable avec la PALib. Il faut opérer une conversion pour que l'image soit utilisable par la console. On utilise pour cela le programme PAGfx disponible dans l'arborescence de la PALib « devkitPro/PAlib/tools/PAGfx/Mono ». Comme l'indique le répertoire, il vous faudra installer Mono pour pouvoir l'utiliser sous Linux. Il existe 2 versions : une en ligne de commandes et une avec un frontal graphique. Ces deux programmes sont utilisables en lançant :

```
$ mono PAGfxFrontend.exe
```

La version console du programme s'utilise avec un fichier **.ini** contenant la définition des directives de conversion.

En bon linuxien, il est aussi possible de générer la version console de PAGfx à partir des sources de l'adresse **[25]** après avoir installé les outils de compilation pour le langage C# (outils du projet Mono). Un fois désarchivé, on compile le programme par :

```
$ ./autogen.sh
$ ./configure && make
```

Le programme exécutable se trouve dans le répertoire « bin/Release/PAGfx.exe ». Il suffit ensuite de copier les fichiers « PAGfx.exe » et « PAGfx.ini » dans le répertoire « gfx » où l'on aura placé nos fichiers images.

Ensuite, on remplit le fichier « PAGfx.ini » comme suit :

```
#TranspColor Magenta
#Sprites :
xwing_sprite_64_v4.png 256colors xwing
```

« TranspColor Magenta » est la couleur de transparence que vous utilisez pour vos sprites. « xwing_sprite_64_v4.png » est l'image source qui sera transformée en sprite, « 256colors » est le mode de couleurs, et « xwing » est le nom de la palette qui contiendra les couleurs de votre sprite, elle sera renommée « xwing_Pal ».

Le lancement du programme donne :

```
$ ./PAGfx.exe
Converting PAGfx.ini
xwing_sprite_64_v4.png
Transparent Color : Magenta
1 sprites :
xwing_sprite_64_v4 : 256colors, 64x64, Pal : xwing_Pal, -> xwing_sprite_64_v4_Sprite
1 palettes :
xwing_Pal, 115 colors
```

Cela génère un fichier « .c » contenant la définition de notre sprite et les fichiers « all_gfx.c » et « all_gfx.h » permettant l'utilisation de tous les sprites utilisés en une seule inclusion. Nous allons maintenant afficher notre sprite sur l'écran de notre émulateur.

```
#include <PA9.h>

// PAGfxConverter Include
#include "../gfx/all_gfx.h"

int main(void){

  PA_Init(); //PAlib inits

  //Graphic Part
  // fix the space between screens to 0 -> as one big screen
  PA_SetScreenSpace(0);

  //Dual Version allow to use the two screen as one for Sprites
  //Palette Load
  // Loads x-wing sprite palette as 0
  PA_DualLoadSpritePal(
     0, // Palette number
     (void*)xwing_Pal); // Palette name
  //XWing[0].palette = 0;
    // Creates x-wing sprites as 0 : top level sprite
  PA_DualCreateSprite(
     0, // Sprite number
     (void*)xwing_sprite_64_v4_Sprite, // Sprite name
     OBJ_SIZE_64X64, // Sprite size
     1, // 256 color mode
     0, // Sprite palette number
     20, 20); // X and Y position on the screen

  while(1) // Infinite loops
    {

    PA_WaitForVBL();
  }
return 0;
}
```

Une fois compilé et lancé, on obtient la figure 8. Je ne détaillerai pas les instructions car le code C est commenté. Cependant, on remarquera la fonction « PA_SetScreenSpace(0); » qui permet de fixer l'espace entre les deux écrans qui n'est pas nul par défaut. On utilise les deux écrans comme un seul. Cela permet d'éviter une discontinuité lors du déplacement d'un sprite d'un écran à l'autre. L'utilisation des versions « Dual » des fonctions de chargement de palette et de création de sprite permet l'utilisation des 2 écrans.

6.4 Déplacement du sprite

Nous allons maintenant réaliser le déplacement de notre sprite. Pour cela, nous utilisons le « Pad » qui est plus pratique que le stylet pour ce genre de jeux. La PALib fournit un accès direct aux valeurs des boutons du pad par l'intermédiaire

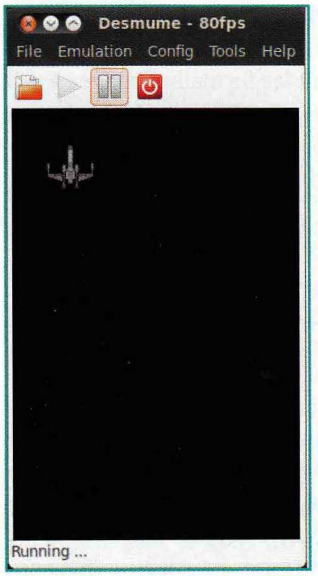


Figure 8

Figure 9

de structures. Ces valeurs sont mises à jour à chaque cycle (1/60ème de seconde, puisqu'on se cale sur le VBI).

- « Held » est, par défaut, à 0 et il passe à 1 lorsque le bouton est pressé.
- « Released » passe à 1 lorsque le bouton est relâché, reste à 1 durant seulement 1 « cycle » ou frame.
- « Newpress » passe à 1 juste lorsque vous pressez le bouton, et reste à 1 durant seulement un cycle.

Chaque structure de pression contient alors une liste de toutes les touches :

- « Left, Right, Up, Down » : pour les flèches ;
- « A, B, X, Y » : pour les touches A, B, X et Y ;
- « L, R » : pour les gâchettes gauche et droite ;
- « Start, Select » : pour les boutons start et select.

Pour vérifier que A est appuyé, on utilise la forme « Pad.Held.A ». « Pad.Released.A » correspond à vérifier le relâchement de A, et « Pad.Newpress.A » est utilisé pour détecter une nouvelle pression...

Nous allons ajouter le code suivant à l'intérieur de la boucle infinie :

```
// Update the position according to the keypad...
x += (Pad.Held.Right - Pad.Held.Left) * speed;
y += (Pad.Held.Down - Pad.Held.Up) * speed;

// bounds testing for x-wing
if (x < -8)
  x = -8;
if (x > (255 - 56)) //screen width - sprite width
  x = 255 - 56;
if (y < 0)
  y = 0;
if (y > 383 - 48) //screen height - sprite height
  y = 383 - 48;

// Set the X-Wing sprite's position
PA_DualSetSpriteXY(
      0, // sprite 0 : x-wing
      x, // x position
      y); // y position
```

La première partie du code permet, dans une version optimisée, de calculer les nouvelles coordonnées de notre vaisseau en fonction des boutons que l'on appuie.

De la même manière, nous pourrons animer les tirs de laser du vaisseau. On commence par charger les sprites :

```
int Sprites[256];
int lazer_x = 0, lazer_y = 0;  //laser position
s32 gunspeed = 4;  // speed of laser shot
// Loads laser sprite palette as 2
PA_DualLoadSpritePal(
   2, // Palette number
   (void*)laseryellow_Pal); // Palette name
PA_DualCreateSprite(2, // Sprite number
   (void*)yellow_Sprite, // Sprite name
   OBJ_SIZE_8X8, // Sprite size
   1, // 256 color mode
   2, // Sprite palette number
   0, 0); // X and Y position on the screen
Sprites[2]=1;
PA_DualCreateSprite(3, // Sprite number
   (void*)yellow_Sprite, // Sprite name
   OBJ_SIZE_8X8, // Sprite size
   1, // 256 color mode
   2, // Sprite palette number
   0, 0); // X and Y position on the screen
Sprites[3]=1;
```

Ensuite, le déplacement rectiligne des lasers peut être réalisé par le code suivant (il n'y a pas de gestion des collisions) :

```
if (lazer_y <= -8)  //gun shot are out of the screen
 {
 //Guns management
 if(Pad.Newpress.A) //if fire is pressed
   {
   // put the sprites just above the guns :-)
   lazer_x = x + 48;
   lazer_y = y + 24;
   PA_DualSetSpriteXY(
         2, // sprite
         lazer_x, // x position
         lazer_y); // y...
   PA_DualSetSpriteXY(
         3, // sprite
         lazer_x-40, // x position
         lazer_y); // y...
   }
 }
else
 {
 // Laser move management
 lazer_y -= gunspeed;
 PA_DualSetSpriteXY(
   2,
   lazer_x,
   lazer_y);
 PA_DualSetSpriteXY(
   3,
   lazer_x-40,
   lazer_y); //already add gun speed :-)
 }
```

6.5 Animation de notre sprite

Pour pouvoir animer un sprite, il faut tout d'abord définir une série de frames (images qui définissent l'animation du sprite). La première chose à respecter est de mettre toutes les frames du sprite dans le même fichier image en les plaçant les unes au-dessus des autres. Nous réaliserons une animation simple : les réacteurs du vaisseau s'activeront lorsque nous appuierons sur les touches de déplacement, et nous changerons l'orientation du vaisseau en fonction des touches Droite-Gauche. Notre fichier de sprite sera celui de la figure 9.

Remarque

Si vous modifiez vos sprites, n'oubliez pas d'ajouter les lignes correspondantes dans le fichier « PAGfx.ini » et de relancer la génération des sprites.

Ensuite, la gestion est réalisée par le code suivant :

```
if(((Pad.Newpress.Up )||(Pad.Newpress.Down)) && (!(Pad.Held.Right ||
Pad.Held.Left))) //button pressed = reactor activated
  PA_DualStartSpriteAnim(0, // sprite number
     1, // first frame is 0
     3, // last frame is 8, since we have 9 frames...
     4); // Speed, set to 8 frames per second
if(Pad.Newpress.Right)
  {
  PA_DualStartSpriteAnim(0, 5, 8, 4);
  PA_DualSetSpriteHflip(0, 1); // we only create le left turn, right
turn is made with Horizontal Flip
  }
if(Pad.Newpress.Left)
  {
  PA_DualStartSpriteAnim(0, 5, 8, 4);
  PA_DualSetSpriteHflip(0, 0); //right animation thanks to flipping
  }
if(!((Pad.Held.Left)||(Pad.Held.Up)||(Pad.Held.Down)||(Pad.Held.
Right)))  //no mouvement
  PA_DualSetSpriteAnimFrame(0,0); //1st frame, ship without motors
```

La gestion du sprite d'explosion est identique. La page **[26]** vous fournit tous les fichiers ainsi que le code source complet.

6.6 Arrière-plan animé

Nous allons maintenant mettre un arrière-plan que nous allons animer. De cette manière, on aura l'impression que le vaisseau avance continuellement dans l'espace.

Un fois l'image d'arrière-plan choisie, on ajoute dans le fichier « PAGfx.ini » la ligne suivante, puis on relance la génération des sprites :

```
#Backgrounds :
BG3.png TileBg
```

Lors de mes essais, pour pouvoir utiliser les fonctions « PA_DualLoadBackground » et « PA_DualBGScrollXY », j'ai dû ajouter la définition de la structure « BG3Dual ». En effet, les structures d'arrière-plan générées par le programme PAGfx ne sont pas compatibles avec les fonctions de la PALib.

```
#include <PA_BgStruct.h> //needed by scrollBG
// needed for scrollbg (the new structure PAGfx_struct does not work)
const PA_BgStruct BG3Dual = {
  PA_BgNormal,
  256, 256,
  BG3_Tiles,
  BG3_Map,
  {BG3_Pal},
  65536,
  {2048}
};
// Load the dual background...
PA_DualLoadBackground(3, //bg number
   &BG3Dual); //background name (struct name)
s32 scrollx = 0; // No X scroll by default...
s32 scrolly = 0; // No Y scroll by default...
```

Le code ci-dessous, placé dans la boucle infinie, réalise la gestion du déplacement de l'arrière-plan en fonction des actions sur le pad.

```
//Backgroud Scrolling
// We'll modify scrollx and scrolly according to the keys pressed
scrollx -= (Pad.Held.Left - Pad.Held.Right) * 1; // Move 1 pixels per press
scrolly -= (Pad.Held.Up ) * 1; // Move 1 pixels per press without going backward
// Scroll the background to scrollx, scrolly...
PA_DualBGScrollXY(3, // Background number
   scrollx, // X scroll
   scrolly-=1); // Y scroll Ship is always going forward
```

Il ne reste plus qu'à ajouter quelques ennemis sous la forme des chasseurs TIE :-) et la gestion des collisions. Une fois tout ceci mis en place, nous obtenons la figure 10.

Figure 10

6.7 Gestion des collisions

Dans cette partie, nous allons voir une méthode simple et efficace de détection de collisions. Elle est rapide à mettre en œuvre et se base sur les collisions d'objet circulaire.

Nous considérerons les gabarits de nos sprites comme des cercles. Avec cette approximation, il est assez simple de détecter une collision en comparant les distances entre les centres respectifs de chaque sprite comme montré sur la figure 11 :

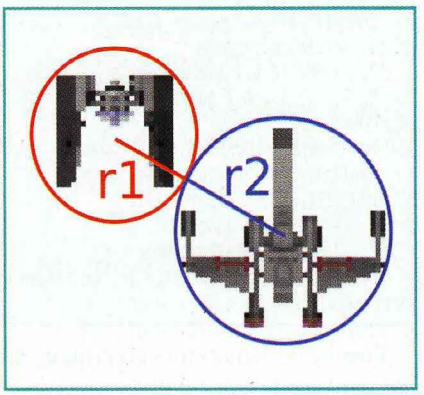


Figure 11

On remarque que lors d'une collision, la distance entre les centres est inférieure à r1+r2. C'est donc le test sur cette distance qui va nous permettre de démarrer l'animation de l'explosion ou non. Il ne faudra pas oublier aussi de supprimer les sprites de l'écran. Dans le code suivant, mon chasseur TIE est fixe, seul le tir laser se déplace :

```
if (Sprites[5]==1)
{
  if (Sprites[2]==1) //test if gun1 is active (not destroyed)
    if (PA_Distance(lazer_x, lazer_y, 140+32, 20) < (12+4)*(12+4))
      {
      mmEffect(SFX_BLKFOOT4);
      PA_DualStartSpriteAnimEx(7, 0, 7, 8,ANIM_ONESHOT);
      PA_DualDeleteSprite(5);
      Sprites[5]=0;
      PA_DualDeleteSprite(2);
      Sprites[2]=0;
      }
  if (Sprites[3]==1) //test if gun2 is active (not destroyed)
    if (PA_Distance(lazer_x-40, lazer_y, 140+32, 20) < (12+4)*(12+4))
      {
      mmEffect(SFX_BLKFOOT4);
      PA_DualStartSpriteAnimEx(7, 0, 7, 8,ANIM_ONESHOT);
      PA_DualDeleteSprite(5);
      Sprites[5]=0;
      PA_DualDeleteSprite(3);
      Sprites[3]=0;
    }
}
```

Pour calculer la distance entre les 2 sprites, on utilise la fonction « PA_Distance » qui renvoie le carré de la distance entre les deux objets passés en paramètres par l'intermédiaire de leurs coordonnées. Cela évite de calculer une racine carrée gourmande en temps processeur. J'ai volontairement décomposé le calcul de distance : 12 correspond au rayon r1 du chasseur TIE (le sprite fait 64x64, mais la partie utile est bien plus petite : environ 24x24), 4 correspond au rayon r2 du tir laser. Bien sûr, dans une version finale du programme, vous remplacez tout le calcul par sa valeur entière, afin de gagner quelques cycles.

Dans le cas d'une collision, on joue un effet d'explosion grâce à la fonction « mmEffect », puis on lance l'animation du sprite d'explosion que l'on a préalablement créée. Ensuite, on efface les sprites du chasseur abattu et du tir laser avec la fonction « PA_DualDeleteSprite ». Le tableau « Sprites » permet de conserver un état des sprites actifs/effacés afin de ne pas lancer des tests de collisions inutiles.

Vous pouvez tester une autre méthode de détection de collision basée sur des sprites rectangulaires à l'adresse **[13]**.

6.8 Musique et sons

Terminons ce petit projet en ajoutant un peu de son : une musique de fond et des bruitages (laser et explosion). Pour cela, nous utilisons la bibliothèque « maxmod » installée avec le reste des outils. Dans un premier temps, il faut modifier le fichier « Makefile » pour prendre en compte la bibliothèque « maxmod » pour l'utilisation du son. On active la ligne suivante :

```
ARM7_SELECTED := ARM7_MAXMOD_DSWIFI
```

Puis on place les fichiers son à utiliser dans le répertoire « audio » préalablement créé :

```
$ ls audio
battle_explosion.wav  blkfoot4.wav  explosin.wav  space.mod  wlkrsht2.wav
```

Ensuite, on ajoute les lignes suivantes dans le fichier « main.c » pour pouvoir utiliser la bibliothèque :

```
//Just be sure the Makefile for your project has ARM7_MAXMOD_DSWIFI
//as the selected ARM7 binary.
#include <maxmod9.h> // Maxmod include
// Include the soundbank created by the ARM7_MAXMOD_DSWIFI directive in Makefile
#include "soundbank_bin.h"
#include "soundbank.h"
```

Lors de la compilation, les fichiers audio sont convertis et regroupés dans une banque de sons : « soundbank_bin.bin » utilisable à l'aide de « soundbank_bin.h » et « soundbank.h ». Les fichiers de musique « mod » sont utilisables par l'intermédiaire des variables « MOD_FILENAME » et les fichiers de bruitage « .wav » sont utilisables grâce aux variables « SFX_FILENAME ». Le code suivant initialise la bibliothèque MaxMod, charge les différents sons, lance la musique de fond et fixe le niveau sonore :

```
//Sound part
// Initialize Maxmod
mmInitDefaultMem((mm_addr)soundbank_bin);
// Load and play the music
mmLoad(MOD_SPACE);
mmStart(MOD_SPACE, MM_PLAY_LOOP);
mmSetModuleVolume( 128 ); //volume : 0->1024 representing 0%->100% volume
// Load the sounds
// Please note that they are regular .wav
// files put into the audio folder
mmLoadEffect(SFX_WLKRSHT2);
mmLoadEffect(SFX_BLKFOOT4);
mmEffectVolume( SFX_WLKRSHT2, 255 ); //volume : 0..255 = silent..normal
```

Ensuite, il est possible de jouer un bruitage à un moment précis à l'aide de la commande « mmEffect » :

```
if(Pad.Newpress.A) //if fire is pressed
  {
  //Gun sound
  mmEffect(SFX_WLKRSHT2);
  }
```

Voila, nous avons démarré un petit projet de « Shoot Them Up ». Il reste cependant encore pas mal de travail : gestion des collisions, gestion des sprites ennemis, des scores, etc. Vous pouvez retrouver le code source complet de ce projet à l'adresse **[23]**.

Conclusion

Nous avons vu dans cet article l'utilisation de la bibliothèque PALib pour le développement de programme pour Nintendo DS. Cela nous a permis de mettre en œuvre les sprites, les arrière-plans, les sons et musiques grâce aux outils de développement devkitpro et le tout sous Linux. Les nombreux exemples fournis avec la PALib vous guideront lors de votre développement. N'hésitez pas à recompiler ces exemples, à les tester et à en étudier le code source. Bonne programmation... ■

Références

[1] http://www.free80sarcade.com/index.php

[2] http://www.monkeyisland.fr/

[3] http://fr.wikipedia.org/wiki/SWIV

[4] http://fr.wikipedia.org/wiki/Nintendo_DS

[5] http://www.01net.com/editorial/543108/piratage-les-linkers-nintendo-ds-juges-illegaux-en-france/

[6] http://fr.wikipedia.org/wiki/MoonShell

[7] http://libnds.devkitpro.org/

[8] http://www.playeradvance.org/forum/showthread.php?t=6103

[9] http://palib-dev.com/

[10] http://sourceforge.net/projects/pands/

[11] http://www.palib.info/wikifr/doku.php

[12] http://www.siteduzero.com/tutoriel-3-101567-programmez-sur-votre-nintendo-ds.html

[13] http://palib-dev.com/manual.html

[14] http://palib.info/wikifr/doku.php?id=day1#installation_automatique

[15] http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html#innerconsole

[16] http://downloads.sourceforge.net/project/devkitpro/

[17] http://sourceforge.net/projects/desmume/files/desmume/

[18] http://www.dslinux.org/

[19] http://www.uclinux.org/description/

[20] http://www.dslinux.org/wiki/moin.cgi/DSLinuxFAQ

[21] http://www.editions-diamond.com/opensilicium/index.php/interfaces-materielles-et-os-libres-pour-nintendo-ds-dslinux-et-rtems

[22] http://www.dslinux.org/builds/dslinux.nds

[23] http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html

[24] http://igm.univ-mlv.fr/~dr/XPOSE2007/rcalabroDevNintendoDS/index.html#introduction

[25] http://www.palib.info/downloads/PAGfx/Linux/

[26] http://yann.morere.free.fr/spip/spip.php?article149

PARCE QUE C'EST BIENTÔT LA SAINT VALENTIN...

UN CŒUR DE PHOTONS

par Yann Guidon
[Photoniste compulsif]

Les fêtes de fin d'année sont passées et vous ne savez pas quoi offrir à votre Douce (ou Doux) pour la Saint Valentin qui approche rapidement ? Les chocolats, ça fond et ça fait grossir (et ça ne dure pas), les fleurs, c'est plus présentable mais périssable, le restaurant, ce n'est pas du tout geek... Je vous propose une idée qui fera plaisir à la fois à la personne qui offre (vous ?) et à celle qui reçoit, un cadeau original qu'on ne trouve pas (encore) dans les magasins ! Faites vite avant que les copies asiatiques inondent le marché et cassent tout le romantisme...

1 Introduction

Le cadeau geek par excellence est utile, beau, et contrairement à un produit de Cupertino, économique et fait par votre douce panne de fer à souder à 300° C... Donc rien à voir avec un gadget ou un produit de luxe, il a été pensé spécialement par vous ! C'est ce qui compte, non ? Enfin, je vais quand même vous aider et montrer, photos à l'appui, les étapes de la fabrication et les astuces pour que cela marche du premier coup. Ensuite, vous faites ce que vous voulez...

Le principe de cet objet décoratif (enfin, c'est vous qui voyez) est d'abord de proposer une sorte de gazon de LED soudées sur une plaque de circuit imprimé. Rien de très folichon, ça fait de la lumière au plafond et ça prend de la surface horizontale sur le dernier haut-parleur libre.

Détail important, les broches des LED sont toutes gardées à la longueur originale, contrairement à des objets vaguement similaires qu'on trouve maintenant dans le commerce, tels le « Projecteur extérieur à LED blanc - 21W - 270 LED » (79€ chez Sélectronic, ref 8689-3). Couper les pattes, c'est d'abord un problème thermique : 21W ça chauffe ! Pour information, il existe des fers à souder de moindre puissance.

Connaissant les systèmes d'éclairage du commerce, la chaleur ne s'y échappe pas idéalement. Garder les pattes entières permet donc de dissiper la chaleur plus facilement et éviter que la puce de la LED n'atteigne des températures dangereuses. Ça n'a l'air de rien, mais ces deux pattes peuvent presque dissiper les 80mW nominaux... Les couper, c'est réduire leur durée de vie. Et un système qui chauffe n'est pas un bon système.

Mais surtout, ce que permettent ces longues pattes, c'est orienter la lumière. En effet, les LED en boîtier 5mm sont très directives, avec un faisceau d'environ 5° . Des LED au boîtier plus court, au format dit « strawhat », ont une ouverture plus grande, plus adaptée aux systèmes d'éclairage, mais ici nous allons plutôt faire dans la signalisation.

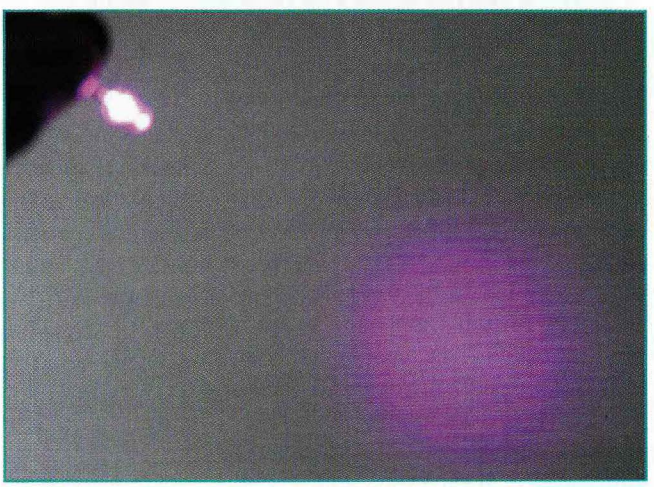


Figure 1 : Une LED en boîtier 5mm est assez directive pour créer un petit halo.

Donc une LED seule peut faire un joli point à quelques mètres (dans le noir et à puissance nominale). Un projecteur va mettre toutes les LED dans un faisceau parallèle (elles sont alors posées exactement à l'horizontale par une machine) et cela projette une sorte de tache. Avec nos longues pattes, nous pouvons mettre les points individuels où nous voulons ! Et que faire avec des points ? Moi j'ai choisi le dessin (approximatif) d'un cœur...

2 Des LED roses

Ce sont des LED de la couleur de l'amour (et des filles) que j'utilise pour ce projet (mais n'importe quelle LED 5mm en boîtier non diffusant fait l'affaire). Elles sont construites à partir de LED bleues, donc ont la même tension de fonctionnement, autour de 3,3V selon le fabricant, la luminosité et l'âge de la cousine de l'arrière-grand-mère par alliance du capitaine.

La couleur rose est obtenue selon le même principe que les LED blanches. On ajoute un produit sur la puce bleue, qui absorbe une partie de la lumière et la réémet en... rouge. Ce qui donne une sorte de violet. Pour les LED blanches, le produit absorbe le bleu pour donner du jaune car tout le monde sait que bleu+jaune, ça fait blanc. Je parle bien sûr du monde de la synthèse additive, comme votre écran d'ordinateur, qui n'a rien à voir avec la synthèse soustractive de la peinture ou de l'impression, où il faut du jaune, du cyan, du magenta et du noir pour obtenir toutes les couleurs.

Le rose est apparu il y a plusieurs années, après le blanc qui est plus utile, mais on commence à en voir un peu partout aussi. Et les couleurs ne sont pas toutes égales, la qualité de la LED dépend du fabricant, du soin qu'il apporte au dosage des produits... On a ce qu'on achète. Alors j'ai testé plusieurs sources, qui donnent des roses assez distincts, car par rapport au blanc, le rose est moins facile à étalonner. Fournissez-vous chez un seul distributeur si vous voulez une qualité uniforme ! À moins de vouloir jouer sur les variations de tons, si vous vous sentez l'âme artistique :)

3 Des LED, encore des LED...

Sur les 2 sachets de 100 exemplaires achetés sur eBay en Chine, le nombre de LED exact est de 197. Effectivement, j'ai joué avec lorsque je les ai reçues et

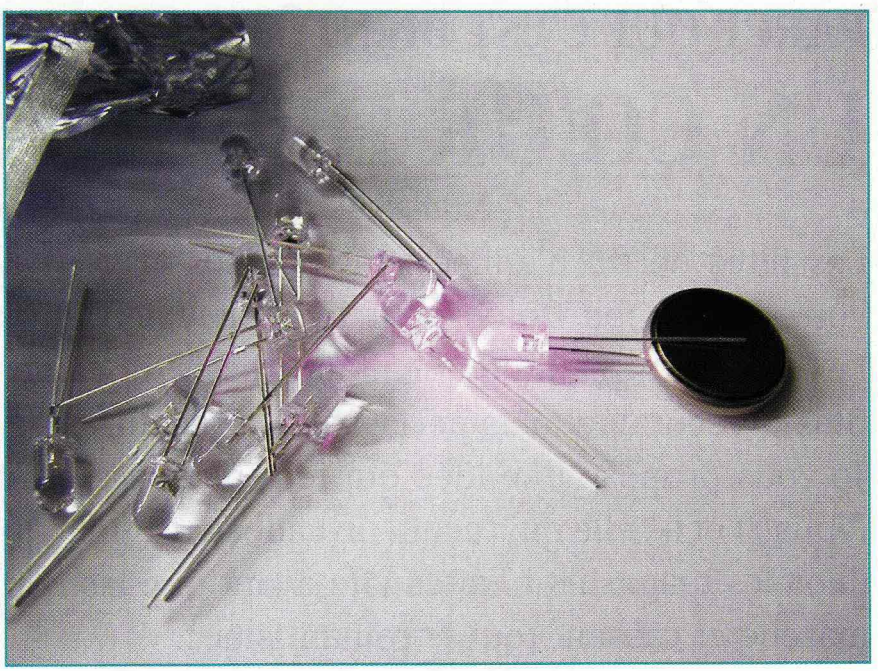


Figure 2 : Les LED roses utilisent la même technologie que les LED blanches.

j'en ai dispersé quelques-unes... Mais avec 197 LED, que pouvons-nous faire ?

Il faut idéalement les organiser en carré, c'est plus pratique. La racine carrée de 197 tombe à 14^2+1, c'est inespéré ! Heureusement que je n'en ai pas perdu d'autres et il faudra faire attention à ne pas en perdre plus...

Ensuite, il faut alimenter tout ce beau monde. Comme elles sont électriquement à base de LED bleues, elles ont une tension de fonctionnement d'un peu plus de 3V ; 3,1 ou 3,2V est une tension raisonnable pour ne pas les faire chauffer de trop. Si on prend une tension de 3,4V par LED et 14 LED en série, on arrive à une alimentation de presque 48V. Par contre, si on n'en met que la moitié en série, cela fait juste 7 LED et une alimentation de 24V, ce qui est une tension standard et facile à trouver dans le commerce.

Cela donne au total 28 segments de 7 LED et on peut calculer le courant consommé : 28×0,02=0,56A si on alimente les LED avec 20mA. La consommation totale sous 24V sera d'environ 14W, on peut arrondir à l'excès et prendre une alimentation de 20W.

Nous voilà donc avec d'une part la spécification de l'alimentation, d'autre part une géométrie.

4 L'alimentation

Donc c'est simple, on a besoin de 24V. Mais attention : bien que les LED doivent être alimentées en courant constant, nous allons avoir plusieurs circuits parallèles, chacun avec une résistance en série, ce qui transforme le système en tension constante. Il y a diverses façons d'y arriver et selon la disponibilité des composants, on obtiendra un système différent.

Figure 3 : Un module AC/DC compact

On peut partir d'une basse tension de sécurité, et un module élévateur de tension à découpage permet d'obtenir une tension régulée. Une version plus simple, si vous ne disposez que d'une basse tension, consiste à la faire osciller (ou de disposer directement d'une source alternative) pour créer une pompe de charges à diodes et condensateurs. C'est d'une simplicité et fiabilité redoutable et parfaitement adapté aux LED car l'impédance d'une pompe de charge réduit naturellement la tension si le courant augmente.

J'ai aussi un petit faible pour ce type de produit : « Module à découpage circuit imprimé Vertical » (Sortie 24V/1,5A sur picots au pas de 2,54mm, référence 2696-5 toujours chez Selectronic). Comme le nom l'indique, il s'agit de petits modules moulés à découpage qu'on peut souder sur un circuit imprimé. Ils sont disponibles avec les tensions de sortie 6V, 9V, 12V, 15V et 24V. Cependant, les dimensions et surtout la sécurité mécanique de l'arrivée du secteur sont à prendre en compte. Le circuit imprimé avec les LED sera presque à nu et le poids du fil du secteur risque de faire bouger l'ensemble, le rendant instable donc potentiellement dangereux.

Un autre paramètre est la sécurité électrique en cas de court-circuit au secondaire ou pour éviter de suralimenter les LED. C'est aussi pour cela que je retourne depuis peu aux transformateurs classiques, qui sont plus fiables sur le long terme que les alimentations à découpage. 3 transformateurs à découpage qui claquent en une semaine, ça fait réfléchir et en plus on ne peut même pas les réparer...

Les alimentations à bobinage sont plus primitives et ont une impédance interne (provenant de plusieurs facteurs) qui réduit la tension de sortie lorsque le courant augmente. Dans de nombreux cas, ce n'est pas désiré, mais justement, ici, cela réduit les chances d'emballement thermique des LED. Il faut cependant prendre grand soin de calculer et trouver les paramètres de fonctionnement idéaux, donc bien choisir la tension de sortie.

Vous savez que la tension crête de sortie est égale à la tension nominale multipliée par 1,4 environ, non ? Par exemple, les transformateurs habituels du commerce sont donnés pour 12V nominal mais la tension peut atteindre 17V à vide. C'est beaucoup, même si c'est encore insuffisant pour atteindre les 24V requis. Mais on peut ruser !

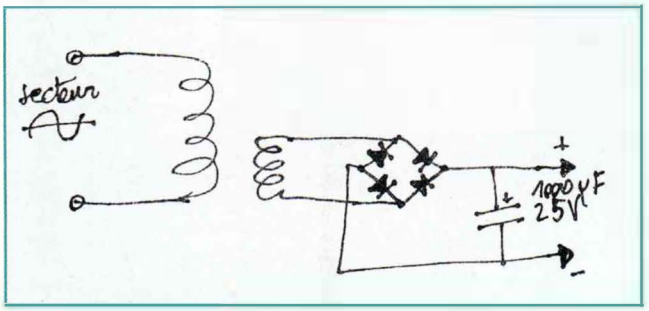


Figure 4 : Schéma d'un bloc d'alimentation non régulé

Les transformateurs classiques non régulés sont vraiment simples, ils sont normalement constitués des enroulements du primaire et secondaire, suivis d'un pont de diodes et d'un condensateur de filtrage. Le pont de diodes est à double alternance, ce qui assure un bon rendement, mais ce n'est pas la seule topologie de redressement possible. Si on dispose d'une tension alternative, on peut obtenir cette tension en positif et négatif avec juste deux diodes ! Il faut par contre ajouter un autre condensateur.

Nous désirons environ 24V en sortie, tension double de la crête du transformateur, avec un coefficient de 1,4 supplémentaire, donc on a besoin de 24/(2*1,4)=8V alternatif. Les transformateurs fournissent environ 9V, qui chute d'environ 0,7V au travers des diodes, ce qui donne quasiment nos 8V nominaux.

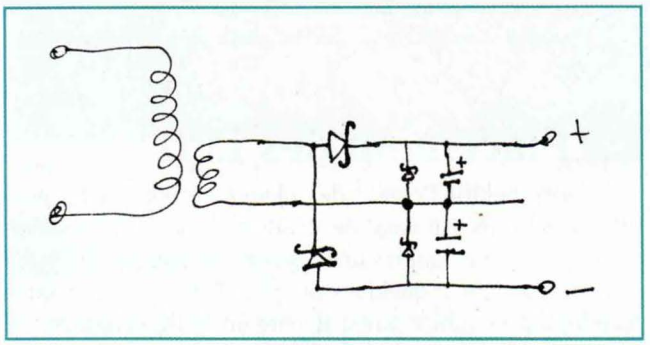


Figure 5 : Comment créer une tension double en modifiant la topologie du pont de diodes. Cette version peut aussi être alimentée en courant continu. Les diodes en parallèle des condensateurs servent à éviter une éventuelle inversion néfaste de la polarité à ce niveau, ce qui les endommagerait.

Pour créer ce montage, il faut évidemment disposer d'un transformateur facile à démonter et remonter (donc avec des vraies vis). Au passage, en plus de recâbler le pont de diodes, on peut aussi changer les condensateurs pour augmenter leur capacité et réduire le scintillement puisque les condensateurs ne sont plus chargés à 100Hz (comme en double alternance) mais à 50Hz.

L'autre solution consiste (encore une fois) à avoir de la chance... J'ai effectivement trouvé un transformateur moulé de 9V 1,3A à sortie alternative ! Le redressement et le filtrage seront donc réalisés sur le circuit imprimé, au dernier moment.

Par contre la mesure indique 10,67V et l'oscilloscope montre 31V crête à crête, il va falloir ruser encore une fois. Je n'avais pas envie de me déplacer pour aller acheter un régulateur LM7824 « tout fait », et aussi je ne voulais pas gaspiller quelques volts de chute dans le régulateur bipolaire, donc j'ai imaginé un montage « limiteur de tension », inspiré d'un limiteur de courant, à base de diodes Zeners, de transistors bipolaires et d'un MOSFET. En attendant qu'il soit validé, j'utilise une alimentation qui fournit directement 24V.

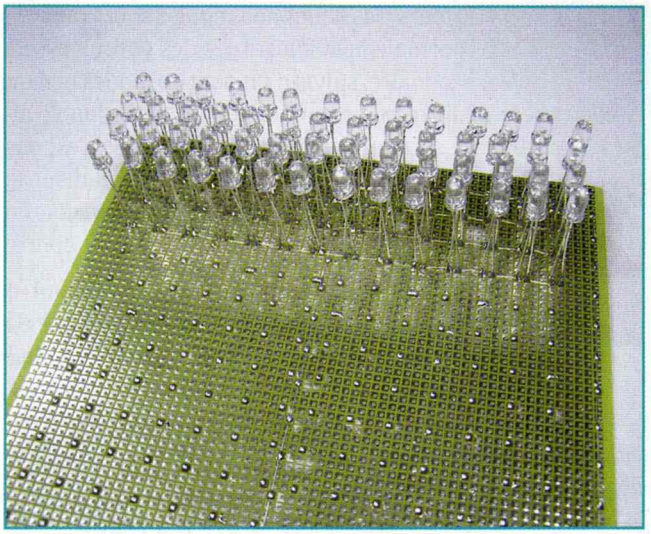


Figure 6 : Les LED sont soudées côté cuivre et vrillées.

Figure 7 : les cavaliers, ou jumpers, permettent d'alimenter indépendamment chaque chaîne de 7 LED pour faciliter le réglage du motif.

5 Placement des LED

J'utilise habituellement des plaques prétrouées à pastilles avec un écartement de 2,54mm. Les LED sont (dans ce cas) placées à 1cm les unes des autres, ce qui fait 14cm pour chaque côté, et quelques-uns pour la marge (nécessaire pour les autres composants). Encore un heureux hasard, j'ai justement une plaque de 16cm de large, que je peux couper pour faire un carré.

L'alimentation arrive par un connecteur standard et répartit les électrons sur trois rails, avec le commun au milieu, et le plus aux deux extrémités. On ne doit pas oublier de placer les LED dans un sens sur une moitié, et dans l'autre sens pour l'autre moitié.

J'effectue la soudure du côté composants, sur le dessus, car cela facilite le placement avec les longues pattes. Cela n'a pas que des avantages, mais autrement, il faudrait retourner le circuit et la LED se baladerait hors de vue, et quand on a plein de LED, c'est absolument ingérable... Donc le montage en surface est une nécessité pratique. Si c'est plus beau ou moins beau, comme certains artistes le confirmeront, cela tient juste à l'explication qu'on donne à l'observateur qui découvre.

Les pattes des LED n'ont pas la même longueur, mais la différence est (à peu près) égale à l'épaisseur du circuit imprimé (soit presque 1,6mm) et donc la patte courte (le pôle négatif) peut reposer sur le circuit alors que l'autre traverse le trou. Ainsi, toutes les LED sont quasiment à la même hauteur et vont paraître uniformes vues du dessus.

Enfin, pour faciliter le réglage mécanique final, l'astuce consiste à « vriller » la LED sur ses pattes en la tournant

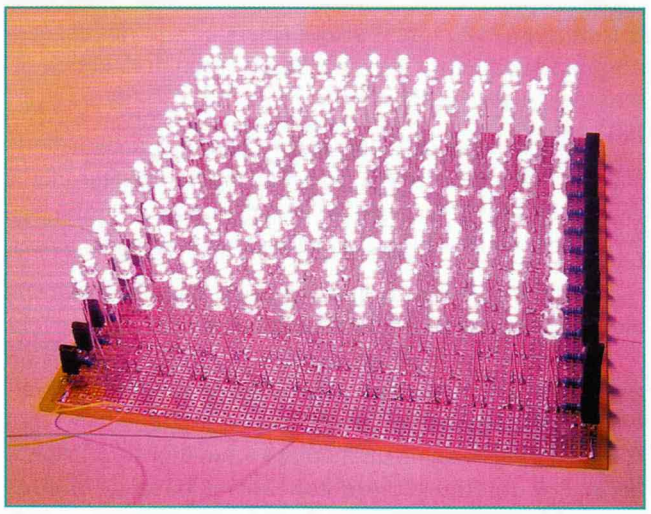


Figure 8 : Les 196 LED sont assemblées et on fait un petit burning test. De l'air chaud s'élève, mais aucune partie n'est brûlante, le système fonctionne comme prévu !

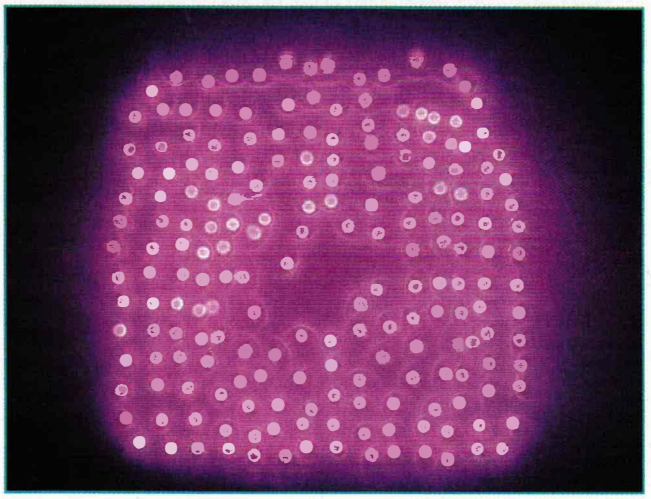


Figure 9 : Une simple feuille de papier aide à orienter les LED.

d'un quart de tour. Cela permettra donc de l'orienter dans deux dimensions sans forcer sur les pattes, mais en faisant bien attention à ne pas faire de court-circuit si les pattes se touchent...

6 La résistance

En théorie, il y a le calcul. En pratique, la mesure fait foi...

7 LED du sachet sont placées en série sur une plaque d'expérimentation sans soudure et une alimentation de laboratoire monte progressivement à 20mA, et on trouve 21,4V.

La tension à chuter dans la résistance est de 24-21,4=2,6V et la résistance correspondante est R=U/I=2,6/0,2=130 Ohms. Une résistance de 100 ou 120 Ohms convient donc.

Avec 100 Ohms, le courant mesuré est de 23mA et j'ai trouvé une chute de 2,48V aux bornes de la résistance, soit une dissipation de puissance de 0,023 x 2,48 = 57mW. Pour ce système, je vais utiliser des résistances traversantes standards « quart de watt », mais des résistances CMS en boîtier 1206 (1/8e de Watt) conviendraient aussi. Et puis ça ajoute un peu de couleur pour rompre la monotonie des pattes des LED :)

Il n'y a pas d'élèvement notable de la température au niveau des pattes, ce qui valide le choix. La température est plus élevée près de la partie plastique, mais descend à l'extrémité. Si on avait coupé, la LED n'aurait pas pu refroidir correctement !

Il est aussi intéressant de mesurer la dispersion des caractéristiques des LED. Avec le même courant, je mesure à leurs bornes : 3,03V, 3,21V, 3,02V, 3,14V, 3,10V, 3,08V, 3,14V. La résistance est donc vraiment nécessaire pour stabiliser le courant.

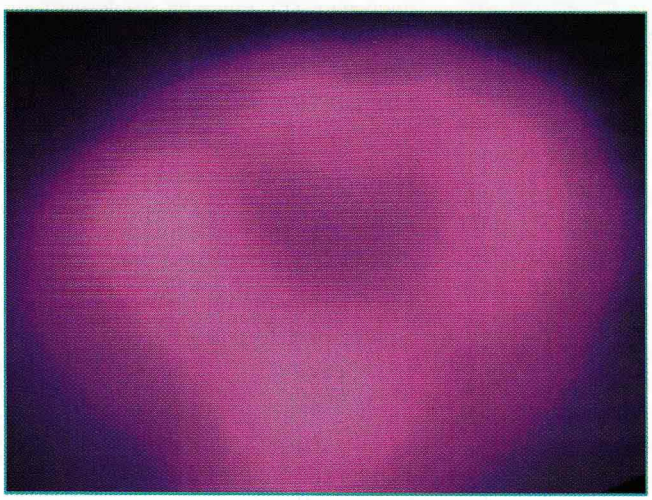


Figure 10 : Le résultat n'est pas encore parfait, mais si on sait qu'il y a une forme vaguement cardioïde, on arrive à la voir :)

Figure 11 : Le résultat varie beaucoup en fonction de la distance.

7 Le réglage

La partie vraiment délicate du système est le réglage du dessin. Si on allume toutes les LED, alors on ne voit plus rien...

La solution consiste à alimenter les LED par petits groupes. Des barrettes sécables et des jumpers font l'affaire, cela permet d'allumer chaque groupe de 7 LED indépendamment. Cela fait 28 jumpers, mais d'autres solutions existent.

Pour régler le dessin, on peut s'aider en plaçant une grande feuille dans le flux lumineux des LED. Le motif est dessiné sur la feuille et les parties sont subdivisées, comme ça on sait quelle LED doit pointer où.

Conclusion

Je suis sûr qu'il y a encore plein de variations possibles sur cette idée d'un « jardin de LED ». Mélanger les couleurs, faire varier les groupes indépendamment, projeter le logo de votre société sur une vitre dépolie... À vous de jouer !

Cet article a été dédié aux LED et bien que le défi soit purement matériel et électronique (pas un poil de ligne de code), il y a encore plein d'articles à faire à ce sujet, qui est très très vaste !

Envoyez-moi les photos de vos réalisations, les remarques et les réactions de votre amoureux/se et n'oubliez pas de vous amuser ! Qu'il s'agisse de logiciel ou de bricolage, c'est bien le partage qui compte. Quant à moi, j'ai la fierté d'avoir créé le cadeau parfait pour mon Ange... ■

CRÉEZ UN CAPTEUR DOMOTIQUE AVEC ARDUINO ET XBEE

...ET INTERFACEZ-LE AVEC VOTRE SOLUTION DOMOTIQUE GRÂCE AU PROTOCOLE XPL

par Frédéric Le Roy

Dans le numéro 140 de Linux Magazine, nous avons vu avec un Arduino et son shield ethernet comment gérer un ruban de LED grâce au protocole xPL. Aujourd'hui, nous allons voir comment créer un capteur qui va émettre ses valeurs par xPL via une liaison Xbee.

1 Problématique de départ

1.1 La théorie

En domotique, il existe de nombreux types de capteurs et dans de nombreuses technologies : capteurs de température, d'humidité, de présence, d'ouverture, de fumée, ... Toutefois, il n'existe pas toujours de capteur correspondant à tous les besoins de l'habitant de la maison. De même, certains capteurs peuvent se retrouver dans des endroits où il sera impossible d'emmener un câble réseau (par exemple). De même, dans le cas d'un capteur extérieur, emmener un câble réseau relié à votre LAN dans le jardin présente un risque pour votre installation informatique et domotique.

1.2 36 15 code ma vie

Étant possesseur d'une cuve d'eau enterrée dans le jardin, j'ai voulu connaître le niveau d'eau de cette cuve. Il existe plusieurs méthodes pour connaître un niveau d'eau : capteur de pression, utilisation de plusieurs fils pour utiliser la conductivité de l'eau, capteur de distance à ultrasons. Dans mon cas, j'ai choisi d'utiliser un capteur de distance à ultrasons. La question suivante s'est donc posée : comment relier ce capteur à mon installation domotique ? Ayant déjà pratiqué l'Arduino, j'ai voulu utiliser un Arduino comme brique de base.

La solution domotique que j'utilise (et à laquelle je contribue) s'appelle Domogik **[Domogik]** et repose sur le protocole xPL, je désirais donc créer un capteur émettant ses valeurs via xPL. XPL est un protocole de communication qui utilise le protocole UDP comme couche de transport. Il faudrait donc logiquement passer par une interface réseau (filaire ou Wi-Fi). Comme mentionné plus haut, un câble réseau à l'extérieur de la maison et connecté au LAN présente un risque. Dans ce cas, le shield Wi-Fi me faisait penser à un bazooka pour tuer une mouche, sans compter son prix élevé. La solution du vrai réseau n'étant pas envisageable, je me suis penché sur les solutions radio. Xbee, émetteur 433Mhz, Bluetooth, il y avait plusieurs solutions, les moins chères étant le bluetooth (pas assez de portée dans notre cas) et l'émetteur 433MHz. L'inconvénient de ce dernier étant qu'il faille aussi réaliser un récepteur 433MHz. Comme je souhaitais réaliser quelque chose de facile à mettre en œuvre (afin que tout le monde puisse en profiter ou s'en inspirer), je me suis penché sur le Xbee : longue portée, faible consommation et mode veille (pour des futurs capteurs sur piles) et surtout une communication de type série native avec les puces de Série 1 : cette solution avait décidément plusieurs avantages ! Il ne restait qu'à faire un plugin sous Domogik pour faire une passerelle Xbee > UDP et des messages xPL envoyés sur le port série de l'Arduino iraient directement sur le LAN !

1.3 Quel capteur à ultrasons ?

Quelques recherches m'ont rapidement orienté vers le HC-SR04 **[HC-SR04]** :

- Il s'alimente en 5v, ce qui est une tension fournie nativement par l'Arduino.
- Son angle effectif est inférieur à 15°, ce qui assure une mesure droit devant (ou presque).
- Il peut mesurer de 2cm à 5m avec une résolution de 0,3cm. Ma cuve

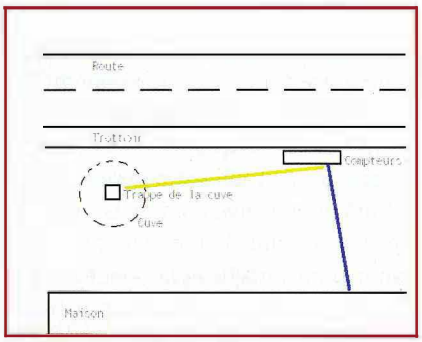
fait moins de 3 mètres de profondeur (ouverture comprise) et une telle précision est plus que suffisante dans mon cas.

- Il est très simple à utiliser.

Notez qu'il existe des modèles de capteur à ultrasons faits pour les environnements humides. Il sont par contre quatre fois plus chers.

1.4 Emplacement des éléments

Les premiers soucis que j'ai eus à résoudre étaient la répartition géographique des différents éléments.



Situation géographique de la cuve

J'avais déjà une gaine vide (trait bleu) qui allait de mon tableau électrique jusqu'à côté des compteurs. À cet emplacement, on retrouve également ma boîte aux lettres et le compteur d'eau. Il n'y avait par contre aucune gaine arrivant à proximité de la cuve.

J'ai choisi de placer l'Arduino dans un boîtier étanche à côté des compteurs. Ceci me permettra d'ajouter d'autres fonctionnalités par la suite :

- relevé de la consommation d'eau ;
- détection de courrier dans la boîte aux lettres ;
- capteur PIR pour détecter une présence devant la maison (afin d'allumer les lumières) ;
- mesure de température extérieure, de la luminosité, de l'humidité ;
- etc.

J'ai donc ajouté une gaine (trait jaune) entre l'Arduino et la cuve.

1.5 Lien entre les éléments

Pour alimenter l'Arduino, un câble réseau sera utilisé (enfin, 2 des 8 fils). De la même manière, un second câble sera utilisé pour relier le capteur à ultrasons à l'Arduino. L'utilisation de ce type de câble se justifie par le peu de perte qu'ils occasionnent.

1.6 Bilan

Pour réaliser ce capteur, il va donc falloir :

- un Arduino (Uno dans mon cas) ;
- un shield Xbee ;
- une puce Xbee Série 1 ;
- un capteur à ultrasons (HC-SR04).

Du côté de l'ordinateur qui héberge la solution de domotique, il va falloir :

- une interface USB/xbee (fournie sans puce Xbee généralement) ;
- une puce Xbee Série 1.

2 Ce qui ne sera pas traité dans l'article

Avant toute chose, sachez que j'ai fait l'impasse sur l'aspect autonome du montage dans cet article. En effet, le montage et le code que je vais présenter ne permettent pas l'utilisation de piles ou batteries (ou alors il vous faudra les changer plusieurs fois par semaine...). L'Arduino Uno, à cause de son régulateur de tension et du port USB, a une consommation minimale qui supprime la possibilité de tenir 2 ou 3 ans avec des piles ou une batterie malgré les modes d'économie d'énergie de l'Arduino et du shield Xbee.

Si vous désirez creuser ces aspects, je vous invite à consulter le lien **[Battery]** qui donne des pistes pour faire tenir un Arduino 2 à 3 ans avec des piles (à condition de sortir le fer à souder).

3 (Tout petit) focus sur le Xbee

Les composants Xbee sont des composants sans fils qui implémentent plusieurs protocoles en fonction des versions : 802.15.4, ZigBee, ... Ils permettent de réaliser très simplement des liaisons série sans fil.

Il existe notamment 2 séries : Série 1 et Série 2 et chacune est déclinée dans une version non pro (puissance de 1mW) et pro (puissance de 100mW). Les 2 séries ne sont pas interconnectables et il faut donc choisir celle qui correspond le mieux à vos besoins.

Dans mon cas, la série 1 non pro me suffisait amplement : elle supporte nativement la liaison série et est suffisante en termes de portée.

Notez qu'en cas de saturation de la bande des 2.4GHz chez vous (utilisation massive de Wi-Fi, bluetooth et autres protocoles sur cette bande), la portée des puces semble fortement diminuer.

4 Le montage

Le montage ici va être très simple : on connecte respectivement les pins **Vcc** et **Gnd** du HC-SR04 aux pins 5V et Gnd de l'Arduino. Ensuite, il reste à connecter les pins **Trig** et **Echo** à 2 entrées/sorties numériques de l'Arduino. Ici, j'ai choisi les E/S 2 (**Echo**) et 3 (**Trig**).

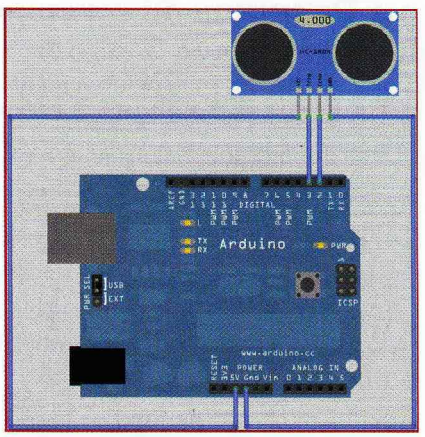
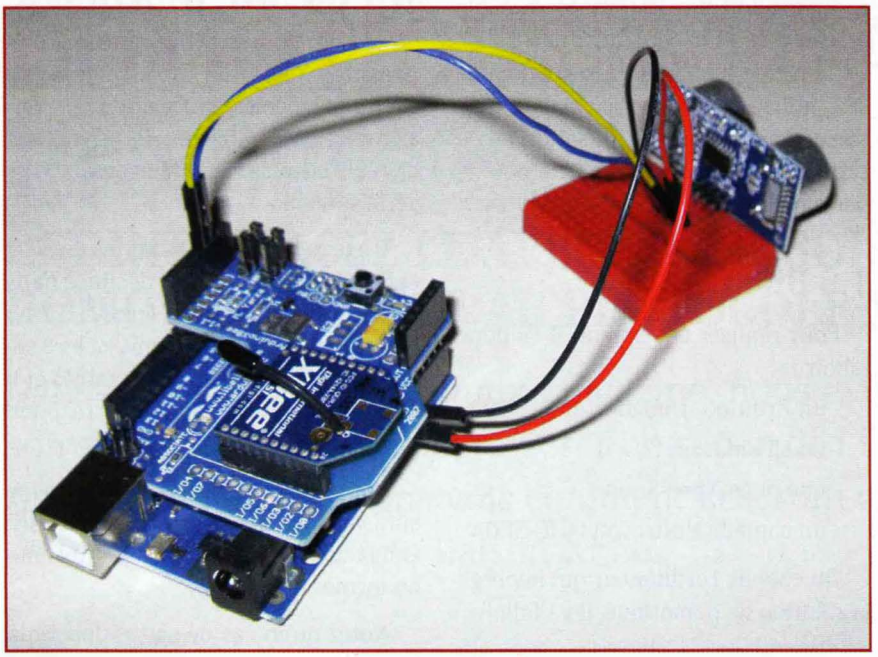


Schéma de branchement

Remarquez sur la photo du montage prototype que le shield Xbee « bouche » l'accès à certaines pins (**Vcc**, **Gnd**, **Reset**, ...). Il faudra donc plier le bout des straps pour accéder à ces pins.



Montage de test

4.1 Le shield Xbee

Le shield Xbee est équipé de deux cavaliers. Si les deux sont mis sur la position USB, la liaison série passe par USB. Dans le cas contraire (position Xbee), la liaison se fait via ondes radio.

5 xPL

Contrairement à la dernière fois, je n'expliquerai pas aujourd'hui le protocole xPL. La description des points de spécification nécessaires à la compréhension du code xPL du capteur avait été abordée dans l'article précédent.

5.1 Définition du schéma xPL utilisé pour envoyer les valeurs

5.1.1 Choix des valeurs à envoyer sur xPL

Tout d'abord, que voulons-nous recevoir comme valeurs ? Il y a la distance entre le capteur et l'eau. Toutefois, cette valeur n'est pas très intuitive :

- Cela représente la quantité de « vide » dans la cuve.
- Le niveau d'eau dans la cuve n'arrivera jamais au niveau du capteur car s'il y a trop d'eau, elle sera évacuée par un trop plein (un tuyau allant au tout à l'égout). Sur la distance mesurée, il y a donc une partie qui est en trop.

Y a-t-il un réel intérêt à envoyer cette valeur ? Pas vraiment à première vue. L'utilisateur lambda désirera connaître soit le volume d'eau en pourcentage (éventuellement en m3), soit la hauteur d'eau dans la cuve en pourcentage.

Calculer la hauteur d'eau en pourcentage est relativement simple dès que l'on connaît la profondeur totale de la cuve (depuis le capteur) et la distance entre le capteur et le trop plein. Il nous suffit donc d'ajouter ces deux paramètres dans la configuration du sketch.

Calculer le volume est plus complexe : ceci dépend de la forme de la cuve. N'ayant pas les dimensions précises de la mienne, j'ai fait l'impasse sur ce point.

Au final, nous n'avons qu'à envoyer en valeur la hauteur de la cuve en pourcentage.

Non !!!! Pourquoi non ? Parce que !

J'ai choisi d'envoyer tout de même la distance mesurée par le capteur via xPL. Logiquement, cette valeur ne devrait pas être exploitée. Toutefois, je ne sais pas si l'eau est pompée depuis le fond de ma cuve ou depuis un point légèrement plus haut que le fond. Souhaitant pouvoir ajuster la valeur lorsque ma cuve sera vidée lors de la prochaine sécheresse, j'ai décidé de garder cette valeur : une mise à jour massive de la hauteur en base en fonction de la distance me permettra de remettre les statistiques d'aplomb.

5.1.2 Le schéma xPL

Le schéma xPL le plus adapté pour envoyer les valeurs d'un capteur est **SENSOR.BASIC** **[sensor.basic]**. Il se présente ainsi :

```
sensor.basic
{
device=<adresse du capteur ou son nom>
type=<type de capteur>
current=<valeur>
[lowest=<valeur minimale enregistrée>]
[highest=<valeur maximale enregistrée>]
[units=<unité de la valeur>
}
```

Device devra être paramétrable par l'utilisateur, **current** sera la valeur envoyée et **type** correspond au type de données. **Lowest** et **highest** sont des informations dont nous n'avons pas besoin lorsque la solution domotique

utilisée historise les valeurs, ce qui est le cas pour la majorité d'entre elles. **Units** permet de préciser l'unité : nous l'utiliserons à titre informatif.

Les deux messages que nous enverrons à chaque mesure auront la même valeur **device**, mais auront deux **type** (et 2 **units**) différents :

- **percent** : pour le pourcentage de niveau d'eau (**units** = %). Il ne s'agit pas d'un type officiel de ce schéma, mais comme il n'y a aucun type officiel compatible, nous pouvons en utiliser un nouveau.
- **distance** : pour la distance en centimètres (**units** = cm). Il s'agit d'un type officiel.

Si la valeur mesurée est la même que la précédente, nous utiliserons un message de type (xPL) **xpl-stat**, dans le cas contraire, nous utiliserons le type **xpl-trig**.

5.2 Message de heartbeat

Pour rappel, le message de heartbeat est **HBEAT.BASIC** pour un client xPL hardware ou **HBEAT.APP** pour un client xPL sur PC et fait partie intégrante du protocole xPL. Il doit (entre autres) être émis au démarrage du client, puis de manière régulière, et indiquer dans son corps à quel rythme il est émis. Ayant déjà abordé le sujet dans le précédent article, j'invite ceux qui l'ont loupé à lire la page décrivant le protocole xPL **[proto xPL]**. Dans la majorité des cas, les clients xPL émettent ce message toutes les 5 minutes. Personnellement, je préfère mettre un intervalle de 1 minute afin que le statut du montage dans l'administration de Domogik représente au mieux la vérité.

5.3 Choix du vendor id et device id

Quand un message xPL est émis, dans l'en-tête, on retrouve l'adresse du client qui a émis le message : la **source**.

Exemple sur un message envoyé par le composant REST de Domogik :

```
xpl-stat
{
hop=1
source=domogik-rest.darkstar
target=*
}
...
```

La source est composée ainsi : **<vendor id>-<device id>.<instance id>** :

- **vendor id** est l'identifiant du constructeur du matériel. Ici, pas de vendeur, mais un matériel bien connu : l'Arduino. Nous allons donc utiliser cette valeur.
- **device id** est l'identifiant du type de device chez le constructeur. Dans un premier temps, notre montage va gérer uniquement la cuve. Je vais donc choisir « tank » comme valeur. Par la suite, lorsque j'ajouterai des capteurs au montage, je changerai cette valeur.
- L'instance **id** est une valeur définie par le constructeur. Dans le cas d'un client xPL software, il s'agit généralement du nom de la machine hébergeant le client. Dans le cas d'un client xPL hardware, je conseille de donner comme valeur l'emplacement géographique du montage ou un nom qui vous permet de comprendre tout de suite de quel montage il s'agit. À chacun de personnaliser cette valeur, donc.

5.4 Les messages à envoyer

Au final, nous devrons donc envoyer les cinq messages suivants :

- le **HBEAT.BASIC** :

```
xpl-stat
{
hop=1
source=arduino-tank.macuve
target=*
}
hbeat.basic
{
interval=5
}
```

- le **SENSOR.BASIC** pour la distance en centimètres :

```
xpl-stat
{
hop=1
source=arduino-tank.macuve
target=*
}
sensor.basic
{
type=distance
device=macuve
current=82.67
units=cm
}
```

- le **SENSOR.BASIC** pour le niveau de la cuve en pourcentage :

```
xpl-stat
{
hop=1
source=arduino-tank.macuve
target=*
}
sensor.basic
{
type=percent
device=macuve
current=81.0
units=%
}
```

ainsi que la version **xpl-trig** des deux derniers messages.

6 Le code

Le code ci-dessous peut être utilisé avec ou sans Xbee : il utilise les commandes Arduino pour gérer et écrire sur le port série. Sans puce Xbee, les données seront émises sur le port série (via USB, donc, dans le cas de l'Arduino Uno). Avec la puce Xbee (si les jumpers du shield Xbee sont dans la bonne position), les données seront émises via ondes radio.

Vous pouvez donc uploader ce code sur un Arduino Uno et voir les messages xPL passer dans le moniteur série, ce qui est bien plus simple pour debugger !

6.1 Organisation du code

Le code va être réparti dans 4 fichiers :

- **ar_tank.pde** : le fichier principal ;
- **float.pde** : un fichier contenant une fonction permettant l'affichage de flottants via **sprintf** ;
- **time.pde** : gestion du temps ;
- **xpl.pde** : les fonctions xPL.

6.2 ar_tank.pde

```
#define PIN_DISTANCE_TRIG 2      // pin ECHO du HC-SR04
#define PIN_DISTANCE_ECHO 3      // pin TRIG du HC-SR04

/***************** Configuration xPL ******************/
// Adresse du device dans le corps du message xPL
#define MY_DEVICE "macuve"

// Source : <vendor id>-<device id>.<instance>
// Vendor id et device id ne devraient pas être changés
// Le montage gère une seule cuve, l'instance peut donc avoir la
valeur de MY_DEVICE
#define MY_SOURCE "arduino-tank." MY_DEVICE

// Durée en secondes d'une minute (utile pour accélérer le temps en
phase de dev)
#define MINUTE 60

// Intervalle en minutes entre l'envoi de 2 message de Heartbeat
interval
#define HBEAT_INTERVAL 1

// Intervalle en minutes entre 2 mesures de distance
#define DISTANCE_INTERVAL 1

/***************** Configuration de la cuve *****************/
// Profondeur totale de la cuve (depuis la fermeture jusqu'au fond)
#define TANK_DEPTH 230
// Distance entre le capteur et le niveau maximum que peut atteindre
l'eau
#define TANK_ZERO 50

/******************** Compteurs du temps ******************/
int second = 0;
int lastTimeHbeat = 0;
int lastTimeDistance = 0;

/******************** Historique des valeurs ***************/
float lastDistance = 0;
int lastPercent = 0;

/************************** Setup **************************/

void setup() {
    // En cas de problème dans le code (reboot dès le début à cause
    // d'un mauvais accès mémoire, ...) on se donne du temps pour
    // uploader un autre sketch
    delay(10);
    // configuration des entrées/sorties
    pinMode(PIN_DISTANCE_ECHO, OUTPUT);
    pinMode(PIN_DISTANCE_TRIG, INPUT);
    // initialisation de la laison série
    Serial.begin(9600);

    // Envoi d'un message xPL Hbeat au démarrage
    sendHbeat();
}


/************************** Loop ***************************/

void loop() {
    // Gestion des secondes
    pulse();
    // Chaque seconde on va vérifier si on a des choses à faire
    if (second == 1) {
```

```
        // Hbeat
        lastTimeHbeat += 1;
        // Toutes les N minutes, on envoit un Hbeat
        if (lastTimeHbeat == (MINUTE*HBEAT_INTERVAL)) {
            sendHbeat();
            lastTimeHbeat = 0;
        }

        // Mesure de distance
        lastTimeDistance += 1;
        // Toutes les N minutes on mesure la distance
        if (lastTimeDistance == (MINUTE*DISTANCE_INTERVAL)) {
            // Récupération de la distance avec le capteur à ultra sons
            float distance = getDistance();
            // On détermine si on envoie un xpl-trig ou un xpl-stat
            char xpltype[9];
            if (distance != lastDistance)
                sprintf(xpltype, "xpl-trig");
            else
                sprintf(xpltype, "xpl-stat");
            lastDistance = distance;
            char strDistance[15];
            // Envoi de la valeur via xPL
            sendSensorBasic(xpltype, "distance", MY_DEVICE,
            ftoa(strDistance, distance, 2), "cm");
            // Calcul du pourcentage d'eau (en hauteur) dans la cuve
            int percent = distanceToPercent(distance);
            // On détermine si on envoie un xpl-trig ou un xpl-stat
            if (percent != lastPercent)
                sprintf(xpltype, "xpl-trig");
            else
                sprintf(xpltype, "xpl-stat");
            lastPercent = percent;          // Envoi de la valeur via xPL
            sendSensorBasic(xpltype, "level", MY_DEVICE, ftoa(strDistance,
            percent, 2), "%");
            lastTimeDistance = 0;
        }
    }
}


/*********************************************
   getDistance
   Mesure de la distance avec un HC-SR04
   Input : n/a
   Output : distance en cm
*********************************************/
float getDistance() {
    // On met un niveau bas sur la pin Echo
    digitalWrite(PIN_DISTANCE_ECHO, LOW);
    delayMicroseconds(2);
    // On envoi l'écho (niveau haut sur la pin Echo pendant 10us)
    digitalWrite(PIN_DISTANCE_ECHO, HIGH);
    delayMicroseconds(10);
    digitalWrite(PIN_DISTANCE_ECHO, LOW);
    // On récupère le temps de réponse en millisecondes
    long responseTime = pulseIn(PIN_DISTANCE_TRIG, HIGH);
    // Conversion du temps en distance (cm) :
    // - La vitesse du son est de 340m/s
    // - Ici nous avons le temps en ms et nous voulons la distance en cm
    // - Vitesse du son avec les unités qui nous intéressent : 0,034cm/us
    // - Il faudra diviser la valeur par 2 à cause de l'aller retour
    float distance = 0.034 * responseTime / 2;
    return distance;
}
```

```
/*********************************************
   distanceToPercent
   Conversion de la distance entre le capteur et l'eau en un niveau
   d'eau en pourcentage de la hauteur de la cuve
   Input : n/a
   Output : niveau d'eau en pourcents
   Note : cette fonction peut renvoyer des valeurs négatives ou
supérieures à 100
         si la configuration des paramètres de la cuve est mauvais.
*********************************************/
float distanceToPercent(float distance) {
    int percent;
    // profondeur utile de la cuve
    int tankDepthFromZero = TANK_DEPTH - TANK_ZERO;
    // distance entre le niveau maximum d'eau et le niveau actuel
    float distanceFromZero = distance - TANK_ZERO;
    // calcul du pourcentage
    percent = 100 - (100 * distanceFromZero)/tankDepthFromZero;
    return percent;
}
```

6.3 xpl.pde

```
/*********************************************
   sendHbeat
   Envoi d'un message xPL de Hbeat sur la liaison série
   Input : n/a
   Output : n/a
*********************************************/

void sendHbeat() {
    char buffer[200];

    // en-tête
    sprintf(buffer, "xpl-stat\n{\n");
    sprintf(buffer, "%shop=1\n", buffer);
    sprintf(buffer, "%ssource=%s\n", buffer, MY_SOURCE);
    sprintf(buffer, "%starget=*\n}\n", buffer);

    // corps
    sprintf(buffer, "%shbeat.basic\n{\n", buffer);
    sprintf(buffer, "%sinterval=%i\n", buffer, HBEAT_INTERVAL);

    sprintf(buffer, "%s}\n", buffer);

    // envoi du message sur le port série
    Serial.println(buffer);
}


/*********************************************
   sendSensorBasic
   Envoi d'un message sensor.basic sur la liaison série
   Input : char* : xpltype : type du message xpl à envoyer
           char* : type : type de données
           char* : device : adresse du device
           char* : current : valeur
           char* : units : unités
   Output : n/a
*********************************************/

void sendSensorBasic(char * xpltype, char *type, char *device, char
*current, char *units) {
    char buffer[200];

    // en-tête
    sprintf(buffer, "%s\n{\n", xpltype);
    sprintf(buffer, "%shop=1\n", buffer);
    sprintf(buffer, "%ssource=%s\n", buffer, MY_SOURCE);
    sprintf(buffer, "%starget=*\n}\n", buffer);

    // corps
    sprintf(buffer, "%ssensor.basic\n{\n", buffer);
    sprintf(buffer, "%stype=%s\n", buffer, type);
    sprintf(buffer, "%sdevice=%s\n", buffer, device);
    sprintf(buffer, "%scurrent=%s\n", buffer, current);
    sprintf(buffer, "%sunits=%s\n", buffer, units);

    sprintf(buffer, "%s}\n", buffer);

    // envoi du message sur le port série
    Serial.println(buffer);
}
```

6.4 time.pde

```
long previousSecond = 0;

/*********************************************
   pulse
   Valorise la variable second à 1 chaque seconde (et à 0 sinon)
   Input : n/a
   Output : n/a
   Note : il faut déclarer "int second = 0" dans le pde principal
*********************************************/

void pulse() {
    // on regarde si une seconde s'ets écoulée
    if (abs(millis() - previousSecond) > 1000) {
        second = 1;
        previousSecond = millis();
    }
    else {
        second = 0;
    }
}
```

6.5 float.pde

```
/*********************************************
   ftoa
   Convertit un float en une chaîne. Cette fonction est faite pour être appelée
   directement depuis un sprintf
   Input : char* : chaîne buffer
           float : valeur à afficher
        int : précision voulue à l'affichage
   Output : valeur convertie en chaîne de caractères
   Source : http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1164927646/6#6
*********************************************/
char *ftoa(char *a, double f, int precision) {
    long p[] = {0,10,100,1000,10000,100000,1000000,10000000,100000000};

    char *ret = a;
    long heiltal = (long)f;
    itoa(heiltal, a, 10);
    while (*a != '\0') a++;
        *a++ = '.';
    long desimal = abs((long)((f - heiltal) * p[precision]));
    itoa(desimal, a, 10);
    return ret;
}
```

6.6 Sources

Les sources sont disponibles entre autres sur le dépôt du projet Domogik dans la branche **default** au chemin **src/external/hardwares/ar_rgb**. Le dépôt de Domogik est accessible à l'URL http://hg.domogik.org.

7 Passerelle Xbee > UDP

Il existe un plugin de Domogik faisant office de passerelle liaison série > UDP (à ce jour, il ne permet le lien que dans ce sens, mais il sera rendu bidirectionnel par la suite). Il permet d'écouter N liaisons séries avec différentes vitesses de connexion.

Le fonctionnement de ce plugin est relativement simple. Voyons ce qu'il fait pour chaque liaison paramétrée (en zappant le code spécifique à Domogik qui ne nous intéresse pas ici).

7.1 En premier, le matériel !

Voyons d'abord comment créer une règle **udev** afin d'avoir toujours le même chemin (et les bons droits !) vers l'interface USB/xbee.

Lorsque je la branche la première fois, elle a pour chemin **/dev/ttyUSB0**. Récupérons le chemin USB associé :

```
# udevadm info -q path -n /dev/ttyUSB0
/devices/pci0000:00/0000:00:1d.1/usb7/7-1/7-1:1.0/ttyUSB0/tty/ttyUSB0
```

et utilisons cette adresse pour lister les informations **udev** liées :

```
# udevadm info -a -p $(udevadm info -q path -n /dev/ttyUSB0)
[...]
  looking at device '/devices/pci0000:00/0000:00:1d.1/usb7/7-1/7-1:1.0/ttyUSB0/
tty/ttyUSB0':
    KERNEL=="ttyUSB0"
    SUBSYSTEM=="tty"
    DRIVER==""

  looking at parent device '/devices/pci0000:00/0000:00:1d.1/usb7/7-1/7-1:1.0/
ttyUSB0':
    KERNELS=="ttyUSB0"
    SUBSYSTEMS=="usb-serial"
    DRIVERS=="ftdi_sio"
[...]

  looking at parent device '/devices/pci0000:00/0000:00:1d.1/usb7/7-1/7-1:1.0':
[...]

  looking at parent device '/devices/pci0000:00/0000:00:1d.1/usb7/7-1':
    KERNELS=="7-1"
    SUBSYSTEMS=="usb"
    DRIVERS=="usb"
[...]
    ATTRS{idVendor}=="0403"
    ATTRS{idProduct}=="6001"
[...]
    ATTRS{serial}=="A40081oe"
[...]
```

Habituellement, pour rédiger une règle udev pour du matériel, je me base sur **idVendor** et **idProduct** qui donnent un couple unique pour un type de matériel. Toutefois, ici, ce couple correspond à la puce FTDI : « FTDI FT232 USB-Serial ». Par contre, ici, **serial** permet de différencier le montage qui est derrière la puce FTDI. Nous allons donc créer la règle suivante qui sera dans le fichier **/etc/udev/rules.d/xbee.rules** :

```
SUBSYSTEM=="tty" ATTRS{idVendor}=="0403" ATTRS{idProduct}=="6001"
ATTRS{serial}=="A40081oe" SYMLINK+="xbee-explorer" MODE="0666"
```

En débranchant/rebranchant le câble USB, l'interface est maintenant accessible en tant que **/dev/xbee-explorer** avec les accès en lecture/écriture pour tous.

7.2 Ensuite, le code

7.2.1 Expressions régulières

Deux expressions régulières sont définies afin de repérer les messages xPL :

```
REGEXP_TYPE = r"""(?P<type_>xpl-cmnd | xpl-trig | xpl-stat)
REGEXP_GLOBAL = r"""(?P<type_>.*)\n
                    \{\n
                    hop=(?P<hop_count>.*)\n
                    source=(?P<source>.*)\n
                    target=(?P<target>.*)\n
                    \}\n
                    (?P<schema>.*)\n
                    \{\n
                    (?P<data>.*)\n
                    \}
                """
```

Elles seront compilées :

```
__regexp_type = re.compile(REGEXP_TYPE, re.UNICODE | re.VERBOSE)
__regexp_global = re.compile(REGEXP_GLOBAL, re.DOTALL | re.UNICODE |
re.VERBOSE)
```

7.2.2 Ouverture de la liaison série

Une première fonction permet de tenter l'ouverture du port série :

```
def open(self, device, baudrate):
    """ Ouverture de la liaison série

    @param device : chemin vers le device
    @param baudrate : vitesse de la liaison (9600, ...)
    """
    try:
        self._ser = serial.Serial(device, baudrate)
    except:
        error = "Error while opening serial device : %s : %s" %
        (device, str(traceback.format_exc()))
        raise Xpl1BridgeException(error)
```

Ici, rien de complexe : on ouvre le device à la vitesse de connexion paramétrée.

7.2.3 Écoute du device

Une fois le device ouvert, une seconde fonction est appelée :

```
def listen(self):
    """ Ecoute la liaison série pour des messages xPL
    """
    current_msg = ""

    while not self._stop.isSet():
        resp = self._ser.readline()
        // début d'un message xPL détecté
        if self.__regexp_type.match(resp):
            print("New start of xpl message detected")
            current_msg = resp
        else:
            current_msg += resp
            // le message xPL est complet
            if self.__regexp_global.match(current_msg):
                self._cb(current_msg)
                current_msg = ""
```

Une boucle qui s'arrête en même temps que le plugin va lire en continu la liaison série. Si un début de message xPL est repéré (via **REGEXP_TYPE**), la variable **current_msg** est initialisée avec le début du message. Elle sera ensuite complétée à chaque nouvelle donnée jusqu'à ce qu'elle contienne un message xPL valide et complet (avec **REGEXP_GLOBAL**). Si entre temps un nouveau début de message xPL est repéré, le message qui était en cours est perdu (on considère qu'il y a eu un souci d'émission ou de réception).

7.2.4 Envoi du message xPL sur le LAN

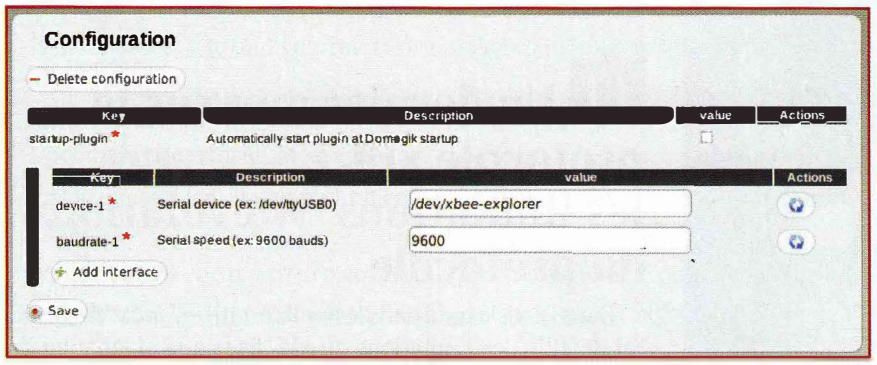
Lorsqu'un message xPL complet est reçu, une fonction de callback (**self._cb**) est appelée depuis la fonction **listen**. Il s'agit en fait de la fonction suivante :

```
    def send_xpl(self, resp):
        """ Envoi d'un message xPL sur le réseau
            @param resp : message xPL
        """
        try:
            msg = XplMessage(resp)
            self.myxpl.send(msg)
        except XplMessageError:
            error = "Bad data : %s" % traceback.format_exc()
            print(error)
```

Cette fonction fait appel à une fonction dont héritent tous les plugins de Domogik : **self.myxpl.send**. Cette fonction va émettre le message en UDP.

7.3 Configuration du plugin

Le plugin se configure très simplement depuis l'interface d'administration. Voici comment le paramétrer :



Configuration du plugin

7.4 Simple, non ?

Cette passerelle n'a finalement rien de complexe et vous pourrez facilement en créer une pour votre propre installation dans le langage de votre choix !

8 Intégration dans Domogik

Je ne vais pas détailler précisément comment intégrer un montage Arduino dans Domogik, tout étant décrit sur le wiki, je vais me contenter de vous donner les étapes et les liens ;)

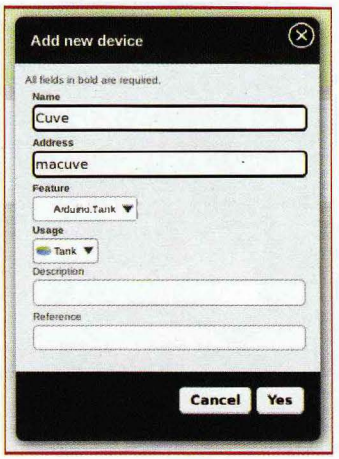
8.1 Fichiers xml

Tout d'abord, il faut créer un fichier qui va décrire les fonctionnalités du hardware (distance et hauteur en pourcentage dans notre cas) puis charger les données du fichier en base **[DMG - xml plugin]**. Le chemin du fichier dans les sources de Domogik est : **src/share/domogik/hardwares/ar_tank.xml**.

Ensuite, vous pourrez créer un fichier qui décrira les données xPL à écouter et comment les insérer en base **[DMG - stat file]**. Son chemin dans les sources est : **src/share/domogik/stats/arduino/sensor.basic-ar_tank.xml**.

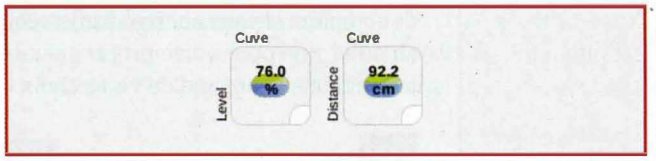
8.2 Création du device et visualisation

Ensuite, il ne reste qu'à créer un device ainsi pour obtenir le niveau en pourcentage :



Création du device

Placez ensuite les widgets correspondant aux deux fonctionnalités dans l'IHM et vous obtenez ceci :



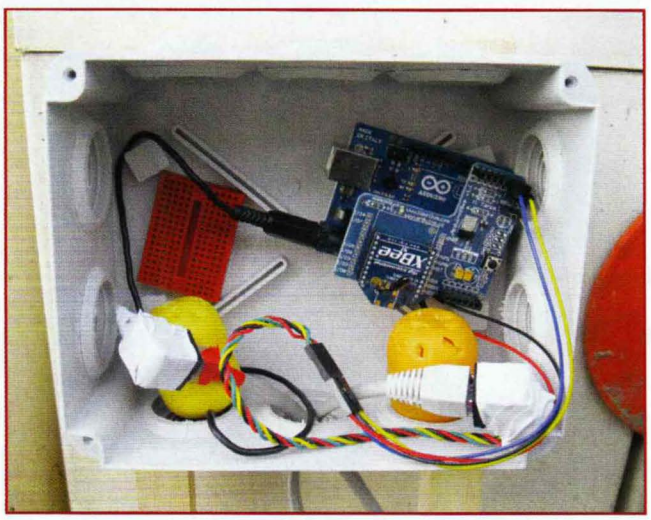
Les widgets

En cliquant sur un des widgets, vous obtenez l'historique de la fonctionnalité. Ici, le niveau d'eau en pourcentage :

Graphique

9 Mise en place

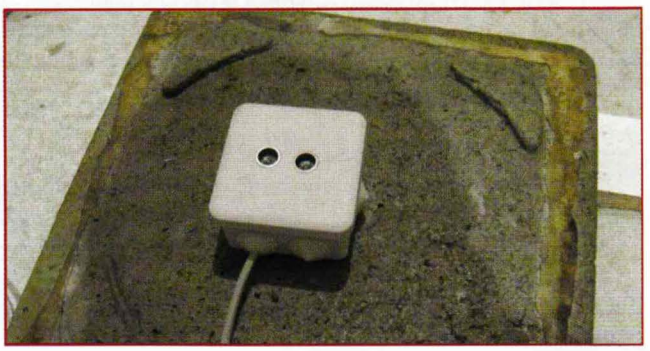
Pour finir, voici quelques photos de la mise en place du capteur. Tout d'abord, l'Arduino dans sa boîte étanche dehors. Notez les deux boîtes de Kinder. Elles sont remplies de riz afin d'absorber l'humidité. L'avenir me dira si c'est réellement efficace...



L'Arduino dans sa boîte

Voici l'intégration du capteur à ultrasons dans un boîtier étanche :

Intégration dans le boîtier



Le boîtier fermé

10 Configuration des puces Xbee

Par défaut, toutes les puces Xbee sont paramétrées de la même manière, ce qui leur permet de communiquer entre elles dès leur première utilisation. Toutefois, pour ne pas interférer avec d'autres projets, il peut être intéressant de les reconfigurer. Nous verrons ce point dans un autre article.

11 Un dernier mot sur le protocole xPL : les limitations^W évolutions du protocole

Dans l'article paru dans le numéro 140 de *Linux Magazine*, j'ai détaillé des limitations du xPL liées à la stagnation du protocole. Depuis, la situation a changé et c'est tant mieux !

Pour rappel, voici quelques limitations que je jugeais bloquantes pour la pérennité du xPL et ce qui a été mis en place par l'équipe du projet xPL :

11.1 Un message est limité à 1500 caractères

Un nouveau schéma, **FRAGMENT.BASIC** **[xpl fragment]** (et son cousin, **FRAGMENT.REQUEST**), a été spécifié : si un message qui doit être émis fait plus de 1500 caractères, il sera divisé en plusieurs schémas **FRAGMENT.BASIC**. Le client xPL qui recevra des messages **fragment.basic** attendra d'avoir récupéré tous les fragments puis reconstruira le message xPL d'origine avant de le traiter. Si un fragment n'arrive pas, il est possible de le redemander grâce au message **FRAGMENT.REQUEST**.

Cette manière de faire permet de rendre cette évolution transparente pour tous les clients xPL déjà existants.

Domogik gère d'ores et déjà ce nouveau schéma.

11.2 Une valeur n'est plus limitée à 128 caractères

Auparavant, dans le couple « clé=valeur » d'un message xPL, la longueur de la valeur était limitée à 128 caractères. Cette limite a été levée.

11.3 Le jeu de caractères utilisables est limité

Aucun caractère accentué (entre autres) n'était autorisé dans les valeurs des couples clé/valeur d'un schéma xPL. L'UTF8 est maintenant toléré **[xpl utf8]**.

Conclusion

Il est donc finalement relativement simple de créer ses propres capteurs avec émission des valeurs via Xbee et le fait de ne pas avoir de réel lien IP n'empêche pas pour autant d'en faire des capteurs xPL à partir du moment où une passerelle existe. Il ne vous reste maintenant qu'à créer les capteurs qu'il vous manque ;) ■

Références

[Battery] (en) : http://interface.khm.de/index.php/lab/experiments/sleep_watchdog_battery/

[Domogik] : http://www.domogik.org

[DMG – xml plugin] : http://wiki.domogik.org/PluginXmlDescriptionFile

[DMG – stat plugin] : http://wiki.domogik.org/REST_statmgr

[sensor.basic] : http://xplproject.org.uk/wiki/index.php?title=Schema_-_SENSOR.BASIC

[proto xPL] : http://xplproject.org.uk/wiki/index.php?title=XPL_Specification_Document#Heartbeat_Messages

[xpl fragment] (en) : http://xplproject.org.uk/wiki/index.php?title=Schema_-_FRAGMENT

[xpl utf8] (en) : http://xplproject.org.uk/forums/viewtopic.php?f=2&t=1094

[HC-SR04] (en) : http://avrproject.ru/sonar_hc_sr04/HC-SR04.pdf

[Shield Xbee] (en) : http://www.arduino.cc/en/Main/ArduinoXbeeShield

NETCAT : COUTEAU SUISSE POUR LE RÉSEAU

par Denis Bodor

Ne vous est-il jamais arrivé de devoir diagnostiquer des problèmes réseau ou des connexions à des serveurs/services avec comme seules alternatives soit d'utiliser ou tester les outils initialement prévus par l'applicatif (navigateur, client mail, code ou client dédié), soit de, passez-moi l'expression, littéralement galérer avec ping, Telnet, netstat et quelques autres utilitaires basiques ? Pour avoir goûté à ce genre de réjouissances plus d'une fois, j'ai appris à toujours installer un outil magique dans tous mes systèmes embarqués : Netcat.

Il est à la fois facile et délicat de décrire précisément Netcat. Il faut savoir, en premier lieu, qu'il n'existe pas un seul Netcat, mais plusieurs dont la plupart sont des implémentations open source et en logiciel libre. Les trois plus importantes versions sont GNU Netcat, OpenBSD Netcat et un Netcat pour Windows auquel il faut ajouter les portages des deux précédents sur ce système. Notez que BusyBox intègre également une version légère de la commande **nc**.

Ensuite, il faut qualifier Netcat : un utilitaire à tout faire dialoguant sur le réseau. Il est en effet capable de :

- Initier ou attendre des connexions sur n'importe quel port TCP ou UDP.
- Gérer la résolution des noms d'hôtes.
- Permettre l'utilisation de n'importe quel port local.
- Permettre l'utilisation de n'importe quelle adresse locale.
- Procéder à des scans de ports.
- Utiliser STDIN et STDOUT (entrée et sortie standard).
- Permettre de régler la vitesse de transmission des données.
- Reformater les données affichées (hexdump, par exemple).
- Remplacer Telnet.
- Mettre en œuvre des fonctionnalité de *tunneling*.

S'il fallait résumer ce qu'il est possible de faire avec Netcat, cela se tiendrait à la décomposition de son nom : un **cat** pour le réseau. Un outil dans la plus parfaite lignée des commandes traditionnelles d'UNIX. Ma version préférée est sans l'ombre d'un doute celle d'OpenBSD (installable sous Debian et dérivés, comme Ubuntu, sous le nom **netcat-openbsd**).

1 Utilisons Netcat

Netcat est tout d'abord un client réseau, et à ce titre, il permet de faire exactement ce que permet **telnet** et, en premier lieu, le test de certains services ayant un protocole « orienté texte » comme POP3, IMAP, SMTP, HTTP, etc. :

```
% nc  192.168.122.12 110
+OK Qpopper (version 4.0.9) at ks.kimsufi.com starting.
USER nomutilisateur
+OK Password required for nomutilisateur.
PASS mot2passe
+OK nomutilisateur has 0 visible messages (0 hidden).
QUIT
+OK Pop server at ks.kimsufi.com signing off.
```

La syntaxe utilisée consiste simplement à spécifier l'hôte de destination et le port. Par défaut, la connexion se fait en TCP, il n'y a rien à préciser de plus. En l'absence de tout autre outil, Netcat permet également d'établir une connexion entre deux machines :

Machine A :

```
% nc -lv 4242
```

L'option **-l** signifie *listen* et permet d'attendre les connexions sur le port (TCP par défaut) précisé. L'option **-v** est ici ajoutée afin de rendre Netcat plus bavard et d'avoir une idée de ce qui se passe.

Machine B :

```
$ nc -v 192.168.10.166 4242
Connection to 192.168.10.166 4242 port [tcp/*] succeeded!
```

Nous connectons la machine A comme nous l'avons fait avec notre serveur POP3 précédemment, en spécifiant cette fois le port 4242.

Machine A :

```
Connection from 192.168.10.144 port 4242 [tcp/*] accepted
coucou
```

La machine A signifie la connexion depuis la machine B et nous saisissons et validons un simple mot.

Machine B :

```
coucou
```

Immédiatement le texte saisi apparaît sur l'écran de la machine B.

Qui dit « communication » dit forcément échange et donc « possibilité de transférer des fichiers ». Nous n'avons cependant pas à nous amuser à encoder/décoder des données binaires car

nous pouvons reposer sur les fonctions de gestion STDIN/STDOUT de Netcat :

Machine A :

```
$ nc -l 4242 > image.iso

$ md5sum image.iso
734795a8542b161296a1e02e5bdda083  image.iso
```

Machine B :

```
% md5sum u1015powerpc.iso
734795a8542b161296a1e02e5bdda083  u1015powerpc.iso

% cat u1015powerpc.iso | \
nc 192.168.10.144 4242
```

Comme avec n'importe quelle commande (ou presque), nous pouvons utiliser le *pipe* (**|**) et le **>** pour rediriger les entrées et les sorties standards d'un programme. La machine A reçoit ici un fichier et se met donc en attente en redirigeant sa sortie vers un fichier **image.iso**. La machine C, quant à elle, envoie le fichier en utilisant **cat** pour copier le contenu sur la sortie standard que nous redirigeons vers le Netcat initiant la connexion. Les commandes **md5sum** sont là uniquement pour démontrer/vérifier que le transfert s'est bien passé. Les sommes de contrôle sont identiques.

En utilisant le petit outil **pv** permettant de monitorer le trafic dans un flux, nous pouvons même obtenir une progression du transfert en live ainsi que le débit effectif :

```
$ nc -l 4242 | pv -br > image.iso
81,4MO [12,2MB/s]
```

Notez que la machine qui envoie n'est pas nécessairement celle qui initie la connexion. Nous pouvons également faire :

Machine A :

```
cat image.iso | pv -rb | nc -l 4242
418MO [11,2MB/s]
```

Machine B :

```
% nc 192.168.10.144 4242 | pv -rb > iii.iso
493MO [11,2MB/s]
```

L'utilisation de redirection permet toutes sortes de choses. Vous pouvez, par exemple, compresser à la volée les données en ajoutant **| gzip |** :

Machine A :

```
$ nc -l 4242 | pv -rb | gunzip > image.iso
```

Machine B :

```
% cat u1015powerpc.iso | gzip | pv -rb | \
nc 192.168.10.144 4242
```

La source de données peut également être librement choisie. Vous pouvez ainsi sauvegarder très simplement une image disque complète d'un système vers un autre en utilisant **dd if=/dev/sda5 | gzip -9 | nc 192.168.10.144 4242** (le **-9** de **gzip** indique le niveau de compression exigé). Dans le même genre d'idées, il est possible d'ajouter l'option **-k** à la commande **nc** sur la machine en attente de connexion. On arrive ainsi à faire des choses comme :

Machine A :

```
$ nc -kl 4242 | pv -rb > fichier
```

Machine B :

```
% echo "coucou" | pv -rb | nc 192.168.10.144 4242
   7O [ 402kB/s]
% echo "plopinou" | pv -rb | nc 192.168.10.144 4242
   9O [ 517kB/s]
% echo "tralalalala" | pv -rb | nc 192.168.10.144 4242
  12O [ 651kB/s]
```

Machine A :

```
  28O [ 952mB/s]
^C

$ cat fichier
coucou
plopinou
tralalalala
```

-k, en conjonction avec **-l**, oblige Netcat à continuer d'attendre des connexions même si un marqueur de fin de fichier (OEF) est reçu. Ce marqueur envoyé, comme son nom l'indique, en fin de fichier ou via un **^D**, coupe la connexion, et généralement, Netcat se termine à cet instant. Grâce à **-k**, nous pouvons laisser un serveur en attente stockant, par exemple, tout et n'importe quoi provenant de multiples connexions, dans un seul fichier.

Une autre fonctionnalité intéressante est le scan de ports :

```
$ nc -vnzw 2 192.168.10.166 20-25
nc: connect to 192.168.10.166 port 20 (tcp) failed:
Connection refused
nc: connect to 192.168.10.166 port 21 (tcp) failed:
Connection refused
Connection to 192.168.10.166 22 port [tcp/*] succeeded!
nc: connect to 192.168.10.166 port 23 (tcp) failed:
Connection refused
nc: connect to 192.168.10.166 port 24 (tcp) failed:
Connection refused
nc: connect to 192.168.10.166 port 25 (tcp) failed:
Connection refused
```

N'oubliez pas de préciser l'option **-v**, car dans le cas contraire, **nc** restera muet comme une carpe. **-n** permet d'utiliser le mode numérique (sans résolution DNS) et **-w** indique un délai d'attente qui terminera le programme si rien ne circule sur STDIN ou la connexion au-delà du nombre de secondes spécifié. Enfin, c'est l'option **-z** qui provoque le scan. La plage de ports a tester est donnée en fin de ligne.

Certaines versions de Netcat, comme celle empaquetée sous le nom **netcat-traditional** pour Ubuntu/Debian, intègre des options différentes. Il en est une que la version OpenBSD semble ne pas intégrer, sans doute pour des raisons évidentes de sécurité : l'option **-e**, qui est aussi puissante que dangereuse. Voici ce qu'elle permet de faire :

Machine A :

```
% nc -vlp 4242 -e /bin/bash
listening on [any] 4242 ...
```

Machine B :

```
$ nc -v 192.168.10.166 4242
Connection to 192.168.10.166 4242 port [tcp/*]
succeeded!
echo $SHELL
/bin/bash
whoami
denis
uname -srv
Linux 2.6.32-5-686-bigmem #1 SMP Tue Mar 8
22:14:55 UTC 2011
exit
```

Cette option magique permet de transformer n'importe quel programme utilisant STDIN/STDOUT en serveur. Netcat, via **-e**, va lancer le binaire spécifié et rediriger les entrées/sorties. Le Netcat distant (ici celui de la machine B) va alors donner à l'utilisateur un accès direct au programme s'exécutant à l'autre bout de la connexion.

Comme vous le voyez, Netcat, que vous travailliez dans un environnement embarqué ou non, est en mesure de vous sortir de bien des situations. Je vous recommande donc fortement, pour quelques dizaines de kilo-octets, de systématiquement l'inclure dans vos systèmes... juste au cas où... ■

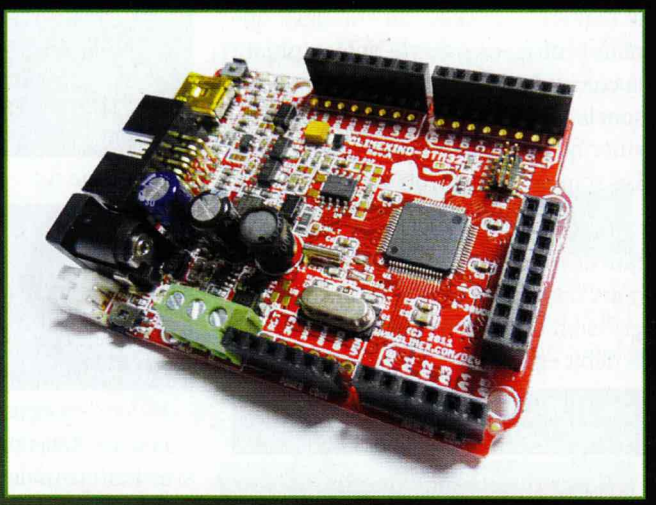
DANS LE PROCHAIN NUMÉRO*

Par expérience, nous avons appris à la rédaction qu'il n'est pas très malin de vendre la peau de l'ours avant de l'avoir tué. En d'autres termes, le fait d'annoncer un sommaire du prochain numéro d'un magazine avant d'avoir la majorité des articles sous les yeux est souvent considéré, par les dieux mystiques et sans pitié de la presse, comme un acte présomptueux impliquant généralement des conséquences pour tout ou partie de l'équipe rédactionnelle. Néanmoins, et c'est dire comme nous sommes courageux dans notre quête sans fin de vous satisfaire, voici sans doute ce que vous risquez de trouver dans le prochain numéro d'Open Silicium 6 disponible le 31 mars prochain (oui, ça fait loin, je sais)...

Des capteurs de gaz butane, propane et CO2

Plein de choses autour du RFID et du NFC

Une découverte d'un oscilloscope portable chinois économique



Quelques réjouissances autour d'un STM32

De l'espionnage de bus USB

* Sous réserve de toutes modifications, de combustion spontanée du matériel utilisé/testé, de tuile de dernière minute remettant en cause un article complet parce qu'une phrase d'une datasheet a été mal interprétée ou de la destruction du laboratoire souterrain secret d'Open Silicium suite à l'intervention de Metro Man. ■

EN KIOSQUE LE 31 MARS 2012

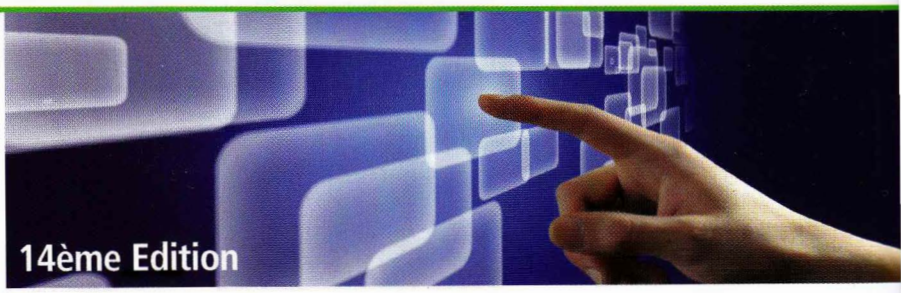
EXPOSITION – CONFÉRENCES – ATELIERS

rts Le salon des solutions informatiques temps réel et des systèmes embarqués
The real-time solutions and embedded systems show

EMBEDDED SYSTEMS

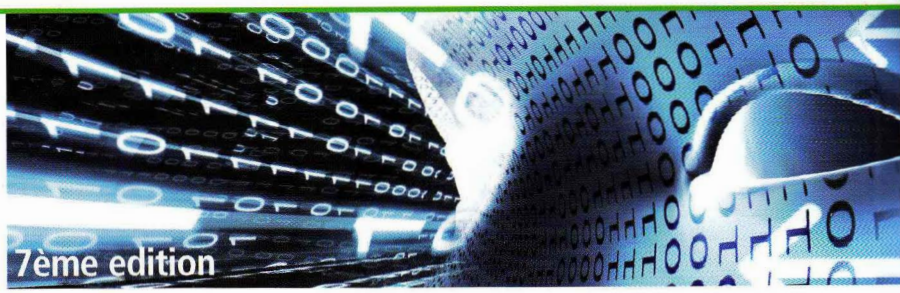
SOLUTIONS INFORMATIQUES TEMPS RÉEL ET SYSTÈMES EMBARQUÉS

display



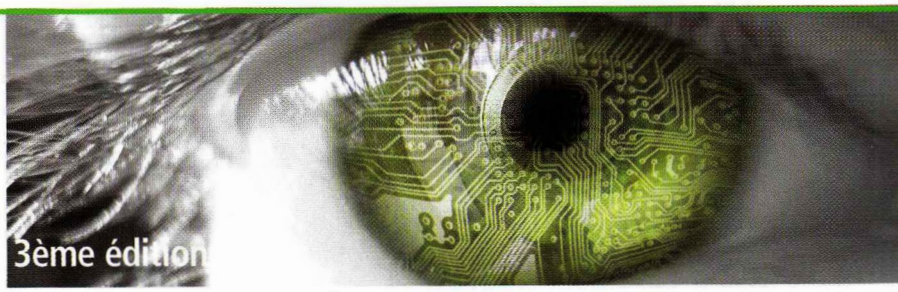
SOLUTIONS D'AFFICHAGE ET DE VISUALISATION ÉLECTRONIQUES

Machine to Machine



SOLUTIONS MTOM

ESDT
ELECTRONIC SYSTEM DESIGN & TESTING



SOLUTIONS EN CONCEPTION ET TEST ÉLECTRONIQUES

Les Salons Solutions ELECTRONIQUES

Les 3, 4 & 5 Avril 2012

Paris - Porte de Versailles

Pavillon 7.1

www.salons-solutions-electroniques.com