AOÛT/SEPTEMBRE 2009 N°43

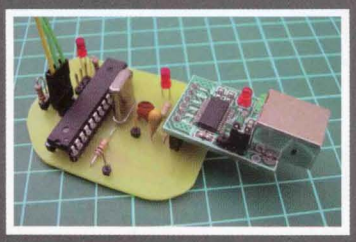
GNU LINUX MAGAZINE / FRANCE

HORS-SÉRIE

Administration et développement sur systèmes UNIX

MICROCONTRÔLEURS

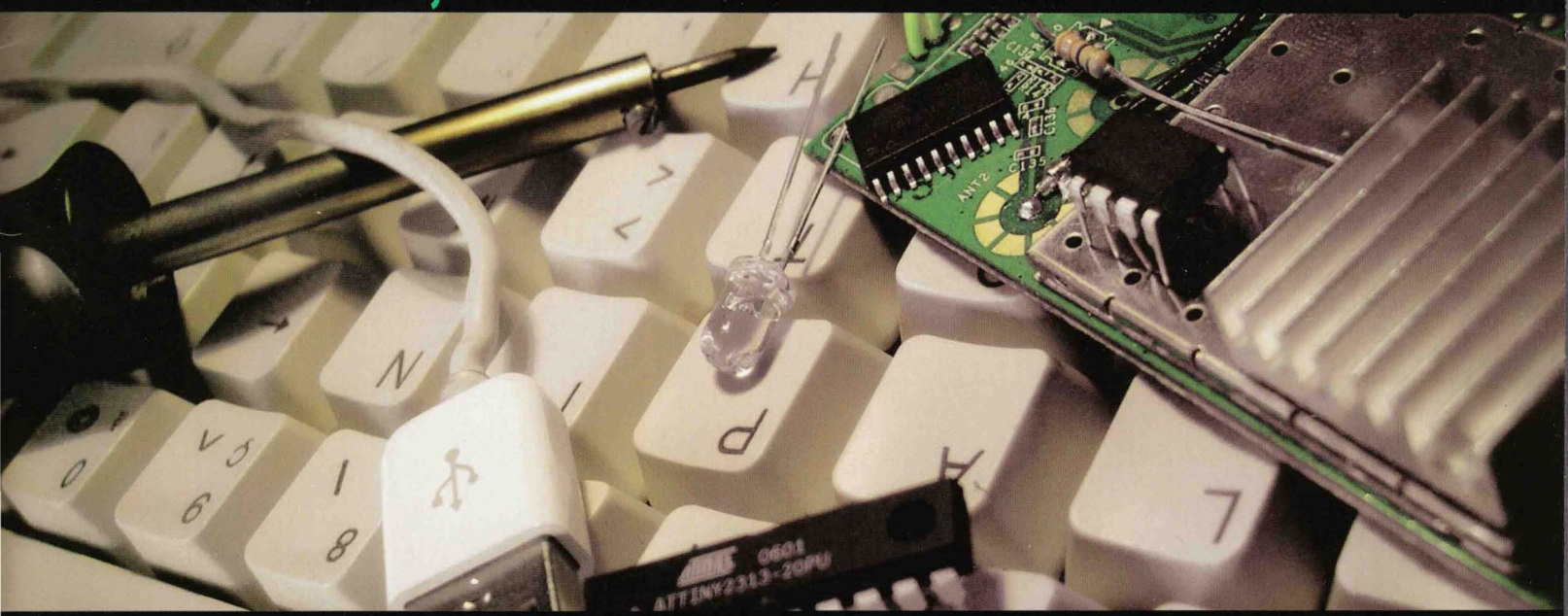
Apprenez à développer en C sur microcontrôleur Atmel AVR (p. 08)



PRÊT À REPOUSSER LES FRONTIÈRES DE VOTRE SYSTÈME ?

ÉLECTRONIQUE, EMBARQUÉ ET HACKS

INTRODUCTION, EXPÉRIMENTATIONS ET MISES EN ŒUVRE



SMARTPHONE

Découvrez les firmwares OE/SHR, FSO et Hackable:1 pour FreeRunner et développez vos applications (p. 46)

PLAYSTATION 3

Installez GNU/Linux et accédez à la puissance de calcul du processeur Cell BE en C (p. 74)

NINTENDO DS

Transformez votre DS en plateforme pour l'embarqué sous GNU/Linux ou RTEMS (p. 52)

L 15066 - 43 H - F: 6,50 € - RD

France Métro : 6,50 € - DOM : 7,00 €
TOM Surface : 950 XPF
POL. A : 1400 XPF
BEL/PORT.CONT : 7,50 €
CH : 13,8 CHF - CAN : 13 $CAD
TUNISIE : 8,80 TND
MAR : 75 MAD

Comment stocker, de manière sécurisée, des données sur un espace disque se trouvant hors de votre machine ?

GNU/LINUX MAGAZINE JUILLET / AOÛT

UTILISEZ GFS NBD OVER SSH POUR VOTRE STOCKAGE RÉSEAU

AU SOMMAIRE :

NEWS

p. 04 Nordic Perl Workshop 2009 / Go Open 2009

p. 12 The Go Open Expert Café

KERNEL

p. 18 Nouveautés du noyau 2.6.30

LIVRE(S)

p. 25 Critique : « Algorithmique Raisonner pour concevoir »

SYSADMIN

p. 26 Confinement de processus sous FreeBSD (Jail) et Linux (OpenVZ)

p. 35 I can haz IPMI ? Ou comment gérer l'aspect matériel de ses serveurs

p. 38 Installation sur clef USB

p. 42 Gestion de sources distribuée avec Mercurial

p. 48 Gestion de déploiement centralisée

NETADMIN

p. 56 Stockage en ligne : NBD dans un tunnel SSH. L'objectif de cet article est de vous présenter comment stocker, de manière sécurisée, des données sur un espace disque se trouvant hors de votre machine, quelque part sur le réseau et accessible de n'importe où. Le tout au niveau « block device » et en utilisant SSH.

p. 62 OpenVPN à travers un proxy restrictif

p. 70 À la découverte de LemonLDAP::NG

REPÈRES

p. 76 Processus de publication chez Debian et Ubuntu

p. 80 Parce qu'y'en a marre

CODE(S)

p. 82 NetBSD a enfin son APT : un pkgin sans glace pour la 6

p. 94 Petite histoire du portage d'un logiciel libre sur HP-UX

DISPONIBLE

JUSQU'AU 28 AOÛT 2009

chez votre marchand de journaux

www.ed-diamond.com

Sommaire

News

p. 04 **Pas Sage en Seine 2009**

Introduction

p. 06 **Bien débuter en électronique**

p. 08 **Le monde des microcontrôleurs AVR 8 bits**

Montages

p. 18 **Un éthylomètre pour s'authentifier sur le système : le montage**

Embarqué

p. 24 **Développement embarqué sur GTA01 et GTA02**

p. 46 **Installez et utilisez la plateforme de développement Maemo**

p. 52 **Interfaces matérielles et OS libres pour Nintendo DS : DSLinux et RTEMS**

Plateforme

p. 74 **Initiation au calcul intensif sur PS3**

Abonnement

p. 34, 55, 56 **Bons d'abonnement et de commande**

Édito

Distribution, grosses applications, framework, environnement...

Ne vous est-il jamais arrivé de vouloir en revenir au bon vieux temps de l'informatique où l'on pouvait encore envisager un système dans son ensemble, en comprendre le moindre recoin et sentir une vraie interaction avec le matériel ? Vous souvenez-vous de cette bonne vielle époque où programmer avait encore un véritable rapport avec les électrons qui circulaient dans les circuits des machines ? Cette époque est-elle révolue ?

Pas du tout ! Le monde de l'informatique moderne n'est qu'illusion, les bureaux 3D qu'une façade et les couches d'abstraction de simples écrans de fumée pour ceux qui ne préfèrent pas savoir. Programmer et utiliser un ordinateur est une véritable relation homme/machine, pas un « boulot », pas un « gagne-pain ». Cette relation est basée, plus que sur toute autre chose, sur la curiosité et la soif de connaissances. Cela nous pousse à comprendre ce à quoi nous avons affaire, à en chercher les limites, puis à les repousser.

Comment installer GNU/Linux sur une console de jeu bi-écran ? Comment accéder à la puissance des processeurs vectoriels d'une PS3 ? Comment créer mes propres périphériques ? Comment développer des applications pour mon téléphone ? Autant de questions que se sont posés les auteurs des articles qui vont suivre et auxquelles ils répondent.

Des petits composants, comme les microcontrôleurs Atmel AVR, aux consoles de jeu, chaque plateforme, chaque environnement et chaque système deviennent un terrain de jeu et un univers à explorer. L'esprit de l'informatique homebrew n'est pas mort, loin de là. Il suffit de creuser un peu et de chercher les réponses. Ces terrains de jeu, il y en a légion. Plus que ne pourrait en contenir un simple hors-série comme celui-ci. Bon nombre de sujets n'ont pas trouvé leur place faute de pages, mais ce n'est que partie remise. Plus important encore, au cœur même des articles, se bousculant derrière cette page, se trouvent des sujets et des domaines à explorer... Si vous lisez ce texte, il est probablement déjà trop tard.

Rangez vos tongs et sortez fer à souder, tournevis et pinces. C'est l'été ! Eh non, ce n'est pas une très bonne idée que de tenter de dérober cette DS-lite à ce gamin qui joue sur la place. Prêtez-lui votre GLMF et, peut-être, comprendra-t-il qu'il y a plus ludique que de jouer à Pokemon version Platine ;)

Denis Bodor

Gnu /Linux Magazine France Hors-série

LES ÉDITIONS DIAMOND est édité par Diamond Editions
B.P. 20142 - 67603 Sélestat Cedex
Contactez-nous via notre site internet
http://www.ed-diamond.com

E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor

Secrétaire de rédaction : Véronique Wilhelm
Relecture : Dominique Grosse
Conception graphique : Fabrice Krachenfels
Responsable publicité : pub@ed-diamond.com
Service abonnement : cial@ed-diamond.com
Impression : VPM Druck 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 61 72 76 24
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution /N° ISSN : 1291-78 34
Commission paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 6,50 €

Membre April promouvoir et défendre le logiciel libre www.april.org

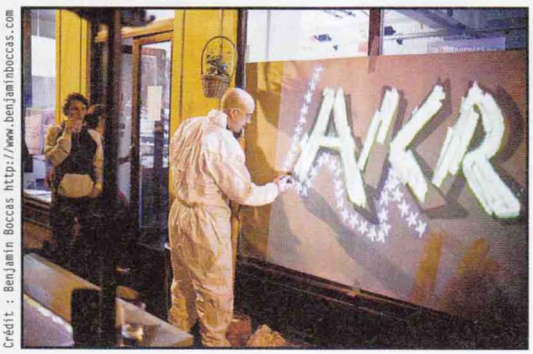
La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Linux Magazine France est interdite sans accord écrit de la société Diamond Editions. Sauf accord particulier, les manuscrits,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

Pas Sage en Seine 2009

Auteur

- Olivier Laurelli

La première édition de *Pas Sage en Seine* s'est tenue les 4 et 5 juin à Paris IIe, dans la Galerie des panoramas et avec l'aimable participation des commerçants de la galerie.



Crédit : Benjamin Boccas http://www.benjaminboccas.com

Se tenant un peu en marge de la première de *Futur en Seine*, *Pas Sage en Seine* est une manifestation coorganisée au pied levé en moins de deux semaines par l'Association Silicon Sentier, La Cantine, Bearstech, Toonux, Mozilla, hackable:device, Cap Digital et la Région Île de France.

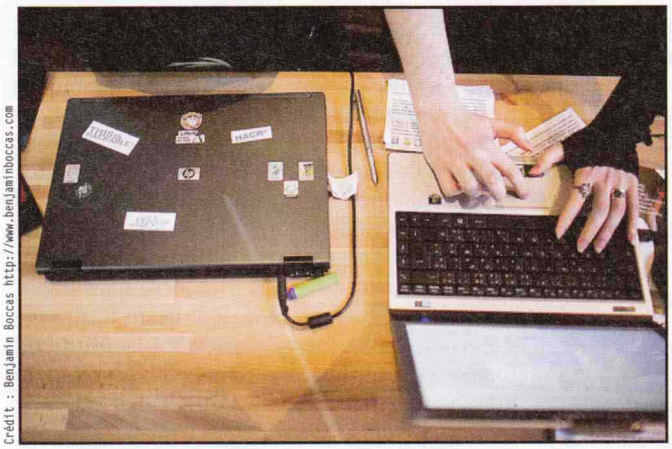
Cet événement avait pour but de faire se rencontrer des hackers, des artistes aux créations libres et monsieur ou madame tout le monde (commerçants, passants, *geeks*, moins geeks, curieux...). *Pas Sage en Seine* revendique une dimension pédagogique : c'est un observatoire du hack expliqué au grand public où se mêlent sécurité, bidouille matérielle, partage, hacktivisme et cultures underground alternatives. Il s'agit avant tout d'exposer et d'expliquer les motivations des hacktivistes et des hackers de tous poils, pour enfin comprendre leur démarche et permettre au plus grand nombre d'en tirer bénéfice.

Les participants aux ateliers se sont prêtés au jeu des questions réponses sur de nombreux sujets abordés par les visiteurs : risques liés au *phishing*, aux réseaux wifi, à l'absurdité de l'HADOPI ou du filtrage du web, aux alternatives culturelles libres (Art Libre, Creative Commons...). La Quadrature du Net a été maintes fois saluée par les visiteurs pour ses actions sur le projet de loi Création et Internet, qui a occupé de nombreux débats pendant *Pas Sage en Seine*.

Crédit : Benjamin Boccas http://www.benjaminboccas.com

Pas Sage en Seine se tenait dans une galerie commerçante parisienne où a été déployé pour l'occasion un réseau wifi Open Mesh, une solution libre pour créer des réseaux maillés. Ce réseau wifi servait de plate-forme d'expérimentation et de démonstration d'interceptions de données (données restituées sur écran géant, un *Wall of Shame*), de l'ARP *spoofing* ou du *man-in-the-middle* pour voler des sessions SSL de Gmail. Ces petits ateliers se tenaient de manière plus ou moins formelle au beau milieu de la galerie, aux yeux de tous et toujours dans la bonne humeur.

Crédit : Benjamin Boccas http://www.benjaminboccas.com


Crédit : Benjamin Boccas http://www.benjaminboccas.com

Côté hardware, il y avait de quoi bidouiller. En présentant du matériel libre comme le Neo Freerunner d'Openmoko sous hackable:1 (une distribution embarquée basée sur Debian GNU/Linux) et en expliquant l'intérêt, dans un premier temps pour le geek et après pour tous, on éveille vite la curiosité et l'intérêt des passants. Ce fut d'ailleurs l'occasion pour certains de découvrir qu'ils avaient un Linux dans la poche depuis plusieurs mois (GNU Linux est très largement adopté par les constructeurs de téléphones mobiles).

Crédit : Benjamin Boccas http://www.benjaminboccas.com

Toujours côté Hardware : des Fonera, beaucoup de matériel wifi, deux énormes machines qui écoutaient le réseau wifi maillé ont également été martyrisés.

Crédit : Benjamin Boccas http://www.benjaminboccas.com

Le 5 juin au soir, l'évènement s'achevait sur une présentation par Alexandre Korber de **/tmp/lab**, d'une machine RepRap, machine de prototypage rapide, peu onéreuse et auto-réplicante.

Cette première de *Pas Sage en Seine* a vu défiler beaucoup de curieux aux compétences et aux niveaux très hétérogènes, ce qui a permis de ne pas se cantonner à un cadre strictement technique. Les échanges se sont naturellement ouverts à l'art et aux logiciels libres, à l'importance de l'hacktivisme à l'heure de l'HADOPI et de la LOPPSI. Une opération de rencontre et de sensibilisation réussie qui aura une suite. Une seconde édition de *Pas Sage en Seine* est déjà prévue pour mai 2010. Une association devrait bientôt se former pour assurer l'organisation de *Pas Sage en Seine*, second du nom.

Crédit : Benjamin Boccas http://www.benjaminboccas.com

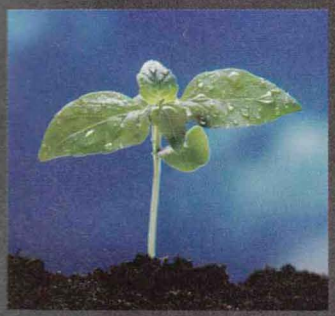
Si vous souhaitez participer à cette organisation ou proposer une conférence, un atelier, une démo... vous pouvez vous rendre sur le site web de l'événement et envoyer votre proposition. *Pas Sage en Seine* est l'occasion de rencontrer des gens aussi passionnés que vous, dans d'autres domaines, mais animés, dans leur discipline, par l'esprit du hacking.

Auteur : Olivier Laurelli
« Bluetouff »

Ressources

- http://passageenseine.org/
- http://www.futur-en-seine.fr/
- http://www.hackerspace.net/
- http://wiki.openmoko.org/wiki/ble:1
- http://www.reprap.org/bin/view/Main/WebHome

Bien débuter en électronique



Souvent, lorsque je parle avec des personnes qui connaissent mon intérêt pour l'électronique, mes interlocuteurs en arrivent à la même constatation : « je trouve le domaine très intéressant, j'ai plein d'idées, mais je ne sais pas par où commencer ».

Voilà une chose très amusante : alors qu'on pourrait penser que l'électronique et tripoter les composants conduit à l'informatique et à se détacher du hardware pour exercer dans de plus hautes sphères, force est de constater que c'est souvent l'effet inverse qui se produit. Bon nombre de mes connaissances, relations et amis sysadmins/loutres/BOFH s'intéressent à ce qui se passe au niveau inférieur (au sens technique du terme). Mais voilà, il existe une frontière qu'il est difficile de franchir entre le monde du software et celui du hardware.

Pendant longtemps, l'électronique était une affaire de données analogiques ou de simples circuits logiques. On retrouve encore de manière récurrente, dans les publications spécialisées, des articles sur la 376ème version de l'ampli 2×45 watts ou de l'alimentation stabilisée 5-48V. En parallèle, les publications informatiques traitent de code, de configuration et de pilotes de périphériques sans vraiment se soucier de ce que font les composants et les gentils électrons qui s'y promènent.

Pourtant, il existe tout un monde entre ces domaines : celui peuplé par les microcontrôleurs et les systèmes embarqués. La technologie actuelle fait des premiers des ensembles complets dont les performances dépassent souvent celles des machines avec lesquelles vous avez découvert la programmation. Ces petites choses de la taille d'une lentille à celle d'un cornichon se programment en assembleur, en C ou en basic (si, si !). Certains d'entre eux sont même capables de faire fonctionner des systèmes d'exploitation simples. Ils forment désormais le chaînon manquant entre électronique et informatique.

Disposer d'une étape intermédiaire entre les deux mondes ne signifie en rien l'absence d'effort nécessaire à la transition. L'électronicien, et on le constate malheureusement encore, aura tendance à développer un code qu'il gardera secret alors que le design même du circuit est parfaitement visible. Cette tendance est en voie de disparition, mais la notion de « diffusion des sources » n'est pas encore totalement admise. Pour l'informaticien, la barrière prend plusieurs formes : connaissances et notions de base en électronique, matériel, consommables, etc. Nous allons ici apporter quelques éléments qui vous permettront, je l'espère, de faciliter vos premiers pas.

1 Le matériel

Voici une grosse différence entre informatique pure et électronique. Pour un développeur le matériel se résume souvent à une machine plus ou moins puissante. L'administrateur système ou réseau en aura plusieurs auxquelles s'ajouteront *switchs*, câbles, etc. En électronique, le choix du matériel est capital. Il ne s'agit pas de code : avec de mauvais outils, vous ferez de mauvais travaux. Bien entendu, le prix est proportionnel à la qualité. À vous de définir votre limite de tolérance. Est-ce juste un coup de folie passager ou un hobby que vous comptez conserver pour une longue période ? Comptez-vous investir beaucoup de temps et d'énergie dans cette nouvelle passion ? Tout cela risque-t-il d'augmenter vos compétences professionnelles ?

Voici une petite *checklist* du matériel indispensable. Commençons par l'outillage :

- Le fer à souder. Attention, je ne parle pas du gros monstre de 120 watts de l'électricien, mais d'un matériel d'électronicien. Vous devrez y mettre le prix, ce n'est pas un choix, mais une question de bon sens. Vous pouvez trouver des fers pour quelques euros, mais vous en changerez régulièrement. Sur le long terme, mieux vaut investir une bonne centaine d'euros dans une station de soudage composée d'un fer et d'une unité d'alimentation. Un modèle avec un réglage de température et un jeu de pannes (embouts) sera le meilleur choix. Ce genre d'outil de bonne facture s'utilise plusieurs années, sinon dizaines d'années, en en prenant soin. Quoi qu'en disent certains, la qualité d'une soudure tient tout autant de la qualité du fer que de celle du soudeur. Avec un mauvais fer, vous ferez très certainement de mauvaises soudures.
- Pinces & tournevis : couper, visser, dévisser, serrer... Il vous faudra un outillage de précision. Les boutiques en ligne vous permettront d'acquérir ce type de matériel pour quelques euros sous la forme de jeux complets de pinces et de tournevis de différentes formes.

- Multimètre : Indispensable outil de mesure. Là aussi, le choix est vaste. N'optez pas pour le modèle le plus simpliste et le moins cher. La précision est un point important et les multimètres, même ceux proposant les mesures de courant, de tension, de capacité, de fréquence et de température ne sont pas nécessairement un bon choix s'ils semblent être un peu trop « des affaires ». Il faut que le matériel soit de bonne qualité et surtout calibré. À l'instar du fer à souder, un bon multimètre est un accessoire toujours utile qui ne vieillit pas.

Une fois bien équipé, il vous faudra des consommables. Dans cette catégorie, je classe les composants qui gravitent autour de l'élément central d'un montage et les accessoires à usage unique :

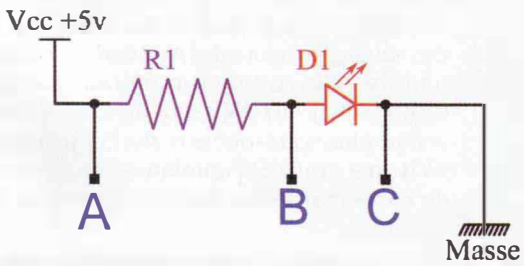
- Petites composants : Résistances, condensateurs, diodes, leds, transistors, régulateur de tension... Il s'agit du minimum vital. Les résistances et condensateurs peuvent s'acheter par lot à bas prix. Ceci vous permettra d'avoir un « capital » de départ. Il en va de même pour les leds et les diodes. Les régulateurs de tension et le 78L05 en particulier sont souvent indispensables lorsqu'on touche à l'embarqué ou aux montages communiquant avec un PC. Enfin, les transistors, selon les projets que vous comptez mettre en œuvre, sont souvent nécessaires. En dehors de leur aspect pédagogique, ils sont utiles sur plus d'un point. Créez-vous un petit stock avec des modèles courants comme les BC547, BC557 et 2N2222. Pour tous ces composants, je vous conseille clairement les sites d'enchères en ligne. Le matériel est souvent de bonne qualité et pour un délai de livraison un peu plus important (Chine, Hong Kong, Corée), les tarifs sont vraiment très avantageux.

- Les consommables : Souder et dessouder, c'est bien, mais encore faut-il le faire correctement. Pour cela, il vous faut de l'étain bien sûr, mais également du flux de soudure. Ceci peut paraître secondaire, mais une fois l'habitude prise d'utiliser cette pâte décapante, vous ne pourrez plus vous en passer. Si vous comptez récupérer des composants ou faire des erreurs (sic), achetez de la tresse à dessouder qui vous permettra très simplement de retirer l'étain des circuits et de dessouder très facilement les composants sans les détruire.

2 La base de la base de la théorie

S'il n'y avait qu'une seule chose à comprendre en électronique, ce serait ceci : la loi d'Ohm. La loi d'Ohm est simple, mais bien l'assimiler et en comprendre les implications est capital. La voici : U = R.I.

- U est la tension exprimée en volts.
- R est la résistance exprimée en ohms.
- I est l'intensité du courant exprimée en ampères.



Si nous prenons le circuit ci-dessus, nous avons une différence de potentiel (une tension) entre Vcc et la masse de +5 volts. La présence de la résistance limite le courant qui passe dans le circuit (à l'image d'une pression exercée sur un tuyau d'arrosage). Une led est dite « pilotée par un courant ». Comprenez par-là que, pour fonctionner correctement et de manière optimale, elle doit être traversée par un courant déterminé par le fabricant. Lorsqu'elle est alimentée dans ces conditions, une tension apparaît entre ses bornes (entre B et C).

Dans le cas présent, ces valeurs sont 10mA (milliampères) et 2 volts. Nous avons toutes les informations nécessaires pour calculer la valeur de la résistance en utilisant la loi d'Ohm : U = R.I. Soit U=3, puisque notre tension de départ est 5 et la tension aux bornes de la led 2, et I=0.01 ampère. Nous calculons donc :

```
U = R . I
3 = R x 0.01
R = 3 / 0.01
R = 300
```

Nous avons besoin d'une résistance de 300 ohms pour obtenir le résultat souhaité. 300 ohms n'est pas une valeur courante pour une résistance. On utilisera dont 330 ou 470, ce qui nous donnera respectivement un courant de :

```
U = R . I
3 = 330 x I
I = 3 / 330
I =~ 0.009
U =~ 9mA
```

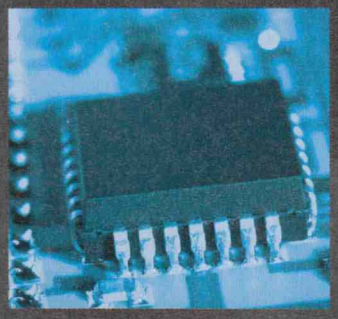
ou

```
U = R . I
3 = 470 x I
I = 3 / 470
I =~ 0.006
U =~ 6mA
```

La led est sous-alimentée, mais fonctionnera très bien. On utilise généralement des résistances de 330 ou 470 ohms pour les leds standards (rouges), mais le calcul est nécessaire pour des leds spéciales comme les leds super-lumineuses bleues ou blanches. Celles-ci, en fonction de leur puissance, peuvent utiliser des valeurs très variées. Une led bleue de 11000 mcd (très lumineuse) nécessite souvent un courant de 35mA et une tension à ses bornes de 3.3 volts.

INTRODUCTION

Le monde des microcontrôleurs



À la frontière entre informatique et électronique, les microcontrôleurs contiennent aujourd'hui autant de puissance que nos PC d'hier. Intégrant un grand nombre de périphériques, ils permettent à l'informaticien et au développeur de faire ses premiers pas en électronique, loin des interfaces graphiques, des grosses bibliothèques et des projets titanesques. Le microcontrôleur sera pour le programmeur un retour aux sources, une plateforme de développement entièrement maîtrisée et parfaitement bornée.

Auteur

- Denis Bodor

Un microcontrôleur comme un ordinateur repose sur une architecture de von Neumann composée de mémoire, d'une unité de calcul et de périphériques (dispositifs d'entrée/sortie). À la différence d'un ordinateur, tous ces éléments sont réunis sur une seule puce, un composant unique. Les premiers microcontrôleurs étaient relativement chers, peu puissants, limités en ressources et difficiles à programmer. Puis, est arrivé le très connu PIC de Microchip et le modèle 16f84 en particulier. Ce microcontrôleur a changé la donne et a rendu accessible cette technologie au plus grand nombre. Depuis, les choses ont encore changé. En effet, les microcontrôleur d'Atmel et, en particulier, la gamme AVR 8 bits occupe de plus en plus de place sur le marché, et ce, aussi bien dans l'industrie et dans l'éducation que chez l'amateur. Il faut dire que le projet Arduino y est pour quelque chose.

1 Arduino

En quelques mots, Arduino est une plateforme de prototypage *open source* basée sur des microcontrôleurs AVR et utilisant des cartes prototypes préfabriquées et peu coûteuses. Le projet livre un environnement de développement et un langage spécifique très proche du C. En réalité, c'est précisément du C, mais utilisant des bibliothèques et des macros spécifiques qui rendent le développement plus aisé. Les microcontrôleurs utilisés par les cartes Arduino sont programmés avec un *bootloader* chargé de faire le lien entre l'environnement de développement et la mémoire de l'AVR. C'est exactement le même principe qu'un bootloader comme GRUB, mais intégrant des fonctionnalités propres aux microcontrôleurs.

Le présent article, tout comme ceux présentant des montages autour des AVR dans ce magazine, n'utilise pas Arduino, préférant s'attacher directement aux AVR eux-mêmes. C'est une vision plus « *hardcore* » de l'utilisation des AVR en « grattant » quelques octets de Flash en l'absence de bootloader. C'est un choix.

2 Les AVR

AVR, en plus d'être une gamme, est avant tout une architecture RISC étendue reposant sur une structure de type Harvard (séparation physique des bus d'accès à la mémoire de données et la mémoire programme). Ces microcontrôleurs, entre autres particularités, utilisent des instructions exécutées en un seul cycle d'horloge. Comme l'horloge interne est à la même fréquence que l'externe (en fonction de la configuration par défaut de l'AVR), le cœur du microcontrôleur atteint donc 1 MIPS par mégahertz.

Les AVR peuvent être programmés en assembleur AVR ou en C. Notez cependant qu'Atmel précise que le cœur des AVR est optimisé pour le code généré à partir d'un compilateur C. Sous Windows, c'est l'environnement de développement livré par le fabricant qui est généralement le plus populaire : AVRstudio et WinAVR. Ce dernier est basé sur GCC en version Win32. Sous GNU/Linux ou tout autre système Unix en logiciel libre (ou non), on utilisera directement la version GCC destinée

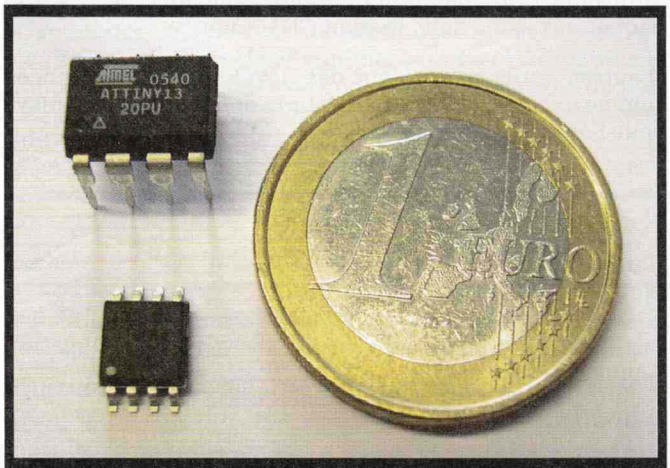
AVR 8 bits

aux AVR. La plupart des microcontrôleurs sont supportés pour peu qu'ils possèdent suffisamment de RAM.

Côté assembleur pour GNU/Linux, on se tournera généralement vers le Gerd's AVR assembler, AVRA ou tout simplement l'assembleur GCC avec une préférence personnelle pour Gavrasm d'un point de vue technique (ce n'est pas un logiciel libre : « *Free use of the source code and the compiled versions for non-commercial purposes. Distribution allowed if the copyright information is included* »). L'assembleur sera cependant réservé aux AVR de petites capacités, comme l'Attiny15L, qui ne possède pas de RAM, ou pour des développements pointus.

Il faut distinguer, dans la famille AVR, plusieurs catégories de microcontrôleurs :

- TinyAVR : Ce sont les plus petits de la gamme, disposant de 1 à 8 Ko de Flash, entre 32 et 512 octets de RAM (SRAM) et jusqu'à 512 octets d'EEPROM (*Electrically-erasable programmable read-only memory*). Certains TinyAVR ne disposent pas du tout de RAM, mais uniquement de 32 registres dont une partie seulement permet de travailler avec des valeurs immédiates. Le plus polyvalent, à mon goût, des TinyAVR est l'Attiny2313 disponible en boîtier DIP 20 broches aisément utilisable par le hobbyiste. L'Attiny13 est également très intéressant, car économique, en boîtier DIP 8 et disposant d'1 Ko de Flash et de 32 de RAM (donc programmable en C)

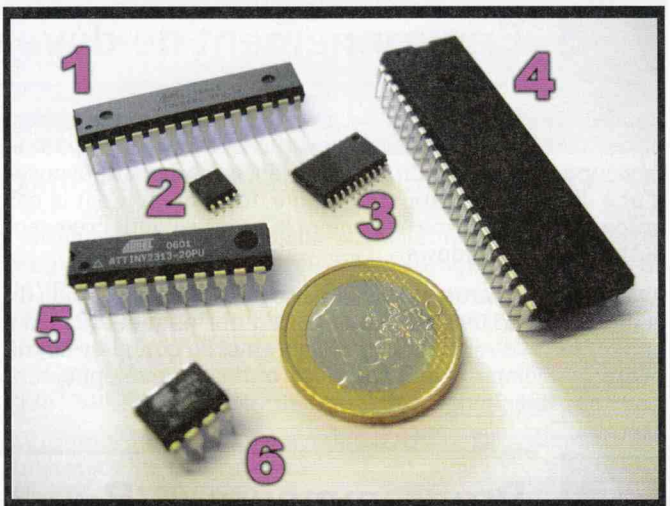


Un Attiny13 en boitier PDIP en haut et un Attiny85 en SOIC en bas. Respectivement 1 Ko de Flash, 64o d'EEPROM et 64o de SRAM pour l'un et 8Ko de Flash, 512o d'EEPROM et 512o de SRAM pour l'autre. Le tout avec un encombrement minimum.

- MegaAVR : Grands frères des TinyAVR, les MegaAVR sont équipés de plus de Flash (entre 4 et 256 Ko), mais également de plus de RAM (1 à 16 Ko) et davantage de périphériques. Le tout avec un jeu d'instructions légèrement différent et plus étendu.
- XMEGA : Ce sont les « monstres » des AVR avec 16 à 384 Ko de Flash et 2 à 16 Ko de RAM. Ils intègrent généralement l'ensemble des périphériques et bus disponibles pour la gamme AVR.
- AVR spéciaux : Ce sont des MegaAVR avec des possibilités étendues et spécialisées comme l'intégration d'un contrôleur LCD, du bus USB ou encore de PWM avancée. Il s'agit donc de circuits spécialisés reposant sur un cœur AVR.

Note

L'adressage immédiat permet de charger directement une valeur dans un registre par opposition à l'adressage indirect où le contenu d'un registre est initialisé par le contenu d'une zone mémoire. C'est un élément important à prendre en considération dans le développement assembleur sur AVR. Les registres `r0` à `r15` ne permettent pas de travailler avec les valeurs immédiates, mais les registres `r16` à `r31`, oui. De plus, certains registres permettent également de pointer des valeurs dans la mémoire Flash. `r30` et `r31` stockent ainsi une adresse en Flash et le registre `r0` contiendra la valeur après exécution de l'instruction `lpm` (*Load Program Memory*).



1) Atmega8 en boitier PDIP, un modèle courant largement utilisé. 2) Attiny85 en boitier SOIC, minuscule, mais puissant. 3) Attiny2313 en version SOIC, polyvalent et réduit. 4) Atmega32 en PDIP, beaucoup d'E/S, de Flash et de fonctionnalités intégrées. 5) Attiny2313 version PDIP, plus simple d'utilisation pour le hobbyiste. 6) Attiny13, petit, économique et programmable en C.

D'un point de vue de l'architecture microcontrôleur, les AVR intègrent généralement ce type de périphériques :

- De la mémoire EEPROM qui est gérée comme un périphérique. C'est absolument transparent pour le développeur C (avr-gcc + avr-libc) qui se bornera à une notification particulière et à l'utilisation de fonctions de lecture/écriture gracieusement mises à sa disposition (voir le site officiel de la avr-libc).
- Des *timers/counters* 8 et 16 bits. Un timer est un périphérique qui, une fois configuré, va incrémenter automatiquement et à une fréquence choisie, un compteur. Un mode *compare match* permet de fixer une valeur limite du compteur. Le fait de passer cette limite ou de faire déborder le compteur peut provoquer une interruption qui nous permettra d'exécuter une routine spécifique dans nos codes. Enfin, les timers/counters permettent

également un mode PWM (en bon français : modulation de largeur d'impulsion). Sans entrer dans les détails, il s'agit d'une technique utilisée pour créer des signaux continus à l'aide de circuits à fonctionnement tout ou rien. En d'autres termes, c'est une manière d'obtenir une sortie analogique par une logique binaire.

- Une interface sérielle asynchrone/synchrone USART (voir article sur l'éthylomètre page 18).
- Une interface sérielle synchrone SPI (*Serial Peripheral Interface*) via 2 ou 3 lignes (MISO, MOSI, SCK).
- Une interface sérielle sur bus TWI (compatible I²C) sur 2 fils (SDA et SCL).
- Une interface sérielle 1-wire (1 ligne de données).
- Des ports d'entrée/sortie (GPIO) d'usage général avec résistances de *pull-up* intégrées et configurables. Il s'agit là de ports accessibles librement depuis le code afin de contrôler des périphériques et lire des données.
- Un oscillateur interne calibré. Ceci permet aux AVR de fonctionner sans autres composants externes. Il suffit de les alimenter pour qu'ils fonctionnent contrairement à d'autres microcontrôleurs nécessitant la mise en œuvre de circuits parfois complexes.
- Un ou plusieurs convertisseurs analogique/numérique 10 bits permettant la mesure de valeurs analogiques (voir l'article sur l'éthylomètre page 18).
- Un ou plusieurs comparateurs analogiques permettant un fonctionnement binaire par comparaison de valeurs.

Comme vous pouvez le voir, les AVR sont très riches en fonctionnalités auxquelles il faudra encore ajouter différents modes de mise en veille, une détection de bruit d'alimentation, un *watchdog* interne avec oscillateur séparé, une gestion de bootloader pour les MegaAVR, une protection de la mémoire Flash, etc. Nous ne traiterons pas l'ensemble de ces fonctionnalités dans les quelques articles sur base AVR dans les pages qui suivent. La tâche n'est pas raisonnablement possible en si peu de pages. Les AVR représentent un terrain de jeu bien plus vaste qu'il n'y paraît. Je puis vous assurer qu'un petit composant comme un Atmega8 ou même un Attiny85 peut vous conduire à de longues heures de recherches et de découvertes tout aussi passionnantes que consommatrices d'heures de sommeil.

3 Environnement de développement

Sous GNU/Linux, puisqu'il s'agit de la plateforme de choix du magazine, nous ne manquons pas de ressources pour le développement AVR. En premier lieu, nous devons disposer d'une chaîne de compilation digne de ce nom. C'est le cas grâce à GCC qui forme également la base de l'outil Freeware d'Atmel pour Windows.

Avec une distribution Debian, par exemple, l'installation du compilateur se fera via l'installation des paquets **`gcc-avr`** et **`binutils-avr`**. Vous disposerez ainsi du compilateur, de l'éditeur de liens, et des différents outils de développement (**`avr-objdump`**, **`avr-nm`**, **`avr-as`**, etc.).

Ce n'est pas tout. Le projet AVR libc fournit des bibliothèques C sous licence BSD modifiée pour une utilisation de concert avec GCC. L'ensemble forme alors une suite logicielle vous permettant de rapidement faire vos premiers pas dans le développement AVR sous GNU/Linux, mais également les différents BSD. Les bibliothèques sont installées via le paquet **`avr-libc`** sous Debian GNU/Linux.

D'autres outils permettent des développements avancés comme **`avr-gdb`** et AVaRICE pour le débogage ou Simulavr pour l'émulation/simulation.

4 Programmeurs AVR

Une fois le code pour votre AVR compilé et transformé dans un format adapté, il est nécessaire de le charger dans la mémoire Flash du composant. De nos jours, la plupart des microcontrôleurs utilisent l'ISP (*In-System Programming*) ou programmation in-situ en français. Les utilisateurs ayant fait l'expérience des PIC 16f84 il y a quelques années se souviennent sans doute du montage JDM permettant la programmation du PIC. Il était alors nécessaire d'utiliser un montage spécial pour programmer la mémoire du composant pour ensuite le déplacer et l'intégrer dans le montage définitif.

Avec la programmation in-situ, les choses sont différentes. Le composant est programmé directement sur le montage final. Il suffit de prévoir la connectique en conséquence. Les AVR utilisent une connexion SPI pour la programmation in-situ. Seulement trois lignes/signaux sont nécessaires :

- SCK (*Serial Clock*) : Il s'agit du signal d'horloge pour les instructions vers et depuis le microcontrôleur à programmer. Ce signal est toujours généré par le programmeur in-situ.
- MISO (*Master In Slave Out*) : Voici la ligne de communication du microcontrôleur (esclave) vers le programmeur (maître).
- MOSI (*Master Out Slave In*) : C'est la ligne de communication inverse, du programmeur (maître) vers l'AVR (esclave).

En plus de ces signaux, il faut prendre en compte :

- GND, la masse (classique).
- VCC, la tension d'alimentation du microcontrôleur (identique à la tension d'alimentation en situation).
- RESET. Pour que l'AVR puisse être programmé, il doit passer dans un mode particulier. Pour ce faire, c'est la ligne de reset qui est utilisée. Elle est donc généralement contrôlée par le programmeur in-situ.

La programmation du composant en elle-même utilise un protocole particulier, largement documenté par le constructeur Atmel. Ainsi, il existe plus d'une solution pour charger votre code compilé dans un AVR. En premier lieu, il convient de parler des solutions proposées par Atmel sous forme de kit de développement. Le plus connu de ces kits est le STK500.

La platine est non seulement un programmeur ISP, mais également une plateforme d'initiation et d'expérimentation. Son prix est de l'ordre d'une centaine d'euros.

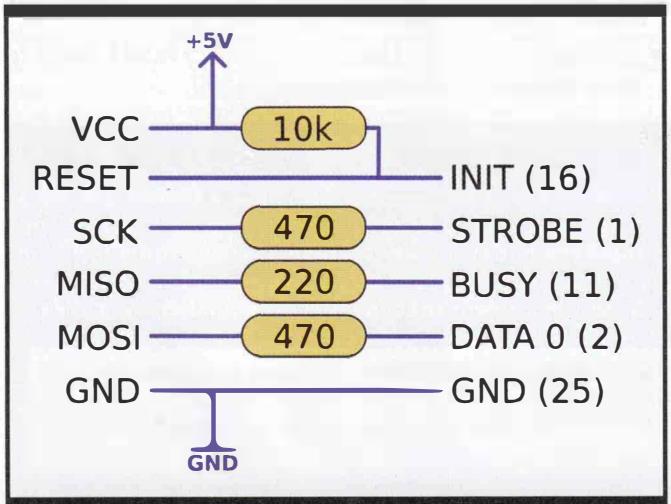


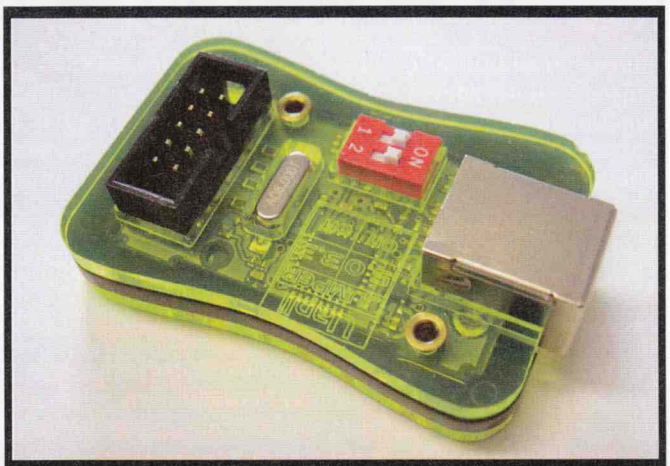
Figure 1

D'autres solutions existent puisque les spécifications de la méthode de programmation sont largement diffusées ainsi que la description du protocole utilisé. Ainsi, sans doute le plus économique des programmeurs in-situ est l'adaptateur DAPA. Celui-ci est constitué de 3 résistances de 220 et 470 ohms (voir schéma en figure 1). Malheureusement, bien que parfaitement supporté par les outils de programmation comme **uisp** et AVRdude, cette solution comporte un gros problème : le montage fort simple utilise un port parallèle. Vous n'êtes pas sans savoir que ce type de ports se fait de plus en plus rare sur les machines x86 modernes. De plus, les adaptateurs USB ne vous seront d'aucun secours puisqu'il s'agit d'adaptateurs USB/imprimante et non USB/parallèle. L'accès direct aux lignes de données et de contrôle du port est, en effet, indispensable. Notez qu'il existe également une solution du même type pour le port série (RS-232), mais je n'ai, personnellement, réussi qu'une seule fois à le faire fonctionner, et ce, sur une seule machine et avec un « vrai » port série (non pas un adaptateur USB/série).

Dernière solution présentée ici, l'adaptateur USB compatible STK500. Plusieurs solutions existent. Parmi elles, figure la fabrication d'un adaptateur comme celui proposé par LadyAda, l'USBtinyISP. Plusieurs schémas sont disponibles sur le Web et permettent tous de remplir la même fonction : émuler un kit STK500. Petit problème du type « l'œuf et la poule » : il est nécessaire de programmer un AVR pour construire le programmeur in-situ. Le programmeur de LadyAda utilise un Attiny2313 (http://www.ladyada.net/make/usbtinyisp/), celui de Thomas Fischl un Atmega8 ou Atmega48 (http://www.fischl.de/usbasp/).

Toujours dans le domaine du programmeur compatible, vous avez finalement la solution de l'achat. Les sites de vente aux enchères en ligne regorgent de ce type de matériel pour un tarif allant de 15 à 30 euros, port compris. Le matériel est généralement de bonne qualité, avec d'excellentes finitions. C'est, à mon goût, un juste équilibre entre le kit officiel et le montage maison souvent moins compact et moins transportable.

Notez que l'Arduino règle le problème d'une autre manière, ce qui explique en partie son succès. Les montages Arduino utilisent un bootloader qui se charge de flasher le code dans le microcontrôleur. Ce code est transmis directement en USB, ce qui évite l'utilisation d'un programmeur in-situ supplémentaire. En revanche, on est plus proche de la platine d'expérimentation que du réel montage électronique, puisque chaque platine Arduino intègre, en quelque sorte, la solution de développement. Si vous comptez vous investir dans les AVR et multiplier les montages, la méthode classique montage+ISP+programmeur est souvent préférable et plus économique.



Un programmeur USB compatible stk500v2 provenant d'un vendeur sur un site d'enchères en ligne. Ce type de matériel est un bon compromis entre coût et facilité d'utilisation. Pour quelques 15 ou 20 euros, vous pouvez disposer d'un programmeur USB utilisable sans problème avec AVRdude sous GNU/Linux.

Quel que soit le programmeur in-situ utilisé, il répondra très certainement à l'un des protocoles standards. Attention, j'entends ici par « protocole » la manière de contrôler le programmeur et non le protocole de programmation ISP de l'AVR (qui, lui, est défini une fois pour toutes par Atmel). Les deux utilitaires les plus connus et utilisés sous GNU/Linux sont « uisp » et AVRdude. Ce dernier est souvent largement préféré, car supportant un grand nombre de programmeurs et offrant une souplesse d'utilisation exemplaire.

AVRdude pour AVR Downloader/UploaDEr dispose des fonctionnalités suivantes :

- Utilisation en ligne de commande pour le chargement et la récupération du code d'un grand nombre d'AVR. Ceci inclut le contrôle des *fuses* (voir plus loin).
- Intégration aisée dans les **Makefile**.
- Possibilité de fonctionnement en mode interactif pour l'examen de zone mémoire et le contrôle de plusieurs éléments de configuration des platines STK500.
- Support d'une gamme importante de programmeurs allant du plus simple montage aux platines d'expérimentation intelligentes.
- Support de plusieurs formats de binaires. Votre code une fois compilé peut être encodé dans un format standard comme Intel Hex, Motorola S-Record, etc. AVRdude supporte ses formats sans problème et est également capable d'autodétecter le format utilisé.
- Fonctionnement avec GNU/Linux, *BSD, Windows et Solaris.

AVRdude s'installe avec toutes les distributions sous la forme d'un paquet (**avrdude** sous Debian). Le fichier **/etc/avrdude.conf** contient les éléments de configuration de l'outil. C'est dans ce fichier que sont décrits les différents programmeurs in-situ utilisables et leurs caractéristiques

(brochages, etc.). Le choix du programmeur se fera en ligne de commande via l'option `-c` suivie du nom du programmeur et l'option `-P` pour désigner le port. Ce dernier élément pourra également être spécifié dans un fichier `~/.avrduderc` sous la forme `default_serial = "/dev/STK500";` par exemple.

Dans les deux cas de figure qui nous intéressent ici, nous pouvons donc avoir :

- `avrdude -c dapa -P /dev/parport0` pour le programmeur simpliste sur port parallèle (DAPA) ;
- `avrdude -c stk500v2 -P /dev/STK500` pour le programmeur USB compatible STK500.

Note

Remarquez l'utilisation du périphérique **/dev/STK500**. Il s'agit d'une astuce permettant de ne pas « courir après les entrées » dans **/dev**. Un programmeur USB compatible STK500 apparaît sous la forme d'un nouveau port série et, en fonction de la présence d'autres adaptateurs USB/série, il peut rapidement devenir énervant de chercher quelle entrée correspond à quel adaptateur. Pour contourner le problème, il suffit d'écrire quelques règles **udev** pour arbitrairement définir tout cela via des liens symboliques. Voici un exemple de fichier de règles (**/etc/udev/rules.d/USBtty.rules**) :

```
SUBSYSTEMS=="usb", KERNEL=="ttyUSB*", ATTRS{product}=="FT232R USB UART",  SYSFS{serial}=="A90066j7", SYMLINK+="TTYUSB0"
SUBSYSTEMS=="usb", KERNEL=="ttyUSB*", ATTRS{product}=="FT232R USB UART",  SYSFS{serial}=="A90066ac", SYMLINK+="TTYUSB1"
SUBSYSTEMS=="usb", KERNEL=="ttyUSB*", ATTRS{product}=="FT232R USB UART",  SYSFS{serial}=="A900669W", SYMLINK+="TTYUSB2"
SUBSYSTEMS=="usb", KERNEL=="ttyACM*", ATTRS{product}=="DX-ISP",  SYMLINK+="STK500"
```

Les trois premières lignes correspondent à des adaptateurs USB/série utilisant un composant FT232 qui sont généralement représentés par un fichier **/dev/TTYUSBn** où **n** est le numéro attribué en fonction de l'ordre de connexion. Ici, nous utilisons le numéro de série du périphérique pour identifier chaque adaptateur et lui attribuer une entrée sous la forme **/dev/TTYUSBn** où **n** est un numéro choisi (qu'on renseignera également au feutre indélébile sur l'adaptateur lui-même). La dernière ligne est le programmeur in-situ qui sera référencé par **/dev/STK500**. Il s'agit, là encore, d'un lien symbolique pointant sur **/dev/ttyACM0**, l'entrée standard pour les adaptateurs USB/série reposant sur les spécifications *Universal Serial Bus Communication Device Class Abstract Control Model* (habituellement des modems USB). Il n'est pas certain que votre programmeur in-situ utilise l'ACM. Il peut s'agir d'un convertisseur à base de FT232 (**ftdi_sio**), de Silicon Labs CP2101/CP2102 (**cp2101**) ou encore de Prolific PL2303 (**pl2303**). Consultez la sortie de la commande **dmesg** après connexion du programmeur pour connaître la référence utilisée, puis utilisez le chemin pour obtenir des informations avec la commande **udevinfo -a -p /sys/bus/usb/devices/[…]**. Là, vous trouverez des données utiles pour créer votre règle **udev** (numéro de série, nom du constructeur, nom du produit, etc.). Ainsi, vous n'aurez plus à vous soucier du nom de l'adaptateur USB et utiliserez tout le temps la même désignation avec AVRdude.

5 Fusibles ou fuses

Comme précisé dans la section précédente, AVRdude permet également de configurer les fuses ou fusibles des AVR. Les fuses servent à configurer le comportement général du microcontrôleur comme sa source d'horloge, le temps de démarrage ou la gestion de l'EEPROM en cas de programmation de la mémoire Flash. Les fuses sont classés en trois groupes :

- l'octet haut (*Fuse High Byte*) ;
- l'octet bas (*Fuse Low Byte*) ;
- l'octet supplémentaire (*Fuse Extended Byte*).

Habituellement, la configuration par défaut telle que prévue par le fabricant correspond à un usage normal du composant. On s'intéressera à la configuration des fuses pour différentes raisons :

- Changement de la source d'horloge. Par défaut, les AVR utilisent l'oscillateur interne pour fonctionner. Cependant, en fonction de l'usage du microcontrôleur, il peut être nécessaire d'utiliser un quartz. C'est le cas par exemple pour faire communiquer un AVR via un port série avec un taux d'erreur nul.
- Configuration du BOD (*Brown-out Detector*) pour un reset automatique en cas de variation ou d'instabilité d'alimentation.
- Configuration du watchdog, un système permettant un reset automatique en cas de plantage du code dans l'AVR.
- Configuration de divers éléments comme la protection du code dans l'AVR, le temps de démarrage, la division de la fréquence du signal d'horloge, l'activation des fonctions de *debug*, etc.

Note

Il est important de remarquer qu'un bit est programmé lorsqu'il est à 0 et non programmé à 1. C'est une source de confusion courante. Une mauvaise configuration peut rendre le microcontrôleur difficile à reprogrammer (mauvaise source d'horloge, bits de protection, désactivation de l'ISP). Soyez attentif !

La configuration des bits de fuses se fera avec AVRdude aussi simplement que le chargement du code lui-même. Généralement, ces deux étapes sont placées dans le `Makefile` du projet.

Si vous avez des difficultés pour vous y retrouver dans les bits, leurs fonctions et leurs valeurs par défaut, un certain nombre de sites proposent de les calculer pour vous. Un bon exemple est http://www.engbedded.com/fusecalc/, où il vous suffira de cocher les cases du formulaire HTML pour obtenir la valeur souhaitée et inversement obtenir la configuration en fonction de la valeur High Fuse et Low Fuse.

6 Pas à pas : mon premier programme

Nous allons maintenant voir ensemble comment se passe la réalisation d'un code pour un AVR, du développement jusqu'à l'étape de programmation du composant. Il nous faut choisir un AVR, puisqu'ils possèdent tous leur lot de fonctionnalités et que le code pour un AVR, bien que portable, nécessite des petites retouches avant utilisation sur un autre modèle. Fondamentalement et en dehors des fonctionnalités absentes de l'un ou l'autre composant, ceci se limite souvent à la désignation des ports d'entrée/sortie. Ce n'est que lorsqu'on touche à des caractéristiques spécifiques que le portage devient plus délicat (utilisation du port série de l'AVR, convertisseurs analogique/numérique, timers, etc.).

Nous allons donc développer le **hello World** des microcontrôleurs pour un Attiny2313, j'ai nommé : la led qui clignote. Nous partons du principe que tous les outils décrits plus haut sont installés (GCC et AVRdude), ainsi que ceux nécessaires à tout développeur comme **make** ou Vim (oui, c'est un appeau à trolls).

6.1 Code en C

Entrons directement dans le vif du sujet avec le code pour l'Attiny2313. Le montage est relativement simple, puisqu'il suffit de connecter la broche 20 (VCC) à +5 volts et la 10 à la masse. Nous connecterons ensuite la broche 12 (PB0) à une résistance de 470 ohms reliée à l'anode d'une led standard. La cathode de la led sera reliée à la masse. Un 1 logique sur le bit correspondant à cette ligne sur le port donné allumera donc la led, témoin du bon fonctionnement du code. Notre montage est prêt. Voici le source :

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <avr/io.h>

#define F_CPU 1000000UL
#include <util/delay.h>

int main (void) {
  DDRB |= (1<<PB0);
  while (1) {
    PORTB |= (1 << PB0);
    _delay_ms(150);
    PORTB &= ~(1 << PB0);
    _delay_ms(500);
  }
}
```

Nous commençons par l'inclusion des fichiers d'en-tête standards avec, dans l'ordre, les fonctions utilitaires générales, les entrées/sorties standards et les types entiers standards. Comme nous ferons usage des ports d'entrée/sortie, nous ajoutons **io.h**. Nous n'avons pas à nous préoccuper du modèle de microcontrôleur, puisque nous spécifierons cela en argument du compilateur. Le bon fichier d'en-tête (**iotn2313.h**) sera automatique inclus. Enfin, deux lignes importantes lorsqu'on travaille avec des délais. Nous devons spécifier la valeur de **F_CPU** correspondant à la fréquence utilisée par l'horloge de l'AVR avant d'inclure le fichier **delay.h**.

Le lecteur attentif qui aura pris le temps de parcourir la *datasheet* de l'Attiny2313 aura sans doute remarqué que celui-ci est, par défaut, cadencé à 8 Mhz par l'oscillateur interne. Pourquoi donc alors spécifier **1000000UL** ? L'oscillateur interne fonctionne effectivement à 8 Mhz, mais la configuration des fuses par défaut active le bit **CKDIV8** (bit 7 du Low Fuse byte) divisant la fréquence par 8. Nous avons donc bien 1 Mhz par défaut. Ceci est valable pour l'Attiny2313, mais pas forcément pour les autres modèles. Pensez à toujours vérifier ce genre de choses et en particulier la valeur par défaut des fuses en passant d'un AVR à un autre. L'Atmega8, par exemple, ne possède pas ce bit et cette fonctionnalité. Il ne faut, en aucun cas, copier bêtement la valeur d'un fuse d'un modèle à un autre !

Nous en arrivons à la fonction **main()** qui, comme pour un code C standard, est la fonction principale de notre code. Un certain nombre de registres gouvernent le fonctionnement général d'un AVR, d'autres permettent de configurer les périphériques intégrés. Les noms de ces registres sont définis par le fichier inclus par **io.h** (ici **iotn2313.h**) et correspondent aux noms présents dans la documentation du constructeur.

Le registre **DDRB** utilisé ici est le registre qui détermine le fonctionnement du port d'E/S appelé B. **DDRB** signifie *Data Direction Register port B*. Ce registre de 8 bits précise donc si chaque ligne est utilisée comme entrée ou comme sortie. Par défaut, après initialisation, les ports génériques d'un AVR sont toujours configurés en entrée. Il convient donc d'écrire un 1 à l'emplacement correspondant à la sortie que nous souhaitons activer. Ici, c'est la broche 12 (PB0) qui nous intéresse. Nous passons donc le bit **PB0** (**0**) à 1.

Note

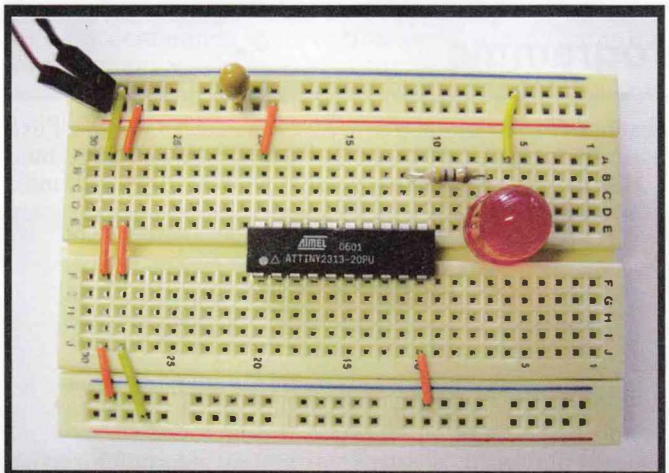
Résistances de rappel

Lorsqu'une ligne d'un port est configurée en entrée, elle se trouve par défaut dans une configuration où elle n'utilise pas de résistance de rappel. Ainsi, si la circuiterie externe utilise une sortie à collecteur ouvert (Vcc ou rien), la lecture du bit est plus ou moins aléatoire. Pour corriger cela, on utilise une résistance de forte valeur pour ramener le signal à Vcc si rien n'est branché. Si le composant externe utilise un 0 logique équivalent à la mise à la masse, la lecture retourne 0, sinon Vcc. Les AVR disposent en interne de résistances de rappel configurables. Pour activer une résistance sur un ligne d'un port, il suffit d'écrire 1 dans le registre du port (avec **DDR*** à 0).

Notez la syntaxe utilisée permettant de rendre le code beaucoup plus lisible. Nous utilisons la valeur 1 que nous décalons de **PB0** position(s) vers la gauche et nous opérons un OU logique avec les valeurs actuelles du registre **DDRB**. Une autre syntaxe que vous risquez de retrouver dans du code d'autres développeurs est : **DDRB |= _BV(PB0)**. **_BV** est défini dans **avr/sfr_defs.h**, **#define _BV(bit) (1 " (bit))** et remplace les anciennes macros **sbi** (*set*) et **cbi** (*clear*).

Nous en arrivons au cœur du programme avec une boucle **while** infinie. C'est quelque chose qu'on oublie facilement lorsqu'on travaille avec des machines de bureau ou des serveurs, mais un processeur ne s'arrête jamais. Il fait toujours quelque chose (si on exclut, bien entendu, les fonctions de sommeil et autres). Cette réalité resurgit clairement dans le monde des microcontrôleurs. Sans une boucle infinie, votre AVR va tout simplement s'arrêter et ne plus rien faire. Il sera planté. À l'intérieur de notre boucle, nous allons :

- Activer le bit 0 du registre correspondant à l'état du port B comme nous l'avons fait avec le registre **DDRB**.



Notre montage exemple sur platine à essai. Comme vous pouvez le voir, la mise en œuvre d'un AVR ne nécessite que très peu de connectique. Certains modèles nécessitent cependant l'utilisation d'une résistance de 10 kOhm entre la broche 1 et le Vcc (alimentation) afin de forcer la ligne de reset au niveau haut. Dans le cas de l'Attiny2313, cette résistance est intégrée dans le composant. Notez l'utilisation d'un condensateur tantale pour stabiliser l'alimentation.

- Utiliser une fonction de retardement mise à disposition par **delay.h** en passant en argument un nombre de millisecondes.
- Désactiver le bit PB0.
- Utiliser un nouvelle fois la fonction de retardement avec une durée un peu plus longue.

La libc pour AVR (paquet **avr-libc**) est quelque chose de très bien fait. La fonction **_delay_ms(double _ms)** prend, comme vous le voyez, un double en argument. La valeur maximale est normalement de 262.14 ms divisée par **F_CPU** en mégahertz (ici 1). Si cette valeur est dépassée, la résolution de la fonction est réduite à 1/10 ms nous permettant alors d'aller jusqu'à 65535 millisecondes, soit 6 secondes et demi. Ceci est totalement transparent pour le développeur. De la même manière, il est possible d'utiliser la fonction **_delay_us(double _us)** pour spécifier un délai en microsecondes. Là encore, une valeur maximum existe, 768 µs divisé par **F_CPU** en MHz. Si cette valeur est dépassée, **_delay_ms(double __ms)** est automatiquement utilisé. Dans la plupart des cas, ceci n'est pas un problème, mais il est important de le relever si vous travaillez un jour avec des timings précis. Jetez toujours un coup d'œil aux fichiers d'en-tête. Ils sont largement commentés.

Notre code est terminé. Il ne vous reste plus qu'à l'enregistrer sous, par exemple, le nom **main.c** et passer à la suite.

6.2 Compilation

Une fois notre code vérifié et stocké dans **main.c**, il est temps d'en faire quelque chose d'utilisable par notre AVR. Nous allons donc le compiler avec GCC :

```
% avr-gcc -Os -g -Wall -mmcu=attiny2313 -o main.elf main.c
```

La quasi-totalité des options utilisées ne sont en rien spécifiques aux AVR. Nous avons ainsi **-g** pour inclure les informations de debug, **-Os** pour l'optimisation pour la taille, **-Wall** pour obtenir des avertissements sur tous les problèmes et **-o** pour spécifier le fichier de sortie. **-mmcu** est utilisé pour spécifier le jeu d'instructions pour le microcontrôleur cible. Il y a 5 jeux d'instructions différents (**avr1** à **avr5**), mais nous spécifions ici le microcontrôleur. À la charge de GCC d'adapter et d'utiliser le jeu d'instructions qui convient (cf. **man gcc**).

En sortie de la commande, nous obtenons un fichier au format ELF **main.elf** qui n'est pas utilisable en tant que tel. Il contient bien plus que le code binaire pour l'AVR et n'est pas dans un format utilisable avec AVRdude. Nous utilisons donc **objcopy** à la fois pour extraire les données qui nous intéressent et les stocker dans un fichier au format Intel Hex :

```
% avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

Nous récupérons les sections (ou segments) **.text** pour le code et **.data** pour les variables et les exportons dans le fichier **main.hex**, directement utilisable par AVRdude.

Le fichier **main.elf**, quant à lui, nous apporte son lot d'informations utiles comme l'encombrement du composant :

```
% avr-size --format=avr --mcu=attiny2313 main.elf
AVR Memory Usage
----------------
Device: attiny2313

Program:     130 bytes (6.3% Full)
(.text + .data + .bootloader)

Data:          0 bytes (0.0% Full)
(.data + .bss + .noinit)
```

Comme vous pouvez le voir, nous arrivons à un total de 130 octets, ce qui représente quelques 6% d'occupation de la mémoire Flash d'un Atiny2313. Les outils de développement ne vous signaleront pas un éventuel débordement de taille. Sans **avr-size**, vous en auriez la surprise (parfois très désagréable) au moment de charger le code dans le composant.

Vous remarquerez ici que la section **.data** est vide. En effet, nous n'utilisons pas une seule variable, initialisée ou non. Concernant l'occupation de la mémoire vive, d'autres commandes sont utiles comme l'option **-Wl,-Map,fichier.map** de l'éditeur de liens qui vous permettra d'inspecter l'allocation mémoire et d'éventuels problèmes découlant de l'optimisation du code. **avr-nm**, utilisé avec l'option **-n**, vous permettra d'obtenir la liste triée des symboles également très appréciable en cas de problème d'allocation.

Note

Allocation et optimisation

Le compilateur GCC tentera d'optimiser l'utilisation de la mémoire pour ne pas surcharger inutilement le code. Malheureusement, il arrive que cette optimisation ne soit pas optimale. Un exemple classique consiste à déclarer une variable qui sera modifiée dans une routine d'interruption. Lors de l'analyse, comme cette variable n'est pas modifiée dans le cheminement normal, le code généré n'actualisera pas sa valeur. Exemple :

```
uint8_t variable;
...
ISR(SOME_vect) {
  variable = 42;
}
[...]
  if (variable != 0) {
    [...]
  }
```

Nous déclarons ici un entier sur 8 bits qui est testé, mais pas modifié dans le corps du code.

Note

Le compilateur considérera donc un accès unique à **variable** et optimisera en conséquence. Le test ne fonctionnera donc pas, car **variable** restera à 0. Pour contourner le problème, il est donc nécessaire d'utiliser la directive **volatile** lors de la déclaration avec `volatile uint8_t variable;`. Ainsi, on s'assure d'un rechargement systématique de la variable dans un registre. C'est un problème courant lorsqu'on travaille avec des interruptions. Pensez-y !

6.3 Programmation/chargement

Nous voici donc en possession d'un magnifique fichier binaire au format Intel Hex. La figure 2 montre l'exemple de code compilé dans ce format. C'est théoriquement exactement la même chose que ce que vous devez obtenir (avec des versions GCC et libc identiques).

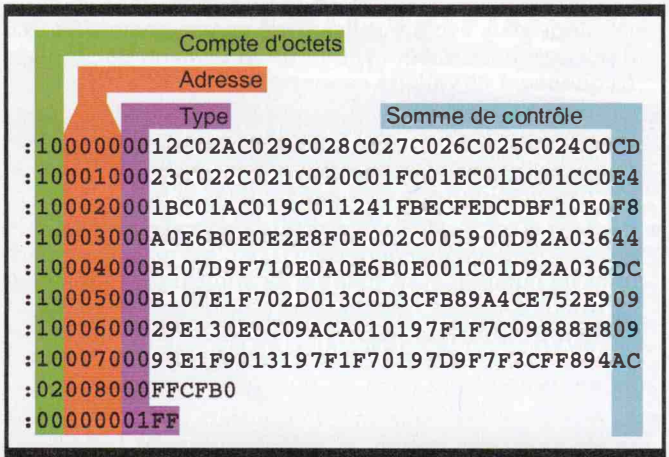


Figure 2

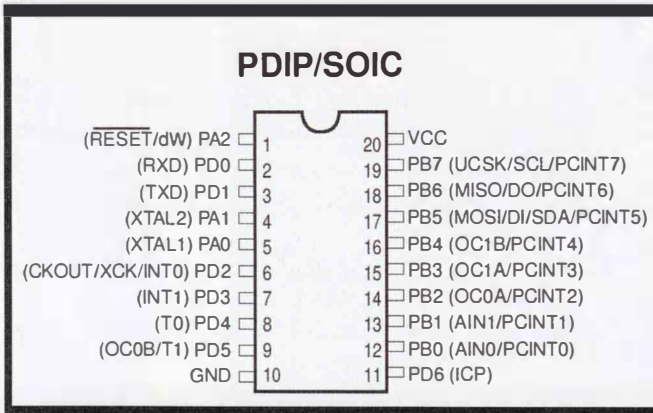
Dans notre exemple, nous utiliserons un programmeur in-situ compatible STK500v2 acquis sur un site d'enchères en ligne et vendu pour quelque 20 euros sous la désignation « USB ISP-Programmer für ATMEL AVR, ATmega , ATTiny » (le produit vient d'Allemagne). Comme détaillé plus haut, le programmeur est accessible sous **/dev/STK500**. Nous utiliserons la commande **avrdude** ainsi :

```
% avrdude -p t2313 -c stk500v2 -i 8 flash:w:main.hex
avrdude: AVR device initialized and ready to accept instructions
Reading | [redacted] | 100% 0.01s
avrdude: Device signature = 0x1e910a
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

L'option **-p** permet de préciser le modèle d'AVR utilisé. Remarquez que la désignation de ce dernier n'est pas la même que celle utilisée par GCC. La documentation de l'utilitaire fournit une liste de microcontrôleurs supportés (quasiment tous les modèles). AVRdude tentera de vérifier votre choix en utilisant la signature numérique du composant comme le précise la sortie. Il est possible d'outrepasser cette vérification. Cependant, s'il s'agit bien du composant utilisé et qu'il est supporté par AVRdude, mieux vaut se repencher sur les connexions ou éventuellement sur le délai spécifié via l'option **-i**.

-c vous permettra de préciser le modèle de programmeur utilisé. Ici, nous avons **stk500v2** pour le programmeur compatible. **dapa** serait utilisé pour le programmeur simpliste via port parallèle. Pour spécifier le port utilisé, soit vous créez **~/.avrduderc** contenant **default_serial = "/dev/STK500";**, soit vous spécifiez le port avec **-P /dev/STK500**. AVRdude, juste après le chargement du code, vérifiera automatiquement les données enregistrées avec une phase de lecture post-chargement. En cas de problème, un message d'avertissement sera automatiquement affiché vous indiquant la valeur du premier octet problématique et son emplacement. Ce type de problème peut avoir plusieurs sources, mauvaise connexion, lenteur du programmeur ou, plus rarement, composant défectueux. Vérifiez la connectique et éventuellement allongez le délai (**-i**). Réétudiez également votre montage. Il est possible que des composants connectés aux broches utilisées par la programmation in-situ interfèrent avec la programmation (généralement, les composants utilisant une liaison SPI).

Une fois le code chargé, l'AVR opère un reset et se met immédiatement à utiliser le code. Vous venez de programmer votre premier microcontrôleur Atmel AVR 8 bits. Félicitations !



Chaque composant dispose de sa documentation officielle du constructeur, la datasheet. Ici, un schéma provenant de la documentation de l'Attiny2313 présentant les broches disponibles, leurs numéros, leur nom et leur nom dans d'autres modes d'utilisation. Pour information, la datasheet du composant en question fait quelques 230 pages dans sa version complète. Absolument tout y est décrit, des spécifications à la manière d'utiliser chaque fonctionnalité en assembleur ou en C.

6.4 Tout dans un Makefile

Lors de la phase de découverte des AVR ou du monde des microcontrôleurs en général, il est intéressant de procéder par étapes. Plus tard, lorsque vous n'aurez plus besoin de cet aspect pédagogique, vous souhaiterez sans doute aller un peu plus vite. Pour ce faire, et à l'instar du développement avec un système Unix classique, l'utilisation de **make** s'avère indispensable.

Nous allons prévoir dans le **Makefile** plus de choses que nous n'en avons utilisées pour ces premiers pas dans le monde AVR :

```
TARGET_ARCH     = -mmcu=attiny2313
OBJECTS         = main.o
STACK           = 32
FLASH           = 2048
SRAM            = 128
PGMER           = stk500v2

CC      = avr-gcc
CFLAGS  = -Os -g -Wall -I.
```

```
ASFLAGS = -Os -g -Wall -I.
LOFLAGS = -g -Wl,-Map,main.map
MODULES = $(OBJECTS)

all:            main.hex

clean:
        rm -f main.elf *.o tags *.sch~ gschem.log *.hex *.map *.sym *~

main.elf:       $(MODULES)
        $(LINK.o) -o $@ $(MODULES)

main.hex:       main.elf
        avr-objcopy -j .text -j .data -O ihex main.elf main.hex
        avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
           --change-section-lma .eeprom=0 -O ihex main.elf eeprom.hex
        avr-nm -n main.elf > main.sym

main.o:

dprog: all
        avrdude -p t2313 -c $(PGMER) -i 8 -U flash:w:main.hex

rd_fuses:
        avrdude -p t2313 -c $(PGMER) -i 8  -U hfuse:r:-:b
        avrdude -p t2313 -c $(PGMER) -i 8  -U lfuse:r:-:b

wr_fuses:
        # preserve eeprom
        avrdude -p t2313 -c $(PGMER) -i 8  -U hfuse:w:0x9f:m
        # serial quartz clock
        avrdude -p t2313 -c $(PGMER) -i 8  -U lfuse:w:0xff:m

eeprom: eeprom.hex
        avrdude -p t2313 -c $(PGMER) -i 8 -U eeprom:w:eeprom.hex

reeprom:
        avrdude -q -q -p t2313 -c $(PGMER) -i 8 -U eeprom:r:-:h
        avrdude -q -q -p t2313 -c $(PGMER) -i 8 -U eeprom:r:-:b
        avrdude -q -q -p t2313 -c $(PGMER) -i 8 -U eeprom:r:-:d

size: main.elf all
        avr-size --format=avr --mcu=attiny2313 main.elf
```

Faisons un petit tour des cibles :

- **main.hex** : C'est notre cible principale où nous retrouvons les options utilisées en ligne de commande avec **avr-gcc** pour le compilateur et l'éditeur de liens. Rien de nouveau ici, si ce n'est l'intégration de la génération du fichier **main.map**. On retrouve également l'utilisation de **avr-objcopy** pour produire le fichier **main.hex** et, juste en dessous, la même commande pour obtenir un fichier **eeprom.hex** comprenant les valeurs à stocker dans l'EEPROM de l'AVR. Nous n'avons pas utilisé l'EEPROM intégrée dans l'Attiny2313 dans notre code, mais cela pourra vous être utile en d'autres circonstances. La libc pour AVR intègre un certain nombre de fonctions utilitaires concernant ce point. Je vous laisse parcourir le contenu du fichier **avr/eeprom.h**, ainsi que le site officiel (http://www.nongnu.org/avr-libc/) pour plus d'informations. Notez les options de la commande **avr-objcopy** permettant, entre autres, de réaligner les données pour une utilisation avec AVRdude (**--change-section-lma .eeprom=0**).
- **dprog** : Comme *Device PROgramming*. C'est ici que nous appelons **avrdude** en utilisant le programmeur in-situ en place et en récupérant le port à utiliser dans le fichier **~/.avrdude**.
- **rd_fuses** : Cette cible permet de lire les fuses de l'AVR. Notez l'utilisation de deux lignes de commande distinctes. Normalement, il est possible de spécifier les deux ordres **hfuse:r:-:b** et **lfuse:r:-:b** en une seule commande **avrdude** (deux occurrences de l'option **-U**). Il semblerait que le programmeur USB compatible ait un peu de mal sur ce point. Le problème ne se pose pas avec mon adaptateur **DAPA**. Le **b** en fin d'ordre précise un affichage en binaire, ce qui, pour la lecture des fuses, est bien plus pratique en termes de compréhension.
- **wr_fuses** : Écriture des fuses. Avec le code exemple, la question de la programmation des fuses ne se pose pas. Cette cible est placée là pour une utilisation avec un montage plus évolué utilisant un quartz comme source d'horloge et la désactivation de la division par 8 de la fréquence. Les valeurs sont spécifiées *in*line.
- **eeprom** : C'est la cible permettant de charger les données dans l'EEPROM de l'AVR. Eh oui, AVRdude permet aussi bien la programmation de la mémoire flash du microcontrôleur que de son EEPROM.
- **reeprom** : Il est parfois nécessaire de récupérer les valeurs dans l'EEPROM du composant. Ici, c'est principalement dans un but d'analyse et de vérification, d'où l'utilisation de trois commandes permettant l'affichage (pseudo fichier **-**) dans trois formats différents (hexadécimal, binaire et décimal). Ceci s'avère fort utile lorsqu'on commence à jouer avec l'EEPROM et les fonctions dédiées de la libc.
- **size** : Dernière cible que l'on aurait pu ajouter dans la phase de génération du fichier au format Intel Hex : l'affichage de la taille et de l'occupation de la mémoire de l'AVR. Encore une fois, la question ne se pose pas ici avec notre code de démonstration. Dans d'autres circonstances, il est fort probable que vous risquiez de jouer du **make size** assez souvent. En particulier, au moment où vous vous rendrez compte qu'il est possible de stocker des données directement dans la mémoire flash et de les lire depuis votre code.

Ce **Makefile** est destiné à servir de base de travail à vos projets. Il peut être intéressant d'imaginer une division des instructions dans un fichier commun pour ne garder dans le **Makefile** d'un projet que la partie variable. Ceci se fera très facilement en utilisant la directive **include** dans votre **Makefile** local.

Conclusion

Nous arrivons au terme de cet article d'introduction. Vous devez être en mesure maintenant de poursuivre seul vos expérimentations autour des composants AVR 8 bits d'Atmel. Je vous conseille de vous équiper avec plusieurs AVR de familles différentes afin de pouvoir explorer à loisir tous les types de périphériques intégrés. Personnellement, je travaille principalement avec quelques modèles dont l'Atmega8, l'Attiny2313 et l'Attiny13. Ces trois composants offrent des avantages et des fonctionnalités différents tout en restant très économiques. Mon domaine de prédilection se résume généralement à la communication série ou SPI et... au clignotement de leds. Ne riez pas, il est possible de faire des choses fantastiques avec 2, 16, 64 ou 192 leds connectées à un AVR lui-même branché à un PC par un port USB/série. Mais, chacun son terrain de jeu. Peut-être serez-vous davantage séduit par les écrans LCD, les capteurs en tous genres ou la construction d'un AVR communiquant sur Ethernet. Il n'y a que deux limites, celle des fonctionnalités de l'AVR choisi et, surtout, celle de votre imagination.

Auteur : Denis Bodor

VOS PHOTOS, FILMS, MAILS ET DOCUMENTS SONT-ILS EN SÉCURITÉ ?

LINUX PRATIQUE 54

PROTÉGEZ VOS DISQUES DES REGARDS INDISCRETS !

AU SOMMAIRE...

CONTENU DU CD :
5 Fedora 11 « Leonidas » (préversion). Distribution live et installable, version GNOME 2.26.

DÉCOUVERTE :
8 Réalisez vos screencasts avec RecordMyDesktop
11 Une solution pour réaliser et programmer aisément vos sauvegardes
14 Personnalisez le démarrage de votre système
15 Guake, un terminal à portée de main
16 Créez des applications ludo-éducatives avec JClic Author
22 Envie de réaménager votre intérieur ? Rencontre avec le développeur de Sweet Home 3D
24 L'artiste tortue – 1/2

MATÉRIEL :
28 Bien démarrer avec Linux Linpus Lite sur Acer Aspire One
31 Installer une tablette Aiptek avec Ubuntu 9.04

AUDIO/VIDÉO :
32 2ManDVD, le DVD Maker nouvelle génération !

OUTILS INTERNET :
36 Extensions de Firefox : notre sélection
41 Ubiquity – La nouvelle façon d'interagir avec le web

BUREAUTIQUE :
44 Créez des étiquettes en un tour de main !
46 Définir une autocorrection dans OpenOffice.org

GRAPHISME/3D/PHOTO :
48 Deux ou trois originalités de XaraLX

CONFIGURATION :
54 En couverture : Vos photos, films, mails et documents sont-ils en sécurité ? PROTÉGEZ VOS DISQUES DES REGARDS INDISCRETS ! Comprenez et utilisez le chiffrement grâce à Truecrypt
59 Quoi de neuf Dock !?... Des tiroirs pour Window Maker !
62 Installer Ubuntu avec LVM

SOLUTIONS PROFESSIONNELLES :
67 Dolibarr : une plateforme collaborative pour gérer une petite entreprise ou une association

CAHIER DU WEBMASTER :
70 Ajax simplifié avec Prototype
74 Programmez en PHP avec le ZendFramework : les bases
78 Des pop-up 100% CSS
80 Piwigo 2.0 : exposez vos photos sur le Web !

ENCORE DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 28 AOÛT 2009

Un éthylomètre pour s'authentifier

Auteur

- Denis Bodor

GNU/Linux, comme d'autres systèmes Unix, utilise le système PAM pour modulariser toutes les fonctions d'authentification. Ajouter une méthode pour s'assurer de la validité d'une action revient simplement à ajouter un module et l'utiliser pour un service spécifique. Pourquoi ne pas tester le taux d'alcoolémie de l'utilisateur ? Après tout, un utilisateur saoul est un véritable danger pour le système.

Avant d'entrer dans le vif du sujet, précisons que ce montage a fait l'objet d'une présentation (RUMP) à SSTIC 2009 par votre serviteur et Sébastien Tricaud, créateur de Picviz entre autres choses. Les explications qui vont suivre seront, bien entendu, plus détaillées que les 4 malheureuses minutes de présentation éclair (qui dans les faits ont été plus proches des 15 minutes). Derrière l'aspect ludique du projet se cache la mise en œuvre d'une des fonctionnalités les plus utiles des microcontrôleurs : la conversion analogique vers numérique reposant sur l'utilisation d'un ADC (Analog-to-Digital Converter) ici intégré directement au microcontrôleur.

1 Choix des composants

Le composant de base choisi sera un Atmel AVR Atmega8 disposant de 8Ko de mémoire Flash, 1Ko de RAM, 512o d'EEPROM (non utilisée ici) et surtout d'un ADC 6 canaux intégré. De plus, l'Atmega8 dispose d'un USART (*Universal Synchronous and Asynchronous serial Receiver and Transmitter*) lui permettant de communiquer facilement via une liaison série asynchrone. De quoi le connecter à un PC avec un convertisseur USB/série. Notez au passage que les AVR intègrent très souvent une SPI (*Serial Peripheral Interface*) permettant une communication série synchrone, mais pas nécessairement d'USART. Il nous faut donc utiliser un AVR contenant ADC et USART, ce qui écarte de fait le très économique Attiny2313. Nous parlerons de la SPI dans d'autres articles de GLMF ;)

L'autre composant important du montage est le capteur. Le modèle utilisé est un MQ-3 fabriqué par HANWEI ELETRONICS acquis auprès du détaillant SparkFun. Il s'agit d'un composant relativement simple constitué d'un élément chauffant et de deux électrodes connectées à un tube de céramique en alumine (Al2O3) recouvert dioxyde d'étain (SnO2). Ce dernier possède une propriété particulière : sa conductivité augmente en fonction de la présence d'alcool éthylique. L'éthanol, ou alcool éthylique, est l'alcool qui se retrouve dans les boissons et qui, par conséquent, est propre à la consommation contrairement à d'autres alcools comme le méthanol qui est toxique (voire mortel). Afin de garantir le bon fonctionnement du capteur, celui-ci doit être porté à une certaine température pour obtenir de bonnes conditions de fonctionnement du dispositif.

Note

Certains microcontrôleurs, plus « petits », disposent uniquement d'un comparateur analogique (Attiny2313 par exemple). Il ne s'agit pas ici de mesurer directement une tension (la donnée analogique), mais de comparer celle-ci à une tension de référence pour obtenir une réponse binaire (oui/non). Il est possible de contourner ainsi l'absence d'ADC au prix de quelques astuces relativement pénibles à mettre en œuvre (variations de la tension de référence et tests successifs).

sur le système : le montage

Nous avons utilisé ici un capteur répondant à un certain niveau d'alcool dans l'air, mais les explications qui vont suivre peuvent être appliquées à toutes sortes de capteurs non intelligents (dioxyde de carbone, humidité, ozone, température, niveau de luminosité, gaz GPL, gaz naturel, champ électrique, etc.). Dans la plupart des cas, il suffit d'injecter du courant et de mesurer la tension en sortie, précisément ce que permet de faire l'ADC.

2 Mise en œuvre

Le circuit de ce montage est relativement simple. L'Atmega8 ayant été choisi pour son ADC et sa capacité à communiquer via un port série, le reste de ses fonctionnalités et de ses broches n'est pas utilisé. Ce qui nous intéresse ici ce sont :

- Les broches 2 et 3 correspondantes respectivement aux lignes RXD (réception série) et TXD (émission série).
- Les broches 23 à 28 libellées ADC0 à ADC5 correspondantes aux 6 canaux de mesure de l'ADC. Ici, un seul canal nous sera utile pour connecter le capteur.
- Les broches 9 et 10 permettant de connecter un quartz de 14.7456 MHz servant de source d'horloge. L'Atmega8 dispose, comme toute la série AVR 8bit d'un oscillateur interne. Cependant, celui-ci ne permet pas une précision optimale pour une communication série. La *datasheet* intègre un certain nombre de tableaux détaillant les taux d'erreurs en fonction de la fréquence d'horloge (page 159 à 162).
- Pour le bon fonctionnement du convertisseur A/N, nous devons utiliser les pattes 20 et 21 pour y présenter la tension de référence. Dans le cas présent, il s'agit, tout simplement, de relier ces broches à VCC (7), puisque l'AVR est alimenté en +5 volts, tout comme le capteur MQ-3.
- Les broches 7 et 8 correspondent respectivement à la tension d'alimentation et la masse.
- La broche 1 correspond à la fonction de reset du microcontrôleur. Elle n'a pas besoin d'être reliée à VCC via une résistance de rappel au +5 volts. En effet, contrairement à d'autres modèles d'AVR, l'Atmega8 intègre déjà cette résistance de rappel comme le montre le schéma page 38 de la datasheet.
- La broche 15 peut être directement liée au comportement d'un circuit interne (*timer*) et change alors de nom de PB1 à OC1A. Ceci nous servira à avoir un retour visuel sur le fonctionnement de notre code et la prise de mesure.

Le circuit s'avère donc relativement simple et pourra facilement être testé sur une platine à essai (*breadboard*) avant d'être réalisé sur plaque pastillée ou via les méthodes classiques de réalisation de circuits imprimés.

Le schéma en figure 1 présente le circuit complet. Notez l'utilisation de condensateurs de filtrage permettant de stabiliser l'alimentation de l'AVR. Ceci s'avère indispensable lorsqu'on travaille à ces fréquences et via l'utilisation d'un quartz ou d'un quelconque autre oscillateur externe. De nombreux problèmes aléatoires relativement difficiles à cerner apparaissent en l'absence de ce type de filtrage. Ceci est tout aussi valable pour un montage communiquant via un port série que pour l'USB. Sachez, en effet, qu'il est parfaitement possible de faire communiquer un AVR « standard » sur un bus USB. Pour peu que le microcontrôleur dispose de ressources suffisantes, il est possible de gérer le protocole de manière entièrement logicielle (voir *GLMF* 100 p. 56).

3 Principe de fonctionnement

Le code pour l'AVR repose sur plusieurs éléments importants. Bien que les ressources mises à disposition par le microcontrôleur soient largement inutilisées, il est important de bien faire le tour des fonctionnalités utilisés pour les lecteurs faisant leurs premiers pas dans ce type de développement.

3.1 Port série

Comme spécifié précédemment, l'Atmega8 intègre ce qui est décrit sous le nom USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) dans la documentation du fabricant. Il s'agit d'un périphérique de communication extrêmement configurable. L'objectif ici est son utilisation en mode asynchrone à la manière d'un port série de PC. Attention, bien que le protocole de communication soit similaire, il n'en va pas de même pour les broches (ou lignes) en présence et surtout les tensions utilisées. Hors de question donc de connecter les deux lignes RX (réception de données) et TX (transmission de données) directement sur les broches correspondantes d'un port RS-232 (si vous en disposez encore sur votre machine récente). On travaille ici en TTL avec des niveaux logiques entre 0 et 5 volts par opposition au RS-232 qui fait usage de +12/-12 volts. Un montage simple à base de transistors et de résistances permet de faire la conversion assez facilement, mais nous avons choisi ici une solution encore plus simple : le convertisseur USB/Série TTL. Celui-ci se présente sous la forme d'un nouveau port série pour le système hôte, mais utilise des tensions TTL. Ce type de produit se trouve un peu partout sur le Web pour quelques euros avec une préférence pour les sites de ventes aux enchères en ligne proposant souvent des productions en provenance de pays de l'Est ou d'Asie.

L'utilisation de l'USART dans le code AVR est relativement simple et se passe en plusieurs étapes :

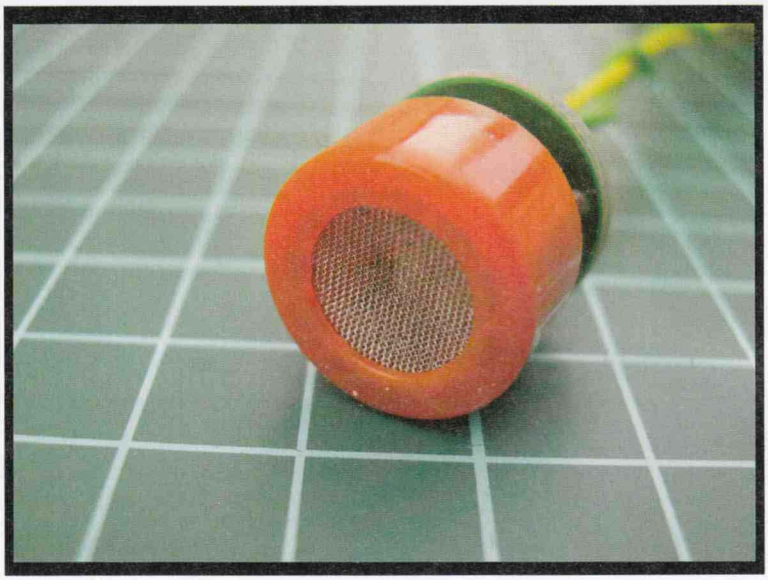
- Initialisation de l'USART via des registres spécifiques de configuration (**UBRRL**, **UCSRB** et **UCSRC**).
- L'envoi de données sur le port via l'utilisation d'un registre dédié (**UDR**) après vérification de la possibilité de le faire.
- La réception de données, toujours via le registre **UDR**, mais par l'utilisation d'une interruption spécifique.

Dans les deux derniers cas, nous utiliserons des fonctions spécifiques permettant de simplifier l'écriture et la réutilisation du code. Mais, commençons par le début, activer et configurer l'USART via les registres :

- **UBRRL** est la partie basse du registre de 16 bits UBRR (*USART Baud Rate Register*) qui détermine la vitesse de communication du port. Un tableau dans la datasheet donne les valeurs correspondantes en fonction de la fréquence d'horloge du microcontrôleur.
- **UCSRB** pour *USART Control and Status Register B*, est l'un des registres de configuration du port. Ce qui nous intéresse ici prend la forme des bits **RXEN** (activation de la réception), **TXEN** (activation de l'envoi) et **RXCIE** (le déclenchement d'une interruption en cas de réception). Les deux premiers bits feront passer les broches PD0 et PD1 de leur fonctionnement classique (port D) au mode USART.
- **UCSRC** pour *USART Control and Status Register C* détermine, entre autres choses, le format de données sur le port série. Ce registre est un peu particulier, car il partage le même point d'accès (port) que le registre UBRRH. Un bit particulier, **URSEL**, doit impérativement être écrit à 1 lorsqu'on souhaite accéder au registre **UCSRC**. Dans le cas contraire, on écrira dans **UBRRH**. L'autre bit qui nous intéresse est **UCSZ0** qui permet, en compagnie d'**UCSZ1** et **UCSZ2**, de choisir la taille des données. Ici, nous voulons travailler en 8N1 (8 bits de data, pas de parité et un bit de stop). Les valeurs par défaut des autres bits du registre correspondent déjà à nos intentions. Pour information, **USBS** détermine le nombre de bits de stop, **UPM0** et **UPM1** définissent la parité.

Côté envoi de données (post-initialisation donc) la procédure est relativement simple. Nous nous assurons que le registre **UDR** est disponible en écriture, puis, lorsque c'est le cas, nous y plaçons notre donnée qui sera envoyée automatiquement. La vérification tient en la lecture d'un simple bit, **UDRE** dans le registre **UCSRA**, le registre A de contrôle et d'état de l'USART.

La réception est plus complexe. Nous avons configuré la génération d'une interruption lors de la réception. Un vecteur d'interruption est donc associé et nous n'avons qu'à positionner notre routine à cet endroit. Une macro permet cela sans aucune difficulté. Lors de l'interruption, le registre **UDR** contient la donnée reçue, mais il ne s'agit que d'un seul caractère. Nous souhaitons travailler un peu à la manière d'un modem ou d'une console. Il nous faut récupérer des lignes complètes (habituellement terminées pas `\r`) et, faisons les choses correctement, nous affichons un écho des données envoyées (fort pratique en phase de test). Lors de l'interruption, nous traitons donc le caractère en le plaçant dans un *buffer* et en incrémentant un compteur. Lorsque nous recevons `\r`, nous ajoutons **0x00** en fin de buffer, nous activons un flag propre à notre code permettant de signaler qu'une chaîne est disponible et nous désactivons la génération d'une interruption lors de la réception. Ceci nous permet de ne plus accepter de nouveau caractère tant que la chaîne disponible n'aura pas été traitée.



3.2 ADC

Tout comme pour l'USART, l'ADC doit être configuré avant utilisation. Les broches 23 à 28, respectivement PC0 à PC5 en mode de fonctionnement classique (bit 0 à 5 du port C), deviennent ADC0 à ADC5 en activant le bit **ADEN** du registre **ADCSRA** (*ADC Control and Status Register A*). Il faut également configurer, dans ce même registre, les bits ADPS0 à ADPS2 définissant le *prescaler* pour l'ADC. La datasheet est très explicite de ce point de vue : pour un fonctionnement optimal, l'ADC doit fonctionner à une fréquence d'échantillonnage entre 50 Khz et 200 Khz. Notre AVR fonctionnant à plus de 14 Mhz, nous configurons les bits pour une division de la fréquence système, le prescaler, par 128 (**111**), qui nous donne un peu moins de 100 Khz. C'est un point important à prendre en considération pour d'autres montages. Ici, nous utilisons un oscillateur à quartz nous permettant de faire fonctionner l'USART avec un taux d'erreur nul. Dans d'autres cas, l'Atmega8 pourra fonctionner avec son oscillateur interne à 1 Mhz par exemple (configuration par défaut).

Ce n'est pas tout. L'ADC peut fonctionner, à l'instar de l'USART, en utilisant une interruption. Le temps de conversion n'est pas nul. Lorsqu'on déclenche une mesure, il faut attendre la fin de la conversion pour obtenir le résultat. Il faut donc activer l'utilisation de l'interruption avec le bit **ADIE** du registre **ADCSRA**. Deux modes de fonctionnement sont disponibles pour la conversion. Soit nous désirons une conversion ponctuelle, auquel cas notre configuration s'arrête là et nous pouvons déclencher arbitrairement une mesure en positionnant le bit **ADSC**, soit nous utilisons le mode Free Running. Dans ce cas, le fait de positionner **ADSC** déclenchera une première conversion qui, une fois terminée en provoquera une seconde et ainsi de suite. Ici, nous préférons utiliser le premier mode. Nous déclencherons une conversion de manière récurrente, mais avec l'aide d'un timer préalablement configuré.

Le schéma de fonctionnement est donc simple. Nous déclenchons une conversion, puis traitons la donnée dans la routine d'interruption qui signalera, via une variable globale, la disponibilité des données. Une fois la donnée traitée, nous réinitialisons la variable globale et la routine de timer lancera une nouvelle conversion. Afin de signaler visuellement ces multiples demandes de conversion, nous

associons au timer un basculement d'état sur une broche de l'AVR. Une *led* nous permettra donc de voir quand une demande est faite. De plus, les leds clignotantes sont toujours du plus bel effet sur un montage.

3.3 Timer

Voilà qui nous permet d'enchaîner sur le dernier élément de configuration, le timer. L'Atmega8 dispose de 3 timers programmables et d'une souplesse d'utilisation exemplaire. Nous ferons usage ici du timer 1 dont le fonctionnement peut être associé à la broche OC1A (PB1).

Nous ne nous étendrons pas sur la configuration du timer relativement simple par définition. Celui-ci sera configuré via les registres **TCCR1A** et **TCCR1B** pour un basculement de la broche en cas de comparaison vérifiée (bit **COM1A0**) et avec une remise à zéro dans cette même situation (bit **WGM12** anciennement **CTC** pour *Clear Timer on Compare match*). Nous configurons également le bit **OCIE1A** du registre **TIMSK** pour déclencher une interruption dans ce cas. Un prescaler **/256** est configuré pour le timer grâce aux bits **CS10** à **CS12** de **TCCR1B**. Ceci fait, nous n'avons plus qu'à renseigner le registre de 16 bits **OCR1A** pour placer notre indice de comparaison.

Nous avons donc un AVR qui fonctionne à 14745600 Hz avec un prescaler de 256 qui nous donne une incrémentation du compteur/timer à 57,6 Khz. Nous plaçons la valeur **0x7FFF** dans **OCR1A** et allons donc avoir une comparaison vérifiée tous les 32767 « pas » avec une remise à zéro du compteur/timer, soit 57600/32767 hertz = 1.75 Hz. En d'autres termes, une interruption toutes les 0,56 secondes. En jouant sur la valeur de presacler et celle d'**OCR1A**, nous pouvons faire varier de délai comme bon nous semble. Du fait de l'utilisation de **WGM12**, le compteur ne débordera donc jamais.

4 Code et programmation

Il est maintenant temps d'entrer dans le vif du sujet avec un peu de code. Notre *firmware* est très orienté vers les interruptions. Le code de la fonction **main** s'en trouve donc simplifié. Notez que le code présenté ici se veut délibérément didactique et qu'il est possible, bien entendu, d'optimiser l'ensemble aussi bien d'un point de vue de la configuration que de l'algorithmique.

4.1 Configuration

Nous débutons par la configuration du timer 1.

```
// broche PB1 en sortie
DDRB |= (1<<PB1);
// arrêt du timer
TCCR1B = 0;
// OC1A bascule on Compare Match
TCCR1A = (1<<COM1A0);
// Timer 1 Output Compare Register A
// Valeur de seuil
OCR1A = 0x7FFF;
// interruption sur comparaison
TIMSK = (1<<OCIE1A);
// prescaler à /256 et mise à zéro (CTC)
TCCR1B |= (1<<CS12)|(0<<CS11)|(0<<CS10)|(1<<WGM12);
```

Les commentaires dans le code parlent d'eux-mêmes. Le seuil (**OCR1A**) est ici fixé à la moitié de la taille du registre de 16 bits, ce qui conduit au même résultat qu'un prescaler **/128** avec un débordement de compteur. Ceci permettra un ajustement plus facile de la fréquence de mesure du capteur. On peut éventuellement envisager l'utilisation d'un autre canal ADC pour y brancher un potentiomètre et ainsi régler cette fréquence sans avoir à toucher au firmware. Enfin, solution plus simple, rien n'interdit de définir la valeur de **OCR1A** via une commande reçue par le port série.

Nous passons ensuite à la configuration de l'ADC :

```
ADMUX = 0;
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
```

ADMUX est un registre de configuration dont nous n'avons pas parlé. Quatre bits permettent le choix du canal ADC à utiliser. Les autres bits permettent de choisir la tension de référence pour la mesure et la manière de présenter le résultat de la conversion. La valeur par défaut de **0** convient ici parfaitement. Cette ligne n'est présente dans le code que pour d'éventuels ajustement en cas d'évolution du circuit.

Dernière étape de configuration, le port série :

```
InitUART (47);
```

Nous choisissons ici d'utiliser une fonction spécifique. C'est un simple choix pratique permettant de faciliter la réutilisation du code. De plus, la valeur passée définit à elle seule la vitesse de communication en fonction de la fréquence d'horloge. Nous pouvons donc très rapidement changer cette valeur lors des tests. La fonction **InitUART()** est implémentée comme ceci :

```
void InitUART (unsigned char baudrate) {
  // Vitesse p. 161 datasheet pour 14.7456 MHz
  UBRRL = baudrate;
  // réception et émission
  UCSRB = (1 << RXEN) | (1 << TXEN);
  // Format 8N1
  UCSRC = (1<<URSEL)|(3<<UCSZ0);
  // Interruption sur RX
  UCSRB |= (1 << RXCIE);
}
```

Notez que nous n'utilisons que le registre **UBRRL**, la partie basse du registre de 12 bits **UBRR**. En d'autres termes, des valeurs comme 383 ne peuvent être utilisées avec cette fonction (vitesse de 2400 bps avec un MCU à 14.7456 MHz).

Avant d'entrer dans la boucle principale, nous affichons un petit message sur le port série (voir plus loin dans l'article), activons les interruptions et lançons une première conversion :

```
printstr_p(PSTR("== EthyloSSTIC v0.1 ==\n"));
sei();
ADCSRA |= (1 << ADIE);
ADCSRA |= (1 << ADSC);
```

4.2 Interruptions

Commençons par l'interruption provoquée par le timer ou plus exactement la comparaison vérifiée du compteur incrémenté :

```
ISR(TIMER1_COMPA_vect) {
  if(a_ready) {
    ADCSRA |= _BV(ADIE);
    ADCSRA |= _BV(ADSC);
    a_ready=0;
  }
}
```

Nous testons simplement la valeur de **a_ready**. Si celle-ci est non nulle, c'est tout simplement que la précédente conversion est terminée et nous pouvons en lancer une nouvelle. Chose que nous provoquons en utilisant les bits **ADSC** et **ADIE** de **ADCSRA**. Ceci fait, nous n'oublions pas de rendre la valeur actuelle non valide en réinitialisant **a_ready** à 0.

Puisque nous en sommes à parler de conversion, traitons de l'interruption associée :

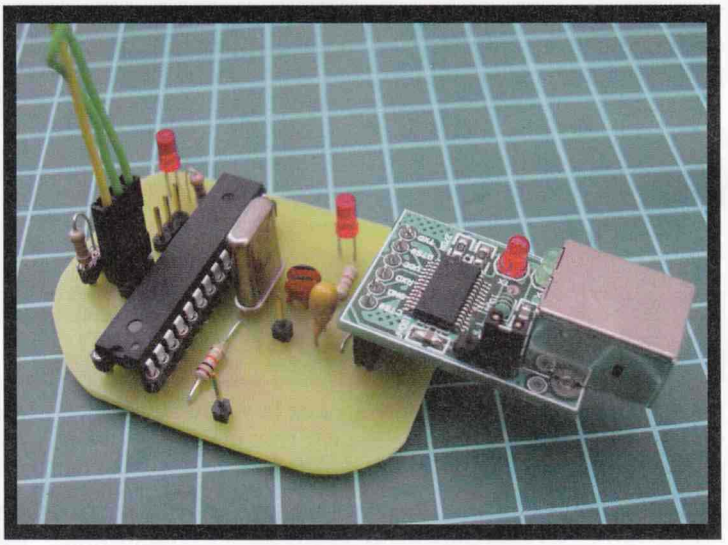
```
ISR(ADC_vect) {
  adcval = ADCW;
  a_ready=1;
  ADCSRA &= ~_BV(ADIE);
}
```

Cette interruption n'intervient que lorsque la conversion est terminée. Nous récupérons donc la valeur mesurée depuis le registre **ADCW** dans une variable globale appelée **adcval** traitée par ailleurs. Nous signalons la présence d'une valeur valide via **a_ready** et nous désactivons la génération d'interruption en jouant sur le bit **ADIE**. Nous pouvons enfin en venir à la routine d'interruption concernant la réception de données sur le port série (USART). Celle-ci sera appelée pour chaque caractère reçu, vérifié ou non (bits de données, parité, bit de stop) :

```
ISR(USART_RXC_vect) {
  char c;
  c = UDR;
  if (bit_is_clear(UCSRA, FE)) {
    if (c != '\r' && r_index < 47) {
      putchr(c);
      r_buffer[r_index++]=c;
    } else {
      putchr('\r');
      putchr('\n');
      r_buffer[r_index]=0x00;
      r_ready = 1;
      UCSRB &= ~_BV(RXCIE);
    }
  }
}
```

Le caractère obtenu est stocké dans **UDR**. Nous testons tout d'abord le bit **FE** (*Frame Error*) pour nous assurer de traiter une donnée valide. Si le bit est à 0, nous pouvons opérer. La fonction **putchr** dont nous parlerons plus loin permet simplement d'avoir un écho. Nous renvoyons le caractère obtenu s'il ne s'agit pas d'un retour chariot (**\r**) marquant la validation d'une commande. Nous plaçons également le caractère en question dans **r_buffer[]** en incrémentant une variable d'index, **r_index**. Vous comprendrez tout l'intérêt de la variable **c** recevant la copie de **UDR**, le registre étant également utilisé pour l'envoi sur le port série.

En cas de réception de **\r**, nous renvoyons un retour chariot et un caractère « nouvelle ligne » (**\n**) avant d'ajouter **\0** (**0x00**) en fin de buffer de réception. **r_ready**, tout comme **a_ready**, est utilisé comme drapeau marquant la disponibilité de données valides. Enfin, nous désactivons l'interruption en réception sur le port série. Tant que la donnée n'est pas traitée d'une manière ou d'une autre, l'interruption est masquée et **r_ready** reste à 1. Notez au passage que la taille de **r_buffer** est déterminée globalement et que l'espace RAM est limité. Il est donc normalement impératif de contrôler la masse de données stockée dans **r_buffer** pour éviter le *buffer overflow*. Celui-ci se traduit sur AVR comme avec d'autres architectures par un écrasement des données situées en RAM après la dernière position du buffer, ce qui se traduit généralement en comportements étranges, plantages et longues nuits de débogage.



4.3 Fonctions pour le port série

Afin de simplifier le traitement des envois sur le port série de l'AVR, nous allons nous attacher à écrire quelques routines réutilisables. L'envoi de données via l'USART est extrêmement simple, puisqu'il suffit de placer les caractères à émettre dans le registre **UDR** après avoir vérifié l'état du bit **UDRE** (*USART Data Register Empty*) signalant la disponibilité du registre. Notre première fonction destinée à émettre un caractère se résume donc à ceci :

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

loop_until_bit_is_set est une simple macro définie dans **avr/sfr_defs.h** incluse par les fichiers d'en-tête usuels. Grâce à notre fonction **putchr()**, nous pouvons construire tout aussi simplement une fonction permettant l'envoi d'une chaîne :

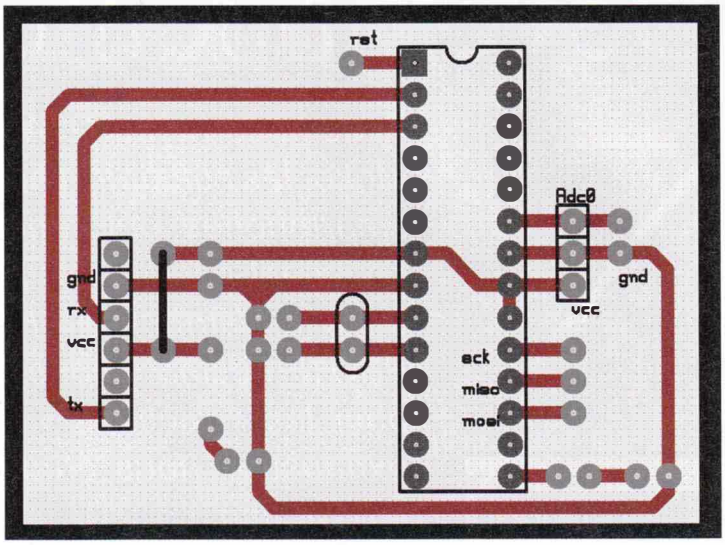
```
static void printstr(unsigned char *s) {
  while (*s) {
    putchr(*s++);
  }
  putchr('\r');
  putchr('\n');
}
```

Celle-ci prend en argument un pointeur sur une chaîne et boucle jusqu'à arriver sur **\0**. En fin de chaîne, nous ajoutons le retour chariot et le passage à la ligne (**\r\n**). Cette fonction comporte cependant un problème : sa consommation de RAM, une denrée rare sur l'Atmega8 comme avec tout microcontrôleur. Nous n'avons pas nécessairement besoin d'afficher des données susceptibles de changer. Inutile donc pour le compilateur de réserver l'espace en RAM pour tous nos envois. Nous pouvons utiliser la mémoire Flash pour stocker nos chaînes. Dans ce sens, nous écrivons donc une autre fonction :

```
static void printstr_p(const char *s) {
  char c;
  for (c = pgm_read_byte(s); c; ++s, c = pgm_read_byte(s))  {
    if (c == '\n')
      putchr('\r');
    putchr(c);
  }
}
```

La fonction prend en argument un pointeur sur une chaîne placée en flash : **printstr_p(PSTR("ma chaîne\n"))**. On parcourt ensuite tout simplement la chaîne en question via **pgm_read_byte()** jusqu'au **\0** fatidique. Si le caractère lu est **\n**, nous plaçons tout d'abord un **\r** avant l'envoi via **putchr()**.

Voilà, nous disposons maintenant de tous les éléments pour construire notre programme AVR et nous pouvons passer au cœur du code : la boucle principale.



4.4 Fonction main

```
volatile unsigned char r_index;
char r_buffer[48];
char itoa_buffer[8];
volatile unsigned char r_ready=0;
volatile uint16_t adcval;
volatile unsigned char a_ready=0;
int main (void) {
  char cmd_id[3]  = "id";
  char cmd_get[4] = "get";
  char cmd_rst[6] = "reset";
  while (1) {
    if(r_ready) {
      if(!strcmp(cmd_rst, r_buffer)) {
        soft_reset();
      }
      if(!strcmp(cmd_id, r_buffer)) {
        printstr_p(PSTR("ethylosstic\n"));
      }
      if(!strcmp(cmd_get, r_buffer)) {
        if(a_ready) {
          printstr((unsigned char *) itoa(adcval,
              itoa_buffer, 10));
        } else {
          printstr_p(PSTR("0"));
        }
      }
      gets_int();
    }
  }
}
```

Un certain nombre de fichiers d'en-tête sont nécessaires comme **<string.h>** pour la gestion des chaînes. Il faut faire très attention en utilisant les fonctions comme **strcmp()** et **itoa()**, car elles sont certes pratiques, mais consomment une quantité de flash non négligeable (à l'échelle d'un microcontrôleur).

La boucle principale en elle-même se contentera d'analyser une chaîne de caractères reçue par le port série dont la disponibilité est signalée par **r_ready**. Nous disposons ainsi de trois commandes. **id** retourne une chaîne afin de permettre l'identification du périphérique. Dans l'absolu, un numéro de série serait ajouté pour éviter toutes confusions en cas d'utilisation de plusieurs montages sur une seule machine. **get** nous retournera simplement la valeur mesurée par l'ADC sur le capteur MQ-3.

Nous avons également ajouté une commande **reset** qui est plus un *proof of concept* qu'autre chose, ce type de fonctionnalités n'étant pas d'une grande utilité dans ce cas précis. Il n'existe pas de solution intégrée permettant un reset de l'AVR par le code en Flash. Il est en effet possible de configurer le microcontrôleur pour un reset automatique en cas de baisse de tension d'alimentation (*Brown out detection*) ou un *reset hard* par utilisation de la broche dédiée (1, PC5). Pour obtenir une fonction de *soft-reset*, nous jouons sur le *watchdog* avec la macro :

```
#define soft_reset()        \
do                          \
{                           \
      wdt_enable(WDTO_15MS); \
      for(;;)               \
      {                     \
      }                     \
} while(0)
```

Nous activons le watchdog et partons dans une boucle **for** infinie. Conséquence directe, nous ne remettons pas à zéro le compteur du watchdog et un reset est donc automatiquement provoqué au bout de 15ms.

Conclusion

La partie matérielle s'arrête là. Du côté hôte (PC), il nous suffit d'utiliser, dans un premier temps, un outil comme **screen** ou **minicom** pour dialoguer avec le montage et procéder aux premiers essai. Ensuite, nous pouvons envisager le développement d'un module PAM relativement simple puisqu'il se contentera de lire la valeur retournée par la commande **get** émise sur le port série. Comme nous venons de le voir, la construction de ce type de montage est avant tout une histoire de rigueur. Comme avec Perl, il existe souvent plus d'une manière d'envisager le code pour un AVR. Le nombre de fonctionnalités disponibles (timers/compteurs, interruptions, etc.) vous offre un grand nombre de libertés. Il ne vous reste finalement plus qu'à choisir la méthode et la structure qui vous conviennent le mieux.

Auteur : Denis Bodor

Développement embarqué

Avec la sortie du GTA01 et du Freerunner (GTA02) d'OpenMoko, les solutions libres pour embarqué, et particulièrement smartphones, ont franchi un pas de plus. Nous étudierons ici comment développer pour ces systèmes en prenant comme base la distribution SHR (Stable Hybrid Release), basée sur FSO (Freesmartphone.Org) en middleware et OpenEmbedded pour les « recettes » de génération de paquets. Afin d'être exhaustifs, nous étudierons également Hackable:1, basée sur Debian.

Auteurs

- Jérome Blondon,
- Julien « Ainulindalë » Cassignol
- Dr Michael « mickeyl » Lauer

1 Contexte

Nous voyons tous les jours, dans nos métiers et dans nos usages, se démocratiser l'utilisation du libre dans nos divers périphériques et logiciels, pour certains d'entre nous avec joie, pour d'autres avec récalcitrance, au vu de la maturité de certaines solutions. Il n'en reste pas moins qu'un fait est patent : au moment où nous voyons de plus en plus des solutions libres pérennes faire leur entrée chez de grands comptes, où des solutions libres font figure de standard de fait, technologiquement plus évoluées ou abouties que leurs concurrents, le milieu de l'embarqué reste encore un petit îlot paisible où il n'est pas aisé de déployer des logiciels libres.

Certes, il y a eu des expériences, par Nokia notamment, ou plus récemment Texas Instruments, mais ces exemples ne sont pas légion. La nécessité, ici, pour le développeur, l'utilisateur et le professionnel qui souhaitent une solution adaptée à son besoin, est de pouvoir avoir accès tant en termes de software que de hardware, au « cœur » de sa machine, afin d'être en mesure de pouvoir remplir son besoin au plus proche.

Cette tentative a partiellement été assurée par OpenMoko (modulo le *chipset* graphique, dit « Glamo », qui reste sous NDA), qui a donné les spécifications électroniques de son Freerunner, mais qui a également fondé des initiatives communautaires pour le développement d'une base logicielle permettant de profiter du concept de « libre » sur tout le système.

Nous considèrerons particulièrement dans cet article l'exemple de Freesmartphone.Org, qui, lorsque le projet a été lancé début 2008, faisait preuve d'un concept novateur et permet aujourd'hui l'exploitation de toutes les couches « de base » d'un périphérique embarqué. Depuis, de nouveaux challengers, tels qu'oFono, ont fait leur apparition, avec une cible quelque peu différente.

Afin de nous donner une base de réflexion plus simple à comprendre, nous considérerons également la distribution SHR (*Stable Hybrid Release*), dont un des auteurs de cet article a la joie d'être le « *benevolent dictator* ». Nous utiliserons donc les « aides » développées pour générer une image de cette distribution, tout en expliquant les commandes de base permettant de générer une image ou un paquet selon la méthode ancestrale.

En dernier lieu, nous considérerons également une alternative, Hackable:1. Elle est développée par Bearstech dans le cadre de son activité hackable:devices. L'objectif est de développer des services innovants dans le domaine de la mobilité tout en mettant à la disposition des développeurs une plate-forme de développement rapide, simple et respectant les standards existants. Elle est basée sur Debian et Gnome mobile. Elle est également pensée pour tourner sur d'autres matériels que le Freerunner. Il s'agit d'une distribution assez jeune, puisque son développement a démarré fin 2008.

Pour toutes les distributions, il est à noter qu'il existe une communauté francophone très active, dont le point de rencontre est sur http://www.openmoko-fr.org.

sur GTA01 et GTA02

2 OpenEmbedded

OpenEmbedded est un système englobant un ensemble d'outils et de recettes de génération de paquets, permettant de construire dans un contexte « embarqué » des distributions et images complètes.

2.1 Commencer simplement : le Makefile SHR

Afin de pouvoir mettre en place un environnement complet de travail avec OE (*OpenEmbedded*), il est nécessaire de configurer un certain nombre d'éléments. Cette configuration est relativement aisée lorsqu'on est habitué à OE et sa logique, mais, pour un débutant, cela peut paraître très obscur : la barrière d'entrée est malheureusement assez haute.

En considérant ce problème, un certain nombre de solutions « *packagées* » fournissent un ensemble de primitives qui vous simplifieront la vie. Considérons donc ici le Makefile SHR, basé sur le MokoMakefile de Rod Whitby.

```
$ wget http://build.shr-project.org/Makefile
$ make setup
```

Cette dernière commande va vous permettre de remettre à jour le Makefile en fonction de la dernière version sur git **[SHRGIT]**, ainsi que de télécharger et installer un arbre complet OpenEmbedded (basé sur la branche spécifique à SHR, **shr/import**, que vous pouvez trouver en ligne **[OEGIT]**), ainsi que l'outil **bitbake** et un ensemble de scripts facilitant l'utilisation de ce dernier. Deux principaux dossiers seront créés : **shr-testing** et **shr-unstable**, correspondant aux deux versions majeures actuelles de la distribution. Par la suite, dans cet article, nous considèrerons **shr-unstable**.

Il nous reste à adapter un peu cette structure de base en modifiant la configuration par défaut. Éditons donc le fichier **local.conf** se trouvant dans **shr-unstable/conf/local.conf**, et rajoutons les quelques paramètres suivants :

```
# Générer les locales française uniquement
GLIBC_GENERATE_LOCALES = "fr_FR.UTF-8"
# Si notre hôte est dual core ou multi processeur,
# cela permettra de compiler plus PARALLEL_MAKE = "-j 4"
rapidement les packages
# Idem
BB_NUMBER_THREADS = "4"
```

Si nous souhaitons simplement construire une image se basant sur les données et logiciels écrits par l'équipe SHR, nous nous contenterons alors de taper les commandes suivantes, à partir de là où nous avions téléchargé le **Makefile** :

```
$ cd shr-unstable
$ make image # Par défaut, image pour GTA02
```

Bien entendu, il vous faudra vous armer de patience et attendre que toutes les dépendances soient compilées pour une cible qui n'a pas la même architecture processeur que votre hôte, a priori (puisque ici, nous allons compiler pour ARM). À titre informatif, sur un PC « normal », en une nuit, on peut s'en sortir avec une image. Il est également important de considérer l'espace : avec les sources, et une certaine durée de vie, l'ensemble des fichiers de l'environnement de travail occupe 7,6 G sur l'ordinateur de l'auteur.

Éventuellement, pour les possesseurs de GTA01, il est possible de taper la commande suivante pour générer la configuration nécessaire pour une image sur ce dernier :

```
$ make setup-machine-om-gta01
```

Dès lors, afin de récupérer les changements *commités* sur le dépôt git SHR, et mettre à jour son ensemble de sources OE, il suffira de taper la commande suivante à la racine où nous avons téléchargé le **Makefile** :

```
$ make update
```

Dès lors, réitérer **make image** reconstruira une image à jour et correspondant à la dernière version des sources. Dépendant de votre architecture cible pour l'image, vous trouverez par défaut un fichier **jffs2** et **tgz**, ainsi qu'un *kernel*, dans **shr-unstable/tmp/deploy/glibc/images/**. Il vous suffira dès lors de flasher ou installer l'image et le kernel selon la méthode traditionnelle (expliquée plus bas).

Accessoirement, si vous souhaitez compiler l'ensemble du « *feed* » SHR, c'est-à-dire l'ensemble des paquets compilés sur le serveur hébergeant SHR, vous pouvez également utiliser les commandes suivantes :

```
$ make distro # Compile la tâche task-shr-feed
$ make index # Remet à jour l'index des paquets
```

Une fois que l'index est mis à jour, vous pouvez exporter votre « feed », disponible dans **tmp/deploy/glibc/ipk**, et l'utiliser comme un dépôt de paquet pour votre périphérique. Le plus simple est de le rendre accessible par HTTP, et de le rajouter aux sources de packages d'**opkg**.

2.2 Bitbake

bitbake est un outil assez similaire à portage (de Gentoo) et permettant de construire des packages et images à partir de recettes **.bb**. Il dispose d'un certain nombre de commandes vous permettant de manipuler complètement les recettes OE. Le **Makefile**, décrit précédemment, englobe ces commandes, mais ne vous permet pas d'avoir autant de granularité que l'outil standard. Les auteurs vous enjoignent à regarder le **Makefile** afin de pouvoir constater l'ensemble des possibilités, et de comparer avec **bitbake**.

Prenons l'exemple précédent, et considérons que nous souhaitons à présent construire une image avec **bitbake**. Dès lors, nous taperons les commandes suivantes, de la racine du **Makefile** :

```
$ cd shr-unstable
$ . setup-env
$ bitbake shr-image # shr-image est le nom de la recette bb générant l'image
```

Ceci permet de charger l'environnement précédemment défini par le **Makefile** (les variables d'environnement permettant de situer où **bitbake** doit trouver ses recettes **bb**, mais aussi où se trouvent les dossiers avec les binaires compilés pour ARM, etc.), puis de générer une image en prenant en compte toutes les dépendances.

Supposons que nous souhaitions à présent compiler un paquet spécifique qui n'est pas disponible dans l'image, par exemple **dictator**. Dans ce cas, nous taperons la commande suivante :

```
$ bitbake dictator
```

Une fois cette commande terminée, vous trouverez le paquet dans **tmp/deploy/glibc/ipk/ARCH/**, où **ARCH** correspond à l'**ARCH** générique ou spécifique selon votre paquet. Dans le cas de **dictator**, nous trouverons donc cela dans **tmp/deploy/glibc/ipk/armv4t**, correspondant à l'architecture générique pour périphériques à processeur armv4t.

Attention, n'oubliez pas qu'il est nécessaire avant d'utiliser **bitbake** d'être sûr d'avoir au moins effectué une fois la commande **. setup-env**, afin de définir les chemins nécessaires.

Supposons maintenant que nous souhaitions éviter de « *reparser* » à chaque fois l'ensemble des recettes, voire que nous nous fichions des dépendances d'un paquet, sachant qu'elles sont déjà là. Dans ce cas, nous utiliserons le paramètre **-b** de **bitbake**, qui nous permet de cibler une recette spécifique.

```
$ bitbake -b openembedded/recipes/dictator/dictator_0.2.bb
```

Cette commande effectuera l'ensemble de la procédure permettant de construire le paquet, à la condition où celui-ci ne l'a pas déjà été, c'est-à-dire, si les *stamps* que l'on peut trouver dans **tmp/stamps/ARCH** n'existent pas pour le paquet donné. Dans le cas contraire, **bitbake** reprendra où il s'était arrêté.

De fait, on peut donc demander à forcer ces actions (**-f**) ou, par exemple, à exécuter une action particulière. Il existe un ensemble fini d'actions, mais on peut écrire des procédures spécifiques dans les recettes **bb** permettant des les étendre. Parmi ces actions, on citera **fetch**, **unpack**, **patch**, **configure**, « **compile** », **install**, qui revêtent pour nous un intérêt particulier dans l'utilisation de **bitbake**. Si donc, je souhaite forcer la réexécution de la procédure de **configure** de **dictator**, j'aurais à effectuer la commande suivante :

```
$ bitbake -b openembedded/recipes/dictator/dictator_0.2.bb -f -c configure
```

Enfin, comme de bien nombreux programmes, **bitbake** vous fournit des éléments de *debug*, lorsque vous concevez des recettes **bb**. Il apparaît nécessaire de préciser que les options **-v** (**verbose**), **-D** (**debug**) sont indispensables lorsque vous vous posez des questions métaphysiques sur la raison qui fait que votre recette ne veut pas se packager, mais également **-e**, qui affichera l'environnement complet d'exécution de **bitbake**, et donc tous les **PATH**, **INCLUDE**, et autres : on n'est jamais à l'abri d'une mauvaise configuration et régulièrement des erreurs apparaissent dans certaines recettes. Il n'est donc pas inutile de contrôler.

Il n'est pas inutile non plus de préciser que la commande **-c devshell** appliquée à une recette vous lancera un terminal avec toutes les variables d'environnement nécessaires à pouvoir faire vos changements directement dans le répertoire de travail de **bitbake** : vous pourrez donc éditer les sources, et taper **make** comme vous le feriez normalement.

2.3 Recettes « bb »

Afin de nous « sensibiliser » à la création d'une recette **bitbake**, nous allons ensemble essayer de créer une recette pour le paquet d'une application « Hello World », que nous appellerons **helloworld-shr**.

L'ensemble des sources développés pour cet exemple se trouve sur le site de SHR : http://build.shr-project.org/tutorial

2.3.1 En-tête d'une recette

Considérons d'abord la structure d'une recette **bb**. Comme dans toutes les distributions, nous avons, au départ, des informations de qualification de la recette :

```
#Description du paquet, telle qu'apparaissant
# dans les recherches avec opkg
DESCRIPTION = "Hello World from SHR"
SECTION = "utils" # La section du paquet
LICENSE = "GPL" # La licence du paquet
PR="r0" # La révision du paquet
```

Il existe d'autres variables, telles que **PN** (nom du paquet), **PV** (version du paquet), qui sont initialisées d'après le nom du fichier : si ma recette **bb** s'appelle **helloworld-shr_0.1.bb**, alors **PN** = **helloworld-shr**, et **PV** = **0.1**. Au final, ces variables seront utilisées afin de qualifier si l'on doit ou non mettre à jour un paquet (si **PV-PR** est supérieur à l'ancien **PV-PR**).

Dans le cas de paquets récupérant des sources directement depuis un serveur de contrôle de version, il est nécessaire d'utiliser des conventions de nommage différentes. Par exemple, si, ici, **helloworld-shr** était directement téléchargé depuis git, la recette **bb** s'appellerait alors **helloworld-shr_git.bb**, et j'aurais à définir **PV** directement dans ma recette, comme suit :

```
PV = "0.1+gitr${SRCREV}"
```

Ici, **${SRCREV}** est une variable qui permet de retourner la révision au sens contrôle de version : dans le cas de git, ce sera le hash du commit (précédé de son numéro afin de garder un ordre lexicographique permettant de mettre à jour).

Deux autres variables sont indispensables dans la description d'une recette **bb** : ses dépendances. Il en existe deux types, les dépendances à l'exécution, et à la compilation. Considérons un programme qui ait besoin de **eet** à la compilation, et de **e-wm** et **ophonekitd** à l'exécution, on écrira comme suit :

```
DEPENDS += "eet"
RDEPENDS += "e-wm ophonekitd"
```

Notez qu'on utilise la notation **+=**, car selon les dépendances rajoutées, on peut avoir des compléments de dépendances à la compilation issus des dépendances à l'exécution, et vice-versa. Les noms inscrits ici en dépendance sont des noms de paquets pour lesquels il existe des recettes **bb**.

Ensuite, logiquement, nous allons indiquer où récupérer les fichiers à compiler.

```
SRC_URI = "http://build.shr-project.org/tutorial/${PN}-${PV}.tar.gz"
```

Ici, nous utilisons **${PN}** comme une partie du nom, c'est-à-dire **helloworld-shr**. Cette notation fait partie des recommandations sur OE. Dans le cas où nous considèrerons un gestionnaire de versions, cela pourrait ressembler à ceci :

```
SRC_URI = "git://git.shr-project.org/repo/ophonekitd.git;protocol=http;
branch=master"
```

Ici, on indique le chemin d'un dépôt git, et on indique que l'on récupèrera le source de la branche **master** via le protocole HTTP. **bitbake** se chargera de lui-même d'initialiser le dépôt local, de prendre les sources, et de les compresser.

Il est également possible d'indiquer plusieurs sources, ce qui revêt une grande utilité dans le cadre de patchs. Si, dans le cadre de mon exemple précédent, je souhaite modifier les sources de mon « Hello world » afin de pouvoir, que sais-je, dire « Goodbye », je peux réaliser un patch sur les sources, le mettre à disposition dans mon arbre OE, et l'indiquer dans les sources :

```
SRC_URI = "http://build.shr-project.org/tutorial/${PN}-${PV}.tar.gz \
                file://hello2goodbye.patch;patch=1"
```

Dès lors, cela signifie que le fichier **hello2goodbye.patch** doit se trouver dans un dossier correspondant au nom de la recette **bb**, au même niveau que celle-ci, c'est-à-dire l'arborescence suivante :

```
-rw-r--r-- 1 ainu wheel 206 juin  20 14:50 helloworld-shr_0.1.bb
helloworld-shr:
-rw-r--r-- 1 ainu wheel   0 juin  21 15:15 hello2goodbye.patch
```

Le fichier, indiqué dans les sources, sera utilisé en tant que patch (**patch=1**), avec un **PNUM** de 1 (le **PNUM** étant le paramètre donné à patch **-pPNUM**), et appliqué après la phase de copie des fichiers sources.

Il est à noter que nous pouvons également faire référence à des fichiers (sans préciser **patch=1**), et que ces fichiers seront copiés dans le dossier de travail. Cela peut-être utile dans certains cas, où les sources ne fournissent pas un fichier de configuration par défaut, par exemple.

Enfin, nous aurons également besoin de dire à **bitbake** où se trouve la racine de travail des sources téléchargés. Dans notre cas, c'est-à-dire depuis le fichier fourni en http, nous savons que celui-ci contient un dossier **helloworld-shr-0.1** où se trouvent tous les sources. Nous écrirons donc cela dans la recette :

```
S = ${WORKDIR}/${PN}-${PV}
```

Il est à noter que S change entre les différents gestionnaires de versions ou selon les sources téléchargés. Par exemple, dans le cas classique d'un paquet utilisant les *autotools*, S aura la forme précitée. Pour un paquet que l'on récupèrera depuis git, cela ressemblera plutôt à ce qui suit :

```
S = ${WORKDIR}/git/${PN}
```

2.3.2 Simulation d'une compilation manuelle

Nous avons à présent des en-têtes correspondant à ce que nous souhaitons empaqueter, que nous avons écrit dans une nouvelle recette dans **openembedded/recipes/linuxmag/helloworld-shr_0.1.bb**. Il nous reste donc à réaliser les opérations de compilation, d'installation et d'empaquetage.

Nous avons précédemment décrit les différentes « phases » d'une recette. Par convention, chaque phase possède une méthode **do_** suivie du nom de la phase : **do_compile**, **do_install**...

Donc, en premier lieu, nous souhaitons compiler notre fichier source **helloworld.c**. À la main, nous effectuerons la commande suivante :

```
$ gcc helloworld.c -o helloworld
```

Sachant que **bitbake** nous fournit une variable **${CC}** correspondant au compilateur de la cible pour laquelle nous compilons, nous écrirons donc la méthode suivante :

```
do_compile() {
        ${CC} helloworld.c -o helloworld
}
```

Constatons ici que le chemin dans lequel nous nous trouvons à l'exécution de la procédure correspond au chemin que nous avions indiqué dans le paramètre **S**. Rajoutons donc cette méthode à la suite, dans notre recette, et essayons de compiler celle-ci, en tapant la commande suivante :

```
$ bitbake -b openembedded/recipes/linuxmag/helloworld-shr_0.1.bb
```

Nous constatons, si l'option **OE_ALLOW_INSECURE_DOWNLOADS** n'est pas activée dans votre configuration locale (celle-ci étant activée dans la configuration SHR par défaut, vous pourrez donc tester sans si vous le souhaitez ;-)), que la compilation échoue avec le message d'erreur suivant (au pouillème près) :

```
NOTE: Running task 1 of 19 (ID: 1, /home/ainu/shr/shr-unstable/
openembedded/recipes/linuxmag/helloworld_0.1.bb, do_setscene)
NOTE: package helloworld-shr-0.1: started
NOTE: package helloworld-shr-0.1-r0: task do_setscene: started
NOTE: Checking if staging package installed
NOTE: No. Manually removing any installed files
NOTE: package helloworld-shr-0.1-r0: task do_setscene: completed
NOTE: package helloworld-shr-0.1: completed
NOTE: Running task 2 of 19 (ID: 2, /home/ainu/shr/shr-unstable/
openembedded/recipes/linuxmag/helloworld_0.1.bb, do_fetch)
NOTE: package helloworld-shr-0.1: started
NOTE: package helloworld-shr-0.1-r0: task do_fetch: started
ERROR: helloworld-shr-0.1: http://build.shr-project.org/tutorial/helloworld-
shr-0.1.tar.gz has no entry in conf/checksums.ini, not checking URI
```

En effet, il est nécessaire de certifier que les sources sont bien identiques à ceux pour lesquels le paquet a été créé. Suite à cette erreur, vous trouverez dans **tmp/checksums.ini** les lignes correspondant aux *checksums* des sources que **bitbake** vient de télécharger. Il vous suffit de les copier, de les insérer dans **openembedded/conf/checksums.ini**, afin de pouvoir fonctionner avec des téléchargements « sécurisés ».

Une fois cette étape faite, relançons notre commande, qui devrait se terminer correctement, et regardons ce que **bitbake** a généré comme paquets :

```
$ ls -al tmp/deploy/glibc/ipk/armv4t/helloworld-*
-rw-r--r-- 1 ainu wheel 684 juin 20 13:41 tmp/deploy/glibc/ipk/armv4t/
helloworld-shr-dbg_0.1-r0_armv4t.ipk
-rw-r--r-- 1 ainu wheel 662 juin 20 13:41 tmp/deploy/glibc/ipk/armv4t/
helloworld-shr-dev_0.1-r0_armv4t.ipk
```

Nous constatons qu'il n'y a que des paquets **debug** et **développement**. En effet, nous n'avons aucunement installé notre binaire, non plus que précisé qu'il faisait partie du paquet standard !

Tout d'abord, gérons donc l'installation, comme nous le ferions à la main :

```
do_install() {
    install -d ${D}${bindir}
    install -m 755 helloworld ${D}${bindir}
}
```

Nous créons ici le dossier cible où se trouvent les binaires, dans notre dossier d'installation temporaire. En effet, **bitbake** possède un dossier d'installation dit « bac à sable » où tout ce qui doit être installé doit se trouver avant la phase d'empaquetage, c'est notre **${D}**. Dans ce dossier, nous créons le chemin vers le dossier des binaires (**/usr/bin**, par défaut, ici appelé **${bindir}**). Ensuite, nous y installons le binaire que nous avons généré. Constatons ici que le chemin dans lequel nous nous trouvons à l'exécution de la procédure correspond au chemin que nous avions indiqué dans le paramètre **S**. Il existe également d'autres variables disponibles, pour les chemins, telles que **${sysconfdir}** pour **/etc**, **${libdir}** pour **/usr/lib**, et je vous ramène à la documentation de **bitbake** pour les moins indispensables !

Enfin, indiquons que ce binaire fait partie des fichiers délivrés par le paquet « standard ».

```
FILES_${PN} += " ${bindir}/helloworld"
```

Cette directive nous assure que dans les fichiers du paquet « standard », à différencier d'avec le paquet **-dbg** ou **-dev**, ce qui se trouve dans **${bindir}/helloworld** est livré.

Nous venons de changer la définition de notre recette, afin de pouvoir la corriger. Étant donné que celle-ci a déjà été compilée, nous allons devoir soit la recompiler (**-c rebuild**), soit forcer une montée en version. Choisissons cette dernière solution, et montons la révision du paquet (ceci faisant encore une fois partie des bonnes pratiques OE) :

```
PR = " r1 "
```

Dès lors, nous obtenons la recette suivante :

```
DESCRIPTION = "Hello World from SHR"
SECTION = "utils"
LICENSE = "GPL"
PR="r1"

SRC_URI = "http://build.shr-project.org/tutorial/${PN}-${PV}.tar.gz"

S = ${WORKDIR}/${PN}-${PV}

FILES_${PN} += " ${bindir}/helloworld"

do_compile() {
        ${CC} helloworld.c -o helloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 755 helloworld ${D}${bindir}
}
```

Si nous tentons d'empaqueter **helloworld-shr**, nous obtiendrons dans **tmp/deploy/glibc/ipk/armv4t** un paquet correspondant, qui sera installable sur votre périphérique, et délivrera un exécutable **helloworld**, compilé pour ARM. Félicitations, vous venez de réaliser votre première recette **bb** !

2.3.3 Utilisation des autotools

Bien entendu, il serait fastidieux de réaliser systématiquement les compilations « à la main », ou selon la méthode ancestrale. Vous constaterez que les fichiers sources de **helloworld-shr** contiennent les éléments nécessaires pour pouvoir utiliser les autotools.

Utiliser ces derniers sous **bitbake**, il n'y a rien de plus simple : ce dernier permet d'hériter d'un système de « classes », dont une, spécifique, appelée « autotools ».

Il nous suffit donc d'écrire :

```
inherit autotools
```

Dès lors, les procédures de configuration, compilation et installation seront gérées par **bitbake** et les autotools. Notre recette (dans laquelle nous n'oublierons pas de changer PR), ressemblera dès lors à ceci :

```
DESCRIPTION = "Hello World from SHR"
SECTION = "utils"
LICENSE = "GPL"
PR="r2"

SRC_URI = "http://build.shr-project.org/tutorial/${PN}-${PV}.tar.gz"

S = ${WORKDIR}/${PN}-${PV}

inherit autotools
```

Beaucoup plus simple, non ? Notez qu'il n'est plus utile de préciser quels sont les fichiers délivrés, les autotools permettent de faire cette opération.

2.3.4 Staging des headers par les autotools

Dans le cadre de bibliothèques, par exemple, nous pouvons avoir envie de « *stager* » des *headers* permettant à des

programmes incluant ces headers d'être compilés sur notre hôte (et donc de faire référence au dossier d'inclusion de notre hôte où se trouvent les headers). De fait, il serait fastidieux d'installer ces headers « à la main ».

De la même façon que pour les autotools standards, il existe une classe permettant donc d'effectuer ce « staging », c'est-à-dire la copie de ces headers dans l'environnement « simulé » pour l'hôte, et contenant le compilateur, les headers, etc.

Afin de mieux comprendre ce qu'est le « *staging* », considérez l'exemple suivant : je développe une bibliothèque, par exemple **libframeworkd-phonegui**, qui fournit des headers que des bibliothèques doivent implémenter afin d'être conforme à un standard défini. De fait, ces headers doivent être présents dans l'environnement de cross-compilation utilisé avec **bitbake**. Le fait de « stager » ces headers consiste à les rendre disponibles de cette manière. Il en va de même des bibliothèques, de la libc, etc.

Pour ce faire, il nous suffit de faire hériter notre recette de **autotools_stage**.

```
inherit autotools_stage
```

Attention : **autotools_stage** est une addition à autotools, il n'effectue que le staging. Afin de stager, et d'utiliser les autotools normalement, il faudra donc préciser les deux :

```
inherit autotools_stage autotools
```

2.4 Tâches et images

Il existe deux éléments spécifiques dans le cadre de la constitution d'une image générée par **bitbake** grâce aux recettes **bb** d'OE : les tâches et les images.

Les tâches permettent de constituer un ensemble cohérent de paquets, potentiellement dépendants de l'architecture cible, qui feront donc l'agrégation d'un certain nombre de logiciels, et dont l'installation permettra, en une seule fois, d'installer cet ensemble.

Par exemple, la tâche standard **task-boot** est une tâche qui permet d'installer tout ce qui est nécessaire à booter sur un périphérique, **sysvinit** par exemple.

Vous pouvez à cet effet regarder les tâches **task-shr** et **task-shr-minimal** qui se trouvent dans **openembedded/recipes/tasks/**.

Une tâche est constituée de la même façon qu'un paquet, à la différence près qu'elle ne fait que dépendre sur d'autres paquets, et n'a pas de directive spécifique de configuration ou compilation. Elle se trouve dans **recipes/tasks** et hérite de la classe **task**.

Dans le cadre de l'image, le principe est identique : il existe une classe **image** dont hérite la recette, et celle-ci se trouve dans **recipes/images**. Une phase importante est cependant rajoutée, c'est-à-dire la phase de constitution du **rootfs**. Durant cette phase, les paquets dont dépend l'image sont installés, et des opérations complémentaires peuvent être effectuées.

À cet effet, le principe reste le même que ce que vous avez pu observer dans la constitution d'un paquet **bitbake** standard, et observer certains exemples vous permettra de comprendre l'ensemble des tâches nécessaires à la constitution d'une image (dont on pourra néanmoins citer **task-boot** et **task-base**).

2.5 Versioning

Nous avons eu l'occasion de parler de paquets issus directement depuis des gestionnaires de contrôle de versions. Il va de soi que ces paquets nécessitent une attention particulière, puisqu'on ne voudra pas forcément récupérer la toute dernière version systématiquement. Pour ce faire, OE et **bitbake** proposent deux éléments à considérer, **PREFERRED_VERSION** et **SRCREV**.

En effet, admettons que je dispose de plusieurs versions d'un même paquet, et que, pour des raisons qui me sont propres, je souhaite utiliser une version spécifique, et non la dernière (qui sera utilisée par défaut). Dans ce cas, il me faut définir dans un fichier de configuration quelle version je souhaite utiliser.

Prenons pour exemple les fichiers inclus par SHR : dans **openembedded/conf/distro/include/shr-autorev.inc**, nous pouvons trouver les lignes suivantes :

```
PREFERRED_VERSION_shared-mime-info = "0.51"
PREFERREO_VERSION_navit = "0.1.0+svnrev${SRCREV}"
```

Ces deux lignes illustrent ce qu'il faut faire lorsque l'on souhaite choisir une version spécifique d'un paquet. La première nous permettra de choisir, selon la version du package (**PV**), la version 0.51 de **shared-mime-info**. A contrario, la deuxième nous permettra de choisir la version SVN de **navit** (en effet, lorsqu'on regarde la **PV** de **navit**, celle-ci correspond bien à ce que nous pouvons voir indiqué ici).

Attention : par défaut, les versions issues d'un gestionnaire de versions sont prioritaires ! Aussi, il faut les « désactiver » si on ne souhaite pas les utiliser par défaut, au moyen de la directive suivante dans leur recette **bb** :

```
DEFAULT_PREFERENCE= " -1 "
```

Par la suite, afin de ne pas récupérer n'importe quelle version depuis le gestionnaire de versions, il nous faudra (impérativement) préciser quel commit utiliser. Voici quelques exemples que l'on peut trouver dans le fichier mentionné précédemment :

```
SRCREV_pn-navit = "2309"
SRCREV_pn-e-wm-config-illume-shr = "${AUTOREV}"
SRCREV_pn-libframeworkd-glib =
"9765f6989d09d8f9b8676e1ef7e2501fe67fbdbc"
```

La première directive nous assure que pour **navit**, qui utilise Subversion, nous récupérerons les sources au commit 2309. La troisième directive est typique d'un git. Nous irons jusqu'au commit de hash **9765f6989d09d8f9b8676e1ef7e2501fe67fbdbc**. La deuxième ligne, quant à elle, est spéciale, et permet de dire que nous utiliserons systématiquement la dernière version disponible.

Attention, il n'existe pas de **SRCREV** par défaut : si vous ne la définissez pas, vous obtiendrez des erreurs lors de la récupération des sources...

Dès lors, l'utilisation de **${SRCREV}** dans la variable **PV** de vos recettes retournera un champ calculé permettant de pouvoir assurer que chaque « mise à jour » vous donnera un nom de paquet supérieur dans l'ordre lexicographique.

2.6 Providers

Il arrive que plusieurs paquets fournissent le même service. C'est le cas, par exemple, de **openembedded/recipes/gpsd/gpsd_2.37.bb** et **openembedded/recipes/freesmartphone/fso-gpsd_git.bb**. C'est d'ailleurs reconnaissable dans ce dernier à la ligne :

```
RPROVIDES_${PN} = "gpsd"
```

Cette ligne signifie que le paquet **fso-gpsd** fournit l'équivalent de **gpsd** à l'exécution.

De fait, afin de choisir parmi ces multiples fournisseurs, il est nécessaire de dire explicitement lequel sera intégré, comme l'on peut voir dans l'exemple suivant, visible dans **openembedded/conf/distro/shr.conf** :

```
PREFERRED_PROVIDER_gpsd = "fso-gpsd"
```

Ici, cette ligne nous permet d'assurer que lorsque **bitbake** enregistre **gpsd** en tant que dépendance d'un paquet, ce sera en fait **fso-gpsd** qui sera considéré.

2.7 Overrides

Il existe dans OE/**bitbake** un mécanisme appelé « Override », qui permet de définir des options spécifiques selon le paquet, le matériel ou encore selon un mot-clé défini par la distribution. Par exemple, SHR définit le mot-clé **shr** dans ses *overrides*, comme vu dans **openembedded/conf/distro/shr.conf** :

```
OVERRIOES = "local:${MACHINE}:shr:angstrom:${TARGET_OS}:${TARGET_
ARCH}:build-${BUILO_OS}:fail-fast:pn-${PN}"
```

On constatera par exemple que ce mot-clé est utilisé dans la recette de **e-wm**, dans **openembedded/recipes/e17/e-wm_svn.bb** :

```
SRC_URI_append_shr = " file://illume-disable-screensaver.patch;patch=1 "
```

Autrement dit, lorsque le mot-clé **shr** fait partie des **OVERRIDES**, un fichier additionnel sera présent au niveau des sources (en l'occurrence, un patch). Selon la configuration locale de votre environnement, donc, vous aurez (ou non) ce patch. Ceci permet de rendre un paquet « spécifique » selon la situation : la machine, la distribution... le tout sans affecter tout le monde ! Ici, pour SHR, le patch désactive le mode économiseur d'écran d'**illume**, le gestionnaire de fenêtres. Cette option était indésirable pour SHR, mais potentiellement utile pour d'autres.

Cette logique est identique pour le matériel (on utilisera les mots-clés correspondant au nom de la machine, **om-gta02** par exemple), tel que visible dans la recette de génération de l'image SHR (**openembedded/recipes/images/shr-image.inc**) :

```
ROOTFS_POSTPROCESS_COMMAND_append_om-gta02 = ";shr_rootfs_gta02_postprocess"
```

Cette directive ajoute une procédure lors de la génération du **rootfs** (l'image) d'un GTA02.

Ces « overrides » peuvent être ciblés, comme ici, ou génériques, comme on peut le voir dans la recette d'**e-wm**, encore, dans **openembedded/recipes/e17/e-wm_svn.bb** :

```
do_install_append() {
#Do some stuff
}
```

Ici, la phase d'installation générée par **bitbake** sera étendue par cette procédure. Il existe à cet effet **append** et **prepend**, qui permettent respectivement de suffixer ou de préfixer une procédure.

2.8 Testez votre paquet

Il existe plusieurs solutions pour tester votre paquet. La plus simple est bien entendu de le copier via **scp** sur votre périphérique.

Vous pouvez également construire un *feed* de votre arbre de paquets, et l'exposer via HTTP, afin de pouvoir le rajouter dans les sources d'opkg.

Enfin, il est également possible d'émuler un périphérique, via Qemu. Les instructions afin de faire fonctionner ce dernier étant relativement complexes, nous vous conseillons de vous référer à http://wiki.openmoko.org/wiki/Qemu afin d'obtenir l'ensemble des instructions nécessaires.

3 FSO (Freesmartphone.Org)

3.1 Introduction

Depuis quelques années, l'importance des ordinateurs tournant sous GNU/Linux est devenue de plus en plus apparente, particulièrement dans le monde des périphériques embarqués, PDA, téléphones, Tablet PC et netbooks. Les bénéfices d'une architecture *kernel* efficace, et d'une abstraction matérielle permettent un portage relativement « simple » vers de nouveaux périphériques (dans l'intérêt des professionnels faisant commerce de telles choses), et donc de les supporter en un temps très court pour les mettre sur le marché avec peu d'investissement. Des périphériques tels que le Motorola série EZX, l'Openmoko Freerunner, les Nokia N8x0, les HTC G1/G2, et le très récemment présenté Palm Pre montrent le succès que peut

apporter Linux en tant que système d'exploitation sur des périphériques embarqués, et encouragent à l'adoption de systèmes d'exploitation libres pour de telles applications.

Autant cela peut avoir un sens pour le kernel, autant la situation est complètement différente en ce qui concerne l'*userland*, c'est-à-dire les logiciels tournant au-dessus du kernel, les applications en contact direct avec l'utilisateur. Dans ce domaine, le libre a donné cours à des douzaines de solutions mutuellement incompatibles, et parfois à moitié finies.

Lorsqu'on regarde outre l'apparent bénéfice de la « liberté de choix », qui est un des facteurs majeurs ayant amené à cette situation, on rencontre des problèmes presque insurmontables, tant pour les utilisateurs que pour les développeurs :

- Pour chaque produit, les développeurs applicatifs ont à supporter une énorme quantité de bibliothèques disparates, de langages de programmation spécifiques, et de nouvelles abstractions. Supporter plus d'une plate-forme embarquée sous GNU/Linux est complexe, consommateur en temps et coûteux.
- De la même façon, les utilisateurs finaux devront apprendre comment opère chaque programme depuis le début pour chaque nouvelle plate-forme. Réutiliser des « métaphores » éprouvées et fonctionnelles est presque impossible.

L'initiative Freesmartphone.Org est une approche pour défragmenter l'espace « mobile », améliorer l'intéroceptivité des plateformes embarquées basées sur GNU/Linux actuelles et futures. Elle consiste en un *middleware* qui se base sur Dbus en tant que standard de communication inter-processus. Les personnes derrière freesmartphone.org se considèrent elles-mêmes comme un complément du très réussi projet « freedesktop.org » **[FDO]**, qui se base lui-même sur Dbus, afin de supporter l'interopérabilité au niveau du monde du PC de bureau, par exemple entre GNOME et KDE.

Freesmartphone.org offre donc un *framework* basé sur Dbus, de services matériels abstraits pour les développeurs, tels que du « *power management* », de la téléphonie GSM, de la configuration réseau, gestion du temps, de la position géographique, des profils, etc. Utiliser ce framework simplifie énormément le développement d'applications pour les périphériques embarqués. Il supporte une architecture applicative « propre » (c'est-à-dire une séparation de la représentation et de la logique métier), et encourage les développeurs à se concentrer sur des applications innovantes ou des interfaces utilisateurs, plutôt que d'avoir à traiter les problèmes issus des spécificités des périphériques ou des technologies sous-jacentes (de nombreux modems GSM, par exemple, ont des commandes spécifiques...)

Ainsi, des solutions comme openBmap **[OBM]**, un programme permettant d'enregistrer les positions GPS de la couverture du réseau GSM, dans le but à terme d'obtenir sa position géographique à partir de l'identité de la cellule GSM à laquelle le téléphone est actuellement connecté, peuvent faire usage de différentes ressources, sans se soucier de la façon « réelle » par laquelle ils doivent interagir avec le matériel : ces programmes s'enregistrent au niveau du framework, et communiquent avec lui de la même façon quel que soit le périphérique sous-jacent. Le programme est donc automatiquement portable sur différents matériels !

3.2 Architecture

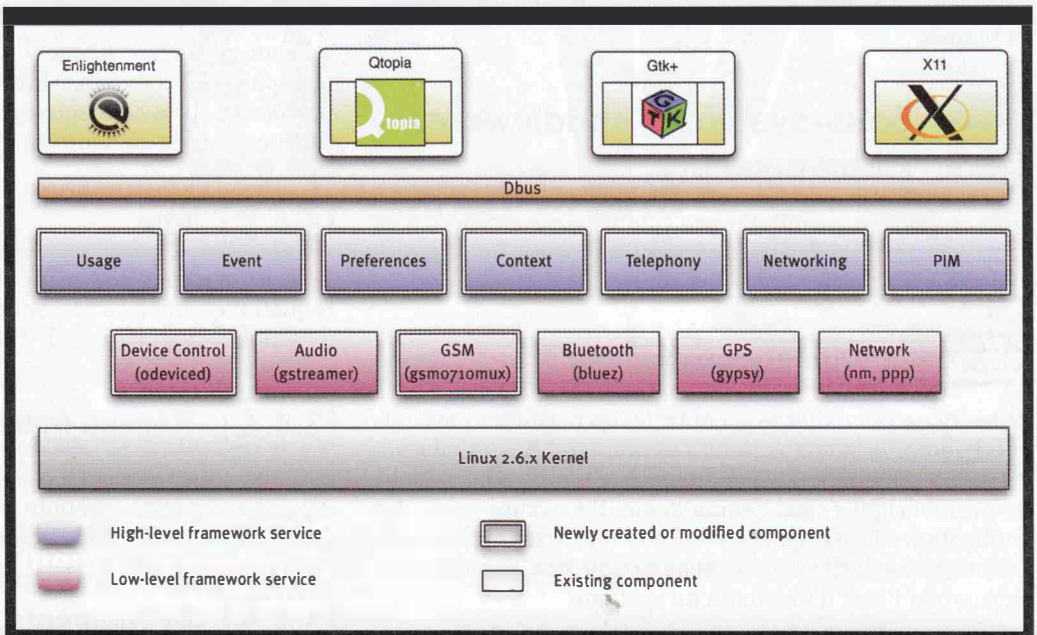


Fig. 1 : Diagramme d'architecture de freesmartphone.org

Bien que l'initiative freesmartphone.org ne fournisse qu'un middleware, il est important de noter comment celui-ci interagit avec les autres composants. Le diagramme de référence de l'architecture du framework montre une architecture quatre-tiers, séparée selon les couches suivantes :

3.2.1 Couche applicative

Cette couche communique avec la couche middleware seulement via Dbus. Ceci permet un grand niveau de liberté pour la couche applicative, puisque cela implique qu'il n'existe aucune contrainte de langage ou de *toolkit* graphique.

3.2.2 Couche middleware

La couche middleware contient l'ensemble des services freesmartphone.org. Le middleware est composé de sous-systèmes distincts, séparés en deux niveaux d'abstraction. Les sous-systèmes communiquent tant au niveau horizontal que vertical via Dbus : ceci est une fonctionnalité importante puisqu'elle permet de multiples niveaux de granularité en termes de serveur/processus.

3.2.3 Couche de services de bas niveau

Cette couche de services de bas niveau contient les solutions existantes pour manipuler des choses telles que la configuration réseau, le bluetooth, l'audio, etc. Certains, parmi ces services, offrent également des primitives Dbus.

3.2.4 Couche kernel

La couche kernel contient les pilotes matériels, et fournit un accès bas niveau via sysfs, procfs, ioctls et des interfaces similaires.

3.3 Sous-systèmes middleware « bas-niveau »

Les sous-systèmes middleware sont séparés en service de haut niveau (en bleu) et de bas niveau (en rouge). Considérons d'abord les services de bas niveau :

3.3.1 Périphérique (device)

Ce service a pour charge le contrôle des périphériques, tels que l'audio, la luminosité de l'écran, les LED, le vibreur, les accéléromètres, mais également le *power control* pour les périphériques sans démon dédié. Il s'occupe aussi des notifications de charge et de la RTC. Sur certains systèmes, il fait également suivre les évènements de boutons, et notifie au sujet de l'état d'inactivité du système.

3.3.2 GSM

Le service de bas niveau GSM s'attend à disposer d'un modem compatible avec le standard GSM 07.07, GSM 07.05 et les spécifications GSM afférentes, parlant un protocole AT par un lien série. Si GSM 07.10 est supporté, un démon de multiplexage est utilisé afin d'exporter les différents liens séries virtuels capables de communiquer via AT.

3.3.3 GPS

Le service de bas-niveau GPS s'attend à un périphérique GPS communiquant via NMEA ou UBLOX UBX4 par un nœud « *device* ». Un service auxiliaire fonctionnant séparément implémente la compatibilité avec le démon **gpsd**.

3.4 Sous-systèmes middleware « haut-niveau »

3.4.1 Usage

Le sous-système « *Usage* » s'occupe de coordonner les pré-requis entrée/sortie des applications (pensez notamment au comptage de référence pour les ressources). Les applications ne sont pas supposées « allumer » ou « éteindre » les périphériques, puisqu'elles n'ont aucune connaissance des logiciels concurrents qui pourraient également utiliser le périphérique en question. Il s'occupe aussi des fonctions de « *suspend* » et « *resume* » haut niveau, notamment car préparer un modem GSM à se suspendre ne peut pas sainement être fait en espace kernel, puisque l'on a besoin d'envoyer plusieurs commandes AT afin de prévenir des mauvais « *wakeup* » . Ces commandes sont spécifiques aux périphériques et à la situation, et doivent donc être gérées au niveau de l'espace utilisateur.

3.4.2 Téléphone (phone)

Le sous-système « *phone* » peut-être utilisé pour créer et manipuler des communications type « voix » sur de nombreux protocoles. Il se connecte également au bluetooth, afin de faciliter l'accès au contrôle des oreillettes.

3.4.3 PIM

Le sous-système PIM offre un moyen d'enregistrer et de requêter les données personnelles de l'utilisateur : contacts, messages, tâches...

3.4.4 Réseau (networking)

L'API de « *networking* » (réseau) fournit des méthodes de haut niveau pour la configuration réseau des périphériques disponibles (WiFi, USB...).

3.4.5 Évènements (events)

Le service d'évènements gère des règles qui connectent des évènements (tels qu'un appel entrant, l'insertion du chargeur, etc.), via des filtres (par exemple selon le profil choisi, silencieux ou vibreur), à d'autres évènements (comme jouer la sonnerie). Dépendant des règles, le service d'évènements peut faire appel à d'autres sous-systèmes.

Notez que le service d'évènements n'est là qu'en tant qu'« aide », des applications complètes pouvant elles-mêmes gérer ce genre de choses.

3.4.6 Preferences

Le sous-système « *preferences* » contient l'ensemble des préférences utilisateurs : les différents types de profils, et leur sonnerie associée, etc.

Abonnez-vous !

Vous lisez d'autres magazines des Éditions Diamond ?
Des offres de couplage sont disponibles au verso.

Économisez Plus de 20%*

11 Numéros de GNU/Linux Magazine pour le prix de 9*

* Sur le prix de vente unitaire France Métropolitaine

* Gain pour un abonnement France Métropolitaine, par rapport au prix unitaire France Métropolitaine

11 Numéros de GNU/Linux Magazine à 55 €
(Offre France Métro)

Soit votre GNU/Linux Magazine à 5,00 €
(Tarif au numéro dans le cadre d'un abonnement France Métro)

Les 3 bonnes raisons de vous abonner !

- Ne manquez plus aucun numéro.
- Recevez GNU/Linux Magazine chaque mois chez vous ou dans votre entreprise.
- Économisez 16,50 €/an ! (soit plus de 2 magazines offerts !)

Pour les tarifs hors France Métropolitaine, consultez notre site : www.ed-diamond.com.

Bon d'abonnement à découper

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

GNU LINUX MAGAZINE / FRANCE

Édité par Les Éditions Diamond

Voici mes coordonnées postales :
Nom :
Prénom :
Adresse :
Code Postal :
Ville :

Vos remarques : .....

.....

.....

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.

Offres d'abonnement

(Nos tarifs s'entendent TTC et en euros)

		F	D	T	E1	E2	EUC	A	RM
		France Métro	DOM	TOM	Europe 1	Europe 2	Etats-unis Canada	Afrique	Reste du Monde
1	Abonnement Linux Magazine	55 €	59 €	67 €	69 €	66 €	70 €	68 €	77 €
2	Linux Magazine + Hors-série	83 €	89 €	101 €	104 €	100 €	105 €	103 €	116 €
3	Linux Magazine + MISC	84 €	90 €	102 €	105 €	101 €	107 €	104 €	117 €
4	Linux Magazine + Linux Pratique	78 €	85€	96 €	99 €	95 €	101 €	98 €	111 €
5	Linux Magazine + Hors-série + Linux Pratique	110 €	119 €	134 €	138€	133 €	140 €	137 €	154 €
6	Linux Magazine + Hors-série + MISC	116 €	124 €	140 €	144 €	139 €	146 €	143 €	160 €
7	Linux Magazine + Hors-série + MISC + Linux Pratique	143 €	154 €	173 €	178 €	172 €	181 €	177 €	198 €
8	Linux Pratique Essentiel + Linux Pratique	57 €	62 €	69 €	71 €	69 €	73 €	71 €	79 €

- 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-dessous correspondant à la zone France Métro (F)

(Vous pouvez également vous abonner sur : www.ed-diamond.com)

Linux Magazine (11 n°)

offre 1 — par ABO : 55€ — Economie : 16,50 €

en kiosque : 71,50€

Linux Magazine (11 n°) + Linux Magazine hors-série (6 n°)

en kiosque : 110,50€

offre 2 — par ABO : 83€ — Economie : 27,50 €

Linux Magazine (11 n°) + Misc (6 n°)

en kiosque : 119,50€

offre 3 — par ABO : 84€ — Economie : 35,50 €

Linux Magazine (11 n°) + Linux Pratique (6 n°)

en kiosque : 107,20€

offre 4 — par ABO : 78€ — Economie : 29,20 €

Linux Pratique Essentiel (6 n°) + Linux Pratique (6 n°)

en kiosque : 74,70€

par ABO : 57€ — Economie : 17,70 €

offre 8

Linux Magazine (11 n°) + Linux Magazine hors-série (6 n°) + Linux Pratique (6 n°)

offre 5 — par ABO : 110€ — Economie : 36,20 €

en kiosque : 146,20€

Linux Magazine (11 n°) + Linux Magazine hors-série (6 n°) + Misc (6 n°)

offre 6 — par ABO : 116€ — Economie : 42,50 €

en kiosque : 158,50€

Linux Magazine (11 n°) + Linux Magazine hors-série (6 n°) + Misc (6 n°) + Linux Pratique (6 n°)

en kiosque : 194,20€

offre 7 — par ABO : 143€ — Economie : 51,20 €

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

Je fais mon choix de la 1ère offre :

Je sélectionne le N° (1 à 8) de l'offre choisie :	
Je sélectionne ma zone géographique (F à RM) :	
J'indique la somme due : (Total 1)	€

Exemple : je souhaite m'abonner à l'offre Linux Magazine + Hors-série + MISC (offre 6) et je vis en Belgique (E1), ma référence est donc 6E1 et le montant de l'abonnement est de 144 euros.

Je fais mon choix de la 2ème offre :

Je sélectionne le N° (1 à 8) de l'offre choisie :	
Je sélectionne ma zone géographique (F à RM) :	
J'indique la somme due : (Total 2)	€
Montant Total à régler (Total 1 + Total 2)	€

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre de Diamond Editions
- ❑ Carte bancaire n° ⌊⌊⌊⌊ ⌊⌊⌊⌊ ⌊⌊⌊⌊ ⌊⌊⌊⌊
  Expire le : ⌊⌊⌊⌊
  Cryptogramme visuel : ⌊⌊⌊

Date et signature obligatoire

Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex

3.4.7 Temps (time)

Le sous-système « *time* » agrège différents sources afin de pouvoir manipuler et mettre à jour la « *timezone* » et l'heure correctement. Il contient également un service d'alarme que vous pouvez utiliser.

3.5 Interfaces applicatives

3.5.1 Dbus

Un des paradigmes majeurs de freesmartphone.org (FSO) est l'homogénéité des interfaces : FSO supporte des interfaces Dbus cohérentes destinées aux développeurs d'applications. Ceci est dû à la compréhension du fait que les développeurs d'applications doivent se concentrer sur la logique métier plutôt que tenter de visualiser les problèmes que peuvent causer une multitude de bibliothèques ou des interfaces de communication entre les processus, disponibles dans le monde du libre. L'homogénéité et la cohérence sont des paradigmes importants, et c'est pourquoi l'architecture n'offre qu'un et un seul moyen d'interagir avec le middleware.

Bien que cela puisse paraître comme la « réinvention de la roue », c'est un point important pour l'adoption d'un middleware : souvent, des sous-systèmes réutilisent en fait des roues existantes, mais en couvrent les détails peu ragoûtants derrière des interfaces Dbus.

Dbus est une technologie très flexible de communication entre processus client/serveur : il supporte des interfaces orientées procédurales et objets, des invocations synchrones ou asynchrones, tout autant que de la communication point à point ou point à multipoint.

Puisque la popularité de Dbus augmente sans cesse, des solutions tierces basées sur Dbus, telles que bluez4 **[BLUEZ]**, connman **[CONN]**, wpa-supplicant **[WPAS]**, peuvent souvent être utilisées par le framework directement, et, si nécessaire, être améliorées par des appels de plus haut niveau afin de pouvoir supporter les problématiques transversales.

3.5.2 Comment comprendre les interfaces ?

Afin de communiquer avec un sous-système de FSO via son interface Dbus, il est nécessaire de connaître trois choses :

- Nom du bus : Le nom du bus identifie le processus au niveau du serveur auquel les « invocations » Dbus seront délivrées. Dépendant de l'implémentation, de multiples sous-systèmes peuvent être dans le même processus ou être répartis sur plusieurs. Dans tous les cas, un sous-système freesmartphone.org possède un nom unique de la forme `org.freesmartphone.subsystem`, où `subsystem` est, à la rédaction de cet article, un parmi les suivants : `odeviced`, `ousaged`, `ogsmd`, `ogpsd`, `oeventsd`, `otimed`, `onetworkd`, `opreferencesd`, `opimd`.
- Chemin de l'objet : Le chemin de l'objet identifie quel objet au niveau serveur recevra les invocations Dbus. De multiples objets peuvent être implémentés sur un seul chemin au niveau du serveur. Les chemins d'objets doivent être formatés comme des chemins de fichier, avec `/` en tant que séparateur. Chaque objet Dbus freesmartphone.org possède le préfixe `/org/freesmartphone/`. La liste suivante montre l'ensemble des objets exportés par le sous-système `org.freesmartphone.odeviced`. Notez que les objets sont dynamiques : dépendant du cas d'utilisation et de la disponibilité du matériel et des composants logiciels, les objets peuvent apparaître et disparaître à tout moment durant le cycle de vie du sous-système.

```
/
/org
/org/freesmartphone
/org/freesmartphone/Device
/org/freesmartphone/Device/Audio
/org/freesmartphone/Device/CPU
/org/freesmartphone/Device/Display
/org/freesmartphone/Device/Display/0
/org/freesmartphone/Device/Display/gta02_bl
/org/freesmartphone/Device/IdleNotifier
/org/freesmartphone/Device/IdleNotifier/0
/org/freesmartphone/Device/Info
/org/freesmartphone/Device/Input
/org/freesmartphone/Device/LED
/org/freesmartphone/Device/LED/gta02_aux_red
/org/freesmartphone/Device/LED/gta02_power_blue
/org/freesmartphone/Device/LED/gta02_power_orange
/org/freesmartphone/Device/LED/neo1973_vibrator
/org/freesmartphone/Device/PowerControl
/org/freesmartphone/Device/PowerControl/Bluetogta02_aux_redoth
/org/freesmartphone/Device/PowerControl/UsbHost
/org/freesmartphone/Device/PowerControl/WiFi
/org/freesmartphone/Device/PowerSupply
/org/freesmartphone/Device/PowerSupply/ac
/org/freesmartphone/Device/PowerSupply/adapter
/org/freesmartphone/Device/PowerSupply/apm
/org/freesmartphone/Device/PowerSupply/battery
/org/freesmartphone/Device/PowerSupply/usb
/org/freesmartphone/Device/RealTimeClock
/org/freesmartphone/Device/RealTimeClock/0
/org/freesmartphone/Device/RealTimeClock/rtc0
```

- Nom de l'interface : Les interfaces regroupent des invocations de méthodes Dbus en catégories. Une interface Dbus définit les signatures des méthodes et signaux disponibles : c'est le contrat d'API entre un client et un serveur Dbus. De multiples interfaces peuvent être implémentées par un objet. Chaque interface freesmartphone.org a le préfixe `org.freesmartphone`. La liste suivante présente les interfaces exposées par l'objet Dbus `/org/freesmartphone/Device/LED/0`, fourni par `org.freesmartphone.odeviced`.

```
[METHOD] org.freedesktop.DBus.Introspectable.Introspect()
[METHOD] org.freesmartphone.Device.LED.BlinkSeconds( i:seconds, i:delay_on,
i:delay_off )
[METHOD] org.freesmartphone.Device.LED.GetName()
[METHOD] org.freesmartphone.Device.LED.SetBlinking( i:delay_on, i:delay_off )
[METHOD] org.freesmartphone.Device.LED.SetBrightness( i:brightness )
[METHOD] org.freesmartphone.Device.LED.SetNetworking( s:interface, s:mode )
```

3.5.3 Comment utiliser les interfaces ?

En tant qu'exemple d'utilisation des services Dbus freesmartphone.org, nous allons montrer la séquence d'initialisation « typique » de la téléphonie GSM, où nous nous enregistrons auprès du sous-système, puis du réseau GSM, et enfin, émettons un appel. Avant de commencer, veuillez noter les choses suivantes :

- Les exemples de code suivants utilisent les primitives Dbus disponibles dans le langage Python **[PYTH]**, qui offre une manière très simple et aisée d'utiliser Dbus via un interpréteur de commandes interactif. Les concepts s'appliquent bien entendu à tous les autres langages supportant Dbus, bien que la quantité de code nécessaire puisse varier du tout au tout.
- L'exemple fourni se servira de Dbus de manière synchrone. Bien que la méthode asynchrone soit presque toujours à préférer (puisque le client local restera actif et pourra faire autre chose en attendant la réponse du serveur Dbus distant), les appels synchrones sont plus simples à visualiser et à mettre en place.
- Nous ne montrerons que la gestion d'erreur applicative la plus basique. En travaillant avec Dbus, vous devez être prêt à gérer le fait que les objets peuvent apparaître ou disparaître à tout moment, ce qui signifie bien entendu que chaque appel peut échouer. Des applications robustes prendront ceci en considération.

Bien entendu, il sera également supposé que vous disposez d'un périphérique embarqué de type GTA02, avec le framework installé. Veuillez vous référer à la procédure d'installation de SHR afin d'en disposer un prêt à l'emploi.

La première étape est d'importer le module de support Dbus où toutes les déclarations et définitions nécessaires se trouvent. Connectons-nous donc à notre GTA02, et effectuons la commande suivante :

```
$ python
Python 2.6.2 (r262:71600, Apr 23 2009, 11:37:56)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import dbus
>>>
```

L'étape suivante est de récupérer un identifiant vers un bus Dbus, où se trouvent les objets qui nous intéressent. Dbus supporte deux types de bus par défaut : un bus système (pour des services au niveau système, qui démarrent avec celui-ci), et un bus de session (par utilisateur, qui démarre avec chaque session X11). Les sous-systèmes freesmartphone.org se trouvent sur le bus système, similaires en cela à beaucoup d'autres services d'abstraction matérielle.

```
>>> bus = dbus.SystemBus()
>>>
```

Comme avec de nombreux autres systèmes client/serveur, avant de pouvoir communiquer avec un serveur distant, vous devez récupérer un identifiant local qui agira en tant que « proxy » du service distant. Ici, vous avez besoin du triplet susnommé : nom du bus, chemin de l'objet, et signature de l'interface. Une séquence typique d'initialisation GSM doit appeler des méthodes localisées sur l'objet GSM, mais appartenant à des interfaces différentes. Aussi, nous allons créer des proxies pour tous au préalable :

```
>>> oproxy = bus.get_object('org.freesmartphone.ogsmd', '/org/
freesmartphone/GSM/Device')
>>> gsmdevice = dbus.Interface(oproxy, 'org.freesmartphone.GSM.Device')
>>> gsmsim = dbus.Interface(oproxy, 'org.freesmartphone.GSM.SIM')
>>> gsmnet = dbus.Interface(oproxy, 'org.freesmartphone.GSM.Network')
>>> gsmcall = dbus.Interface(oproxy, 'org.freesmartphone.GSM.Call')
```

Une fois que nous disposons de ces proxies d'interface, nous sommes prêts à appeler des méthodes sur ces objets. Afin d'être capable d'écouter des signaux Dbus, vous avez besoin d'un *mainloop*, ce qui est plus complexe. Dans le cadre de cet exemple « simple », nous ne montrerons que les appels de méthodes.

La procédure pour initialiser la téléphonie GSM ressemble « en gros » à :

- Allumer l'antenne. Si nécessaire, envoyer le code PIN.
- S'enregistrer avec le fournisseur de téléphonie.

Allumer l'antenne peut-être fait grâce à `org.freesmartphone.GSM.SetAntennaPower(bool)`. Cette méthode ainsi que toutes les autres peuvent être trouvées dans la documentation de référence de freesmartphone.org **[FSODOCS]**. Selon la documentation pour cet appel, nous pouvons obtenir une erreur potentielle `org.freesmartphone.GSM.SIM.AuthFailed`, dans quel cas nous aurons à envoyer le code PIN avant de pouvoir aller plus loin.

```
try:
   gsmdevice.SetAntennaPower(true)
except DBus.DBusException( e ):
   if e.get_dbus_name() == 'org.freesmartphone.GSM.SIM.AuthFailed':
      gsmsim.SendAuthCode( '1234' )
   else:
      print 'Unknown Error'
```

Si cet extrait de code (que vous pouvez directement écrire dans l'interpréteur Python) marche correctement, alors nous sommes prêts pour nous enregistrer avec le fournisseur de téléphonie.

```
>>> gsmnet.Register()
```

Veuillez noter que s'enregistrer auprès d'un réseau GSM peut prendre quelque temps. Ne soyez donc pas surpris si cet appel bloque pendant 30 secondes ou plus. Pour vérifier que nous sommes bien enregistrés auprès du réseau que nous souhaitons, nous pouvons, par exemple, afficher le nom du fournisseur de téléphonie auquel nous sommes connectés. Celui-ci s'obtient via une structure qui est envoyée en tant que résultat de la méthode `org.freesmartphone.GSM.Network.GetStatus()` :

```
>>> print gsmnet.GetStatus()
dbus.Dictionary({dbus.String(u'code'): dbus.String(u'20810', variant_
level=1), dbus.String(u'cid'): dbus.String(u'72CD', variant_level=1),
dbus.String(u'act'): dbus.String(u'GSM', variant_level=1), dbus.
String(u'provider'): dbus.String(u'F SFR', variant_level=1), dbus.
String(u'lac'): dbus.String(u'5529', variant_level=1), dbus.
String(u'strength'): dbus.Int32(72, variant_level=1), dbus.
String(u'mode'): dbus.String(u'automatic', variant_level=1), dbus.
String(u'registration'): dbus.String(u'home', variant_level=1)},
signature=dbus.Signature('sv'))
```

Si tout marche comme prévu, le résultat devrait donc être similaire à celui affiché ci-dessus (veuillez consulter la documentation freesmartphone.org afin de connaître la signification de chaque champ). Nous constatons bien que le champ **provider** a pour valeur **F SFR**, ce qui était attendu.

Enfin, puisque nous sommes enregistrés auprès du réseau GSM, nous pouvons maintenant essayer d'appeler quelqu'un. Ceci peut être effectué via **org.freesmartphone.GSM.Call.Initiate(string,string)**, le premier paramètre étant le numéro à appeler, et le deuxième étant le type d'appel.

```
>>> gsmcall.Initiate('123', 'voice')
```

3.6 Étendre le framework

Les développeurs de freesmartphone.org ont tout récemment commencé à travailler sur la seconde implémentation de référence : FSO 2.0 (iso fonctionnelle avec la version 1). Pour un grand nombre de raisons, qui n'ont que peu d'intérêt pour cet article, la première implémentation avait été faite en Python, qui impose des limitations dues à la haute consommation processeur, et à la signature mémoire de la machine virtuelle. La prochaine version est en train d'être implémentée avec le langage Vala **[VALA]**, un langage de haut niveau aux multiples possibilités, qui se compile en C ANSI, et s'intègre extrêmement bien avec des bibliothèques supportant des API C.

Dans cette section, nous allons montrer un exemple bref permettant de créer un sous-système freesmartphone.org et un module se conformant à FSO 2.0. La plupart du travail à effectuer sera couvert par la bibliothèque **libfsoframework**, qui contient les parties suivantes :

- un « *logger* » avec de multiples destinations possibles ;
- l'accès à des fichiers de configuration ;
- un gestionnaire de modules, qui gère le cycle de vie de ces modules ;
- une abstraction de sous-système qui enregistre les modules auprès de Dbus ;
- des classes permettant la gestion des travaux asynchrones ;
- le support des classes de communication Kobject et Netlink ;
- de nombreuses fonctions de type « helper ».

3.6.1 Programme principal

Le programme principal de **fsoexampled** :

```
GLib.MainLoop mainloop;
FsoFramework.Logger logger;

public static int main( string[] args )
{
    logger = FsoFramework.createLogger( "fsoexample" );
    logger.info( "fsoexample starting up..." );
    var subsystem = new FsoFramework.DBusSubsystem( "fsoexample");
    subsystem.registerPlugins();
    uint count = subsystem.loadPlugins();
    logger.info( "loaded %u plugins".printf( count ) );
    mainloop = new GLib.MainLoop( null, false );
    mainloop.run();
    return 0;
}
```

Après la création d'un logger au niveau processus, il s'enregistre lui-même en tant que nouveau sous-système Dbus appelé **fsoexample**. L'abstraction de sous-système utilisera ce nom afin de chercher les valeurs de configuration se trouvant dans **/etc/frameworkd.conf** pour les modules définis.

Pour enregistrer un module au sous-système, vous devez donc rajouter une section (vide) à ce fichier.

```
# Subsystem configuration for fsoexample:
[fsoexample]
# Example plugin
[fsoexample.example]
```

Après avoir compilé ce programme, et l'avoir *linké* avec **libfsoframework**, le framework cherchera votre module **example** dans le chemin canonique : **${prefix}/lib/cornucopia/modules/<subsystem>/<plugin>.so**

3.6.2 Code du module

Le sous-système appellera une fonction spécifique de votre module appelée **fso_factory_function()**. C'est le lieu idoine pour créer vos objets Dbus ou pour faire une initialisation du module. Ici, nous allons juste créer un objet **HelloWorld**. Notez que la fonction **factory** a comme paramètre le sous-système que vous avez créé précédemment. Sauvegardez ce paramètre quelque part, afin de pouvoir l'utiliser pour enregistrer des objets Dbus.

```
internal FsoExample.HelloWorld instance;

public static string fso_factory_function( FsoFramework.Subsystem
subsystem ) throws Error
{
    instance = FsoExample.HelloWorld( subsystem );
    return 'fsoexample.example';
}
```

3.6.3 Interface Dbus du module

Notre objet **HelloWorld** doit exposer une méthode **HelloWorld()** qui retournera une chaîne. Pour ceci, nous allons créer la classe d'interface Dbus suivante :

```
[DBus (name = 'org.freesmartphone.Example') ]
public interface ExampleInterface
{
    public abstract string HelloWorld();
}
```

3.6.4 Implémentation du module Dbus

L'implémentation du module ressemble donc à la classe suivante :

```
class HelloWorld : public ExampleInterface
{
    private FsoFramework.Subsystem subsystem;

    public HelloWorld( FsoFramework.Subsystem subsystem )
    {
        this.subsystem = subsystem;
        subsystem.registerServiceName( 'org.freesmartphone.fsoexampled' );
        subsystem.registerServiceObject( 'org.freesmartphone.
fsoexampled', '/org/freesmartphone/Example/HelloWorld', this );
    }

    public string HelloWorld()
    {
        return 'Hello World from the Example Plugin!';
    }
}
```

Le constructeur contient deux appels importants utilisant le paramètre **subsystem** fourni par la fonction **factory** (voir plus haut) :

- **registerServiceName()** est une méthode de sous-système qui enregistre un service Dbus sous un nom donné. Cette méthode peut-être utilisée plusieurs fois, tant que cela est fait depuis le même processus, puisque les noms de bus ne peuvent pas être partagés entre plusieurs processus.
- **registerServiceObject()** est une méthode de sous-système qui enregistre un objet Dbus donné sous un chemin donné.

Enfin, nous implémentons la méthode **HelloWorld** susnommée, méthode qui sera accessible en tant que **org.freesmartphone.ExampleInterface.HelloWorld()**.

Le résultat de tout cela est que, lorsque vous lancerez le processus **fsoexampled**, et que ce module sera chargé, un service Dbus sera présent sur **org.freesmartphone.fsoexampled**, et fournira notre objet **HelloWorld** sur le chemin **/org/freesmartphone/Example/HelloWorld**.

4 SHR (Stable hybrid release)

4.1 Installer SHR

Deux images sont disponibles : la version « *lite* », et la version normale, cette dernière contenant plus de logiciels que la première. Veuillez vous référer au site de SHR afin d'obtenir des informations complémentaires sur le contenu **[SHR]**. Afin de l'installer, il vous faudra donc télécharger l'image et son kernel sur http://build.shr-project.org.

Pour un développeur, le choix se porte tout naturellement sur *unstable*. Téléchargeons donc l'image et son kernel (il sera supposé que le développeur souhaite installer une « lite » sur un GTA02) :

```
$ wget http://build.shr-project.org/shr-unstable/
images/om-gta02/uImage-om-gta02-latest.bin
$ wget http://build.shr-project.org/shr-unstable/
images/om-gta02/shr-lite-image-om-gta02.jffs2
```

Il nous reste ensuite à flasher ces deux fichiers sur notre périphérique, via **dfu-util** (disponible dans toutes les bonnes crémeries ou gestionnaires de paquets).

La procédure afin de mettre votre périphérique dans un état permettant de le flasher est la suivante :

Fig. 2 : L'interface standard de SHR

- Appuyez sur [AUX] et restez appuyé.
- Appuyez sur [POWER] jusqu'à ce que le périphérique démarre.
- Connectez-le à votre ordinateur, et tapez les commandes suivantes sur ce dernier :

```
$ dfu-util -a kernel -R -D uImage-om-gta02-
latest.bin
$ dfu-util -a rootfs -R -D shr-lite-image-
om-gta02.jffs2
```

Il ne vous reste plus qu'à redémarrer votre périphérique, et à profiter de la distribution !

4.2 Architecture

Suite à de nouvelles discussions sur la liste de diffusion de développement de SHR, voici le schéma d'architecture générale adoptée récemment.

Dans ce schéma, nous constatons qu'Ophonekitd, en noir, est un démon central, qui va être le point d'accès global pour tous les appels de l'interface utilisateur : ce démon écoute les évènements du framework FSO via les bibliothèques en orange (qui permettent de simplifier l'interfaçage avec Dbus en C ou Vala), et va les redistribuer vers l'interface utilisateur correspondante.

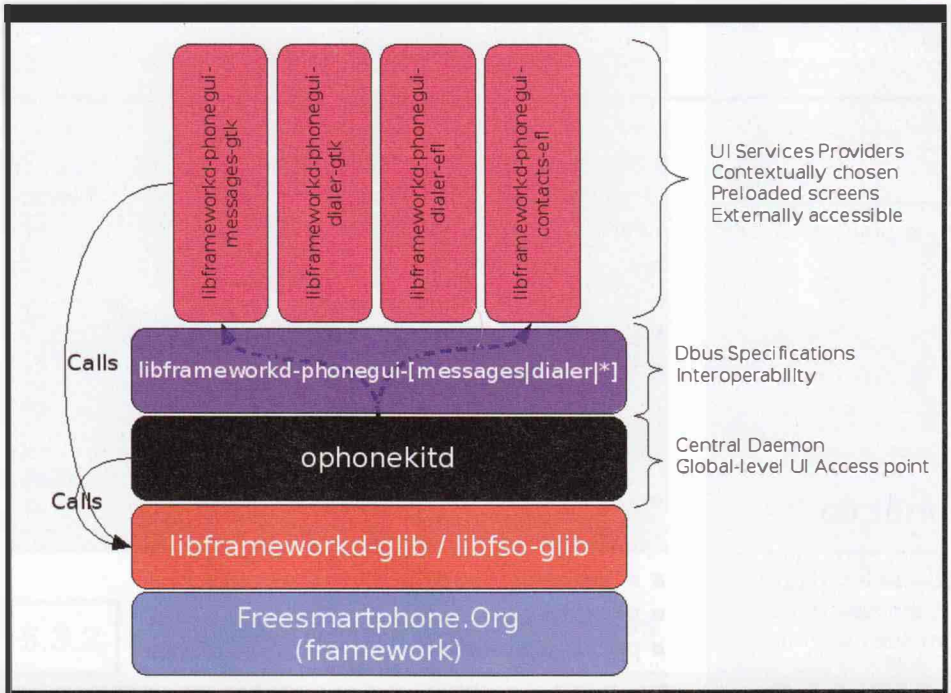


Fig. 3 : Diagramme d'architecture de SHR

Un ensemble fonctionnel de « primitives interfaces utilisateur » a ainsi été défini, délimitant en contextes fonctionnels les différentes parties de ce que les développeurs SHR jugent une « interface utilisateur standard » pour un smartphone. On citera, par exemple, le contexte « Dialer », le contexte « Contact », « Message », etc. Ces différents contextes fonctionnels sont définis par des spécifications Dbus, à partir desquelles des headers C et Vala seront produits, permettant d'implémenter directement des bibliothèques satisfaisant aux contraintes des spécifications.

Considérons un exemple : si l'évènement d'un appel entrant est émis par le framework, Ophonekitd aura pour charge d'instancier l'écran utilisateur correspondant, en choisissant la bonne « bibliothèque » dans laquelle se trouve cet écran. Cet écran continuera son cycle de vie par lui-même, et aura pour charge d'émettre des appels vers le framework pour les tâches de plus « bas » niveau qu'il souhaitera accomplir : passer un appel, lister les contacts, etc. Il pourra néanmoins faire appel aux services Dbus exposés par Ophonekitd, afin d'interagir avec les autres écrans : par exemple, l'écran pour passer un appel pourra comporter un bouton pour afficher la liste de contacts, qui ne fera qu'appeler (via Dbus) l'écran de sélection de contacts.

Cette architecture permet ainsi d'offrir un niveau de modularité supplémentaire à l'utilisateur : il n'a pas besoin de choisir une « suite » correspondant à une bibliothèque gérant toutes les parties de la téléphonie. S'il préfère la bibliothèque A pour les contacts, il peut préférer la bibliothèque B pour les messages, et le but de cet architecture est typiquement de répondre à ce besoin. Cela permet donc, par corrélation, d'avoir des interfaces utilisant des toolkits différents, voire d'instancier dynamiquement en fonction du contexte une bibliothèque spécifique (par exemple, si je n'ai plus de batterie, je souhaiterai limiter mes ressources CPU, et donc j'utiliserai une bibliothèque « graphique » **ncurses** pour afficher mes appels, parce que **ncurses**, c'est bien !).

Enfin, cela permet aussi de limiter les constructions d'écrans en les « préchargeant », afin non seulement de limiter les ressources consommées (qui sont malheureusement rares), mais également de rendre l'affichage des écrans plus rapide.

4.3 Travailler sur un paquet « SHR » localement

SHR fournit de base, dans la configuration, un fichier appelé **local-builds.inc** que vous pouvez trouver dans **shr-unstable/conf**, par exemple, si vous suivez les instructions de la première partie.

Si vous souhaitez être capable de « cloner » les dépôts de sources SHR, et être capable de les modifier localement dans votre copie du dépôt git, alors il vous faudra vous servir de ce fichier. La première chose à faire est de l'inclure dans votre **conf/local.conf** :

```
$ echo "require local-builds.inc" >> shr-unstable/conf/local.conf
```

Ensuite, il vous faut définir sur quel paquet vous souhaitez travailler. L'exemple par défaut concerne **libframeworkd-phonegui-efl**, qui est l'interface utilisateur de téléphonie délivrée par défaut. Supposons que nous souhaitions travailler dessus, nous aurons juste à changer les paramètres suivants :

```
SRC_URI_pn-libframeworkd-phonegui-efl = "file:///path/to/source/shr"
SRCREV_pn-libframeworkd-phonegui-efl = "LOCAL"
S_pn-libframeworkd-phonegui-efl = "${WORKDIR}/shr/${PN}"
```

Ici, **SRC_URI_pn-libframeworkd-phonegui-efl** indique le chemin complet vers la copie locale du dépôt de sources dans lequel se trouve sur le git original les sources de **libframeworkd-phonegui-efl**. Ce dossier sera entièrement copié pour la phase « fetch » par **bitbake** et, donc, l'endroit où se trouvent les sources dans le répertoire de travail de **bitbake** doit être également modifié (troisième ligne). Enfin, comme on ne considère plus le gestionnaire de versions, mais des fichiers locaux, il faut changer la **SRCREV**, afin que **bitbake** ne plante plus au calcul de celle-ci, ne pouvant pas en trouver en local. C'est ce que fait la deuxième ligne.

Lorsque vous compilerez ce package, donc, le nom des paquets générés contiendra **LOCAL** au lieu du nom normal de révision.

Si vous souhaitez travailler sur d'autres logiciels, il vous suffit de répéter cette procédure en changeant les chemins, et noms, en fonction du logiciel sur lequel vous souhaitez travailler. Il ne vous reste plus qu'à vous y mettre, bonne chance !

5 Hackable:1

5.1 Pourquoi Debian ?

L'utilisation de Debian a été privilégiée pour les raisons suivantes :

- 18000 paquets disponibles ;
- 1500 développeurs ;
- strict au niveau des licences ;
- qualité des paquets.

5.2 Pourquoi Gnome mobile ?

Gnome mobile est une spécification qui décrit les composants privilégiés de Gnome pour les plateformes mobiles. Hackable:1 adhère à cette spécification pour les raisons suivantes :

- framework mature ;
- beaucoup de développeurs sont disponibles ;
- documentation assez riche et de bonne facture ;
- support du i18n (y compris le bidi).

5.3 Installation et lancement

5.3.1 Préparation de la micro SD

Actuellement, Hackable:1 est fournie sous la forme d'images pré-installées qu'il suffit de décompresser sur une carte micro SD. Il est évidemment nécessaire de posséder :

- un Openmoko Freerunner ;
- un ordinateur avec un lecteur de carte micro SD ;
- une carte micro SD d'une capacité d'au moins 1 Go (2 Go conseillés).

Concernant le choix de la carte, il faut veiller à prendre une classe 4 au minimum (classe 6 conseillée).

Préparation de la carte :

Nous allons formater la carte en 2 partitions :

- partition 1 : 8Mo en fat ;
- partition 2 : ext2 pour tout le reste.

Insérez la carte dans le lecteur de votre poste de travail. La première chose à déterminer est le nom du périphérique. Pour cela, lancez la commande suivante :

```
# dmesg | tail
```

Puis, notez le nom du périphérique qui gère la carte micro SD. Ensuite, démontez-la (**umount**) et, en *root*, lancez **fdisk** :

```
# fdisk /dev/sdX
```

Avec sdX comme nom du périphérique.

Si des partitions existent déjà, il faut les détruire en tapant **d** (il faut taper **d** autant de fois qu'il y a de partitions). Ensuite, ajoutez une nouvelle partition :

- **n**
- **primary partition**
- **partition number** : 1
- **size** : +8M

Changez le type en vfat :

- **t**
- **4**

Ajoutez une autre partition :

- **n**
- **primary partition**
- **partition number** : 2

Finalisez les modifications en tapant **w**.

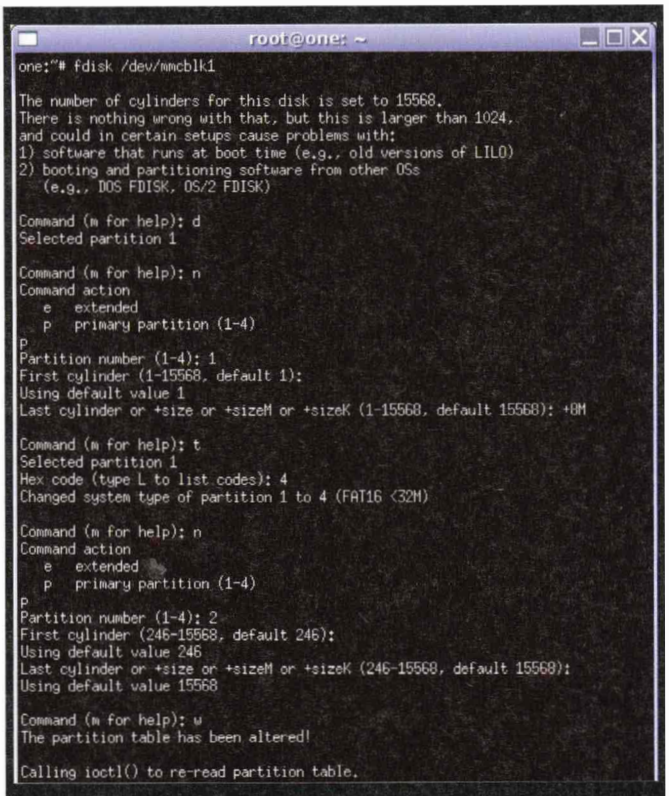


Fig. 1 : Partitionnement de la carte micro SD

Il faut maintenant formater les deux nouvelles partitions. La première partition doit être en vfat :

```
# mkfs.vfat /dev/sdX1
```

Et la seconde en ext2 :

```
# mkfs.ext2 /dev/sdX2
```

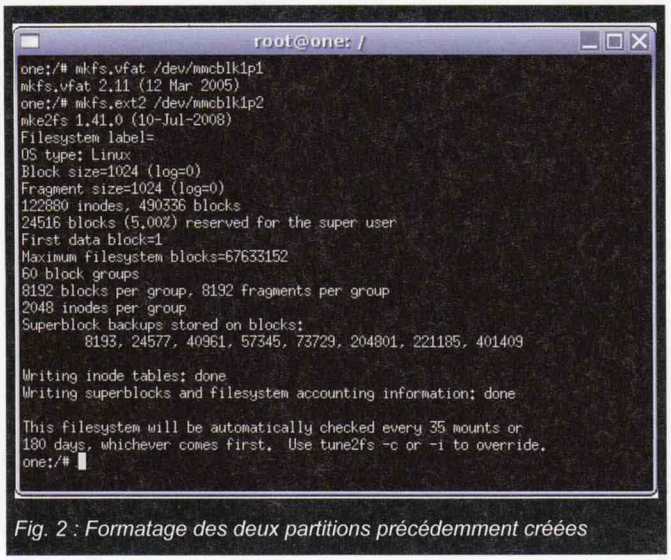


Fig. 2 : Formatage des deux partitions précédemment créées

5.3.2 Récupération et installation des archives

Téléchargez les archives sur votre poste de travail :

```
# wget http://download.hackable1.org/h1-fat_partition-rev4.tar.gz
# wget http://download.hackable1.org/h1-ext2_partition-rev4-classic-rc1.tar.gz
```

En root, extrayez les archives dans un répertoire temporaire, puis montez les 2 partitions précédemment créées.

Allez dans le répertoire de la partition fat, puis :

```
# tar xzmof /path/to/h1-fat_partition-rev4.tar.gz
```

Faites la même chose avec la partition ext2 :

```
# tar xzf /path/to/h1-ext2_partition-rev4-classic-rc1.tar.gz
```

5.3.3 Démarrage et premiers pas

Le Freerunner doit démarrer sur la carte SD. Pour cela :

- Appuyez sur le bouton [AUX].
- En même temps, appuyez sur le bouton [POWER].
- Relâchez le bouton [AUX] en gardant le bouton [POWER] appuyé, un menu doit apparaître.
- Choisissez la deuxième ligne avec le bouton [AUX], puis appuyez sur [POWER] pour démarrer depuis la carte.

Note

Si vous souhaitez que le Freerunner démarre par défaut sur la carte SD, un tutoriel est disponible à l'adresse suivante : **http://www.hackable1.org/wiki/DefaultBoot**.

Une fois démarré, la fenêtre de sélection du code PIN apparaît, ainsi que la page d'accueil.

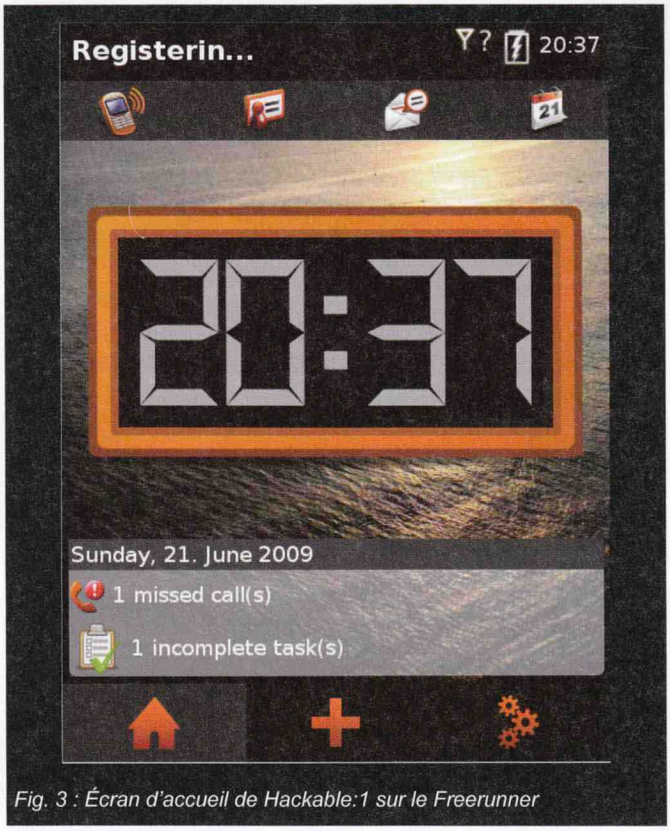


Fig. 3 : Écran d'accueil de Hackable:1 sur le Freerunner

Afin de pouvoir se connecter au Freerunnervia le câble USB, lancez sur votre poste de travail, en root, les commandes suivantes :

```
# iptables -A POSTROUTING -t nat -j MASQUERADE -s 192.168.0.0/24
# sysctl -w net.ipv4.ip_forward=1
# ip addr add 192.168.0.200/24 dev usb0
# ifconfig usb0 up
```

Ensuite, connectez-vous avec :

```
# ssh root@192.168.0.202
root@192.168.0.202's password:
```

Le mot de passe est **root**.

Et voilà, vous êtes maintenant connecté et prêt à travailler !

5.4 Compilation d'un programme : prototypage

Hackable:1 possède tous les outils pour être directement opérationnel. Dans ce chapitre, je vais prendre le classique **hello world** disponible sous Debian Lenny.

Connectez-vous sur le Freerunner, puis ajoutez la ligne suivante dans **/etc/apt/sources.list** :

```
deb-src http://ftp2.de.debian.org/debian lenny main
```

Ensuite :

```
apt-get update && apt-get source -b hello
```

Après quelques minutes, le paquet devrait être construit et disponible. En effet, tout est disponible sur la micro SD pour compiler directement dessus. Afin d'accélérer la phase de compilation, on peut (et c'est même conseillé) utiliser **distcc** (voir plus bas).

Cette méthode, assez pratique, sert principalement pour de la mise au point et du prototypage rapide de programmes pas trop volumineux. En effet, la phase d'édition des liens est toujours réalisée sur le Freerunner et est consommatrice de ressources.

Pour des logiciels plus conséquents et un déploiement plus classique, il convient d'utiliser un cross-compilateur.

5.5 Utilisation du cross-compilateur

5.5.1 Installation

Hackable:1 met à disposition des développeurs un cross-compilateur complet et fonctionnel, basé sur **emdebian**. Il est disponible sous plusieurs formes :

- en version précompilée stable pour les architectures i386 et amd64 ;
- en version *daily build* toujours pour les architectures i386 et amd64 ;
- depuis le dépôt SVN de Hackable:1 et qu'il convient de générer sur son poste.

Ici, nous utiliserons la version précompilée stable pour i386.

Récupérez l'archive sur votre poste de travail :

```
# wget http://download.hackable1.org/cross/Hackable1-Openmoko-
Freerunner-cross-2009.529-i386.tar.gz
```

Ensuite, en root :

```
# mkdir cross
# tar -xzvpf path/to/Hackable1-Openmoko-Freerunner-cross-2009.529-i386.tar.gz -C cross
```

Configuration du réseau :

Il sera peut-être nécessaire de configurer **/etc/resolv.conf** avec les bons serveurs DNS. Le plus simple est de copier celui de votre système :

```
# cp /etc/resolv.conf cross/etc/resolv.conf
```

Démarrage.

En root, tapez la commande suivante :

```
# /usr/bin/env -i HOME=/root PATH=$PATH SHELL=$SHELL TERM=$TERM /usr/
sbin/chroot cross /bin/bash
```

Et voilà vous êtes maintenant prêt à utiliser le cross compilateur !

5.5.2 Utilisation

Lors de la compilation, il faut utiliser le compilateur dédié et les bibliothèques associées. Ceux-ci sont situés dans le répertoire **/usr/arm-linux-gnueabi/**.

pkg-config est configuré pour aller chercher ces bibliothèques grâce aux variables d'environnement suivantes :

```
PKG_CONFIG_LIBDIR="/usr/arm-linux-gnueabi/usr/lib/pkgconfig:/usr/arm-
linux-gnueabi/usr/share/pkgconfig"
PKG_CONFIG_SYSROOT_DIR="/usr/arm-linux-gnueabi"
```

Compilation d'un logiciel qui utilise Autoconf/Automake : l'utilisation de l'option **--host** devrait fonctionner dans la plupart des cas :

```
# ./configure --host=arm-linux-gnueabi
# make
```

Autre possibilité :

```
make CC=arm-linux-gnueabi-gcc
```

Compilation du noyau Linux : pour configurer le noyau pour une plateforme ARM.

```
# make ARCH=arm menuconfig
```

Pour lancer la compilation :

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi
```

5.5.3 Exemple : compilation de hello, world

Ajoutez la ligne suivante dans **/etc/apt/sources.list** :

```
deb-src http://ftp2.de.debian.org/debian lenny main
```

Ensuite :

```
apt-get update && apt-get source  hello
```

Allez dans le répertoire **hello-2.2**, puis tapez :

```
#./configure --host=arm-linux-gnueabi
#make
```

Copiez le binaire **src/hello** sur le Freerunner (via un **scp** par exemple) et exécutez-le. Vous devriez voir apparaître :

```
# ./hello
Hello, world!
```

5.5.4 Exemple : compilation de Neod

Neod est le logiciel qui gère entre autres les boutons [AUX], [POWER] et la gestion de l'énergie. Il est un peu moins trivial à compiler que le simple **hello world**.

Récupérez les sources :

```
# svn co svn://svn.hackable1.org/hackable1
```

Allez dans le répertoire de **neod** :

```
# cd hackable1/trunk/src/core/neod
```

Puis, lancez **autogen.sh** :

```
# autogen.sh --host=arm-linux-gnueabi
```

Il est nécessaire de modifier le fichier **ltmain.sh** qui vient d'être généré pour que **libtool** utilise les bibliothèques du Freerunner plutôt que les natives (en utilisant l'option **-inst-prefix-dir**).

Ajoutez à la ligne 2784 du fichier **ltmain.sh** :

```
        esac
        fi
        test -f "$inst_prefix_dir$add" && add="$inst_prefix_dir$add"
# ajouter cette ligne
        if test "$linkmode" = prog; then
```

Ensuite, relancez le script **configure** :

```
# ./configure --host=arm-linux-gnueabi
```

Puis, lancez la compilation :

```
make LDFLAGS="-inst-prefix-dir /usr/arm-linux-gnueabi -L/usr/arm-linux-gnueabi/lib -L/usr/arm-linux-gnueabi/usr/lib -Wl,-rpath-link /usr/arm-linux-gnueabi -Wl,-rpath-link /usr/arm-linux-gnueabi/usr/lib"
```

Après quelques minutes, le binaire devrait être construit. Un **file** sur **src/neod** nous indique que le format est bien le bon :

```
# file neod
# neod: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.14, not stripped
```

On peut s'en assurer en le copiant en lieu et place de celui présent sur le Freerunner et en relançant le processus **neod**.

5.5.5 Utilisation de distcc

Le cross compilateur est fourni avec un environnement complet dont **distcc** qui permet de déporter les phases de compilation sur votre machine. Voici les étapes pour le démarrer.

Lancez l'environnement (en root) :

```
# /usr/bin/env -i HOME=/root PATH=$PATH SHELL=$SHELL TERM=$TERM /usr/sbin/chroot cross /bin/bash
```

Loguez-vous sous le compte utilisateur **hackable1** :

```
# su - hackable1
```

Démarrez le démon **distcc** :

```
# distccd --daemon --allow '192.168.0.202'
```

Et voilà, ensuite il suffit de se connecter sur le Freerunner et de lancer une compilation classique !

Hackable:1 sur le Freerunner utilise par défaut **distcc** s'il est disponible sur votre poste de travail. Ce comportement est configuré dans le fichier **~/.bashrc** avec les lignes suivantes :

```
export CC="distcc arm-linux-gnueabi-gcc"
alias dh='export DISTCC_HOSTS="192.168.0.200"'
alias dh1='export DISTCC_HOSTS="192.168.1.200"'
export DISTCC_HOSTS="192.168.0.200"
```

5.6 Trucs et astuces

5.6.1 Ajouter une entrée dans la page d'accueil

Si vous voulez que votre application soit visible depuis le menu d'accueil, il suffit de créer un fichier **monappli.desktop** dans le répertoire **/usr/share/applications/** du Freerunner. Ce fichier suit la spécification de freedesktop.org (http://standards.freedesktop.org/desktop-entry-spec/latest/).

Exemple de fichier :

```
[Desktop Entry]
Name=woosh! Browser
Comment=Minimalistic but fast Browser
Encoding=UTF-8
Version=1.0
Type=Application
Exec=woosh
Icon=woosh
Terminal=false
Categories=GTK;Application;PIM;mainapps
SingleInstance=true
StartupNotify=true
```

Les catégories sont définies dans le répertoire **/usr/share/matchbox/vfolders**. Vous pouvez modifier et créer d'autres catégories sur le modèle de celles existantes.

5.6.2 Relancer l'interface graphique sans redémarrer

Si, lors de vos développements, vous avez besoin de relancer l'interface graphique et tous les logiciels lancés au démarrage, vous pouvez utiliser la commande suivante afin d'éviter un redémarrage complet du Freerunner :

```
#/etc/init.d/xserver-hackable1 restart
```

5.6.3 Utiliser x2x

x2x permet d'utiliser sa souris et son clavier sur le Freerunner. Pour l'activer, lancez la commande suivante sur votre poste de travail (en étant déjà connecté avec le Freerunner) :

```
# ssh -X root@192.168.0.202 "/usr/bin/x2x -east -to :0.0"
root@192.168.0.202's password:
```

Rentrez le mot de passe (root).

Maintenant, vous pouvez déplacer votre souris sur la droite de votre écran et elle devrait apparaître sur le Freerunner. Votre clavier devient alors actif sur le Freerunner.

5.6.4 Lancer une application graphique depuis votre poste

Afin de lancer une application graphique depuis votre poste (depuis la connexion SSH), il faut lancer la commande suivante :

```
# ssh  root@192.168.0.202
192.168.0.202's password:
debian-gta02:~# export DISPLAY=:0
```

Ensuite, vous pouvez lancer n'importe quelle application graphique depuis cette console.

5.6.5 Déporter l'affichage depuis le Freerunner sur votre poste

Vous pouvez utiliser votre poste de travail comme périphérique d'affichage.

Préparation du poste de travail :

- Installez le logiciel Xephyr (`apt-get install xserver-xephyr` sous Debian) sur votre poste de travail.
- Démarrez-le avec la commande :

```
# Xephyr :1 -ac -2button -host-cursor -screen 480x640 -dpi 150
```

Sur une console ssh connectée sur le Freerunner, lancez un `export DISPLAY` vers votre poste :

```
# export DISPLAY=192.168.0.200:1
```

Ensuite, n'importe quelle application graphique lancée depuis cette console sera affichée sur votre poste.

5.7 Hackable:1, en quelques mots

Hackable:1 permet d'effectuer ses premiers pas en développement sur le Freerunner en quelques minutes. On peut même s'amuser avec des langages plus exotiques comme Erlang avec un simple `apt-get install erlang`.

Le cross-compilateur est présenté de manière rapide. Je vous invite à aller voir la page dédiée sur le wiki Dbus avec plus particulièrement la méthode pour cross-compiler un paquet Debian.

Pour information, ce cross-compilateur est généré par un outil similaire à Debootstrap qui permet de générer à partir du dépôt SVN :

- un cross-compilateur ;
- des images prêtes à l'emploi pour des cartes SD ou à flasher directement sur le Freerunner.

Conclusion

À l'issue de cet article, vous devriez être en mesure de pouvoir développer des logiciels pour un GTA01/GTA02, qui est, rappelons-le, le seul essai commercial de téléphone (presque) entièrement libre sur tous les points. Que vos préférences vous mènent vers Debian ou vers OpenEmbedded, vous devriez être en mesure de choisir la distribution adaptée à vos besoins, et commencer à donner de vous-même pour participer à cette fabuleuse aventure qui, nous l'espérons, ne manque pas d'avenir. Aussi, n'hésitez pas à réagir sur nos listes de diffusion, soumettre des rapports de bogues ou encore à créer vos propres logiciels ! Tous les outils sont dès à présent entre vos mains.

Références

- [OBM] : http://www.openbmap.org
- [OEGIT] : http://cgit.openembedded.net
- [SHR] : http://trac.shr-project.org
- [SHRGIT] : http://git.shr-project.org
- [FSO] : http://trac.freesmartphone.org
- [FSOGIT] : http://git.freesmartphone.org
- [FSODOCS] : http://docs.freesmartphone.org
- [FDO] : http://www.freedesktop.org
- [BLUEZ] : http://www.bluez.org
- [CONN] : http://moblin.org/projects/connection-manager
- [WPAS] : http://hostap.epitest.fi/wpa_supplicant
- [GPSD] : http://gpsd.berlios.de
- [PYTH] : http://www.python.org
- [VALA] : http://www.vala-project.org

Auteurs : Jérome Blondon, Julien Cassignol, Dr Michael Lauer

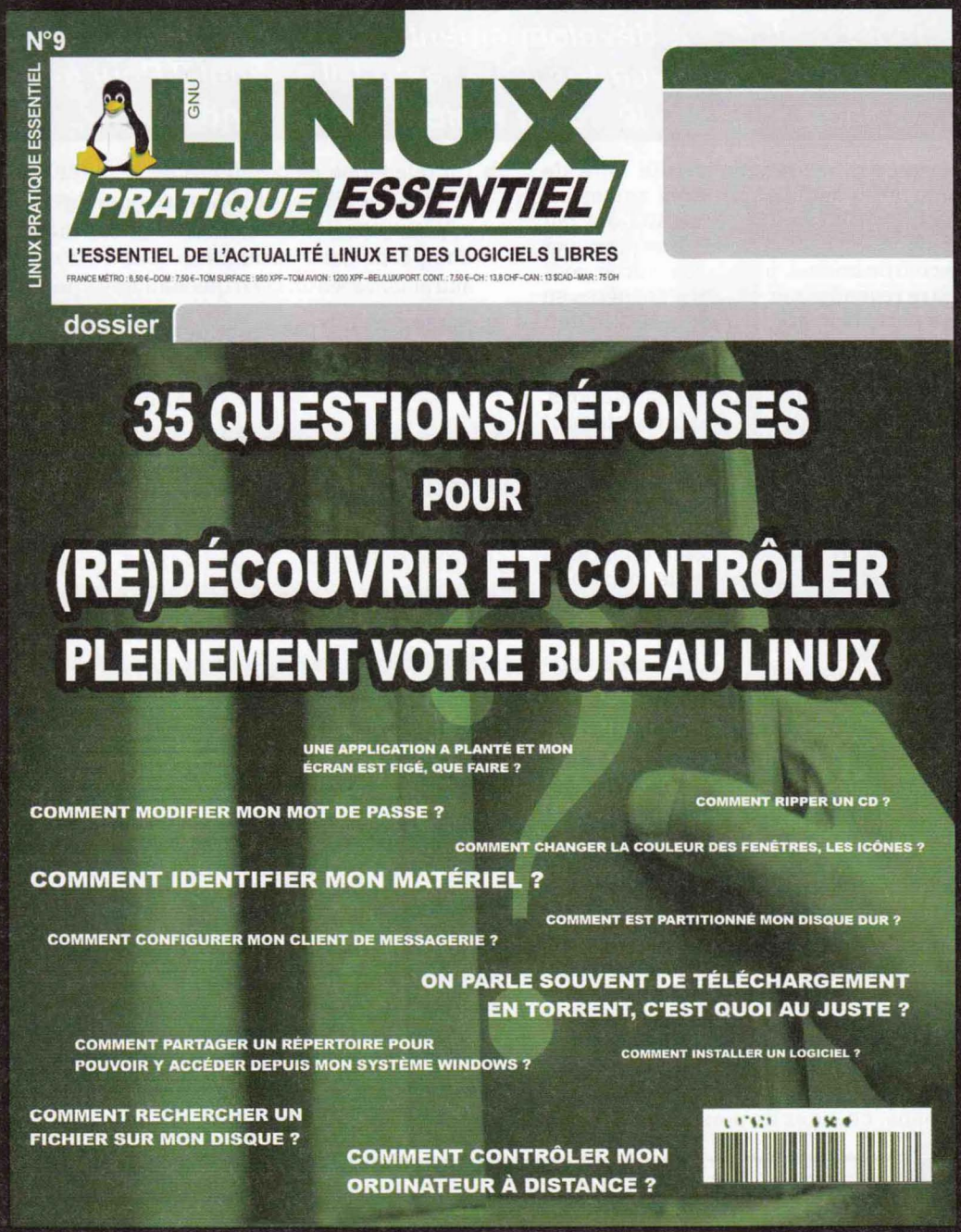
Jérome Blondon :
Projets et R&D chez Bearstech.

Julien Cassignol :
Projets et R&D chez Bearstech.
Dictateur bénévole de SHR.

Dr Michael Lauer :
Dictateur bienveillant de FSO.

Connaissez-vous vraiment votre environnement de bureau ?

LINUX PRATIQUE ESSENTIEL N°9



Comment modifier mon mot de passe ?

Comment changer la couleur des fenêtres, les icônes ?

Comment est partitionné mon disque dur ?

Comment configurer mon client de messagerie ?

On parle souvent de téléchargement en torrent, c'est quoi au juste ?

Comment contrôler mon ordinateur à distance ?

Comment installer un logiciel ?

Comment ripper un CD ?

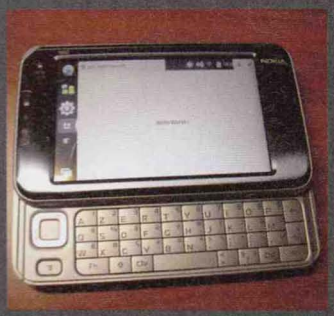
Comment rechercher un fichier sur mon disque ?

Une application a planté et mon écran est figé, que faire ?

*sous réserve de toute modification

DISPONIBLE DÈS LE 31 JUILLET 2009*
CHEZ VOTRE MARCHAND DE JOURNAUX

Installez et utilisez la plateforme



Trouver un périphérique sous GNU/Linux, possédant une plateforme de développement en logiciel libre et relativement active, n'est pas chose facile. Si l'on souhaite, de plus, que le matériel correspondant soit véritablement utilisable et non un simple gadget de développement, un certain nombre de périphériques seront écartés. Que reste-t-il au final ? Pour ma part, le choix s'est porté sur le tablet PC Nokia n810.

Auteur

- Denis Bodor

Je suis un utilisateur de longue date de périphériques mobiles. Toutes sortes de produits, Palm, Psion, iPaQ, GTA02 ont fait l'objet d'une acquisition avant de se retrouver sur un coin de bureau, puis dans un tiroir pour enfin être revendus sur un site d'enchères en ligne (à perte bien sûr !). Voilà le destin tout tracé des produits que l'on achète avec les meilleures intentions du monde et qui, après avoir fait l'objet de maintes expérimentations, se retrouvent délaissés. Des produits qui n'étaient finalement pas destinés à une véritable utilisation quotidienne.

Parmi les principales erreurs que l'on commet dans le choix de ce type de périphériques mobiles, nous avons la nécessité apparente de tout centraliser sur une seule machine : GPS, Mail, Web, Shell, Wifi, GSM/3G, etc. C'est un réflexe facile, mais non un choix valable pour tous.

Dans la réalité, il n'est pas sûr que vous soyez véritablement compatible avec ce type de produits. Dans bien des cas, l'utilisateur averti possédera déjà des serveurs, une ou plusieurs machines de bureau/développement, un *laptop* et un téléphone mobile. Il est également probable que le GPS soit de la fête dans la voiture ou en promenade. Pourquoi donc alors réunir tout cela (ou presque) dans un seul périphérique ?

- Un ultra-portable n'est-il pas un peu gros pour une sortie au restaurant ? C'est certes bien pratique, mais ça prend pas mal de place entre l'entrée et le plat principal. Je passe sur l'aspect totalement asocial et la perception qu'en aura la ou les personnes qui vous accompagnent (certes, mauvaise personne, changez de personne).
- Un téléphone ne doit-il pas avoir la taille la plus réduite possible en cas de sortie impromptue ? Un téléphone se glisse dans une poche sans encombrer. Il en va tout autrement avec un *smartphone* hors de prix.
- Le GPS n'est-il pas conçu pour trouver son chemin soit sur la route, soit en balade champêtre ?

Ma solution est ainsi la suivante :

- un téléphone Nokia petit, mais bluetooth ;
- un *thinkpad* pour les déplacements (semi)-professionnels ;
- une machine de bureau aux ressources conséquentes pour le développement et les tests ;
- quelques vieilleries destinées aux expérimentations les plus exotiques ;
- des serveurs un peu partout (comment faire autrement) ;
- et tout un tas de petits systèmes embarqués qui vivent partout dans l'appartement.

Il ne manquait finalement qu'un élément : le périphérique nomade offrant un accès à mes différentes ressources (mail, VPN, wiki, SSH, etc.) tout en restant plus transportable qu'un Thinkpad et sans prendre la place du GPS de randonnée (un mignon petit Garmin Geko 301), ni même du téléphone. Ce périphérique est, pour moi, le Nokia n810.

1 Nokia n810

Le n810 est le successeur du n800 (étonnant, non ?). Je le répète, il ne s'agit pas d'un téléphone, mais d'un *tablet PC*. Voici les caractéristiques de la bête :

- Processeur Texas Instruments OMAP 2420 à 400 MHz basé sur une architecture ARM (famille ARM11 version ARMv6) et intégrant un DSP TMS320 C55x et un processeur graphique (GPU) PowerVR MBX avec accélération 2D et 3D.
- Écran tactile 800×480 pixels de 4.1 pouces de diagonale. Il s'agit d'un écran transflectif

de développement Maemo

utilisable en pleine lumière et dans l'obscurité. Un capteur intégré permet un réglage automatique de l'écran en fonction de la luminosité ambiante.

- 256 Mo + 2 Go de mémoire de stockage.
- 128 Mo de RAM.
- Connectivité Wifi IEEE 802.11b/g.
- Bluetooth 2.0.
- Un lecteur de carte MiniSDHC.
- Un connecteur Micro-USB 2.0 pour connexion à une machine hôte (le n810 apparaît comme un disque USB) ou en mode hôte pour le branchement de périphérique (technologique USB *On-The-Go*).
- Clavier rétro-éclairé disponible en version AZERTY.
- Webcam 640 × 480 pixels en façade.
- GPS intégré.
- Batterie 1500 mAh Nokia BP-4L.

Le matériel est sympathique et polyvalent, mais le vrai plus vient clairement de l'aspect logiciel. Le n810 fonctionne sous un environnement appelé « Maemo » qui n'est rien d'autre, dans les grandes lignes, qu'un GNU/Linux intégrant Gnome, le système de gestion de paquets Debian APT et une interface graphique Hildon/Maemo.

Vous disposez donc d'un environnement connu, voire très connu pour les utilisateurs Debian/Ubuntu. On retrouve ainsi des éléments déjà présents dans un environnement de bureau classique :

- noyau Linux ;
- shell et bibliothèques standards (libc, OpenSSL, curlHTTP, etc.) ;
- gestion de paquets APT/DPKG ;
- services système (DBus, SQLite, etc.) ;
- composants GNOME (GStreamer, GNOME VFS, GConf, etc.) ;
- *framework* classique GTK+/GNOME.

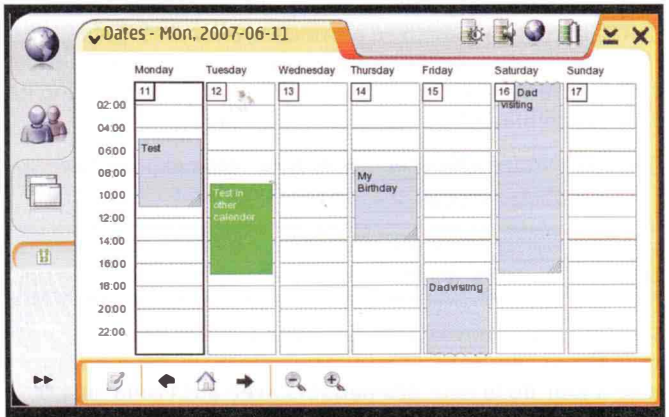
Se greffent simplement sur cette pile, le framework Hildon reposant à la fois sur GTK+/GNOME, X Window et le gestionnaire de fenêtre Matchbox. Hildon est, avant tout, un jeu d'extensions spécifiques GTK+ qui permet d'obtenir un bureau cohérent et ergonomique composé d'un panneau de contrôle, d'une barre d'état, d'un navigateur de tâches et d'un ensemble d'*applets*.

La dernière version de Maemo, la 4.1, est le résultat d'une longue évolution sur plusieurs années. Tout comme la tablette elle-même, avec le n770, n800 et n810, le système a gagné en maturité comme le montrent les différentes captures. Au-delà de l'interface utilisateur, c'est l'ensemble du système, des applications et du SDK qui se sont bonifiés avec le temps.

OS2005 livré avec le n770 en novembre 2005 intégrait Opéra comme navigateur Web, et un client mail relativement basique. Une interface graphique pour APT accompagnait un petit jeu d'outils et d'applications (visualisateur PDF, navigateur de fichiers, etc.). La philosophie de développement consistant à fournir un SDK *open source* était déjà de mise et les applications pouvaient ainsi être facilement portées de GNU/Linux à Maemo. L'ergonomie générale en revanche n'était pas vraiment au rendez-vous. Nous étions au début de l'ère tablet PC.

Maemo tel qu'il existait pour le n770. Un look un peu vieillot alors que le système n'a que 4 ans.

OS2006. C'est en mai 2006 que Nokia annonçait la nouvelle version de Maemo et une version définitive arriva fin juin pour le n770. L'intégration de nouvelles applications, comme un client Google Talk, accompagnait la stabilisation du système et de l'interface. Le passage au noyau 2.6.16 en version multitâche préemptif améliora la réactivité générale de la plateforme. Cette version de Maemo est la dernière officiellement supportée pour le n770.

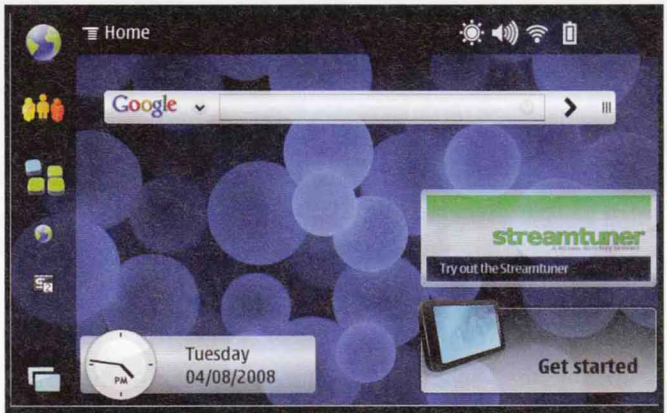


OS2006 est la dernière version de Maemo pour le n770. Une légère évolution apportant son petit lot d'améliorations.

OS2007 est la version accompagnant le n800, le successeur du n770. L'interface changea, tout comme l'architecture interne du système, devant s'adapter à l'évolution du matériel et aux nouveaux périphériques. Une nouvelle version d'Opera et Flash furent également intégrés.

Un design plus sobre et plus moderne pour le n800. OS2007 peut être vu comme une version charnière de Maemo, tout comme le n800 lui-même.

OS2008 est le système équipant le n810. Le système aussi bien que le matériel sont une étape significative vers la maturité. Opera cède la place à MicroB, un navigateur basé sur Mozilla. Un médiaplayer plus performant est également ajouté, tout comme un système de partage de fichiers basé sur Samba. L'autonomie gagne également en stabilité avec l'utilisation de l'ajustement automatique de fréquence processeur. Côté interface, le gestionnaire d'applications est plus ergonomique, plus stable et davantage dans l'esprit d'APT (notion de dépôts, etc.). Mais, la fonctionnalité majeure et sans doute celle la plus attendue par les utilisateurs est le système de mise à jour. Il devient possible de mettre à jour le système de manière incrémentale et non plus en une seule fois par flashage et perte de données. Nommé « SSU », pour *Seamless Software Update*, le système opère à la manière d'une mise à jour telle qu'on la connaît pour les GNU/Linux de type *desktop* comme Ubuntu. Cette version du système OS2008 ou Maemo 4.1 est souvent appelée « Diablo ». Ce nom de code désigne en réalité non pas le système lui-même, mais la mise à jour des fonctionnalités impliquées comme c'était le cas avec Bora (OS2007/n800) et Gregale (OS2006/n770).



Maemo tel qu'il est aujourd'hui. Aboutissement de plusieurs années de développement matériel et logiciel. On retrouve ainsi une barre d'état, des applets sur le bureau et un environnement entièrement thémable. Mais, la vraie réussite de Maemo se trouve derrière les fioritures graphiques : un noyau Linux, un système de paquets Debian et des bibliothèques GTK+/GNOME.

De manière générale, sur le site officiel (http://maemo.org), les versions sont référencées sous les désignations OS2006, OS2007 et OS2008.

2 Installer l'environnement de développement Maemo

L'une des principales raisons du choix de l'acquisition d'un n810 est la possibilité de porter et de développer rapidement des applications sous l'environnement Maemo. Pour ce faire, il faut, avant toutes choses, installer le SDK. Comme le système utilisé par le tablet PC est construit autour de GNU/Linux et de Debian en particulier, en toute logique, c'est cette distribution qui se trouve favorisée pour l'installation. Il s'agit d'un savant mélange entre **chroot**, installation de paquets Debian et installation semi-automatique.

Première étape, ajoutez un fichier, **/etc/apt/sources.list.d/n810.list** par exemple, contenant :

```
# /etc/apt/sources.list.d/maemo.list
deb http://scratchbox.org/debian/ apophis main
```

Ceci fait, vous pouvez, installer avec **aptitude** les différents paquets permettant l'installation du SDK Maemo après mise à jour de la liste des paquets avec **aptitude update** :

```
scratchbox-core
scratchbox-devkit-cputransp
scratchbox-devkit-debian
scratchbox-devkit-maemo3
scratchbox-devkit-perl
scratchbox-libs
scratchbox-toolchain-cs2005q3.2-glibc2.5-arm
scratchbox-toolchain-cs2005q3.2-glibc2.5-i386
```

À la fin de l'installation, le système de configuration vous propose d'ajouter le groupe **sbox** et d'inclure un ou plusieurs utilisateurs à l'intérieur de ce dernier. Répondez affirmativement et ajoutez l'utilisateur qui fera usage de la plateforme de développement (ici **denis**). Vous devez ensuite ajouter, dans Scratchbox, le ou les utilisateurs en question avec :

```
% sudo sb-adduser denis
Scratchbox user account for user denis added
```

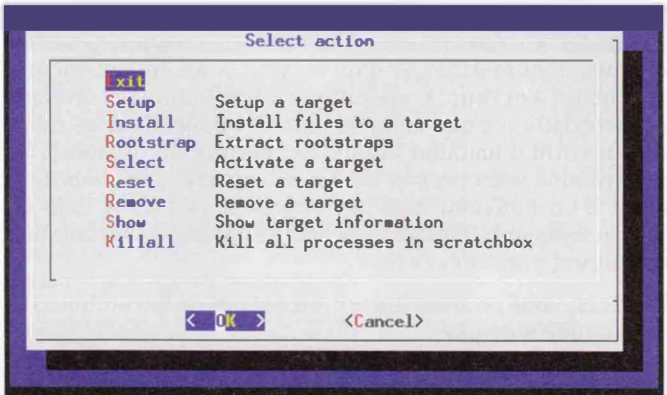
Dès lors, vous pourrez installer le SDK Maemo en commençant par télécharger le script http://repository.maemo.org/stable/diablo/maemo-sdk-install_4.1.2.sh. L'installation est relativement longue. Je vous conseille donc fortement de lancer ce script depuis une session GNU Screen afin, éventuellement, de reprendre la main à distance :

```
% sh maemo-sdk-install_4.1.2.sh
```

Le script affiche les configurations qui seront utilisées :

```
Target configuration for armel (DIABLO_ARMEL)
compiler=cs2005q3.2-glibc2.5-arm
devkits=perl:debian-etch:maemo3-tools:cputransp
cputransp=qemu-arm-0.8.2-sb2

Target configuration for i386 (DIABLO_X86)
compiler=cs2005q3.2-glibc2.5-i386
devkits=perl:debian-etch:maemo3-tools
```

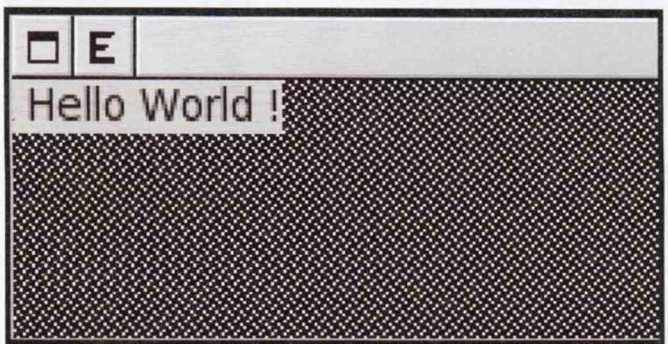


Le menu de configuration de Scratchbox accessible via la commande sb-menu

Ce sont les deux *cibles* de construction des binaires. Nous avons, d'une part, la plateforme **ARMEL** (dont l'ARM OMAP 2420) pour la construction directement à destination du n810 et, d'autre part, **X86** pour l'exécution locale sans émulation.

Le script affiche également la bonne marche des opérations et l'on peut constater que les dépendances sont totalement satisfaites :

```
wget tool in path... /usr/bin/wget
Checking for dpkg tool in path... /usr/bin/dpkg
Running outside of scratchbox... yes
Script not run as user root... yes
Scratchbox installation found... yes
Scratchbox home directory... yes
Scratchbox sb-conf tool... yes
Scratchbox user's bind mount... yes
Scratchbox user's /dev set up... yes
Scratchbox home directory... yes
Scratchbox login... yes
Scratchbox login executable... yes
Scratchbox sb-conf tool works... yes
Scratchbox version requirement met... yes
Scratchbox CPU transparency method for ARMEL is present... yes
Scratchbox toolchains... yes
Scratchbox required devkits... yes
Host kernel VDSO support... yes
Check for scratchbox sessions running...
Check the mmap_min_addr value... yes
```



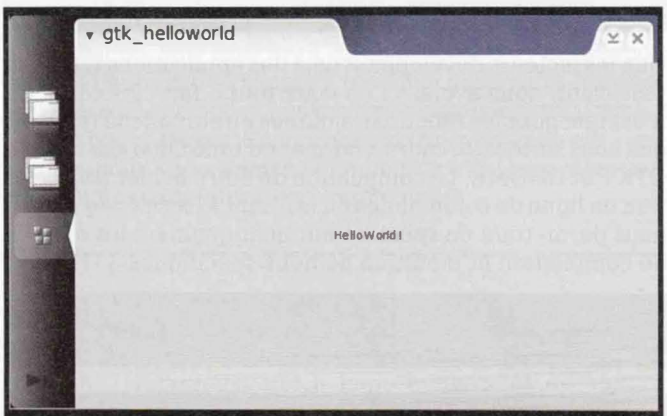
Une simple application GTK+ lancée depuis l'environnement Scratchbox. Esthétiquement très « années 90 ». Pourtant, il ne faut pas grand-chose pour en faire une véritable application pour le n810 : un environnement Maemo fonctionnel et quelques variables d'environnement, mais aucune recompilation.

Vous devrez ensuite accepter la limitation de garantie (Nokia ne doit pas être tenu responsable si vous provoquez la fin du monde avant 2012 avec ce SDK ;)). Choisissez ensuite les paquets du SDK à installer. Choisir ***Runtime Environment + All Dev Packages*** vous permettra de tester la plateforme et de développer/construire de nouvelles applications. C'est le choix par défaut. On vous demandera si des binaires propriétaires de Nokia doivent être installés avant de vous afficher un résumé des opérations qui seront exécutées.

Allez prendre un café, à partir de là, le processus se poursuit seul avec téléchargement et installation. Après plusieurs tasses, l'installation se termine avec ces quelques mots bien agréables :

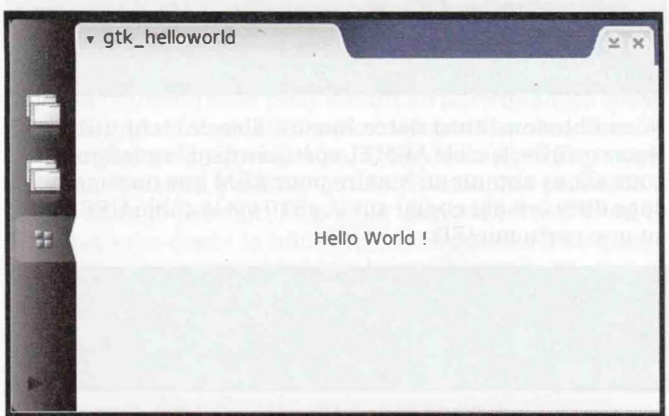
```
Happy hacking!
```

Il ne vous reste plus qu'à lancer la commande **scratchbox** pour vous retrouvez dans l'environnement de développement *chrooté*. Notez au passage que le répertoire personnel dans lequel vous vous trouvez n'est pas celui de votre installation hôte. Il s'agit d'un *home directory* contenu dans l'installation de Scratchbox.



Notre première application GTK+ lancée de manière standard en ligne de commande depuis l'environnement Scratchbox

Ce simple message très encourageant nous indique que nous pouvons commencer les joyeusetés. À savoir, le développement à destination de la plateforme Maemo. À tout moment, vous pouvez reconfigurer votre environnement via la commande **sb-menu**. Là, vous pourrez, entre autres choses, définir une cible pour votre environnement de développement (X86 ou ARMEL). Entrons, sans plus attendre dans le vif du sujet. Je vous fais grâce du classique **Hello World** en mode console, puisque nous n'avons là rien de véritablement spécifique à la plateforme. Mieux vaut ainsi nous pencher sur un code utilisant le *toolkit* GTK+ :

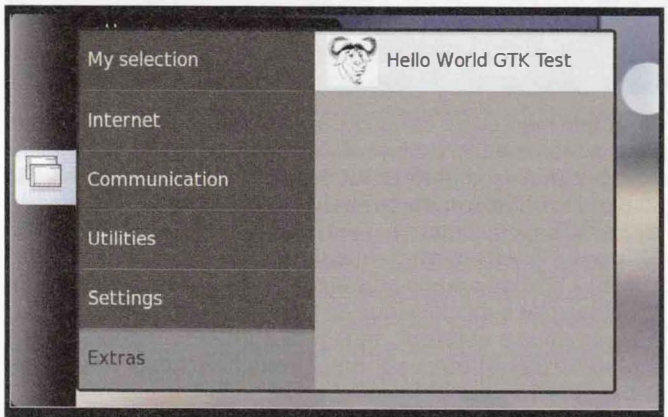


Toujours notre application de test, mais cette fois parfaitement intégrée dans l'environnement Maemo

```
# include <stdlib.h>
# include <gtk/gtk.h>
int main(int argc , char ** argv) {
  GtkWindow * window ;
  GtkLabel * label;
  gtk_init (&argc , &argv);
  window = g_object_new ( GTK_TYPE_WINDOW ,
    "border - width", 12,
    "title ", "Hello GTK+",
    NULL);
  label = g_object_new ( GTK_TYPE_LABEL ,
    "label", " Hello World !",
    NULL);
  gtk_container_add ( GTK_CONTAINER ( window ), GTK_WIDGET ( label ));
  gtk_widget_show_all ( GTK_WIDGET ( window ));
  g_print ("main: calling gtk_main \n");
  gtk_main ();
  g_print ("main: returned from gtk_main and exiting with success \n");
  return EXIT_SUCCESS ;
}
```

Que les lecteurs développant déjà des applications GTK+ se rassurent, nous avons ici un code tout à fait classique. Ce n'est que pour les fonctions ajoutées en surcouche (Hildon) que nous sortons du cadre « normal » d'utilisation des toolkits GTK+ et GNOME. La compilation de notre fichier source se fera en ligne de commandes en utilisant le script **pkg-config** nous permettant de spécifier automatiquement les options de compilation et d'édition de liens spécifiques à GTK+ :

```
[sbox-DIABLO_ARMEL: ~/N810] > gcc -Wall -g gtk_helloworld.c \
`pkg-config --cflags --libs gtk+-2.0` -o gtk_helloworld
```



L'installation d'un paquet Debian dans l'environnement de développement provoque l'apparition d'une boite de dialogue dans l'environnement graphique fonctionnant dans un serveur Xephyr.

Nous obtenons ainsi notre binaire directement utilisable. Notez qu'avec la cible ARMEL spécifiée dans la configuration, nous allons obtenir un binaire pour ARM que nous pouvons donc directement copier sur le n810 via le câble USB fourni ou une carte miniSD :

```
[sbox-DIABLO_ARMEL: ~/N810] > file gtk_helloworld
gtk_helloworld: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
for GNU/Linux 2.6.8, dynamically linked (uses shared libs),
not stripped
```

Cependant, grâce à l'émulation (CPU *transparency*), nous ne sommes pas obligé de disposer d'un n810 dans la phase de développement si l'application ne fait pas directement appel aux spécificités non émulées de la plateforme. En effet, nous pouvons très simplement lancer notre application en spécifiant le **DISPLAY** d'un serveur X ou, mieux encore, lancer un serveur X spécifique répondant aux mêmes caractéristiques que celui du matériel cible. Pour ce faire, il vous suffit d'installer Xephyr, le remplaçant de Xnest. Ce programme vous permet de lancer un serveur X disposant donc d'un nouveau **DISPLAY** reposant sur un X déjà en fonctionnement. Sous Debian, ceci se résume à l'installation du paquet **xserver-xephyr**.

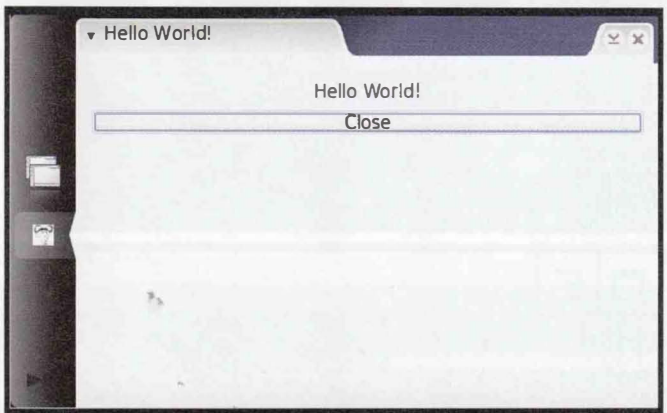
Dès lors, vous pourrez lancer (en dehors de Scratchbox) la commande suivante :

```
% Xephyr :1 -host-cursor -screen 800x480x16 \
-dpi 96 -ac -extension Composite
```

Apparaîtra, sous la forme d'une fenêtre, un nouveau **DISPLAY** X affichant classiquement le maillage X. Vous pourriez à ce stade spécifier le **DISPLAY** en positionnant la variable d'environnement correspondante et lancer notre application fraîchement compilée. Le résultat n'est cependant pas très esthétique. Sans recompiler votre application, vous allez comprendre en quoi l'environnement Maemo apporte un vrai plus pour le développeur. Ainsi, exportez la variable **DISPLAY**, puis lancez l'environnement Maemo :

```
[sbox-DIABLO_ARMEL: ~/N810] > export DISPLAY=:1
[sbox-DIABLO_ARMEL: ~/N810] > af-sb-init.sh start
```

Vous allez voir s'afficher dans votre fenêtre Xephy l'environnement tel qu'il est disponible dans votre installation Scratchbox. Un certain nombre d'éléments sont absents en comparaison de la version fonctionnant sur votre n810. En termes de développement, ce n'est pas un problème. Lancez ensuite votre application :



L'interface de l'environnement émulé n'est pas aussi riche que celle du n810. On retrouve cependant le minimum vital, dont notre paquet fraîchement compilé/construit/installé.

```
[sbox-DIABLO_ARMEL: ~/N810] > ./gtk_helloworld
```

Cette première exécution n'est pas vraiment représentative de l'aspect de notre application. En effet, celle-ci se lance comme n'importe quelle application GTK+ avec l'aspect par défaut d'une application GTK+. Bref, elle fonctionne, mais n'est pas intégrée du tout dans l'environnement. Il nous manque un bon nombre de variables d'environnement qui sont normalement définies par le lanceur Maemo ou une session Dbus. Pour régler le problème automatiquement, il vous suffit de lancer votre binaire comme ceci :

```
[sbox-DIABLD_ARMEL: ~/N810] > run-standalone.sh ./gtk_helloworld
```

Et voilà ! Votre application est fonctionnelle, tout comme votre environnement de développement. Vous voici prêt pour développer à loisir vos applications pour votre tablet PC Nokia. Si vous ne vous sentez pas l'âme d'un développeur, n'oubliez pas que vous disposez là d'un environnement de développement très orienté Debian ainsi que d'une plateforme embarqué pensée dans le même sens. Voyez-vous où je veux en venir ?

Essayez donc la commande **dpkg -l** dans votre environnement Scratchbox. Surpris ? Ce n'est pas fini. Essayez maintenant :

```
[sbox-DIABLD_ARMEL: ~/N810] > apt-get source hello-world-app
Reading package lists... Done
Building dependency tree... Done
Need to get 16.8kB of source archives.
Get:1 http://repository.maemo.org diablo/sdk/free
   hello-world-app 2.1 (dsc) [380B]
Get:2 http://repository.maemo.org diablo/sdk/free
   hello-world-app 2.1 (tar) [16.4kB]
Fetched 16.8kB in 1s (8711B/s)
dpkg-source: warning: extracting unsigned source package
   (./hello-world-app_2.1.dsc)
dpkg-source: extracting hello-world-app in hello-world-app-2.1
dpkg-source: unpacking hello-world-app_2.1.tar.gz
```

Placez-vous dans le répertoire **hello-world-app-2.1** qui vient d'être créé et lancez **dpkg-buildpackage -b -rfakeroot**. Après analyse des dépendances et compilation, un paquet **../hello-world-app_2.1_armel.deb** a été construit. Si vous êtes utilisateur de Debian, vous devez penser comme moi lors de mes premiers essais que « non, ce n'est pas possible. Ce n'est pas aussi simple ! ». Si, c'est aussi simple que (l'environnement Maemo doit être lancé) :

```
[sbox-DIABLO_ARMEL: ~/N810] > dpkg -i hello-world-app_2.1_armel.deb
(Reading database ... 28646 files and directories currently installed.)
Preparing to replace hello-world-app 2.1 (using hello-world-app_2.1_
armel.deb) ...
Unpacking replacement hello-world-app ...
Setting up hello-world-app (2.1) ...
Cache file created successfully.
maemo-launcher: invoking
  '/usr/bin/maemo-select-menu-location.launch'
```

Et dans la fenêtre Xephyr apparaît la boîte de dialogue d'installation. Il ne vous reste, là, qu'à confirmer l'installation et voici l'application exemple installée dans votre Maemo. Vous venez de compiler, construire et installer votre premier paquet pour Maemo. Encore sous le choc, jouez du **apt-get source** pour vous assurer que vous n'avez pas rêvé tout cela. Difficile de faire plus simple, non ?

Conclusion

Le paquet « Hello World » est bien plus qu'une simple application Maemo. Il s'agit d'une application parfaitement intégrée et composée d'une application graphique et d'une applet. De quoi bien débuter dans le développement pour le n810 par analyse des sources.

Comme vous venez de le constater, l'environnement livré par Nokia est mature et ouvert. Il permettra à tout un chacun (ou presque) de développer et de porter des applications vers un n810 ou un n800. Mieux encore, sans même disposer d'un exemplaire de ce tablet PC, vous voici en mesure de faire de même via Scratchbox. De quoi largement expérimenter sur cette plateforme avant de vous lancer dans l'acquisition du matériel. Le *background* Debian simplifie grandement les choses si vous êtes déjà utilisateur de la distribution. Le système de paquets et les méthodes de construction sont identiques. Ainsi, la logithèque disponible servira à la fois de ressources d'applications, de méthode de contribution (portage, débogage, etc.), mais également de source d'inspiration et de documentation. Le simple paquet source **hello-world-app** vous permettra déjà de faire connaissance avec les principes de base de Maemo. En effet, ce paquet est bien plus qu'un simple **Hello World**.

Enfin, puisqu'il faut conclure, je signalerais que les rumeurs concernant le n900 sont de plus en plus précises. Bon nombre d'utilisateurs souhaitaient voir venir un n810 offrant une connectivité GSM/GPRS intégrée (et non via bluetooth). Nokia est sur le point de répondre à ces attentes. Un certain nombre de spécifications ont déjà été annoncées pour ce nouveau jouet dont le nom de code semble être « Rover ». Une chose est sûre, le produit ne sera pas disponible en Europe avant octobre et le SDK Maemo 5 est disponible en version bêta au moment où cet article est rédigé (http://maemo.org/development/sdks/maemo_5_beta_sdk/).

La connectivité GSM/GPRS (3G ?) rapproche le n900 d'un smartphone sans pour autant en faire quelque chose de ressemblant à un HTC ou un iPhone. La structure et la maturation du produit sont totalement différentes. Personnellement, j'estime que la philosophie présente à la fois dans le produit, l'OS et la documentation rend le n810 (et sans doute le n900) très comparable à ce que je trouve dans mes machines Debian. Une certaine forme d'ordre et de cohérence que je ne retrouve pas avec Androïd par exemple. C'est une plateforme plus proche d'un PC que ne saurait l'être n'importe quel *firmware* de smartphone.

Auteur : Denis Bodor

Interfaces matérielles et OS libres pour

Auteurs

- J.-M Friedt
- G. Goavec-Merou

« A good lesson when you fly COTS stuff – make sure you know how it works. »
Glenn Reeves, *Mars Pathfinder Flight Software Cognizant Engineer*

Alors que les protocoles de communication parallèles (port imprimante, bus ISA, bus IDE, bus des processeurs) tendent à disparaître au profit des modes de communication série (USB, SATA, I2C ou SPI) – moins encombrants et plus rapides, mais plus complexes à appréhender – il devient de plus en plus difficile, à l'amateur désireux d'expérimenter, de trouver des plateformes sur lesquelles découvrir le fonctionnement d'un processeur. La console de jeu portable de Nintendo Dual Screen (DS) est suffisamment ancienne tout en ayant eu assez de succès pour être encore largement disponible pour fournir une plateforme d'expérimentation intéressante.

Chaque génération de console portable Nintendo conserve une compatibilité avec son prédécesseur, et, ainsi, la DS fournit un port de communication compatible avec le Gameboy Advance. Ce port se caractérise par l'accès à un bus 16 bits similaire à celui qui équipait le microprocesseur 8086 **[1, p. 156]**, avec accès à une interruption, aux bus de données et d'adresse, ainsi qu'aux signaux de contrôle associés.

Nous proposons ici d'exploiter la Nintendo DS en exécutant un système GNU/Linux pour se familiariser avec quelques méthodes d'instrumentation, d'interfaçage et de contrôle de périphériques. Ayant constaté que les ressources requises par uClinux sont excessives devant celles mises à disposition par la console, nous nous tournerons vers un environnement de développement d'applications à faible empreinte mémoire, RTEMS. Nous verrons que la console de jeu, malgré ses ressources réduites, propose un environnement de développement propice aux découvertes tant du point de vue matériel que logiciel, avec, au final, sa conversion en système d'acquisitions de données et de transmission par wifi.

1 Matériel disponible

La console de jeu Nintendo DS – et son évolution plus récente DS Lite que nous utiliserons ici – est un système embarqué contenant deux processeurs ARM : un ARM9 (67 MHz) comme processeur principal et un ARM7 (33 MHz) **[2]** comme coprocesseur chargé de la gestion des périphériques tels que son, graphisme 2D, écran tactile. Deux ports permettent de charger des exécutables pour les processeurs : le slot 1 supporte des cartouches exclusivement dédiées à la DS, avec protocole de communication série en partie crypté, tandis que le slot 2 est compatible avec les cartouches de GameBoy Advance, donnant accès à un bus de communication parallèle très bien documenté. Les 4 MB de RAM et les deux écrans de 256×192 pixels rendent le développement plus agréable et ludique qu'une carte nue dont les seuls ports de communication sont Ethernet ou série. Évitez la version la plus récente de la console – DSi – qui a perdu le slot 2 au profit d'une quantité de RAM plus importante.

Notre objectif est d'exploiter divers outils libres disponibles pour ces processeurs bien connus des développeurs de systèmes embarqués – et notamment la version pour ARM de gcc – pour exploiter au mieux cette console. La quantité de mémoire proposée est suffisante pour supporter un système d'exploitation : nous allons donc développer des applications au-dessus d'un environnement qui nous protège des couches les plus basses entre logiciel et matériel, pour n'utiliser que des méthodes génériques, exploitables dans d'autres contextes que sur la console de jeu. En commençant par le port pour Nintendo DS de uClinux, nommé DSLinux, nous nous accrochons le plus longtemps possible à l'environnement de travail des lecteurs de ce journal, avant de devoir l'abandonner faute de ressources lorsque nous voudrons utiliser des périphériques plus gourmands tels que le wifi.

Nintendo DS : DSLinux et RTEMS

2 DSLinux : toolchain et fonctionnalités

L'exécution de nos propres logiciels, et, en particulier, de la distribution d'uClinux dédiée à la Nintendo DS nommée DSLinux[1], nécessite une cartouche permettant de contourner certaines protections censées interdire l'utilisation de logiciels autres que ceux validés par Nintendo. En effet, les jeux sont transférés de la cartouche dans le slot 1 selon un protocole synchrone série dans un format crypté lors de l'allumage de la console. L'exécution de logiciels libres sur cette console nécessite de fournir de telles données cryptées : ces fonctionnalités sont par exemple fournies par la cartouche M3DS Real[2] .Nous avons utilisé cet outil, acquis pour environ 25 euros auprès de www.consoleup.com. Cette cartouche fournit un médium de développement confortable puisqu'il accepte des cartes mémoire au format MicroSD (1 GB pour environ 5 euros), facilement lisible sur PC, avec son formatage original VFAT. Un ensemble de logiciels[3] dédiés doit être désarchivé dans la racine de cette carte pour permettre le chargement des applications stockées dans le répertoire **nds**. Nous compilerons donc nos logiciels au moyen d'une *toolchain* de *crosscompilation* sur PC et transférerons le logiciel sur carte MicroSD en vue de son exécution sur la console de jeu.

Le site web de DSLinux propose une toolchain au format binaire[4] : exceptionnellement, nous allons nous autoriser à ne pas recompiler à la main nos outils, mais allons nous contenter de décompresser et désarchiver ces outils dans un répertoire dédié que nous ajouterons à notre **$PATH**.

Une fois la toolchain disponible, il nous faut installer l'ensemble de l'arborescence de DSLinux : même si nous ne désirons pas aborder le développement de modules noyau (qui nécessitent les sources du noyau pour compiler), l'arborescence de DSLinux contient quelques utilitaires nécessaires à la compilation de programmes en espace utilisateur. Là encore, nous nous contentons de décompresser et désarchiver le fichier fourni à http://stsp.spline.de/dslinux/dslinux-snapshot.tar.gz.

Un clavier virtuel dessiné sur l'écran tactile permet de facilement interagir avec ses programmes en mode console, avec notamment la tabulation qui rend l'interaction avec le shell efficace. Les touches en forme de croix à gauche de la console font office de flèches.

Dans un premier temps, nous allons développer quelques petits programmes en espace utilisateur : après avoir inclus **dslinux-toolchain-2008-01-24-i686/bin/** dans son **$PATH**, nous entrons dans la racine de l'arborescence DSLinux et la commande **make xsh** initialise les variables nécessaires à la compilation de programmes. Ainsi, pour compiler un programme nommé **rumble.c**, nous utilisons **$CC $CFLAGS $LDFLAGS rumble.c -o rumble**. Le résultat de la compilation est, comme toujours sous uClinux, un binaire au format Binary Flat (BFLT). Notez que pour pouvoir exécuter **make xsh** avec succès, il faut avoir effectué un **make menuconfig**, dont nous accepterons toutes les options par défaut, au moins une fois afin de générer les fichiers de configuration nécessaires à la mise en place de l'environnement de cross-compilation.

Avant de recompiler son propre noyau ciselé à ses besoins, nous nous contentons d'exploiter une archive Linux précompilée et fonctionnelle, disponible à http://kineox.free.fr/DS/dslinux-dldi.tgz. L'arborescence du système est placée à la racine de la carte MicroSD, tandis que les deux fichiers aux extensions **.nds** vont dans le répertoire **nds** (un des fichiers est un noyau avec support de la communication avec la carte SD – DLDI (*Dynamically Linked Device Interface*) – et l'autre fichier est un noyau supportant en plus les extensions mémoire. En l'absence d'un tel périphérique, le lecteur n'utilisera que le logiciel **dslinux.nds**). Ainsi, GNU/Linux apparaît comme un jeu pour la console, dont l'arborescence est placée sur la carte SD pour une modification aisée : les fichiers fondamentaux au démarrage de GNU/Linux sont dans le **rootfs** avec notamment un répertoire **bin** contenant les outils nécessaires à l'initialisation du système, tandis que tous les fichiers de configuration, utilisateur, etc. se trouvent dans le répertoire **linux** en haut de l'arborescence sur la carte MicroSD, monté dans **/media** au démarrage.

```
jmfriedt@ns39351:~/dslinux/romfs$ ls -l | cut -c 53-100
 bin
 boot
 dev
 etc -> media/linux/etc
 home -> media/linux/home
 lib -> media/linux/lib
 media
 opt
 proc
 sbin
 tmp
 usr -> media/linux/usr
 var -> media/linux/var
```

La compilation manuelle d'une image s'obtient par le classique **make menuconfig** suivi de **make** : le résultat est, d'une part, une arborescence disponible dans le sous-répertoire **rootfs**, et, d'autre part, l'archive compressée **images/dslinux-dldi.tgz** qui contient le « jeu » **dslinux.nds** pour lancer uClinux sur la DS, et l'arborescence **linux** que nous placerons dans le répertoire racine de la carte MicroSD. Mentionnons dès maintenant que l'ajout de fonctionnalités se fait dans **vendors/Nintendo/DLDI** : par exemple, l'ajout d'une entrée dans **/dev** (major 32, minor 0) se fait dans **vendors/Nintendo/DLDI/Makefile** en complétant la liste des **DEVICES** par **skeleton,c,32,0 ** ou l'accès automatique à des systèmes de fichiers dans **inittab**. Toute modification dans un autre emplacement de l'arborescence **romfs** sera perdue à la prochaine compilation.

Ayant obtenu un système d'exploitation fonctionnel sur DS, nous allons proposer d'en exploiter les fonctionnalités accessibles compte tenu de la quantité réduite de mémoire disponible (4 MB initialement, quelques centaines de KB après chargement du système) : notre objectif va être d'interagir avec le monde extérieur en commandant des systèmes numériques (LED, selon des méthodes applicables à toute commande binaire) et en acquérant une valeur analogique. Comme DSLinux est basé sur uClinux, le processeur ARM9 de la DS

n'étant pas équipé de gestionnaire de mémoire, le contrôle de ces périphériques pourra se faire soit depuis l'espace utilisateur (comme nous le ferions sur un microcontrôleur monotâche sans système d'exploitation), soit depuis l'espace noyau pour tenir compte des couches d'abstraction isolant le développeur sous GNU/Linux des couches matérielles.

3 Hello World

Nous avons vu qu'un programme se compile en passant dans l'environnement de cross-compilation proposé dans la toolchain DSLinux : **`make xsh`** suivi, pour la compilation elle-même, de **`$CC $CFLAGS $LDFLAGS rumble.c -o rumble`** dans le répertoire de développement. Nous pouvons ainsi valider notre capacité à générer un binaire au bon format pour être exécuté sous DSLinux depuis un emplacement de l'arborescence contenue dans la MicroSD, de communiquer avec l'utilisateur en mode texte et d'effectuer des opérations simples sur les entiers et les réels (code Tab. 1).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

int main(int argc,char **argv)
{printf("demo rumble : 1/3=%f\n",1./3.);
 if (argc>1) {
   *(unsigned short*)(0x8000000)=
         (unsigned short)atoi(argv[1]);
   sleep(1);     // active le moteur sur argv[1]=2
   *(unsigned short*)(0x8000000)=0;
 }
 return(0);
}
```

Tableau 1 : Programme de démonstration d'un affichage en mode console et l'accès aux périphériques du slot 2 depuis l'espace utilisateur (section 5). Si le rumble pack est inséré dans ce slot, le moteur s'activera pendant une seconde si l'argument est un nombre pair.

Notons que ces fonctionnalités simples qui n'accèdent pas au matériel peuvent rapidement être validées sur l'émulateur **desmume** (Fig. 1), disponible comme paquet Debian au moins avec la version Lenny. Il semblerait que les versions plus récentes incluses dans Squeeze et Sid corrigent un certain nombre d'erreurs. Cet émulateur permet soit de valider un algorithme sur PC en évitant de copier son programme sur MicroSD et booter uClinux à chaque nouvel essai, soit au lecteur ne possédant pas (encore) de console de jeu de tester quelques-unes des idées présentées dans ce document. Par ailleurs, une fonctionnalité intéressante de l'émulateur est la capacité à afficher l'état de certains registres et, pour la version MS-Windows, de désassembler le contenu de la mémoire, en faisant donc un outil de débogage relativement pratique.

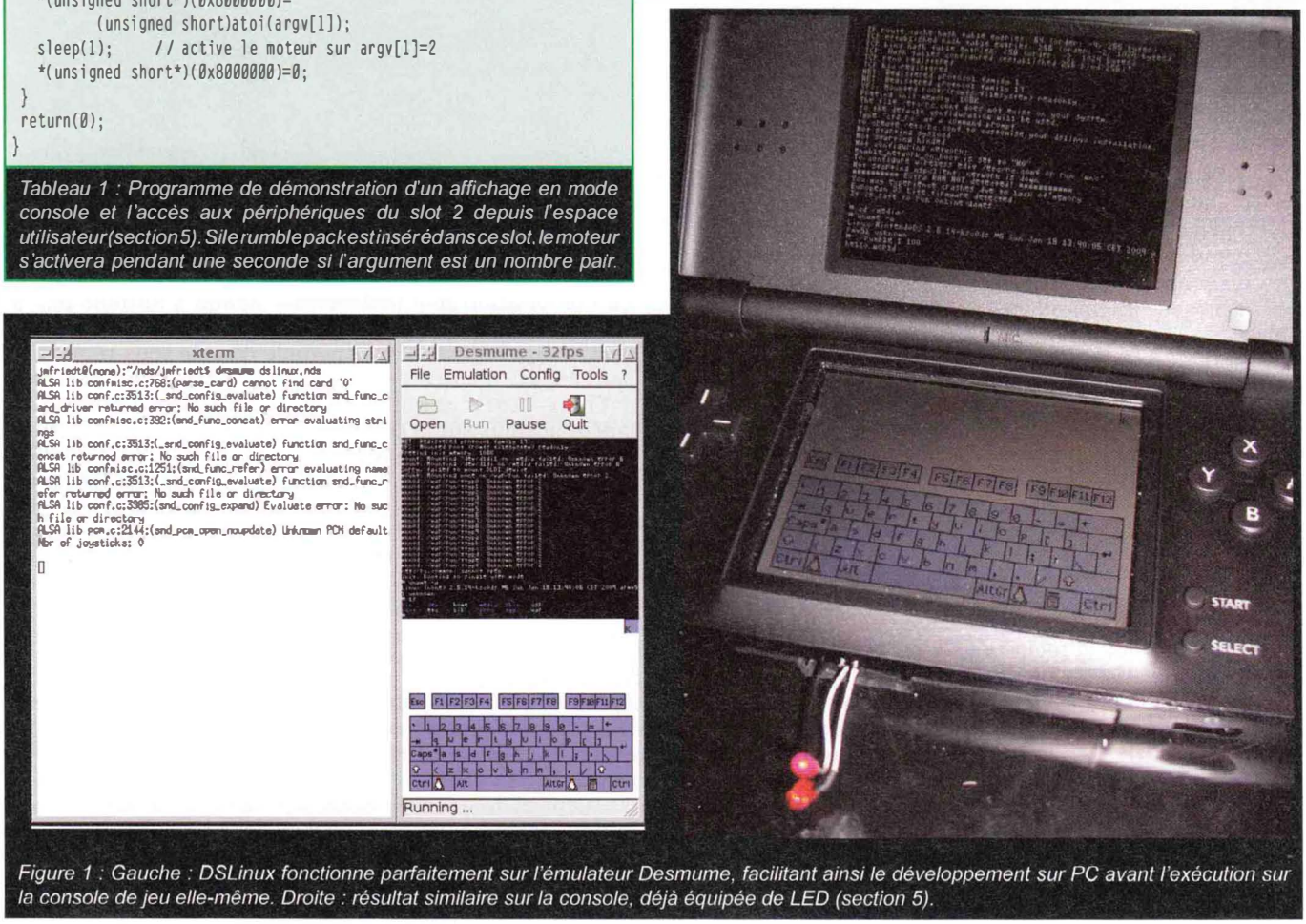


Figure 1 : Gauche : DSLinux fonctionne parfaitement sur l'émulateur Desmume, facilitant ainsi le développement sur PC avant l'exécution sur la console de jeu elle-même. Droite : résultat similaire sur la console, déjà équipée de LED (section 5).

Avez-vous l'âme du collectionneur ?

Boostez votre collection !

Vous recherchez un magazine en particulier ? Allez sur www.ed-diamond.com pour voir le sommaire détaillé de chaque magazine et ensuite... Boostez votre collection avec les « Power packs x5 », soit 5 GNU/Linux Magazine pour 15€ et les « Power packs x10 », soit 10 GNU/Linux Magazine pour 25€ à choisir dans la liste ci-dessous :

Vous pouvez commander :

Par courrier
En nous renvoyant ce bon de commande.

Par le Web
Sur notre site : www.ed-diamond.com.

Choisissez vos numéros dans le tableau ci-dessous*
* Seuls les numéros ci-dessous sont disponibles pour une commande de Power Packs x5 et x10

N°06	GNOME - The Gimp	N°57	Maîtrisez la gestion... Slots & Signaux ... des événements en C++	N°94	PABX Vidéo avec Asterisk
N°07	Dopez Linux	N°58	Djbdns enfin une alternative viable à BIND !	N°95	Administration et configuration centralisées avec Cfengine
N°09	Prêt pour le jeu !	N°59	Zopix, Créez un CD "Live" Zope en 10 minutes !	N°96	Géolocalisation de photos numériques
N°10	The HURD : 100% GNU	N°60	JBoss serveur d'applications J2EE OpenSource	N°97	Haute-disponibilité & équilibrage de charge
N°11	Exclusif : l'avenir de G.N.O.M.E	N°61	Découvrez MySQL 5 et les procédures stockées	N°98	Supervision avec NAGIOS
N°12	NT et Linux : Guerre ou complément ?	N°62	Créez votre OS, principe et implémentation	N°99	Java & JNI Tirez le meilleur de Java
N°13	Cryptage : la clé de la sécurité	N°63	Les threads : kernel 2.6 et 2.4	N°100	Le serveur PARFAIT
N°14	XFree 4.0 : le futur à notre portée	N°64	Adamoto	N°101	Chiffrez vos disques
N°15	Passez à la vitesse supérieure	N°65	Théorie et pratique : Supervision avec Nagios	N°102	LTSP 5 Clients légers
N°16	OpenSources : Est-ce suffisant?	N°66	Créez votre Distribution Live	N°103	PostgreSQL 8.3
N°17	Linux : Système embarqué	N°67	C# .NET	N°104	Gestion des services avec OpenSolaris
N°18	Spécial interview : l'avenir de Linux	N°68	Le crash disque vous guette	N°105	Domotique avec MisterHouse
N°19	Dossier spécial : Postgre SQL 7.0	N°69	La réponse de Sun à Linux ? SOLARIS 10	N°106	Récupérez vos données après une défaillance RAID
N°22	Le multi-hreading : Une manière moderne de programmer le Multitâche	N°70	Découvrez et comprenez la technologie GRID	N°107	Sauvegardez vos bases de données MySql
N°24	Palm et Linux	N°71	Présentation et installation du Hurd		
N°25	Kernel 2.4.0	N°72	Services Web... C/C++ et gSOAP		
N°26	<Dossier> XML </Dossier>	N°73	Compression théorie algorithmes et programmation		
N°27	Les systèmes de fichiers journalisés	N°74	VFS : Système de fichiers virtuel		
N°28	Scripting : la force d'Unix	N°75	Tuning de code		
N°29	L.F.S. Linux From Scratch	N°76	Algorithmes évolutionnistes		
N°30	Le chiffrement des données	N°77	Systèmes de fichiers chiffrés		
N°32	Changez de coquille	N°78	Bluetooth, Spécifications, protocoles et configuration		
N°34	XSL - FO : TeX Killer ?	N°79	Sécurisation du Noyau avec PAX		
N°35	QoS et iproute : optimisation et contrôle du trafic IP	N°80	Run in memory		
N°37	L'impression sous Linux	N°81	Comment fonctionnent les générateurs de nombres pseudo-aléatoires		
N°38	Le desktop Shell : Enlightenment	N°82	eCos, une solution libre pour systèmes embarqués		
N°39	Sécurité : Patchez votre noyau !	N°83	Greylist Eliminez le SPAM à la racine		
N40	MySQL : la base de donnée OpenSource	N°84	Déploiement de hotspots Wifi sécurisés		
N°41	Steganographie ou l'art de la dissimulation de données	N°85	Firewall : Netfilter & NuFW		
N°44	Comprenez NetBios pour Maîtriser l'interopérabilité windows GNU Linux	N°86	Serveur SMTP: Routage des mails avec Postfix		
N°48	Caudium, votre prochain serveur Web !	N°87	Le point sur Mono .NET Java et les Brevets		
N°49	Après MySQL & PostgreSQL SAP DB : La base de données libre & puisante	N°88	Sécurité: Smartcards & Tokens		
N°50	Créez un album Photo avec PHP ...et sans MySQL	N°89	Utilisation avancée de XEN		
N°52	Linux Temps réel où en est-on aujourd'hui ?	N°91	Ajax Avancé		
N°53	Linux sur PDA : Linux dans votre poche !	N°92	Paravirtualisation XEN & SLO Répartition de charge		
N°56	Développez vos applications Mozilla avec XPFE & XPCOM	N°93	Développez vos extensions Firefox		

Numéros GNU/Linux Magazine épuisés
N°01, N°02, N°03, N°04, N°05, N°08, N°20, N°21, N°23, N°31, N°33, N°36, N°42, N°43, N°45, N°46, N°47, N°51, N°55, N°54 et N°90

Bon de commande power packs

à remplir (ou photocopie) et à retourner aux Éditions Diamond – GNU/Linux Magazine – BP 20142 – 67603 sélestat Cedex

	OUI, je désire acquérir un power pack X5	1er 1PP* X5	2éme 2PP* X5	3éme 3PP* X5
Cochez ici ▲ POWER PACKS X5	1, GNU/Linux Magazine N°			
	2, GNU/Linux Magazine N°			
	3, GNU/Linux Magazine N°			
	4, GNU/Linux Magazine N°			
	5, GNU/Linux Magazine N°			
	Total par série de POWER PACKS X5 :	15 €	30 €	45 €
	OUI, je désire acquérir un power pack X10	**1er 1PP* X10**	**2éme 2PP* X10**	**3éme 3PP* X10**
Cochez ici ▲ POWER PACKS X10	1, GNU/Linux Magazine N°			
	2, GNU/Linux Magazine N°			
	3, GNU/Linux Magazine N°			
	4, GNU/Linux Magazine N°			
	5, GNU/Linux Magazine N°			
	6, GNU/Linux Magazine N°			
	7, GNU/Linux Magazine N°			
	8, GNU/Linux Magazine N°			
	9, GNU/Linux Magazine N°			
	10, GNU/Linux Magazine N°			
	Total par série de POWER PACKS X10 :	25 €	50 €	75 €
	Les hors-séries et numéros spéciaux sont exclus des POWER PACKS. Montant TOTAL 15€ + 3,81€ de frais de port. Le TOTAL s'élève à 18,81€ pour l'achat d'un POWER PACK x5. SEULEMENT EN FRANCE MÉTROPOLITAINE ! *PP= POWER PACK		TOTAL :	
			FRAIS DE PORT :	+3,81 €
			TOTAL :	

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Je choisis de régler par :

❑ Chèque bancaire ou postal à l'ordre de Diamond Editions

❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Complétez votre collection des anciens numéros de...

Vous pouvez commander :

Par courrier
En nous renvoyant ce bon de commande.

Par le Web
Sur notre site : www.ed-diamond.com.

GNU/LINUX MAGAZINE

GNU/LINUX MAGAZINE HORS-SÉRIE

Bon de commande GNU/Linux Magazine

Réf.	Désignation	Prix / N°s
LM108	**Linux Magazine 108** Comment vos utilisateurs traversent votre Firewall	6,50 €
LM109	**Linux Magazine 109** Port Knocking - Ne laissez pas la porte ouverte aux intrusions	6,50 €
LM110	**Linux Magazine 110** Sécurisez votre système en le migrant vers LVM2 & RAID 1	6,50 €
LM111	**Linux Magazine 111** 18 Recettes pour tirer le meilleur de OPENLDAP	6,50 €
LM112	**Linux Magazine 112** Besoin d'une solution centralisée et efficace d'ADMINISTRATION SYSTÈME ? Installez et déployez Puppet !	6,50 €
LM113	**Linux Magazine 113** Single Sign-On / SSO et authentification centralisée avec CAS-TOOLBOX	6,50 €
LM114	**Linux Magazine 114** Découvrir, installer et configurer - Passer à ZFS ! Le système de fichiers révolutionnaire	6,50 €
LM115	**Linux Magazine 115** Créez et personnalisez votre système DEBIAN 5.0 LIVE sur CD/DVD ou clef USB	6,50 €
LM116	**Linux Magazine 116** Construisez et administrez votre VPN L2TP/IPSEC avec OpenSSL, Openswan, PPP et XL2TPD	6,50 €
LM117	**Linux Magazine 117** Base de données orientée document : APACHE COUCHDB - Introduction, installation et utilisation	6,50 €

Bon de commande GNU/Linux Magazine Hors-série

Réf.	Désignation	Prix / N°s
LMHS27	**LM HS 27** Électronique et Linux	6,40 €
LMHS28	**LM HS 28** Administration système avec Debian	6,40 €
LMHS29	**LM HS 29** BSD Acte 1	6,40 €
LMHS30	**LM HS 30** BSD Acte 2	6,40 €
LMHS31	**LM HS 31** Spécial TheGIMP	6,40 €
LMHS32	**LM HS 32** VIRUS Unix, Gnu/Linux & MacOSX	6,40 €
LMHS33	**LM HS 33** Ruby & Ruby on Rails	6,40 €
LMHS34	**LM HS 34** Dominez votre système et allez au-delà de l'interface graphique...	6,50 €
LMHS35	**LM HS 35** Serveur Web, Installez, configurez et optimisez votre serveur HTTP !	6,50 €
LMHS36	**LM HS 36** Serveur Mail, Installez, configurez et optimisez votre serveur SMTP !	6,50 €
LMHS37	**LM HS 37** Serveur Dédié, Supervision, VoIP, VPN, Syslog...	6,50 €
LMHS38	**LM HS 38** Électronique Embarqué & Domotique	6,50 €
LMHS39	**LM HS 39** Cartes à puce administration et utilisation	6,50 €
LMHS40	**LM HS 40** Explorez les richesses du langage PYTHON	6,50 €
LMHS41	**LM HS 41** Configurez et optimisez votre FIREWALL	6,50 €
LMHS42	**LM HS 42** Surpervision & Surveillance - Gardez un œil sur votre réseau et vos utilisateurs	6,50 €

Bon de commande

à remplir (ou photocopie) et à retourner aux Éditions Diamond - GNU/Linux Magazine - BP 20142 - 67603 sélestat Cedex

	Référence	Prix / N°s	Qté	Total
EXEMPLE :	LM111	6,50 €	1	6,50 €
		TOTAL :		
		FRAIS DE PORT FRANCE MÉTRO :		+3,81 €
		FRAIS DE PORT ÉTRANGER :		+5,34 €
		TOTAL :		

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Je choisis de régler par :

☐ Chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Retrouvez les sommaires et commandez tous nos magazines sur notre site :
http://www.ed-diamond.com

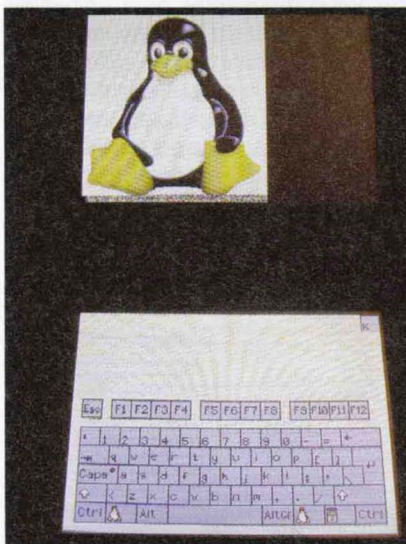
4 Accès au framebuffer et programmation sous GNU/Linux

Notre objectif étant de tracer des courbes présentant des points expérimentaux acquis selon les méthodes présentées par la suite (section 5.2), nous désirons maîtriser une méthode d'affichage de graphiques sur les écrans de la console. Nos 4 MB de mémoire, en grande partie occupés par le système d'exploitation, sont largement insuffisants pour faire tourner une application compilée avec Qtopia, et même la bibliothèque SDL semble trop gourmande pour fonctionner avec les ressources disponibles. Il ne reste donc que la solution la plus efficace, la plus rapide et requérant le moins de mémoire volatile : l'accès au *framebuffer* **[3]**.

Afin de nous familiariser avec la compilation de programmes pour DSLinux, et en particulier pour accéder à ce périphérique qui n'est autre qu'un segment de la mémoire représentant la couleur de chaque pixel à l'écran, nous nous fixons pour objectif d'afficher une image couleur au format PNM (Fig. 2).

Le programme est très classique, avec ouverture du périphérique (`open("/dev/fb0", O_RDWR);`), récupération des paramètres de l'écran par `ioctl()`, pour finalement modifier la couleur des pixels composant l'image selon les consignes fournies dans le fichier PNM. La principale subtilité a été d'identifier l'encodage des couleurs dans un framebuffer configuré pour définir la couleur de chaque pixel sur 16 bits : les couleurs sont chacune codées sur 5 bits, avec le bit de poids le plus fort systématiquement à 1. Cela signifie que nous masquons les 3 octets représentant les 3 couleurs lues dans le fichier PNM pour générer un mot de 16 bits contenant la concaténation de 3 couleurs sur 5 bits chacune. Ainsi, nous utilisons

```
for (y=0;y<sy;y++)
  for (x=0;x<sx;x++) {
    fscanf(f,"%c",&r); fscanf(f,"%c",&g); fscanf(f,"%c",&b);
    bpp=(((int)(r&0xf8))>>3)+(((int)(g&0xf8))<<2)+(((int)(b&0xf8))<<8);
    draw_pixel(x, y, bpp);
   }
```



```
struct fb_var_screeninfo sinfo;
unsigned short * s_ptr;

inline void draw_pixel(int x, int y, int color)
{unsigned short *loc = s_ptr + \
    ((y+sinfo.yoffset)*sinfo.xres)+x+sinfo.xoffset;
 *loc = color;     // 5R, 5G, 5B
 *loc |= 1 << 15; // transparence ?
}

int main(int argc, char *argv[])
{char c;
 screen_fd = open("/dev/fb0", O_RDWR);
 ioctl(screen_fd, FBIOGET_VSCREENINFO, &sinfo);

 s_ptr = mmap(0,screen_height*screen_width/8,
    PROT_READ|PROT_WRITE,MAP_SHARED,
    screen_fd, 0);
 [...]
}
```

Figure 2 : Exemple d'affichage sur l'écran du haut accessible par /dev/fb0. L'affichage d'une image en couleur démontre la bonne interprétation de l'organisation de la mémoire et de l'affectation des couleurs au sein de chaque mot.

pour tracer les `sx * sy` pixels qui forment l'image. La fonction `draw_pixel` et le résultat sont tous deux proposés sur la fig. 2. Insistons sur l'intérêt du système d'exploitation : aucune connaissance sur l'architecture matérielle ou l'emplacement de la mémoire vidéo de la NDS n'est nécessaire pour cet exemple, qui est par ailleurs directement portable sur un PC.

Notez qu'avant d'exécuter un tel programme qui nécessite un écran dont la couleur de chaque pixel est définie sur 16 bits, on pensera à lancer depuis le shell `fbset -depth 16 -n`.

5 Interfaçage de périphériques : la cartouche du slot 2 (Gameboy Advance)

Bien que la Nintendo DS propose une architecture considérablement moins complexe (et fournissant moins de puissance de calcul) que ses concurrentes, la compatibilité avec la GameBoy Advance, plus ancienne, est l'opportunité d'exploiter toute la documentation associée aux protocoles de communication de cet ancêtre de la DS. Nous trouverons notamment facilement sur le web les plans du bus de communication parallèle accessible sur la cartouche (www.reinerziegler.de/GBA) qui va s'avérer un environnement de prototypage idéal, digne des bus ISA et autres ports parallèles malheureusement disparus de nos PC modernes.

Nous pourrons, dans un premier temps, nous familiariser avec la cartouche du Rumble Pack : ce périphérique, commercialisé en même temps que la cartouche de programmation citée plus haut, est aussi simple d'un point de vue électronique qu'inutile d'un point de vue ludique. Nous n'aurons donc aucun remord à la sacrifier pour nos expérimentations.

Nous allons, dans un premier temps, identifier la plage d'adresses mémoire permettant l'accès au bus du slot 2 de la DS, que nous démontrerons en activant et désactivant le moteur du Rumble Pack. Ce moteur sera ensuite retiré pour être remplacé par des périphériques plus intéressants.

5.1 Accès à une plage mémoire en sortie

Avant d'attaquer au fer à souder la cartouche Rumble Pack, nous pouvons déjà valider quelques informations sur la plage d'adresses permettant l'accès aux périphériques sur slot 2 en activant et désactivant le moteur. L'observation (Fig. 3) de la cartouche[5] indique que seuls les signaux WR# et AD1 sont câblés : l'activation se fait donc soit lors de l'écriture sur une adresse paire, soit en écrivant une valeur paire dans la plage d'adresses de communication avec la cartouche[6]. Le fait que WR# (actif lors de la phase de communication des données) et non CS# (actif pendant les phases de communication des adresses et des données, avec une transition d'état sur une adresse valide) soit utilisé laisse penser que la seconde solution est la bonne. En effet, les broches AD0 à AD15 fournissent deux fonctions : dans un premier temps, le mot de poids faible de l'adresse (lorsque CS# est bas mais WR# encore haut), suivi dans un second temps de la donnée transmise (sur 16 bits, lorsque CS# et WR# sont tous deux bas). La première phase est valide sur le front descendant de CS#, tandis que la seconde phase est active avec une donnée valide sur le front montant de WR#. Par conséquent, l'instruction `*(unsigned short*)0x8000000=2` active le moteur, et `*(unsigned short*)0x8000000=0` l'arrête (code Tab. 1).

Les signaux accessibles sur le connecteur du slot 2 n'appartiennent pas à proprement parler à un bus, même s'ils en fournissent ponctuellement les fonctions : le décodage d'adresse se fait en amont, puisque, par exemple, le signal WR# ne s'active que lors de l'accès à la plage d'adressage de la carte.

L'ajout ou le remplacement de fonctionnalités dans la cartouche du Rumble Pack peut se faire de deux façons : tirer un câble en nappe en dehors de la cartouche pour prototyper sans contrainte d'encombrement, ou respecter le facteur de forme de la cartouche. La première solution semble trop fragile à l'usage : nous nous sommes efforcés de placer tous les composants dans la cartouche, ce qui impose d'exploiter des composants de petites dimensions prévus pour être montés en surface (CMS, Fig. 3 au milieu).

L'utilisation de tels composants n'est pas difficile, mais nécessite de la soudure fine, de la tresse à dessouder pour corriger ses erreurs, et surtout une binoculaire pour travailler avec un grossissement suffisant.

Cette méthode d'accès direct aux bus de communication entre le processeur et le matériel ne respecte cependant pas les principes d'abstraction mis en place par le système d'exploitation : nous verrons plus bas comment obtenir les mêmes fonctionnalités au travers d'un module noyau accessible via son entrée dans `/dev` : nous aborderons ce point avec l'écriture d'un module noyau.

5.2 Lecture d'une conversion analogique-numérique

Le bus de données est bidirectionnel, la direction de communication étant déterminée par les signaux de contrôle WR# (écriture) ou RD# (lecture). Un composant susceptible de fournir des informations au processeur via son bus de données doit présenter un état au repos en haute impédance dans lequel il n'impose pas l'état du bus (état généralement noté Z dans la description des composants), pour ne passer en basse impédance et imposer l'état du bus que sur ordre du processeur (transition au niveau bas du signal de contrôle RD#). Toute erreur de câblage en ce sens se traduira par des courts-circuits potentiels entre composants imposant simultanément leur niveau sur le bus, avec potentiellement destruction des composants les moins bien protégés contre la consommation excessive de courant.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define TAILLE 255
```

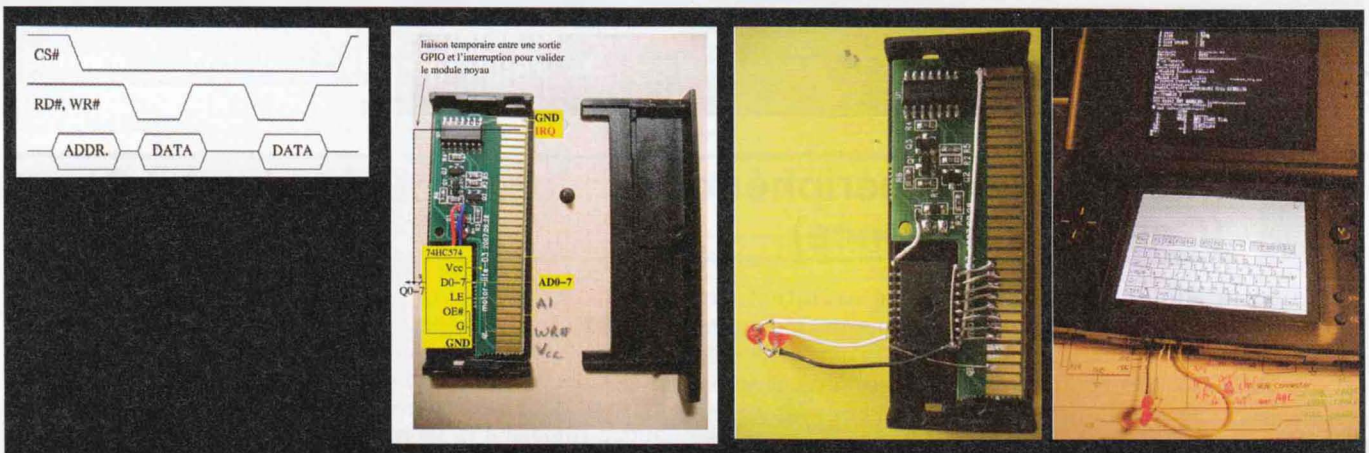


Figure 3 : Chronogramme des signaux disponibles sur le bus du slot 2 de la DS : un signal d'activation de la carte Chip Select, des signaux de contrôle indiquant la direction de la transaction (RD# et WR#) et 16 bits d'adresse dont le mot de poids faible est multiplexé avec le bus de données. Un circuit exploitant l'ensemble de la plage d'adresses doit donc mémoriser (latch) la valeur du mot de poids faible du bus d'adresses sur le front descendant de CS# avant d'exploiter les données fournies sur ces mêmes broches sur le front montant de RD# ou WR#.

```
int main(int argc,char **argv)
{int f,taille=TAILLE;
 volatile int k;
 char *c;

 c=(char*)malloc(TAILLE); // demontre malloc en l'absence de MMU
 for (f=0;f<TAILLE;f++)
   {*(unsigned short*)(0x8000000)=(unsigned short)0;
    for (k=0;k<10;k++) {} // NE PAS compiler en -O2
    // usleep(7);         // l'appel a usleep est trop long !
    c[f]=*(unsigned short*)(0x8000000)&0xff;
   }
 for (f=0;f<TAILLE;f++) printf("%x ",c[f]);printf("\n");
 return(0);
}
```

Tableau 2 : Lecture d'une valeur fournie par un périphérique connecté au bus du slot 2. Dans le cas qui nous intéresse ici de la conversion analogique-numérique, la conversion est amorcée par une écriture tandis que la lecture se fait après un temps prédéfini obtenu par une boucle vide pour une fréquence d'échantillonnage maximale.

Nous allons exploiter un composant de conversion analogique-numérique, AD7492, comme exemple de système capable de fournir une information au processeur (Fig. 4). Le convertisseur est un cas un peu plus complexe que la moyenne, car il nécessite dans un premier temps un ordre en écriture pour annoncer le début de la conversion (impulsion au niveau bas sur la broche CONVST# du convertisseur), et ce n'est qu'une fois la conversion achevée que le composant est prêt à fournir le résultat de sa mesure (données sur 12 bits placés sur D0 à D11 lorsque les signaux d'activation des sorties CS# et RD# passent à l'état bas). Dans un premier temps, nous nous contenterons d'attendre un délai prédéfini entre début et fin de conversion, pour aborder plus tard le cas du message (interruption) annonçant la fin de conversion (associée à une transition du signal BUSY).

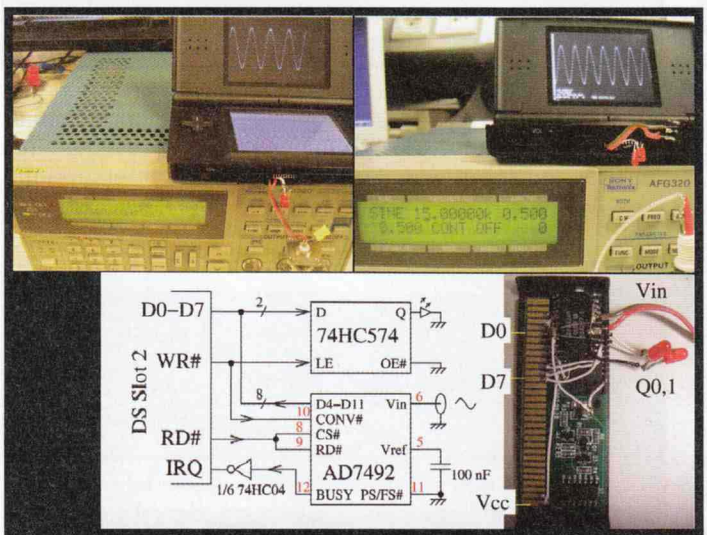


Figure 4 : En haut : un synthétiseur de fréquences génère un signal sinusoïdal périodique d'amplitude 1 V et d'offset 0,5 V à des fréquences de 1 Hz (gauche, représentatif d'une mesure d'une grandeur telle qu'une température) et 15 kHz (droite, pour évaluer la fréquence maximale d'échantillonnage) en vue de la conversion par le circuit décrit en bas à gauche, connecté sur une cartouche de Rumble Pack (en bas à droite) insérée dans le slot 2 de la console. Le programme de gestion de la conversion (Tab. 2) et d'affichage sur framebuffer tourne sous DSLinux. La soudure des composants CMS n'est pas difficile mais nécessite une loupe ou une binoculaire.

Une amélioration possible du concept est d'exploiter pleinement les capacités du bus de données de 16 bits de large pour câbler tous les signaux du convertisseur analogique-numérique 12 bits. Les modifications du matériel sont significatives (4 fils de plus, Fig. 5) tandis que du point de vue logiciel, nous nous contentons de remplacer le type du tableau dans lequel les informations sont stockées de **`unsigned char*`** à **`unsigned short*`** qui comprendra les 12 bits de données significatives (code Tab. 2).

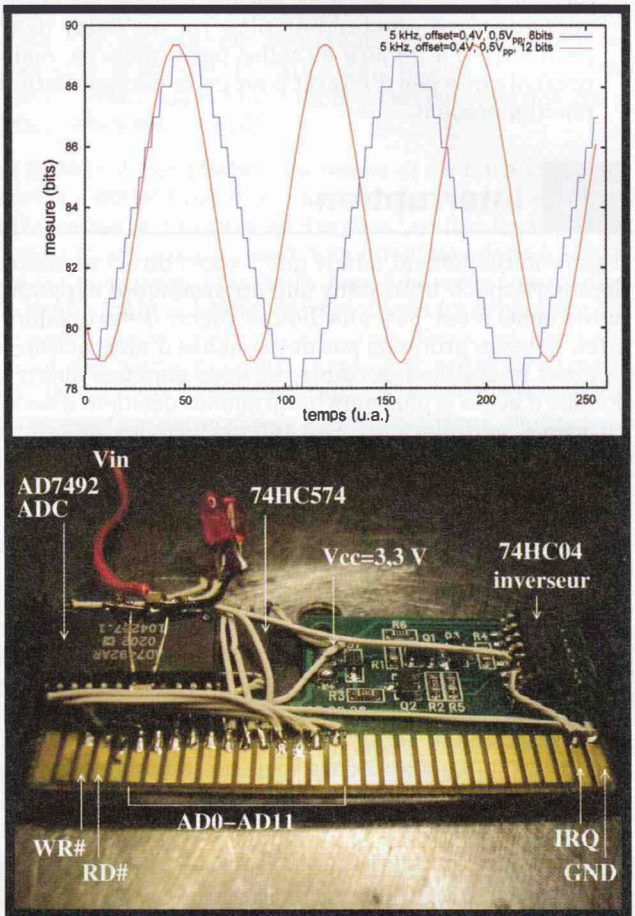


Figure 5 : En haut : comparaison entre la conversion analogique-numérique sur 8 bits (boucle vide de 10 itérations entre le début de conversion et lecture du résultat) et 12 bits (boucle vide de 5 itérations entre le début de conversion et la lecture du résultat, résultat divisé par 16 pour fournir la même échelle). Dans les deux cas, le signal en entrée est une sinusoïde à 5 kHz, amplitude 0,05 V et offset 0,4 V. Le gain en résolution se fait néanmoins au détriment d'une carte plus encombrée (en bas). Sur cette version de la cartouche, la broche BUSY du convertisseur est associée à l'interruption IRQ au travers d'un inverseur 74H04 monté au-dessus de la bascule D (4013) fournie d'origine dans le Rumble Pack.

La vitesse de conversion peut se caractériser de plusieurs façons : nous avons pour notre part exploité la transformée de Fourier du signal numérisé afin d'établir la fréquence d'échantillonnage f_e. En effet, la transformée de Fourier (fonction **`fft`** de GNU/Octave) s'étale de $-f_e/2$ à $f_e/2$ en N points (par défaut, N est le nombre de points acquis). Connaissant la fréquence f du signal observé comme un maximum de puissance au point n, la fréquence d'échantillonnage est $f_e = N/n \times f$. L'incertitude sur la fréquence de conversion Δf_e, en

supposant la fréquence mesurée parfaitement connue, est $\Delta f_e = f_e/n$: plus f est élevée, plus f_e est identifiée avec précision. Alternativement, le calcul dans le domaine temporel donne le même résultat : sur la conversion en 8 bits (bleue) de la fig. 5, nous constatons qu'une période du signal à 5 kHz est échantillonnée en 23 points, soit un échantillonnage à $f_e = 23 \times 5 = 115 \pm 5$ kHz (avec un délai de 10 itérations d'une boucle vide entre le début et la fin de la conversion). La vitesse maximum théorique du convertisseur AD7492 est de 1 méga-échantillon/seconde : un ajustement plus fin du délai devrait permettre d'atteindre de telles performances, mais ne présente que peu d'intérêt pour cette démonstration de fonctionnement.

5.3 Interruption

Il est habituellement admis que l'accès direct au matériel depuis l'espace utilisateur sur un système d'exploitation multitâche n'est pas une bonne façon de procéder. En effet, nous ne profitons pas des couches d'abstractions qui rendent le système portable, et nous sommes sujets aux conflits d'accès si plusieurs programmes décident d'accéder au même périphérique. Par ailleurs, sur les plateformes proposant une MMU, les accès aux ports et à la mémoire sont en général interdits. Nous allons donc proposer une implémentation « propre » de l'accès aux données du convertisseur analogique-numérique, notamment pour profiter de la possibilité d'être informé par interruption de la fin de conversion. Théoriquement, nous devrions par cette méthode atteindre la vitesse de conversion la plus élevée possible : nous lançons une conversion en activant CONVST# en écrivant dans la plage d'adresses de la cartouche. Nous vaquons à nos occupations jusqu'à être prévenu de la fin de conversion par interruption. À ce moment, nous lisons le résultat que nous plaçons dans un tampon et relançons immédiatement une nouvelle conversion. Nous allons voir que ce beau scénario idéal va rencontrer quelques difficultés...

Le module noyau donnant accès au périphérique de type caractère a été largement décrit dans ces pages et nous ne reviendrons pas sur les généralités. Nous partons de l'exemple de module avec interruption sur port parallèle de PC fourni à http://www.captain.at/howto-linux-device-driver-template-skeleton.php (qui contient quelques erreurs, notamment sur la syntaxe des arguments à **request_irq()**).

Dans notre cas, nous n'avons besoin que d'implémenter les méthodes **open**, **read** et **close** du module. Par ailleurs, nous exploiterons un **ioctl** pour déclencher la conversion et la mise en mémoire du résultat de la conversion afin de séparer la phase lecture des données (restitution des données acquises vers l'espace utilisateur dans la méthode **read**) et acquisition.

La structure de données contenant les méthodes implémentées se présente donc sous la forme

```
// define which file operations are supported
struct file_operations skeleton_fops = {
        .owner  =       THIS_MODULE,
        .llseek =       NULL,
        .read   =       skeleton_read,
        .write  =       NULL,
        .readdir=       NULL,
        .poll   =       NULL,
        .ioctl  =       skeleton_ioctl,
        .mmap   =       NULL,
        .open   =       skeleton_open,
        .flush  =       NULL,
        .release=       skeleton_release,
        .fsync  =       NULL,
        .fasync =       NULL,
        .lock   =       NULL,
        .readv  =       NULL,
        .writev =       NULL,
};
```

Une broche nommée IRQ est disponible sur le connecteur du slot 2 : la lecture de **linux/include/asm-asmnommu/arch-nds/irqs.h** nous indique qu'il s'agit de **IRQ_CART**, interruption numéro 13 (Fig. 6).

Les méthodes **open** et **release** ne présentent aucun intérêt : elles ne font qu'afficher un message validant l'opération. La méthode d'initialisation du module enregistre la communication entre espaces noyau et utilisateur au travers de **/dev/skeleton** de major number 32 et minor 0 :

```
int i = register_chrdev (32, "skeleton", &skeleton_fops);
```

ainsi que la gestion de l'interruption numéro 13 déclenchée par un front montant sur la broche IRQ du slot 2 de la DS :

```
ret=request_irq(13, interrupt_handler, SA_INTERRUPT, "SLOT2cart", NULL);
enable_irq(13); printk("interrupt enabled\n");
```

Nous penserons simplement dans la méthode **release** du module à abandonner l'interruption lorsque le module est déchargé :

```
unregister_chrdev (32, "skeleton"); // /dev/skeleton 32 0
disable_irq(13);
free_irq(13, interrupt_handler);
```

Le gestionnaire d'interruption se contente d'effectuer une opération la plus brève possible, ici incrémenter un compteur indiquant qu'une interruption a eu lieu, et réveiller une fonction qui serait endormie en attente de l'interruption.

```
static irqreturn_t interrupt_handler(int irq, void *dummy)
{
        interruptcount++;
#ifdef jmf_debug
       printk(">>> Slot2 INT HANDLED: interruptcount=%d\n", interruptcount);
#endif
        wake_up_interruptible(&skeleton_wait);
        return IRQ_HANDLED;
}
```

Cette interruption a réveillé le processus endormi et en attente du signal associé à **skeleton_wait** : il s'agit dans notre cas de l'ioctl chargé de gérer la conversion :

```
static int skeleton_ioctl(struct inode *inode, struct file *file,
                unsigned int cmd, unsigned long arg) {
[...]
 wait_queue_t wait;
 switch ( cmd ) {
   case 1:
    copy_from_user(&count, (int *)arg, sizeof(int));
    init_waitqueue_entry(&wait, current);
```

```
    interruptcount=0,oldcount=0;
    while (interruptcount<count)
          {*(unsigned short*)(0x8000000)=(unsigned short)0;
//         interruptible_sleep_on(&skeleton_wait);          <-- race !
           add_wait_queue(&skeleton_wait, &wait);
           while (1) {
             set_current_state(TASK_INTERRUPTIBLE);
             if (interruptcount != oldcount )
                break;
                schedule();
                }
             set_current_state(TASK_RUNNING);
             remove_wait_queue(&skeleton_wait, &wait);
             oldcount=interruptcount;

           string[interruptcount-1]=*(char*)(0x8000000)&0xff;
          }
          len  =interruptcount;
          interruptcount=0;
          break;
   default: retval = -EINVAL;
 }
 return retval;
}
```

Dans ce gestionnaire d'ioctl, nous passons en argument à **skeleton_ioctl**, en plus de l'action à effectuer, le nombre de conversions à effectuer (variable **count**). Tant que le nombre d'acquisitions n'a pas atteint **count**, nous amorçons la conversion en écrivant dans la plage d'adresses du slot 2, nous nous endormons le temps que l'interruption indique la fin de conversion, et nous lisons dans la plage d'adresses du slot 2 le résultat de la mesure. Ces résultats successifs sont accumulés dans le tableau **static char string [256]** qui servira de tampon lors de la fonction **read**.

Une subtilité est apparue dans l'implémentation de ce code avec la méthode classique de sommeil et réveil par interruption **interruptible_sleep_on()** : la conversion est tellement rapide que le processeur n'a pas le temps d'exécuter toutes les instructions de cette fonction et appels associés avant que l'interruption ne soit déclenchée. Le résultat est que le module reste en veille au lieu d'acquérir les données successives. Ce problème est bien documenté dans **[4]**, donc nous nous sommes contentés de prendre la solution qui consiste à explicitement séparer les étapes de mise en veille pour être capable de rapidement réagir à une interruption déclenchée trop tôt.

5.4 Exploitation du module noyau

Afin de compiler ce module noyau, il nous faut générer une image GNU/Linux comprenant :

- un noyau 2.6.x supportant le chargement dynamique des modules (option *Loadable Module Support / Enable loadable module support*)
- une version de Busybox comprenant les outils associés à **insmod**, et nettoyée des outils inutiles ou qui ne compilent pas afin d'alléger la taille de l'image (notamment **consoletools** dans Misc. Appls. et les jeux tels que xrick, Microwindows et pixil).

Une première compilation du noyau et de l'arborescence associée s'obtient par **make menuconfig** pour accéder à la configuration du noyau et de Busybox : retirez les éléments qui ne compilent pas ou ne fonctionnent pas dans 4 MB de RAM tel que décrit plus haut. Une fois cette compilation convenablement achevée, nous ajoutons un répertoire qui contiendra notre propre programme. Dans cet exemple, nous allons créer **~/dslinux/linux-2.6.x/drivers/char/test_jm**. Ce répertoire va contenir les sources nécessaires à la compilation du module – en fait un unique programme en C nommé **rumble_irq.c**. Afin que Linux sache, lors de la compilation, qu'il faut fabriquer ce module, nous ajoutons au **Makefile** disponible dans **~/dslinux/linux-2.6.x/drivers/char** la ligne **obj-m += test_jm/** (**obj-m** contient en effet dans chaque sous-répertoire la liste des modules à compiler), et, dans le répertoire **test_jm**, nous incluons un **Makefile** réduit à sa plus simple expression :

```
~/dslinux/linux-2.6.x/drivers/char/test_jm$ cat Makefile
obj-m += rumble_irq.o
```

Afin de ne pas se contenter des messages aseptisés de la compilation du noyau Linux 2.6, mais de retrouver le vrai sens des commandes de compilation, la compilation des modules se fait avec l'option **V=1** :

```
jmfriedt@ns39351:~/dslinux$ make modules V=1
[...]
make -f scripts/Makefile.build obj=drivers/char
make -f scripts/Makefile.build obj=drivers/char/test_jm
  arm-linux-elf-gcc -Wp,-MD,drivers/char/test_jm/.rumble_irq.o.d  -nostdinc -isystem \
/home/jmfriedt/dslinux-toolchain-2008-01-24-i686/bin/../lib/gcc/arm-linux-
elf/4.0.4/include \
-D__KERNEL__ -Iinclude  -mlittle-endian -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
-fno-strict-aliasing -fno-common -ffreestanding -O2      -fomit-frame-pointer
-mno-thumb-interwork \
-D__LINUX_ARM_ARCH__=5 -march=armv5te   -msoft-float -Uarm -mswp-byte-writes
-Wdeclaration-after-statement \
-Wno-pointer-sign   -DMO MODNAME=rumble_irq \
-c -o drivers/char/test_jm/rumble_irq.o drivers/char/test_jm/rumble_irq.c
[...]
```

Figure 6 : Capture d'écran présentant une séquence classique d'opérations : les programmes sont accessibles sur la carte MicroSD montée dans /media, le module noyau gérant l'interruption 13 – rumble_irq.ko – est lié au noyau : l'activation (transition du niveau bas de repos au niveau haut) d'une broche du 74HC574 dont la sortie est connectée au signal d'interruption déclenche un message du noyau validant la détection de l'évènement. La liste des interruptions gérées – dont le numéro 13 sous le nom SLOT2cart – est visible dans le pseudo-fichier /proc/interrupts.

```
  scripts/mod/modpost   -o /home/jmfriedt/dslinux/linux-2.6.x/Module.symvers vmlinux \
drivers/char/test_jm/rumble_irq.o
  arm-linux-elf-gcc -Wp,-MD,drivers/char/test_jm/.rumble_irq.mod.o.d
-nostdinc -isystem \
/home/jmfriedt/dslinux-toolchain-2008-01-24-i686/bin/../lib/gcc/arm-
linux-elf/4.0.4/include \
-D__KERNEL__ -Iinclude  -mlittle-endian -Wall -Wundef -Wstrict-
prototypes -Wno-trigraphs \
-fno-strict-aliasing -fno-common -ffreestanding -O2      -fomit-frame-
pointer  -mno-thumb-interwork \
-D__LINUX_ARM_ARCH__=5 -march=armv5te   -msoft-float -Uarm -mswp-byte-
writes -Wdeclaration-after-statement \
-Wno-pointer-sign     -DKBUILD_BASENAME=rumble_irq -DKBUILD_
MODNAME=rumble_irq -DMODULE \
-c -o drivers/char/test_jm/rumble_irq.mod.o drivers/char/test_jm/rumble_irq.mod.c
  arm-linux-elf-ld -EL  -r -o drivers/char/test_jm/rumble_irq.ko \
drivers/char/test_jm/rumble_irq.o drivers/char/test_jm/rumble_irq.mod.o
```

Nous constatons que la génération d'un module pour le noyau 2.6 n'est plus l'affaire d'une unique commande de compilation.

La dernière ligne s'est conclue par la génération de **rumble_irq.ko**, module prêt à être lié (**insmod**) au noyau pour y ajouter nos fonctionnalités (Fig. 6).

Le résultat est décevant en termes de performances : nous avons vu que, depuis l'espace utilisateur, avec un délai sous forme de boucle vide de 5 à 10 itérations, la conversion se fait correctement avec une fréquence d'échantillonnage de l'ordre de la centaine de kHz (535±10 kHz avec un délai de 5 itérations d'une boucle vide). Lorsque les données sont acquises par le module noyau, avec réveil par interruption associée à la fin de conversion, accumulation dans le tableau *string* pour finalement être restitué à l'utilisateur lors du *read*, nous n'atteignons que quelques kHz (44 kHz en exploitant le code de la table Tab. 3 associé au module noyau décrit ici). Ce résultat est néanmoins prévisible, puisque la mise en veille et toutes les méthodes associées sont considérablement plus gourmandes en ressources de calcul que quelques itérations d'une boucle vide. La solution « sale » de la conversion en espace utilisateur, aux performances très dépendantes de la charge du processeur et des opérations demandées au système d'exploitation, fournit néanmoins les meilleures performances lorsqu'il s'agit de la seule tâche effectuée par le processeur. Rappelons que cette seconde méthode de programmation n'est permise que sous uClinux et interdite sur les systèmes possédant un gestionnaire de mémoire, sur lesquels nous trouverons Linux.

```
// ~/dslinux/vendors/Nintendo/DLDI$
// ajouter skeleton,c,32,0 \ dans DEVICES

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define TAILLE 255

int main(int argc,char **argv)
{int f,taille=TAILLE;
 volatile int k;
 char *c;
printf("ADC go\n");
 c=(char*)malloc(TAILLE);;
 f=open("/dev/skeleton",O_RDWR);
 if (f!=-1) printf("sekeleton open OK");
 ioctl(f,1,&taille);     // lance la conversion
 read(f,c,TAILLE);     // lit le resultat
 close(f);  // conclusion : avec ioctl et interruption, on a 44 kHz
 for (f=0;f<TAILLE;f++) printf("%x ",c[f]);printf("\n");
 return(0);
}
```

Tableau 3 : Programme proposant l'accès aux périphériques du slot 2 au travers du point d'accès /dev/skeleton.

Nous avons donc démontré jusqu'ici que la Nintendo DS fournit une plateforme capable de supporter un environnement de type GNU/Linux, avec un shell et des périphériques accessibles selon les méthodes classiques de programmation des modules noyau. L'interface matérielle de la majorité des périphériques supportés par le noyau est cachée au développeur, tel que ce fut le cas pour le framebuffer ou la gestion de l'interruption. Nous avons par ailleurs vu comment ajouter nos propres interfaces via le bus accessible sur le slot 2. Cependant, le développeur se voit rapidement limité par la mémoire disponible, avec la majorité des 4 MB occupés par DSLinux. L'extension mémoire de 32 MB sur le slot 2 permet d'exécuter des programmes plus ambitieux – et notamment faire fonctionner la communication wifi avec les habituels **iwconfig** et **ifconfig** – mais au détriment de nos périphériques. Nous nous étions fixés comme objectif au cours de ces développements de transmettre les données acquises par le convertisseur analogique-numérique par wifi : DSLinux n'est simplement pas approprié pour cette tâche, car, trop gourmand, il nous faut trouver un autre environnement de travail mieux adapté aux 4 MB disponibles.

6 Un OS temps réel : RTEMS

Exploiter GNU/Linux, ou sa version pour systèmes sans MMU et à faible empreinte mémoire uClinux, est certes satisfaisant, mais surtout un exercice académique compte tenu de la quantité de mémoire restante une fois le système exécuté. Par ailleurs, nous avons vu que pour de l'instrumentation, d'une part un shell interactif n'a que peu d'intérêt, et, d'autre part, nous n'avons aucune prétention de garantir une latence bornée sous uClinux (l'intervalle de temps entre deux acquisitions dépend de la charge du processeur et de la bonne volonté du *scheduler* à laisser la main suffisamment longtemps à notre application pour obtenir tous les points requis).

Nous allons par conséquent proposer d'expérimenter avec un système d'exploitation temps-réel à faible empreinte mémoire qui semble plus approprié pour exploiter pleinement les fonctionnalités de la DS : RTEMS[7] .RTEMS (*Real Time Executive for M* Systems*, avec M* signifiant actuellement *Multiprocessor*) fournit un certain nombre d'outils tels que pile réseau, système de fichiers, gestion de plusieurs *threads* et éventuellement un *shell* qui en font un vrai système d'exploitation. Néanmoins, RTEMS ne gère pas la multiplicité de processus : son modèle de programmation est un processus unique multithreadé, sans protection de la mémoire. Ainsi, une image RTEMS se présente comme un

binaire monolithique contenant l'ensemble des instructions à exécuter, sans chargement dynamique d'exécutables ou de bibliothèques, d'où le nom d'Executive au lieu de OS : le lecteur prendra soin de corriger le titre de cette section de façon appropriée[8].

Un BSP (*board support package*) a récemment été inclus, suite au travail de M. Bucchianeri, B. Ratier, R. Voltz et C. Gestes, dans RTEMS pour la console DS : son utilisation est décrite à http://www.rtems.com/ftp/pub/rtems/current_contrib/nds-bsp/manual.html.

Afin de permettre la compilation de RTEMS pour DS, il nous faut obtenir une nouvelle toolchain : celle obtenue auparavant pour DSLinux ne fonctionne pas. Toutes les étapes sont parfaitement décrites dans http://www.rtems.com/onlinedocs//doc-current/share/rtems/pdf/started.pdf[9] pour le lecteur suffisamment patient pour en lire l'ensemble du contenu. En résumé, il suffit de récupérer les dernières versions de **gcc**, **binutils** et **newlib** sur ftp://ftp.rtems.com/pub/rtems/SOURCES/4.9/ et de les configurer pour la cible ARM par **--target=arm-rtems4.9**. Notez que compiler dans les archives pose un certain nombre de problèmes[10] : mieux vaut s'éviter ces ennuis et compiler les outils de la toolchain dans des répertoires autres que ceux de l'archive. Une fois la toolchain complète placée dans un répertoire accessible et inclus dans le **$PATH** (espace disque : environ 100 MB après compilation), il reste à compiler RTEMS.

Ici encore, nous prendrons soin de compiler dans un répertoire autre que celui contenant l'archive. Partant d'une arborescence propre de rtems-4.9.1, la configuration pour le BSP de la console d'obtient par **../rtems-4.9.1/configure -target=arm-rtems4.9 -enable-rtemsbsp=nds**. La compilation elle-même s'obtient par **make**, toujours en prenant soin d'avoir les outils de la toolchain (**arm-rtems4.9***) dans son **$PATH**.

Les résultats de la compilation exploitables se trouvent dans **jmf/arm-rtems4.9/c/nds/testsuites/samples** en supposant que la compilation de RTEMS-4.9.1 ait été effectuée dans le répertoire **jmf**.

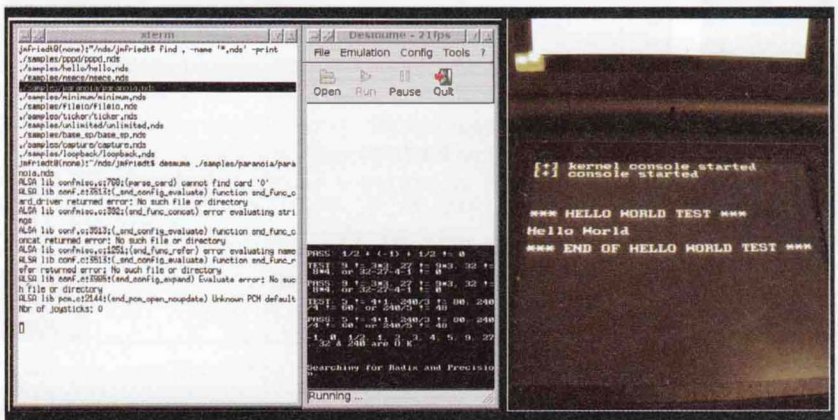


Figure 7 : Gauche : RTEMS exécuté dans l'émulateur desmume. Droite : un des exemples les plus simples de RTEMS, exécuté sur la console.

Afin de compiler ses propres applications ou, pour l'occasion, les exemples de la distribution « officielle » RTEMS 4.9.1[11], nous définissons les variables d'environnement

```
export PATH=/home/jmfriedt/rtems/bin/:$PATH
export RTEMS_ROOT=/opt/rtems-4.9
export RTEMS_MAKEFILE_PATH=/opt/rtems-4.9/arm-rtems4.9/nds
```

(évidemment adaptées à son propre environnement) avant de récupérer http://www.rtems.com/ftp/pub/rtems/4.9.1/examples-4.9.1.tar.bz2 dont le contenu se compile simplement par **make** (Figs 7 et 8, programme Tab. 4).

Les **Makefile** ainsi obtenus serviront d'exemple pour nos programmes.

La console DS fournit donc un support idéal pour se former à cet environnement de développement d'applications temps-réels disposant notamment une API compatible POSIX, qui ne dépayse donc pas fondamentalement le développeur sous GNU/Linux. Par ailleurs, un grand nombre de fonctionnalités restent à implémenter et le développeur amateur peut encore prétendre fournir une contribution significative dans ce cas (exploitation de la communication wifi, accès à la carte mémoire microSD).

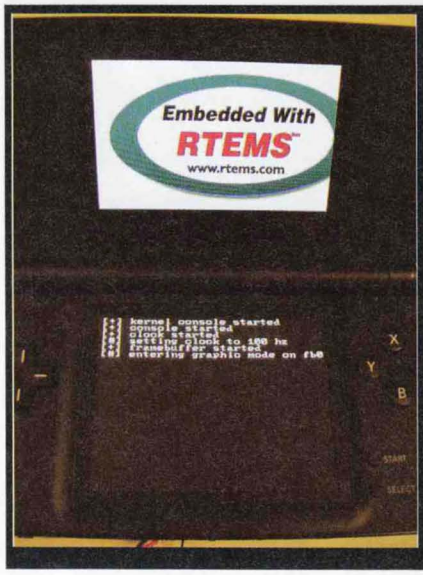


Figure 8 : Exploitation du framebuffer selon le même principe que proposé dans le cas de uClinux, mais cette fois sous RTEMS. Cet exemple est issu d'un programme de démonstration fourni par M. Bucchianeri. Étant incapables d'accéder depuis RTEMS au support de stockage de masse µSD, l'image est stockée dans le programme.

```
#include <bsp.h>
#include <rtems/fb.h>
#include <rtems/console.h>
#include <rtems/clockdrv.h>
#include "fb.h"

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems/mw_fb.h>

static struct fb_screeninfo fb_info;

inline void draw_pixel(int x, int y, int color)
{
  uint16_t* loc = fb_info.smem_start;
  loc += y * fb_info.xres + x;

  *loc = color;    // 5R, 5G, 5B
  *loc |= 1 << 15; // transparence ?
}

void draw_ppm()
{int x,y,bpp;char r,g,b;
 int l=0;
 for (y=0;y<161;y++)
   for (x=0;x<296;x++) {
     r=image[l++];g=image[l++];b=image[l++];
     bpp=(((int)(r&0xf8))>>3)+(((int)
(g&0xf8))<<2)+(((int)(b&0xf8))<<8);
     if (x<256) draw_pixel(x, y, bpp);
   }
}
```

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  struct fb_exec_function exec;
  int fd = open("/dev/fb0", O_RDWR);

  if (fd < 0)
    { printk("failed\n");
      exit(0);
    }

  exec.func_no = FB_FUNC_ENTER_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  ioctl(fd, FB_SCREENINFO, (void*)&fb_info);

  draw_ppm();

  while (1) ;

  exec.func_no = FB_FUNC_EXIT_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  close(fd);printk("done.\n");exit(0);
}

/* configuration information */

#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE

rtems_driver_address_table Device_drivers[] =
{
  CONSOLE_DRIVER_TABLE_ENTRY,
  CLOCK_DRIVER_TABLE_ENTRY,
  FB_DRIVER_TABLE_ENTRY,
  { NULL,NULL, NULL,NULL,NULL, NULL }
};

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      10
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

Tableau 4 : Programme d'affichage d'une image au format PPM sur le framebuffer, issu d'un programme de démonstration fourni par M. Bucchianeri.

Nous ne nous étendrons par sur les exemples de base fournis sur le site de RTEMS, tous fonctionnels. Mentionnons néanmoins quelques points qui peuvent surprendre à première vue :

- RTEMS fournit un environnement de développement s'apparentant à un système d'exploitation, mais le résultat est un unique binaire avec notamment l'absence de notion de chargement dynamique d'exécutables. L'exécution du programme se décompose en tâches appelées selon les évènements qui influent sur le comportement du scheduler :
- Dans ce contexte de système autonome à faible empreinte mémoire, le shell interactif avec l'utilisateur est une option gourmande en ressources qui n'est pas activée par défaut.
- Bien qu'écrite en C, une application RTEMS a une structure différente des programmes C habituellement écrits sous GNU/Linux. Une couche de compatibilité POSIX rend néanmoins le passage à RTEMS rapide pour le développeur sous Unix. Comme souvent, un certain nombre de **#include** en en-tête de fichier annoncent quels fichiers de configuration charger, mais surtout un grand nombre de **#define** définissent quelles fonctionnalités du BSP activer. Chaque fonctionnalité a un coût en termes de ressources et on prendra soin de se limiter au strict minimum. Réciproquement, l'oubli d'une ressource peut induire un dysfonctionnement de l'application qui n'est pas toujours facile à déboguer : par exemple, l'appel à la fonction **sleep** sans activer de *timer* induit nécessairement une attente infinie telle que décrite à http://www.rtems.com/wiki/index.php/DebuggingHints. L'ordre de ces **#define** est important, car des variables définies dans un premier fichier peuvent être exploitées dans une définition ultérieure.

Nous allons dépasser le cadre de la simple application logicielle pour nous tourner, comme dans l'exemple d'utilisation de DSLinux, vers l'exploitation de périphériques matériels. Afin de faire clignoter (Code 5) les diodes connectées au *latch* installé à la place du moteur dans le Rumble Pack (bus du slot 2), il faut affecter ce bus au processeur ARM9 (et non à son coprocesseur ARM7) par la commande **sysSetCartOwner(BUS_OWNER_ARM9);** : sans cette fonction, le processeur qui exécute RTEMS n'a pas le contrôle du bus du slot 2 et les diodes ne changent pas d'état. La macro de cette commande, définie dans **rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c**, a été développée pour faciliter la compilation : il s'agit de la ligne modifiant la valeur de l'emplacement mémoire **0x04000204**. Notez que cette affectation du bus à l'ARM9 était déjà effectuée sous DSLinux.

```
// compiler avec -I/home/jmfriedt/rtems/jmf/arm-rtems4.9/nds/lib/include/libnds/

#include <bsp.h>
#include <stdlib.h>
#include <stdio.h>
#include <nds/memory.h>

rtems_id timer_id;
uint16_t l=0;

void callback()
{ printk("Callback %x\n",l);
  (*(volatile uint16_t*)0x08000000)=l;
  l=0xffff-l;
  rtems_timer_fire_after(timer_id, 100, callback, NULL);
}

rtems_task Init(rtems_task_argument ignored)
{ rtems_status_code status;
  rtems_name timer_name = rtems_build_name('C','P','U','T');

  printk( "\n\n*** HELLO WORLD TEST ***\n" );
// cf rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c
// sysSetCartOwner(BUS_OWNER_ARM9);
// defini dans rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/include/nds/memory.h
  (*(vuint16*)0x04000204) = ((*(vuint16*)0x04000204) & ~ARM7_OWNS_ROM);

  status = rtems_timer_create(timer_name,&timer_id);
  rtems_timer_fire_after(timer_id, 1, callback, NULL);
  rtems_stack_checker_report_usage(); // requires #define CONFIGURE_INIT

  printk( "*** END OF HELLO WORLD TEST ***\n" );
  while(1)
    ;
  exit( 0 );
}
```

```
/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

/* configuration information */
#define CONFIGURE_MAXIMUM_DEVICES                    40
#define CONFIGURE_MAXIMUM_TASKS                      100
#define CONFIGURE_MAXIMUM_TIMERS                     32
#define CONFIGURE_MAXIMUM_SEMAPHORES                 100
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES             20
#define CONFIGURE_MAXIMUM_PARTITIONS                 100
#define CONFIGURE_MAXIMUM_REGIONS                    100

/* This seetings overwrite the ones defined in confdefs.h */
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES              32
#define CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES  32
#define CONFIGURE_MAXIMUM_POSIX_KEYS                 32
#define CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS       10
#define CONFIGURE_MAXIMUM_POSIX_THREADS              128
#define CONFIGURE_MAXIMUM_POSIX_TIMERS               10
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS     200

#define STACK_CHECKER_ON
#define CONFIGURE_INIT

#include <rtems/confdefs.h>

/* end of file */
```

Tableau 5 : Programme pour faire clignoter les diodes connectées au 74HC574 inséré dans le Rumble Pack à la place du moteur.

Étant capable de valider l'accès au bus du slot 2, il est trivial de convertir les divers exemples proposés sur DSLinux – et notamment la conversion analogique numérique et l'affichage sur framebuffer – sur RTEMS et obtenir les mêmes résultats. Nous voulons cependant aller plus loin et atteindre notre objectif de communication des données acquises par wifi : dans un premier temps, nous allons voir comment activer un shell interactif sous RTEMS (afin de se rapprocher de l'environnement de travail habituel sous GNU/Linux et illustrer quelques-unes des instructions originales proposées par RTEMS pour déboguer des applications embarquées), avant de nous lancer dans l'aventure du fonctionnement du wifi.

6.1 Un shell interactif

Bien que son utilité dans un système fortement embarqué à faible empreinte mémoire soit discutable – l'interactivité avec le contrôleur d'une machine à laver est rarement utile à l'utilisateur en dehors des phases de débogage – nous nous sommes proposés d'exploiter le shell de RTEMS et d'y ajouter nos propres fonctionnalités. Le shell de RTEMS est avant tout orienté, dans sa configuration par défaut, vers le diagnostic de l'état du système en cours d'exécution : occupation du processeur par chacune des tâches (**cpuuse**), mémoire occupée et taille de la pile allouée à chaque tâche (**stackuse**, Fig. 9) et contenu de cette mémoire (**mdump**). Quelques fonctions d'accès à un éventuel système de fichiers et interface de communication sont disponibles en option.

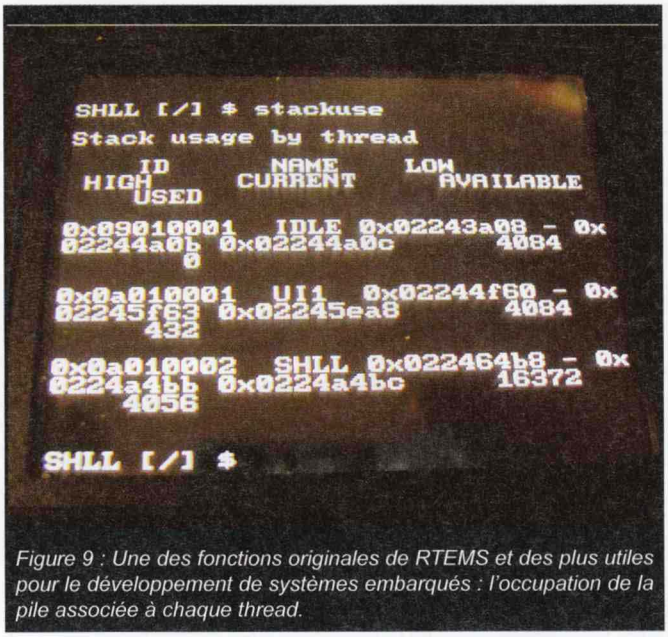


Figure 9 : Une des fonctions originales de RTEMS et des plus utiles pour le développement de systèmes embarqués : l'occupation de la pile associée à chaque thread.

L'interaction avec un shell nécessite évidemment d'entrer des caractères formant les commandes des actions à communiquer au système. Le mode de communication proposé par le BSP NDS de RTEMS est la reconnaissance de l'écriture manuelle des lettres sur l'écran tactile suivant un algorithme implémenté dans la PALib : la façon d'écrire les lettres anglo-saxonnes doit être suivie scrupuleusement pour espérer entrer le bon caractère, selon la méthode décrite à http://www.palib.info/wiki/doku.php?id=day3#keyboard et reproduite sur la fig. 10. Notez que l'activation de l'interface Graffiti nécessite d'appuyer sur la touche [L] de la console tout en écrivant les lettres sur l'écran tactile.

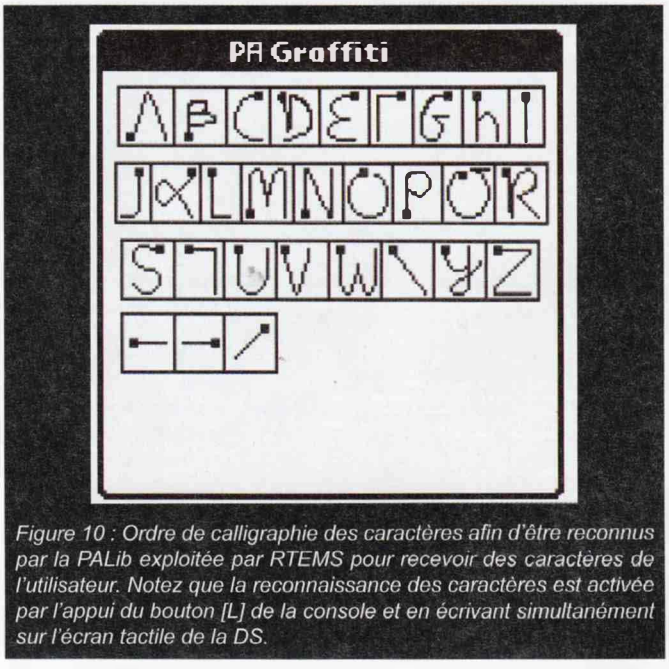


Figure 10 : Ordre de calligraphie des caractères afin d'être reconnus par la PALib exploitée par RTEMS pour recevoir des caractères de l'utilisateur. Notez que la reconnaissance des caractères est activée par l'appui du bouton [L] de la console et en écrivant simultanément sur l'écran tactile de la DS.

Afin de valider notre capacité à ajouter une fonction qui utilise un maximum de ressources, nous avons implémenté la fonction **newt** qui affiche la fractale de Newton **[2]** pour le polynôme z^3-1, $z \in \mathbb{C}$: cette application démontre la fonctionnalité de l'émulation logicielle du calcul flottant,

l'accès à une interface graphique sous forme de framebuffer, et le passage de paramètres.

Le prototype de la fonction à ajouter au shell s'apparente au `main(int, char**)` classique du C. Ainsi, une commande `newtn` qui exécute le contenu de la fonction `mon_main` qui reçoit ses arguments de la même façon que le main du C s'obtient par :

```
rtems_shell_cmd_t Shell_USERCMD_Command = {
  "newtn",                  /* name */
  "newt [mag.]",            /* usage */
  "user",                   /* topic */
  mon_main,                 /* command */
  NULL,                     /* alias */
  NULL                      /* next */
};
```

Comme, par ailleurs, Graffiti est relativement pénible à utiliser et que les chances de taper une commande de 5 lettres sans erreur sont quasiment nulles, nous créons un alias sur la commande par :

```
rtems_shell_alias_t Shell_USERECHO_Alias={
"newtn", /* command*/
"n"      /* alias */
};
```

Ces commandes sont ajoutées au shell[12] par :

```
#define CONFIGURE_SHELL_USER_COMMANDS &Shell_USERCMD_Command
#define CONFIGURE_SHELL_USER_ALIASES  &Shell_USERECHO_Alias
#define CONFIGURE_SHELL_COMMANDS_INIT
#define CONFIGURE_SHELL_COMMANDS_ALL
#include <rtems/shellconfig.h>  // doit etre APRES les #define
```

Dans cet exemple, la commande `newtn` (ou `n`) dessine la fractale de Newton avec un facteur de grossissement défini par l'argument (Fig. 11).

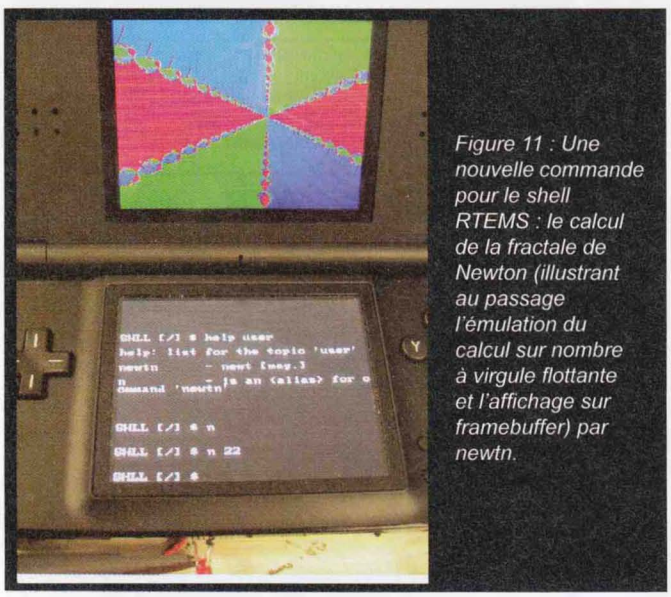


Figure 11 : Une nouvelle commande pour le shell RTEMS : le calcul de la fractale de Newton (illustrant au passage l'émulation du calcul sur nombre à virgule flottante et l'affichage sur framebuffer) par newtn.

6.2 Estimation des latences

RTEMS s'annonce comme un environnement temps réel. Nous avons donc tenté de caractériser les latences associées aux systèmes DSLinux et RTEMS lorsque trois threads sont actifs, deux threads pour faire clignoter des diodes avec des intervalles de temps définis par une attente (`usleep()`) et un troisième thread, activé manuellement par l'utilisateur, chargé d'effectuer un calcul long (le calcul et l'affichage simultané de la fractale de Newton sur le polynôme z^3-1) (Fig. 12). Ces tests ont pour objectif :

- de valider le fonctionnement de la bibliothèque `pthreads` classiquement utilisée sous GNU/Linux, et en particulier son exploitation sous uClinux dans le cadre de DSLinux ;
- de comparer les méthodes d'implémentation des threads entre DSLinux[13] et RTEMS (Codes 6 et 7) ;
- de comparer les latences **[5]** en mesurant à l'oscilloscope la période de commutation des diodes – période définie dans deux des trois threads – et la robustesse de l'application selon que le troisième thread chargé d'un calcul lourd (calcul et tracé de la fractale de Newton calculée pour tous les points de l'écran, soit une dizaine de secondes de calculs) soit actif ou non.

Les résultats de ce test grossier montrent que les latences sous RTEMS (Fig. 12, gauche) sont plus stables lors de la charge que sous DSLinux, mais la différence est moins impressionnante qu'on pourrait l'attendre. Un aspect visible à l'oscilloscope, mais qui ne peut pas être retranscrit sur ces images statiques est que la variation de période sous DSLinux intervient au moment de lancer le calcul de la fractale, et non pendant le calcul : la variation de charge induit plus de latence que la charge de calcul elle-même. Nous n'avons pas cherché à identifier ce phénomène, qui serait probablement visible en analysant les sources du scheduler de chaque système d'exploitation. Notez que tous les threads sous DSLinux ont la même priorité, tandis que, sous RTEMS, nous affectons la plus grande priorité au thread contenant le shell.

```
// $CC $CFLAGS $LDFLAGS -L/home/jmfriedt/dslinux/uClibc/lib/  fb_
thread.c -o fb_thread -lpthread
// fbset -depth 16 -n
// colormap = 5R, 5G, 5B
// bien penser a sync quand on sauve un fichier

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <linux/fb.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <pthread.h>

inline void draw_pixel(int, int, int);
void mon_main();

char * device = "/dev/fb0";
```

```
struct fb_var_screeninfo screeninfo;
int screen_fd;
unsigned short * screen_ptr;
int screen_width;
int screen_height;

///// threads /////
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  struct timespec timeout;
  pthread_attr_t attr;

void * print_hello(void * arg){
  unsigned short l=0x01;

  printf("<child>: Hello World! task with max priority \n");
  while (1) {
    usleep(100000);
    (*(volatile unsigned short*)0x08000000)=l;
    l=0xffff-l;
  }
  return NULL;
}

void * print_hello_a(void * arg){
  int c;
  printf(" <child>: Hello World! Task with lowest priority ");
  printf("Press Enter to continue\n");
  while (1) {c=getchar();printf("%c\n",c);
             mon_main();printf("done\n");
            }

  return NULL;
}
///// fin threads /////

[...] CALCUL DE LA FRACTALE DE NEWTON
float module(float r,float i);

void mon_main()
{float ri,ii,r,i,df;
 int x=0,y=0,iter=0,maxiter=0,itertot=0;;

 df=1.;

 for (x=0;x<pointsx;x++)
  for (y=0;y<pointsy;y++)
    draw_pixel(x,y,0);
 x=0;y=0;

 for (ii=imin/df;ii<imax/df;ii+=(imax-imin)/pointsy/df)    // terminal : 105x54
   {for (ri=rmin/df;ri<rmax/df;ri+=(rmax-rmin)/pointsx/df)
     {iter=0;
      r=ri; i=ii;
     do {newton(&r,&i);iter++        } while ((module(r,i)<0.99) || (module(r,i)>1.01));
        if (iter>maxiter) maxiter=iter;
      if (r>0) draw_pixel(x,y,16); else
         if (i>0) draw_pixel(x,y,512); else draw_pixel(x,y,16384);
//    pset(x,y,iter%10);   // affiche nbre d'iterations
       x++;
       itertot+=iter;
     }
     y++;x=0;
   }
}
```

```
////////////////////////////////// fin / newton ////////////////////////
inline void draw_pixel(int x, int y, int color)
{
        unsigned short *loc = screen_ptr + \
           ((y + screeninfo.yoffset) * screen_width ) + x + screeninfo.xoffset;

        if ((x<0) || (x>=screen_width) || (y<0) || (y>=screen_height))
                return;
        *loc = color;     // 5R, 5G, 5B
        *loc |= 1 << 15; // transparence ?
}

int main(int argc, char *argv[])
{
        open(device, O_RDWR);
        ioctl(screen_fd, FBIOGET_VSCREENINFO, &screeninfo);
        screen_width = screeninfo.xres;
        screen_height = screeninfo.yres;
        screen_ptr=mmap(0,screen_height*screen_width/8,PROT_READ|PROT_
WRITE,MAP_SHARED,screen_fd,0);

        if (screen_ptr==MAP_FAILED)
           perror("Unable to mmap frame buffer");

  pthread_t child1;
  pthread_t child2;
  pthread_attr_init(&attr);
  pthread_mutex_init( &mutex, NULL );
  pthread_cond_init( &cond, NULL );

  printf("Creating first task \n");
  pthread_create( &child1, &attr, print_hello, NULL);
  printf("Creating second task \n");
  pthread_create( &child2, &attr, print_hello_a, NULL);

  printf("<main> Out of the main\n");
  pthread_join( child1, NULL );
  pthread_join( child2, NULL );
  printf("The END\n");fflush(stdout);
  close(screen_fd);exit(0);
}
```

Tableau 6 : Exemple de pthread sous DSLinux.

```
#include <bsp.h>
#include <rtems/fb.h>
#include <rtems/console.h>
#include <rtems/clockdrv.h>
#include "fb.h"
#include <sched.h>
#include <pthread.h>

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems/mw_fb.h>

static struct fb_screeninfo fb_info;
inline void draw_pixel(int, int, int);
int mon_main();

///// threads /////
  pthread_mutex_t mutex;
  pthread_cond_t cond;
```

```
  struct sched_param param;
  pthread_attr_t attr;

void * print_hello(void * arg){
  unsigned short l=0x01;
  printf("<child>: Hello World! task with max priority \n");
  while (1) {
    (*(volatile uint16_t*)0x08000000)=l;
    l=0xffff-l;
    usleep(50000);  // 50 ms = 20 Hz
  }
  return NULL;
}

void * print_hello_a(void * arg){
  int c;
  printf(" <child>: Hello World! Task with lowest priority ");
  printf("Press Enter to continue\n");
  while (1) {c=getchar();printf("%c\n",c);
             mon_main();printf("done\n");
            }

  return NULL;
}

////////////////// newton ////////////////
[...] COOE POUR LE CALCUL DE LA FRACTALE DE NEWTON

inline void draw_pixel(int x, int y, int color)
{ uint16_t* loc = fb_info.smem_start;
  loc += y * fb_info.xres + x;
  *loc = color;    // 5R, 5G, 5B
  *loc |= 1 << 15; // transparence ?
}

// void *POSIX_Init()
rtems_task Init(rtems_task_argument ignored)
{int c;
 struct fb_exec_function exec;
 int fd = open("/dev/fb0", O_RDWR);
 if (fd < 0) {printk("failed\n");exit(0);}

 exec.func_no = FB_FUNC_ENTER_GRAPHICS;
 ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
 ioctl(fd, FB_SCREENINFO, (void*)&fb_info);

// rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c
// sysSetCartOwner(BUS_OWNER_ARM9);
// defini dans rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/include/nds/memory.h
 (*(volatile unsigned short*)0x04000204) = ((*(volatile unsigned
short*)0x04000204) & ~ (1 << (7)));

// void *POSIX_Init() {
  pthread_t child1;
  pthread_t child2;
  pthread_attr_init(&attr);
  pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
  pthread_mutex_init( &mutex, NULL );
  pthread_cond_init( &cond, NULL );

  printf("<main> Enter in the main \n");

  printf("Creating first task \n");
  param.sched_priority = sched_get_priority_max(SCHED_FIFO);
  pthread_attr_setschedparam(&attr, &param);
  if ( pthread_create( &child1, &attr, print_hello, NULL) ||  \
    pthread_setschedparam(child1, SCHED_FIFO, &param) ) {
      printf("Thread cannot be created \n");exit(1);
  }
  printf("Creating second task \n");
  param.sched_priority = sched_get_priority_max(SCHED_FIFO) - 1;
  pthread_attr_setschedparam(&attr, &param);
  if ( pthread_create( &child2, &attr, print_hello_a, NULL) || \
pthread_setschedparam(child2, SCHED_FIFO, &param) ) {
       printf("Thread cannot be created \n");exit(1);
  }
  printf("<main> Out of the main\n");
  pthread_join( child1, NULL );
  pthread_join( child2, NULL );

  exec.func_no = FB_FUNC_EXIT_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  close(fd);printk("done.\n");exit(0);
}

/* configuration information */

#define CONFIGURE_HAS_OWN_OEVICE_DRIVER_TABLE

rtems_driver_address_table Device_drivers[] =
{
  CONSOLE_DRIVER_TABLE_ENTRY,
  CLOCK_DRIVER_TABLE_ENTRY,
  FB_DRIVER_TABLE_ENTRY,
  { NULL,NULL, NULL,NULL,NULL, NULL }
};

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      10

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_APPLICATION_NEEOS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEOS_CLOCK_ORIVER

#define CONFIGURE_MAXIMUM_POSIX_THREADS              10
#define CONFIGURE_MAXIMUM_POSIX_CONOITION_VARIABLES  10
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES              10

#define CONFIGURE_MAXIMUM_OEVICES                     40
#define CONFIGURE_MAXIMUM_TASKS                       100
#define CONFIGURE_MAXIMUM_TIMERS                      32
#define CONFIGURE_MAXIMUM_SEMAPHORES                  100
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES              20
#define CONFIGURE_MAXIMUM_PARTITIONS                  100
#define CONFIGURE_MAXIMUM_REGIONS                     100

/* This seetings overwrite the ones defined in confdefs.h */
#define CONFIGURE_MAXIMUM_POSIX_KEYS                  32
#define CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS        10
#define CONFIGURE_MAXIMUM_POSIX_TIMERS                10

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

Tableau 7 : Exemple de pthread sous RTEMS. L'initialisation du framebuffer (code Tab. 4) et l'affichage de la fractale censée ralentir les divers threads sont omis pour fournir un code plus concis.

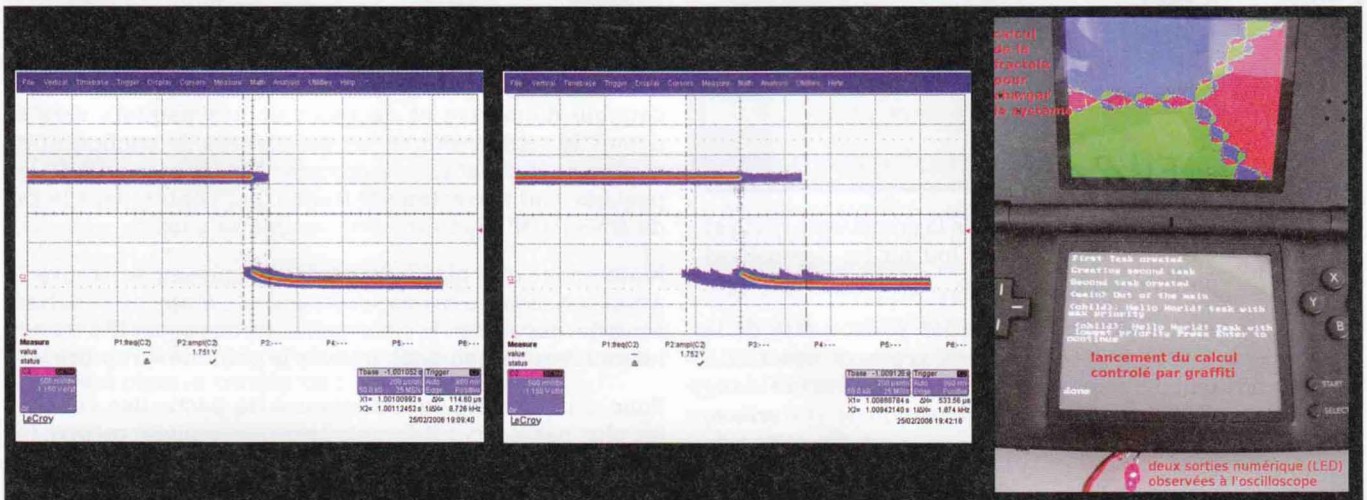


Figure 12 : Un oscilloscope numérique déclenche sa mesure sur le front montant d'un créneau généré de façon logicielle en programmant le latch connecté à la carte du slot 2. Nous agrandissons la mesure autour du front descendant pour estimer la variance de la longueur du créneau en fonction de la charge du système : à gauche, le résultat de la mesure pour RTEMS dans le cas défavorable d'un shell de priorité maximale et de threads pour le clignotement de priorité faible, au milieu pour DSLinux. À droite : montage expérimental sous RTEMS, avec le déclenchement du calcul (induisant les latences lors de l'ajout de cette charge de calcul additionnelle) par un retour chariot sous Graffiti (bouton [L] et un trait sur l'écran tactile de en haut à droite vers en bas à gauche).

6.3 Communication wifi

RTEMS ayant une empreinte mémoire plus faible de DSLinux, celui-ci permet l'utilisation du wifi de la DS sans devoir ajouter l'expansion pack.

Le wifi, tel que fourni par Nintendo, offre une fonctionnalité inhabituelle. En effet, la NDS est capable de stocker les paramètres de configuration pour la connexion à trois points d'accès[14], afin que les mêmes informations soient disponibles pour n'importe quel jeu requérant le wifi. Bien que DSLinux prouve qu'il est possible d'utiliser le *chipset* d'une manière classique (à l'aide des applications **iwconfig** ou **wpa_supplicant**), tout comme le logiciel de démonstration **wifi_lib_test**[15], RTEMS exploite les données stockées dans la console comme le ferait un jeu. Il n'est d'ailleurs pas possible, à l'heure actuelle, de configurer la connexion à un Access Point depuis RTEMS, imposant, de ce fait, l'utilisation d'un jeu commercial supportant le wifi. Le pilote fourni par RTEMS n'étant pas complètement fonctionnel, nous nous sommes efforcés de corriger quelques incompatibilités entre la couche réseau de RTEMS et le pilote, afin de rendre la communication wifi sur NDS opérationnelle. Le patch proposé est disponible à http://jmfriedt.free.fr/rtems_nds_wireless.patch pour les versions 4.9.1 et 4.9.2 de RTEMS.

La présentation du wifi avec RTEMS se fera en trois temps :

1. Dans un premier temps, pour valider le bon fonctionnement de la communication, nous allons utiliser **gethostbyname()**.

2. Puis, nous mettrons en œuvre un serveur **telnetd** disponible dans les bibliothèques de RTEMS afin d'obtenir un shell sans devoir passer par Graffiti.

3. Et, pour finir, une application qui fera parvenir à un ordinateur des valeurs issues d'une conversion analogique-numérique pour une application de télémesure.

6.4 gethostbyname

Comme la couche réseau de RTEMS est basée sur celle des BSD et compatible POSIX, cet exemple pourrait fonctionner directement sur GNU/Linux ou un Unix.

Son intérêt est, d'une part, de mettre en avant les configurations nécessaires pour faire fonctionner le réseau et, d'autre part, d'illustrer le bon fonctionnement du BSP NDS (RTEMS 4.9.1, 4.9.2) après application de notre patch. Cet exemple simple permettra de présenter une solution de débogage et de développer comment la solution a été identifiée puis mise en œuvre.

L'utilisation du réseau à partir de RTEMS consiste en la configuration de deux structures :

```
static struct rtems_bsdnet_ifconfig netdriver_config = {
 RTEMS_BSP_NETWORK_DRIVER_NAME,
 RTEMS_BSP_NETWORK_DRIVER_ATTACH,
 NULL, /* No more interfaces */
 "10.0.1.20", /* IP address */
 "255.255.255.0", /* IP net mask */
 NULL, /* Driver supplies hardware address */
};
```

Cette première structure, équivalente à **ifconfig**, configure l'interface réseau. **RTEMS_BSP_NETWORK_DRIVER_NAME** donne le nom de l'interface et **RTEMS_BSP_NETWORK_DRIVER_ATTACH** fournit la méthode d'initialisation.

```
struct rtems_bsdnet_config rtems_bsdnet_config = {
 &netdriver_config,
 NULL, /* do not use bootp */
 0, /* Default network task priority */
 0, /* Default mbuf capacity */
 0, /* Default mbuf cluster capacity */
```

```
"rtems", /* Host name */
"trabucayre.com", /* Domain name */
"10.0.1.1", /* Gateway */
"10.0.1.13", /* Log host */
{"10.0.1.13" }, /* Name server(s) */
{"10.0.1.13" }, /* NTP server(s) */
};
```

Cette seconde structure, à l'instar de la commande route et du fichier **resolv.conf**, permet de fournir les paramètres globaux de configuration du réseau.

Ensuite, au lancement de l'application (c'est-à-dire de la fonction **rtems_task Init(rtems_task_argument ignored)**), l'application doit faire appel à **rtems_bsdnet_initialize_network();** qui se charge de l'initialisation de l'interface, de la connexion au point d'accès et de la configuration de l'ensemble des paramètres du réseau.

Ce premier exemple, une fois compilé et installé sur la µSD, affichera l'adresse IP d'une machine dont le nom a été fourni à **gethostbyname()**.

En l'absence de l'interlocuteur ou avant application du patch (c'est-à-dire avec un RTEMS obtenu sur l'archive officielle), après un peu d'attente, un message d'erreur sera affiché en lieu et place de l'IP. Pour comprendre la cause de cette erreur, nous allons voir une solution pour déboguer le réseau, celle-ci se déroulant en deux étapes.

La première partie concerne l'analyse du réseau afin de pouvoir cibler le problème **[6]**. À l'instar du débogage d'un logiciel, ce type de travail nécessite l'utilisation d'un outil adéquat, tel que le sniffer réseau Ethereal.

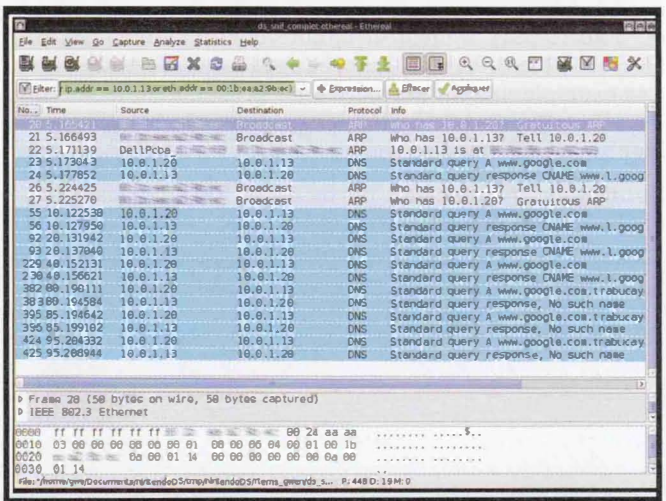


Figure 13 : Capture réseau à l'aide d'Ethereal

En examinant les trames affichées dans Ethereal (Fig. 13), nous pouvons suivre l'ensemble du dialogue réseau avec la NDS. La première partie est une requête ARP (trames 21,22) où la DS demande l'adresse MAC du serveur DNS, la machine concernée lui retourne l'information. Ce premier dialogue permet de déduire que les requêtes ARP sont correctement gérées, car (trame 23) la DS fait une requête DNS auprès du serveur. Ce dernier lui répond (trame 24), mais la DS émet à nouveau le même message comme le montrent les trames qui suivent : les paquets sont donc correctement émis et reçus, mais un point de la chaîne de traitement ne fonctionne pas.

Nous avons fait globalement le tour des points importants de l'analyse et nous pouvons déterminer que la DS est capable d'émettre et de recevoir des paquets vers et depuis le réseau, et que les paramètres de configuration sont correctement pris en compte. Dans le cas de l'ARP, les paquets sont correctement traités, par contre, dans le cas du DNS (UDP), l'information semble se perdre.

Maintenant que nous savons à quel niveau se trouve le problème, nous arrivons sur la seconde étape de l'analyse. Nous devons étudier le cheminement d'un paquet IP à travers les couches réseau pour trouver le point de « rupture ».

Pour cela, il n'est pas nécessaire de partir des couches les plus basses, car il semble bien que le pilote reçoive les paquets correctement. Il faut donc trouver où est faite la distinction entre paquet ARP et paquet IP.

En partant donc des sources du BSP NDS au niveau de la fonction qui transfère le paquet à la bibliothèque RTEMS (**wifi/wifi.c** : **wifi_rxd**), et en suivant les appels de fonctions, la distinction entre le type du paquet se fait dans **net/if_ethersubr.c** : **ether_input**. En affichant dans cette fonction des messages avec **printk** pour des paquets de type **ETHERTYPE_IP**, nous pouvons voir que celui-ci existe toujours. Cette fonction fait indirectement appel pour la suite du transfert à **netinet/ip_input.c** : **ip_input**. En suivant l'exécution de cette fonction nous arrivons sur un **return**, appelé lorsque le paquet reçu n'est pas destiné à la console. Pourtant, le paquet lui est bien destiné…

Pour comprendre la raison de ce comportement, il suffit d'afficher l'adresse du destinataire contenue dans la **struct ip**. L'IP de l'émetteur se trouve à la place de celle du destinataire, ceci expliquant le rejet, car le système s'attend à y trouver sa propre IP.

Il est donc possible de conclure que la conversion faite entre la **struct mbuf** et la **struct ip** n'aligne pas correctement les informations et qu'a priori les données arrivant du réseau dans le cas d'un paquet IP ne sont pas correctement stockées.

Notre solution simple, mais fonctionnelle, est l'utilisation de **m_pullup(...)** qui permet d'assurer que le contenu de la structure est bien contigu. Il ne reste donc plus qu'à déterminer dans le pilote du BSP ce qui entraîne ce problème, corrigé, actuellement, par réagencement des informations de la structure de données.

6.5 telnetd

Une fois la liaison wifi fonctionnelle, nous pouvons aborder des applications plus ambitieuses. RTEMS offre une implémentation d'un serveur telnet, permettant de pouvoir se connecter par le réseau sur la DS depuis n'importe quel terminal, d'une manière simple.

Cette implémentation n'implique pas forcément l'ouverture d'un shell. Dans le code présenté ci-dessous, le *login* n'est dû qu'à l'appel à **rtems_shell_main_loop()** qui lance un nouveau shell, qui sera disponible ensuite pour l'utilisateur distant.

En plus de permettre de valider la bonne intégration dans RTEMS, l'exemple suivant offre une solution pratique pour s'éviter l'utilisation (laborieuse) de Graffiti (Fig. 14). Son utilisation est triviale, car reposant uniquement sur l'appel à une fonction, après l'initialisation du réseau, de la manière suivante :

```
  rtemsShell, /* "shell" function */
  NULL, /* no context necessary for echoShell */
  false, /* listen on sockets */
  RTEMS_MINIMUM_STACK_SIZE * 20, /* shell needs a large stack */
  1, /* priority */
  false /* telnetd does NOT ask for password */
);
```

Le premier paramètre est le nom de la fonction qui sera appelée à chaque connexion :

```
  printk("======== Starting Shell ========\n");
  rtems_shell_main_loop( NULL );
  printk("======== Exiting Shell ========\n");
}
```

Son utilisation nécessite également l'inclusion du fichier `rtems/telnetd.h` dans les sources, ainsi que l'ajout de la ligne `LD_LIBS += -ltelnetd` dans le `Makefile` de l'application.

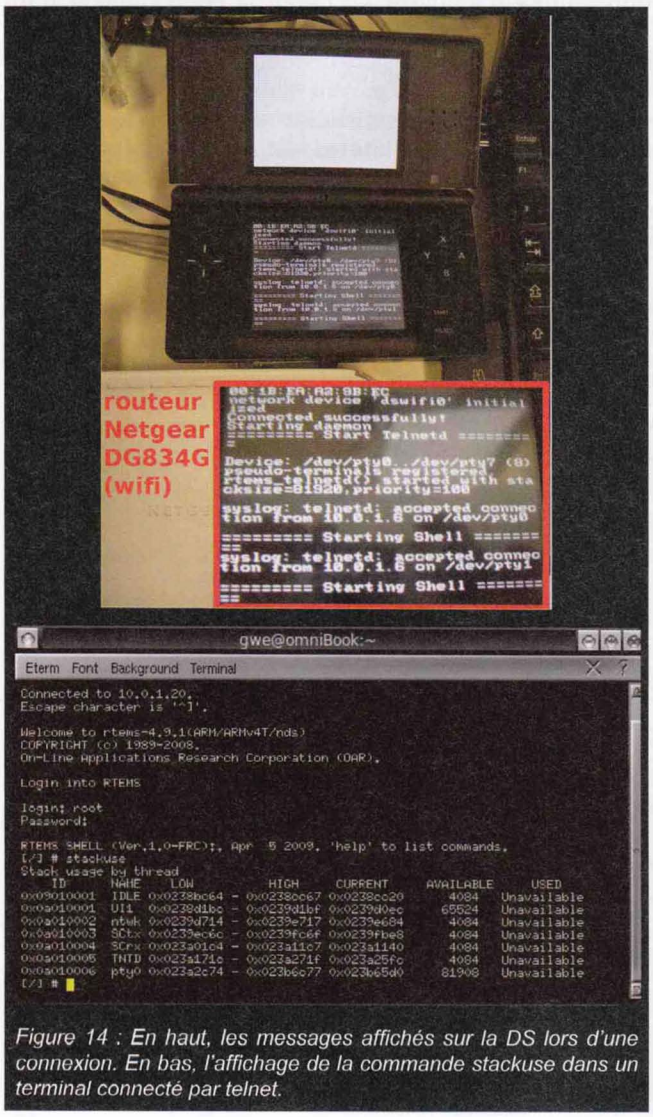


Figure 14 : En haut, les messages affichés sur la DS lors d'une connexion. En bas, l'affichage de la commande stackuse dans un terminal connecté par telnet.

La connexion de l'utilisateur se fera grâce à la commande `telnet ip_de_la_DS`.

6.6 Transfert de données issues du convertisseur analogique-numérique

Le dernier exemple repose sur une reprise de l'application de conversion analogique-numérique, présentée plus haut pour DSLinux. Dans ce cas, les valeurs ne seront plus affichées sur la DS, mais vont être transférées à un ordinateur personnel à travers la connexion wifi via un point d'accès.

Ce code d'exemple a été allégé de l'initialisation du réseau qui se fait exactement comme dans le premier exemple (section 6.4) :

```
#include <stdlib.h>
#include <stdio.h>
#include <bsp.h>
#include <rtems/telnetd.h>
#include <nds/memory.h>
#include <rtems/rtems_bsdnet.h>

// [...] CONFIGURATION DU RESEAU
```

```
#define TAILLE 1024 /* callback for telnet */
void telnetADC( char *pty_name, void *cmd_arg) {
  char *c;
  int f;

  printk( "Connected to %s with argument %p \n", pty_name, cmd_arg );

  c=(char*)malloc(TAILLE);

  while (1) {
    for (f=0;f<TAILLE;f++) {
       *(unsigned short*)(0x8000000)=(unsigned short)0;
       c[f]=*(unsigned short*)(0x8000000)&0xff;
    }
    for (f=0;f<TAILLE;f++) printf("%x ",c[f]);
    printf("\n");
  }
}
/* Init task */
rtems_task Init(rtems_task_argument argument){
 fprintf(stderr, "\n\n*** Telnetd Server Test ***\n\r" );
  fprintf(stderr, "======== Initializing Network ========\n");
  rtems_bsdnet_initialize_network ();
  fprintf(stderr, "======== Start Telnetd ========\n");
  (*(volatile uint16_t*)0x04000204) = ((*(volatile uint16_t*)0x04000204)
& ~ARM7_OWNS_ROM);
  rtems_telnetd_initialize(
     telnetADC, /* callback function */
     NULL, /* no context necessary for echoShell */
     false, /* false == listen on sockets */
     RTEMS_MINIMUM_STACK_SIZE * 20, /* shell needs a large stack */
     1, /* priority .. we feel important today */
     false /* telnetd does NOT ask for password */
  ); while(1);
}

#include "../rtems_common.h"
```

Tableau 8 : Code complet de l'application de conversion analogique-numérique vers le réseau

réseau. Du côté client, la solution la plus simple est l'utilitaire **telnet**, puisque nous affichons en hexadécimal et en ASCII le résultat des mesures. Les valeurs sont alors transmises à l'utilisateur distant, qui peut les enregistrer par redirection dans un fichier avec une commande telle que : **telnet ip > acquiss.ds**.

Un traitement ultérieur avec GNU/Octave a permis d'obtenir la courbe présentée en figure 16. Nous y observons une sinusoïde – en accord avec le signal injecté depuis un synthétiseur de fonctions – de fréquence variable, puisque nous avons successivement programmé une fréquence de 500 et 1000 Hz. Les discontinuités entre les séries successives de 1024 points résultent de l'absence d'un second tampon chargé des mesures pendant le transfert par wifi des données accumulées.

En comptant le nombre de points de mesure par période, nous déduisons une fréquence d'échantillonnage de l'ordre de 300 kHz.

Nous avons donc atteint notre objectif de transmission de données au moyen d'un protocole réseau TCP/IP sur support wifi au moyen de la DS. Cette application a fourni l'opportunité d'analyser en détail le cheminement d'un paquet IP dans RTEMS. Néanmoins, un point d'amélioration subsiste avec cette nécessité de fournir la configuration du point d'accès distant au moyen d'un jeu commercial. Bien que cette solution soit satisfaisante pour une application ludique, il pourrait être intéressant d'ajouter la possibilité de pouvoir soit utiliser les données stockées dans la DS, soit de fixer les paramètres à travers RTEMS et le rendre indépendant d'une application ou d'un jeu tiers.

Comme présenté précédemment, la bibliothèque **telnetd** ne fait qu'appeler une fonction (**telnetADC**) lors de la connexion d'un utilisateur. Au lieu d'un shell, il est possible de lancer la conversion, tel que présenté Fig. 15. Les descripteurs d'entrée et de sortie standards (**stdin**, **stdout**) pointant sur la *socket* à laquelle est connectée l'application cliente côté PC, les valeurs obtenues, affichées avec **printf**, sont automatiquement transférées par le

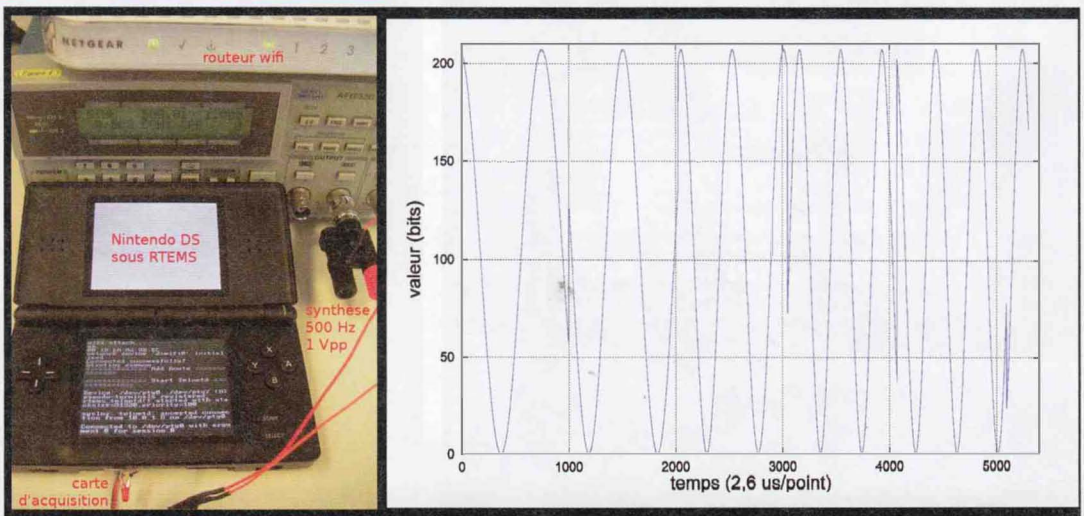


Figure 15 : Gauche : montage pour l'acquisition de signaux par conversion analogique-numérique sous RTEMS et transfert par paquets de 1024 mots des données par Wifi au travers d'un routeur. Droite : signaux acquis, dans un premier temps une sinusoïde à 500 Hz puis dans un second temps à 1 kHz.

Conclusion et perspectives

Nous avons donc proposé dans cette présentation d'exploiter notre système d'exploitation favori pour tirer le meilleur parti d'une console de jeu, et étendu nos compétences en exécutant quelques applications simples d'un environnement

de développement d'applications temps-réel. Ces activités auront été l'occasion d'appréhender ce qui est possible (et ce qui ne l'est pas) avec des ressources réduites en termes de puissance de calcul et surtout de mémoire – réduites certes, mais malgré tout importantes par rapport à ce que proposent de nombreux systèmes embarqués. Nous nous sommes efforcés de compléter les fonctionnalités de la console de jeu par des périphériques dédiés plus appropriés pour des applications de contrôle d'expériences ou d'acquisitions de données.

Nous avons étendu notre champ d'investigation à un environnement de travail plus approprié pour les ressources de la DS, en exploitant l'environnement exécutif RTEMS. Après avoir reproduit un certain nombre des fonctionnalités exploitées sous DSLinux, nous avons rendu fonctionnelle la liaison wifi pour exploiter pleinement les périphériques de la console.

Une fois que nous maîtrisons les concepts de communication bidirectionnelle sur le bus de données, de gestion des signaux de contrôle et de l'interruption fournie par IRQ, nous sommes capables d'interfacer à peu près n'importe quel composant numérique à la DS via son slot 2. Un exemple intéressant pour approfondir ces connaissances serait d'ajouter un UART pour la communication asynchrone (type RS232), périphérique qui manque cruellement à cette console de jeu[16] pour une liaison avec microcontrôleurs ou PC sans passer par une couche aussi lourde que la liaison sans fil.

Ces briques de base ouvrent des perspectives intéressantes d'exploitation d'une console, activité probablement plus ludique que la majorité des jeux disponibles.

Remerciements

Pierre Kestener (CEA/IRFU, Saclay) nous a informé de la disponibilité du BSP pour Nintendo DS de RTEMS. Matthieu Bucchianeri a patiemment répondu à nos questions alors que nous découvrions RTEMS.

Références

[1] RAFIQUZZAMAN (M.), *Microprocessors and microcomputer-based system design*, 2nd Ed., CRC Press, 1995.

[2] FRIEDT (J.-M.) & CARRY (É.), « Développement sur processeur à base de cœur ARM7 sous GNU/Linux », *GNU/Linux Magazine France* 117, juin 2009, p. 40-59.

[3] GUIDON (Y.), « Programmation graphique : vers le framebuffer et au-delà », *GNU/Linux Magazine France* 112, p. 82-94, janvier 2009.

[4] RUBINI (A.) & CORBET (J.), *Linux Device Drivers*, 2nd Ed., O'Reilly, 2001, disponible à **http://www.xml.com/ldd/chapter/book/ch09.html\#t8.**

[5] FICHEUX (P.) & KADIONIK (P.), « Temps réel sous Linux (reloaded) », *GNU/Linux Magazine France*, Hors-série 24, février/mars 2006), disponible à **http://www.unixgarden.com/index.php/comprendre/temps-reel-sous-linux-reloaded.**

[6] BODOR (D.), « Inspectez le trafic réseau en utilisant la libpcap », *GNU/Linux Magazine France*, Hors-série 42, juin/juillet 2009, p. 52-58.

Notes

1 **http://dslinux.org**

2 **www.m3adapter.com**

3 **http://www.megaupload.com/?d=J3T9UVPE**

4 **http://stsp.spline.de/dslinux/toolchain**, version 2008-01-24 à la date de rédaction de cet article.

5 **http://nocash.emubase.de/gbatek.htm\#dscartrumblepak**

6 **http://nocash.emubase.de/gbatek.htm\#dsmemorymaps**

7 **http://www.rtems.com/**

8 P. Kestener nous informe qu'un shell capable de charger dynamiquement des applications (modules) – le Cexp – est disponible à **http://www.slac.stanford.edu/~strauman/rtems/cexp/index.html**

9 http://www.rtems.com/onlinedocs//doc-current/share/rtems/pdf/started.pdf, §4.1.4.2, page 22.

10 Dont une solution, fonctionnelle à défaut d'être élégante, est fournie à **http://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg191702.html**

11 Disponibles à **http://rtems.org/wiki/index.php/RTEMS_CVS_Repository**

12 **http://www.rtems.com/onlinedocs//doc-current/share/rtems/html/shell/shell00008.html**

13 La compilation d'une application exploitant la bibliothèque des POSIX threads s'obtient dans l'environnement issu de make xsh par **`$CC $CFLAGS $LDFLAGS -L/home/jmfriedt/dslinux/uClibc/lib/ fb_thread.c -o fb_thread -lpthread`**

14 Liste des *Access points* compatibles disponible sur **http://www.nintendo.com/consumer/wfc/en_na/ds/routerInfo.jsp**

15 **http://www.akkit.org/dswifi/**

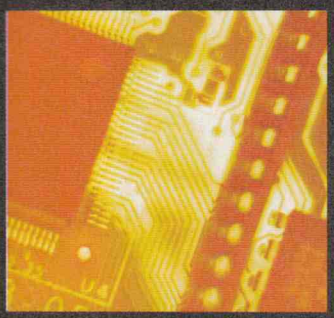
16 Par exemple, le NXP – anciennement Philips – SCC2691 présente les fonctionnalités recherchées avec un nombre raisonnable de pattes pour un coût unitaire inférieur à 7 euros chez Farnell.

Auteurs : Jean-Michel Friedt & Gwenhaël Goavec-Merou

Gwenhaël Goavec-Merou finit son master en informatique et est actuellement en stage chez ARMadeus Systems. Il est propriétaire d'une DS... Citroën.

Jean-Michel Friedt est membre de l'association Projet Aurore pour la diffusion de la culture scientifique et technique sur le campus bisontin de l'université de Franche-Comté. Il s'intéresse aux développements de systèmes embarqués, dont les consoles de jeu portables fournissent un excellent support. Quand il ne code pas du logiciel libre, il est ingénieur dans la société Senseor.

Initiation au calcul intensif



Sony avec sa fameuse console de jeux Playstation 3 a en réalité conçu un calculateur très puissant. La puissance de calcul requise pour un divertissement en haute définition est si grande qu'on peut envisager d'en faire un tout autre usage. La possibilité d'installer GNU/Linux sur cette console la métamorphose en un ordinateur dont le ratio coût/performance est sans doute encore en ce début 2009 le meilleur du marché, grâce aux capacités et à l'architecture originale de son processeur Cell BE pour lequel un kit de développement est librement distribué.

Auteur

- Sébastien Devaux

1 Un jouet comme calculateur hautes performances ?

1.1 Fiche technique de la Playstation 3

La console de jeux Playstation 3, comme toute console de jeux moderne est en fait une véritable unité centrale d'ordinateur. Afin de pouvoir produire des animations en trois dimensions fluides et en haute définition, elle contient des moyens de calcul très performants qui sont principalement un processeur multi-coeur à unités vectorielles, un processeur graphique et un bus mémoire rapide. La configuration la plus répandue est la suivante :

- processeur Cell BE, 3,2 GHz, 1 cœur PowerPC accompagné de 7 unités vectorielles dont une est réservée au système ;
- mémoire vive : 256 Mo XDR ;
- processeur graphique NVidia RSX 550 MHz ;
- mémoire vidéo : 256 Mo GDDR2 à 700 MHz ;
- port réseau Ethernet 1 Gb/s ;
- wifi 802.11g (54 Mb/s) ;
- ports USB ;
- Bluetooth 2.0 ;
- disque dur SATA 2 pouces 1/2 de 40 Go. À noter que ce disque peut être remplacé sans difficulté par un de plus grande capacité ;
- lecteur optique Blu-Ray ;
- ports audio, vidéo et HDMI.

Pour transformer cette console en ordinateur, l'investissement matériel se réduit à un clavier, une souris USB et un moniteur. Pour ce dernier, une entrée HDMI est obligatoire si on souhaite profiter d'une résolution d'affichage correcte, car avec un signal vidéo analogique classique, il faudra se contenter du mode console à 80 colonnes.

1.2 Le processeur Cell BE

Ce processeur conçu conjointement par IBM, Sony et Toshiba a été construit selon une architecture multi-cœur hybride assez originale. Il est constitué d'un cœur principal de type PowerPC 64bits avec support du jeu d'instruction Altivec (plus ou moins équivalent à MMX et SSE). Il est « *multithread* » (SMT) à deux voies. Les autres cœurs, au nombre de huit, sont des unités de calcul vectoriel. C'est-à-dire que ce sont des unités capables d'accomplir en une seule opération le même calcul sur un paquet de données, ce qui est très efficace pour tout type de calcul matriciel et parfaitement adapté à des applications typiques comme le traitement du signal, la compression/décompression de données, les calculs 3D, bref tout ce qui est requis pour les jeux vidéo actuels et les loisirs numériques en général, mais pas seulement.

Pour reprendre la terminologie du fabricant, le cœur principal est appelé *Power Processing Element* (PPE) et les cœurs vectoriels sont appelés *Synergistic Processing Element* (SPE). Tous ces cœurs incorporent un contrôleur d'accès direct à la mémoire, via un bus d'interconnexion interne à la puce Cell BE.

Le PPE contient un cache de niveau 2 de 512 Ko et l'unité de calcul Power (PPU). Le PPU comprend entre autres :

Le jeu complet de registres PowerPC d'une longueur de 64 bits :

- 32 registres vectoriels de 128 bits ;
- un cache niveau 1 d'instruction de 32 Ko ;
- un cache niveau 1 de données de 32 Ko ;
- des unités de calcul entier, flottant, vectoriel ;
- un gestionnaire de mémoire virtuelle.

Un SPE contient un contrôleur DMA et l'unité de calcul vectorielle (SPU) qui elle-même dispose d'une mémoire locale (local store). Le SPU dispose de 128 registres de 128 bits et d'un jeu d'instructions spécifique : SIMD. La mémoire locale est d'une taille de 256 Ko.

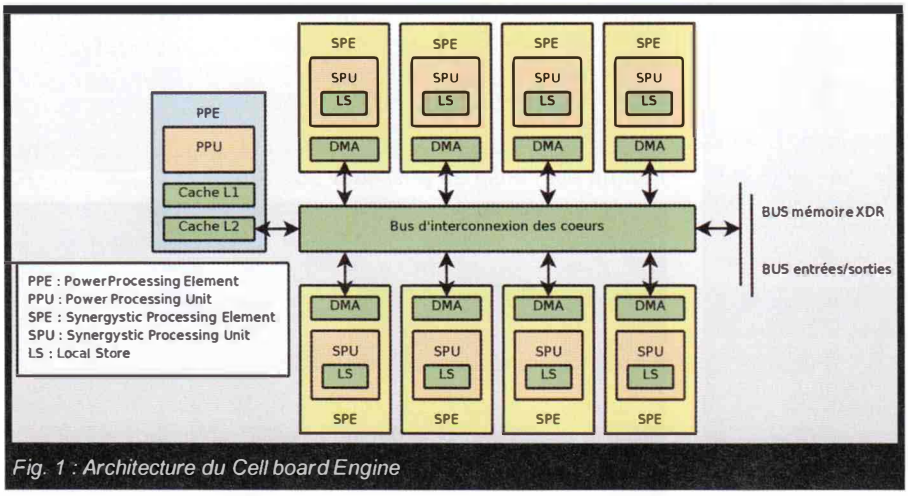


Fig. 1 : Architecture du Cell board Engine

La puissance de calcul maximum théorique est calculée par l'addition de la puissance de chaque élément ce qui donne environ 200 GFlops (milliards d'instructions sur nombre flottant par seconde).

Bien que le Cell BE soit toujours fabriqué avec 8 SPE, sa déclinaison pour la Playstation 3 n'en conserve que 7 actifs, afin de réduire les rejets en sortie de fabrication et ainsi réduire les coûts de production. De plus, l'un d'entre eux est complètement réservé au système natif et n'est pas disponible pour les programmeurs.

1.3 Applications

Une telle puissance de calcul peut évidemment être employée à tout autre chose que le jeu vidéo. On citera par exemple :

- Chiffrement et déchiffrement en flux tendus, signature numérique de gros volumes de données.
- Traitement du signal :
- Encodage, décodage, filtrage,
- Compression/décompression.
- Production audio-vidéo :
- Calcul de rendu d'images de synthèse haute définition,
- Post production audio-vidéo,
- Traitements audio-numériques temps-réels.
- Sciences de l'ingénieur :
- Simulation mécanique,
- Simulation électronique,
- Calcul scientifique.

Dans cette dernière catégorie, le département de physique de l'université de Dartmouth **[1]** a construit une petite grille de calcul **[2]** à l'aide de 16 Playstation 3 qu'il emploie, entre autres, pour simuler la propagation des ondes gravitationnelles à proximité de trous noirs.

Disposer de grandes puissances de calcul dans un petit volume financièrement abordable permet d'envisager l'ajout d'interactivité sur des traitements qui jusque-là ne pouvaient être réalisés que par lots en temps différé, et ceci même pour des petites structures (PME, Laboratoires,...).

2 Premiers pas vers la programmation du Cell

Note

Les sources complets illustrant cet article sont disponibles dans une archive complémentaire **[3]**. Sa structure suit les étapes de progression de cet article :

- step1 : deux architectures, deux compilateurs ;
- step2-1 et step2-2 : intégration des tâches PPU et SPU ;
- step3 : communication entre SPU et PPU ;
- step4 : boîte à message ;
- step5 : instructions vectorielles.

IBM publie en accès libre sur son site web la documentation de référence **[4]** du CellBE et de son kit de développement.

2.1 Installation de Linux sur PS3

2.1.1 Préparation de la PS3

Par construction, la Playstation permet l'installation d'un système d'exploitation alternatif. Il n'est pas nécessaire d'ajouter une puce ou de recourir à quelques acrobaties exploitant un bug d'un jeu comme le désormais célèbre Twilight Hack **[5]** de la console Nintendo Wii. Non, pour une fois, il n'y a aucune inquiétude à avoir concernant la sécurité et la légalité de l'opération. Elle est légitime et décrite dans le manuel utilisateur. Si on souhaite installer un système autre que le système natif Xcross Media Bar, il suffit d'accomplir les opérations suivantes :

1 - Se munir d'un clavier et d'une souris USB à brancher directement sur les ports USB de la console.

2 - Configurer un accès internet même s'il n'est pas utilisé, simplement pour activer une interface réseau avant l'installation du nouveau système.

3 - Créer une partition pour le nouveau système. Il n'est pas permis d'utiliser l'intégralité du disque pour le nouveau système, 10 Go minimum sont conservés obligatoirement pour le système natif. (Menu : ***paramètres / paramètres système / Format / Formater disque dur***).

Attention

Le formatage du disque entraîne un effacement de tout ce qu'il contient. Pensez à sauvegarder son contenu s'il héberge des données personnelles.

4 - Insérer le DVD d'installation du nouveau système

5 - Activer la recherche d'une installation depuis le DVD-ROM (Menu : ***paramètres / paramètres système / installer autre système d'exploitation***)

6 - Basculer le démarrage sur le nouveau système (Menu : ***paramètres / paramètres système / système par défaut / autre système***)

7 - Redémarrer la console.

Note

À tout moment, il est possible de rétablir le démarrage du système Xross Media Bar. Il suffit de procéder à une mise hors tension, puis une mise sous tension de la console avec une pression prolongée (environ cinq secondes) sur « l'interrupteur » de la console (c'est-à-dire la touche sensitive 0/1 située sur la face avant).

La distribution Linux que nous avons utilisée possède aussi la commande `ps3-boot-game-os` qui produit le même effet. Pour activer à nouveau le démarrage du système alternatif, il faudra répéter les étapes 6 et 7 de la procédure de préparation de la console.

2.1.2 Boot et installation du système

Nous avons installé le système Yellow dog linux **[6]**. Ce système est dérivé d'une distribution RedHat PPC. La procédure d'installation via Anaconda est quasiment identique et suffisamment bien commentée et documentée pour ne pas s'étendre plus sur le sujet ici. Il conviendra toutefois de faire attention à deux points particuliers :

- Les modes vidéo disponibles dépendent fortement de l'écran disponible et, en l'absence de moniteur équipé en HDMI, il faudra se contenter la plupart du temps d'un signal analogique PAL qui ne permet pas un affichage correct d'un quelconque mode graphique. On prendra soin dans ce cas de procéder à une installation en mode console.
- L'interface réseau activée par défaut (c'est-à-dire soit Ethernet, soit wifi) sera celle qui a été configurée pour l'accès à internet depuis le système Xross Media Bar.

2.1.3 Exploration de la PS3 avec Linux

Malgré son apparente ouverture, la console de Sony reste un système propriétaire qui ne se dévoile que partiellement au système alternatif. Il y a en fait une sorte d'hyperviseur qui démarre Linux dans une machine virtuelle très légère. C'est-à-dire qu'une couche intermédiaire dissimule partiellement certaines fonctionnalités du matériel :

À peine 210 Mo de mémoire RAM sont utilisables alors que l'engin est censé en posséder 256 Mo.

```
/* ### Espace mémoire accessible depuis Linux ### */
sdevaux@porto:~$ free
             total       used       free     shared    buffers     cached
Mem:        215504     212644       2860          0        888      56744
-/+ buffers/cache:     155012      60492
Swap:       522104      10072     512032
```

La console possède à la fois un contrôleur wifi et un contrôleur Ethernet, mais un seul des deux est visible par Linux. Tout dépend de celui qui a été activé lors de la configuration du réseau depuis le système Xcross Media Bar originel de la console. Dans l'exemple ci-après, nous avions choisi le port Ethernet.

```
/* ### inventaire de interface réseau ### */
sdevaux@porto:~$ /sbin/ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:1D:0D:B6:2F:FD
          inet adr:192.168.0.4  Bcast:192.168.0.255  Masque:255.255.255.0
          adr inet6: fe80::21d:dff:feb6:2ffd/64 Scope:Lien
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3678234 errors:0 dropped:0 overruns:0 frame:0
          TX packets:355509 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 lg file transmission:1000
          RX bytes:727329249 (693.6 MiB)  TX bytes:51908744 (49.5 MiB)
          Interruption:52

lo        Link encap:Boucle locale
          inet adr:127.0.0.1  Masque:255.0.0.0
          adr inet6: ::1/128 Scope:Hôte
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:87652 errors:0 dropped:0 overruns:0 frame:0
          TX packets:87652 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 lg file transmission:0
          RX bytes:9620166 (9.1 MiB)  TX bytes:9620166 (9.1 MiB)
```

Même si le port virtuel eth0 est le port Ethernet, sa capacité à basculer sur un port wifi est visible.

```
/* ### Extension wifi visible même quand eth0 pointe sur l'interface ethernet. ### */
sdevaux@porto:~$ /sbin/iwconfig
lo        no wireless extensions.
```

```
Warning: Driver for device eth0 has been compiled with version 22
of Wireless Extension, while this program supports up to version 20.
Some things may be broken...

eth0      IEEE 802.11bg  ESSID:off/any
          Mode:Managed  Channel:0  Access Point: Not-Associated
          Bit Rate:54 Mb/s
          Link Quality:0  Signal level:0  Noise level:0
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

L'accès au lecteur Blu-Ray semble limité. On peut y lire depuis Linux des CD/DVD-ROM/BRD-ROM, mais pas de BRD-video. Nous n'avons pas testé la lecture de DVD-Vidéo selon les moyens habituellement utilisés sous Linux.

Le disque dur, bien qu'interchangeable, n'est pas entièrement accessible :

```
sdevaux@porto:~$ df
Sys. de fich.        1K-blocs       Occupé Disponible Capacité Monté sur
/dev/ps3da2          24789380   4061840  19447964  18% /
/dev/ps3da1            101086     20135     75732  22% /boot
tmpfs                  107752         0    107752   0% /dev/shm
```

Ce qui nous intéresse principalement, le processeur est par contre complètement visible, dans la limite des 6 SPE sur les 8 gravés comme déjà indiqué précédemment :

```
sdevaux@porto:~$ cat /proc/cpuinfo
processor       : 0
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 16.0 (pvr 0070 1000)

processor       : 1
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 16.0 (pvr 0070 1000)

timebase        : 79800000
platform        : PS3
```

Des outils et commandes spécifiques ont été créés pour administrer cette plateforme particulière :

```
sdevaux@porto:~$ spu-ps
  PID   TID USERNAME   S F SPE     TIME BINARY
15248 15252 sdevaux    S     0 22525.825 setiathome_2.3A
15248 15251 sdevaux    S     1 22525.829 setiathome_2.3A
15248 15250 sdevaux    S     2 22525.833 setiathome_2.3A
15987 15991 sdevaux    U     3   778.335 setiathome_2.3A
15987 15990 sdevaux    U     4   451.602 setiathome_2.3A
15987 15989 sdevaux    U     5   405.364 setiathome_2.3A
sdevaux@porto:~$ spu-top
spu-top: Context View
Cpu(s) load avg:   0.0%,  0.0%,  0.0%
Spu(s) load avg:   7.0%,  6.8%,  4.9%
Cpu(s):  0.0%us,  0.5%sys,  0.0%wait,  0.0%nice, 99.5%idle
Spu(s):  0.0%us,  0.0%sys,  0.0%wait,100.0%idle

  PID   TID USERNAME   S F  %SPU SPE     TIME BINARY
11662 11668 sdevaux    L     0.0   0    0.000 step2-1
11662 11667 sdevaux    L     0.0   1    0.000 step2-1
11662 11666 sdevaux    L     0.0   2    0.000 step2-1
11662 11665 sdevaux    L     0.0   3    0.000 step2-1
11662 11664 sdevaux    L     0.0   4    0.000 step2-1
11662 11663 sdevaux    L     0.0   5    0.000 step2-1
```

Pour un processeur dont l'architecture est assez originale, il faut une suite de compilation dédiée :

```
/* ### Installation de la suite de compilation GNU pour le Cell BE ### */
yum install ppu-gcc ppu-binutils ppu-gdb spu-gcc spu-tools
```

Au final, ceci devrait conduire à disposer des paquets suivants :

```
/* ### Paquets spécifiques PPC/Cell ### */
ppu-gcc.ppc                    4.1.1-57              installed
ppu-binutils.ppc               2.17.50-32.ydl.1      installed
ppu-gdb.ppc64                  6.6.50-28             installed
ppu-gdb.ppc                    6.6.50-28             installed
spu-binutils.ppc               2.17.50-33            installed
spu-tools.ppc                  1.0-2                 installed
spu-gdb.ppc                    6.6.50-12.ydl.1       installed
spu-gcc.ppc                    4.1.1-107             installed
spu-gcc-c++.ppc                4.1.1-107             installed
crash-spu-commands.ppc         1.0-2.ydl.1           installed
spu-newlib.ppc                 1.15.0-82             installed
spu-gcc-fortran.ppc            4.1.1-107             installed
ppu-gdb.ppc64                  6.6.50-28             installed
ppu-gdb.ppc                    6.6.50-28             installed
libspe.ppc64                   1.2.2-2               installed
libspe-devel.ppc               1.2.2-2               installed
libspe2man.noarch              2.2.0-5               installed
libspe.ppc                     1.2.2-2               installed
elfspe2.ppc                    2.2.0-91              installed
libspe2.ppc64                  2.2.0-91              installed
libspe2-devel.ppc              2.2.0-91              installed
libspe-devel.ppc64             1.2.2-2               installed
libspe2.ppc                    2.2.0-91              installed
libspe2-devel.ppc64            2.2.0-91              installed
```

Note

Les heureux possesseurs de plusieurs PS3 (!) pourront profiter du portage de OpenMPI [7], une API de communication dédiée à la mise en œuvre de grilles de calcul :

```
/* ### Installation OpenMPI ### */
yum install openmpi openmpi-devel openmpi-libs
```

Note

Une version optimisée pour le Cell du programme d'analyse de signaux radio lointains SETI@Home [8] est disponible sur le site de G. Khanna [9], chercheur du « groupe de recherche gravité » [10] de l'université Dartmouth du Massachusetts [11].

2.2 Deux architectures, deux compilateurs

Dans un premier temps, on peut assimiler le processeur Cell à un système multiprocesseur qui a la particularité de posséder des processeurs de types différents, dans le cas de la PS3 :

- un processeur PowerPC ;
- six processeurs vectoriels SPU.

Pour chaque type, on utilise un compilateur C dédié qui pourra prendre en charge les mêmes programmes source à partir du moment où toutes les bibliothèques utilisées sont compatibles avec le type de processeur ciblé.

Prenons un premier programme simple (**step1.c**) qui ne fait qu'afficher un message à l'écran et attendre la frappe d'une touche pour s'arrêter, ce qui nous permettra de le laisser actif indéfiniment pour nous laisser le temps de faire quelques observations :

```
#include <stdlib.h>

int main() {
        printf("Step1 example is running.\n");
        printf("Hit key to exit...");
        getchar();
        printf("\ndone.\n");
}
```

Compilons ce programme comme un simple programme PowerPC :

```
ppu-gcc -o step1.ppu step1.c
```

Ensuite, compilons-le pour l'architecture vectorielle :

```
spu-gcc -o step1.spu step1.c
```

Et lançons les deux binaires obtenus à tour de rôle en tâche de fond pour observer l'état du système pendant leur exécution :

```
/* ### Trace exécution step1 ### */
sdevaux@porto:~/local/didacdev/step1$ ./step1.ppu &
[1] 16139
Exemple Step1 en cours d'execution.
Une touche pour continuer...
[1]+  Stopped                 ./step1.ppu
sdevaux@porto:~/local/didacdev/step1$ ps | grep step
16139 pts/0    00:00:00 step1.ppu
sdevaux@porto:~/local/didacdev/step1$ fg
./step1.ppu
done.
sdevaux@porto:~/local/didacdev/step1$ ./step1.spu &
[1] 16148
Exemple Step1 en cours d'execution.
Une touche pour continuer...
[1]+  Stopped                 ./step1.spu
sdevaux@porto:~/local/didacdev/step1$ ps -ef | grep step
sdevaux  16148 15919  0 11:41 pts/0    00:00:00 /usr/bin/elfspe ./step1.spu
sdevaux@porto:~/local/didacdev/step1$ spu-ps
  PID   TID USERNAME   S F SPE     TIME BINARY
16148 16148 sdevaux    L     5    0.000 step1.spu
sdevaux@porto:~/local/didacdev/step1$ fg
./step1.spu
done.
sdevaux@porto:~/local/didacdev/step1$ spu-ps
  PID   TID USERNAME   S F SPE     TIME BINARY
```

Que s'est-il passé ? Le lancement de **step1.ppu** n'a rien d'inhabituel, l'unité PowerPC étant en quelque sorte l'unité principale qui prend en charge le système d'exploitation, tout fonctionne de façon classique, le programme démarre et les outils habituels d'observation des processus en rendent compte normalement. Par contre, le lancement de **step1.spu** révèle des différences importantes. La commande **ps** montre qu'en réalité un programme nommé **elfspe** a été démarré avec comme paramètre notre programme **step1.spu**. Il s'agit en fait d'un utilitaire qui va s'occuper d'allouer à un SPU libre notre programme, comme le montre la commande **spu-ps**, qui est une déclinaison spécifiques aux SPU de la commande **ps**.

Cette étape montre que le Cell est un système multiprocesseur à deux architectures qui peut prendre en charge simultanément plusieurs programmes ce qui démultiplie d'autant sa puissance et qui est tout à fait semblable aux autres architectures multi-cœur mises en œuvre maintenant par la plupart des fabricants de processeurs. Mais, le Cell va déjà plus loin, puisque, outre le cœur PowerPC, il propose jusqu'à huit cœurs vectoriels (six « seulement » dans la version PS3).

2.3 Intégration des tâches PPU et SPU

Malgré ce que peut laisser croire l'exemple précédent, les unités vectorielles n'ont pas vocation à prendre en charge des programmes complets. Il faut plutôt les considérer comme des coprocesseurs spécialisés dans le calcul vectoriel à disposition des programmes plus conventionnels qui seront élaborés pour le cœur PowerPC. Il devient donc indispensable de pouvoir depuis un programme PPU, déléguer des tâches à une routine SPU.

Ainsi, il est possible, depuis un programme PPU, de démarrer des tâches SPU. Pour cela, on commence d'abord par écrire le code de la tâche déléguée au SPU. Dans l'exemple minimaliste qui suit, ce programme SPU est repris du paragraphe précédent et ne fait qu'afficher un message, puis attendre la frappe d'une touche.

```
/* ### spuprog.c ### */
#include <stdio.h>

int main(unsigned long long id) {
        printf("Entree dans le programme SPU. Id=0x%llx\n",id);
        getchar();
        printf("Fin du programme SPU. Id=0x%llx\n",id);
        return 0;
}
```

Dans le programme PPU, on commence par déclarer un *handler* sur le programme SPU auquel on délèguera les traitements définis dans **spuprog.c** :

```
/* ### extrait #1 step2-1.c ### */
#include <libspe2.h>
extern spe_program_handle_t spe_program;
```

Il suffit ensuite de créer un contexte d'exécution, de lui associer la tâche à réaliser pour enfin démarrer la tâche :

```
/* ### extrait #2 step2-1.c ### */
/* création du contexte */
spe_context_ptr_t ctx=spe_context_create(0,NULL);
/* chargement de la tâche pour le nouveau contexte */
spe_program_load(ctx,&spe_program);
/* demarrage de la tâche */
unsigned int entry=SPE_DEFAULT_ENTRY;
spe_context_run(ctx,&entry,0,NULL,NULL,NULL);
```

Afin de pouvoir confier simultanément cette même tâche définie par **spu_program** simultanément à tous les cœurs SPU, on réalisera les trois opérations ci-dessus dans un *thread* propre pour chaque SPU, car l'appel **spe_context_run** est bloquant jusqu'à terminaison de la tâche.

La compilation suit le même principe que pour l'exemple précédent, mais, cette fois, pour ne produire qu'un seul binaire, l'édition des liens est un peu particulière. Mais, tout d'abord, commençons par la compilation des deux programmes, pour chaque architecture :

```
/* ### Compilation ### */
ppu-gcc -c step2-1.c -o step2-1.o
spu-gcc -c spuprog.c -o spuprog.o
```

Ensuite, on réalise une première édition des liens du programme **spu**, puis on le prépare pour son encapsulation dans le programme **ppu**. C'est à cette étape que l'on associe réellement la tâche avec son handler dans le programme PPU. On lui attribuera pour cela lors de la procédure d'encapsulation le même identificateur que celui utilisé pour le handler dans le programme PPU :

```
/* ### Edition des liens SPU et encapsulation ### */
spu-gcc -o spuprog spuprog.o
ppu-embedspu spe_program spuprog spuprog.emb
```

On peut maintenant conclure par l'édition des liens finale du programme PPU en prenant soin de bien faire référence au programme SPU encapsulé :

```
/* ### Edition des lien PPU ### */
ppu-gcc -lspe2 -lpthread -o step2-1 step2-1.o spuprog.emb
```

À l'exécution, on observe bien chaque instance de tâche SPU qui démarre, puis qui s'arrête au fur et à mesure des frappes clavier. Il n'y a plus de processus **elfspu**, puisque le programme **step2-1** se charge lui-même de piloter les tâches SPU :

```
/* ### trace exécution step2-1 ### */
sdevaux@porto:~/local/didacdev/step2-1$ ./step2-1
Entree dans le programme SPU. Id=0x1001b950
Entree dans le programme SPU. Id=0x1001bd60
Entree dans le programme SPU. Id=0x1001c170
Entree dans le programme SPU. Id=0x1001c580
Entree dans le programme SPU. Id=0x1001c990
Entree dans le programme SPU. Id=0x1001cda0
Fin du programme SPU. Id=0x1001b950
Fin du programme SPU. Id=0x1001bd60
Fin du programme SPU. Id=0x1001c170
Fin du programme SPU. Id=0x1001c580
Fin du programme SPU. Id=0x1001c990
Fin du programme SPU. Id=0x1001cda0
Fin du programme OK.
```

Si, pendant l'exécution du programme, on observe l'état des processus, on observe bien un seul programme PPU et six instances du même programme sur le groupe de SPU.

```
/* ### état système pendant exécution step2-1 ### */
sdevaux@porto:~/local/didacdev/step2.1$ ps -fu sdevaux
UID        PID  PPID  C STIME TTY          TIME CMD
sdevaux  10356 10309  0 12:04 pts/1    00:00:00 ./step2-1
sdevaux@porto:~/local/didacdev/step2.1$ spu-ps
  PID   TID USERNAME   S F SPE     TIME BINARY
10356 10362 sdevaux    L     0    0.000 step2-1
10356 10361 sdevaux    L     1    0.000 step2-1
10356 10360 sdevaux    L     2    0.000 step2-1
10356 10359 sdevaux    L     3    0.000 step2-1
10356 10358 sdevaux    L     4    0.000 step2-1
10356 10357 sdevaux    L     5    0.000 step2-1
```

Jusque-là, nous avons envoyé sur plusieurs SPU, la même tâche. Il est bien entendu possible d'envoyer des tâches différentes. Pour cela, il faut simplement déclarer un handler spécifique pour chaque tâche et initialiser un contexte propre à chacune.

```
/* ### extrait step2-2.c simpliste ### */
/* Declarer deux handler de tache. */
extern spe_program_handle_t spe_programA;
extern spe_program_handle_t spe_programB;
/* ... */
/* initialiser et demarrer la tache A */
spe_context_ptr_t ctxA=spe_context_create(0,NULL);
spe_program_load(ctxA,&spe_programA);
unsigned int entryA=SPE_DEFAULT_ENTRY;
spe_context_run(ctxA,&entryA,0,NULL,NULL,NULL);
/* ... */
/* initialiser et demarrer la tache B */
spe_context_ptr_t ctxB=spe_context_create(0,NULL);
spe_program_load(ctxB,&spe_programB);
unsigned int entryB=SPE_DEFAULT_ENTRY;
spe_context_run(ctxB,&entryB,0,NULL,NULL,NULL);
```

Le fichier intégral **step-2.2.c** met en œuvre ceci un peu plus proprement pour ne pas avoir à faire un tel copier/coller de code pour chaque tâche.

La compilation suit le même principe que précédemment en l'appliquant à chaque tâche SPU, tâches décrites dans les fichiers **spuprogA.c** et **spuprogB.c**, pour produire les binaires prêts à encapsuler dans le binaire final en prenant bien garde de leur associer l'identificateur utilisé pour le handler correspondant dans le programme PPU :

```
/* ### compilation de deux tâches SPU ### */
spu-gcc -c spuprogA.c -o spuprogA.o
spu-gcc -o spuprogA spuprogA.o
ppu-embedspu spe_programA spuprogA spuprogA.emb
spu-gcc -c spuprogB.c -o spuprogB.o
spu-gcc -o spuprogB spuprogB.o
ppu-embedspu spe_programB spuprogB spuprogB.emb
```

La compilation du programme PPU est, elle aussi, quasiment identique. Il s'agira cette fois, lors de l'édition des liens, de faire référence à chaque encapsulation produite à l'étape précédente :

```
/* ### Compilation du programme PPU lié à deux tâches SPU ### */
ppu-gcc -c step2-2.c -o step2-2.o
ppu-gcc -lspe2 -lpthread -o step2-2 step2-2.o spuprogA.emb spuprogB.emb
```

Note

Pour faciliter la répétition des opérations de compilation, on se reportera aux fichiers *Makefile* joints aux exemples qui automatisent l'enchaînement de toutes ces commandes de génération.

L'exécution de cet exemple montre bien (par les mentions AAA et BBB) que deux tâches différentes ont bien été lancées :

```
/* ### trace exécution step2-2 ### */
sdevaux@porto:~/local/didacdev/step2.2$ ./step2-2
Entree dans le programme SPU AAA. Id=0x10023950
Entree dans le programme SPU BBB. Id=0x10023d60
Fin du programme SPU AAA. Id=0x10023950
Fin du programme SPU BBB. Id=0x10023d60
```

L'examen des processus pendant l'exécution révèle que ces deux tâches distinctes ont bien été prises en charge par le même processus :

```
/* ### état système pendant exécution step2-2 ### */
sdevaux@porto:~/local/didacdev/step2.2$ ps -fu sdevaux
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
sdevaux  10431 10309  0 12:30 pts/1    00:00:00 ./step2-2
sdevaux@porto:~/local/didacdev/step2.2$ spu-ps
  PID   TID USERNAME   S F SPE     TIME BINARY
10431 10433 sdevaux    L     4    0.000 step2-2
10431 10432 sdevaux    L     5    0.000 step2-2
```

3 Utilisation des particularités du Cell

3.1 Communication entre SPU et PPU

Les SPE ont accès directement à la mémoire tout comme le PPE, mais les données doivent d'abord être transférées dans l'espace de mémoire local du SPU avant de pouvoir les utiliser. L'architecture SPU propose pour cela un ensemble d'instructions spécifiques pour lire et écrire via l'interface DMA. Mais, avant d'aller lire la mémoire, il faut d'abord savoir où se trouvent les données à y prélever. Tout l'art de la communication entre SPU et PPU repose sur un habile échange de pointeurs pour informer le premier de la localisation des zones d'échange que le second lui aura au préalable alloué. La transmission d'un premier pointeur du PPU vers le SPU peut se faire au démarrage de la tâche SPU par le PPU via la transmission d'un argument lors de l'appel à **spe_context_run** :

```
/* ### Extraits step3.c - transmission d'un paramètre du PPU au SPU ### */
extern spe_program_handle_t spe_program;
typedef struct {
  /* structure de donnée décrivant les données à échanger
     entre SPU et PPU */
} shared_data_t
/* données partagées */
shared_data_t data __attribute__ ((aligned (16)));
/* initialisation du contexte SPU */
spe_context_ptr_t ctx=spe_context_create(0,NULL);
spe_program_load(ctx,&spe_program);
unsigned int entry = SPE_DEFAULT_ENTRY;
/* démarrage de la tâche avec transmission du
   pointeur sur la zone d'échange */
spe_context_run(ctx,&entry, 0, &data, NULL, NULL);
```

De son côté, le programme SPU reçoit le pointeur transmis en tant que paramètre de sa fonction **main**. Il sera alors possible de copier les données entre la mémoire DMA et la mémoire locale :

```
/* ### Extraits spuprog.c - lecture et écriture DMA ### */
#include <spu_mfcio.h>

typedef struct {
  /* Définition de la même structure. */
} shared_data_t

/* Point d'entrée, le pointeur transmis par le PPU est le paramètre arg */
int main(unsigned long long id, unsigned long long arg) {
  /* variable locale recevant les données partagée */
  shared_data_t locdata __attribute__ ((aligned (16)));
  /* définition du canal de communication DMA */
  int tag=1;
  int tag_mask=1<<tag;
  /* copie dans locdata des données DMA */
  mfc_get(&locdata,arg,sizeof(locdata),tag,0,0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_any();

  /* travail utile : exploitation et modification des données */
  /* ... */
  /* copie de locdata modifié dans l'espace DMA */
  mfc_put(&data,arg,sizeof(data),tag,0,0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_any();
}
```

La taille du bloc de données recopié par un appel à **mfc_get** ou **mfc_put** est limité (cf. note ci-après). Pour permettre des échanges plus souples et plus importants, la structure de données commune aux programmes SPU et PPU peut contenir des pointeurs dans l'espace mémoire PPU qui pourront être exploités à leur tour pour des échanges supplémentaires à l'aide de **mfc_put** et **mfc_get**.

Note

Alignement et tailles restreints

Le mode d'accès à la mémoire est fortement contraint par l'architecture particulière des SPE. Il est impératif que la structure de données commune soit alignée sur 16 octets (cf. **__attribute__ ((aligned (16))** – pour de meilleures performances un alignement de 128 octets est recommandé) et que la taille de la structure soit un multiple de 16 octets et au maximum 16384 octets.

Pour respecter ces contraintes, on prendra soin en particulier de compléter les structures d'échange par un champ de remplissage de taille adéquat pour atteindre la taille multiple de 16 octets la plus proche.

3.2 Boîte à message

La communication entre SPU et PPU peut également être prise en charge par un système de boîte à message. Il s'agit de files d'attente de messages courts (mots de 32 bits). Ce mode de communication sera donc réservé à des transmissions de codes d'instructions ou d'adresse, car il n'est pas adapté au transport de données volumineuses.

Chaque SPU dispose de trois boîtes :

- Une boîte de réception.
- Deux boîtes d'émission :
- boite d'émission ;

- boîte d'émission avec interruption.

La distinction étant la capacité de lecture bloquante pour la deuxième, par le PPU, des données qui y sont émises.

Notre code d'exemple est repris pour accomplir deux échanges :

- Côté PPU : envoi d'un message, puis attente d'une réponse.
- Côté SPU : attente d'un message puis envoi d'une réponse.
- Côté PPU, les boîtes de messages sont prises en charge par les fonctions **spe_in_mbox_write**, **spe_out_mbox_read** et **spu_out_intr_mbox_read**. Chacune demande en première argument un contexte SPE qui identifie l'instance de programme SPE ciblée. Diverses options sont également disponibles pour contrôler le flot des lectures et écritures, c'est-à-dire le nombre de messages à transférer, ainsi que le mode de blocage selon l'état courant des boîtes.

```
/* ### PPU envoi un message puis attend une réponse ### */
pthread_t thspu;
spe_context_ptr_t ctx;
unsigned int message=42;
/* initialiser le context spu */
ctx=new_spu_context(&spe_program);
/* Demarrer le thread spu. */
pthread_create (&thspu, NULL,&ppu_pthread_function,ctx);
/*
...
*/
/* envoyer un message par mbox au SPU */
spe_in_mbox_write(ctx,&message,1,SPE_MBOX_ANY_NONBLOCKING);
/* Lire la reponse */
spe_out_intr_mbox_read(ctx,&message,1,SPE_MBOX_ALL_BLOCKING);
printf("Reponse recu par PPU : %d\n",message);
```

Côté SPU, les fonctions duales **spu_read_in_mbox**, **spu_write_out_mbox** et **spu_write_out_intr_mbox** contrôlent l'autre extrémité du canal. Ces fonctions sont plus simples. Aucun paramètre n'est exigé pour préciser ni le canal de communication, ni le mode de lecture (bloquant, non bloquant...). Chaque SPU est relativement isolé et ne voit que ses propres boîtes d'émission et de réception. Seul le PPU peut adresser les boîtes de tous les SPU. Ce système ne permet donc pas une communication directe de SPE à SPE, un relais par le PPU est nécessaire.

```
/* ### SPU attend un message puis envoie une réponse ### */
unsigned int message=spu_read_in_mbox();
printf("Message recu SPU : %d\n",message);
getchar();
message++;
printf("Envoi reponse : %d\n",message);
spu_write_out_intr_mbox(message);
```

Il existe également, aussi bien du côté SPU que PPU, des fonctions permettant de connaître l'état courant des boîtes sans pour autant y lire ou y écrire un message.

Note

La possibilité de blocage sur les actions d'écriture et de lecture des boîtes à lettre permet de les détourner de leur vocation première d'échange de message pour synchroniser l'exécution des tâches entre les SPU et le PPU.

3.3 Instructions vectorielles

Une instruction vectorielle est une instruction qui va pouvoir s'appliquer, dans la limite de la taille des registres disponibles à plusieurs données simultanément. D'habitude, dès qu'une opération de calcul doit être menée sur des vecteurs ou des matrices, il est nécessaire de programmer des boucles pour parcourir les vecteurs et matrices afin de procéder au calcul sur chacun de leurs éléments. Ici, il n'est plus nécessaire de traiter les éléments un par un. Il subsiste toujours des boucles de parcours, mais, à chaque itération, on traite plusieurs éléments à la fois. La quantité dépend de la taille de chaque élément face à la taille des registres vectoriels.

Un jeu d'instruction vectoriel est disponible aussi bien sur les SPU que sur le PPU. Dans le cas de ce dernier, il s'agit du jeu d'instructions ALTIVEC, qui est à peu de chose près l'équivalent dans l'architecture PowerPC des extensions MMX ou SSE que l'on trouve maintenant communément sur la plupart des processeurs x86. Nous n'aborderons pas ici le jeu d'instructions ALTIVEC qui n'est pas spécifique au Cell BE, puisqu'il est disponible sur de nombreux processeurs de la famille PowerPC.

Pour utiliser explicitement les fonctionnalités vectorielles des SPU, le langage C a été étendu d'un nouveau modificateur de type, et l'API du Cell BE contient un ensemble de fonctions dédiées à la manipulation de ces types étendus. La taille d'un registre de calcul d'un SPU étant de 128 bits (i.e. 16 octets), le modificateur **vector** étend le type qui le suit en un tableau de la taille du registre :

- **vector char** est assimilable à un tableau de 16 caractères.
- **vector short int** est un tableau de 8 entiers courts.
- **vector float** est un tableau de 4 flottants.
- ...

Un ensemble de fonctions permet d'effectuer sur ces vecteurs toutes les opérations de base comme s'il s'agissait de simples scalaires. Ainsi, on pourra par exemple réaliser quatre additions entières en une seule opération :

```
/* ### quatre additions d'entiers en une opération ### */
vector int a={10,20,30,40};
vector int b={1,2,3,4};
vector int res=spu_add(a,b);
/* A l'issu de cette instruction c={11,22,33,44} */
```

La représentation en mémoire des types **vector** est compatible avec celle des tableaux classiques et le passage de l'un à l'autre est réalisé par un simple « cast » explicite. Les fonctions vectorielles, si bien utilisées, vont ainsi permettre de réduire significativement le nombre d'itérations nécessaires pour effectuer un calcul. Au lieu de parcourir les tableaux élément par élément, on déclare un pointeur sur une fenêtre de calcul dont la taille correspond à celle d'un registre vectoriel, et, à chaque itération, on déplace cette fenêtre de sa propre taille, et ce, quel que soit le type de données traitées :

```
/* ### Somme de deux vecteurs d'entiers cours ### */
/* Taille des vecteurs */
#define VECT_SIZE 32
/* Taille en octet des registres SPU (128bits = 16octets) */
#define REG_SIZE 16
signed short in1[VECT_SIZE];
signed short in2[VECT_SIZE];
signed short out[VECT_SIZE];
/* ... */
vector signed short * v1;
```

```
vector signed short * v2;
vector signed short * o;
int i=0;
int ipr=REG_SIZE/sizeof(signed short);
do {
   v1=(vector signed short *)&(in1[i]);
   v2=(vector signed short *)&(in2[i]);
   o=(vector signed short *)&(out[i]);
   *o=spu_add(*v1,*v2);
   i+=ipr;
} while(i<VECT_SIZE);
```

Conclusion

Les qualités et les performances du Cell sont indéniables, mais il ne révèlera son plein potentiel que pour des classes de problèmes « vectorisables », c'est-à-dire ceux pour lesquels les calculs peuvent être menés en parallèle. Ceci n'est le cas que pour des classes de problèmes particulières où le calcul global peut se modéliser sous forme matriciel, c'est-à-dire que le calcul de chaque élément du résultat ne dépend que de petits sous-ensembles si possibles disjoints des données d'entrées. Dit plus simplement, le CellBE calculera vite si le résultat final n'est pas dépendant de résultats intermédiaires eux-mêmes interdépendants.

Même si on est confronté à un tel problème, il n'est pas toujours aisé de le modéliser sous forme matriciel, ce n'est pas un mode de pensée naturel quand on a passé beaucoup de temps à réfléchir selon la démarche algorithmique procédurale classique. Toutefois, il y a quand même pas mal d'acquis dans ce domaine, puisque les techniques de calcul vectoriel sont utilisées de longue date dans le calcul scientifique, depuis que Cray Inc. **[12]** a innové dans le domaine et commercialisé les premiers calculateurs à architecture vectorielle.

Savoir calculer en parallèle ne suffit pas. Il faut aussi pouvoir coordonner cette architecture composite. La bibliothèque C fournie avec le SDK de base est assez bas niveau. Pour être productif, il faudra impérativement acheter ou développer une bibliothèque de plus haut niveau qui fournirait des services du genre :

- Socket PPE/SPE et SPE/SPE ;
- Sémaphores pour la synchronisation PPE/SPE et SPE/SPE.

Il existe une communauté de programmeur Cell assez active et de nombreux outils sont librement disponibles, parmi lesquels il y a :

- libBLAS **[13]** (*Basic Linear Algebra Subprograms*) : bibliothèque de calcul matriciel utilisée de longue date pour le calcul scientifique (initialement écrite en FORTRAN).
- OpenMPI **[14]** : si cette bibliothèque ne tire pas directement parti des spécificités du CellBE. Son implémentation sur Linux PPC permet un portage plus facile, vers une grille constituée de PS3, des applications qui y ont déjà recours.

Le Cell propose une architecture originale, même encore aujourd'hui, après la démocratisation des processeurs multi-cœur. Cette originalité est à la base de ses principaux atouts. Mais, avant de s'engager vers cette technologie, il faudra garder en mémoire deux contraintes importantes :

- Pour du code existant, il y aura obligatoirement un travail de portage non négligeable si on souhaite bénéficier de tout le potentiel du processeur.
- Le code porté ou écrit spécifiquement pour le Cell ne sera pas directement exploitable avec une autre architecture. Les fonctions avancées reposant sur des extensions du langage C et des bibliothèques spécifiques, le code produit n'est pas portable.

Références

- [1] http://www.umassd.edu/engineering/phy/
- [2] http://gravity.phy.umassd.edu/ps3.html
- [3] http://dexter.detexia.net/files/AFF065-2009-NT063_src.tar.gz
- [4] http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine
- [5] http://wiibrew.org/wiki/Twilight_Hack/fr
- [6] http://us.fixstars.com/products/ydl/
- [7] http://www.open-mpi.org/
- [8] http://setiathome.berkeley.edu/
- [9] http://gravity.phy.umassd.edu/~gkhanna/sah-cell.tar.gz
- [10] http://gravity.phy.emassd.edu
- [11] http://www.umassd.edu/
- [12] http://fr.wikipedia.org/wiki/Cray_Inc.
- [13] http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_blas_sdk30_2
- [14] http://www.open-mpi.org/

Auteur : Sébastien Devaux

Utilisateur GNU/Linux depuis 1998. Directeur Technique de Detexia, responsable du pôle R&D DexTER : http://dexter.detexia.net/

DÉCOUVREZ LA NOUVELLE FORMULE

CONTRÔLEZ-VOUS VRAIMENT LES FUITES D'INFORMATIONS ?

MISC N°44

QUAND VOS MACHINES DIFFUSENT VOS DONNÉES À VOTRE INSU

AU SOMMAIRE

SOCIÉTÉ

[04-13] All your pills are belong to us (1/2)

DOSSIER :
Compromissions électromagnétiques : Quand vos machines diffusent vos données à votre insu

[14-21] La compromission électromagnétique

[22-27] Émanations électromagnétiques compromettantes des claviers filaires et sans fil

[28-37] Organiser la fuite d'information d'un poste isolé : méthodes logicielles

RÉSEAU

[38-43] Attaque des numéros de séquences cryptographiques sur OSPF

CODE

[44-53] Des échanges SOAP propres et sans bavure : signature et chiffrement

SCIENCE & TECHNOLOGIE

[54-63] Courbes elliptiques et attaques par canaux auxiliaires

[64-74] Vulgarisation des aspects formels de la notion de furtivité

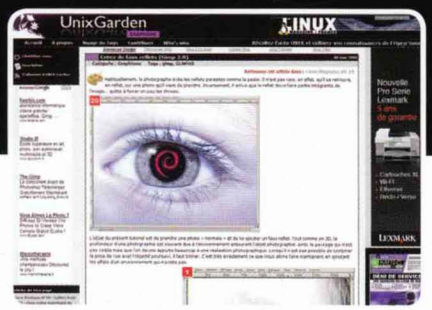
APPLICATION

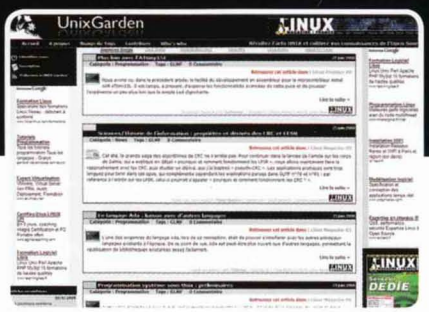
[76-82] La sécurité des wikis

ENCORE DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 4 SEPTEMBRE 2009

www.unixgarden.com

Récoltez l'actu UNIX et cultivez vos connaissances de l'Open Source !





Administration système

Utilitaires

Comprendre

Graphisme

Embarqué

Environnement de bureau

Bureautique

Audio-vidéo

Administration réseau

News

Distribution

Programmation

Agenda-Interview

Sécurité

Matériel

Web

Jeux

Réfléchir

UnixGarden