JANVIER / FÉVRIER / MARS 2011
N°1

Open Silicium

MAGAZINE
INFORMATIQUE
OPEN SOURCE
EMBARQUÉ
ÉLECTRONIQUE

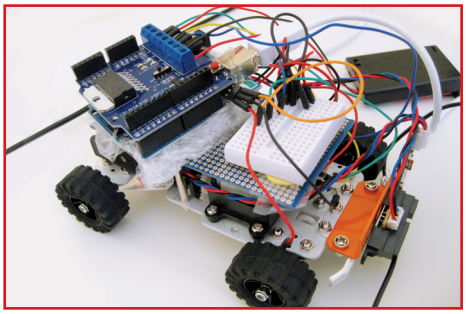
LE MAGAZINE DE L'OPEN SOURCE POUR L'ÉLECTRONIQUE & L'EMBARQUÉ

SPARC / NAS

Configurez, explorez et personnalisez le ReadyNAS Duo de Netgear p.92

SYST. CRITIQUES

Découvrez l'implémentation de systèmes critiques en pratique avec TASTE p.54



SIMULATION

Faites vos premiers pas avec SPICE, la simulation de circuits open source p.6

FPGS / PERFORMANCES

Comprenez l'impact des choix architecturaux dans vos développements Verilog et VHDL p.74

ARM9 / LCD TACTILE / LINUX

L'EMBARQUÉ DEVIENT ACCESSIBLE À TOUS !

Prise en main et évaluation de la carte Mini2440 de FriendlyARM p.38

ANDROID

Utilisez le SDK Android de Google sans Eclipse sous GNU/Linux, c'est possible ! p.21

DEBUG / QEMU

Déboguez vos applications ARM depuis Linux/x86 grâce QEMU et GDB p.30

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

L 18310 - 1 - F: 9,00 € - RD

Participez à l'événement 100% Embedded et Real-Time !

Les Salons Solutions ELECTRONIQUES

rts Le salon des solutions informatiques temps réel et des systèmes embarqués
The real-time solutions and embedded systems show

19ème édition

EMBEDDED SYSTEMS

29-30-31 mars 2011
Paris – Porte de Versailles

Pour exposer, visiter l'exposition ou s'inscrire aux conférences :
www.salons-solutions-electroniques.com

Salon strictement réservé aux professionnels.

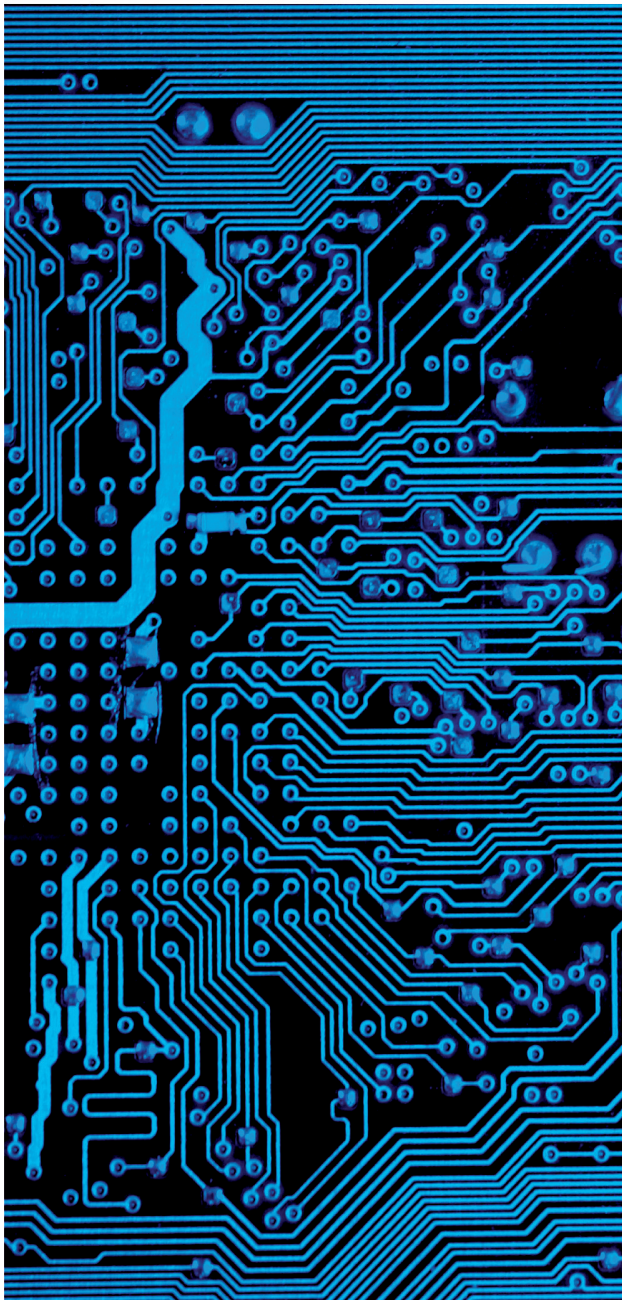
150 exposants attendus sur un même plateau :

Technologies & matériels

- Cartes et composants
- Modules
- OS temps réel et embarqués
- Conception
- Environnement de développement
- Outils de test et de validation

Conférences & tables rondes, ateliers

- Architectures multicoeurs
- FPGA
- Techniques de virtualisation
- Linux embarqué
- Modules processeurs
- Recherches appliquées en ingénierie logicielle
- Automobile
- Standards matériels
- Langages et environnements



Rts EMBEDDED SYSTEMS est organisé en parallèle à :

toMachine to Machine — display — ESDT ELECTRONIC SYSTEM DESIGN & TESTING

ORGANISATION :
97, rue du Cherche-Midi - 75006 Paris - France
Tél. : +33 (0)1 44 39 85 00 - Fax : +33 (0)1 44 39 85 36
e-mail : exposer@salons-solutions-electroniques.com

SOMMAIRE N°1/2011

Open Silicium Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

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

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

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

Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN** : en cours
Commission paritaire : en cours

Périodicité : Trimestriel
Prix de vente : 9 €

ÉDITO

Bienvenue !

Bienvenue dans ce tout premier numéro d'*Open Silicum*, la nouvelle publication des Éditions Diamond consacrée à l'électronique et l'embarqué open source. Ce magazine se veut être le miroir magique de l'évolution des technologies. Magique, parce qu'il a pour objectif d'analyser, de détailler et de mettre en pratique toutes les techniques utilisées dans un univers proche du matériel.

La philosophie qui nous motive est identique à celle de nos autres publications. Il s'agit d'une volonté de diffusion des connaissances et des expériences. L'open source a radicalement changé le monde de l'informatique, de la programmation et de l'administration système ces dernières années. De la même manière, aujourd'hui, il fait évoluer les habitudes, les stratégies et les choix des fabricants de matériels. Nous nous proposons ainsi de vous accompagner dans cette évolution en vous apportant, tous les trois mois, 96 pages d'informations concrètes sur des technologies « traditionnelles », modernes et/ou émergentes.

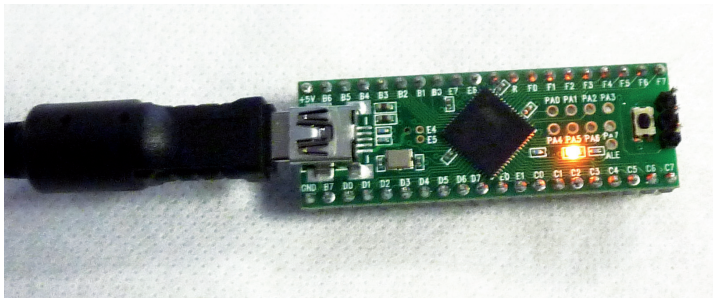
Ainsi, que vous soyez un professionnel de l'embarqué ou de l'électronique, un développeur GNU/Linux ou encore un passionné avide de comprendre les objets qui l'entourent et d'en faire pleinement usage, vous devriez trouver votre bonheur dans les pages qui suivent. Dans ce numéro, tout comme dans ceux qui suivront, nous avons cherché à repousser les limites, à explorer et à comprendre en détail les périphériques et les applicatifs que nous avions face à nous. Car ce n'est qu'à ce prix qu'il est possible d'exploiter au mieux une architecture, un système ou une plateforme. Il y a peu de temps, ceci se limitait au logiciel. Aujourd'hui, le matériel nous est enfin accessible et prêt à bénéficier, lui aussi, de l'effet open source.

À vous de choisir, soit vous prenez la pilule bleue et vous continuez à utiliser toutes ces choses, comme écrit dans le « User manual », soit vous choisissez la pilule rouge et vous vous enfoncez dans le terrier du lapin blanc, avec votre fer à souder et vos cross-compilateurs...

Je vous souhaite une bonne lecture ainsi que quelques nuit blanches. N'hésitez pas à nous faire part de vos commentaires, vos suggestions, vos remarques, vos propositions et vos souhaits en nous contactant à lecteurs@opensilicum.fr.

Denis Bodor

LA CARTE TEENSY 2.0 VICTIME DE SON SUCCÈS



Les cartes/modules Teensy 2.0 et Teensy++ 2.0 sont des platines d'expérimentation basées respectivement sur les microcontrôleurs Atmel AVR ATMEGA32U4 et AT90USB1286. Économiques, ils permettent de facilement appréhender la technologie USB, que cela soit du côté périphérique ou hôte (Teensy++ 2.0). On peut les considérer comme une excellente alternative à l'AT90USBKEY d'Atmel (AT90USB1287).

Malheureusement, l'annonce récente du code PSGroove permettant de faire fonctionner des programmes personnalisés (*homebrew*) sur PlayStation 3 via un microcontrôleur AT90USB* provoque un véritable raz de marée, tant au niveau des visites sur les sites de développeurs que par la création de sites de ventes d'accessoires pour PS3. L'excellent site de Mathieu Sonet, elasticsheep.com, en est un bel exemple, avec des visites ayant fait un bon fantastique en quelques jours.

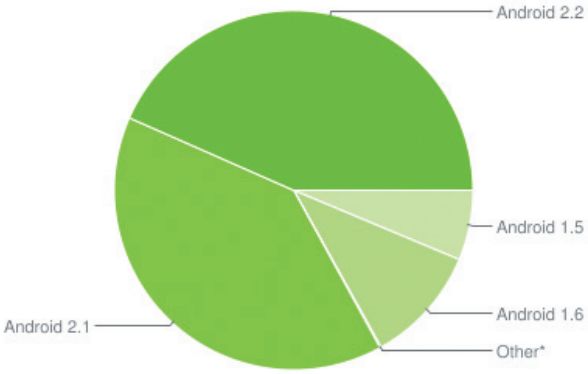
L'autre conséquence de l'effet PSGroove est la diffusion massive d'accessoires appelés AVR-Key ou Flip-Key. Vendus généralement deux fois le prix d'un module de développement pourtant plus complet, ce type de périphériques se présente sous la forme d'une simple clé USB sans E/S, destinée à être programmée par l'utilisateur (les tutoriels fleurissent) dans le seul but de procéder à la manipulation sur la console. PJRC, fabricant de cartes Teensy, signale par ailleurs la diffusion de clones n'hésitant pas à utiliser le même nom, mais intégrant un microcontrôleur dont des fonctionnalités sont bien inférieures.

Le grand gagnant dans l'histoire est bien entendu Atmel. Malheureusement, contrairement à ce qui se passe généralement lorsqu'une plateforme gagne en popularité, il n'y a ici presque aucun retour pour la communauté des développeurs. Moralité, pour qu'une émulsion prenne dans le monde de l'open source, mieux vaut éviter de populariser une plateforme ou un environnement par un effet d'annonce réducteur.

- PJRC, le site officiel des cartes Teensy : http://www.pjrc.com/teensy/
- PSGroove : http://github.com/psgroove/psgroove
- Atmel : http://www.atmel.com ■

ANDROID CUPCAKE ET DONUT SUR UNE VOIE DE GARAGE

En cette fin d'année, la progression des terminaux Android se connectant au Market Google affiche clairement une nette réduction de l'utilisation des versions 1.5 et 1.6 du système pour smartphone de Google. Android 2.1 et 2.2 affichent à eux deux plus de 80 % des connexions (respectivement 39 % et 43 %).



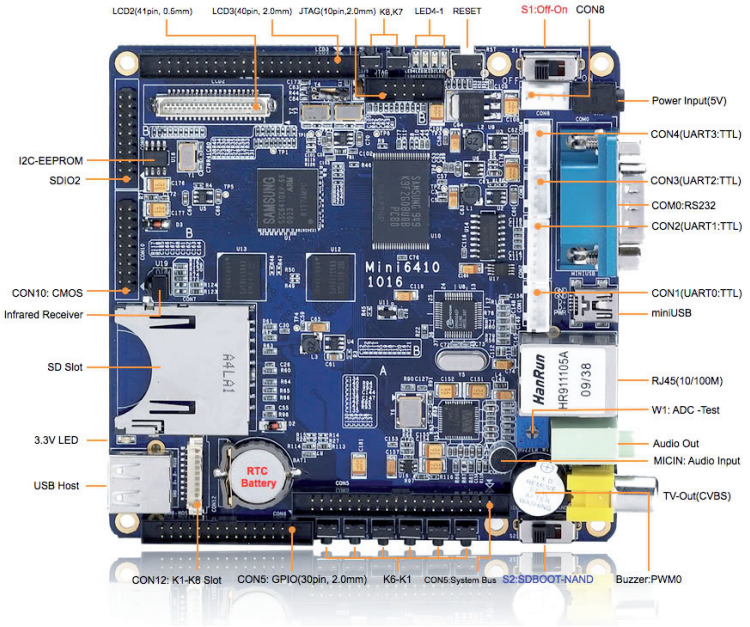
Il faut reconnaître que l'arrivée des HTC Desire HD & Desire Z sous Froyo couplée aux ventes de Samsung ne sont certainement pas étrangères à ce phénomène. Ce graphe, pourtant, n'est pas vraiment représentatif du parc Android étant donné l'absence des tablettes internet comme celles d'Archos dans les données collectées. Certaines de ces tablettes n'ont, en effet, pas d'accès au Market et ne sont donc pas comptabilisées. Le futur nous dira si l'arrivée de Gingerbread (Android 2.3), il y a quelques jours, risque de changer la donne, et ce, rapidement ou non.

Source : http://developer.android.com/resources/dashboard/platform-versions.html ■

UNE CARTE ARM11 POUR TOUTES LES BOURSES : FRIENDLYARM 6410

Si la carte FriendlyARM mini2440 décrite et testée dans ce numéro vous paraît un peu légère pour vos développements, vous serez heureux d'apprendre qu'une nouvelle version est disponible. Celle-ci, appelée mini6410, est une révision complète basée sur le processeur ARM11 S3C6410A. Disposant toujours de 128Mo à 1Go de Flash NAND et 128Mo ou 256Mo de RAM DDR, elle reprend les principales caractéristiques de sa petite sœur.

On notera cependant la disparition de la Flash NOR et l'ajout d'un récepteur infrarouge ainsi que d'un connecteur RCA pour une sortie vidéo composite (CVBS). L'écran tactile proposé en option est malheureusement toujours basé sur la technologie résistive. Côté logiciel, des images sont disponibles pour Windows CE 6, un Linux « maison », Android eclair et Ubuntu.

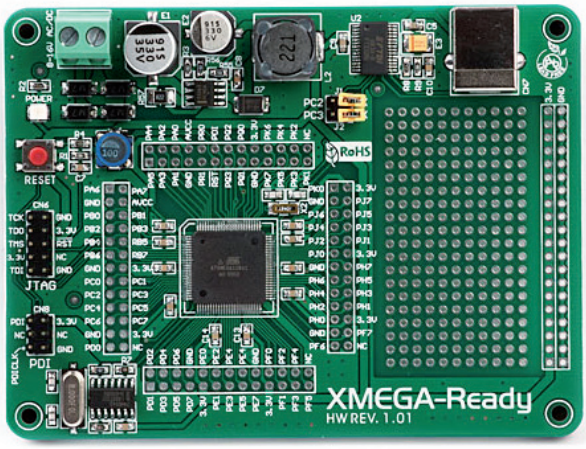


Le gain de cette évolution réside surtout dans le changement de processeur et l'ajout de 64Mo de RAM sur la version la plus petite. L'autre avantage est, bien entendu, le coût de la solution. Les principales cartes d'expérimentation basées sur le s3c6410 affichent généralement des tarifs de l'ordre de 500 euros ou plus. Ici, l'ensemble est disponible, directement de Chine, pour quelque 150 euros.

Un comparatif détaillé est fourni par le constructeur : http://www.friendlyarm.net/products/comparison ■

MIKROBOOTLOADER POUR XMEGA : LE CLIENT GNU/LINUX EST DISPONIBLE

Depuis le 30 novembre dernier, un client officiel pour GNU/Linux est disponible pour le *bootloader* MikroBootloader. Cette application packagée Debian (oui monsieur) permet de charger vos codes HEX directement sous l'OS open source. Rappelons que ce bootloader est dédié à la carte d'évaluation XMEGA-Ready fabriquée par MikroElektronika, incluant un microcontrôleur Atmel AVR ATxmega128A1.



La gamme Xmega est constituée de « gros » microcontrôleurs AVR 8bit incluant, entre autres choses, des convertisseurs A/N & N/A 12bit rapides et un moteur cryptographique AES et DES. La gamme Xmega, cependant, ne dispose pas de possibilité de programmation ISP (*In-System Programming*), mais uniquement d'interfaces PDI (*Programming and Debugging Interface*) et JTAG. Une alternative efficace consiste donc, à l'instar de ce qui est fait avec la plateforme Arduino, à utiliser un booloader chargé de programmer la Flash du microcontrôleur. C'est précisément ce que permet MikroBootloader. Bien entendu, il faut que le bootloader puisse être en mesure de communiquer avec la machine de développement pour faire son travail. Voilà l'objet du client maintenant disponible pour GNU/Linux. Regrettons simplement qu'il s'agit d'un code propriétaire, les sources ne sont donc pas disponibles, ce qui est le cas également de celui du booloader lui-même.

http://www.mikroe.com/eng/products/view/579/xmega-ready-board/ ■

INTRODUCTION À SPICE3 :

SIMULATION DE CIRCUITS ÉLECTRONIQUES, ET AU-DELÀ

par J-M Friedt

Toute conception de circuit électronique analogique qui n'est pas complètement triviale passe par une phase de simulation, ne serait-ce que pour identifier l'influence des divers blocs de référence entre eux, ou l'effet de l'incertitude de la valeur des composants sur la fonction de transfert du montage.

Nous nous proposons d'illustrer quelques exemples concrets de mise en œuvre d'un outil de simulation open source – SPICE – pour la simulation du comportement de circuits électroniques, avec une extension vers les simulations multiphysiques.

SPICE (*Simulation Program with Integrated Circuit Emphasis*) est un ensemble d'outils de simulation de circuits électroniques initialement développés à l'Université de Berkeley. Comme tout logiciel développé sur des fonds publics américains[1], il est librement disponible et est ainsi devenu la référence des outils pour simuler des circuits à basse fréquence. SPICE ne tient pas compte de l'agencement des composants ni du substrat diélectrique qui servira à la réalisation du circuit imprimé, mais ne considère qu'un ensemble de composants reliés entre eux : il est donc performant pour des fréquences allant du continu (DC) aux radio-fréquences. Les modèles de composants sont cependant complexes et tiennent compte d'effets aussi subtils que les variations de propriétés avec la température.

La stratégie d'utilisation de SPICE ne dépaysera pas l'utilisateur de GNU/Linux : après avoir tapé avec son éditeur favori la liste des composants et leur agencement en un circuit électronique (*netlist*), l'utilisateur appelle depuis une fenêtre de commandes interactive divers types de modélisations sur ce circuit. Les résultats de ces calculs sont disponibles soit immédiatement sous forme graphique, soit sous forme de tableaux de points redirigeables dans un fichier pour un traitement ultérieur – notamment avec des outils plus performants pour le traitement et l'affichage de données que les fonctions rudimentaires proposées par SPICE **[1]**. Alternativement, la netlist est traitée en mode batch depuis la ligne de commandes, permettant une interaction efficace avec le shell pour automatiser les simulations : dans ce mode, SPICE est capable de sauver l'intégralité des résultats de la simulation dans un fichier binaire (format `.raw`), dont les informations sont accessibles depuis GNU/Octave au moyen des scripts fournis à ngspice.sourceforge.net/osctavespice.html.

Ce document n'a en aucun cas vocation à se substituer à la documentation officielle **[2]** ou à la fonction `help` accessible depuis la ligne de commandes pour détailler la syntaxe et la fonctionnalité de chaque commande, mais plutôt d'introduire quelques concepts ludiques accessibles au moyen de cet outil de simulation.

1 Installation

La licence de SPICE n'est pas GNU puisque SPICE (années 1970) est antérieur à GNU (1989) et ce logiciel n'est donc disponible sous forme de paquet Debian que dans l'archive non *free*. De toute façon, il est toujours meilleur, par souci de cohérence avec son système, de recompiler depuis les sources : pour ce faire, nous récupérons la dernière archive disponible sur sourceforge à http://ngspice.sourceforge.net/. La compilation par le traditionnel trio `./configure && make && make install` nécessite de disposer des bibliothèques de développement `libxt-dev` et `libxaw-dev` (faute de quoi, SPICE compile sans la possibilité d'affichage graphique sous X11, et perd un peu de son interactivité sans pour autant perdre de fonctionnalité) **[4]**.

2 Premiers pas

Prenons l'exemple trivialement simple, mais d'un intérêt pratique indéniable, de la capacité de découplage que tout électronicien se doit de placer sur chaque alimentation des composants de son circuit. Une alimentation réelle (de résistance interne non nulle) débite dans

[1] Au même titre que les manuels d'enseignement de l'armée tels que ceux disponibles à http://www.tech-systems-labs.com/navy.htm pour une formation en électronique et électricité, ou http://metalworking.com/tutorial/ARMY-TC-9-524/9-524-index.html pour l'usinage de pièces mécaniques.

un composant qui effectue des appels de courant à une fréquence d'autant plus élevée que les signaux traités par ce circuit sont de fréquence élevée (le pire cas étant celui des composants numériques rapides qui doivent effectuer des transitions de plusieurs volts en quelques nanosecondes). Afin de réduire les fluctuations de tension d'alimentation le long des pistes d'alimentation, nous plaçons des condensateurs (dits de découplage) qui se comportent comme des réservoirs d'électrons qui absorbent les appels de courant. Comment choisir la valeur des condensateurs de découplage ?

Ces concepts se modélisent par le circuit de la figure 1, dans lequel la consommation du circuit est modélisée par la résistance de charge Rc et le condensateur de découplage est C.

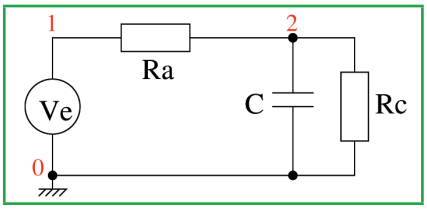


Figure 1 : Circuit servant d'exemple pour la première simulation sous SPICE

Ce circuit se décrit dans un fichier ASCII décrit dans le listing 1.

```
* exemple alim.cir
* toute ligne commencant par une etoile est
  un commentaire
* tout fichier doit commencer par au moins 1
  commentaire
Ra 1 2 10
Rc 2 0 500
C  2 0 500n
Ve 1 0 ac 1 dc 5

.control
ac dec 10 100 1meg
plot v(2)
alter C=500u
ac dec 10 100 1meg
plot v(2)
.endc
```

Listing 1 : Premier exemple d'un circuit de découplage RC fonctionnel sous ngspice

Nous constatons que le circuit est défini par des composants, dont la première lettre définit la nature (**R** pour les résistances, **C** pour les condensateurs, **L** pour les inductances,...). Les extrémités de ces composants sont définies par des nœuds : par convention, le nœud 0 définit la masse. Le premier nœud fournit la borne positive, le second la borne négative (cet ordre n'a évidemment de sens que pour les alimentations et les composants polarisés tels que les diodes **D**). Enfin, à chaque composant est affectée une valeur, avec les suffixes habituels (**n** pour 10^{-9}, **u** pour 10^{-6}, **m** pour 10^{-3}, **k** pour 10^{3} et, plus inhabituel, **meg** pour 10^{6}). La source de tension est notée **V** : cette source sera l'origine de la simulation, donc nous allons définir le comportement selon les modalités proposées ci-dessous.

SPICE permet d'analyser le comportement de ce circuit de diverses façons, dont les plus courantes sont :

- l'analyse du potentiel de sortie en fonction du potentiel constant (DC) d'une source de tension en entrée **dc**.
- l'analyse temporelle, et notamment du comportement lors de la mise en marche ou de la transition d'un signal, i.e. la réponse transitoire **tran**.
- l'analyse spectrale, ou la dépendance de la puissance radiofréquence en sortie connaissant la fréquence du signal en entrée **ac**.

Le fichier de configuration cité auparavant (Listing 1), tapé dans un fichier d'extension par convention **.cir** au moyen de son éditeur de texte favori (donc **vi**), est chargé dans **ngspice** par la commande **source fichier.cir**. Une fois la configuration chargée, nous définissons la nature de l'analyse en fournissant les propriétés de la source de tension **ve** : il peut s'agir d'une analyse **dc ve début fin incrément** pour rechercher le point de fonctionnement en tension continue alors que l'alimentation balaie la gamme de **début** à **fin** par pas d'**incrément**, d'une analyse en signaux alternatifs **ac dec nombre_points début fin** (**dec** signifiant que nous balayons les fréquences de **début** à **fin** Hz par décade, en vue d'une analyse dans un graphique log-log avec la tension en décibels), ou d'une analyse temporelle par pas de **pas** pour une durée de **durée** au moyen de **tran pas durée**. Ainsi, dans l'exemple précédent, l'exécution de la simulation s'obtient au moyen de :

```
jmfriedt@eee:~/$ ngspice
******
** ngspice-20 : Circuit level simulation program
** The U. C. Berkeley CAD Group
** Copyright 1985-1994, Regents of the University of California.
** Please submit bug-reports to: ngspice-bugs@lists.sourceforge.net
** Creation Date: Fri Nov 20 10:19:22 GMT 2009
******
ngspice 1 -> source alim.cir

Circuit: * exemple alim.cir

ngspice 2 -> ac dec 100 50 1meg
Doing analysis at TEMP = 27.000000 and TNOM = 27.000000

ngspice 3 -> plot v(2)
ngspice 4 -> plot vdb(1) vdb(2)
```

Les lignes en fin de l'exemple 1 commençant par **.control** et finissant par **.endc** contiennent la liste de commandes à exécuter par défaut au chargement de la netlist. Ainsi, dans cet exemple, la configuration du circuit elle-même commence avec la première ligne et s'achève avec la définition de la source **Ve**, tandis que les lignes qui suivent lancent la simulation pour une première configuration (**C**=500 nF) du comportement en petits signaux compris entre 10 Hz et 1 MHz, affichent le résultat (**plot v(2)**) ainsi que la source d'excitation d'amplitude constante (**plot v(1)**), puis nous reprenons cette simulation après avoir modifié (commande **alter**) la valeur de **C** pour qu'il prenne la valeur de 500 µF. L'alternative au traitement de commandes

fournies dans le fichier entre **.control** et **.endc** est de taper ces ordres dans la fenêtre interactive afin de rechercher les informations pertinentes caractérisant le circuit sans devoir recharger la netlist pour chaque nouvel essai.

Cet exemple permet de comprendre pourquoi un condensateur de quelques dizaines ou centaines de µF est nécessaire pour filtrer les composantes basses fréquences d'une alimentation connectée au secteur, tandis que les condensateurs de petites valeurs (quelques dizaines ou centaines de nF) éliminent les fluctuations à haute fréquence (Fig. 2). Ce graphique s'analyse de la façon suivante : il fournit l'amplitude du signal au nœud 2 (la borne de la charge Rc, qui en pratique serait le composant à alimenter) pour un signal au nœud 1 (la source de tension) oscillant à une fréquence variable, fournie en abscisse du graphique. L'objectif du découplage est de réduire au maximum ces fluctuations, donc de réduire au maximum les fluctuations au nœud 2 afin d'alimenter Rc par une tension continue présentant peu de fluctuations. Ainsi, notre objectif est d'identifier les conditions sur **C** pour minimiser la courbe de la figure 2. Pour ce faire, nos degrés de liberté sont la valeur de la capacité, en tenant compte de la technologie associée et des effets parasites qui sont associés : au-delà du microfarad, les condensateurs tantale (l'oxyde de tantale présente une permittivité très élevée et est utilisée en industrie des semi-conducteurs pour former des capacités élevées sur des surfaces réduites) et électrochimiques, en deçà du microfarad, le condensateur céramique.

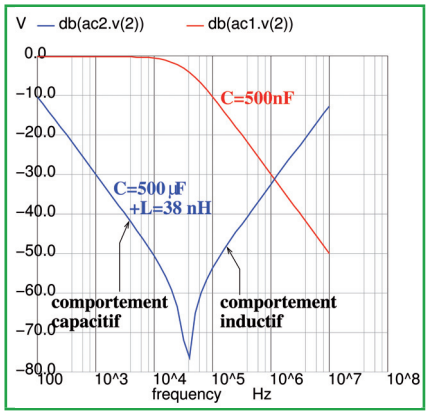


Figure 2 : Résultat de la simulation en petits signaux – AC – du modèle de condensateur de découplage dont le schéma est proposé en figure 1. En rouge, un condensateur de petite valeur, ici 500 nF, typiquement en technologie céramique, donc avec un comportement proche de l'idéal dans cette gamme de fréquences. Le modèle du condensateur électrolytique (500 µF, en bleu) est complété d'une inductance en série de 38 nH, en accord avec la valeur proposée dans une datasheet de fabricant.

En pratique, une capacité électrolytique présente une inductance en série **[6]** qui n'est pas négligeable aux fréquences élevées (au-delà de quelques kHz) : cette valeur est documentée par les fabricants **[7]** : dans l'exemple proposé sur la figure 2, nous avons placé en série avec la capacité de 500 µF une inductance de 38 nH. Dans ces conditions, l'inductance la plus élevée est bien la plus efficace à empêcher un signal basse fréquence d'atteindre le nœud 2 (filtrage efficace de la composante basse fréquence), mais est incapable d'efficacement couper un signal de fréquence supérieure au MHz. Au contraire, le condensateur céramique (technologie différente, ne présentant pas de composante inductive significative dans cette gamme de fréquences), coupe efficacement au-delà de 1 MHz. Cette démonstration justifie la mise en parallèle de ces deux types de composants pour un découplage efficace des composants numériques en vue de rendre leur alimentation immunisée aux fluctuations basses fréquences (condensateur de valeur élevée, électrolytique) et hautes fréquences (condensateur céramique de valeur de l'ordre du nF).

La source peut être plus complexe qu'un simple générateur DC ou AC : nous sommes capables de définir des sinusoïdes ou des impulsions pour exciter le circuit, et ainsi obtenir la réponse temporelle par **tran** à ces sollicitations. Si dans cet exemple, nous remplaçons la définition de la source **Ve** par **Ve 1 0 dc 1 pulse(1 2 10u 0 0 20u 50u)**, alors nous générons une série d'impulsions commençant 10 µs après le début de la simulation, de 20 µs de long et se répétant toutes les 50 µs, avec des temps de montée et de descente de 0 s. De la même façon, **Ve 1 0 dc 1 sin(1 1 30k)** définit une source de tension moyenne de 1 V, d'amplitude 1 V et de fréquence 30 kHz. Le résultat de ces simulations en transitoire (i.e. comportement temporel) pendant 100 µs est illustré sur la figure 3.

Lors d'une série de simulations, SPICE recalcule l'ensemble des paramètres du circuit mais mémorise les anciens résultats. Le choix de la simulation à exploiter lors de l'affichage s'obtient par **setplot**. La liste des variables au sein d'une simulation s'obtient par **display all**. Au lieu de passer d'une simulation à l'autre par **setplot**, il est possible de fournir en préfixe à la variable affichée la simulation qui contient cette variable : dans l'exemple précédent,

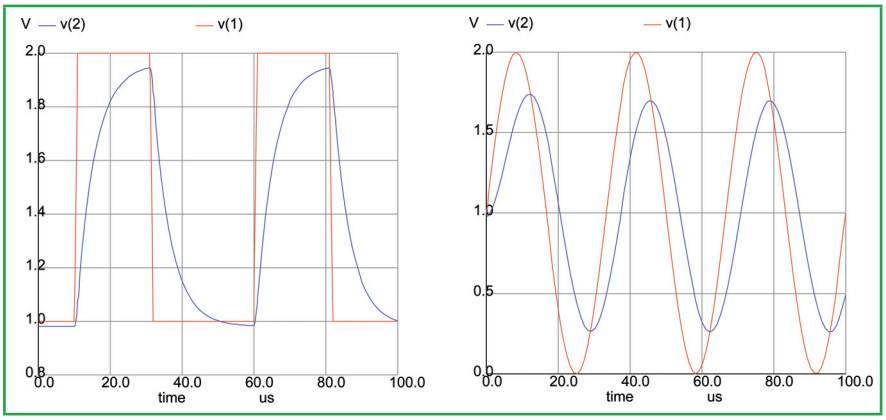


Figure 3 : Gauche : alimentation du circuit de découplage RC série par un créneau (en rouge, la tension d'excitation, et en bleu, la sortie du filtre). Droite : alimentation du circuit de découplage RC série par une sinusoïde (en rouge, la tension d'excitation, et en bleu, la sortie du filtre).

plot ac1.v(2) ac2.v(2) superpose sur un même graphique les simulations obtenues pour les deux valeurs de condensateurs 500 µF et 500 nF.

L'ensemble des paramètres d'un composant s'obtient par **show**, qui prend en argument le nom du composant, tandis qu'une simulation est détruite au moyen de **destroy**. Ces deux commandes acceptent comme argument **all**.

Afin d'inclure le résultat de simulations dans un rapport de bonne qualité graphique, il est naturel de vouloir exporter les schémas représentatifs des simulations au format Postscript. SPICE doit être configuré pour exporter ses graphiques dans ce format : **set hcopydevtype=postscript** passe le périphérique de sortie en mode Postscript, **set hcopypscolor=true** permet d'afficher chaque courbe en couleur au lieu de tons de gris, **set color0=rgb:f/f/f** et **set color1=rgb:0/0/0** définissent la couleur de fond à blanc et les caractères en noir respectivement. Une fois ces configurations mémorisées, l'affichage s'obtient comme dans le cas de **plot**, mais en remplaçant cette commande par **hardcopy fichier.ps tensions** (Figs. 3 et 4).

Nous ne sommes évidemment pas restreints à l'exploitation de composants linéaires : l'exemple de la figure 4 illustre dans un premier temps la caractérisation d'une diode avec ses paramètres par défaut – avec notamment la tension de seuil classique à 0,6 V pour les diodes silicium – puis l'association des diodes dans un circuit redresseur tel qu'utilisé classiquement dans les convertisseurs AC-DC ou DC-DC. On notera dans ce dernier cas la visualisation du transitoire, avec la tension de sortie du régulateur (en orange) qui prend environ 7 périodes (soit 0,7 ms) pour atteindre un régime stationnaire, tandis que la sortie du pont redresseur (en bleu) illustre bien l'exploitation des deux alternances pour charger le condensateur en sortie.

L'option **.model** permet de modifier les paramètres par défaut exploités par les modèles de SPICE. À titre d'exemple pratique, la tension de jonction de diode telle que nous venons de le voir est de 0,6 V, en accord avec le comportement d'une diode silicium. Il se peut cependant que nous voulions varier cette tension de jonction, que ce soit pour avoir une diode sans seuil (telle que nous l'obtiendrions en plaçant une vraie diode dans la boucle de rétroaction négative d'un amplificateur opérationnel), ou pour élever ce seuil à 1,2 voire 1,5 V, tel qu'observé expérimentalement pour une LED. Le modèle complexe de la diode, décrit dans la documentation de **ngspice** **[2, chap.8, p.70]**, ne permet pas de simplement définir la tension de jonction, mais nécessite de passer par le coefficient d'émission N avec lequel elle est proportionnelle. Ainsi, nous utiliserons :

```
.model nom D(N=0.01)
```

pour modifier les paramètres par défaut de la diode D, en remplaçant le paramètre N de 1 à 0,01, et ainsi générer le modèle **nom** qui sera appelé par **D1 noeud1 noeud2 nom** pour une diode de seuil négligeable (une valeur de 0 ne permet pas la convergence d'une solution). De la même façon, un paramètre de N>2 donnera un modèle pertinent pour une LED. Nous encourageons le lecteur à retracer les schémas de la figure 4 avec ce paramètre.

3 Exemples concrets : électronique

Les exemples qui vont suivre ne sont pas pris au hasard, mais on été piochés dans une collection de circuits développés par l'auteur lors de diverses activités plus ludiques que professionnelles.

3.1 Caractérisation de circuits passifs

Un circuit nécessite une source et une charge : en connectant un circuit passif entre une source de tension (d'impédance nulle) et une charge élevée, nous pouvons caractériser le comportement d'un réseau

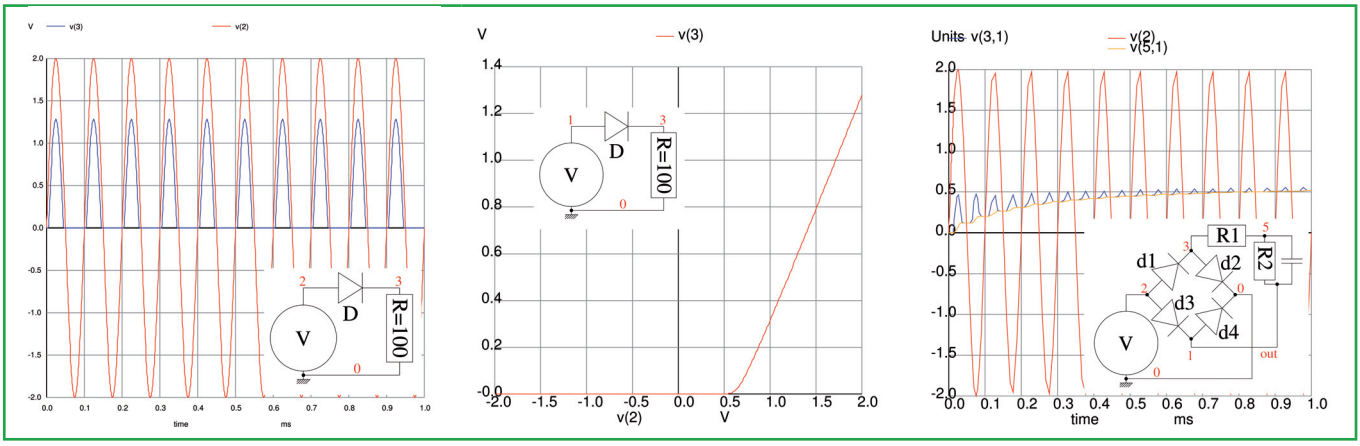
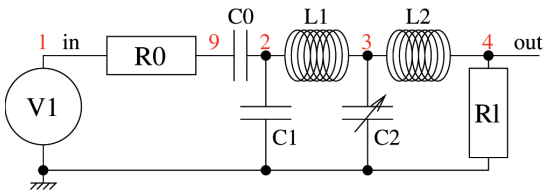


Figure 4 : Gauche : caractérisation de la diode par sa réponse temporelle lorsque excitée par une sinusoïde, et au milieu, la caractérisation par une tension en entrée variant de -2 à +2 V, illustrant la tension de seuil de 0,6 V qui se traduit par cette chute de tension lors du redressement de la sinusoïde. Droite : exploitation de cette diode dans un pont redresseur chargé par un circuit RC de lissage représentatif d'une charge. La tension de seuil est notamment source d'inefficacité dans les applications RFID ou convertisseurs DC-DC qui nécessitent une tension d'entrée au-dessus de ce seuil pour fonctionner.

de composants passifs et la dépendance de son impédance avec les valeurs d'un de ses éléments. Le résultat d'une simulation peut être redirigé vers un fichier comme nous le ferions classiquement dans un shell Unix : **`print vdb(3) > nom_fichier`**, permettant de facilement explorer l'espace des paramètres dont dépend la fonction de transfert[2].

Afin d'illustrer l'analyse paramétrique d'un filtre passe-bas radiofréquence (filtre passe-bas de Chebychev à 4 pôles visant à éliminer les harmoniques d'un oscillateur numérique conçu pour fonctionner à 10 MHz) **[3, p.17.98]**, nous proposons l'exemple qui suit, qui en plus d'illustrer une nouvelle netlist, fournit un exemple de script de commandes SPICE afin de balayer une plage de valeurs d'un composant :

```
* low-pass filter: cutoff @ 20 MHz (p.17.98 ARRL Handbook)
.control
destroy all
set units = degrees
foreach parameter 0.2 0.5 1 2
   set c=$parameter
   echo $c $parameter*1n
   alter @c2[capacitance]=$parameter*1n
   ac dec 50 1meg 200meg
end
* unset parameter
plotdb(ac1.v(4))db(ac2.v(4))db(ac3.v(4))db(ac4.v(4))xlabelf[Hz]ylabelmagnitude[dB]title'ACanalyses'
**plotp(ac1.(4))p(ac2.(4))p(ac3.(4))p(ac4.(4))xlabelf[Hz]ylabelphase[degrees]title'ACanalyses'
unset units
* destroy all
.endc

v1 1 0 dc 0 ac 1 pulse 0 -1 1ms
* R0=impedance d'entree, R1=sortie
R0 1 9 50
C0 9 2 0.1u
L1 2 3 134n
C1 3 0 680p
L2 3 4 433n
C2 4 0 220p
R1 4 0 50
.end
```



L'exemple ci-dessus – sous forme de netlist et de schéma associé – est l'opportunité de mentionner que SPICE implémente un langage scripté pour contrôler l'exécution de la simulation dans le bloc commençant par **`.control`** et finissant par **`.endc`**. Ce langage supporte les notions de variable, de listes et de contrôle du flux d'exécution.

Ainsi, la séquence

```
foreach parameter 0.2 0.5 1 2
   set c=$parameter
   echo $c $parameter*1n
end
```

affecte à la variable **`c`** le contenu courant de **`parameter`** pioché dans la liste définie par **`foreach`**, et affiche cette valeur ainsi que la conversion de **`parameter`** en un format compatible avec une valeur de composant. Nous avons déjà vu auparavant que l'instruction **`alter`** permet de modifier la valeur d'un composant défini dans le fichier source **`.cir`**. Finalement, l'ajout de l'ordre de simulation **`tran`** ou, ici, **`ac`**, au sein de cette boucle, implique une séquence de 4 simulations pour 4 valeurs du paramètre **`c2`**. Une fois ces simulations achevées, nous affichons sur un unique graphique le diagramme de Bode obtenu pour chaque valeur du paramètre (Fig. 5). Nous avions déjà vu auparavant un cas particulier d'affectation de variable du système lors de la définition de **`hcopydevtype`** pour obtenir un affichage en Postscript : l'affectation s'obtient par **`set`**, et l'appel au contenu de la variable nécessite d'en précéder le nom par le symbole **`$`**. Toute combinaison de variable avec les constantes prédéfinies du système (affichables par **`print const.all`**) ou les unités (dans cet exemple, **`1n`**) est possible lors d'une affectation.

Le script commence par effacer toute simulation en mémoire (**`destroy all`**) afin de savoir quelle valeur de **`C2`** a été affectée pour quelle simulation (la séquence des simulations **`ac1`**, **`ac2`**, **`ac3`**, etc., s'incrémentant automatiquement par rapport aux résultats déjà en mémoire au moment de l'exécution du script). Puis nous cyclons les diverses valeurs de paramètres, de 0,2 à 2 nF, que nous affectons à **`C2`** dans la boucle, avant d'afficher le résultat par l'instruction **`plot`** sur un unique graphique de toutes ces simulations. Notez l'exploitation de la syntaxe **`db(ac1.v(4))`** pour afficher le potentiel au point 4 de la simulation **`ac1`** en décibels : la notation **`ac1.vdb(4)`** n'est en effet pas reconnue. La même opération visant à afficher le déphasage induit par le filtre au lieu du gain s'obtient en remplaçant **`db`** par **`p`**.

[2] Attention, dans une version antérieure de ngspice – version 17 notamment – la redirection depuis l'interpréteur ngspice se traduisait par une erreur de segment, erreur qui a été corrigée au moins dans la version 20 de ngspice disponible au moment de la rédaction de ce document.

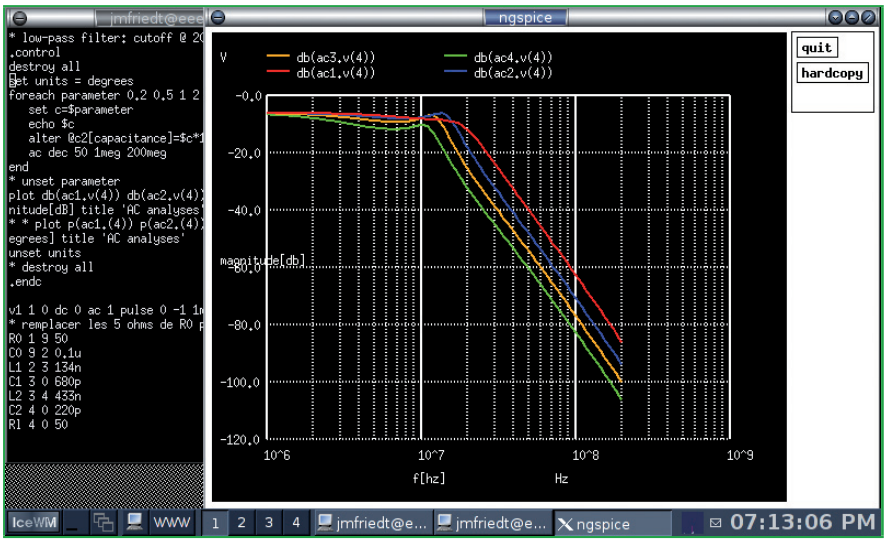


Figure 5 : Capture d'écran de la simulation d'un filtre de Chebychev, avec évolution de la fonction de transfert en fonction de la valeur d'un des condensateurs.

3.2 Filtrage idéal

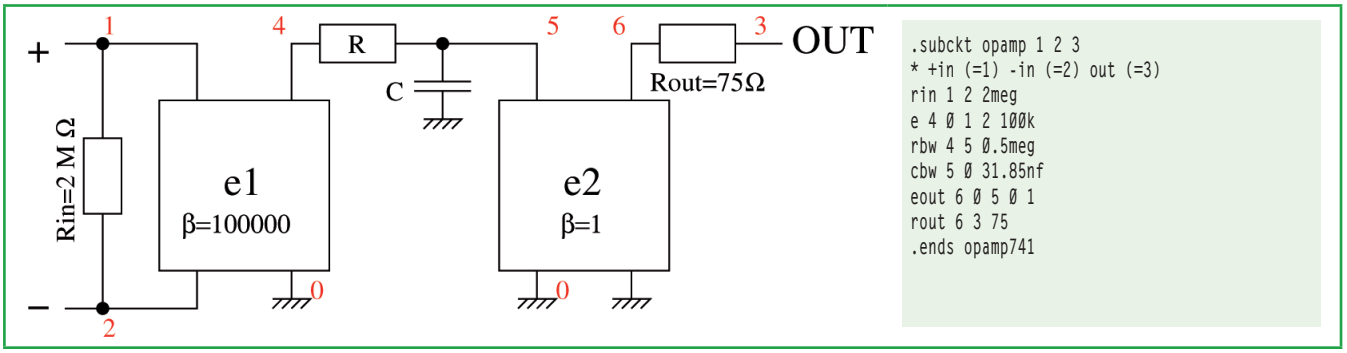
Dans le cas d'un filtrage, le signal électrique ne traverse qu'une fois la chaîne de traitement, entre son entrée et sa sortie, et le comportement du circuit est par conséquent faiblement dépendant des propriétés des composants utilisés (contrairement au cas de l'oscillateur que nous verrons plus loin (section 3.4), qui boucle la sortie de l'amplificateur sur son entrée et réinjecte donc toute erreur du modèle qui se voit amplifiée pendant la durée de la simulation). Ainsi, un modèle idéal d'amplificateur opérationnel suffira en général à décrire convenablement le comportement d'un filtre allant à des fréquences jusqu'au mégahertz. Le rôle de l'amplificateur opérationnel est de rendre le comportement des composants insérés dans un filtre aussi proche de l'idéal que possible. Ainsi, un redresseur sans seuil – éliminant la tension de jonction d'une diode – s'obtient en remplaçant une simple diode par une diode en rétroaction négative sur un amplificateur opérationnel (l'amplificateur opérationnel s'efforce d'ajuster la sortie à l'entrée, et de cette façon somme naturellement la tension de jonction de la diode à la sortie pour fournir un redresseur sans seuil). De façon générale, l'amplificateur opérationnel a pour rôle de compenser les pertes dans les composants contenus dans le filtre.

Nous savons qu'un amplificateur se comporte comme une source de tension contrôlée en tension de gain classiquement noté β : SPICE nous fournit un tel composant sous la nomenclature **e** qui prend comme arguments les bornes positives et négatives de la source, les bornes de la commande et le gain. Pour ajouter un minimum de réalisme au modèle, nous ajoutons la contrainte d'une impédance grande mais finie aux bornes inverseuse et non inverseuse de l'amplificateur (**Rin**), ainsi que la dépendance du gain de l'amplificateur opérationnel en boucle ouverte avec la fréquence de fonctionnement (produit gain-bande passante constant au travers du filtre **RC**) qui permet de garantir la stabilité du circuit, et l'empêche d'osciller à haute fréquence (Fig. 6). Ce modèle servira de base pour les analyses de circuits plus complexes exploitant l'amplificateur opérationnel en fonctionnement linéaire comme bloc de base.

SPICE propose – au même type que les procédures dans les langages de programmation – de découper un circuit complexe en blocs de base individuels nommés sous-circuits (*subcircuits*). La syntaxe de cette fonction est :

```
.subckt nom noeuds
```

avec **noeuds**, l'équivalent des arguments de la procédure, ici la liste des nœuds permettant de connecter le sous-circuit au circuit principal. Un bloc de sous-circuit se termine par **.ends**. Le sous-circuit




```
.subckt opamp 1 2 3
* +in (=1) -in (=2) out (=3)
rin 1 2 2meg
e 4 0 1 2 100k
rbw 4 5 0.5meg
cbw 5 0 31.85nf
eout 6 0 5 0 1
rout 6 3 75
.ends opamp741
```

Figure 6 : Modèle d'amplificateur opérationnel quasi-idéal tenant compte des impédances finies d'entrée (Rin grand) et de sortie (Rout petit) ainsi que de la décroissance du gain avec la fréquence de fonctionnement (filtre passe-bas RC entre les deux sources de tension contrôlées en tension e1 et e2). Les entrées notées « + » et « - » sont les entrées non inverseuse et inverseuse respectivement, OUT est la sortie de l'amplificateur. Ce modèle ne tient pas compte des tensions d'alimentation et ne peut donc pas présenter les limitations de l'amplificateur opérationnel associées au comportement saturé.

de nom **nom** s'appellera dans la netlist principale par un identifiant commençant par **x**. Ainsi, dans cet exemple, si le circuit de la figure 6 fournit le schéma de principe d'un amplificateur idéal que nous nommerons **opamp741** dans une procédure prenant pour argument les entrées non inverseuse, inverseuse et la sortie

```
.subckt opamp741 1 2 3
```

alors tout circuit nécessitant un amplificateur opérationnel fera appel à ce composant par

```
x1 10 11 12 opamp741.
```

Nous pouvons nous convaincre du peu d'influence du modèle d'amplificateur opérationnel dans la conception de filtres fonctionnant aux fréquences audio (< 1 MHz) par le modèle de filtre passe-bande proposé dans le listing 2, qui exploite le modèle simple proposé ci-dessus et un modèle complexe proposé dans **[8]**) (Fig. 7).

3.3 Oscillateur : injection d'énergie en phase et synthèse de facteur de qualité

L'exemple qui suit a pour vocation d'illustrer l'assemblage de circuits simples – validés individuellement lors de simulations n'incluant que l'unique bloc dont on vérifie le comportement – afin de fournir une fonctionnalité complexe. Le listing 3 illustre ce découpage en exploitant explicitement 4 amplificateurs opérationnels, qui forment chacun le cœur d'une fonction de base : convertisseur courant-tension, déphaseur, amplificateur et sommateur. Cette netlist reprend donc le concept de sous-circuit, avec notamment l'amplificateur opérationnel que nous venons de proposer, pour en déduire des blocs aux fonctionnalités plus complexes. Chaque bloc prend en arguments autant de nœuds qu'il a d'entrées et de sorties : une entrée et une sortie pour un déphaseur, deux entrées et une sortie pour un additionneur-inverseur.

```
*band pass filter

* set options
.opt nopage
.width in=72
.width out=80

.subckt opamp741 1 2 3
* +in (=1) -in (=2) out (=3)
rin 1 2 2meg
rout 6 3 75
e 4 0 1 2 100k
rbw 4 5 0.5meg
cbw 5 0 31.85nf
eout 6 0 5 0 1
.ends opamp741

.subckt AD712 13 15 12 16 14
VOS 15 8 DC 0
EC 9 0 14 0 1
C1 6 7 .5p
RP 16 12 12k
GB 11 0 3 0 1.67k
RD1 6 16 16k
RD2 7 16 16k
ISS 12 1 DC 100u
CCI 3 11 150p
GCM 0 3 0 1 1.76n
GA 3 0 7 6 2.3M
RE 1 0 2.5MEG
RGM 3 0 1.69k
VC 12 2 DC 2.8
VE 10 16 DC 2.8
R01 11 14 25
CE 1 0 2p
R02 0 11 30
RS1 1 4 5.77k
RS2 1 5 5.77k
J1 6 13 4 FET
J2 7 8 5 FET
DC 14 2 DIODE
DE 10 14 DIODE
DP 16 12 DIODE
D1 9 11 DIODE
D2 11 9 DIODE
IOS 15 13 5e-12
.MODEL DIODE D
.MODEL FET PJF(VTO=-1 BETA=1M IS=25e-12)
.ENDS

* band pass filter
.subckt BPF 1 6
R1   1 3 22.0k
C1   3 2 560p
R2   3 4 22k
C2   4 0 0.1n
C3   2 5 220n
R3   5 6 820
xao1 4 2 2 opamp741
* xao1 4 2 100 101 2 AD712
.ends BPF

* current and voltage sources
vin 1 0 dc 0 ac 1 sin 0 100u 2.247kHz
vcc 100 0 dc 12
vss 101 0 dc -12

* circuit elements
xc2 1 2 BPF
Rout 2 0 10k

* analysis control 50 Hz-50kHz with 5 points/decade, tran step total
* either dec num start stop ; or lin points start stop
.control
ac dec 100 1 100k
* .tran 1u 10m
* .plot vdb(2)
* .plot vp(2)
plot vdb(2) vp(2)
.endc
```

Linsting 2 : Programme de simulation SPICE pour simuler le comportement d'un filtre passe-bande, soit avec un modèle simple d'amplificateur opérationnel (opamp741) tel que proposé sur la figure 6, soit avec un modèle plus complexe (AD712) tel que proposé dans [8].

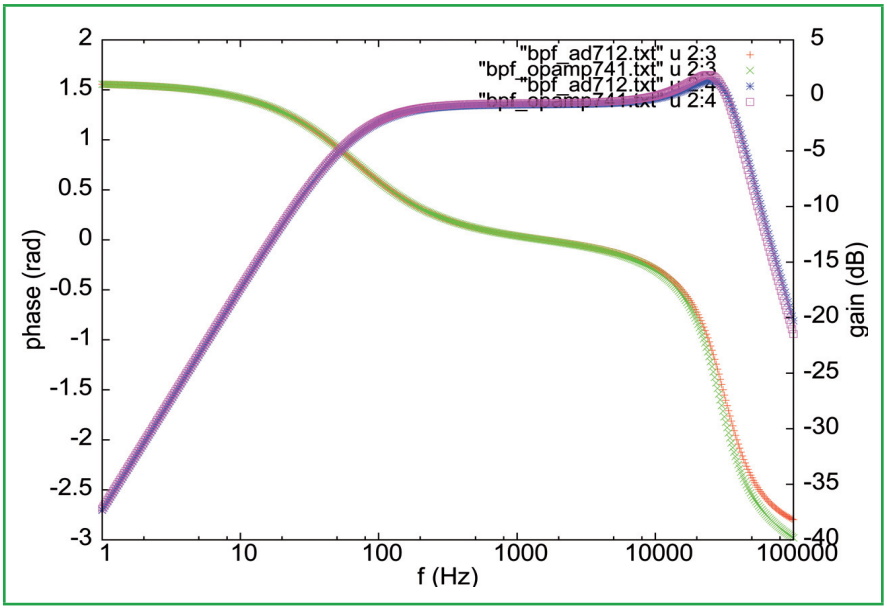


Figure 7 : Résultat de l'exécution du script 2, pour deux modèles d'amplificateurs opérationnels : le circuit se comporte bien comme un filtre passe-bande, et la seule différence significative entre les modèles apparaît aux fréquences les plus élevées.

Nous proposons une approche des oscillateurs qui passe par l'injection en phase d'un signal afin de compenser les pertes induites par un élément dont la fonction de transfert présente un faible encombrement spectral (résonateur, sous-circuit **quartz** dans la netlist 3) **[5]**. Un amplificateur est un dispositif large bande, capable d'amplifier un signal quelle que soit sa fréquence d'entrée, avec un gain d'autant plus faible que la fréquence est élevée. Un oscillateur est en général conçu pour fonctionner à une fréquence bien précise, définie par un élément passif d'encombrement spectral réduit dont le rôle est de tuer toute ambition de l'amplificateur d'entretenir une oscillation à une fréquence autre qu'à celle du résonateur (pertes excessives qui ne peuvent être compensées par l'amplificateur pour les fréquences en dehors de la bande passante du résonateur). L'exemple ci-dessous permet d'illustrer la conception d'un circuit électronique sous forme d'assemblage de blocs de base dont chacun est convenablement caractérisé avant d'être inclus dans la chaîne de traitement, et ce en vue de présenter l'oscillateur comme condition asymptotique de l'injection d'énergie en phase dans un résonateur pour en augmenter le facteur de qualité apparent.

Dans son principe (Listing 3), la sortie d'un amplificateur (**x1**) est injectée dans un résonateur (**x2**). La sortie de ce résonateur, un dipôle qui ne laisse passer du courant que si la tension en entrée respecte une condition de synchronisme avec le dispositif mécanique qui sélectionne la fréquence (comme le ferait un pendule), est convertie en tension au moyen d'un convertisseur courant-tension idéal (du fait de la présence d'un amplificateur opérationnel qui isole la résistance de charge de l'impédance du circuit suivant : **x2b**). Ce signal est déphasé (**x3**), amplifié (**x4**) et sommé (boucle fermée sur **x1**) au signal initial. Si le déphasage (constant) a amené le signal additionné en phase avec le signal original, il y a amplification. Si le signal déphasé est en opposition de phase avec le signal d'origine, il y a annulation ou tout au moins réduction de la puissance réinjectée dans le résonateur. Un choix judicieux de la phase – ajustée en faisant varier les valeurs des composants du déphaseur au moyen de SPICE – permet soit d'étaler l'encombrement spectral équivalent du résonateur[3] (réduction du facteur de qualité apparent), soit de le réduire (augmentation du facteur de qualité apparent) (Fig. 8).

```
* quartz tuning for Q tuning
.options nopage
.width in=72
.width out=80

* current and voltage sources
vin input 0 0V ac 1 sin 0 1m 2.247kHz

x1 input 1 2 adder
x2  2 3 quartz
x2b 3 9 iv
x3  9 4 dphi90min
x4  4 1 amplini

ro 9 0     1000000

* circuit elements:
* Q=2*pi*fs*L1/R1, fs=1/(2*pi*sqrt(L1*C1))
*                  ie L1=1/(4*pi^2*fs^2*C1)
.subckt quartz 1 2
c0  2 1 1.7p
r1  3 1 180k
l1  4 3 6740
c1  4 2 3.5f
.ends quartz

* Op amp: non-inverting; inverting; output
.subckt oa3 2 3 1
Rin 2 3 2e6
E1 4 0 2 3 2e5
RQ 4 5 1000
CQ 5 0 2.12e-5
E2 6 0 5 0 1
Rout 6 1 75
.ends oa3

* I->V converter (V=-RI)
.subckt iv 1 2
R    1 2 100k
xao1 0 1 2 oa3
.ends iv

* adder-inverter
.subckt adder 1 2 3
R1   1 4 10k
R2   2 4 10k
R3   4 3 10k
xao1 0 4 3 oa3
.ends adder

* non inverting amplifier: G=1+R2/R1
.subckt amplini 1 2
xao1 1 3 2 oa3
R2   3 2 1.6k
R1   3 0 1k
.ends amplini

* dphi filter: out=in*(1-j2pi*f*RC)/(1+j2pi*f*RC),
* phi=-2atan(2pifRC)            Rf=R1 -- changer C
.subckt dphi90plus 1 2

R1   1 3 10k
R    0 4 1.0k
Rf   3 2 10k
C    1 4 10n
xao1 4 3 2 oa3
.ends dphi90plus

* dphi filter: out=in*(1-j2pi*f*RC)/(1+j2pi*f*RC),
* phi=-2atan(2pifRC)            Rf=R1 -- changer C
.subckt dphi90min 1 2
R1   1 3 10k
R    1 4 20.0k
Rf   3 2 10k
C    0 4 0.100n
xao1 4 3 2 oa3
.ends dphi90min

.ac lin 5000 32.7k 32.9k
.print ac v(9)
.op
.end
```

Listing 3 : Programme de simulation SPICE pour générer les résultats de la figure 8, dont les variables sont le signal du déphasage (+90 ou -90° par dphi90min ou dphi90plus) du signal injecté dans la seconde branche de l'additionneur, et le gain de la boucle de rétroaction (amplificateur non inverseur amplini).

Un tel circuit est exploité en pratique lorsque l'encombrement spectral d'un résonateur utilisé comme capteur (gyroscope, accéléromètre) doit être réduit pour permettre une identification fine de la fréquence de résonance (cas de l'augmentation du facteur de qualité), ou lorsque le temps de réponse du résonateur – inversement proportionnel au facteur de qualité – à une sollicitation extérieure doit être amélioré (cas des résonateurs exploités en microscopie à sonde locale : réduction du facteur de qualité). Sa plage fréquentielle de fonctionnement est limitée par la bande passante des amplificateurs opérationnels : en pratique, nous utilisons ce circuit dans la bande 1-100 kHz, qui inclut notamment les diapasons à quartz horlogers.

[3] Le facteur de qualité est par définition une qualité intrinsèque au résonateur, égal au ratio de l'énergie emmagasinée par l'énergie dissipée à chaque période. Nous parlons ici de facteur de qualité équivalent car il s'agit de pertes en partie compensées ou augmentées par le circuit de réinjection d'énergie – en phase ou en opposition de phase.

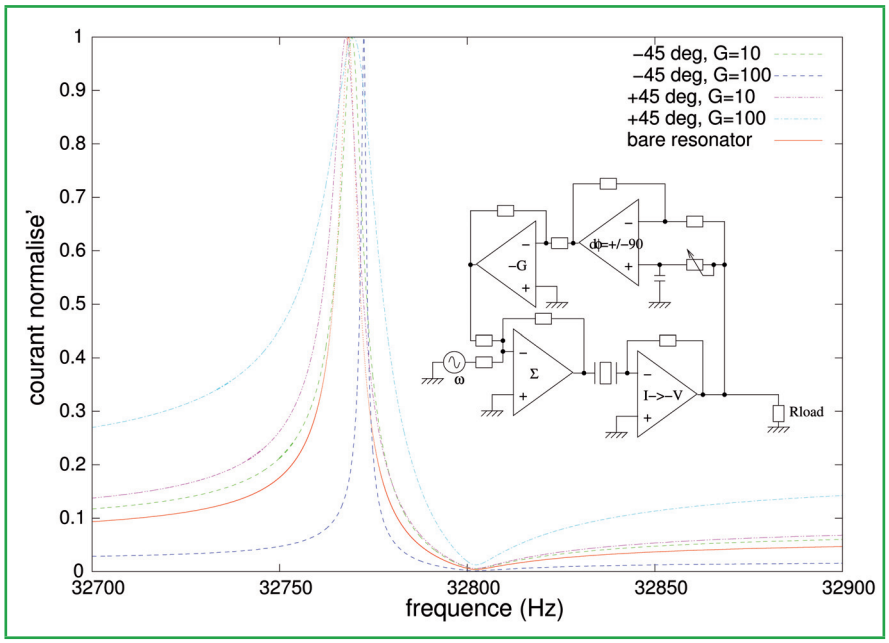


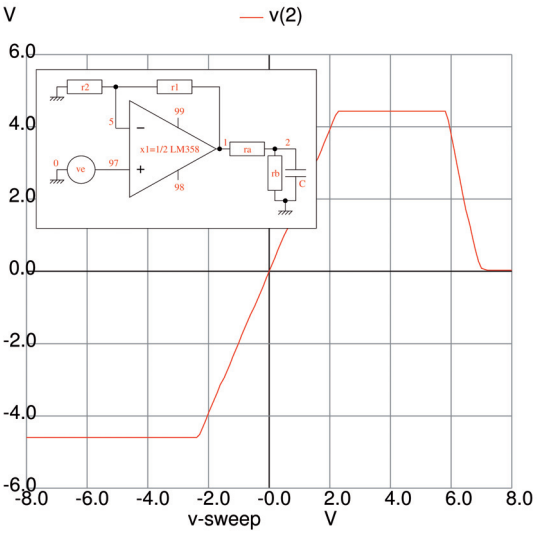
Figure 8 : Résultat de la modélisation par le programme proposé en listing 3, et schéma bloc associé. On constate bien que pour un déphasage de -45°, le facteur de qualité est réduit, et +45° augmente le facteur de qualité. Cet effet est d'autant plus marqué que le gain sur la rétroaction est important. Le cas asymptotique de pertes compensées par le gain, dans le cas d'un déphasage de +45°, qui induit une oscillation à la fréquence propre du résonateur indépendamment de tout signal injecté, n'est pas illustré ici.

Le cas des oscillateurs est plus complexe que celui des filtres, car par conception, le signal en sortie de l'amplificateur est réinjecté en entrée, et l'oscillation est entretenue si les pertes dans l'élément filtrant la fréquence de fonctionnement (résonateur) sont compensées par le gain de l'amplificateur (une des conditions dites de Barkhausen). En l'absence de point de compression, i.e. une amplitude du signal en entrée à partir de laquelle le gain de l'amplificateur chute, la rétroaction positive fait diverger la tension dans la boucle vers l'infini. Ce cas n'est pas réaliste, et un modèle plus précis du comportement de l'amplificateur est nécessaire, limitant son gain à partir d'une certaine puissance de sortie. La façon la plus grossière d'atteindre ce résultat est de définir une tension de saturation à partir de laquelle quelle que soit la tension d'entrée, la sortie reste constante (en pratique égale à la tension d'alimentation).

Une source de modèles de composants – et notamment d'amplificateurs opérationnels réalistes – est le site web de National Semiconductor à www.national.com/analog/amplifiers/spice_models. Les fichiers de modèles **.mod** se chargent au moyen de **.include fichier.mod**. De la même façon, Maxim/Dallas donne accès aux modèles de ses composants à www.maxim-ic.com/tools/spice et Fairchild fournit les modèles de quelques composants (nécessite de s'enregistrer gratuitement). Prenons l'exemple du fameux dual-amplificateur opérationnel LM358 de Maxim : nous récupérons son modèle **lmx358.fam** dans les *Operational Amplifier Macromodel* à www.maxim-ic.com. Nous l'assemblons, pour en valider le comportement, autour d'un amplificateur non inverseur tel que décrit en figure 9.

3.4 Le circuit de Chua

Il est habituel de considérer le comportement des circuits électroniques soit comme trivialement linéaire, et ne proposant que peu d'intérêt autre que d'identifier des points de fonctionnement




```
* demo du modele Maxim LM358

.include lmx358.fam
* OPAMP : VCC VEE IN+ IN- OUT

vcc 99 0 dc 5
vss 98 0 dc -5

ve 97 0 dc 1
x1 99 98 97 5 1 OPAMP
r1 5 1 10k
r2 5 0 10k
Ra 1 2 10
Rc 2 0 500
C  2 0 500n

.control
dc ve -8 8 0.1
plot v(2)
.endc
```

Figure 9 : Exploitation du modèle SPICE de l'amplificateur opérationnel double LM358 proposé par Maxim/Dallas. Ce circuit est monté en amplificateur-inverseur de gain -2, avec une alimentation de ±5 V. La tension d'entrée ve balaie la gamme ±8 V : nous constatons dans un premier temps la saturation négative lorsque la tension de sortie est en-dessous de la tension d'alimentation (tension d'entrée entre -8 et -2,5 V). La région linéaire pour une tension d'entrée entre -2,5 et +2,5 correspond au fonctionnement correct de la boucle de rétroaction négative. La saturation positive est atteinte au-dessus de 2,5 V de tension d'entrée. Le régime au-delà d'une tension d'entrée de 7 V, générant une tension de sortie nulle, se traduirait probablement par une destruction du composant. Notez que nous avons exploité ici un sous-ensemble du modèle proposé par Maxim, en ne prenant qu'un amplificateur opérationnel OPAMP au lieu du composant complet LMX358.

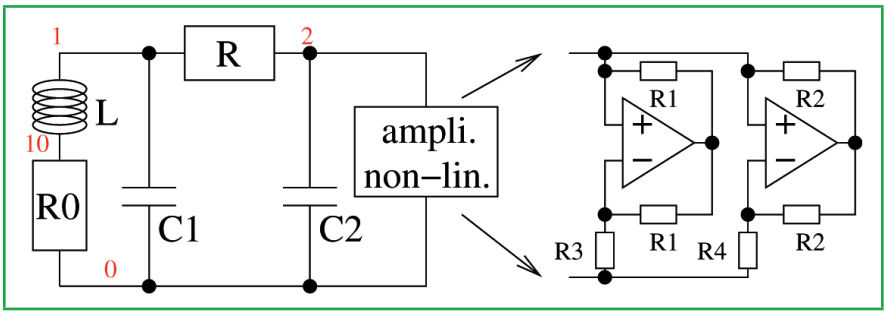


Figure 10 : Schéma de principe du circuit de Chua : un choix approprié des couples (R1,R3) et (R2,R4) permet de définir les pentes et tensions seuils de changements de gain de l'amplificateur non linéaire par saturation d'un des deux étages d'amplification, tel que décrit dans [8]. Le reste du circuit est composé d'un filtre RLC classique, avec R0 représentatif de la résistance finie de la bobine (en pratique, R0 est faible, ici égale à 10 Ω).

correspondant aux attentes du concepteur, soit en mode saturé tel que nous le retrouvons dans les circuits numériques.

Un cas pathologique et élégant, accessible au moyen de SPICE, est le circuit dit de Chua **[8]**, du nom de son concepteur à l'Université de Berkeley, représentatif du comportement d'une classe de circuits dont l'intérêt pratique – au-delà de l'aspect esthétique et d'illustration de concepts physiques associés aux éléments non linéaires – tient en la communication sécurisée **[9]**.

Le circuit de Chua est un circuit chaotique[4] excessivement simple à réaliser et dont le comportement est impossible à prédire. Concrètement, SPICE va nous aider à identifier des ensembles de valeurs de composants amenant au comportement chaotique du circuit, et éventuellement fournir des séries de points issues de modélisations du comportement du circuit (Fig. 10).

L'originalité du circuit de Chua est de comporter un amplificateur explicitement non linéaire dont le gain change en fonction de la tension à son entrée. La stratégie exploitée dans cet exemple consiste à utiliser deux amplificateurs en parallèle, dont un passe en saturation et donc ne participe plus au gain global à partir d'une certaine tension d'entrée. Une alternative est l'exploitation de diodes pour injecter la non-linéarité : le gain de l'amplificateur change selon que la tension d'entrée soit au-dessus ou en-dessous de la tension de jonction de la diode. Ce circuit illustre donc le comportement grossièrement exagéré d'effets qui pourraient se manifester subtilement lorsqu'un amplificateur réel pas tout à fait linéaire est utilisé.

La modélisation de ce circuit par un simple modèle d'amplificateur opérationnel de source de tension commandée en tension avec un gain variable avec la fréquence n'est pas possible, ne serait-ce que parce qu'un modèle aussi simple n'inclut pas la notion de saturation de la sortie lorsque la valeur absolue de l'entrée dépasse un certain seuil. Nous constatons dans cet exemple qu'un modèle complexe, publié par le fabricant de l'amplificateur (Analog Devices AD712), est nécessaire pour illustrer le comportement chaotique du circuit. Le lecteur pourra constater que les modèles plus simples d'amplificateurs ne permettent pas d'atteindre ce résultat.

Afin de compléter l'analyse proposée sur la figure 11, nous avons fait le choix de passer par un fichier décrivant la netlist dans lequel la valeur de l'inductance porte un nom quelconque, et de passer par **`sed`** pour remplacer ce nom par des valeurs issues d'une boucle itérée en shell. L'instruction **`print`** de SPICE, exécutée en

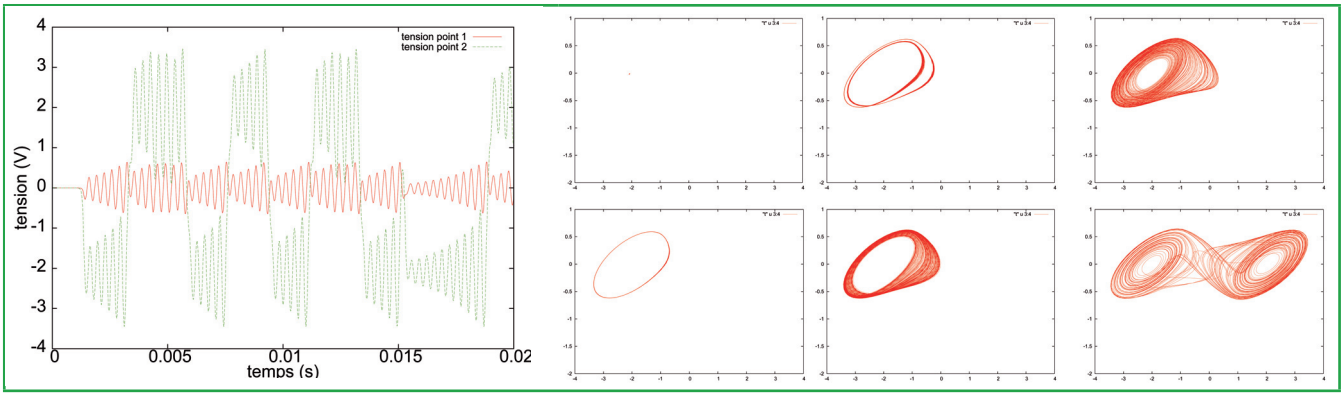


Figure 11 : Gauche : évolution temporelle des tensions V(1) et V(2) (voir figure 10 pour la nomenclature des nœuds), illustrant l'alternance entre deux bassins d'attraction de V(2) alors que V(1) apparaît comme un signal pseudo-périodique (ici, L=18 mH). Droite : évolution de l'attracteur dans l'espace des phases (V(2), V(1)). De gauche à droite et de haut en bas : solution stable où V(1) et V(2) restent constantes (position (-2,0)) avec le temps (L=14,3 mH), solution périodique (L=15,3 mH), bifurcation pour obtenir une solution bi-périodique (16,8 mH), développement d'un attracteur (17,2 et 17,6 mH) avant d'atteindre une nouvelle bifurcation produisant un attracteur à deux bassins (18 mH).

[4] Excessivement sensible à ses conditions initiales, i.e. dont la trajectoire ne peut pas être prédite à long terme du fait de la croissance exponentielle de toute erreur sur les conditions initiales, bien que le comportement général soit borné autour d'un ensemble de trajectoires nommé « attracteur ».

batch mode (**ngspice -b fichier.cir**), renvoie au shell le résultat de la simulation, qui est alors traité et exploité par **gnuplot** pour tracer l'évolution des tensions dans l'espace (V(1),V(2)).

4 Au-delà de l'électronique

Il est classique **[10, pp.786 & 806] [11, p.381]** d'introduire les équations différentielles régissant les systèmes physiques dynamiques pour illustrer la généralité de la représentation mathématique quel que soit le système considéré – électrique, thermique ou mécanique. Deux analogies sont généralement fournies pour présenter la portée des résultats : les analogies entre oscillateurs électroniques et mécaniques, et la diffusion thermique. Dans le premier cas, une équation différentielle du second ordre régissant un oscillateur s'applique aussi bien au pendule (mécanique) qu'à l'oscillateur électronique. L'identification des termes de pulsation et de pertes permet de retrouver les analogies entre position/charge électrique, vitesse/courant, et entre propriétés de composants mécaniques et électroniques. Les valeurs des composants équivalents dépendent de la géométrie des éléments mécaniques ou thermiques (dimensions, masse, ...) et suivent évidemment les mêmes lois de compositions que les composants électriques.

4.1 Thermique

La loi d'évolution temporelle de température T(t) dans un problème unidimensionnel d'un bloc de matière de masse m composé d'un matériau de capacité calorifique c_p et de résistance thermique R dont on chauffe une face avec la puissance P est :

$$m \times c_p \frac{dT}{dt} = P - (T - T_0) \times \frac{1}{R}$$

avec T_0, la température ambiante du bloc de matière (en l'absence de source externe d'énergie). Cette équation différentielle du premier ordre suit la même structure que le formalisme de résolution du premier exemple que nous avons résolu au moyen de SPICE :

$$C \times \frac{dV}{dt} = I - V\frac{1}{R}$$

L'analogie thermique-électrique consiste donc en l'identification des termes, à savoir un condensateur C se comportant comme la capacité calorifique de la masse chauffée (son inertie aux sollicitations externes), le potentiel V par rapport à la masse est analogue à la différence de température T—T0, R la résistance à la propagation du signal et P le flux de l'énergie (en J/s, soit des W), l'apport de puissance, équivalent au courant injecté dans un circuit électronique (Tableau 1).

La puissance de cette analogie tient non seulement en l'exploitation de SPICE pour résoudre un problème de thermique, mais surtout pour modéliser le comportement couplé d'un asservissement thermique par un circuit électronique. Au moyen de composants électroniques équivalents représentatifs du comportement thermique d'un thermostat, les constantes de temps ou dépassement de consigne d'asservissements analogiques au moyen de boucles de rétroactions négatives peuvent être modélisées sous SPICE **[12, 13]**.

4.2 Mécanique

Le même raisonnement s'applique à la mécanique[5], par exemple, avec l'analogie entre l'équation du second ordre régissant le comportement d'un circuit RLC série et un oscillateur mécanique amorti (ressort-masse-piston). Là encore, une masse M liée par un ressort de raideur k avec dissipation d'énergie dans un amortisseur R suit une loi de sa position x(t) en fonction du temps t et de la force F qui lui est appliquée.

$$M\frac{d^2x}{dt^2} + R\frac{dx}{dt} + kx = F$$

dont on déduit les analogies (Tableau 2).

Tableau 1

Thermique (unite)	Electrique (unite)
R (K/W) resistance	R (Ω) résistance
m × c_p (J/K) capacite calorique	C (F) capacite
T (K) temperature	V (V) potentiel
P (W) puissance	i (A) courant

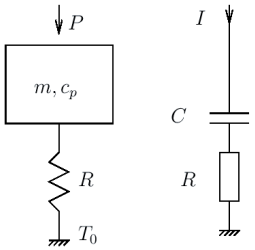
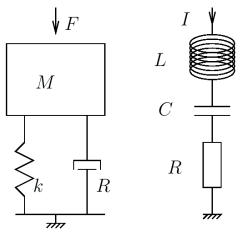


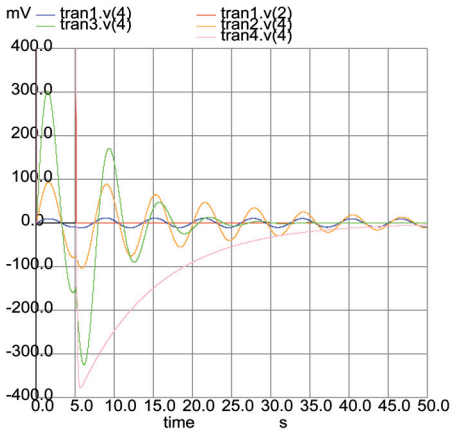
Tableau 2

Thermique (unite)	Electrique (unite)
R (N.s/m) amortissement	R (Ω) résistance
M (kg) masse	L (H) inductance
k (N/m) raideur	1/C (1/F) capacité
x (m) position	q (C) charge
v=dx/dt (m/s) vitesse	i=dq/dt (A) courant
F (N) force	V (V) tension



[5] http://www.swarthmore.edu/NatSci/echeeve1/Ref/LPSA/Analogs/ElectricalMechanicalAnalogs.html

La conséquence de cette analogie est que d'une part, SPICE est un outil approprié pour résoudre le comportement d'un système dynamique mécanique (Fig. 12), mais surtout, qu'il permet de combiner des comportements mécaniques, thermiques et électroniques de systèmes.




```
* pendule

* M*d2x/dt2+R*dx/dt+kx=0
* L*d2q/dt2+R*dq/dt+1/C*q=0
* i=dq/dt sur R, ve=force appliquee sur C

ve 2 0 dc 1 pulse(0 1 100m 10m 10m 5000m 0)
C 2 3 1
L 3 4 1
R 4 0 0.01

.control
destroy all
tran 10m 50
alter R 0.1
tran 10m 50
alter R 0.4
tran 10m 50
alter R 10
tran 10m 50
set hcopydevtype=postscript
set hcopypscolor=true
set  color0=rgb:f/f/f
set color1=rgb:0/0/0
hardcopy pendule.ps tran1.v(2) tran1.v(4) tran2.v(4)
+ tran3.v(4) tran4.v(4) ylimit -0.31 0.31
plot tran1.v(2) tran1.v(4) tran2.v(4)
+ tran2.v(4) tran3.v(4) ylimit -0.31 +0.31
.endc
```

Figure 12 : Simulation d'un montage masse-ressort-amortisseur, pour diverses valeurs d'amortissement. L'analogie mécanique-électrique démontre ici que la période est bien égale à 2π lorsque nous choisissons une raideur et une masse de 1, et que l'amortissement n'influe par sur la période mais uniquement sur la vitesse de décroissance du signal. Dans cet exemple, la masse est excitée par une force (tension) pendant les 5 premières secondes, avant d'être relâchée. En rouge, le signal d'excitation (la force appliquée sur la masse, qui atteint 1 V et sort donc de l'axe des ordonnées), puis respectivement en bleu, orange, vert, et rose, des valeurs croissantes de l'amortissement. Notez que la courbe bleue, qui présente le plus faible amortissement, présente aussi la plus faible amplitude : en effet, un système fortement résonnant (faibles pertes) nécessite de nombreuses périodes d'excitation pour atteindre un régime stationnaire, et les deux impulsions (front montant puis descendant du créneau d'excitation) sont insuffisantes pour fournir suffisamment d'énergie dans ce cas. D'un point de vue syntaxique, toute ligne commençant par « + » est considérée comme la suite de la ligne précédente.

Un cas particulier qui rend l'interaction mécanique-électrique explicite est l'exploitation de composants piézoélectriques, dont les constantes de couplage électromécaniques traduisent explicitement la dépendance du courant avec la vitesse de déplacement (effet piézoélectrique direct), et du déplacement mécanique avec le potentiel appliqué (effet piézoélectrique inverse), pour donner le modèle classique de Butterworth-Van Dyke.

Conclusion

SPICE est un outil libre, portable sur toute plate-forme compatible avec la norme POSIX, de modélisation permettant de caractériser le comportement temporel ou spectral de circuits électroniques. Après un bref rappel des concepts généraux de la simulation – comportant un code source décrivant la topologie du circuit et quelques commandes simples pour paramétrer la simulation – nous avons proposé un certain nombre d'exemples pratiques incluant la simulation du comportement de filtres passifs, actifs et oscillateurs. Nous nous sommes finalement efforcés d'étendre la plage d'utilisation de cet outil à des problèmes autres qu'électroniques – et notamment mécaniques et thermiques, en exploitant les analogies classiques avec l'électronique – en vue de simulations multiphysiques. L'ensemble des références bibliographiques sont disponibles en versions numériques auprès de l'auteur, tandis que les codes sources des exemples proposés dans ce document sont disponibles à http://jmfriedt.free.fr.

Il peut être utile de mentionner que divers projets – notamment **gspiceui** et **xcircuit**[6] – s'efforcent de fournir une interface graphique pour la réalisation des netlists. L'outil propriétaire mais gratuit Eagle de CADSoft propose des scripts (ULP) de conversion de schémas en netlist SPICE. Un point commun de tous ces outils est leur inefficacité à gérer des circuits complexes, et surtout, à maîtriser les paramètres de simulation et les modèles associés aux composants : le lecteur désireux de quitter la ligne de commandes pour ces interfaces graphiques le fera à ses propres risques quant à la validité de ses simulations. ■

Remerciements

L'inspiration concernant l'analyse de problèmes multiphysiques sous SPICE est issue de discussions avec R. Boudot et S. Galliou (département temps-fréquence, FEMTO-ST, Besançon). R. Brendel a proposé une présentation de SPICE dans le cadre de son cours à l'École Nationale Supérieure de Mécanique et Microtechniques (ENSMM), qui a fourni la trame de ce document.

Liens

[1] J.-M Friedt, « Affichage et traitement de données au moyen de logiciels libres », *GNU/Linux Magazine France* 111 (Décembre 2008)

[2] P. Nenzi & H. Vogt, *Ngspice Users Manual*, ver. 22, Sept. 2010, disponible à http://ngspice.sourceforge.net/docs/ngspice-manual.pdf

[3] *The ARRL Handbook for Radio Amateurs*, Ed. ARRL (1998)

[4] http://ubuntuforums.org/showthread.php?p=4953252#post4953252

[5] F.H. Lei, J.- F. Angiboust, W. Qiao, G.D. Sockalingum, S. Dukic, L. Chrit, M. Troyon & M. Manfait, *Shear force near-field optical microscope based on Q-controlled bimorph sensor for biological imaging in liquid*, J. of Microscopy, 216 (3), 2004, pp. 229-233, ou pour une version commerciale d'un produit fournissant les mêmes fonctionnalités : http://www.nanoanalytics.com/en/hardwareproducts/q-control

[6] S.G. Parler Jr., *Improved Spice Models of Aluminum Electrolytic Capacitors for Inverter Applications, IEEE Trans. on Industry Applications 34* (4) (2003) 929-935

[7] http://www.vishay.com/docs/28327/030031.pdf pour les condensateurs électrolytiques commercialisés par Vishay, http://www.murata.com/emc/knowhow/pdfs/te04ea-1/12to16e.pdf pour une note d'application de Murata

[8] M.P. Kennedy, *Three Steps to Chaos – Part II: A Chua's Circuit Primer, IEEE Trans. on Circuits and Systems 40 (10)*, Octobre 1993, disponible à http://www.eecs.berkeley.edu/~chua/papers/Kennedy93b.pdf

[9] J. Neff & T.L. Caroll, *The Amateur Scientist: Circuits That Get Chaos in Sync*, Scientific American pp.101-103, Aout 1993

[10] C. More & D. Augier, Physique 2nde année, MP, MP*, PT, PT*, Editions TEC & Doc, Lavoisier, 2004

[11] S. Olivier, Physique 2nde année PC, PC*, Editions TEC & Doc, Lavoisier, 2004

[12] S. Galliou, *Thermal behaviour simulation of quartz resonators on an oven environment*, IEEE Trans. Ultrason. Ferroelectrics and Frequency Control 42 (5), (1995) 832-839

[13] R. Boudot, C. Rocher, N. Bazin, S. Galliou, & V. Giordano, *High-precision temperature stabilization for sapphire resonators in microwave oscillators*, Rev. Sci. Instrum. 76, 095110 (2005)

[6] http://opencircuitdesign.com/xcircuit

Abonnez-vous !

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

Économisez plus de 20%*

* Sur le prix de vente unitaire France Métropolitaine

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

4 Numéros de Open Silicium

Les 3 bonnes raisons de vous abonner :

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

4 façons de commander facilement :

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

par ABONNEMENT : **27€***

au lieu de 36,00* en kiosque

Économie : 9,00 €*

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

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

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

Open Silicium

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

Vos remarques :

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

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

Tournez SVP pour découvrir toutes les offres d'abonnement

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

➡ Voici une sélection des offres de couplage avec Open Silicium

offre 13 — Open Silicium Magazine (4 nos)

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

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

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

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

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

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

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

Vous pouvez également vous abonner sur : www.ed-diamond.com ou par Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

➡ Nos Tarifs s'entendent TTC et en euros

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

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

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

Mes choix :

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

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

Je choisis de régler par :

❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond

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

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

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Open Silicium

INTRODUCTION AU DÉVELOPPEMENT ANDROID

À L'ATTENTION DE CEUX QUI N'AIMENT PAS ECLIPSE

par Denis Bodor

Android est une plateforme très attractive. De plus en plus populaire, elle rend possible le rêve de disposer d'un système autonome et communicant basé sur Linux à chaque moment. Android a, pour l'utilisateur GNU/Linux de longue date, une saveur authentique qui ne laisse pas de marbre et qui relève de la beauté d'une regex, d'un joli script shell ou tout simplement d'une ligne de commandes bien composée. Mais les tutoriels disponibles sont souvent rédigés à l'attention d'utilisateurs plus tentés par les IDE et les environnements clés en main. Voyons comment développer des applications Android suivant une approche plus Unix...

Posons de suite les bases incontournables : la création d'une application Android est avant tout un développement Java reposant sur un SDK généralement destiné à des programmeurs plus adeptes des environnements de développement graphiques que de la ligne de commandes. La masse de développeurs Unix est importante, mais la majorité du marché est encore sous Windows et, dans une moindre mesure, Mac OS X. En ne suivant pas la large route tracée pour ces utilisateurs, il devient plus difficile de parvenir à un résultat. Mais, à l'instar du développement d'applications pour iPhone (cf *GNU/Linux Magazine* hors-série 51), la tâche n'est pas impossible si nos préférences résident dans l'utilisation stricte des outils traditionnels : son éditeur de texte/code fétiche et un bon shell.

Heureusement pour nous, Android est construit sur une base Linux et en hérite ainsi quelques qualités, généreusement conservées par les équipes de développement de Google. Ainsi, le SDK Android intègre un certain nombre d'outils en ligne de commandes, dont le très sympathique **adb** (*Android Debug Bridge*). La philosophie Unix et une certaine ouverture sont donc bien présentes dans Android, ne serait-ce que par la présence, dans tous les *smartphones*, d'un mode *debug* via USB activable facilement dans l'interface de configuration. Nous sommes très loin des principes appliqués par Apple pour son iPhone, préférant axer sa politique vers une séparation utilisateurs/développeurs et où le développeur aura bien du mal à ignorer les recommandations de la firme de Cupertino. Il semblerait que la notion de « préférences personnelles » en termes de développements ne soit pas unanimement respectée. C'est dommage, mais finalement pas très surprenant.

1 Prérequis

Pour marcher sur le sentier battu de la création d'applications Android, il suffit de suivre les recommandations des différents sites proposant des introductions au sujet. En ce qui concerne la voie que nous emprunterons ici, il vous faudra :

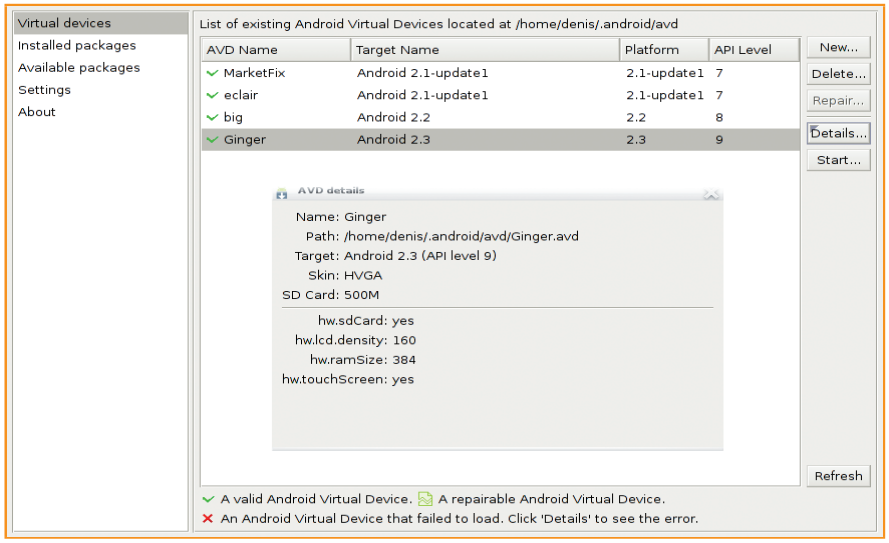
- Une bonne distribution GNU/Linux regroupant les outils courants (ici Debian GNU/Linux).
- Un éditeur de texte étant votre compagnon de route de longue date (ici Vim).
- Un environnement de développement Java fonctionnel. Sur une Debian, ceci se résumera à l'utilisation de la commande `aptitude install ant openjdk-6-jdk`.
- Le SDK Android proposé par Google sur le site officiel http://developer.android.com. Celui-ci se présente

sous la forme du *tarball* **android-sdk_r08-linux_x86.tgz** à la date où est rédigé cet article.

L'installation du SDK Android se résume donc à :

```
% wget http://dl.google.com/android/android-sdk_r08-linux_x86.tgz
% tar xfzv android-sdk_r08-linux_x86.tgz
% cd android-sdk-linux_x86/tools
% ./android
```

En l'absence d'argument passé en ligne de commandes, vous verrez apparaître l'interface graphique de configuration du SDK et du gestionnaire de machines virtuelles Android (AVD pour *Android Virtual Devices*) :



Un tour dans « Available Packages » et vous pourrez télécharger la liste des composants du SDK et procéder à leur installation. C'est gros, allez prendre un café, vous en avez le temps. Au terme de la procédure, un message vous avertira de la nécessité de redémarrer **adb**. En effet, **adb** n'est pas simplement un outil permettant de manipuler le système de fichiers d'un périphérique Android ou d'y obtenir un shell, c'est également un démon permettant de communiquer avec une machine virtuelle Android, manipuler les connexions réseau, déboguer un système, etc. Il forme la base communicante du SDK. Au premier lancement du binaire, celui-ci s'installera en mémoire jusqu'à son arrêt avec **adb kill-server** (ou **killall adb**).

Vous pouvez maintenant quitter l'environnement graphique. Vous y reviendrez éventuellement lorsque vous manipulerez les machines virtuelles pour tester notre première application.

2 Le classique Hello World

Incontournable et imparable premier code que chaque développeur se doit de réaliser lorsqu'il explore un nouveau langage ou un nouvel environnement, le fameux « Hello World » sera notre première application. Nous nous efforcerons de le faire évoluer jusqu'à un certain point dans la suite de cet article en n'oubliant pas qu'il ne s'agit ici que de mettre le pied à l'étrier d'une façon bien trop rarement détaillée sur le Web.

Lorsqu'on se passe de l'environnement de développement Eclipse, il convient de faire quelques sacrifices, mais c'est peu de choses. Tout ce sur quoi il faudra faire l'impasse sera comblé par un éditeur de code efficace, l'outil de construction **ant** et la commande **android** livrée avec le SDK. Ce SDK est relativement bien conçu d'un point de vue du respect de la gestion des paquets pour votre distribution (quelle qu'elle soit). En effet, il suffit de l'installer dans un répertoire quelconque, pour ensuite spécifier le chemin complet vers **android** pour que l'ensemble reste cohérent. Ce n'est qu'en cas de déplacement du SDK ou des sources (d'un hôte vers un autre ayant un SDK désarchivé à un autre endroit) qu'il vous sera nécessaire de procéder à quelques ajustements minimes (dans le fichier **local.properties**). Dans la suite de cet article, les outils livrés avec le SDK dans un sous-répertoire **tools** (**adb**, **android**, **ddms**, etc.) sont utilisés sans spécifier de chemin absolu pour des raisons évidentes de lisibilité du code. Vous pouvez spécifier ce chemin à chaque ligne ou tout simplement l'ajouter dans votre variable d'environnement **PATH**.

La première étape pour démarrer le développement d'une application Android est la création d'un projet. Ceci revient à initialiser un ensemble de fichiers ainsi qu'une arborescence destinée à recevoir votre code. Pour cette première application, nous allons utiliser **android** de la manière suivante :

```
% android create project \
--target 1 --name LefHello \
--path ~/SRC/Java/Android/LefHello \
--activity LefHelloActivity \
--package lef.example.myhello
```

La directive utilisée, comme son nom l'indique, permet la création d'un projet (**create project**) avec les différentes options passées en argument :

- **--target** ou **-t** : l'identifiant numérique (ID) du type de projet (obligatoire). Il s'agit d'une valeur numérique entre 1 et 9 (depuis l'arrivée de Gingerbread) identifiant le niveau d'API Android à utiliser et donc la version d'Android pour laquelle votre application sera créée. Attention, l'ID n'est pas égale au niveau d'API. Utilisez **android list targets** pour obtenir une liste descriptive en fonction des éléments du SDK que vous avez installé. Ici, nous utilisons **1** pour une application Android 1.5 (API niveau 3, révision 4). Notre application pourra donc fonctionner sur n'importe quelle version supérieure ou égale à 1.5, mais nous serons limités aux fonctionnalités présentes dans cette version. Souvenez-vous en, c'est une source typique d'erreur lors de la compilation.

- **--name** ou **-n** : le nom du projet (optionnel). À ne pas confondre avec le nom de l'application, qui sera automatiquement le nom de l'activité que vous avez spécifié par ailleurs.

- **--path** ou **-p** : le chemin complet vers la racine de l'arborescence du projet (obligatoire).

- **--activity** ou **-a** : le nom de l'activité principale (obligatoire). Une application Android possède presque toujours une activité (sauf cas particulier du développement d'un service, d'un *widget* ou d'un fond d'écran animé ne disposant pas d'une interface de configuration, par exemple).

- **--package** ou **-k** : le nom du paquet (obligatoire). Un nom de paquet est un identifiant unique pour votre application. C'est le nom du paquet Java qui la contient.

Après utilisation de cette commande, vous retrouverez, dans l'arborescence de votre projet (spécifié par **--path**), un certain nombre de répertoires. **bin** accueillera votre future application packagée, **libs** les bibliothèques intégrées, **res** l'ensemble des ressources de votre application et **src** vos sources. **res** contient les images (**drawable**), les fichiers XML décrivant vos interfaces (**layout**) et les valeurs statiques (**values**). Vous noterez que dans ce dernier répertoire est automatiquement créé un fichier **strings.xml** contenant une balise **name=app_name** ayant pour valeur le nom de l'activité que vous avez mentionné lors de la création du projet. En jetant un œil au fichier **AndroidManifest.xml**, vous constatez que cette valeur est reprise sous la forme **android:label="@string/app_name"**. C'est un exemple de chaîne de caractères ainsi définie automatiquement pour votre application. Ici, il vous suffira de changer le contenu de **strings.xml** pour renommer votre application (son nom, et non celui du paquet Java qui permet de l'identifier).

Le fichier **AndroidManifest.xml** regroupe les informations essentielles que le système Android cible doit connaître avant de lancer l'application : le nom de l'application, son icône, ses composants, les permissions, le niveau minimum de l'API utilisée, etc.

En regardant du côté du répertoire **src**, vous retrouverez une arborescence complète correspondant au nom de paquet que vous avez spécifié à la création du projet. Dans le dernier répertoire, vous constaterez que, gentiment, le SDK a créé pour vous un premier fichier source Java nommé d'après le nom de l'activité. Il ne contient rien de bien extraordinaire, il ne s'agit que d'une ossature de base pour faciliter vos premiers pas :

```
package lef.example.myhello;

import android.app.Activity;
import android.os.Bundle;

public class LefHelloActivity extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Une seule méthode est définie. Elle sera utilisée au lancement de votre application (une sorte de **main()** du C). À ce stade, vous pouvez déjà obtenir une application compilée et packagée avec **ant debug** à la racine du projet. Une application qui ne fait rien du tout. La méthode **onCreate** que nous déclarons, outrepassant la méthode de la superclasse avec **@Override**, ne fait finalement pas grand chose. Nous utilisons simplement **setContentView** pour définir le contenu de la vue (l'écran de notre activité). Nous pourrions la remplacer par :

```
import android.widget.TextView;
[...]
        TextView tv = new TextView(this);
        tv.setText("Hello, Android from Lefinnois World !\n");
        setContentView(tv);
```

Ici, **tv** est un objet **TextView**, un simple calque de texte que nous remplissons avec la mention de notre choix. Dans le cas précédent, au lieu de passer en paramètre un objet *View*, nous utilisons une référence vers une ressource. **R.layout.main** est un objet créé à partir d'une représentation XML (**res/layout/main.xml**). Dans le cas d'une utilisation de l'IDE Eclipse, c'est le greffon pour Android qui créera l'objet en question. Avec une utilisation en ligne de commandes, le fichier sera généré par **aapt** via **ant** et placé dans **gen/lef/example/myhello.R.java**. Vous pouvez consulter ce fichier pour voir en quoi il consiste, mais il est totalement inutile d'y faire la moindre modification puisque celle-ci sera écrasée lors de la prochaine compilation (le commentaire en début de source Java est d'ailleurs relativement explicite sur ce point) :

```
package lef.example.myhello;

public final class R {
    public static final class attr {
    }
    public static final class layout {
        public static final int main=0x7f020000;
    }
    public static final class string {
        public static final int app_name=0x7f030000;
    }
}
```

À SAVOIR

IMSI et IMEI

L'IMSI, ou *International Mobile Subscriber Identity*, est un numéro unique permettant d'identifier un utilisateur sur un réseau GSM/UMTS. Celui-ci n'est normalement pas connu de l'utilisateur et est stocké dans la carte SIM correspondant à son abonnement. Il se compose d'un code pays (MMC pour *Mobile Country Code*), d'un code opérateur (MNC ou *Mobile Network Code*) et du numéro de l'abonné dans la base de l'opérateur (MSIN ou *Mobile Station Identification Number*).

L'IMEI, quant à lui, désigne l'*International Mobile Equipment Identity*, sorte de numéro de série inter-fabricants du périphérique mobile de téléphonie. Il est utilisé généralement pour identifier un équipement autorisé sur le réseau d'un opérateur. Vous pouvez avoir accès à ce numéro, soit en composant ***#06#** sur le périphérique, soit en regardant tout simplement le numéro sérigraphié sur le téléphone (généralement dans l'emplacement batterie). Le numéro IMEI n'a pas de lien direct avec l'abonné au réseau. Cet identifiant est utilisé pour retrouver les périphériques déclarés volés et permet de blacklister un téléphone de manière globale quel que soit l'opérateur. Cette sécurité est cependant à considérer avec prudence. Le numéro IMEI, bien que difficile à modifier sur certains périphériques, n'est pas infalsifiable. Par effet de bord, certaines opérations de mise à jour de smartphone, par exemple, peuvent réinitialiser l'IMEI et rendre un matériel inutilisable. Il s'agit, bien entendu, d'opérations de mise à jour non prévues ou non autorisées par le fabricant ou l'opérateur.

À ce stade, sans pousser trop loin notre exploration du mode de développement pour Android, nous pouvons déjà jouer un peu. C'est plus ou moins la technique habituelle (parfois non avouée) de la découverte d'un nouvel environnement. À partir d'un code qui fonctionne, ajoutons deux ou trois lignes « juste pour voir » si on appréhende correctement la chose. Ici, nous allons oublier **R** et nous concentrer sur l'utilisation d'un objet **TextView**. L'affichage simple d'un texte n'est pas très complexe, voyons si nous pouvons utiliser d'autres types de ressources. Pourquoi pas afficher quelques informations sur le périphérique utilisé, comme les numéros IMSI et IMEI ? Voici à quoi pourrait ressembler un code brouillon de ce genre (« brouillons » pour éviter que les développeurs Java qui me lisent ne hurlent à la mort) :

```
package lef.example.myhello;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.telephony.TelephonyManager;

public class LefHelloActivity extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        TelephonyManager mTelephonyMgr =
            (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        String myIMSI = mTelephonyMgr.getSubscriberId();
        String myIMEI = mTelephonyMgr.getDeviceId();
        TextView tv = new TextView(this);
        tv.setText("Hello, Android from Lefinnois World !\n");
        tv.append(myIMSI);
        tv.append("\n");
        tv.append(myIMEI);
        setContentView(tv);
    }
}
```

Les lignes mises en évidence sont nos ajouts. Notre code est relativement simple à comprendre. Nous instancions un objet **TelephonyManager** nous permettant de récupérer les informations et utilisons ensuite les méthodes **getSubscriberId()** pour l'IMSI et **getDeviceId()** pour l'IMEI. Il ne nous reste plus, ensuite, qu'à utiliser **append** pour ajouter du contenu sur notre objet **TextView**.

La documentation développeur en ligne sur http://developer.android.com, qui doit être la page ouverte par votre navigateur lorsque vous vous lancez dans ce genre de développement, nous apprend que la classe **TelephonyManager** est présente depuis l'API niveau 1. Il est très important de vérifier ce point pour chaque classe utilisée, car c'est une source de problèmes très courante, et les messages d'erreur Java ne sont pas forcément un modèle de clarté.

3 Finir HelloWorld : coder ne fait pas tout

Notre code est beau et, normalement, il fonctionne. Ceci n'est cependant pas suffisant pour en faire une application. Nous allons ajouter quelques éléments, certains nécessaires et d'autres purement décoratifs. Nous l'avons dit plus haut, les informations sur une application sont listées dans le fichier **AndroidManifest.xml** à la racine de votre projet. Voici le notre :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="lef.example.myhello"
      android:versionCode="1"
      android:versionName="1.0.1">
    <application android:label="@string/app_name"
                 android:icon="@drawable/ic_launcher_lefhello">
        <activity android:name=".LefHelloActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
</manifest>
```

Nous connaissons déjà la plupart des éléments. Nous avons ajouté une référence à un *drawable* ou, en d'autres termes, un élément graphique. Il s'agit d'une petite icône pour notre application. Tout comme pour le nom de l'application, nous pointons simplement le nom du fichier (sans l'extension) et le SDK se chargera du reste. Le fichier est appelé **ic_launcher_lefhello** par convention, comme conseillé dans le *Icon Design Guidelines, Android 2.0*. Si vous avez créé le projet pour une version minimum d'Android en 1.5, vous retrouverez un répertoire **res/drawable** dans votre arborescence. Il vous suffira alors de créer le fichier **ic_launcher_lefhello.png**. Avec un niveau d'API plus important, vous devrez prendre en charge non pas un type d'icône, mais trois. En effet, les nouvelles versions du système sont destinées à supporter une gamme plus diversifiée de périphériques dont les écrans peuvent aller jusqu'à 7 pouces de diagonale et une résolution en conséquence.

Pour régler le problème de diversité, Android utilise deux notions pour répartir les systèmes d'affichage de l'ensemble des périphériques :

- La taille de l'écran. Celle-ci peut être petite (*small*), moyenne (*normal*) ou grande (*large*).
- La densité. Cette valeur est classée sous trois qualificatifs différents : ldpi (*low dpi*), mdpi (*medium dpi*), ou hdpi (*high dpi*).

La notion de densité n'est pas directement liée à la taille d'un écran. On peut parfaitement imaginer un écran de 3 pouces affichant 240×320 et un écran de 7 pouces d'exactement la même résolution. De la même manière, deux écrans de 5 pouces de diagonale pourraient afficher une résolution de 480×800 pour l'un et 240×400 pour l'autre. Pourtant, une icône ayant une taille fixe en pixels ne s'affichera pas de la même manière sur chacun de ces écrans. Tantôt minuscule (petit écran à haut résolution), tantôt géante et laide (grand écran à faible résolution).

Android s'occupe de faire en sorte que votre application s'affiche de manière correcte et utilisable quel que soit le périphérique utilisé. Cependant, il ne peut pas faire de miracle et vous devez fournir les éléments pour qu'il puisse le faire de manière acceptable. Ces considérations portent sur l'agencement (*layout*) de vos activités qui composent votre application et, c'est le cas qui nous intéresse ici, sur la taille des icônes.

Lorsque vous créez un projet pour le niveau d'API 7, par exemple, vous retrouverez non pas un répertoire **drawable** dans **res**, mais trois : **drawable-ldpi**, **drawable-mdpi**, et **drawable-hdpi**. Ainsi, c'est lors de l'exécution de l'application que le système déterminera le jeu d'icônes à utiliser en fonction des caractéristiques du périphérique. Pour une icône de lancement comme la notre, les tailles à utiliser sont les suivantes :

- ldpi : 36 x 36 pixels ;
- mdpi : 48 x 48 pixels ;
- hdpi : 72 x 72 pixels.

Vous retrouverez la correspondance avec les répertoires sans peine, je pense. Sachez simplement que le répertoire **res/drawable**, s'il est présent, est considéré comme **drawable-mdpi**. Testez vos applications avec l'émulateur du SDK pour vous assurer du rendu. Dans le doute, vous pouvez également, en sacrifiant un peu de design, reposer sur les fonctionnalités du système. En effet, la plateforme va automatiquement ajuster la taille et la densité de vos images en fonction du périphérique. Ainsi, même en ne fournissant que des icônes en haute densité, votre application fonctionnera sur un périphérique avec une faible résolution. Ce n'est qu'en précisant explicitement dans le fichier **AndroidManifest.xml** que votre application ne fonctionne que sur certains types d'écrans que vous pourrez limiter ce mécanisme. Il s'agit généralement d'applications graphiques, comme les jeux, qui imposent ce type de limitations. En complément d'informations, sachez qu'à la date où est rédigé cet article, les écrans de moyenne taille à moyenne et haute densités se partagent presque l'ensemble du parc de périphériques avec une répartition de 50/50. Les petits écrans à faible densité représentent moins de 3 % de l'ensemble. Ces informations proviennent des éléments récoltés par l'application Market sur les 14 derniers jours. Ceci est, bien entendu, l'état actuel du parc et celui-ci risque de changer dans les mois à venir avec l'arrivée massive des tablettes Android en fin d'année et la diffusion de Gingerbread et Honeycomb.

Un autre élément important est glissé dans notre fichier **AndroidManifest.xml**. Celui-ci concerne les permissions. La sécurité est un élément important d'Android et, si vous êtes un utilisateur du Market Android, vous avez sans doute remarqué qu'à chaque installation d'application, une confirmation vous est demandée, listant les fonctionnalités dont l'application souhaite faire usage.

Note **Mon application possède des permissions par défaut !**

Il existe un cas très spécifique où une application dispose, par défaut, de deux permissions. Celle permettant de lire le contenu de la carte SD et d'obtenir des informations à propos du téléphone et de sa carte SIM. Ceci nous concerne directement, puisque justement, dans notre code exemple, nous cherchons à afficher des données à la fois du périphérique et de l'utilisateur.

Une application destinée à fonctionner avec une API niveau 3 rev 4 (ID 1 alias android-3 pour Android 1.5) reçoit automatiquement de l'installeur les permissions d'accès à la carte SD et aux informations du téléphone par souci de compatibilité.

Une application prévue pour Android 2.1 et n'étant pas destinée à fonctionner avec une version plus ancienne du système Google ne dispose d'aucune permission par défaut. Si elle tente de faire usage d'une fonctionnalité protégée, son exécution provoquera une erreur.

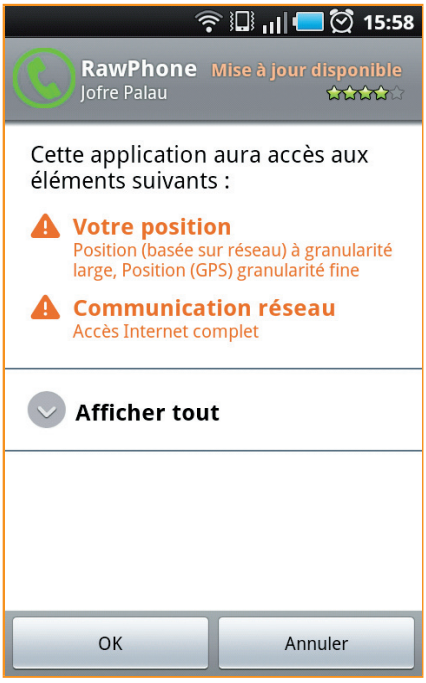
Si votre application est basée sur un SDK Android 1.5, ces deux permissions, qui n'existaient pas au moment de la diffusion du système Google, sont automatiquement attribuées. Voilà pourquoi elles sont en place alors que vous n'avez rien spécifié dans ce sens dans le manifeste. Si, en revanche, vous construisez une application pour Android 2.1 Eclair (ID 5 alias **android-7**), et que vous spécifiez, de plus, que votre application est clairement destinée à cette version de la plateforme en ajoutant ceci dans le manifeste :

```
<uses-sdk android:minSdkVersion="5"
          android:targetSdkVersion="5"
          android:maxSdkVersion="5" />
```

alors, l'installateur ne mettra en place aucune permission spécifique lors de l'installation de votre application. Par contre, lors de l'exécution de cette dernière, une erreur surviendra, mettant fin au processus d'exécution.

En utilisant **ddms**, vous pourrez constater d'où vient l'erreur, puisque les journaux sont relativement clairs à ce niveau :

```
ERROR/AndroidRuntime(7450): java.lang.RuntimeException:
      Unable to start activity ComponentInfo
      {lef.example.myhello/lef.example.myhello.LefHelloActivity}:
      java.lang.SecurityException: Requires READ_PHONE_STATE:
      Neither user 10091 nor current process has
      android.permission.READ_PHONE_STATE.
```



Les applications installées via le Market nécessitent une validation de la part de l'utilisateur en fonction des permissions qu'elles souhaitent obtenir. Ici, l'application Raw Phone demande l'accès aux informations de positionnement et un accès complet à Internet.

Toutes les fonctions disponibles sur un périphérique ne nécessitent pas forcément d'autorisation. Celles-ci sont évaluées par l'installateur de paquets d'Android, le plus souvent via une demande spécifique à l'utilisateur.

Dans le cas présent, notre application fait usage des méthodes fournies par la classe **TelephonyManager**. La documentation développeur nous apprend que celles que nous utilisons nécessitent que l'application dispose de la permission **READ_PHONE_STATE**. Il nous suffit donc d'ajouter dans notre **AndroidManifest.xml** la ligne **<uses-permission android:name = "android.permission.READ_PHONE_STATE" />** dans la portée du **<manifest> </manifest>**. Attention toutefois, il faut bien comprendre que les permissions sont propres aux méthodes et non aux classes qui les implémentent. Ainsi, **getSubscriberId()** nécessite **READ_PHONE_STATE**, mais **getCellLocation()** retournant la position du périphérique nécessite **ACCESS_COARSE_LOCATION** ou **ACCESS_FINE_LOCATION**.

4 On approche de la fin

À ce stade, nous avons une application fonctionnelle, moche, et disposant d'un manifeste adapté. Il ne nous reste plus, donc, qu'à en faire un paquet. Deux options s'offrent à nous, soit créer une version de développement, soit éditer une *release*. La différence réside principalement dans la signature électronique de l'application. En effet, toutes les applications Android sont numériquement signées. Ceci peut paraître contraignant, mais l'utilisation des signature électroniques n'est pas poussée à ses limites. Ainsi, les développeurs utilisent des certificats autosignés, aucune autorité de certification (AC) n'intervient dans le processus. Ceci permet un minimum de sécurité dans l'établissement d'une relation entre des applications et un développeur. Bien entendu, rien ne garantit à l'utilisateur que l'application qu'il installe est bel et bien celle développée par une entité précise. L'infrastructure est cependant disponible et il peut être réaliste d'imaginer qu'un jour, la signature de votre certificat développeur par une AC devienne obligatoire pour diffuser une application sur le Market Android. Rien n'a été annoncé dans ce sens pour l'instant, mais cela semble logique, voire souhaitable à terme.

Bien entendu, lorsque vous testez votre application, il peut être contraignant de procéder à l'étape de signature à chaque construction du paquet. Vous pouvez donc vous passer de cette manipulation en utilisant la commande **`ant debug`** à la racine de votre projet. L'application sera alors signée avec le certificat de développement du SDK :

```
% ant debug
Buildfile: /mnt/mega20/SRC/Java/Android/LefHello/build.xml
    [setup] Android SDK Tools Revision 8
    [setup] Project Target: Android 2.1-update1
    [setup] API level: 7
    [setup]
    [setup] ------------------
    [setup] Resolving library dependencies:
    [setup] ------------------
    [setup] Ordered libraries:
    [setup] ------------------adb tcpip
    [setup]
    [setup]
    [setup] Importing rules file: tools/ant/ant_rules_r3.xml
[...]
debug:
     [echo] Running zip align on final apk...
     [echo] Debug Package: /mnt/mega20/SRC/Java/Android/LefHello/bin/
     LefHello-debug.apk

BUILD SUCCESSFUL
Total time: 2 seconds
```

Si tout se passe correctement, et c'est généralement le cas avec un exemple « Hello World », vous obtenez un fichier **`LefHello-debug.apk`** que vous pourrez installer sur un smartphone Android ou dans une machine émulée par le SDK (AVD). Cette installation peut se faire de bien des manières.

Par exemple :

- copie sur un serveur web et accès depuis le navigateur du smartphone ;
- copie du paquet sur la carte SD du téléphone ;
- installation directe via l'outil **adb** par la connexion USB ou Wireless ;
- copie du paquet sur la carte SD via **adb** et installation depuis le téléphone.

Personnellement, j'ai une petite préférence pour l'utilisation d'une application Android appelée adbWireless de MrSiir (ou éventuellement ADB over Wifi Widget de Mehdy Bohlool). Il s'agit d'un widget permettant, d'une simple pression (clic), de relancer le serveur ADB du smartphone en le mettant à l'écoute de la connexion Wifi. C'est exactement comme si vous utilisiez **adb tcpip 5555** après avoir connecté votre smartphone en USB/Devel, mais en plus rapide.

Après avoir vu votre première application afficher votre IMEI et IMSI sur le magnifique écran AMOLED de votre Galaxy S et ainsi constaté que tout fonctionne correctement, il est temps d'en faire une vraie application. J'entends par là le fait de signer votre paquet avec votre propre certificat et non celui du SDK.

La première phase, vous vous en doutez, consiste à produire un certificat autosigné exactement de la même manière que vous le feriez pour un serveur web intranet. Nous n'utiliserons pas ici la commande **openssl**, mais **keytool** livrée avec le JDK :

```
% keytool -genkey -v -keystore lef-release-key.keystore \
-alias lefinnois -keyalg RSA -keysize 2048 -validity 10000
Tapez le mot de passe du Keystore : ***********
Ressaisissez le nouveau mot de passe : ***********
Quels sont vos prénom et nom ?
  [Unknown] :  Denis Bodor
Quel est le nom de votre unité organisationnelle ?
  [Unknown] :  Lab
Quelle est le nom de votre organisation ?
  [Unknown] :  Lefinnois Labs
Quel est le nom de votre ville de résidence ?
  [Unknown] :  Colmar
Quel est le nom de votre état ou province ?
  [Unknown] :  Alsace
Quel est le code de pays à deux lettres pour cette unité ?
  [Unknown] :  FR
Est-ce CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR ?
  [non] :  o

Génération d'une paire de clés RSA de 2 048 bits et d'un certificat
autosigné (SHA1withRSA) d'une validité de 10 000 jours
pour : CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
Spécifiez le mot de passe de la clé pour <lefinnois>
   (appuyez sur Entrée s'il s'agit du mot de passe du Keystore) :
[Stockage de lef-release-key.keystore]
```

Les options utilisées en argument de **keytool** sont relativement explicites. Notez cependant que la période de validité, exprimée en jours, doit définir une date après le 22 octobre 2033 pour que votre application puisse être diffusée sur le *Market*. Autre élément à prendre en compte et généralement source de problèmes : votre horloge doit être à jour ! Une différence de quelques secondes entre votre machine de développement et votre smartphone et vous vous retrouvez avec une application signée dans le futur, qui ne pourra donc pas être installée.

Votre clé, stockée dans le fichier spécifié par **-keystore** (qui peut en contenir plusieurs), doit être protégée comme toute clé privée qui se respecte. Pour l'utiliser via **ant**, vous devrez éditer le fichier **build.properties** à la racine de votre projet et ajouter les lignes :

```
key.store=/home/denis/lef-release-key.keystore
key.alias=lefinnois
```

Vous précisez ainsi le fichier de stockage des clés et l'alias de la clé à utiliser. Dès lors, il ne vous reste plus qu'à produire une version *release* de votre application avec :

```
% ant release
Buildfile: /mnt/mega20/SRC/Java/Android/LefHello/build.xml
    [setup] Android SDK Tools Revision 8
    [setup] Project Target: Android 2.1-update1
    [setup] API level: 7
[...]
-package-release:
[apkbuilder] Creating LefHello-unsigned.apk for release...

-release-prompt-for-password:
    [input] Please enter keystore password
     (store:/home/denis/lef-release-key.keystore):
MOT_DE_PASSE_VISIBLE
    [input] Please enter password for alias 'lefinnois':
MOT_DE_PASSE_VISIBLE

-release-nosign:

release:
     [echo] Signing final apk...
  [signjar] Signing JAR:
  /mnt/mega20/SRC/Java/Android/LefHello/bin/LefHello-unsigned.apk
  to
  /mnt/mega20/SRC/Java/Android/LefHello/bin/LefHello-unaligned.apk as
  lefinnois
     [echo] Running zip align on final apk...
     [echo] Release Package:
  /mnt/mega20/SRC/Java/Android/LefHello/bin/LefHello-release.apk

BUILD SUCCESSFUL
Total time: 10 seconds
```

Le mot de passe vous est demandé deux fois, la première pour l'accès au fichier et la seconde pour l'utilisation de la clé. Notez que ces mots de passe s'affichent sur l'écran lors de la saisie, ce qui n'est pas très courant dans ce domaine, et pour cause, ce n'est pas une très bonne chose ! Faites donc attention à qui se trouve autour de vous à ce moment.

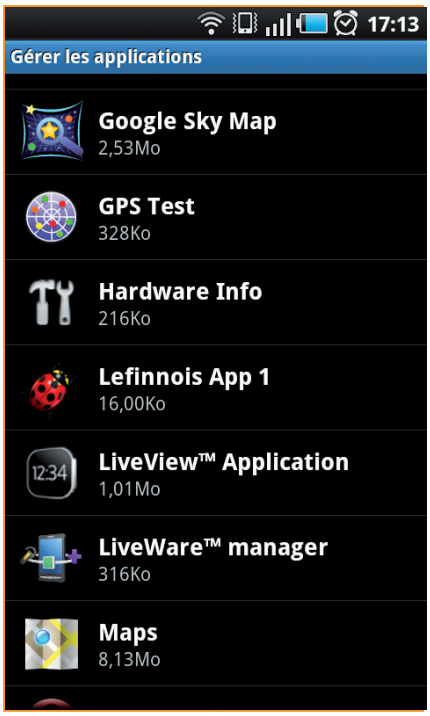
Une fois l'application signée, il ne vous reste plus qu'à l'installer comme précédemment. Il est possible de facilement vérifier une signature sur un fichier **.apk** en utilisant, encore une fois, un outil livré avec le JDK :

```
% jarsigner -verify -verbose -certs \
/mnt/mega20/SRC/Java/Android/LefHello/bin/LefHello-release.apk

        436 Tue Nov 23 10:29:18 GMT 2010 META-INF/MANIFEST.MF
        557 Tue Nov 23 10:29:18 GMT 2010 META-INF/LEFINNOI.SF
       1327 Tue Nov 23 10:29:18 GMT 2010 META-INF/LEFINNOI.RSA
sm     7408 Tue Nov 23 11:29:12 GMT 2010 res/drawable/ic_launcher_lefhello.png

     X.509, CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
     [certificate is valid from 03/11/10 12:54 to 21/03/38 12:54]

sm      704 Tue Nov 23 10:29:12 GMT 2010 res/layout/main.xml

     X.509, CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
     [certificate is valid from 03/11/10 12:54 to 21/03/38 12:54]

sm     1704 Tue Nov 23 10:29:12 GMT 2010 AndroidManifest.xml

     X.509, CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
     [certificate is valid from 03/11/10 12:54 to 21/03/38 12:54]

sm     1008 Tue Nov 23 11:29:12 GMT 2010 resources.arsc

     X.509, CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
     [certificate is valid from 03/11/10 12:54 to 21/03/38 12:54]

sm     2548 Tue Nov 23 10:29:10 GMT 2010 classes.dex

     X.509, CN=Denis Bodor, OU=Lab, O=Lefinnois Labs, L=Colmar, ST=Alsace, C=FR
     [certificate is valid from 03/11/10 12:54 to 21/03/38 12:54]


  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.
```

Nous pouvons constater que chaque élément qui compose le paquet est signé, comme avec n'importe quel fichier **.jar**, qui n'est rien d'autre qu'un fichier ZIP. Il est d'ailleurs très simple de désarchiver le paquet avec **unzip** pour en découvrir le contenu. On retrouve ainsi les métainformations (signatures), le manifeste Android, les ressources (images, XML) et le fichier binaire **.dex** (Dalvik EXecutable) correspondant à l'application exécutée par Dalvik, la machine virtuelle Android. Si vous souhaitez en savoir plus sur ce format de fichier, rendez-vous sur http://www.dalvikvm.com/.

Conclusion : quoi déjà ?!

Eh oui, le but de cet article n'était pas de vous apprendre à développer pour Android, mais de simplement vous exposer la manière de débuter dans ce domaine en vous passant d'un IDE largement utilisé sous d'autres environnements. Nous constatons finalement que, même si les facilités offertes par le greffon Android pour Eclipse sont bien pratiques, il n'est pas très difficile de s'en passer. Libre à vous, ensuite, d'adapter votre éditeur de code (Vim ou Emacs) pour ce type de développements. Il en va de même pour votre machine de développement. Avec une bonne organisation et les outils classiques Unix, vous arriverez sans le moindre doute à explorer Android sans renier vos origines et vos valeurs.



Notre application de test pour une utilisation du SDK sans Eclipse est simpliste, inutile et potentiellement boguée (essayez-la sur un Archos70), mais c'est toujours un plaisir de voir l'un de ses premiers codes dans la liste des application installées...

Je terminerai en précisant que la rédaction de cet article a été l'occasion pour moi de faire connaissance avec Android, mais aussi d'approcher un langage que je n'avais pas eu à utiliser depuis des années : Java. Il est vrai qu'une fois passée la phase « Java pour l'embarqué, c'est n'importe quoi », on est forcé de reconnaître que le travail de Google est admirable. On ne peut cependant s'empêcher, par moment, de se demander quelles pourraient être les performances d'un tel système si tout cela était basé sur des technologies à la fois plus classiques et plus novatrices. Je ne sais pas moi, un micro noyau, une libc légère, et des bibliothèques graphiques performantes comme les EFL. Le tout avec quelques *bindings* sympathiques pour Python et Lua, par exemple. Certes, la communauté de développeurs serait peut-être moins importante et l'engouement des constructeurs plus modéré. On ne le saura jamais... ■

MISE AU POINT À DISTANCE AVEC GDB ET QEMU

par Pierre Ficheux

La mise au point et l'optimisation sont des aspects fondamentaux du développement de logiciels industriels. En effet, les contraintes de qualité de fonctionnement y sont bien plus importantes que dans le cas des logiciels généralistes. Alors qu'un outil classique est utilisé au maximum quelques heures par jour, le logiciel embarqué doit fonctionner 24h/24, et le redémarrage du système est peu fréquent. La moindre fuite de mémoire ou écrasement de mémoire alloué, tolérables sur un logiciel généraliste, deviennent une véritable catastrophe dans le cas d'un logiciel embarqué.

Dans cet article, nous allons présenter quelques techniques de mise au point d'un système cible à partir d'un poste de développement PC/x86 sous Linux utilisant GDB. Nous aborderons tout d'abord le cas simple de la mise au point en espace utilisateur, puis nous verrons enfin l'utilisation de l'émulateur QEMU pour la mise au point du noyau Linux, de modules dynamiques (.ko) ou d'un *bootloader* comme U-Boot. Enfin, nous verrons comment utiliser l'option KGDB du noyau Linux pour la mise au point en espace noyau sous QEMU.

L'article utilise comme cible un environnement ARM9 émulé par QEMU (carte Versatile PB). Le compilateur utilisé peut être la version ELDK-4.2 de DENX Software ou un compilateur croisé produit par Crosstool-NG, dont le préfixe est du type arm-linux-.

Les exemples évoqués sont disponibles à l'adresse http://pficheux.free.fr/articles/lmf/remote_debug/exemples.tgz.

1 Mise au point à distance avec GDB

En cas de problème de fonctionnement, l'utilisation d'un débogueur symbolique comme GDB reste l'ultime solution. Cet outil est universellement utilisé, même en dehors du monde du logiciel libre. En effet, il est fourni depuis longtemps sur bon nombre de versions propriétaires d'Unix. On peut l'utiliser en mode texte, mais il peut également être piloté par une interface graphique. Notons qu'il est – tout comme GCC – fourni avec Mac OS X dans l'environnement de développement Xcode, que ce soit pour le développement natif ou le développement croisé pour iPhone ou iPad. GDB dispose également d'un protocole de communication permettant d'utiliser un agent distant installé sur la cible, la partie principale de GDB étant sur le poste de développement. Grâce à ce protocole, GDB peut être utilisé pour mettre au point des programmes en mode utilisateur (grâce à l'agent `gdbserver`), mais il peut également permettre la mise au point en espace noyau. Les curieux pourront trouver la description détaillée du protocole GDB à l'adresse http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html.

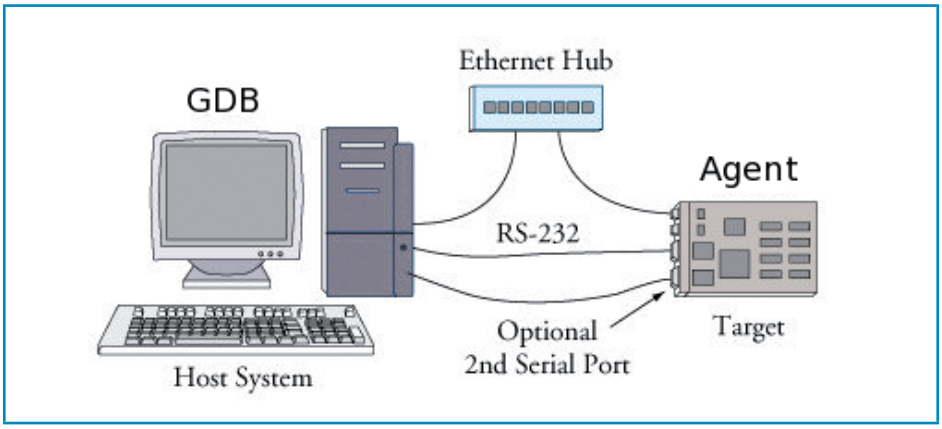


Figure 1 : Mise au point à distance avec GDB

La figure 1 schématise la configuration à mettre en place dans le cas de la mise au point à distance, appelée également *remote debugging*. Le débogueur GDB fonctionnant sur le poste de développement utilise un agent installé sur la cible.

En résumé, les cas à envisager sont les suivants :

1. L'utilisation de **gdbserver** pour la mise au point d'un programme sur la cible en espace utilisateur. Ce cas est le plus simple à mettre en œuvre et ne présente pas de difficulté. Nous rappellerons sa mise en place pour mémoire en guise d'introduction.
2. L'utilisation d'un environnement émulé en espace utilisateur (QEMU ou UML) ou de KGDB pour la mise au point du noyau, d'un module dynamique ou du bootloader. Ce cas constitue la majeure partie de cet article.
3. L'utilisation d'une sonde matérielle JTAG. Cette solution est la plus simple et efficace pour l'espace noyau, mais elle est également la plus onéreuse. Elle ne sera pas traitée dans cet article.

Remarque

Dans cet article, nous n'étudierons pas l'utilisation courante de gdb, à part les commandes dédiées à un environnement embarqué. Pour l'utilisation courante, nous encourageons le lecteur à consulter la documentation en ligne sur http://sourceware.org/gdb/current/onlinedocs.

2 Mise au point avec gdbserver

L'agent **gdbserver** – fourni avec les sources de GDB – permet de mettre au point à distance un programme en espace utilisateur. On peut également produire l'exécutable **gdbserver** dans Buildroot en le précisant au niveau du menu ***Toolchain***. La taille de l'exécutable est très raisonnable (environ 60 kilooctets), et l'on peut donc l'installer sans hésitation de manière permanente sur la cible, ce qui permet d'effectuer la mise au point à tout moment.

Du côté du poste de développement, il faut utiliser une version croisée de la commande **gdb** qui, comme pour GCC, est nommée **arm-linux-gdb** (ou un nom équivalent) dans le cas d'une architecture ARM. Cette commande est donc un exécutable x86, capable de charger en mémoire et de mettre au point un exécutable ARM. Elle peut être utilisée directement en mode texte ou au travers d'un EDI (Environnement de Développement Intégré).

On peut être tenté d'utiliser la commande **gdb** directement sur la cible, qui peut être réelle ou exécutée dans un émulateur comme QEMU. C'est possible, bien entendu, mais l'exécutable **gdb** n'a rien à voir en taille et en consommation mémoire avec **gdbserver**. Il est donc probable que l'on tombe tôt ou tard sur des problèmes d'utilisation si la cible est faiblement dimensionnée en mémoire. De même – sauf si la cible dispose d'une interface graphique – il ne sera pas possible dans ce cas d'utiliser un *frontend* graphique et encore moins un EDI (Environnement de Développement Intégré).

Concernant l'application à mettre au point, il faut suivre la procédure ci-après :

1. Compiler l'application avec l'option **-g** correspondant à l'ajout des informations de débogage.
2. Copier cette application sur la cible (ou utiliser un montage NFS).
3. Côté cible, démarrer l'application en utilisant **gdbserver** ; lors de cette étape, on précise si le lien avec le poste de développement est une ligne série RS-232 ou une connexion TCP sur un port à choisir. Dans notre cas, l'adresse du PC de développement est 192.168.3.109 et le port vaut 9999.
4. Côté poste de développement, ouvrir l'application en utilisant **arm-linux-gdb**.
5. Établir la communication avec **gdbserver** en fonction du mode de communication choisi au point 3, l'application est alors prête à être mise au point ; l'adresse de la cible est 192.168.3.50 dans notre cas, et le port est bien entendu 9999.

```
# gdbserver 192.168.3.109:9999 myprog
Process myprog created; pid = 12810
```

Le numéro de port TCP à choisir ne doit pas être déjà utilisé par un autre service. En pratique, il est fortement déconseillé d'utiliser une valeur de port inférieure à 1024. De même, on devra vérifier qu'aucun pare-feu ne bloque le port choisi entre la cible et le poste de développement.

```
$ arm-linux-gdb myprog
GNU gdb Red Hat Linux (6.7-2rh)
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-linux".
The target architecture is set automatically (currently arm)
..
(gdb) target remote 192.168.3.50:9999
Remote debugging using 192.168.3.50:9999
(gdb)
```

À partir de là, on peut poser des points d'arrêt et poursuivre la mise au point du programme. Vu que le programme est déjà démarré, on reprend son exécution en utilisant la commande **continue** et non pas **run**. À la fin de l'exécution du programme,

on note que **gdbserver** se termine, et l'on devra donc le relancer pour effectuer une nouvelle session de mise au point.

Si l'on utilise un port série, on doit remplacer la spécification de l'adresse IP et du port par le nom du fichier spécial correspondant au port série. Attention, le nom du fichier n'est pas forcément le même côté cible et coté PC de développement. On pourra avoir, côté cible, la commande suivante :

```
# gdbserver /dev/ttyAMA0 myprog
```

puis la commande :

```
(gdb) target remote /dev/ttyS0
```

côté PC de développement, qui utilise un fichier spécial différent pour le port série, puisque ce n'est pas la même architecture.

3 Utilisation de l'émulateur QEMU (espace noyau)

Selon les dires des mainteneurs actuels de KGDB (voir http://kernel.org/pub/linux/kernel/people/jwessel/dbg_webinar/kernel_debugging_tools_webinar.pdf), la solution de l'émulateur est plus intéressante que KGDB, à condition bien entendu que la qualité de l'émulation soit suffisamment bonne. Les deux solutions les plus utilisées sont QEMU et UML. La deuxième correspond à *User Mode Linux* (voir http://user-mode-linux.sourceforge.net), qui est un patch du noyau Linux permettant d'exécuter l'espace utilisateur en question. Nous ne détaillerons pas cette solution sachant que QEMU nous semble mieux adapté à un environnement embarqué.

Nous avons déjà utilisé QEMU en tant qu'outil de test de distribution embarquée dans les deux articles concernant Buildroot (été 2010). Dans l'exemple qui suit, nous considérons que la distribution testée est du même type, soit une architecture ARM9 sur carte émulée de type Versatile PB.

QEMU intègre un agent GDB qui peut être facilement utilisé pour mettre au point le noyau démarré par QEMU ainsi que des modules dynamiques. Si l'on considère un environnement identique à celui des articles sur Buildroot, on pourra démarrer l'émulation en mode « mise au point » en ajoutant les options **-s** et **-S** à la ligne de commandes.

La première option commande l'activation de l'agent GDB, et la deuxième indique de « geler » l'émulation du processeur au démarrage. L'exécution reprendra lors de la connexion de GDB (**arm-linux-gdb** dans notre cas) à l'agent intégré à QEMU. Le port TCP utilisé par défaut par l'agent est 1234, mais on peut le modifier en utilisant l'option **-p**. Dans un premier temps, nous allons tester la mise au point avec un noyau Linux, puis nous ferons un test équivalent avec une image U-Boot.

Les méthodes de mise au point étant basées sur GDB, il est nécessaire de compiler le programme à mettre au point avec l'option **-g**. Dans le cas du noyau, il suffit d'activer l'option ***Kernel hacking*** puis ***Compile the kernel with debug info*** lors de la configuration du noyau avec **make menuconfig**, **xconfig** ou **gconfig**.

3.1 Mise au point du noyau Linux

Dans une fenêtre d'émulation de terminal, on démarre QEMU comme nous l'avons fait précédemment, en ajoutant les options liées à l'agent GDB. Nous utilisons une console texte (option **-nographic**), mais le principe fonctionne de la même manière en utilisant une émulation du *framebuffer*. La ligne de commandes est la suivante :

```
$ qemu-system-arm -M versatilepb -m 64 -kernel zImage -initrd rootfs.arm.cpio -append \
"console=ttyAMA0" -nographic -s -S
```

Dans un autre émulateur de terminal, nous démarrons la commande **arm-linux-gdb** sur le fichier vmlinux qui correspond à la version non compressée du noyau adapté à la cible. Nous posons ensuite un point d'arrêt sur la fonction **start_kernel** définie dans le fichier **init/main.c**, puis nous effectuons la connexion à l'agent de QEMU. Vu que nous sommes sur le même système hôte que QEMU (le PC/x86 de développement), le nom à utiliser est *localhost*. L'exécution s'interrompt au niveau de la fonction et nous pouvons exécuter le noyau pas à pas, à l'aide de la commande **next** de GDB. Finalement, nous continuons l'exécution du noyau par la commande **continue**, ce qui a pour effet de terminer le démarrage du système côté QEMU.

```
$ arm-linux-gdb  vmlinux
GNU gdb Red Hat Linux (6.7-2rh)
Copyright (C) 2007 Free Software Foundation, Inc.License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-linux".
The target architecture is set automatically (currently arm)
..
(gdb) b start_kernel
Breakpoint 1 at 0xc00088f4: file init/main.c, line 541.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: Can not parse XML target description; XML support was disabled at compile time
[New Thread 1]
0x00000000 in ?? ()
(gdb) c
Continuing.
Breakpoint 1, start_kernel () at init/main.c:541
541             smp_setup_processor_id();
(gdb) n
537     {
(gdb) n
```

```
541             smp_setup_processor_id();
(gdb) n
557             local_irq_disable();
(gdb) n
559             early_init_irq_lock_class();
(gdb) n
566             tick_init();
(gdb) c
Continuing.
```

Remarque

Il est fréquent de devoir interrompre l'exécution de GDB afin de poser de nouveaux points d'arrêt. Une solution est de définir un point d'arrêt sur la fonction **sys_sync** avec la commande **break sys_sync**. L'exécution du noyau sera donc interrompue lorsque l'utilisateur tape **sync** sur la ligne de commandes. Il est également possible de taper **Ctrl-C** dans GDB pour interrompre l'exécution.

De même, placer un point d'arrêt sur la fonction **panic** permettra de trouver la cause d'un arrêt brutal du noyau en explorant la pile d'appel.

3.2 Mise au point d'un module dynamique

Nous allons maintenant décrire le cas de la mise au point d'un module ajouté par l'utilisateur. Notre exemple est très simple, puisqu'après le chargement du module **helloworld.ko**, il propose simplement d'afficher une chaîne de caractères en utilisant la commande **cat /proc/helloworld**. La fonction **helloworld_read_proc** est liée à la lecture de **/proc/helloworld** par l'appel à la fonction **create_proc_entry** lors de l'initialisation du module. Le code source du module de test est donné ci-après.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>

static in n = 0;

static int helloworld_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data)
{
  n++;
  return sprintf(page, "read_proc: Hello World %d\n", n);
}

static int __init hello_init(void)
{
  struct proc_dir_entry *entry;

  entry = create_proc_entry("helloworld", 0, NULL);
  if (entry)
    entry->read_proc = helloworld_read_proc;

  printk(KERN_INFO "Hello World\n");

  return 0;
}

static void __exit hello_exit(void)
{
  remove_proc_entry ("helloworld", NULL);
  printk(KERN_INFO "Goodbye, cruel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Pour compiler ce module, on utilise un Makefile adapté, le point important étant la référence au noyau Linux 2.6.30 utilisé pour le test. Lorsque c'est possible, il est préférable de compiler le module sans optimisation (option **-O0**) afin de limiter des problèmes de mise au point. Par défaut, les modules du noyau sont compilés avec l'option **-O2**. Pour cela, nous modifions la variable **EXTRA_CFLAGS** dans le Makefile.

```
# Makefile for helloworld module
#
KDIR= $(HOME)/test/linux-2.6.30
PWD= $(shell pwd)

EXTRA_CFLAGS += -O0

obj-m := helloworld.o

all:
  $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
  rm -f *~
  $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

Après compilation du module par **make**, on obtient le fichier **helloworld.ko** que l'on doit ajouter au *root filesystem* de test, par exemple en le chargeant via le réseau avec la commande **wget**. On doit également copier **helloworld.ko** dans le répertoire où se trouve vmlinux, afin que GDB y ait accès.

La prochaine étape consiste à démarrer le système avec QEMU, comme dans le paragraphe précédent. La version standard de GDB ne peut pas connaître à l'avance les adresses des différentes sections ou segments du module (**.text**, **.data**, **.bss**, **.rodata**), puisque celles-ci sont dynamiques. On doit donc charger le module par **insmod**, puis récupérer les adresses des sections afin de les transmettre à GDB. Ces sections correspondent aux différents éléments constituant un programme exécutable ou un module. Le code exécutable est situé dans la section **.text**, les variables modifiables - globales et statiques - dans **.data**, les variables non initialisées sont dans le segment **.bss** et les variables en lecture seule dans **.rodata**. On peut en savoir plus sur cette notion de section en lisant la page http://en.wikipedia.org/wiki/Data_segment.

Lorsque le module est chargé, ces informations sont disponibles dans les fichiers virtuels **/sys/module/helloworld/sections/<nom_de_section>**. On doit donc charger le module avant de pouvoir lire les valeurs des adresses, puis interrompre l'exécution de GDB en tapant la commande **sync**.

```
# insmod helloworld.ko
Hello World
# cat /sys/module/helloworld/sections/.text
0xbf000000
# cat /sys/module/helloworld/sections/.text
0xbf000000
# cat /sys/module/helloworld/sections/.data
0xbf00052e
# cat /sys/module/helloworld/sections/.bss
0xbf000660
# cat /sys/module/helloworld/sections/.rodata
0xbf0000ac
# sync
```

Du côté de GDB, on ajoute les valeurs des adresses en les associant au nom du module par la commande **add-symbol-file**. On définit ensuite un point d'arrêt sur la fonction **helloworld_read_proc**. Après ajout des adresses des sections, le nom de la fonction **helloworld_read_proc** est connu de GDB, et on peut donc utiliser la complétion automatique disponible avec la touche TAB pour le saisir. Pour finir, on poursuit l'exécution en tapant **continue**.

```
(gdb) add-symbol-file helloworld.ko 0xbf000000 -s .data 0xbf00052e -s
.bss 0xbf000660 -s .rodata 0xbf0000ac
add symbol ta ble from file "helloworld.ko" at
  .text_addr = 0xbf000000
  .data_addr = 0xbf00052e
  .bss_addr = 0xbf000660
  .rodata_addr = 0xbf0000ac
(y or n) y
Reading symbols from /home/pierre/chap11/linux-2.6.30/helloworld.ko...done.

(gdb) b helloworld_read_proc
Breakpoint 2 at 0xbf000018: file /home/pierre/chap11/hello_world/
helloworld.c, line 9.
(gdb) c
Continuing.
```

Lorsque l'on tape la commande **cat /proc/helloworld**, l'exécution du noyau est de nouveau interrompue et l'on peut continuer en mode pas à pas. On affiche la valeur de la variable n, puis on la modifie avant de continuer l'exécution qui provoque l'affichage du message **read_proc: Hello World 10** dans le terminal.

```
Breakpoint 2, helloworld_read_proc (page=0xc3451000 "", start=0xc32a9f30,
    off=0, count=3072, eof=0xc32a9f34, data=0x0)
    at /home/pierre/chap11/hello_world/helloworld.c:9
9   n++;
(gdb) p n
$1 = 0
(gdb) n
11    return sprintf(page, "read_proc: Hello World %d\n", n);
(gdb) p n
$2 = 1
(gdb) p n=10
$3 = 10
(gdb) c
Continuing.

Breakpoint 2, helloworld_read_proc (
    page=0xc3451000 "read_proc: Hello World 10\n", start=0xc32a9f30, off=26,
    count=3072, eof=0xc32a9f34, data=0x0)
    at /home/pierre/chap11/hello_world/helloworld.c:9
9   n++;
(gdb) c
Continuing.
```

Les manipulations ci-dessus sont assez complexes, et on remarque que l'on ne peut pas poser de point d'arrêt dans la fonction d'initialisation du module **hello_init**, car le module doit être chargé afin d'obtenir les adresses des sections. Certaines versions modifiées de GDB permettent de poser un point d'arrêt *pending* (en attente) qui est activé lorsque le module est chargé et l'adresse de la fonction résolue. La variable **solib-search-path** permet de donner le chemin d'accès aux modules à mettre au point. Une telle version de GDB est disponible avec la distribution STLinux pour SH4, voir http://www.stlinux.com/devel/debug/kgdb/modules. Dans ce cas-là, on utilise la séquence de commandes suivante dans la session GDB :

```
(gdb) set breakpoint pending on
(gdb) set solib-search-path /user/my_module_2.6
Reading symbols from /user/my_module_2.6/my_mod.ko...
expanding to full symbols...done.
Loaded symbols for /user/my_module_2.6/my_mod.ko
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0xc0168000  0xc01680c0  Yes         /user/my_module_2.6/my_mod.ko

(gdb) b my_module_init
Function "my_module_init" not defined
Breakpoint 1 (my_module_init) pending.
```

3.3 Mise au point du bootloader U-Boot

Le cas de la mise au point de U-Boot sous QEMU est encore plus intéressant, car il n'est pas possible de l'effectuer avec KGDB, qui concerne uniquement le noyau Linux.

Le nom U-Boot signifie *Universal Bootloader*, car il peut être utilisé sur la majorité des architectures embarquées du marché (ARM, PowerPC, MIPS, SH4, etc.). Le projet est développé par la société DENX Software, déjà à l'origine de la distribution ELDK, et dont nous utilisons le compilateur croisé. La documentation complète de U-Boot est disponible sur http://www.denx.de/wiki/DULG/Manual.

L'utilisation complète de U-Boot - avec émulation de la mémoire flash - dans QEMU nécessite de légères modifications sous forme de patches appliqués aux sources de U-Boot (version 1.3.4) et de QEMU. Ces patches sont issus des travaux publiés par Thomas Petazzoni en 2008, et disponibles à

l'adresse http://thomas.enix.org/Blog-20081002153859-Technologie. La version adaptée à un QEMU récent est disponible en téléchargement avec les compléments de l'article.

Dans notre cas, U-Boot est exécuté en RAM, donc aucun patch n'est nécessaire. Le test devrait fonctionner sur les versions récentes d'U-Boot, mais nous avons uniquement testé la version 1.3.4.

3.3.1 Compilation de U-Boot

On peut compiler la version 1.3.4 modifiée de U-Boot en utilisant la procédure suivante. Bien entendu, il faut disposer de l'environnement de compilation ARM avec la variable **CROSS_COMPILE** affectée à la valeur **arm-linux-**.

Le bootloader est compilé par défaut avec l'option **-g** nécessaire pour l'utilisation de GDB. On peut vérifier cela grâce à la valeur de la variable **DBGFLAGS** dans le fichier **config.mk**. La compilation s'effectue comme décrit ci-dessous :

```
$ tar xjvf u-boot-1.3.4.tar.bz2
$ cd u-boot-1.3.4
$ make versatile_config
Configuring for versatile board... Variant::
PB926EJ-S
$ make
```

À l'issue de la compilation, on obtient l'image U-Boot sous trois formes :

- une image binaire brute, soit **u-boot.bin** ;
- une image binaire au format ELF (*Executable and Linkable Format*), soit **u-boot** ;
- une image ASCII au format Motorola S-Records, soit **u-boot.srec**.

3.3.2 Exécution dans QEMU

Pour exécuter U-Boot dans QEMU, on utilise la commande suivante, sans oublier d'utiliser les options **-s** et **-S**. L'option **-kernel** permet de passer le nom de l'image U-Boot exécutable.

```
$ qemu-system-arm  -M versatilepb -m 64 \
-kernel u-boot.bin -nographic -s -S
```

On peut ensuite exécuter **arm-linux-gdb** sur le binaire **u-boot**. Attention, le fichier **u-boot.bin** n'est pas utilisable côté Linux, car contrairement à **u-boot**, ce dernier n'est pas au format ELF, qui est le seul format binaire utilisable par la commande **arm-linux-gdb**.

Nous posons tout d'abord un point d'arrêt sur la fonction **start_armboot**, ce qui nous permet de démarrer le bootloader en mode pas à pas. Nous définissons ensuite un autre point d'arrêt sur la fonction **main_loop**, qui correspond à la boucle interactive de saisie des commandes. Nous pouvons alors afficher la valeur de la variable **bootdelay** avec **print**, puis continuer l'exécution de U-Boot en utilisant la commande **continue**.

```
$ arm-linux-gdb  u-boot
GNU gdb Red Hat Linux (6.7-2rh)
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-linux".
The target architecture is set automatically (currently arm)
..
(gdb) b start_armboot
Breakpoint 1 at 0x10004e4: file board.c, line 315.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: Can not parse XML target description; XML support was disabled at compile time
[New Thread 1]
0x00000000 in ?? ()
(gdb) c
Breakpoint 1, start_armboot () at board.c:315
315             gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
(gdb) n
304     {
(gdb) n
315             gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
(gdb) n
319             memset ((void*)gd, 0, sizeof (gd_t));
(gdb) n
320             gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
(gdb) n
321             memset (gd->bd, 0, sizeof (bd_t));
(gdb) n
323             monitor_flash_len = _bss_start - _armboot_start;
(gdb) n
304     {
(gdb) n
323             monitor_flash_len = _bss_start - _armboot_start;
(gdb) n
...
(gdb) b main_loop
Breakpoint 2 at 0x100618c: file main.c, line 361.
(gdb) n
362  bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
(gdb) n
384    s = getenv ("bootcmd");
(gdb) p bootdelay
$1 = 2
(gdb) c
Continuing.
```

3.4 Utilisation de KGDB

Historiquement, le patch KGDB fut la première solution libre utilisable pour la mise au point en espace noyau. Pendant longtemps, l'utilisation de KGDB passa par l'application d'un patch maintenu par la société LinSysSoft Technologies

(http://www.linsyssoft.com). La version gratuite était disponible uniquement pour certaines versions du noyau. Pour les autres versions, on devait obtenir la version payante, ce qui rendait la solution peu intéressante par rapport au choix d'une sonde JTAG. Deux articles très complets ont déjà été rédigés en 2008 pour *Linux Magazine* et sont cités en bibliographie.

Le patch KGDB a été repris récemment par les développeurs de Wind River (Intel), en collaboration avec LinSysSoft, et intégré à la version standard (ou *main line*) du noyau Linux, et ce, à partir de la version 2.6.26. Une nouvelle version devrait être intégrée à la version 2.6.35. Le wiki de KGDB est disponible sur http://kgdb.wiki.kernel.org.

Pour utiliser KGDB, on doit tout d'abord valider l'option ***Kernel hacking*** puis ***KGDB*** : *kernel debugging with remote gdb* dans la configuration du noyau Linux. Dans le cas précédent, nous avons utilisé QEMU en tant que serveur GDB, mais le noyau n'était pas configuré pour utiliser KGDB (option **CONFIG_KGDB** désactivée). Le fonctionnement actuel est différent, puisque nous utilisons QEMU dans le seul but d'exécuter un noyau Linux compatible KGDB. Le même test pourrait bien entendu être fait avec une carte ARM réelle à la place de QEMU. Nous remarquons d'ailleurs que les options **-s** et **-S** ne sont plus utilisées.

Il existe plusieurs méthodes pour mettre au point le noyau avec KGDB. Les différentes options sont en général passées en paramètres au noyau Linux, mais on peut également les modifier en utilisant des entrées dans **/sys**. On peut donc :

1. Utiliser un port série avec l'option **kgdboc** (pour *KGDB Over Console*). C'est la solution que nous utiliserons dans ce paragraphe. Le paramètre correspond au nom du fichier spécial associé à la ligne série, ainsi que le débit utilisé, par exemple kgdboc=ttyAMA0,115200.
2. Utiliser un lien Ethernet avec l'option **kgdboe** (pour *KGDB Over Ethernet*). Cette possibilité est décrite comme moins fiable dans la dernière documentation KGDB et la dernière version de KGDB basée sur 2.6.35 ne devrait plus intégrer cette possibilité.

En plus de **kgdboc**, il faut ajouter l'option **kgdbwait**, qui indique au noyau d'attendre la connexion du client GDB (**arm-linux-gdb** dans notre cas). Si nous voulons tester le fonctionnement avec QEMU, nous devons utiliser la console graphique sur le framebuffer, sachant que le port série associé à **/dev/ttyAMA0** est déjà utilisé par KGDB. Il est possible de partager la console série avec KGDB, mais la mise en place est plus délicate. L'utilisateur doit pour cela envoyer manuellement une combinaison « magique » de touches SysRq-g. Pour cela, nous conseillons la lecture des documentations suivantes :

http://kgdb.wiki.kernel.org

http://kernel.org/pub/linux/kernel/people/jwessel/kgdb/ch03s03.html

http://en.wikipedia.org/wiki/Magic_SysRq_key

Nous ajoutons l'option **-serial pty** à la ligne de démarrage de QEMU, afin d'émuler le port série avec un pseudo-terminal.

```
$ qemu-system-arm  -M versatilepb -m 64 -kernel zImage -initrd rootfs.arm.cpio -append \
"kgdboc=ttyAMA0 kgdbwait" -serial pty

char device redirected to /dev/pts/5
```

À l'issue de cette commande, la fenêtre graphique de QEMU est affichée. Le noyau Linux interrompt son exécution avec le message **kgdb: Waiting for connection from remote gdb**, visible dans cette fenêtre.

On démarre ensuite **arm-linux-gdb** sur le noyau vmlinux, puis on se connecte au noyau en attente en utilisant **target remote** sur le nom du pseudo-terminal utilisé. On pose ensuite un point d'arrêt sur la fonction **sys_sync**. L'utilisation de la commande **continue** provoque la fin du démarrage du système.

```
$ arm-linux-gdb vmlinux
GNU gdb Red Hat Linux (6.7-2rh)
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-linux".
The target architecture is set automatically (currently arm)
..
(gdb) b sys_sync
Breakpoint 1 at 0xc00a4eb0: file fs/sync.c, line 41.

(gdb) target remote /dev/pts/5
Remote debugging using /dev/pts/5
0xc006107c in kgdb_breakpoint () at kernel/kgdb.c:1718
1718   atomic_set(&kgdb_setting_breakpoint, 1);
(gdb) c
Continuing.
root[New Thread 324]
[Switching to Thread 324]
```

Lorsque l'on tape la commande **sync** sur le système exécuté dans QEMU, cela provoque l'exécution du point d'arrêt.

```
Breakpoint 1, sys_sync () at fs/sync.c:41
41   do_sync(1);
```

On peut alors mettre au point le noyau, comme nous l'avons fait avec QEMU en mode serveur GDB. On pourra également mettre au point un module dynamique en appliquant la méthode décrite précédemment.

Conclusion

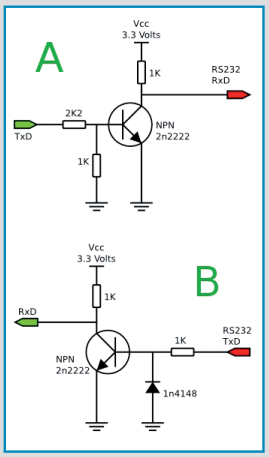
Cet article nous a permis d'évaluer les capacités de QEMU comme outil de mise au point en complément de GDB. Cet outil est utilisé entre autres dans l'environnement de développement Android. N'oublions pas non plus que QEMU peut être utilisé en émulation/virtualisation x86, ce qui permet également de mettre au point un noyau pour une cible native Intel. ■

Bibliographie

- Le projet QEMU sur http://bellard.org/qemu
- Le dépôt Git de QEMU sur http://git.savannah.gnu.org/cgit/qemu.git
- Les sources de U-Boot sur http://www.denx.de/wiki/U-Boot
- Le wiki KGDB sur https://kgdb.wiki.kernel.org/index.php/Main_Page
- Débogage du noyau Linux avec KGDB sur http://www.unixgarden.com/index.php/programmation/debogage-du-noyau-linux-avec-kgdb
- Débogage dans l'espace noyau avec KGDB sur http://www.unixgarden.com/index.php/programmation/debogage-dans-l%E2%80%99espace-noyau-de-linux-avec-kgdb
- Travaux de Thomas Petazzoni sur QEMU et U_boot sur http://thomas.enix.org/Blog-20081002153859-Technologie
- *Booting Linux with U-Boot on QEMU ARM* sur http://balau82.wordpress.com/2010/04/12/booting-linux-with-u-boot-on-qemu-arm

QUICKTIPS : RS-232 vers TTL

Les ports parallèles ont disparu depuis fort longtemps de nos très chers PC. Le port série RS-232 ou EIA232 va très certainement avoir la même destinée. Cependant, il est encore présent sur une bonne partie des configurations récentes et, bien entendu, sur bon nombre de périphériques en production.



Interconnecter un port de ce type avec une plateforme embarquée moderne, ou inversement, une carte embarquée avec un périphérique RS-232, nécessite une adaptation électrique. Bien que les interfaces modernes soient également sérielles, les tensions utilisées sont comprises entre 3.3V et 5V, où un 0 logique correspond à 0V et un 1 logique à Vcc. Côté RS-232, par contre, un niveau logique 0 est représenté par une tension de +3V à +25V et un niveau logique 1 par une tension de -3V à -25V. Sur un port série comme on en trouve dans un PC, ces tensions sont généralement de +12V et -12V avec une certaine tolérance en fonction du matériel. Une norme, V.28, précise par ailleurs qu'un 0 est reconnu si la tension est inférieure à -3 V, et 1 si la tension est supérieure à +3 V. Le tableau suivant résume les équivalences :

RS-232	TTL	Valeur logique
-15V à -3V	+2V à +5V	1
+3V à +15V	0V à +0.8V	0

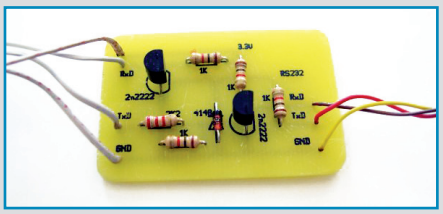
Il existe des circuits spécialisés permettant d'interfacer de tels systèmes. Les plus connus sont très certainement le MAX232, le MAX2323 et le MAX233. Ce dernier, tout particulièrement, présente l'avantage de ne pas nécessiter de composants complémentaires (condensateurs) et est donc très simple à mettre en œuvre.

Malheureusement, on n'a pas forcément ces composants sous la main au moment où on en a le plus besoin. Il existe une solution permettant, à l'aide de quelques composants, d'obtenir une interface équivalente. De plus, dans certains cas, la communication est unidirectionnelle (cas d'un récepteur GPS ou d'un capteur quelconque, par exemple), et de ce fait, le nombre de composants utiles est divisé par deux.

Le schéma présenté ici permettra d'interfacer facilement n'importe quelle interface RS-232 avec n'importe quel montage utilisant des tensions TTL. Attention, tout repose ici sur le fait que Vcc, la tension d'alimentation, est supérieure à la limite basse du 0 logique RS-232 (3V). Certaines rares interfaces séries sont en 2,8V. Dans ce cas, cela ne fonctionnera pas.

La partie A du schéma présente la conversion de tensions avec un montage/hôte/périphérique TTL émetteur et un récepteur RS-232. Le transistor NPN (ici un 2n2222) est utilisé en régime de commutation pour inverser les tensions. Ainsi, un TxD à 0V ne saturera pas le transistor et RxD sera la tension d'alimentation Vcc. Inversement, avec un TxD à 3.3V, le transistor est saturé, le courant passe entre Vcc et la masse via la résistance de 1K Ohm et RxD est à 0V, donc 1 logique pour RxD.

Le circuit B présente l'interface inverse avec une émission par RS-232 et une réception en TTL. En plus d'un transistor NPN et quelques résistances, il faut ici faire usage d'une diode type 1n4148 très courante. Lorsque TxD est à -12V, la diode laisse passer le courant et la base du transistor n'est pas excitée. Vcc est donc présenté via la résistance sur RxD. Dans le cas d'un 0 logique sur RS-232, le transistor est saturé et RxD est mis à la masse.



Cette solution vous permettra de rapidement faire dialoguer deux interfaces séries de conception différente. Remarquez toutefois qu'avec l'utilisation massive de l'USB, il sera préférable d'utiliser un adaptateur USB/Série de type FTDI, par exemple. On trouve facilement aussi bien des modèles USB/RS-232 qu'USB/série-TTL.

PLATEFORME FRIENDLYARM MINI2440

par Denis Bodor

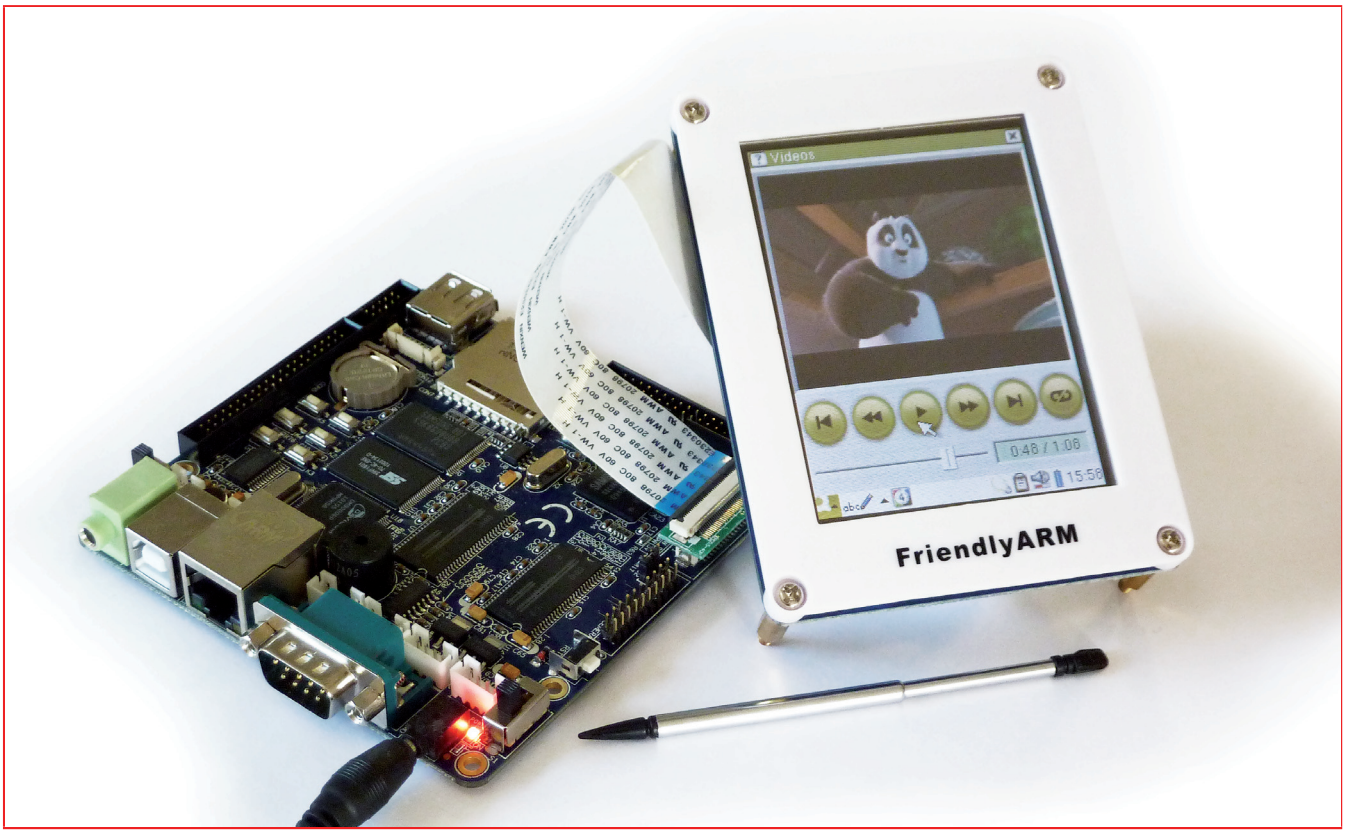
Voilà une plateforme complète ARM de plus en plus populaire. Malgré ses canaux de diffusion peu professionnels (eBay, vendeurs asiatiques, petites boutiques en ligne, etc.), elle fait son petit bonhomme de chemin et s'impose tant dans le domaine de l'expérimentation que du prototypage embarqué. Pourquoi ? Tout simplement parce qu'une carte à base d'ARM9 incluant entre autres choses 2Mo de NOR, 256Mo à 1Go de NAND, 64Mo de SDRAM et un écran tactile LCD 3"1/2 pour moins de 100$, justifie largement le fait de se passer des « filières » habituelles.

Cette plateforme, appelée mini2440, vendue par plusieurs distributeurs européens mais surtout asiatiques, est un produit conçu par FriendlyARM (www.friendlyarm.net). Les caractéristiques sont les suivantes :

- Taille : 100 x 100 mm.
- CPU : 400 MHz Samsung S3C2440A ARM920T.
- Alimentation : connecteur 5V 1A.
- RAM : 64 MB SDRAM, 32 bit 100 MHz Bus.
- Flash : 64 MB / 128 MB / 256 MB / 1GB NAND Flash et 2 MB NOR Flash.
- EEPROM intégrée : 1024 octets 24C08 (I2C).
- Mémoire externe : emplacement SD-Card/MMC SDHC.
- Ports séries : 3 ports (connecteurs TTL sur la carte), dont un avec DB9/RS232 pour console.
- Ports USB : hôte USB-A + USB-B Device.
- Audio : sortie 3,5 mm stéréo jack + micro intégré (avec connecteur) + PWM buzzer.
- Réseau : Ethernet RJ-45 10/100.
- RTC : *Real Time Clock* avec pile.
- E/S :
  - Connecteur 6x boutons + un port A/N ;
  - 4 LED utilisateur sur la carte ;
  - Extension : connecteur 40 bus système + 34 GPIO (au pas de 2.0mm) ;
  - Connecteur JTAG 10 broches (au pas de 2.0mm) ;
  - Connecteur 20 broches pour caméra ;
  - Connecteur écran FriendlyARM (3,5" et 7") et carte VGA.
- Interface pour écrans :
  - Écran STN max 1024×768, 4096 couleurs ;
  - Écran TFT max 1024×768, 64k couleurs ;
  - TouchScreen résistif 4 fils.

Les cartes commandées (chez esky-sh via eBay) pour l'élaboration de cet article se déclinent en deux versions avec 256 Mo et 1 Go de Flash NAND. Toutes deux sont livrées avec l'adaptateur d'alimentation 5V/2A, câbles Ethernet/série/USB, SDK sur DVD-RW, adaptateurs JTAG parallèles et écrans tactiles LCD 3"1/2 240×320. L'une des cartes (1 Go) était accompagnée d'une DebugBox JTAG/USB compatible OpenOCD. À titre d'information, l'ensemble, port compris, aura coûté un peu plus de 300 euros (2 x 256 Mo + 1 x 1 Go + DebugBox). On notera au passage qu'il existe également des versions disposant d'écrans tactiles de 7" pour quelques dizaines d'euros de plus. Pour des versions plus spécifiques (*form factor* différent, cartes d'expérimentations, etc.), il faudra vous tourner vers un site comme www.arm9.net.

Côté logiciel, les cartes ont été livrées avec un GNU/Linux préinstallé sur système de fichiers racine Yaffs. Cette installation disposait d'une interface Qtopia complète permettant de vérifier le fonctionnement de la plateforme (c'est ce que présente la carte en couverture du magazine). Sur le DVD-RW accompagnant les cartes des images pré-construites pour uCos2, Qtopia et WinCE 5.0 sont mises à disposition, ainsi que les chaînes



La carte intègre un grand nombre de composants et d'interfaces directement utilisables sous GNU/Linux via les SDK et/ou images fournies. La FriendlyARM mini2440 s'avère être une excellente plateforme de développement, mais saura également trouver sa place en production. Seul regret, puisque rien n'est jamais parfait, les connecteurs au pas de 2mm rendront plus délicates les interconnexions avec des modules « fait maison ».

de construction, les *datasheets* des composants, un manuel en PDF, ainsi qu'un lot de sources destinées à la reconstruction de systèmes et aux tests de la plateforme. De base, la carte permet de booter sur la NAND ou la NOR via un interrupteur. Dans les deux cas, par défaut, c'est le *bootloader* SuperVivi qui est utilisé.

Bien que la solution proposée en guise d'exemple pour cette plateforme soit parfaitement viable, l'objet du présent article est de mettre en œuvre une installation plus standard reposant sur U-Boot, emDebian et Debian GNU/Linux. En effet, l'utilisation d'un SDK et/ou d'une chaîne de développement fournie sous la forme d'une archive **`tar`** à installer en dépit d'une gestion de paquets cohérente est, bien que courante, une méthode peu souhaitable. Mieux vaut reposer sur une architecture stable, intégrée et maîtrisable.

1 Préliminaires

Avant de pouvoir construire quoi que ce soit à destination d'une plateforme différente de celle utilisée localement, il est nécessaire d'installer (ou de compiler) un environnement de développement. Celui-ci est constitué d'un compilateur croisé (*cross-compiler*) ainsi que des outils (**`binutils`**) et bibliothèques connexes.

Dans le cadre de cet article et de la plateforme ARM9/s3c2440, cet ensemble outils/bibliothèques forme la plateforme connue par Debian sous le nom « Armel » et les outils CPP/GCC **`gnueabi`**. Contrairement à ce qu'il est généralement conseillé de faire par la plupart des sites web et forums, nous n'utiliserons pas ici la chaîne de compilation croisée livrée sur DVD ou téléchargeable sous forme de tarball. Le résultat sera sensiblement le même, tout en conservant une cohérence dans notre gestion d'applications via APT. Pour ce faire, nous devons ajouter un certain nombre de dépôts dans notre **`/etc/apt/sources.list`** ou dans un fichier **`.list`** présent dans **`/etc/apt/sources.list.d`** :

```
% cat /etc/apt/sources.list.d/emdebian.list
deb http://buildd.emdebian.org/debian/ unstable main
deb-src http://buildd.emdebian.org/debian/ unstable main
deb http://buildd.emdebian.org/debian/ testing main
deb-src http://buildd.emdebian.org/debian/ testing main
deb http://buildd.emdebian.org/debian/ lenny main
deb-src http://buildd.emdebian.org/debian/ lenny main
```

Ceci fait, il ne nous reste plus qu'à mettre à jour la liste des paquets avec un petit **aptitude update**, puis à installer les éléments nécessaires à la construction de binaires à destination de la plateforme embarquée :

```
% aptitude install libc6-armel-cross \
libc6-dev-armel-cross binutils-arm-linux-gnueabi \
gcc-4.3-arm-linux-gnueabi g++-4.3-arm-linux-gnueabi
```

Remarque **ABI, EABI, OABI, GNU EABI, ...**

Une ABI pour *Application Binary Interface* permet de définir une interface de bas niveau entre les programmes binaires et un système d'exploitation. En d'autres termes, il s'agit de définir la manière de construire un environnement d'exécution pour les applications. Une telle interface détermine, par exemple, les types de données, l'alignement, les conventions pour le passage d'arguments par les fonctions et les valeurs retournées, les numéros des appels système, etc.

L'objectif est de fournir une compatibilité binaire entre plusieurs implémentations d'un système pour une architecture donnée. On peut ainsi faire fonctionner, sans modification, n'importe quel binaire compatible (application) sur n'importe quel système offrant une ABI compatible. C'est le cas, par exemple, de l'ABI iBCS (*Intel Binary Compatibility Standard*) sur plateforme Intel.

Dans le cas de l'embarqué, des spécifications, ou plus exactement une famille de spécifications décrites par ARM, permettent une implémentation de l'interface. Il faut toutefois remarquer qu'il existe deux ABI découlant des spécifications ARM EABI (*Embedded Application Binary Interface*). L'une, plus ancienne, est maintenant nommée OABI pour *Old ABI* et la nouvelle est appelée EABI. Comprenez cependant qu'OABI comme EABI découlent des mêmes spécifications.

Sous GNU/Linux, l'implémentation utilisée est une sous-ABI appelée GNU EABI. Voilà pourquoi le compilateur GCC, par exemple, que vous venez sans doute de fraîchement installer, s'appelle **arm-linux-gnueabi-gcc-4.3**. Notez également qu'une ABI décrivant l'interface entre les binaires et le système, ainsi que les options correspondantes, existe dans la configuration des sources du noyau Linux (**CONFIG_AEABI** et **CONFIG_OABI_COMPAT**).

Enfin, précisons bien qu'une ABI n'a aucun rapport direct avec le format de fichier. Le format de binaires ELF (*Executable and Linking Format*), utilisé entre autres par GNU/Linux, n'est qu'un format de fichier, rien de plus (voir articles dans GLMF 129 juillet/août 2010).

Nous installons également, via **aptitude**, une bibliothèque standard C pour plateforme Armel, ainsi que les fichiers de développement nécessaires. Techniquement, à ce stade, nous sommes déjà en mesure de compiler un code source à destination de notre plateforme pour peu qu'il repose uniquement sur la libC.

Enfin, nous aurons besoin également d'un outil particulier pour communiquer, dans un premier temps, avec le bootloader par défaut pour accéder à la flash NAND (et éventuellement NOR) de la carte. En effet, celle-ci dispose d'un port USB *device* utilisé par le bootloader pour télécharger des données et des images depuis la machine de développement. Cet outil, appelé **usbpush**, repose sur la libUSB et les sources sont téléchargeables directement sur http://friendlyarm.net/downloads sous la désignation *USB-Push download tool (Linux)*. Il s'agit d'une version modifiée du *QT-2410 USB RAM loader* (**qt2410_boot_usb**/**s3c2410_boot_usb**) développé par *HMW consulting*, destiné au s3c2410 équipant par exemple l'OpenMoko Neo1973.

Après désarchivage, un simple **make** vous permettra de produire un binaire **usbpush** que vous pourrez installer dans un répertoire présent dans votre **$PATH**.

2 Bootloader en NAND : de Supervivi à U-Boot

Maintenant que nous sommes en mesure de compiler et de charger des binaires dans notre carte, il est temps de passer à la première étape de la migration du système préinstallé à quelque chose de plus maîtrisable et de plus standard. Nous commençons donc par le début, à savoir le booloader. Installé par défaut à la fois dans la NAND et la NOR, Supervivi est un booloader fonctionnel et adapté à la plateforme. Cependant, nous préférerons, comme bien d'autres utilisateurs, reposer sur U-Boot, une solution plus polyvalente, plus modulaire et plus courante. Nous conserverons cependant Supervivi dans la NOR pour pallier d'éventuels problèmes dans la suite des opérations. Libre à vous, le moment venu, de le remplacer également par U-Boot.

Des versions spécifiques, déjà patchées, des divers éléments constituant un système GNU/Linux sont disponibles via Git. La construction du booloader s'en trouve facilitée puisqu'il suffit de récupérer les sources et de les compiler :

```
% mkdir u-boot
% cd u-boot
% git clone git://repo.or.cz/u-boot-openmoko/mini2440.git
% cd mini2440
% make mini2440_config
Configuring for mini2440 board...
% make CROSS_COMPILE=arm-linux-gnueabi-
% ls -l u-boot.bin
5587274 -rwxr-xr-x 1 denis denis 248656 18 mai 11:04 u-boot.bin
```

Nous obtenons ainsi le binaire **uboot.bin** que nous pourrons installer dans la flash NAND. Notez l'utilisation de **CROSS_COMPILE=arm-linux-gnueabi-** dans la ligne de commandes **make** permettant de spécifier la racine du nom des outils de développement. Si vous vous intéressez aux spécificités d'U-Boot concernant la mini2440, jetez simplement un œil aux fichiers placés dans **board/mini2440/***.

Le binaire obtenu, nous devons maintenant le charger sur la carte. Cette étape est plus délicate qu'il n'y paraît. La logique utilisée sera la suivante :

- Booter sur Supervivi en NOR ;
- Charger le binaire U-Boot en mémoire ;
- Inscrire le binaire dans la flash NAND ;
- Rebooter sur U-Boot en NAND ;
- Effacer la flash NAND ;
- Réinstaller U-Boot.

Ceci peut paraître un peu chaotique, mais il s'est avéré impossible, pour une raison inconnue, de lancer directement U-Boot depuis Supervivi (alors que d'autres utilisateurs, avec la même carte, ne rapportent aucun problème particulier). La réinstallation d'U-Boot via Supervivi après effacement de la NAND s'avérait, quant à elle, plus simple via la liaison USB que via U-Boot lui-même. Comme disent les développeurs Perl, il y a plus d'une manière de faire les choses.

La carte dispose d'un interrupteur S2 permettant de choisir le type de flash à utiliser pour le démarrage. Nous plaçons donc l'interrupteur sur « NOR » et mettons sous tension la carte (S1) après avoir connecté le câble série RS232 livré à la machine de développement et lancé un émulateur de terminal série configuré en 115200 8N1 (**screen /dev/ttyS0 115200**, par exemple). Nous obtenons ainsi le menu suivant :

```
##### FriendlyARM BIOS 2.0 for 2440 #####
[x] format NAND FLASH for Linux
[v] Download vivi
[k] Download linux kernel
[y] Download root_yaffs image
[a] Absolute User Application
[n] Download Nboot for WinCE
[l] Download WinCE boot-logo
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 0945-2K
Enter your selection: q
Supervivi>
```

C'est là l'écran d'accueil du bootloader Supervivi nous offrant un certain nombre de possibilités (personnellement, la simple présence d'une entrée « Download WinCE boot-logo » dans ce menu est presque une raison suffisante pour justifier le passage à U-Boot). Nous utilisons la lettre « q » afin de quitter le menu et passer à la ligne de commandes du shell intégré. Nous flashons directement les données récupérées via USB avec :

```
Supervivi> load flash 0 248656 u
```

La syntaxe de **load** utilise plusieurs arguments :

- La mémoire cible, **flash** ou **ram** ;
- l'adresse où inscrire les données. Ici **0** pour le début de la flash NAND ;
- la taille des données à traiter. **248656** correspond à la taille en octets de **u-boot.bin** telle qu'affichée par la précédente commande **ls -l** ;
- la méthode utilisée pour la récupération des données entre **x** (Xmodem), **y** (Ymodem), **z** (Zmodem) et **u** pour USB. Il est, en effet, possible d'envoyer les données directement avec l'émulateur de terminal si celui-ci propose l'*upload* dans l'un des trois protocoles disponibles (**minicom**, par exemple, s'en sortira très bien).

Suite à la validation de cette commande, Supervivi reste en attente de données. Nous basculons donc sur la machine de développement à l'autre bout du câble USB et utilisons **usbpush** pour envoyer l'image binaire d'U-Boot :

```
% sudo usbpush u-boot.bin
csum = 0x8b4d
send_file: addr = 0x30000000, len = 0x0003cb50
```

La commande doit être lancée sous l'identité du super-utilisateur **root** afin de pouvoir accéder directement, via la libUSB, au périphérique apparaissant sous GNU/Linux avec le couple VendorID/ProductID 5345:1234. Celui-ci est décrit par **lsusb** comme « Owon PDS6062T Oscilloscope », mais les chaînes dans **iManufacturer** et **iProduct** indiquent bien « System MCU » et « SEC S3C2410X Test B/D ». Si vous avez un peu de temps, vous pouvez concocter une règle **udev** pour éviter le passage en **root**. Dès la commande validée, nous constatons le déroulement des opérations côté mini2440 :

```
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30000000h,TOTAL:248666]
RECEIVED FILE SIZE:  248666 (80KB/S, 3S)
Downloaded file at 0x30000000, size = 248656 bytes
Found block size = 0x00040000
Erasing...    ... done
Writing...    ... done
Written 248656 bytes
```

Le booloader a correctement reçu les données, effacé la zone de NAND et enregistré notre U-Boot. Il ne nous reste plus qu'à éteindre la carte, basculer sur la NAND en changeant la position de S2 et relancer le système. Nous arrivons, comme prévu, dans U-Boot, dont nous stoppons le démarrage automatique avec une simple pression d'une touche :

```
U-Boot 1.3.2-mini2440 (May 18 2010 - 11:04:04)

I2C:   ready
DRAM:  64 MB
NOR Flash not found. Use hardware switch and 'flinit'
Flash:  0 kB
NAND:  Bad block table not found for chip 0
Bad block table not found for chip 0
256 MiB
*** Warning - bad CRC or NAND, using default
environment

USB:   S3C2410 USB Deviced
In:    serial
Out:   serial
Err:   serial
MAC: 08:08:11:18:12:27
Hit any key to stop autoboot:  0
MINI2440 #
```

Nous voilà en terrain plus familier. Plusieurs éléments sont importants dans l'affichage découlant de ce premier démarrage. Nous avons un *warning* signalant l'utilisation des paramètres par défaut. En effet, U-Boot utilise, tout comme Linux, la notion de MTD (*Memory Technology Devices*) pour gérer la mémoire flash via une couche d'abstraction. La flash est ainsi arbitrairement découpée en blocs pour apparaître sous la forme de périphériques. U-Boot est généralement installé sur la première partition/partie (*part*) et sa configuration sur la seconde. Notez que cette notion de partition n'est pas physiquement inscrite dans la flash, mais est simplement définie dans les variables de configuration/environnement d'U-Boot (**`printenv mtdparts`**). Ici, nous n'avons fait qu'inscrire le binaire du booloader en flash sans nous occuper de la configuration. Cet avertissement est donc parfaitement normal.

Plus problématique, mais tout aussi logique, U-Boot nous signale **`Bad block table not found for chip 0`**. Il faut savoir, en effet, que la mémoire flash NAND est un média de stockage très différent d'un disque ou d'une clé USB. Il s'agit d'un simple composant et non d'un périphérique. Ainsi, ce qui apparaît comme transparent pour l'utilisateur doit être ici pris en compte : la présence de zones défectueuses. Il faut bien comprendre qu'il existe une différence notable entre la mémoire flash telle qu'utilisée sur un système embarqué et un périphérique contenant ce même type de mémoire. Dans le premier cas, le (ou les) composant(s) est (sont) accédé(s) directement. Dans le second, comme une clé USB, un micro-contrôleur fait office d'interface. Ces micro-contrôleurs intègrent une FTL (*Flash Translation Layer*), une couche de traduction faisant apparaître la mémoire comme un périphérique bloc. Il en va de même pour les disques SSD ou les cartes SSD. On parle alors de périphériques FTL. Cette couche d'abstraction, en plus de gérer la traduction d'un format à un autre, se charge de dissimuler la gestion des cellules mémoires défectueuses. Eh oui, tout comme avec un disque dur parfaitement neuf, vous achetez généralement un périphérique en partie défectueux. Mais tout ceci est géré bien en amont de l'interface MCI/MMC, USB, sATA ou SAS et de la notion de secteurs défectueux. De plus en plus de plateformes embarquées reposent maintenant sur un stockage sur périphériques FTL. C'est le cas, par exemple, de la carte ACME G20 initialement équipée de 8Mo de NAND, mais destinée à booter sur une carte SD. On notera d'ailleurs que la nouvelle version de cette carte intègre toujours de la NAND, mais seulement 256Ko en lieu et place des 8Mo des premières éditions. Fait révélateur de la tendance générale que beaucoup considèrent comme regrettable d'un point de vue des performances (la NAND n'est pas très rapide, le fait d'ajouter une FTL n'arrange rien).

La mémoire NAND est davantage sujette aux erreurs que la NOR en termes de fabrication. Plus la densité augmente (technologie MLC) plus les puces « fatiguent » vite. Dans le cadre de l'accès direct à la mémoire flash, cette gestion d'erreurs n'existe pas et il est nécessaire de traiter la problématique autrement. Pour cela, il faut initialiser la mémoire en analysant chaque zone pour déterminer sa viabilité et établir une table regroupant les blocs défectueux. Cette table est la BBT pour *Bad-Block Table*. Il faut également savoir que la mémoire est organisée en pages, qui constituent des blocs, et que chaque page est accompagnée d'un certain nombre d'octets utilisés pour le stockage des métadonnées et des codes correcteurs d'erreurs (ECC pour *Error Correcting Code*). Cette zone de 64 octets par page de 2048 octets est appelée zone OOB pour *Out Of Band*. Malheureusement, le marqueur de blocs défectueux présent dans la zone OOB ne permet pas de différencier les blocs défectueux « d'usine » de ceux qui sont devenus défectueux. Voilà pourquoi la BBT est utile. Celle-ci est normalement située, sous forme de carte, dans les deux derniers blocs valides de la NAND. Une copie est faite dans les deux blocs précédents par sécurité.

À ce stade, nous devons créer une BBT pour notre mémoire Flash avec U-Boot. La méthode est relativement simple, puisqu'il suffit d'effacer entièrement la mémoire avec :

```
MINI2440 # nand scrub
NAND scrub: device 0 whole chip
Warning: scrub option will erase all factory set bad blocks!
         There is no reliable way to recover them.
         Use this command only for testing purposes if you
         are sure of what you are doing!

Really scrub this NAND flash? <y/N>
Erasing at 0x3840000 --  22% complete.
NAND 256MiB 3,3V 8-bit: MTD Erase failure: -5
Erasing at 0x3b00000 --  23% complete.
NAND 256MiB 3,3V 8-bit: MTD Erase failure: -5
Erasing at 0x3fe0000 --  25% complete.
NAND 256MiB 3,3V 8-bit: MTD Erase failure: -5
Erasing at 0xffe0000 -- 100% complete.
Bad block table not found for chip 0
Bad block table not found for chip 0
OK
```

L'effacement consiste à écrire **`0xff`** dans toute la flash. Oui, ceci revient donc à effacer notre première installation d'U-Boot. On remarquera qu'un certain nombre d'erreurs apparaissent lors de la manipulation, ce sont les blocs défectueux. Nous pouvons ensuite créer la table avec :

```
MINI2440 # nand createbbt
Create BBT and erase everything ? <y/N>
Skipping bad block at  0x03ae0000
Skipping bad block at  0x03d60000
```

```
Skipping bad block at  0x041c0000
Skipping bad block at  0x0ff80000
Skipping bad block at  0x0ffa0000
Skipping bad block at  0x0ffc0000
Skipping bad block at  0x0ffe0000

Creating BBT. Please wait ...Bad block table not found for chip 0
Bad block table not found for chip 0
[...]
Bad block table written to 0x0ffe0000, version 0x01
Bad block table written to 0x0ffc0000, version 0x01
```

Cette étape est relativement longue. Attention, je dis bien : LONGUE ! N'allez pas croire que la commande a échoué si vous ne reprenez pas la main sur le shell U-Boot après quelques minutes. Plus vous avez de flash NAND, plus ce sera long. Un certain nombre de commentaires et de questions remplissent les forums et listes de diffusion quant à cette commande. Dans 90% des cas, l'utilisateur n'a tout simplement pas assez attendu. Vous voilà prévenu.

Une fois le prompt d'U-Boot revenu à l'écran, vous devrez réinstaller le bootloader. Plusieurs solutions sont envisageables. Ici, nous nous contenterons de redémarrer sur la NOR pour réitérer les manipulations précédentes de chargement d'U-Boot via **usbpush**. Ceci fait, il ne reste plus qu'à démarrer à nouveau sur la NAND. À ce moment, U-Boot ne devrait plus se plaindre de l'absence de BBT, mais il reste un problème : celui du stockage de la configuration du bootloader.

Nous avons parlé des MTD précédemment. Il faut maintenant prendre en compte la BBT. En effet, la taille et la position du début de chaque partition MTD sont dépendantes de la présence de blocs inutilisables. Ainsi, pour réserver un espace identique pour chaque installation d'U-Boot, de sa configuration et du noyau Linux sur toutes les plateformes, il est nécessaire de « sauter » les blocs défectueux pour obtenir des positions (*offset*) valables. Pas qu'inquiétude, U-Boot est en mesure de faire tous les calculs à votre place via une simple commande :

```
MINI2440 # dynpart
mtdparts mtdparts=mini2440-nand:0x00040000(u-boot),
0x00020000(u-boot_env),0x00500000(kernel),
0x0faa0000(rootfs)
```

De la même manière, nous ne pouvons pas écrire n'importe où la configuration d'U-Boot. Utiliser une position fixe revient à risquer de tomber sur un bloc défectueux. Pour préparer le terrain, on utilisera alors la commande suivante :

```
MINI2440 # dynenv set u-boot_env
device 0 offset 0x40000, size 0x20000
45 4e 56 30 - 00 00 04 00
```

Notez qu'on utilise ici le nom MTD de la zone de NAND qui nous intéresse plutôt qu'une position (offset). C'est une simple facilité d'utilisation offerte par le bootloader. Il ne nous reste plus qu'à enregistrer la configuration (l'environnement U-Boot) actuelle dans la flash avec :

```
MINI2440 # saveenv
Saving Environment to NAND...
Erasing Nand...Writing to Nand... done

MINI2440 # reset

U-Boot 1.3.2-mini2440 (May 18 2010 - 11:04:04)

I2C:   ready
DRAM:  64 MB
NOR Flash not found. Use hardware switch and 'flinit'
Flash:  0 kB
NAND:  256 MiB
Found Environment offset in OOB..
USB:   S3C2410 USB Deviced
In:    serial
Out:   serial
Err:   serial
MAC: 08:08:11:18:12:27
Hit any key to stop autoboot:  0
MINI2440 #
```

En enchaînant sur un *reset* après la sauvegarde, nous pouvons constater qu'U-Boot ne trouve plus rien à redire à notre configuration. Nous pouvons d'ailleurs en profiter pour constater l'étendue des dégâts en flash avec :

```
MINI2440 # nand bad

Device 0 bad blocks:
  03ae0000
  03d60000
  041c0000
  0ff80000
  0ffa0000
  0ffc0000
  0ffe0000
```

7 blocs défectueux n'est pas quelque chose d'alarmant sur quelques 256 Mo de NAND dans le cas présent. Concluons cette partie en précisant que le noyau Linux, selon la version utilisée, n'est pas forcément en mesure de gérer la BBT. C'est une information à prendre en compte dans le choix de votre système de fichiers racine, ainsi que dans l'analyse d'éventuels messages d'erreur remontés par certains outils et services. En fonction de la plateforme, il est possible qu'un ou plusieurs patches existent pour prendre en compte la BBT selon le pilote utilisé pour l'accès à la NAND.

3 MTD et Noyau

Nous avons maintenant deux bootloaders à notre disposition, le premier est Supervivi, en NOR, en cas de problème et le second, pour l'usage courant, en NAND, est U-Boot. Il ne nous reste donc plus que deux étapes avant d'obtenir un système pleinement fonctionnel : l'installation d'un noyau et la mise en place d'un système de fichiers racine (rootfs).

Ces deux éléments vont prendre place, à terme, dans la flash NAND, chacun d'eux sur une partition MTD dédiée. La commande **mtd** dans U-Boot vous permet d'afficher la structure et les informations utiles :

```
MINI2440 # mtd

device nand0 <mini2440-nand>, # parts = 4
#: name          size        offset      mask_flags
0: u-boot        0x00040000  0x00000000  0
1: u-boot_env    0x00020000  0x00040000  0
2: kernel        0x00500000  0x00060000  0
3: rootfs        0x0faa0000  0x00560000  0

active partition: nand0,0 - (u-boot) 0x00040000 @ 0x00000000

defaults:
mtdids  : nand0=mini2440-nand
mtdparts: <NULL>
```

Nous retrouvons l'emplacement pour le bootloader à l'adresse **0×00000000**, d'une taille de 256 Ko, suivi de son environnement, puis du noyau Linux sur 5 Mo, et enfin, quelques 250 Mo pour un rootfs. Remarquez que les noms donnés peuvent être utilisés aussi bien que les offsets (position par référence au début de la NAND) dans presque toutes les commandes d'U-Boot. Ainsi, pour effacer la partition destinée au noyau, nous pouvons utiliser :

```
MINI2440 # nand erase kernel
NAND erase: device 0 offset 0x60000, size 0x500000
Erasing at 0x540000 -- 100% complete.
OK

MINI2440 # nand erase rootfs
NAND erase: device 0 offset 0x560000, size 0xfaa0000
Skipping bad block at  0x03ae0000
Skipping bad block at  0x03d60000
Skipping bad block at  0x041c0000
Skipping bad block at  0x0ff80000
Skipping bad block at  0x0ffa0000
Skipping bad block at  0x0ffc0000
Skipping bad block at  0x0ffe0000
OK
```

U-Boot se charge de trouver les informations et fait les calculs à notre place. Notez qu'on trouve parfois décrite sur le Web l'utilisation de la commande **erase.e**. La différence (qui n'en est plus une) avec **erase** tient dans le fait de prendre en compte (sauter) les blocs défectueux. Avec les dernières versions d'U-Boot, les deux commandes sont synonymes et **erase.e** tout comme **write.e** ne sont conservées que pour des raisons de compatibilité.

Notre zone d'installation du noyau est à présent prête, nous pouvons passer à la phase de construction de ce dernier. Nous n'utiliserons pas ici des sources vanilla (sources officielles), mais une version déjà adaptée que nous récupérerons depuis le même serveur Git qu'U-Boot :

```
% mkdir kernel
% cd kernel/
% git clone git://repo.or.cz/linux-2.6/mini2440.git
Initialized empty Git repository in /home/denis/EMB/mini2440/kernel/
mini2440/.git/
remote: Counting objects: 1378064, done.
remote: Compressing objects: 100% (223156/223156), done.
Receiving objects: 100% (1378064/1378064), 345.55 MiB | 888 KiB/s, done.
remote: Total 1378064 (delta 1149518), reused 1370278 (delta 1145373)
Resolving deltas: 100% (1149518/1149518), done.
Checking out files: 100% (30493/30493), done.
```

Au moment où cet article est rédigé, il s'agit d'un 2.6.32-rc8. On pourra, bien entendu, adapter ces sources à n'importe quelle version récente du noyau à grands coups de **diff** et de **patch**. Pour l'heure, contentons-nous de compiler l'ensemble selon la configuration par défaut également fournie, sous la forme du fichier **mini2440_defconfig** :

```
% cd mini2440
% mkdir -p ../kernel-bin
% make mrproper

% make ARCH=arm mini2440_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/basic/docproc
  HOSTCC  scripts/basic/hash
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/kxgettext.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/lex.zconf.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
```

Vous pouvez analyser la configuration, ou éventuellement changer quelques éléments, avec un simple **make ARCH=arm menuconfig**, comme vous le feriez pour un noyau standard. L'une des modifications pourrait être, par exemple, le changement des arguments de boot par défaut avec **console=ttySAC0,115200 noinitrd** ou encore l'activation de l'entrée **config.gz** dans **/proc**. Une fois satisfait de la configuration, lancez la construction avec **make CROSS_COMPILE=arm-linux-gnueabi- CFLAGS="-march=armv4t -mtune=arm920t" CXXFLAGS="-march=armv4t -mtune=arm920t" ARCH=arm** et trouvez quelque chose de plus intéressant à faire en attendant que de regarder les lignes défiler...

Le noyau produit n'est pas directement utilisable par U-Boot. Il est nécessaire d'utiliser un outil fourni avec le bootloader pour produire une image compatible intégrant des informations complémentaires pour le chargeur :

```
% ../../u-boot/mini2440/tools/mkimage -A arm \
-O linux -n "linux-2.6" -T kernel -C none \
-a 0x30008000 -e 0x30008000 \
-d arch/arm/boot/Image ~/EMB/tmp/uImage
Image Name:   Linux-2.6
```

```
Created:      Tue May 18 14:10:58 2010
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4150240 Bytes = 4052.97 kB = 3.96 MB
Load Address: 30008000
Entry Point:  30008000
```

On précise ainsi plusieurs éléments, comme le type d'architecture (ARM), le type de compression utilisé (aucune), l'adresse de chargement et le point d'entrée, etc. Il en résulte la fabrication du fichier **uImage**, ici dans le répertoire **~/EMB/tmp**, qui sera notre racine pour le serveur TFTP. En effet, nous n'utiliserons plus le chargement par USB, mais allons privilégier désormais l'utilisation du réseau puisque celui-ci est pris en charge par U-Boot. Pour ce faire, vous devrez installer un serveur TFTP comme **tftpd-hpa**. Personnellement, j'ai une préférence pour son utilisation via **xinetd** plutôt que le mode daemon, mais ceci n'a pas d'impact sur la suite des opérations.

De retour dans U-Boot sur la plateforme embarquée, assurez-vous de la validité de la configuration réseau. Plusieurs variables d'environnement entrent en jeu :

- **ipaddr** : l'adresse IP de la plateforme ARM ;
- **netmask** : le masque de sous-réseau ;
- **serverip** : l'adresse du serveur (TFTP, HTTP, etc.).

Ces variables doivent être définies avec la commande **setenv** (sans **=**) puis enregistrées dans la configuration avec **saveenv**. Ceci fait, vous pouvez alors télécharger l'image produite par **mkimage** depuis la machine de développement :

```
MINI2440 # tftp 0x32000000 uImage
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:08:11:18:12:27
TFTP from server 192.168.10.61; our IP address is 192.168.10.170
Filename 'uImage'.
Load address: 0x32000000
Loading: T ################################
         ################################
         ################################
         ################################
         ################################
         ################################
         ################
done
Bytes transferred = 4150304 (3f5420 hex)
```

L'image est chargée à l'adresse **0×32000000**. À ce stade et avant même d'inscrire quoi que ce soit en flash, nous pouvons tenter un premier démarrage voué à l'échec en l'absence de système de fichiers racine. Pour cela, nous définissons temporairement le contenu de la variable **bootargs** en **ttySAC0,115200 noinitrd panic=5**. Ainsi, si vous n'avez pas enregistré les arguments de boot dans la configuration du noyau, nous pourrons visualiser le processus de démarrage sur la console série (en plus de l'écran LCD presque illisible en mode console) et, via **panic=5**, provoquer un *reboot* automatique lorsque le noyau se rendra compte qu'il n'accède pas au rootfs. Lancez le noyau en utilisant la commande **bootm** en spécifiant l'adresse de chargement de l'image :

```
MINI2440 # bootm 0x32000000
## Booting kernel from Legacy Image at 32000000 ...
   Image Name:   Linux-2.6
   Created:      2010-05-18  12:10:58 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4150240 Bytes =  4 MB
   Load Address: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Linux version 2.6.32-rc8 (denis@x61)
  (gcc version 4.3.4 (Debian 4.3.4-10) )
  #1 Tue May 18 12:03:19 CEST 2010
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0003177
CPU: VIVT data cache, VIVT instruction cache
Machine: MINI2440
Memory policy: ECC disabled, Data cache writeback
CPU S3C2440A (id 0x32440001)
S3C24XX Clocks, (c) 2004 Simtec Electronics
[...]
```

U-Boot charge le noyau, initialise divers éléments comme les arguments de boot, analyse l'image, puis saute au point d'entrée pour passer la main. S'ensuit alors le démarrage du code Linux avec tous les messages habituels d'initialisation des périphériques. Nous pouvons constater la prise en charge de plusieurs d'entre eux (interface réseau, son, *touchscreen*, contrôleur USB, etc.). Enfin, nous avons confirmation du bon fonctionnement de notre paramètre **panic=5** via un message assez explicite et un redémarrage du système.

Notre noyau semble fonctionner correctement, nous pouvons donc l'enregistrer dans la partition MTD correspondante. Pour ce faire et suite au redémarrage, il nous faut à nouveau le télécharger via TFTP. Ensuite, nous pouvons utiliser la commande **nand write**, non sans avoir procédé à un petit calcul auparavant. En effet, notre image fait 4150240 octets, mais nous devons l'aligner sur un nombre fini de pages de mémoire flash. Ces pages ont une taille fixe sur la plateforme de 2048 octets (information disponible à l'aide d'un simple **nand info**). Nous calculons donc 4150240/2048 = 2026.484375. La taille, *paddée* sur les pages de 2048 octets, sera donc de 2027*2048, soit 4151296 ou 0x3F5800 en notation hexadécimale.

Pour les calculs, je vous conseille le très sympathique **bc** accompagné de l'option **-l**, vous fournissant en un tour de main la valeur hexa attendue :

```
% echo "obase = 16; scale = 0; s=4150304/2048; (s+1)*2048" | bc
3F5800
```

La ligne de commandes complète pour enregistrer le noyau en flash sera donc :

```
MINI2440 # nand write 0x32000000 60000 3F5800

NAND write: device 0 offset 0x60000, size 0x3f5800
4151296 bytes written: OK
```

Nous enregistrons 0x3F5800 octets de données depuis l'adresse 0×32000000 (point de chargement de l'image en mémoire) à partir de l'offset 0×60000 correspondant au début de la zone **kernel**. Nous pouvons, suite à cette opération, vérifier le fonctionnement avec :

```
MINI2440 # iminfo

## Checking Image at 32000000 ...
   Legacy image found
   Image Name:   Linux-2.6
   Created:      2010-05-18  12:10:58 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4150240 Bytes =  4 MB
   Load Address: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
```

Tout va pour le mieux, passons à la suite...

4 Le problème du rootfs

Notre noyau Linux fonctionnel et parfaitement installé, nous pouvons nous pencher sur le système de fichiers racine. L'opération est plus délicate qu'il n'y paraît, selon le contenu du système de fichiers, la distribution utilisée et la manière de transférer les données en flash.

Ici, nous visons l'installation d'emDebian, une version embarquée de Debian et plus exactement l'édition Grip regroupant un nombre réduit de paquets (quelques 2000 seulement). Le système de fichiers utilisé sera JFFS2, très courant dans le monde de Linux pour l'embarqué. Il s'agit d'un système de fichiers compressé spécialement destiné à la mémoire flash. Nous disposons, je le rappelle, de quelques 256Mo de NAND, mais de seulement 64Mo de SDRAM. Le problème avec un rootfs Debian réside dans la taille des données, même compressées. Celle-ci est, en effet, sensiblement identique à la mémoire disponible. Difficile donc, voire impossible, de flasher le système de fichiers racine avec U-Boot.

Deux solutions sont alors envisageables. La première consiste à procéder à un premier démarrage via NFSroot ou, en d'autres termes, utiliser un rootfs réseau. La seconde est similaire, mais consiste à utiliser le slot SD/MMC pouvant accueillir une carte de quelques gigaoctets. Dans les deux cas, il ne s'agit que d'une solution de transition. Nous nous servirons de cette installation pour accéder à la flash NAND pour y placer le rootfs définitif.

4.1 rootfs sur SD/MMC

Chose assez amusante, nous nous servirons ici d'une carte SD/MMC de 8Go de façon temporaire alors que cette manipulation semble, d'après les messages sur les forums consacrés à la mini2440, une fin en soi pour beaucoup d'utilisateurs (et la NAND alors ?). L'utilisation d'une MMC comme rootfs peut paraître séduisante. Les manipulations de mises à jour et de modifications du système s'en trouvent facilitées puisqu'il suffit d'accéder au support depuis la machine de développement. Cependant, pour ma part, je considère que la notion de « média amovible » n'est pas compatible avec celle de système de fichiers racine. Ceci du fait de la nature même du support, mais aussi et surtout qu'il apparaît bien plus légitime et pertinent de disposer du système dans la NAND et des données sur support amovible. Si vous cherchez vraiment à obtenir un système pour l'expérimentation, la solution du rootfs en NFS sera bien plus souple. J'ai eu le loisir de constater la dégradation d'un support MMC suite à des insertions/retraits fréquents, l'usure des contacts et autres joyeusetés dans le même esprit. Un rootfs sur support amovible n'est pas une solution de développement, pas plus qu'une solution de production.

Trêve d'argumentation et passons à la pratique. On se procure donc une carte SD/MMC de quelques 512Mo ou plus (ici 8Go) et on crée une seule partition Linux qu'on initialise en ext3. Ne reste plus ensuite qu'à monter ce système de fichiers, par exemple dans **/mnt/SD** et :

```
% debootstrap --arch=armel --foreign lenny \
/mnt/SD/ http://www.emdebian.org/grip/
```

debootstrap est un outil magique. Il permet d'initier une installation Debian GNU/Linux directement par le réseau (HTTP) en précisant simplement l'architecture, la distribution et le miroir à utiliser. L'ensemble est alors téléchargé et agencé automatiquement. Bien entendu, tout n'est pas parfait et nous devons ajuster quelques éléments de configuration. Nous n'avons pas un seul module noyau présent dans notre futur système de fichiers racine. **debootstrap** installe une distribution, mais aucun élément du noyau. Nous allons corriger cela en retournant dans les sources de notre noyau et en lançant la compilation des modules :

```
# copie des modul% cd /rep/des/source/kernel
% make CFLAGS="-march=armv4t -mtune=arm920t" \
CXXFLAGS="-march=armv4t -mtune=arm920t" \
ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules

% make CFLAGS="-march=armv4t -mtune=arm920t" \
CXXFLAGS="-march=armv4t -mtune=arm920t" \
ARCH=arm INSTALL_MOD_PATH=/mnt/SD modules_install
```

La variable **INSTALL_MOD_PATH** nous permet de directement installer les modules dans notre point de montage de la SD/MMC, à partir de la racine. L'arborescence complète des modules sera créée automatiquement.

Ce n'est pas tout. Nous devons également préciser quelques éléments de configuration non initialisés par **debootstrap** :

```
# cd /mnt/SD
% echo "proc /proc proc none 0 0" >>etc/fstab
% echo "minideb" >etc/hostname
% mknod dev/console c 5 1
% mknod dev/ttySAC0 c 204 64
% echo 'deb http://www.emdebian.org/grip/ lenny main' \
> etc/apt/sources.list
```

Notre futur système Debian est maintenant prêt. N'oublions pas, cependant, que notre objectif n'est pas simplement d'avoir un système complet sur carte SD (sinon, nous nous arrêterions là). Nous allons créer une archive **tar** complète de ce système de fichiers, afin de pouvoir ensuite le copier dans la flash NAND JFFS2 de la mini2440 :

```
% tar jcf /kkpart/emdebian-grip-armel-debootstrap-lenny.tar.bz2 .
% cp /kkpart/emdebian-grip-armel-debootstrap-lenny.tar.bz2 /mnt/SD/root/
% cd
% sync
% umount /mnt/SD
```

Nous plaçons le **tar** à la racine de la carte SD avant de démonter le système de fichiers. Il ne reste plus qu'à placer la carte dans l'emplacement prévu à cet effet sur la plateforme ARM. Remarquez que les emplacements SD/MMC des mini2440 sont relativement trompeurs. Il faut en effet s'assurer que la carte est bien logée « à fond » dans l'emplacement pour éviter un lot de messages d'erreur et une recherche inutile de la cause. N'hésitez pas à enfoncer fermement la carte pour éviter les faux contacts. Après démarrage sur la NAND, on accède au bootloader et nous pouvons dans un premier temps vérifier la lecture de la SD avec :

```
MINI2440 # mmcinit
mmc: Probing for SDHC ...
mmc: SD 2.0 or later card found
trying to detect SD Card...
Manufacturer:       0x02, OEM "TM"
Product name:       "SA08G", revision 0.3
Serial number:      2630396422
Manufacturing date: 10/2009
CRC:                0x39, b0 = 1
READ_BL_LEN=15, C_SIZE_MULT=0, C_SIZE=365
size = 0
SD Card detected RCA: 0x1234 type: SDHC
```

La commande **mmcinit**, comme son nom l'indique, initialise l'interface SD/MMC de la plateforme (et non la carte qui y est connectée, vos données ne risquent rien). Nous récupérons par la même occasion quelques informations sur le support en question (date de fabrication, taille, numéro de série, etc.). Les données de la carte sont maintenant accessibles et nous n'avons finalement qu'à modifier les arguments de boot du noyau Linux (sur une ligne) :

```
MINI2440 # setenv bootargs 'console=ttySAC0,115200 noinitrd
root=/dev/mmcblk0p1 rootdelay=4 rw mini2440=3tb
ip=dhcp init=/bin/sh'
```

console=ttySAC0,115200 noinitrd ainsi que **root=** sont des arguments classiques et très connus des utilisateurs GNU/Linux. Rien de particulier sur ce point. **rootdelay** en revanche, est très important. Le temps d'accès à la carte peut être plus ou moins important en fonction de sa qualité. Avec une valeur de 4 (secondes), nous nous assurons ainsi un délai d'attente du périphérique plus que suffisant. **mini2440=3tb** est un argument à passer au noyau Linux afin qu'il puisse gérer correctement l'affichage et plus particulièrement l'adressage des pixels du framebuffer. L'argument passé en option correspond ici à l'écran tactile 3,5". Jetez un œil au fichier **arch/arm/mach-s3c2440/ mach-mini2440.c**. Vous y trouverez une structure **s3c2410fb_display mini2440_lcd_cfg** listant les différents modèles d'écran. En **[3]** se trouve l'écran TFT 3,5" nouvelle génération, référence T35. Adaptez l'argument **mini2440=xtb** où **x** est le numéro du périphérique dans la structure (**[1]**/**1tb** pour le 7", par exemple). Enfin, on notera l'utilisation de **init=** afin de lancer directement un shell et non un système de démarrage standard. Nous sommes encore dans la première phase de démarrage de la distribution et l'ensemble des paquets n'a pas encore été configuré correctement.

Pour démarrer, il nous suffit de charger le noyau depuis la NAND et de lancer le boot :

```
MINI2440 # nboot.e kernel
Loading from NAND 256MiB 3,3V 8-bit, offset 0x60000
   Image Name:   Linux-2.6
   Created:      2010-05-18  12:10:58 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4150240 Bytes =  4 MB
   Load Address: 30008000
   Entry Point:  30008000

MINI2440 # bootm
```

Après démarrage complet, nous nous trouvons sur un système Debian GNU/Linux en « demi-phase » d'installation. Il nous faut donc terminer la procédure manuellement avec :

```
sh-3.2# mount /proc /proc -t proc
sh-3.2# export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin
sh-3.2# /debootstrap/debootstrap --second-stage
[...]
sh-3.2# dpkg -i /var/cache/apt/archives/udev_0.125-7em1_armel.deb
sh-3.2# echo ttySAC0 >>etc/securetty
sh-3.2# printf "T0:123:respawn:/sbin/getty 115200 ttySAC0\n" >>etc/
inittab
sh-3.2# printf "auto eth0\niface eth0 inet dhcp\n" >>etc/network/
interfaces
```

Après montage du pseudo système de fichiers **/proc** et configuration de la variable d'environnement **PATH**, nous réutilisons **debootstrap** pour terminer la phase 2 de l'installation. Là, les paquets sont installés et configurés correctement. Il est possible que vous ayez une erreur sur le paquet **udev** en raison d'une tentative échouée de lecture de tous les MTD. Si tel est le cas, utilisez alors **dpkg** pour réinstaller le paquet. Enfin, nous configurons le **securetty** et **getty** afin de pouvoir

obtenir un shell **root** via la console série au prochain démarrage et nous créons le **/etc/network/interfaces** indispensable à la configuration du réseau. De là, un petit **sync** et nous pouvons redémarrer le système pour un boot sur le vrai système, toujours sur SD/MMC.

Note **À propos des MTD, d'udev et des erreurs**

L'installation de **udev** échoue en raison d'erreurs lors de son lancement. Ceci vient du fait que le noyau Linux et U-Boot ne gèrent pas de la même manière la zone OOB de la NAND. L'analyse par **udev** va donc provoquer la lecture des blocs défectueux et donc une erreur. Il est possible de régler le problème en *dumpant* le contenu du périphérique MTD incriminé, en effaçant la NAND et en l'écrivant à nouveau. À ce moment, le format sera accepté par Linux, mais vous aurez des messages d'erreur avec U-Boot.

Nous modifions toujours les variables d'environnement comme **bootargs** sans les enregistrer, comme précédemment, et nous redémarrons. Attention cependant, on s'abstiendra bien entendu de spécifier **ip=dhcp init=/bin/sh** en argument du noyau pour obtenir un démarrage standard. Nous voici donc en présence d'un système Debian GNU/Linux fonctionnel. De là, nous pouvons accéder à la NAND directement depuis la console série ou, éventuellement, installer un serveur OpenSSH pour avoir un shell distant parfois plus réactif.

5 Root NFS pour flash NAND

L'utilisation d'une carte SD/MMC est une solution temporaire si l'on ne souhaite pas voir l'opération autrement que comme une phase transitoire. En d'autres termes, s'il ne s'agit que d'initialiser un système dans la NAND, cette solution peut très bien convenir. Je l'ai dit plus haut, utiliser le système sur la SD/MMC ne me semble pas une bonne idée : pour une utilisation courante, un support amovible est inutile, et pour du développement, les opérations d'insertion/retrait de la carte ne sont guère pratiques/agréables/recommandées.

Si l'on souhaite disposer d'un système de test, modifiable à loisir et utilisable, de plus, avec plusieurs cartes mini2440, la solution la plus adaptée est celle du boot réseau et de l'utilisation d'un système de fichiers racine en NFS. C'est précisément ce que nous allons faire maintenant.

Le système de développement se verra tout d'abord équipé du support NFS serveur via la simple installation du paquet **nfs-kernel-server**. On configurera ensuite le serveur NFS via une simple modification du fichier **/etc/exports** (sur une ligne) :

```
# /etc/exports
/chemin/vers/nfsroot   192.168.10.0/255.255.255.0
  (rw,sync,fsid=0,no_root_squash,no_subtree_check,insecure)
```

Si le chemin n'est pas celui précédemment utilisé pour le **debootstrap**, il nous suffira de recopier les fichiers depuis la SD/MMC avec quelque chose comme :

```
% mount /dev/sdf1 /mnt/SD
% cd /mnt/SD
% find . -xdev | cpio -pm /chemin/vers/nfsroot
```

Ceci fait, retour vers U-Boot pour une modification des arguments de démarrage. La composition de la ligne est dépendante de l'état de la distribution. Soit vous partez d'une installation en phase 1 de **debootstrap** et vous recommencez une finalisation (**init=/bin/sh**), soit vous utilisez le système déjà configuré de la SD/MMC et vous démarrez directement la distribution :

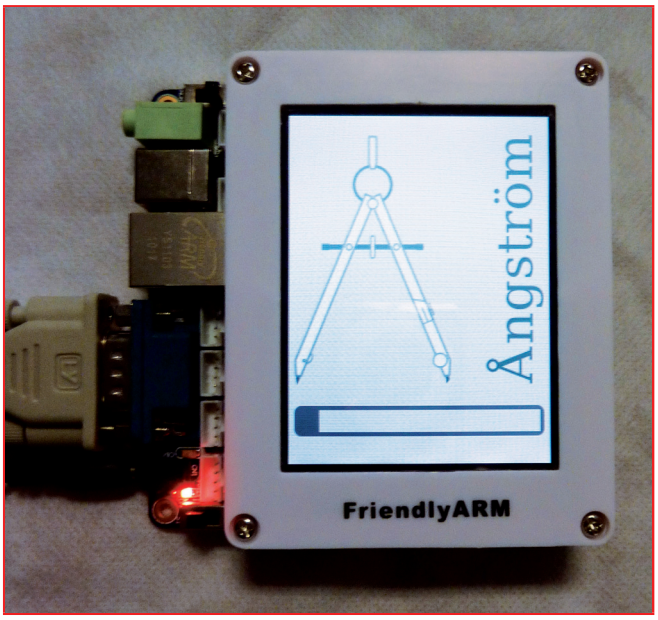
```
MINI2440 # setenv bootargs 'console=ttySAC0,115200 noinitrd
rootdelay=4 mini2440=3tb rw ip=192.168.10.170
nfsroot=192.168.10.166:/chemin/vers/nfsroot
root=/dev/nfs panic=5 ip=dhcp'
```

La suite de la procédure est similaire à celle décrite pour la MMC/SD. En cas de problème, les messages d'erreur concernant NFS n'étant pas très explicites lors du boot, essayez de monter le système de fichiers depuis une autre machine. Très souvent, il ne s'agit que d'une erreur d'autorisation ou d'une faute de frappe dans un chemin.

6 Accès et copie dans la NAND depuis le système Debian temporaire

Vous voilà avec un système Debian fonctionnel, soit en NFS, soit sur SD. Il ne nous reste plus qu'à initialiser la NAND pour notre système de fichiers racine et y copier les fichiers. La distribution Debian est installée avec le paquet **mtd-utils** intégrant la commande **flash_eraseall**. Celle-ci est en mesure d'effacer la flash et via l'option **-j**, de créer un système de fichiers JFFS2 :

```
mini2440:~# flash_eraseall -j /dev/mtd3
Erasing 128 Kibyte @ 3560000 -- 21 % complete. Cleanmarker written at 3560000.
Skipping bad block at 0x03580000
Erasing 128 Kibyte @ 37e0000 -- 22 % complete. Cleanmarker written at 37e0000.
Skipping bad block at 0x03800000
Erasing 128 Kibyte @ 3c40000 -- 24 % complete. Cleanmarker written at 3c40000.
Skipping bad block at 0x03c60000
Erasing 128 Kibyte @ fa80000 -- 99 % complete. Cleanmarker written at fa80000.
```



La mini2440 est en mesure de faire fonctionner plusieurs systèmes d'exploitation, parmi lesquels Debian GNU/Linux, Windows CE 5/6, OpenEmbedded Angström ou encore Android 1.5 cupcake.

Comme c'est Linux qui initialise ce périphérique NAND, vous n'aurez pas d'erreur avec **udev** au démarrage et U-Boot ne s'intéresse pas au rootfs. Dès lors, nous pouvons monter **/dev/mtdblock3** :

```
mini2440:~# mkdir /mnt/fl
mini2440:~# mount -t jffs2 /dev/mtdblock3 /mnt/fl

mini2440:~# df -h
Filesystem           Size  Used Avail Use% Mounted on
tmpfs                 30M     0   30M   0% /lib/init/rw
udev                  10M   24K   10M   1% /dev
tmpfs                 30M     0   30M   0% /dev/shm
rootfs               7.3G  524M  6.5G   8% /
/dev/mtdblock3       251M  5.9M  245M   3% /mnt/fl
```

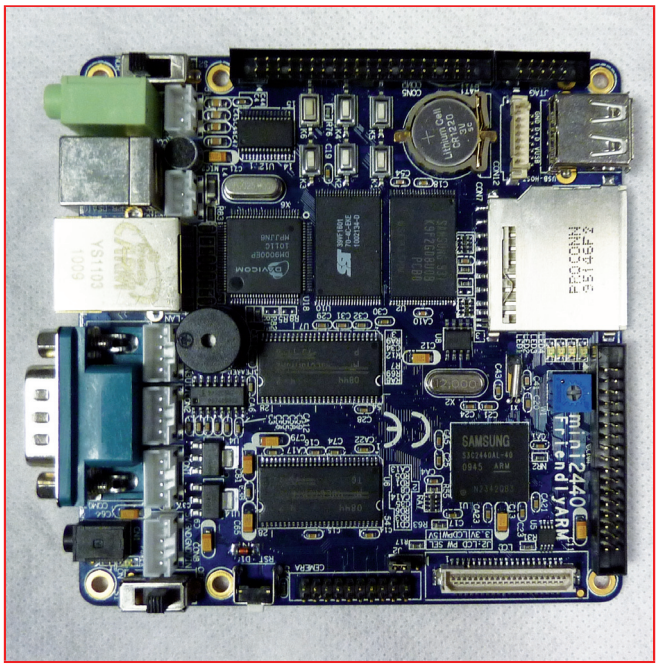
Il ne nous reste plus :

- soit à désarchiver notre tarball dans la flash ;
- soit à copier récursivement le rootfs actuel dans **/mnt/fl** (à éviter si vous êtes en NFS).

La copie terminée et le redémarrage effectué, vous aurez le plaisir de constater que tout est correctement copié :

```
mini2440:~# du -hsc /
194M    /
194M    total

mini2440:~# df -h
Filesystem           Size  Used Avail Use% Mounted on
tmpfs                 30M     0   30M   0% /lib/init/rw
udev                  10M   20K   10M   1% /dev
tmpfs                 30M     0   30M   0% /dev/shm
rootfs               251M  109M  143M  44% /
```

On remarque ici clairement que JFFS2 est bel et bien un système de fichiers compressé (194Mo de fichiers sur **/** et **df** ne nous indique pourtant que **109Mo** occupés). Notre système est maintenant pleinement utilisable et nous pouvons le configurer exactement comme n'importe quelle machine Debian x86 et installer tous les paquets qui nous semblent nécessaires.



La carte est déclinée en plusieurs versions, de la simple platine nue à la version la plus complète avec écran tactile 7", Wifi, caméra, JTAG/usb, etc. L'arrivée récente de la nouvelle mini6410 équipée du processeur Samsung S3C6410 (ARM11) a dernièrement fait chuté le prix de ce modèle aux alentours de $70, soit quelque 50 euros (+ frais de douane et TVA).

Le mot de la presque fin

La mini2440 est une carte pleine de ressources, mais l'installation d'un système différent de ceux proposés sous forme d'images sur le DVD ou le site du fabricant nécessite quelques efforts. Quoi qu'il en soit, passée cette étape, avec un système Debian ou autre, il ne reste plus beaucoup de spécificités à prendre en compte. Le noyau patché prendra en charge la majorité des périphériques intégrés et l'installation d'une interface graphique reposant sur le *framebuffer* ne dépendra finalement que de la présence des bons paquets dans la distribution. Nous reviendrons très certainement sur cette plateforme dans les prochains numéros, sous la forme de « cas pratiques », afin de mieux faire connaissance avec ces périphériques. Je vous recommande, en attendant, de finaliser votre configuration pour obtenir une base de travail stable et, éventuellement, de sauvegarder une image de chaque MTD de votre carte. ■

EMPORTEZ VOTRE HUB ETHERNET EN VOYAGE

par Yann Guidon

En vadrouille, on veut s'embarrasser le moins possible. Mais si on veut faire du développement un peu complexe sur la route, il faut plus d'un ordinateur et un hub Ethernet devient nécessaire. Ce dernier est souvent accompagné d'un bloc secteur plus gros et lourd que le hub lui-même, et dans le cas traité ici, la tension d'entrée (220V 50Hz) n'est pas compatible avec la destination du voyage...

1 Examen de l'alimentation

L'idéal serait d'utiliser une source d'alimentation disponible sous la main, c'est-à-dire l'ordinateur lui-même. Ce dernier dispose naturellement de ports USB qui peuvent fournir 500mA sous 5V, ce qui devrait être largement suffisant pour un petit hub 5 ports... Mais est-ce compatible ?

Le candidat au voyage a été acheté pour une bouchée de pain en 2007. Il est simple et léger, sa compacité lui permet de rentrer facilement dans le sac à dos. Son transformateur, par contre, est lourd et donné pour 9V alternatif (non redressé). Il peut fournir 600mA, ce qui semble un peu surdimensionné, surtout que le port USB est (normalement) limité à 500mA.

À l'ouverture, on repère rapidement la zone dédiée à l'alimentation. Elle se situe derrière le connecteur d'alimentation, et la piste qui y est connectée donne sur un pont de diodes (4 diodes effectuant un redressement complet). Une première inductance filtre les hautes fréquences pour éviter de les réinjecter sur le secteur par le câble d'alimentation. Ensuite, deux petits condensateurs lissent la tension (bien petits d'ailleurs, surtout pour un circuit devant consommer 600mA). Enfin, un circuit à découpage de type MC34063A abaisse la tension au moyen d'une inductance, une diode et quelques autres composants.

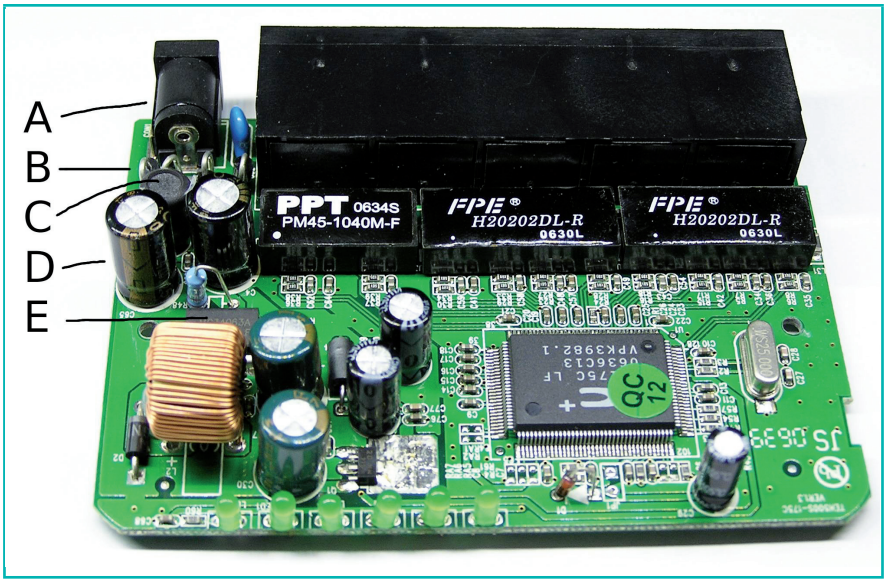


Figure 1 : Le circuit imprimé interne du hub. A : le connecteur d'alimentation. B : le pont de diodes. C : l'inductance d'antiparasitage. D : les condensateurs de filtrage. E : le circuit MC34063A qui contrôle l'abaisseur de tension à découpage.

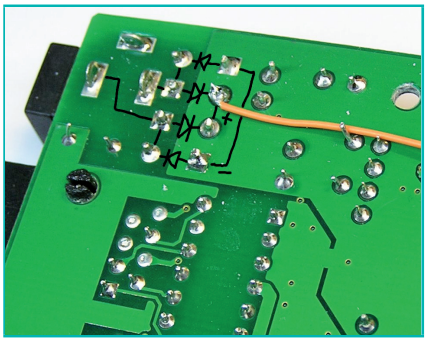
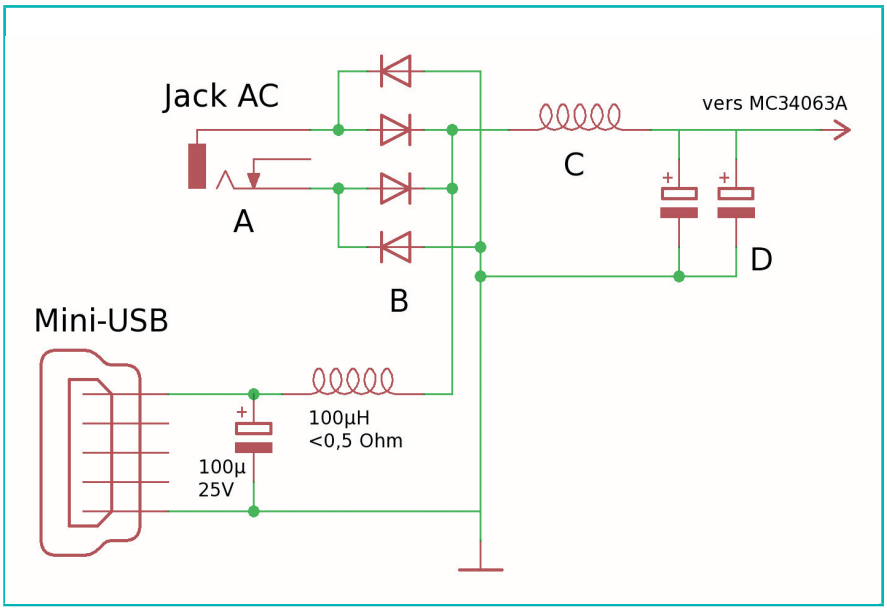


Figure 2 : Connexion d'une source de courant supplémentaire à la sortie du pont de diodes

Il faut noter que les diodes à l'entrée font chuter la tension de 2×0,7V=1,4V, donc si on alimente le hub en passant par le connecteur habituel avec 5V, on obtiendra au mieux 3,6V. Est-ce suffisant ?

Pour découvrir les caractéristiques réelles du hub, le meilleur moyen est de connecter une alimentation de laboratoire juste à la sortie du pont de diodes. Avec 9V, le courant est assez faible (environ 90mA) et il augmente à mesure qu'on réduit la tension (ce qui est caractéristique d'une alimentation à découpage). Avec 5V, la consommation est de moins de 150mA : c'est ce qu'on voulait savoir ! Par contre, un peu en-dessous (vers 4,5V), le hub ne démarre plus correctement.

La sortie du régulateur à découpage, mesurée aux bornes du condensateur électrolytique de filtrage, est de 3,3V. C'est juste ce qu'il faut pour alimenter le circuit intégré central qui remplit toutes les fonctions logiques du hub. C'est 1,7V sous la tension fournie par le port USB et le circuit fonctionne encore sans souci lorsqu'on fournit 5V après le pont de diodes. On peut donc passer à la suite.



2 Ajouter un connecteur

Ensuite, pour le côté pratique, la solution rapide et sale est de prendre un câble USB existant, de couper son connecteur B et de souder directement les fils en sortie du pont de diodes. Évidemment, cela manque de style et cela gâche un câble qui pourrait très bien servir à autre chose, par exemple connecter ou recharger un autre appareil. En voyage, il vaut mieux éviter le matériel redondant.

Le choix a été fait d'installer un connecteur mini USB, puisque déjà deux câbles correspondants sont du voyage.

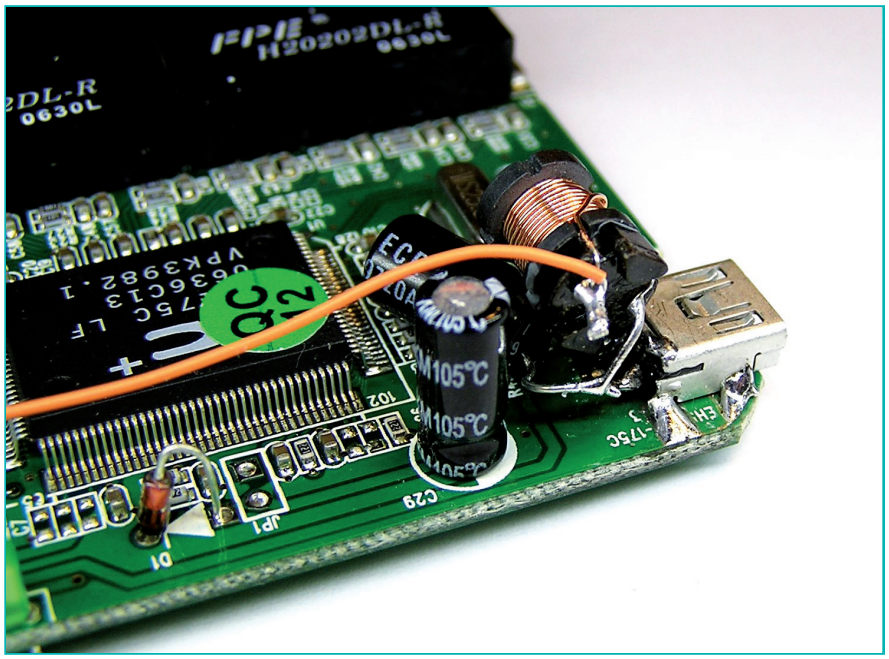


Figure 3 : Le filtre LC est monté « en l'air » avec des composants faciles à trouver.

Un connecteur a été récupéré d'un autre appareil en panne et il a été installé sur le circuit imprimé, soudé sur le plan de masse pour assurer la solidité mécanique. En grattant le vernis vert, on met à jour la surface cuivre sur laquelle on peut souder facilement.

Le brochage du connecteur mini USB est simple : le 0V d'un côté et le 5V de l'autre. Pour repérer le sens, il vaut mieux avoir recours au voltmètre, puisque les dessins sur Internet ne mentionnent jamais si la prise est vue de face ou de dos... Pour pouvoir les souder facilement, les pattes du connecteur sont redressées et celles qui ne servent pas sont coupées.

3 Filtrage supplémentaire de l'alimentation

Comme l'alimentation à découpage hache le courant à environ 50-100KHz, il vaut mieux surfiltrer l'alimentation pour éviter que cela perturbe l'ordinateur fournissant le 5V. Un filtre RC convient, il est composé ici d'un condensateur électrolytique de 100µF 25V et d'une inductance de 100µH.

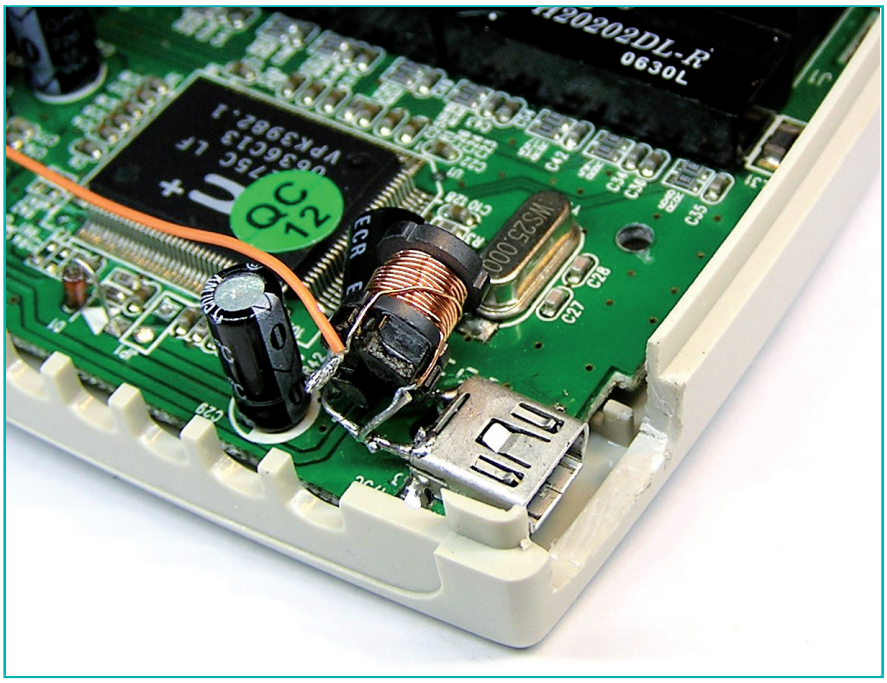


Figure 4 : Le polystyrène de la coque est facile à travailler avec des outils classiques, on peut donc dégager une ouverture qui permet au câble d'accéder au connecteur.

Le choix de ces composants est un peu délicat. Leur présence n'est pas indispensable, puisque cela fonctionne très bien sans, mais si on les met, il vaut mieux s'assurer de quelques détails.

Par exemple, la tension de fonctionnement du condensateur est importante, car si on alimente le hub avec le transformateur normal, la tension peut monter à 20V. Il est toujours recommandé de prendre de la marge sur la tension et un modèle à 25V convient.

Pour l'inductance, un autre paramètre entre en jeu : sa résistance doit être faible ou elle ne permettra pas de faire passer assez de courant pour alimenter le circuit. C'est d'ailleurs ce qui s'est produit la première fois : le hub ne démarrait pas et j'ai découvert que la résistance de l'inductance était de plus de 4 Ohms. Un autre modèle de seulement 0,4 Ohms fonctionne sans souci et le hub peut enfin démarrer.

Les valeurs de capacité et d'inductance ont été choisies un peu surdimensionnées. Le standard USB préconise une capacité maximale plus faible que nos 100 µF, car sinon, le courant d'appel peut être trop important lors du branchement d'un appareil sur le port. Ici, on ne va pas faire certifier notre appareil comme étant conforme aux normes, et ce n'est pas comme si on mettait 0,1F. En plus, le hub sera le seul sur le port, donc il ne devrait pas perturber d'autres appareils.

Le condensateur doit être du côté USB, car lors d'une déconnexion, l'inductance risque de créer une surtension dangereuse. L'inductance réagit aux variations de courant en compensant par une variation de tension à ses bornes. Lors d'une déconnexion, le courant passe brusquement de 150mA à 0mA. Pour maintenir le courant, l'inductance produit une surtension instantanée, que notre gros condensateur encaisse sans souci.

Conclusion

Évidemment, puisque l'alimentation à découpage accepte des tensions variables en entrée (de 5V à facilement 15V), les options de sources d'alimentation sont nombreuses. Le « Power Over Ethernet » n'est pas envisageable, d'une part à cause de la tension trop élevée (48V) et de son indisponibilité sur les ordinateurs portables. Mais un pack de batteries ou de piles, ou même des panneaux solaires (tels qu'utilisés pour recharger des batteries de voitures) feraient l'affaire.

Cela ne règle pas la question de tous les autres câbles (Ethernet et alimentation des *laptops*), mais un de moins, c'est déjà une bonne nouvelle ! C'est même à se demander pourquoi cela n'existe pas déjà dans le commerce... ■

Figure 5 : Le hub Ethernet dispose maintenant d'une prise pour l'alimentation auxiliaire. Attention à ne pas brancher simultanément les deux sources !

DIFFUSEZ VIDÉOS ET MUSIQUE SUR LE RÉSEAU

La Digital Living Network Alliance (DLNA) est une alliance regroupant plus de 250 sociétés, fabricants de matériels et éditeurs de logiciels. L'objectif de la DLNA est de définir un standard d'interopérabilité permettant la lecture, le partage et le contrôle d'appareils multimédias.

Le standard DLNA repose sur un modèle client/serveur. Les clients sont les périphériques et les matériels destinés à afficher les contenus multimédias. Côté serveur, nous avons les appliances comme les NAS personnels et les ordinateurs dispensant le contenu multimédia. Le standard DLNA et le protocole UPnP sont fortement imbriqués. L'*Universal Plug and Play* (UPnP) est un protocole réseau dont l'objectif est de simplifier la construction de réseaux personnels ou d'entreprise. L'un des principaux avantages d'UPnP est de reposer entièrement sur des standards de communication. UPnP distingue deux types d'éléments. Nous avons d'une part les périphériques et d'autre part les points de contrôle. Ces derniers sont les éléments qui permettent de manipuler les premiers et leurs données.

UPnP AV (pour UPnP *Audio and Video*) est un groupe à l'intérieur du standard UPnP, supervisé par la DLNA (voir plus haut). Cet ajout à UPnP est constitué d'un ensemble de composants :

- UPnP MediaServer DCP : le serveur UPnP qui partage ses médias audio, vidéo, images avec les clients UPnP du réseau.
- UPnP MediaServer ControlPoint : le client UPnP qui détecte les serveurs UPnP pour visionner leurs fichiers.
- UPnP MediaRenderer DCP : un dispositif esclave destiné à afficher du contenu (cadre photo numérique, TV, etc.)
- UPnP RenderingControl DCP : un dispositif permettant de contrôler les paramètres de rendu d'un contenu (volume audio, par exemple)
- UPnP *Remote User Interface client/server* : clients ou serveurs UPnP pouvant envoyer des commandes sur le réseau (lecture, pause, stop, etc.).

Notez que les spécifications n'incluent en aucune manière de mécanisme d'authentification ou de contrôle du contenu. Un dispositif UPnP comme un disque réseau, par exemple, diffusera son contenu à n'importe quel point de contrôle UPnP MediaServer sans discrimination.

Il existe plusieurs serveurs UPnP/DLNA disponibles en open source et sous GNU/Linux ou *BSD. L'idée est ici d'en trouver un qui soit suffisamment simple pour éventuellement être utilisé dans un système embarqué. Celui-ci existe et s'appelle judicieusement MiniDLNA. Écrit en C par Justin Maggard, il offre un nombre suffisant de fonctionnalités tout en réduisant les dépendances au minimum. Celles-ci sont tout de même conséquentes du fait des informations à obtenir des différents médias servis (vidéos, images, musique, etc.).

MiniDLNA n'existe pour l'instant pas en version packagée dans Debian ; pour un test, vous devrez donc utiliser les sources disponibles sur http://sourceforge.net/projects/minidlna/. Notez que la version *trunk* d'OpenWrt intègre cet applicatif dans les *feeds*, il suffira donc de faire **./scripts/feeds update** puis **./scripts/feeds install -d m minidlna** pour configurer MiniDLNA et ses dépendances pour une installation modulaire.

La compilation des sources ne pose pas de problème du moment où l'on dispose des paquets de développement adéquats, soit **libexif-dev**, **libjpeg62-dev**, **libid3tag0-dev**, **libflac-dev**, **libvorbis-dev**, **libsqlite3-dev**, **libavformat-dev**, **uuid-dev**, **libavcodec-dev** et **libavutil**. Une fois le binaire obtenu avec un simple **make**, on se penchera sur le fichier de configuration :

```
port=8200
network_interface=eth0
media_dir=/chemin/vers/medias
friendly_name=MyDLNAServer
inotify=yes
enable_tivo=no
strict_dlna=no
notify_interval=900
serial=12345678
model_number=1
```

Les essais se feront simplement via la commande **./minidlna -R -d -f ./minidlna.conf** où **-R** demande une analyse des médias, **-d** active le mode *debug* (le processus ne se détache pas du terminal) et **-f** précise le fichier de configuration.

Il n'est pas nécessaire de redémarrer le serveur pour chaque changement dans le répertoire des médias. En effet, rappelons que le protocole UPnP prévoit un système de notification qui saura prendre tout cela en charge automatiquement. MiniDLNA utilise **inotify** pour détecter les changements dans le système de fichiers et actualiser la liste des médias.

La mise en œuvre d'un serveur UPnP est un jeu d'enfant. On comprend mieux, dès lors, pourquoi les fabricants de NAS intègrent ces fonctionnalités dans leur équipement. Avec un *firmware* basé sur GNU/Linux, ajouter ce genre de service revient généralement à compiler MiniDLNA pour la plateforme et à permettre la gestion du fichier de configuration. ■

IMPLÉMENTATION DE SYSTÈMES CRITIQUES DIRIGÉE PAR DES MODÈLES

par Julien Delange et Maxime Perrotin

La production de systèmes critiques (contrôle de véhicule, détection de fautes, drones, etc.) requiert le respect de nombreuses exigences : ils opèrent dans des environnements contraints (domaine avionique, spatial, militaire) et s'exécutent sur des plateformes embarquées ayant des ressources limitées (capacité de calcul, taille mémoire). De plus, une erreur dans leur implémentation peut avoir de lourdes conséquences (abandon d'une mission, perte de vie) si bien que leur code doit être exempt de bogue. La conception de tels systèmes demande donc un processus de développement rigoureux, s'appuyant sur des technologies détectant tout potentiel vecteur d'erreur. Cet article présente une chaîne d'outils implémentant de tels systèmes au travers d'un cas pratique : l'implémentation d'un drone d'exploration avec Linux.

L'implémentation de systèmes dits « critiques » impose un processus de développement rigoureux : une erreur survenant à l'exécution peut avoir de lourdes conséquences (cas de systèmes du domaine avionique, spatial ou encore militaire). Afin de réduire les potentielles fautes lors de leur production, il est nécessaire de vérifier les exigences du système et détecter toute erreur au plus tôt dans le processus de développement.

Cet article présente une chaîne d'outils dédiée à l'implémentation de systèmes critiques : TASTE. Celle-ci est le fruit de mois de travaux de collaboration entre plusieurs partenaires dans le cadre d'un projet Européen **[ASSERT]**. L'idée générale consiste à capturer les aspects fonctionnels et architecturaux du système à l'aide de modèles et de générer automatiquement le code via des outils dédiés. Cette approche supprime le facteur d'erreur humain (écriture manuelle de code) et assure que les exigences décrites à haut niveau par l'utilisateur (contraintes temporelles, exigences mémoire) sont bien respectées dans l'implémentation (code).

Au cours des sections suivantes, ce processus est illustré par l'implémentation d'un drone embarqué. Celui-ci est composé de capteurs (acquisition du niveau de luminosité, de son et distance) et embarque plusieurs fonctions typiques d'un système embarqué critique (contrôle des moteurs, acquisition de données, etc.). Ce cas d'étude est reproductible à faible coût : la partie matérielle peut être acquise pour une somme modique (< 100€) et les outils sont disponibles sous licence GPL ou open source. Cet article est donc une bonne opportunité pour s'initier à la robotique et acquérir quelques bonnes pratiques de génie logiciel !

1 Aperçu du processus proposé, introduction à TASTE

Le processus abstrait chaque aspect du système par la définition de modèles. Ces derniers sont ensuite utilisés par un ensemble d'outils qui vérifient leur bonne constitution et génèrent automatiquement l'implémentation. Trois types de modèles doivent être définis :

1. **Données** (« *Data View* ») : types (entiers, flottants, etc.) et contraintes associées (bornes, etc.).
2. **Fonctions** (« *Interface View* ») : description des aspects applicatifs, de leurs interfaces (communications, données partagées) et de leurs propriétés (exécution cyclique, sporadique, attente d'événements, ...).
3. **Architecture** (« *Deployment View* ») : définition de la plateforme matérielle (processeurs, bus, périphériques), de sa configuration (adressage, etc.) et de la répartition des fonctions sur chaque processeur.

Les sections suivantes introduisent chaque type de modèle avec le langage qu'ils utilisent.

1.1 Data View

La *Data View* décrit les types de données utilisés dans le système. Elle repose sur l'utilisation du langage ASN.1 **[WIKIASN1]**, très utilisé dans le secteur des télécommunications pour la description des protocoles. Elle garantit une cohérence des données : les types définis sont réutilisés dans les autres modèles (*Interface* et *Deployment view*), assurant alors que les deux modèles utilisent les mêmes types. De plus, cette description est utilisée à des fins d'implémentation (définition du type, génération des fonctions d'encodage/décodage) dans différents langages (Ada, C). Le code généré offre une représentation homogène des types définis, affranchissant le programmeur de la gestion des contraintes architecturales (*endianness*, taille des mots mémoire, etc.).

1.2 Interface View

L'*Interface View* décrit les fonctions du système (ex. : système de pilotage). Chaque fonction est définie par :

1. **Un nom** (par exemple : `fault_detection` pour une fonction se chargeant de détecter les erreurs).
2. **Des interfaces** qui représentent les opérations supportées par la fonction et offrent la possibilité de partager des données ou invoquer du code. Une interface peut être fournie par une fonction (on parle alors de *Provided Interface*) ou requise (une fonction a besoin d'une opération d'une autre fonction, on parle alors de *Required Interface*). Une interface peut avoir des paramètres entrants ou sortants, caractérisés par des types référençant la Data View. Une interface peut être :
   a. **Cyclique** : invoquée périodiquement à intervalles réguliers, elle n'a aucun paramètre.
   b. **Sporadique** : exécutée à la réception d'une donnée entrante, elle peut avoir un ou plusieurs paramètres entrants. Il n'y a aucune synchronisation entre appelant (tâche envoyant les données) et appelé (tâche recevant les données) : la communication est similaire au concept de *oneway* en CORBA, pour les lecteurs familiers avec les intergiciels.
   c. **Protégée** : exécutée dans le contexte de la tâche appelante, elle peut avoir plusieurs paramètres en entrée et sortie et elle bloque l'exécution de toutes les autres interfaces entrantes de la fonction lors de son exécution (utilisation de verrous).
   d. **Non-protégée** : similaire aux interfaces protégées, elle ne bloque cependant pas l'exécution des autres interfaces de la fonction, mais peut ainsi créer des problèmes de cohérence de données (*race condition*). Pour cette raison, leur code doit être écrit avec une attention particulière !
3. **Des propriétés** : langage d'implémentation (C, Ada), etc.

L'Interface View spécifie également les connexions entre interfaces fournies (Provided Interfaces) et requises (Required Interface). Par exemple, dans un système de pilotage, une fonction `fault_detection` fournirait une interface `get_motor_speed` requise par une fonction `motor_control`. Ainsi, le code de la fonction `motor_control` pourrait invoquer cette interface pour fournir la valeur de la vitesse des moteurs, laissant la fonction `fault_detection` effectuer de savants calculs définissant si une erreur est survenue ou non.

1.3 Deployment View

La *Deployment View* décrit l'aspect matériel du système par l'intermédiaire des composants processeurs, périphériques et bus :

- Un processeur est caractérisé par un nom (l'identifiant dans le système distribué) et représente l'architecture du système (x86, PowerPC, SPARC, etc.) mais également le système d'exploitation qu'il exécute (actuellement, Linux, RTEMS et OpenRavenscar - ORK - sont supportés).
- Un périphérique est associé à un processeur et définit sa configuration (nommage, adresse IP, etc.).
- Un bus spécifie le(s) protocole(s) et/ou norme(s) qu'il implémente (ethernet, spacewire, etc.).

Enfin, chaque fonction de l'Interface View est associée au processeur qui l'exécute, décrivant ainsi la distribution des fonctions sur chaque nœud de l'architecture.

1.4 Résumé des vues et inter-dépendances

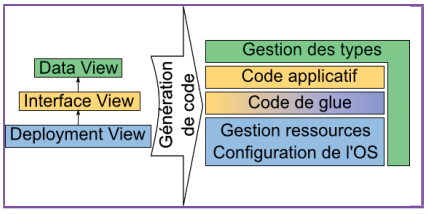


Fig. 1 : Dépendances entre modèles utilisés

L'illustration 1 montre les dépendances et l'utilité des différents modèles dans le

processus d'implémentation et de génération de code. On notera par ailleurs que la Data View permet d'avoir une cohérence des types entre la couche applicative et l'architecture : les définitions des types de données sont utilisées aussi bien dans l'aspect fonctionnel qu'architectural. Le recours à une notation unique et la génération des fonctions d'encodage/décodage permettent de garantir que les données seront cohérentes quels que soient les choix du concepteur du logiciel (système d'exploitation) ou du matériel (architecture du processeur).

2 Présentation du cas d'étude : pilotage d'un drone

Nous proposons de développer un système de contrôle à distance d'un drone motorisé avec notre chaîne d'outils. Il a l'avantage de mettre en œuvre les techniques usuelles de commandes automatisées et d'acquisition de données (récupération de valeurs en provenance de capteurs). L'architecture générale est composée de deux machines :

1. Un commandeur (qui envoie les ordres – aller à gauche/droite, etc. - et affiche les données des capteurs)
2. Un contrôleur qui reçoit les commandes effectue les traitements nécessaires (ajustement des moteurs et récupération des valeurs des capteurs), réalise l'acquisition des données et les envoie au commandeur.

Ces deux nœuds fonctionnent sous Linux et sont connectés physiquement par un réseau ethernet/wifi.

Afin de faciliter l'interfaçage avec les moteurs et autres capteurs, nous utilisons une carte arduino **[ARDUINO]**. Connectée au nœud contrôleur (cf. illustration 2), elle règle la tension à laquelle sont soumis les moteurs et reçoit les données des capteurs. Nous ferons un usage très basique des possibilités de la carte, le lecteur intéressé pourra se référer au portail francophone dédié à cette carte **[ARDUINOFR]** pour plus d'informations.

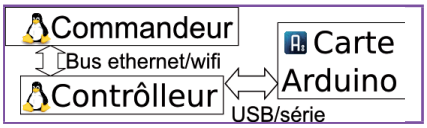


Fig. 2 : Architecture de notre système

2.1 Assemblage de votre propre robot

Le système proposé est facilement reproductible. Les machines contrôleur et commandeur sont de simples PC sous Linux. Les capteurs et le moteur sont interfacés à la machine contrôleur via une carte arduino et ardumoto (pour la commande de deux moteurs). L'ensemble de ces composants s'acquière pour une somme modique (coût de l'ensemble inférieur à 100€). L'illustration 3 montre un assemblage rustique du drone.

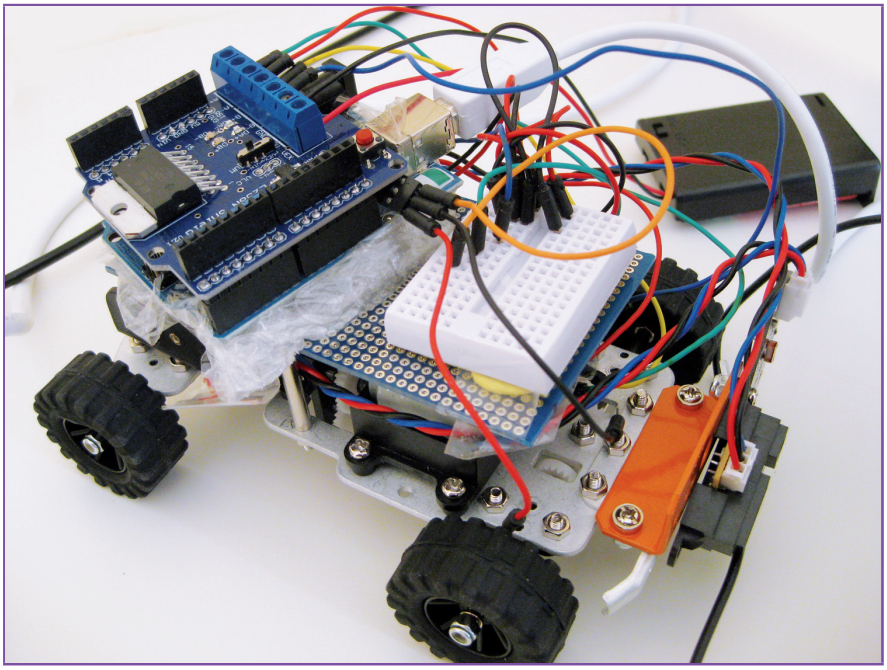


Fig. 3 : Drone assemblé avec la carte arduino et les capteurs

D'un point de vue assemblage, la carte arduino dispose de plusieurs ports d'entrée/sortie en PWM (*Pulse With Modulation*). En entrée, ils acquièrent la valeur des capteurs (valeurs comprises entre 0 et 255). En sortie, ils ajustent la vitesse des moteurs (plus la valeur est élevée, plus le moteur ira vite). La figure 4 montre le montage électronique réalisé : chaque capteur est connecté à la tension d'entrée (3V ou 5V suivant les capteurs) et à la masse. Puis, chaque capteur est connecté à une entrée PWM. La carte ardumoto, quant à elle, est connectée automatiquement sur les PIN 12 et 13 de la carte arduino.

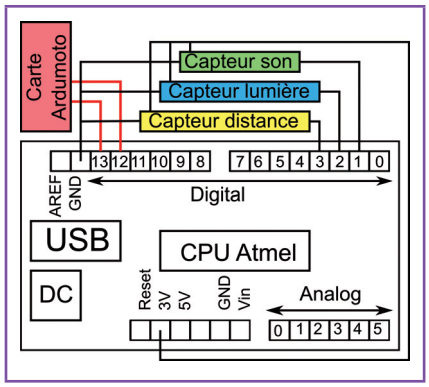


Fig. 4 : Montage des capteurs et de l'ardumoto sur la carte arduino

2.1.1 Code de contrôle déployé sur le robot

Pour communiquer entre le robot et la machine contrôleur, nous définissons notre propre protocole utilisé à travers le

port série. Celui-ci définit deux types de trames pour régler la tension des moteurs et acquérir les valeurs des capteurs :

- **Trames arduino vers contrôleur**. Rapportant les valeurs des capteurs, elles sont composées de 6 octets :
  1. La lettre « L » ;
  2. La valeur du capteur de lumière
  3. La lettre « S » ;
  4. La valeur du capteur de son
  5. La lettre « D » ;
  6. La valeur du capteur de distance.
- **Trames contrôleur vers arduino**. Transportant les commandes, elles sont composées de 3 octets :
  1. Une lettre spécifiant la direction : « L » (*left*), « R » (*right*), « F » (*forward*) ou « B » (*backward*) ;
  2. La valeur de la commande, intensité du mouvement (valeur entre 0 et 255) ;
  3. La lettre « E » (*End of command*).

Pour gérer ce protocole, la carte arduino utilise en boucle la fonction **`loop()`** qui :

1. Lit la valeur des capteurs ;
2. Envoie la valeur des capteurs :
   a. Envoi de la lettre « L » puis de la valeur du capteur correspondant à l'intensité de la lumière,
   b. Envoi de la lettre « S » puis de la valeur du capteur correspondant à l'intensité du son,
   c. Envoi de la lettre « D » puis de la valeur correspondant au capteur de détection de distance ;
3. Si plus de trois octets sont disponibles dans le tampon de réception du port série, alors :
   a. La carte lit le premier octet correspondant à l'identifiant de la commande (lettre L, R, F ou B),
   b. Lit l'octet suivant correspondant à l'intensité du mouvement ou à l'identifiant du capteur à désactiver,
   c. Effectue le traitement correspondant à la commande (avancer/reculer/...),
   d. Lit l'octet suivant jusqu'à ce que celui-ci soit la lettre « E » (End of command).

Le code suivant implémente la gestion de ce protocole à l'aide du langage dédié à la carte arduino. Pour le déployer sur la carte, il faudra le recopier dans l'environnement de développement spécifique à la carte (et disponible sous forme de *package* pour toutes les bonnes distributions et toujours disponible sur le site officiel **[ARDUINO]** pour ceux n'ayant pas la chance d'avoir des outils packagés) et l'envoyer par le port série.

Bien entendu, il faudra que ce protocole soit également géré par la machine contrôleur, nous ne manquerons pas de détailler cela dans la suite de l'article.

```
int lightPin = 2;
int soundPin = 1;
int distancePin = 3;
int dir1PinA = 13;
int dir2PinA = 12;
int speedPinA = 10;
int dir1PinB = 11;
int dir2PinB = 8;
int speedPinB = 9;

uint16_t lightValue          = 0;
uint16_t soundValue       = 0;
uint16_t distanceValue   = 0;

void setup ()
{
   Serial.begin(9600);
}

byte tmp;
byte order;
byte orderval;
byte orderfin;
char orderfinchar;
char orderchar;
uint8_t val;

void loop()
{
  lightValue = analogRead (lightPin);
  soundValue = analogRead (soundPin);
  distanceValue = analogRead (distancePin);
  Serial.print("L");
  val = lightValue / 4;
  Serial.write(&val,1);
  Serial.print("S");
  val = soundValue / 4;
  Serial.write(&val,1);
  Serial.print("D");
  val = distanceValue / 4;
  Serial.write(&val,1);

  if (Serial.available() >= 3)
 {
    order = Serial.read();
    orderval = Serial.read();

    orderchar = (char) order;
    switch (orderchar)
    {
      case 'F':
      {
        analogWrite(speedPinA, 250);
        analogWrite(speedPinB, 250);
        digitalWrite(dir1PinA, HIGH);
        digitalWrite(dir2PinA, LOW);
        digitalWrite(dir1PinB, HIGH);
        digitalWrite(dir2PinB, LOW);
        break;
        break;
      }

      case 'B':
      {
        analogWrite(speedPinA, 250);
        analogWrite(speedPinB, 250);
        digitalWrite(dir1PinA, LOW);
        digitalWrite(dir2PinA, LOW);
        digitalWrite(dir1PinB, HIGH);
        digitalWrite(dir2PinB, HIGH);
        break;
      }

      case 'L':
      {
        analogWrite(speedPinA, 125);
        analogWrite(speedPinB, 250);
        digitalWrite(dir1PinA, HIGH);
        digitalWrite(dir2PinA, HIGH);
        digitalWrite(dir1PinB, LOW);
        digitalWrite(dir2PinB, LOW);
        break;
      }

      case 'R':
      {
        analogWrite(speedPinA, 250);
        analogWrite(speedPinB, 125);
        digitalWrite(dir1PinA, HIGH);
        digitalWrite(dir2PinA, HIGH);
        digitalWrite(dir1PinB, LOW);
        digitalWrite(dir2PinB, LOW);
        break;
      }
      default:
        break;
    }

    orderfinchar = 'B';
    while (orderfinchar != 'E')
    {
      orderfin = Serial.read();
      orderfinchar = (char) orderfin;
    }
  }
  delay(500);
}
```

2.1.2 Téléchargement et installation de la chaîne d'outils

La chaîne utilisée est composée de plusieurs outils qu'il vous faudra télécharger :

1. **TASTE** pour la modélisation de l'Interface View et la Deployment View. Cet outil est disponible sur le site officiel de la société Brestoise Ellidiss **[ELLIDISS]**.

2. Les outils **ASN.1** pour générer le code de définition et de manipulation des données. Cet ensemble de programmes est disponible sur le site web de Semantix **[SEMANTIX]**.
3. **Ocarina** pour générer le code de l'architecture et `buildsupport` pour générer le code de glue entre le code applicatif et le code d'architecture. Ces deux programmes sont disponibles sur le site officiel de TASTE **[TASTE]**.

Un programme additionnel (TASTE GUI), disponible sur le site officiel du projet **[TASTE]**, offre la possibilité d'orchestrer le processus de développement à partir d'une interface graphique. Il est toutefois possible de s'affranchir de cette interface et de réaliser chaque opération à partir du shell. C'est la solution que nous utiliserons au cours de cet article.

À noter que l'installation et le déploiement de la suite d'outils peuvent être facilités :

1. Par l'exécution d'un installeur dédié qui télécharge les sources des logiciels et les installe automatiquement. Ce programme est disponible sur le site officiel du projet **[TASTE]**.
2. Par l'utilisation d'une machine virtuelle disponible sur le site du projet **[TASTE]**. Elle contient une Debian préinstallée avec tous les outils déployés. À noter que cela nécessite l'utilisation du programme gratuit (mais propriétaire) VMWare Player.

2.2 Technologies utilisées

2.2.1 ASN.1 pour la modélisation des données (Data View)

ASN.1 **[WIKIASN1]** est un standard de représentation des données couramment utilisé dans le secteur des télécommunications. Il définit une syntaxe **[ASN1REFCARD]** pour la spécification de types de données. L'avantage est de décrire les types sans se préoccuper de leur implémentation : leur spécification est ensuite utilisée pour générer automatiquement leurs définitions et fonctions d'encodage. Il affranchit ainsi le développeur de toute contrainte d'implémentation (gestion de l'endianess, ...)

L'outillage (aussi bien commercial que libre) autour d'ASN.1 est conséquent. Dans le cadre du projet TASTE, la société SEMANTIX **[SEMANTIX]** a développé un outil spécifique (ASN1Scc) de génération de code à partir de modèles ASN.1. Un nouvel outil était nécessaire, car le code gérant les données doit respecter les contraintes inhérentes aux systèmes critiques (faible empreinte mémoire, absence d'allocation dynamique, etc.).

Le listing 1 donne un exemple de type ASN.1 et spécifie un type `My-Integer` comme un nombre entier ayant une valeur de 0 à 65535.

```
My-Integer ::= INTEGER (0 .. 65535)
```

Listing 1 – Définition d'un entier avec ASN.1

À partir de cette définition, ASN1Scc implémente le type (écriture de `typedef`, déclaration de macros spécifiant la taille requise, etc.) et les fonctions d'encodage/décodage. Le listing 2 donne un exemple illustrant le code généré à partir de la description précédente.

```
typedef asn1SccSint asn1SccMy_Integer;

#define asn1SccMy_Integer_REQUIRED_BYTES_FOR_ENCODING             2
#define asn1SccMy_Integer_REQUIRED_BITS_FOR_ENCODING              16
#define asn1SccMy_Integer_REQUIRED_BYTES_FOR_ACN_ENCODING      2
#define asn1SccMy_Integer_REQUIRED_BITS_FOR_ACN_ENCODING       16

#ifndef ERR_asn1SccMy_Integer
#define ERR_asn1SccMy_Integer                                                 1000 /* (0..65535) */
#endif

void asn1SccMy_Integer_Initialize(asn1SccMy_Integer* pVal);
flag asn1SccMy_Integer_IsConstraintValid(const asn1SccMy_Integer* val, int* pErrCode);
flag asn1SccMy_Integer_Equal(const asn1SccMy_Integer* , const asn1SccMy_Integer* );
flag asn1SccMy_Integer_Encode(const asn1SccMy_Integer* val, BitStream* pBitStrm, int* pErrCode, flag bCheckConstraints);
flag asn1SccMy_Integer_Decode(asn1SccMy_Integer* pVal, BitStream* strm, int* err);
flag asn1SccMy_Integer_ACN_Encode(asn1SccMy_Integer* val, BitStream* strm, int* err, flag bCheckConstraints);
flag asn1SccMy_Integer_ACN_Decode(asn1SccMy_Integer* pVal, BitStream* strm, int* err);
```

Listing 2 – Code C généré à partir de la description ASN.1 d'un entier

L'instruction `typedef` (ligne 1) définit le type de la donnée. Ensuite, plusieurs macros C (`#define`) spécifient les contraintes indépendantes de la machine (nombre d'octets requis pour encoder/décoder la donnée lorsqu'elle doit être extraite d'un flux binaire) ainsi que les erreurs que peuvent retourner les fonctions d'encodage/décodage (non-respect des contraintes, par exemple, si la valeur contenue est supérieure à 65535). Enfin, les fonctions définies permettent d'encoder/décoder le type spécifié, de comparer deux variables contenant une valeur de ce type, etc.

L'utilisation d'ASN1Scc au sein de notre chaîne d'outils présente plusieurs avantages :

1. Il assure une représentation cohérente des types de données indépendamment de la machine, garantissant leur bonne gestion au sein d'un système distribué contenant des nœuds ayant des architectures hétérogènes.
2. Il génère automatiquement le code chargé d'encoder/décoder les données pour les échanger à travers des réseaux, garantissant alors l'absence d'erreur dans la gestion des communications.

2.2.2 AADL

AADL **[AADL]** est un standard pour la modélisation d'architectures. Il définit plusieurs composants (logiciels : *process*, *thread*, *subprogram* matériels : *processor*, *bus* ; et hybrides) pour décrire un système distribué avec ses contraintes de configuration et de déploiement (architecture, protocoles, etc.). Contrairement à d'autres langages de modélisation uniquement graphiques, AADL offre une triple représentation : graphique, textuelle et XML. Cette spécificité permet d'utiliser le langage aussi bien à des fins de communication entre concepteurs et développeurs (représentation graphique) que pour l'exploiter par des outils (*parsing* et analyse des fichiers textuels).

Notre chaîne d'outils ne nécessite pas une écriture manuelle des modèles AADL : ces derniers sont générés automatiquement par les outils graphiques capturant les aspects fonctionnels (Interface View) et architecturaux (Deployment View) du système.

Plusieurs programmes sont actuellement disponibles pour la manipulation de modèles AADL : OSATE (framework utilisant Eclipse), ADELE, etc. Les lecteurs les plus curieux pourront visualiser ou modifier les fichiers AADL générés par nos outils à l'aide de leur éditeur de texte favori (des modes vim et emacs sont disponibles). Ceux souhaitant en apprendre plus sur ce langage pourront se référer à **[AADLTUTORIAL]**.

2.3 Spécification du système

Après cette courte introduction aux technologies utilisées par nos outils, nous présentons chaque vue (Data/Interface/Déploiement) de notre système pour les spécifier à l'aide de nos outils.

2.3.1 Définition des données

Notre système échange deux principaux types de données :

1. **les télécommandes** (commandeur vers contrôleur) pour l'envoi d'ordre au drone (avancer, etc.).
2. **les télémétries** (contrôleur vers commandeur) contenant les valeurs des capteurs placés sur le drone.

Les ordres contenus dans les données de télécommandes ainsi que les télémétries sont codés sur des nombres entiers. Pour des raisons inhérentes à notre implémentation, ces derniers prennent une valeur allant de 0 à 255.

2.3.2 Fonctions

La machine de commande exécute deux fonctions :

1. **keyboard**, dédiée à la gestion du clavier, contenant une interface cyclique qui active le système, acquiert les données du périphérique et les envoie à la fonction **car_command**.
2. **car_command**, utilisée pour gérer l'envoi de télécommandes au drone en fonction des valeurs en provenance du clavier et l'affichage des rapports de télémétries reçus. Cette fonction définit deux interfaces :
   1. **receive_inputs** : réception des événements en provenance du clavier. À la réception d'une donnée sur cette interface, le code correspondant envoie une télécommande sur l'interface **receive_tc** de la fonction **car_controller**. En cas de modification de la configuration du contrôleur, la configuration mise à jour est envoyée sur l'interface **set_configuration** de la fonction **car_controller**.
   2. **receive_tm** : traitement des télémétries en provenance du contrôleur. Lorsqu'une donnée est reçue sur cette interface, le code activé met à jour l'interface graphique avec les nouvelles valeurs des capteurs.

La machine de contrôle exécute deux fonctions :

1. **car_controller** dédiée au traitement des télécommandes du commandeur et à l'envoi des télémétries ;
2. **arduino_handler** spécifique à la communication matérielle avec la carte arduino (acquisition des capteurs et contrôle des moteurs).

La fonction **car_controller** définit 8 interfaces :

1. **receive_tc** : réception d'une télécommande. Lorsqu'elle est déclenchée, cette interface effectue un traitement pour déplacer le drone en fonction des données contenues dans la télécommande.
2. **set_configuration** : utilisée pour modifier la configuration du drone (nombre de capteurs activés, etc.) ;
3. **set_light** : modification de la valeur du capteur de lumière ;
4. **set_temp** : modification de la valeur du capteur de température ;
5. **set_vibration** : modification de la valeur du capteur de vibration ;
6. **set_sound** : modification de la valeur du capteur d'acquisition du niveau sonore ;
7. **set_distance** : modification de la valeur du capteur de distance ;
8. **activator** : via l'interface **receive_tm** les valeurs courantes des capteurs. C'est cette interface qui assure une réception périodique des valeurs des capteurs.

La fonction **arduino_handler**, quant à elle, ne contient qu'une interface cyclique : **activator**. Celle-ci exécute un code de commande qui acquiert les données en provenance des capteurs et les envoie à la fonction **arduino_handler** via les interfaces que cette dernière fournit (**set_light**, **set_temp**, **set_vibration**, **set_sound**, **set_distance**).

À noter enfin que toutes ces fonctions sont implémentées en langage C. Cependant, l'approche proposée n'est pas limitée à ce langage, les fonctions pouvant être implémentées aussi bien avec des langages dits « traditionnels » (tels C ou Ada) qu'avec des modèles applicatifs (SDL/RTDS, Simulink, SCADE, ...).

2.3.3 Description de l'architecture

L'architecture de notre cas d'étude est plutôt simple. Elle s'articule autour de deux machines :

1. La machine de commande, à laquelle est relié le périphérique d'acquisition des entrées en provenance de l'utilisateur (modification de la direction du drone ou de sa configuration). Elle utilise un OS de type Linux sur un processeur x86. Elle exécute les fonctions de gestion du clavier (fonction **keyboard**) ainsi que la gestion de contrôle des télécommandes et l'affichage des télémétries (fonction **car_command**).
2. La machine de contrôle du drone, connectée à la carte arduino pour s'interfacer avec les capteurs et les moteurs. Cette machine fonctionne sous un OS de type Linux avec une architecture x86. Elle exécute les fonctions **arduino_handler** (interface avec la carte arduino pour l'acquisition des valeurs des capteurs) et **car_controller** (réception des télécommandes et adaptation du comportement du drone en fonction de la configuration reçue).

Ces deux machines sont reliées par un bus afin d'échanger les télécommandes, télémétries et changements de configuration. Ce bus peut être un simple réseau ethernet RJ45/wifi. D'un point de vue configuration, il sera nécessaire que l'utilisateur définisse dans ses modèles de déploiement (Deployment view) le type d'OS déployé sur chaque machine ainsi que la configuration des périphériques réseau (adresse IP/port).

2.4 Création des modèles

Une fois les types de données spécifiés, les fonctions décrites et l'architecture explicitée, il reste à créer les modèles correspondants. Les trois parties abordent la modélisation de chaque aspect (Data View, Interface View et Deployment View) à l'aide des outils appropriés.

2.4.1 Types de données (Data View)

À partir de la spécification précédente des types de données, nous sommes en mesure de spécifier les types ASN.1 associés (cf. listing 3). Les types suivants sont définis :

- **Input-T** : touche détectée par la fonction de gestion du clavier (**keyboard**) et envoyée à la fonction **car_command** via l'interface **receive_inputs**. Dans notre cas, nous considérons que seuls 8 événements sont disponibles : 4 pour déplacer le drone et 4 pour contrôler sa configuration.
- **Telecommand-T** : contenu d'un paquet de télécommande envoyé par la fonction **car_command** à la fonction **car_controller** via l'interface **receive_tc**. Il contient le type de direction à prendre par le drone et y associe un nombre, définissant la puissance d'accélération. Par exemple, une donnée définissant le membre **go-backward** avec une valeur de 200 fera reculer le drone plus rapidement que la même donnée contenant la valeur 50.
- **Temperature-T**, **Sound-T**, **Vibration-T**, **Distance-T** et **Light-T** : types utilisés pour la transmission des valeurs des capteurs entre la fonction **arduino_handler** et la fonction **car_controller**.
- **Telemetry-T** : contenu d'un paquet de télémétrie envoyé par la fonction de contrôle (**car_controller**) à la fonction de commande (**car_command**) via l'interface **receive_tm**. Il contient un champ par donnée acquise sur les capteurs.
- **Configuration-T** contient les données de configuration du drone. Ce type est utilisé comme paramètre de l'interface **set_configuration** entre la fonction **car_command** et **car_controller**.

```
DataView DEFINITIONS AUTOMATIC TAGS ::= BEGIN

   Input-T ::= CHOICE
   {
     left         BOOLEAN,
     right        BOOLEAN,
     up           BOOLEAN,
     down         BOOLEAN,
     action1      BOOLEAN,
     action2      BOOLEAN,
     action3      BOOLEAN,
     action4      BOOLEAN
   }

   Telecommand-T ::= CHOICE
   {
     go-left               INTEGER (0 .. 254),
     go-right              INTEGER (0 .. 254),
     go-forward          INTEGER (0 .. 254),
     go-backward       INTEGER (0 .. 254)
   }

   Temperature-T  ::= REAL (0 .. 100)
   Sound-T             ::= INTEGER (0 .. 255)
   Vibration-T        ::= INTEGER (0 .. 255)
   Distance-T         ::= INTEGER (0 .. 255)
   Light-T               ::= INTEGER (0 .. 255)

   Telemetry-T ::= SEQUENCE {
     temperature      Temperature-T,
     sound                 Sound-T,
     vibration            Vibration-T,
     distance             Vibration-T,
     light                   Light-T
   }
   Configuration-T ::= SEQUENCE {
      acquire-temperature BOOLEAN,
      acquire-sound            BOOLEAN,
      acquire-vibration       BOOLEAN,
      acquire-distance        BOOLEAN,
      acquire-light              BOOLEAN,
      acquisition-delay       INTEGER (0 ..
255),
      change                         BOOLEAN
   }
END
```

Listing 3 – Définition des types utilisés dans notre architecture

2.4.2 Types de données : export d'ASN.1 vers AADL

Le modèle de données défini, il est nécessaire de l'exporter en AADL afin de pouvoir référencer les types dans nos outils de définition des fonctions (Interface View) et d'architecture (Deployment View). La commande suivante exporte les types ASN.1 définis (fichier **dataview.asn**) en AADL (génération d'un fichier **dataview.aadl**), permettant ainsi de les référencer dans les fonctions et l'architecture du système.

```
% asn2aadlPlus.py -aadlv2 dataview.asn
dataview.aadl
```

2.5. Architecture logicielle (Interface View)

L'outil **TASTE-IV** permet de définir l'aspect fonctionnel du système. Avant de commencer à créer ou modifier fonctions et interfaces, il est nécessaire de charger les données du système. Pour cela, chargez le fichier **`dataview.aadl`** précédemment généré dans la section précédente à partir du menu ***File > Load Data View*** (cf. figure 5).

Une fois la définition des données importée, on peut commencer à définir les fonctions en utilisant le menu ***Edit > New Function***. Il est ensuite nécessaire de lui assigner un nom en éditant ses propriétés (cf. figure 6) et de définir son langage d'implémentation. Au cours de cet article, toutes les fonctions seront écrites en langage C, probablement le plus populaire parmi notre lectorat. Il est également nécessaire de définir les interfaces fournies (*Provided Interface*, *PI*) ou requises (*Required Interface*, *RI*), en les caractérisant par leur type (**cyclique** – voir les propriétés en figure 7 - ou **sporadique**, voir les propriétés en figure 8). À noter qu'une interface cyclique est toujours une interface fournie (PI) ; elle est activée périodiquement à fréquence fixe, ne prend pas de paramètre et ne peut pas être invoquée par une autre fonction. À l'inverse, une interface sporadique, elle, peut avoir un ou plusieurs paramètres (cf. figure 9), est appelée de manière asynchrone par une autre fonction via une Required Interface (RI), permettant ainsi l'échange de données entre fonctions.

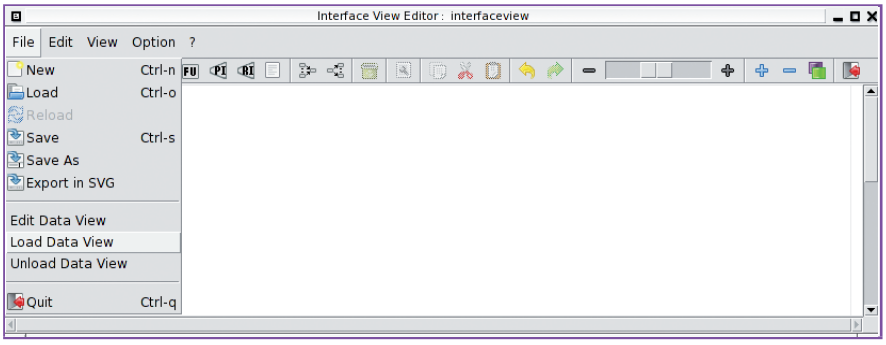


Fig. 5 : Chargement de la Data View dans TASTE-IV

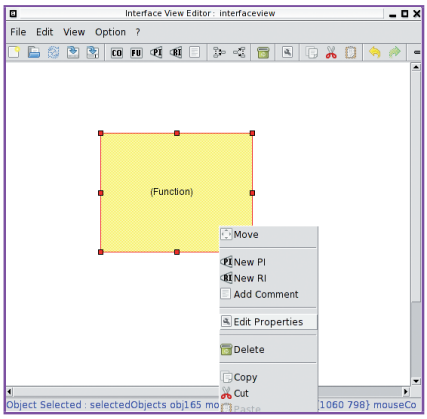


Fig. 6 : Ajout d'une fonction à l'Interface View

À partir de la description des 4 fonctions au cours des précédentes sections, nous pouvons déduire leurs interfaces. Ainsi, la fonction **`keyboard`** définit 2 interfaces :

1. Une interface *cyclique* (**`polling`**), ayant une période de 100ms. Elle sera chargée de vérifier la présence de données entrantes sur le clavier (commandes frappées par l'utilisateur).
2. Une RI pour envoyer les données reçues sur le clavier et connectée à la PI **`receive_inputs`** de la fonction **`car_command`**.

La fonction **`car_command`** contient 4 interfaces :

1. Une PI sporadique pour la réception des événements du clavier : **`receive_inputs`** avec un paramètre de type **`Input-T`**.
2. Une PI sporadique pour la réception des télémétries en provenance de la fonction **`car_control`** et prenant donc un paramètre de type **`Telemetry-T`** : **`receive_tm`**.
3. Une RI vers la PI **`receive_tc`** de la fonction **`car_control`** pour envoyer les télécommandes.
4. Une RI vers la PI **`set_configuration`** de la fonction **`car_control`** pour notifier un changement de configuration.

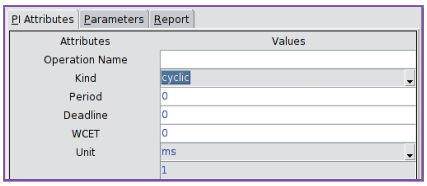


Fig. 7 : Édition des propriétés d'une interface périodique/cyclique

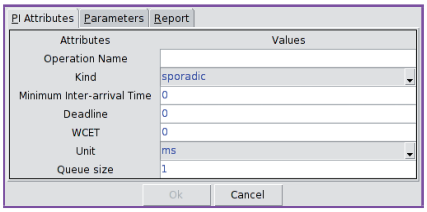


Fig. 8 : Édition des propriétés d'une interface sporadique

La fonction **`car_control`** dispose de 9 interfaces :

1. Une interface cyclique chargée d'activer périodiquement la fonction toutes les 500ms : **`activator`**.
2. Une PI sporadique pour la réception de télécommandes en provenance de la fonction **`car_command`** avec un paramètre de type **`Telecomand-T`** : **`receive_tc`**.
3. Une PI sporadique chargée de recevoir toute nouvelle configuration et prenant donc un paramètre de type **`Configuration-T`** : **`set_configuration`**.
4. Une PI sporadique pour la réception de la valeur du capteur de son (paramètre de type **`Sound-T`**) : **`set_sound`**.
5. Une PI sporadique pour la réception de la valeur du capteur de lumière (paramètre de type **`Light-T`**) : **`set_light`**.

6. Une PI sporadique pour la réception de la valeur du capteur de distance (paramètre de type **Distance-T**) : **set_distance**.
7. Une PI sporadique pour la réception de la valeur du capteur de température (paramètre de type **Temperature-T**) : **set_temperature**.
8. Une PI sporadique pour la réception de la valeur du capteur de vibration (paramètre de type **Vibration-T**) : **set_vibration**.
9. Une RI utilisée pour envoyer les rapports de télémétries et connectée à la PI **receive_tm** de la fonction **car_command**.

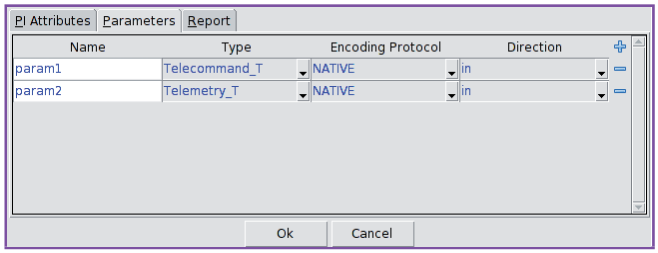


Fig. 9 : Édition des paramètres d'une interface sporadique

Enfin, la fonction **arduino_handler** définit quant à elle 6 interfaces :

1. Une interface cyclique qui active périodiquement la fonction afin de gérer le matériel (acquisition de la valeur des capteurs, stockage dans des buffers/variables globales, communications avec la carte arduino par le port série, etc.) : **activator**.
2. Une RI pour l'envoi de la valeur associée au capteur de température et connectée à la PI **set_temperature** de la fonction **car_controller**.
3. Une RI utilisée pour la transmission de la valeur du capteur de lumière et connectée à la PI **set_light** de la fonction **car_controller**.
4. Une RI connectée à la PI **set_vibration** de la fonction **car_controller** (envoi de la valeur du capteur de vibration).
5. Une RI associée à la PI **set_distance** de la fonction **car_controller** pour l'échange de la valeur du capteur de distance du drone.
6. Une RI pour l'envoi de la valeur associée au capteur de son et connectée à la PI **set_sound** de la fonction **car_controller**.

Une fois l'édition de votre modèle terminée, vous devriez obtenir un joli dessin avec quatre rectangles, quelques flèches et de nombreuses lignes connectant le tout, comme l'illustre la figure 10. Au-delà de l'aspect purement graphique, le lecteur doit garder à l'esprit que ce schéma représente l'architecture

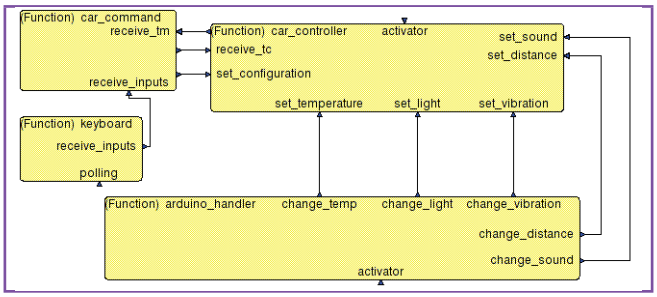


Figure 10 : Interface View du système

logicielle logique du système avec les relations et propriétés de chaque fonction. Il est désormais nécessaire de décrire leur distribution au sein de l'architecture matérielle (architecture physique), ce qui est abordé dans la section suivante au travers de l'écriture d'une *Deployment View*.

2.6 Architecture matérielle (Deployment View)

Une fois l'aspect fonctionnel défini, il est nécessaire de décrire son organisation au sein de l'architecture à l'aide de la Deployment View qui définit l'architecture matérielle (bus, mémoire, ...) et associe chaque fonction à une entité d'exécution. L'outil **TASTE-DV** définit la plateforme et ses caractéristiques : deux processeurs x86 avec un OS de type Linux et connectés par un réseau ethernet. Afin de créer ou éditer une Deployment View, il est nécessaire de passer en argument l'Interface View contenant les fonctions que l'on souhaite associer à l'architecture. L'outil doit donc être invoqué de la manière suivante (nous faisons le prédicat que votre Interface View a été sauvegardée dans le fichier **interfaceview.aadl**) :

```
TASTE-DV -load-interface-view interfaceview.aadl
```

Deux *processor boards* sont donc ajoutés au modèle : **cmd_board** et **control_board**. Un processor board représente un nœud du système et est constitué :

- d'un composant **memory** pour le stockage du code et des données du système, inutilisé actuellement.
- d'un composant **processor** spécifiant le processeur physique (x86, powerpc) et l'OS qu'il exécute (Linux, RTEMS, etc.). La figure 11 illustre la configuration d'un processor d'architecture x86 sous Linux.
- d'un ou plusieurs composants **partition** (concept similaire au processus UNIX) contenant les fonctions de l'Interface View. Dans notre cas, toutes les fonctions seront placées dans une seule partition.
- d'un **driver** représentant le périphérique de communication connecté aux composants bus. Ces derniers spécifient par ailleurs les drivers les implémentant ainsi que leur paramètre de configuration (adressage, etc.)

En plus des processor boards, la Deployment View permet d'ajouter des composants bus. Ceux-ci spécifient les protocoles (TCP, UDP, etc.) qu'ils supportent et connectent les composants device des processors boards, décrivant ainsi l'aspect distribué de l'architecture. Dans notre cas, un bus est ajouté pour connecter les deux périphériques réseau.

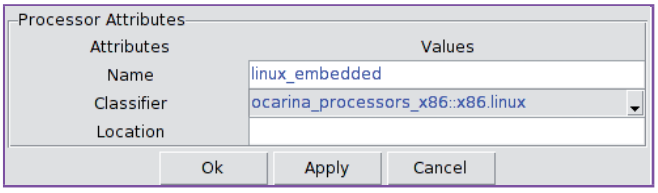


Figure 11 : Configuration du processeur

Chaque composant **processor** doit définir le système d'exploitation par l'édition de ses propriétés. Au vu de notre lectorat potentiel et ne souhaitant pas nous risquer à utiliser des OS peu sûrs, nous décidons que tous les processeurs de notre système exécuteront un OS de type Linux. La figure 11 montre la fenêtre des propriétés du composant permettant de capturer cette exigence.

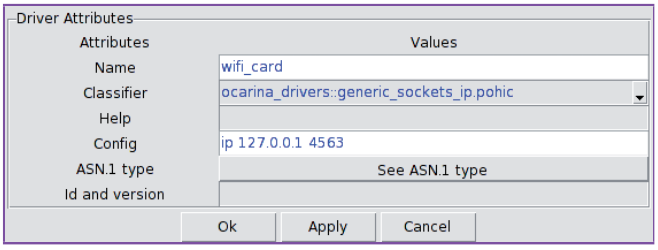


Figure 12 : Configuration du périphérique réseau

De la même manière, chaque composant **driver** (carte réseau dans notre cas) doit être configuré (assignation d'une adresse IP et d'un port). Le type de périphérique est spécifié dans le champ ***Classifier :*** nous choisirons ici le type **generic_sockets_ip.pohic** qui spécifie une interface réseau générique (utilisation des sockets BSD). Enfin, la configuration du périphérique est enregistrée dans le champ ***Config***. La syntaxe de configuration est la suivante : **ip address port**. La partie **address** spécifie l'adresse IP de la carte réseau et **port** le port TCP sur lequel le pilote recevra les paquets réseau entrants. Par exemple, la configuration **ip 127.0.0.1 4563** signifie que le pilote sera associé à la carte réseau portant l'adresse IP 127.0.0.1 et écoutera les requêtes entrantes sur le port 4563. La figure 12 montre la fenêtre de capture de ces paramètres au sein de l'outil.

Enfin, les propriétés du composant bus reliant les cartes doivent être éditées pour spécifier le type de protocole qu'il fournit. Utilisant un simple bus IP standard, nous choisirons la valeur **ocarina_buses::ip.i** pour le champ ***Classifier*** (cf. figure 13).

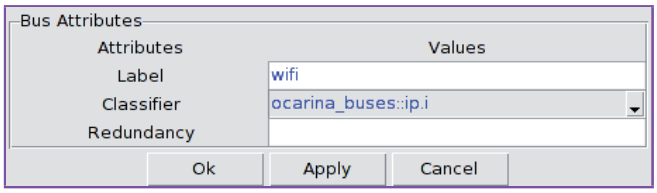


Figure 13 : Configuration du type de bus

Maintenant la partie matérielle décrite, il nous faut associer les fonctions aux plateformes. Ici, le nœud de calcul **cmd_board** exécute deux fonctions : acquisition des touches clavier (**keyboard**) et envoi des télécommandes / affichage des télémétries (**car_command**). Nous ajoutons donc deux composants fonctions à la partition du processor board **cmd_board**.

Le nœud de contrôle du drone exécute également deux fonctions : réception des télécommandes / envoi des télémétries (**car_controller**) et contrôle de la carte arduino (**arduino_handler**). Comme précédemment, nous associons ces deux fonctions à la partition du processor board **control_board**. La figure 14 illustre la Deployment View une fois complétée avec les composants matériels et les fonctions associées.

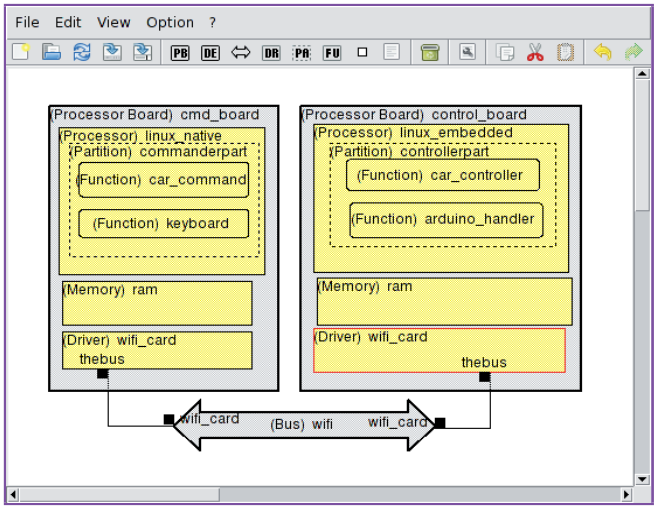


Figure 14 : Aperçu global de la Deployment View

3 Implémentation

Les précédentes sections se sont attachées à la description abstraite des aspects fonctionnels et architecturaux du système. Le but étant toutefois de créer un système exécutable, il est nécessaire d'écrire le code correspondant aux fonctions définies. Les sections suivantes détaillent comment, à partir de ces modèles abstraits, l'implémentation peut être automatiquement générée. Toutefois, si notre approche tend à automatiser tout le processus de développement pour éviter les erreurs introduites par les *geeks* que forme notre lectorat, quelques lignes de code représentant l'aspect comportemental du système restent à écrire !

3.1 Génération de squelettes applicatifs

Afin d'aider le développeur et lui éviter de commettre des erreurs dans l'écriture de son code, l'outil **buildsupport** génère les squelettes du code de chaque fonction, définissant les prototypes C des interfaces fournies. La commande suivante invoque l'outil qui analyse les modèles et produit ce code.

```
% buildsupport -aadlv2 -polyorb-hi-c -d dataview.aadl -c
deploymentview.aadl -i interfaceview.aadl -gw  -o code_skels
```

Le répertoire **code_skels** est créé avec un sous-répertoire par fonction, constitué des fichiers suivants :

- **dataview.h** : définition des types de données ASN.1 utilisés par la fonction. Ce fichier est présent à titre purement informatif, afin que l'utilisateur ait connaissance du contenu des types de données générés en vue de leur utilisation.
- **asn1crt.h** décrit les types génériques (entiers, flottants, ...) utilisés par les fonctions ASN.1.
- **nom_fonction.c** et **nom_fonction.h** contiennent les prototypes des routines C de la fonction **nom** de l'Interface View.

Pour chaque fonction de l'Interface View, une routine **nom_fonction_startup** est générée pour gérer son initialisation (assignation de valeurs à des variables globales, etc.). Puis, pour chacune de ses interfaces, les routines C suivantes sont créées :

- Pour une interface cyclique, une routine C sans paramètre et portant le nom de l'interface est générée. L'utilisateur écrit donc le code exécuté par cette interface dans le fichier C correspondant.
- Dans le cas d'une Provided Interface (PI) sporadique (cas de données entrantes), une routine **nom_fonction_PI_nom_interface** est générée et prend en paramètres les types définis dans l'Interface View.
- Dans le cas d'une Required Interface (RI) sporadique, un prototype est généré dans le fichier d'en-tête afin que l'utilisateur ait connaissance de la fonction à appeler pour invoquer la fonction à distance. Le code implémentant les données ne doit pas être écrit et est automatiquement généré.

Les listings 4 et 5 illustrent les squelettes applicatifs associés à la fonction **car_command**. Elle supporte deux PI : **receive_inputs** et **receive_tm**.. Par conséquent, le squelette applicatif contient trois fonctions :

1. la fonction d'initialisation (**car_command_startup()**) ;
2. une fonction implémentant la PI **receive_inputs** (**car_command_PI_receive_inputs()**) ;
3. une fonction implémentant la PI **receive_tm** (**car_command_PI_receive_tm()**).

Le lecteur assidu aura remarqué que les types des arguments de ces deux dernières fonctions correspondent à des types générés à partir des types définis à l'aide des modèles ASN.1.

```
#include "car_command.h"
void car_command_startup()
{
        /* Write your initialization code here,
           but do not make any call to a required interface!! */
}

void car_command_PI_receive_inputs(const asn1SccKeycode *IN_input)
{
        /* Write your code here! */
}

void car_command_PI_receive_tm(const asn1SccTelemetry_T *IN_tm)
{
        /* Write your code here! */
}
```

Listing 4 – Squelette applicatif du code de la fonction commandeur

```
#include "keyboard.h"
void init_keyboard()
{
        /* Write your initialization code here,
           but do not make any call to a required interface!! */
}

void keyboard_polling()
{
        /*
         * Write your code here!
         */
}
```

Listing 5 – Squelette applicatif de la fonction keyboard

3.1.1 Écriture du code fonctionnel à partir des squelettes

Si les outils nous assistent dans le processus de conception du système en proposant une abstraction de nombreux concepts, il reste toutefois l'aspect fonctionnel à écrire. Il serait en plus dommage de nous priver de toute activité d'écriture de code !

Les squelettes étant générés automatiquement, l'utilisateur doit simplement les compléter. Le code généré requiert que le code comportemental de chaque interface fournie (PI) soit écrit. Dans le cas d'une interface requise (Required Interface, RI), l'utilisateur n'a pas à fournir de code fonctionnel : une RI correspond à une connexion requise, c'est-à-dire un appel de fonction à distance. Pour ce type d'interface, le code généré par **buildsupport** fournit le prototype des routines fournies par l'environnement pour invoquer ces interfaces à distance (voir les fichiers d'en-tête générés).

3.1.1.1 Fonction keyboard

Le listing 6 suivant présente le code d'implémentation de la fonction **keyboard**. Afin d'acquérir les touches tapées par l'utilisateur, le code utilise la bibliothèque curses **[WIKICURSES]** via deux routines :

1. **init_keyboard**, qui effectue des appels à des routines externes pour initialiser la bibliothèque curses.
2. **keyboard_polling**, correspondant à la PI **polling**. Le code associé capture la touche reçue par la bibliothèque curses et la transmet à la fonction **car_command** via un appel à sa PI **receive_inputs**.

```
/* Functions to be filled by the user (never overwritten by buildsupport tool) */
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <getopt.h>

#include "keyboard.h"
#include "dataview.h"

void init_keyboard() /* Initialisation de la fonction keyboard */
{
   WINDOW* win = initscr();
   keypad (stdscr, TRUE);
   noecho ();
   nodelay (win, TRUE);
}

void keyboard_polling() /* Code de la PI cyclique polling de la fonction keyboard */
{
   asn1SccKeycode key_coded;
   int key;
   key = getch ();
   if (key != ERR)
   {
      key_coded = key;
      vm_async_keyboard_receive_inputs (&key_coded, sizeof (asn1SccKeycode));
      /* Appel de la RI receive_inputs */
   }
}
```

Listing 6 : Code de la fonction « keyboard »

3.1.1.2 Fonction car_command

Comme pour la fonction **keyboard**, elle utilise la bibliothèque curses **[WIKICURSES]**, mais cette fois-ci à des fins d'affichage pour présenter les valeurs des données de télémétries.

Pour séparer le code et avoir un design compréhensible, nous définissons les routines suivantes :

- **send_configuration()** : notification de changement de configuration et envoi de la nouvelle configuration sur la RI **set_configuration** (appel de la routine **car_command_RI_set_configuration()**).
- **configure_light ()**, **configure_sound()**, **configure_distance()**, **configure_temperature()** modifient la valeur associée à l'acquisition des capteurs de lumière, son, distance ou température. Ces fonctions sont appelées à chaque fois que l'utilisateur active/désactive leur acquisition.
- **print_telemetry()** affiche les rapports de télémétries via la bibliothèque curses. Elle est appelée par la PI **receive_tm** à chaque fois que de nouvelles données sont reçues en provenance de la fonction **car_control**.

Le code associé aux interfaces définies dans l'Interface View est défini dans les routines suivantes :

- **car_command_PI_receive_inputs()** qui réalise un traitement en fonction du code de la touche reçue. Suivant la valeur du paramètre ***IN_input** (changement de direction ou de configuration), le code génère soit une donnée de télécommande, soit une nouvelle configuration à destination du nœud **car_controller**.
- **car_command_PI_receive_tm()**, exécutée à la réception de données de télémétries en provenance de la fonction **car_control**, affiche les données reçues par un appel à la fonction **print_telemetry()**.

```
#include <types.h>
#include <stdio.h>
#include <curses.h>
#include "car_command.h"

asn1SccTelemetry_T        current_telemetry;
asn1SccConfiguration_T  current_configuration;

static WINDOW *mainwnd;
static WINDOW *screen;
WINDOW           *my_win;

void send_configuration ()
{
   current_configuration.change = 1;
   car_command_RI_set_configuration(&current_configuration);
   /* Appel de la PI set_configuration de la fonction car_control */
}

void configure_light ()
{
   if (current_configuration.acquire_light)
   {
      current_configuration.acquire_light = 0;
   }
   else
   {
      current_configuration.acquire_light = 1;
   }
}

void configure_temperature ()
{
   if (current_configuration.acquire_temperature)
   {
      current_configuration.acquire_temperature = 0;
   }
   else
   {
      current_configuration.acquire_temperature = 1;
   }
}

void configure_vibration ()
{
   if (current_configuration.acquire_vibration)
   {
      current_configuration.acquire_vibration = 0;
   }
   else
   {
```

```
      current_configuration.acquire_vibration = 1;
   }
}

void configure_distance ()
{
   if (current_configuration.acquire_distance)
   {
      current_configuration.acquire_distance = 0;
   }
   else
   {
      current_configuration.acquire_distance = 1;
   }
}

void configure_sound ()
{
   if (current_configuration.acquire_sound)
   {
      current_configuration.acquire_sound = 0;
   }
   else
   {
      current_configuration.acquire_sound = 1;
   }
}

void print_telemetry ()
{
   curs_set(0);
   mvwprintw(screen,1,1,"--------- TELEMETRY VALUES ----------");
   mvwprintw(screen,2,1,"-------------------------------------");
   mvwprintw(screen,3,6,"Temperature          : %2.1f", current_telemetry.temperature);
   mvwprintw(screen,4,6,"Sound level          : %3ld", current_telemetry.sound);
   mvwprintw(screen,5,6,"Vibration level      : %3ld", current_telemetry.vibration);
   mvwprintw(screen,6,6,"Distance from object : %3ld", current_telemetry.distance);
   mvwprintw(screen,7,6,"Light level          : %3ld", current_telemetry.light);
   mvwprintw(screen,10,1,"-------------------------------------");
   mvwprintw(screen,11,1,"          Commands");
   mvwprintw(screen,12,1,"        \\_o<  >o_/");
   mvwprintw(screen,13,1,"up/down/left/right : move the robot");
   mvwprintw(screen,14,1,"y : start/stop temperature report");
   mvwprintw(screen,15,1,"u : start/stop sound report");
   mvwprintw(screen,16,1,"i : start/stop vibration report");
   mvwprintw(screen,17,1,"o : start/stop distance report");
   mvwprintw(screen,18,1,"p : start/stop light report");
   wrefresh(screen);
   wrefresh(mainwnd);
   refresh();
}

void car_command_startup()
{
   asn1SccConfiguration_T_Initialize (&current_configuration);
   asn1SccTelemetry_T_Initialize (&current_telemetry);
   current_configuration.acquire_temperature   = 0;
   current_configuration.acquire_sound          = 0;
   current_configuration.acquire_vibration      = 0;
   current_configuration.acquire_distance       = 0;
   current_configuration.acquire_light          = 0;
   current_configuration.acquisition_delay      = 5;
   current_configuration.change                 = 0;
   mainwnd = initscr();
   noecho();
   cbreak();
   nodelay(mainwnd, TRUE);
   refresh();
   wrefresh(mainwnd);
   screen = newwin(20, 40, 1, 1);
   box(screen, ACS_VLINE, ACS_HLINE);
}
```

```
/* Implementation de la PI receive_inputs */
void car_command_PI_receive_inputs(const asn1SccKeycode *IN_input)
{
   asn1SccTelecommand_T tc;
   current_configuration.change = 0;
   asn1SccTelecommand_T_Initialize (&tc);
   switch (*IN_input)
   {
      case 260: /* 260 correspond au code de la touche gauche */
         tc.kind = Telecommand_T_go_left_PRESENT;
         tc.u.go_left = 10;
         car_command_RI_receive_tc (&tc); /* Appel de la PI receive_tc de car_control */
         break;
      case 261: /* 261 correspond au code de la touche droite */
         tc.kind = Telecommand_T_go_right_PRESENT;
         tc.u.go_right = 10;
         car_command_RI_receive_tc (&tc); /* Appel de la PI receive_tc de car_control */
         break;
      case 259: /* 259 correspond au code de la touche haut */
         tc.kind = Telecommand_T_go_forward_PRESENT;
         tc.u.go_forward = 10;
         car_command_RI_receive_tc (&tc); /* Appel de la PI receive_tc de car_control */
         break;
      case 258: /* 258 correspond au code de la touche haut */
         tc.kind = Telecommand_T_go_backward_PRESENT;
         tc.u.go_backward = 10;
         car_command_RI_receive_tc (&tc); /* Appel de la PI receive_tc de car_control */
         break;
      case 121: /* 121 correspond au code de la touche y */
         configure_temperature ();
         send_configuration ();
         break;
      case 117: /* 117 correspond au code de la touche u */
         configure_sound ();
         send_configuration ();
         break;
      case 105: /* 105 correspond au code de la touche i */
         configure_vibration ();
         send_configuration ();
         break;
      case 111: /* 111 correspond au code de la touche o */
         configure_distance ();
         send_configuration ();
         break;
      case 112: /* 112 correspond au code de la touche p */
         configure_light ();
         send_configuration ();
         break;
   }
   print_telemetry ();
}

/* Implémentation de la PI receive_tm */
void car_command_PI_receive_tm(const asn1SccTelemetry_T *IN_tm)
{
       current_telemetry = *IN_tm;
       print_telemetry ();
}
```

Listing 7 : Code de la fonction car_command

3.1.1.3 Fonction car_control

Le listing 8 présente l'implémentation des routines de la fonction **car_control** :

- **car_controller_startup()** initialise la variable globale **configuration** représentant les données de la configuration initiale du système de contrôle.
- **car_controller_PI_receive_command()** implémente la PI **receive_command** exécutée à la réception des paquets de

télécommandes. En fonction de la valeur reçue, la position du robot est modifiée (appel aux fonctions **move_left()**, **move_right()**, **move_forward()** ou **move_backward()**).

- **car_controller_PI_set_temperature()**, **car_controller_PI_set_light()**, **car_controller_PI_set_vibration()**, **car_controller_PI_set_distance()**, **car_controller_PI_set_sound()** implémentent les PI recevant les valeurs des capteurs transmises par la fonction **arduino_handler**. Ces routines modifient la variable globale **telemetry**, cette dernière contenant en permanence les valeurs mises à jour des capteurs.
- **car_controller_PI_set_configuration** modifie la variable globale **configuration**, mettant ainsi à jour la configuration de la fonction.
- **car_controller_PI_activator()**, appelée périodiquement, envoie le contenu de la variable globale **telemetry** (dernières valeurs connues des capteurs) sur la PI **receive_tm** de la fonction **car_command**.

```
#include "car_controller.h"
#include <stdio.h>

asn1SccConfiguration_T  configuration;
asn1SccTelemetry_T      telemetry;

void move_left ();
void move_forward ();
void move_right ();
void move_backward ();

void car_controller_startup()
{
   configuration.acquire_temperature   = 0;
   configuration.acquire_sound           = 0;
   configuration.acquire_vibration       = 0;
   configuration.acquire_distance        = 0;
   configuration.acquire_light           = 0;
   configuration.acquisition_delay       = 5;
   configuration.change                  = 0;
}
/* Implementation de la PI receive_tc */
void car_controller_PI_receive_tc(const asn1SccTelecommand_T *IN_tc)
{
   switch (IN_tc->kind)
   {
      case Telecommand_T_go_left_PRESENT:
         move_left ();
         break;
      case Telecommand_T_go_right_PRESENT:
         move_right ();
         break;
      case Telecommand_T_go_forward_PRESENT:
       move_forward ();
       break;
      case Telecommand_T_go_backward_PRESENT:
         move_backward ();
         break;
      case IN_tc->kind == Telecommand_T_NONE:
         printf("NONE PRESENT\n");
         break;
}

/* Implementation de la PI set_temperature */
void car_controller_PI_set_temperature(const asn1SccTemperature_T *IN_val)
{
   if (configuration.acquire_temperature)
   {
      telemetry.temperature = *IN_val;
   }
}

/* Implementation de la PI set_light */
void car_controller_PI_set_light(const asn1SccLight_T *IN_val)
{
   if (configuration.acquire_light)
   {
      telemetry.light = *IN_val;
   }
}

/* Implementation de la PI set_vibration */
void car_controller_PI_set_vibration(const asn1SccVibration_T *IN_val)
{
   if (configuration.acquire_vibration)
   {
      telemetry.vibration = *IN_val;
   }
}

/* Implementation de la PI set_distance */
void car_controller_PI_set_distance(const asn1SccDistance_T *IN_val)
{
   if (configuration.acquire_distance)
   {
      telemetry.distance = *IN_val;
   }
}

/* Implementation de la PI set_sound */
void car_controller_PI_set_sound(const asn1SccSound_T *IN_val)
{
   if (configuration.acquire_sound)
   {
      telemetry.sound = *IN_val;
   }
}

/* Implementation de la PI set_configuration */
void car_controller_PI_set_configuration(const asn1SccConfiguration_T *IN_val)
{
   if (IN_val->change == 1)
   {
      configuration = *IN_val;
   }
}

/* Implementation de la PI activator executee toutes les 500ms et chargee d'envoyer
   les rapports de telemetries a la fonction car_command via la RI receive_tm */
void car_controller_PI_activator()
{
   car_controller_RI_receive_tm (&telemetry);
   /* Invocation de la PI receive_tm de la fonction car_command */
}
```

Listing 8 : Code de la fonction car_control

3.1.1.4 Fonction arduino_handler

Le code de cette fonction communique avec la carte matérielle arduino via le port série et nécessite donc l'invocation d'appels systèmes pour lire et écrire sur ce port. Pour simplifier la compréhension du code, nous définissons plusieurs fonctions :

- **serialport_writebyte()** pour écrire un octet sur le port connecté à la carte arduino.
- **open_arduino()** qui ouvre le port série connecté à la carte (macro **DEVICE**) et initialise ses paramètres. Dans cette version du système, nous considérons que la carte arduino

est connectée sur le port **/dev/ttyS0** avec une vitesse de 9600 bauds. Malgré tout l'intérêt que nous avons pour la programmation système, nous ne détaillerons pas ici le code pour des raisons évidentes de place (les lecteurs ne connaissant pas les mécanismes d'interfaçage avec le port série pourront toutefois se reporter à la documentation très fournie disponible sur Internet, telle **[TLDP-SERIAL]**).

Puis nous implémentons les interfaces de la fonction capturées dans l'Interface View :

- **init_arduino_handler()** ouvre le port de communication avec la carte arduino (fonction **open_arduino()**).
- **arduino_handler_activator()**, appelée cycliquement, interroge la carte pour récupérer les valeurs associées aux capteurs. C'est cette routine qui interprète le protocole « maison » mis en place sur la carte arduino (voir sections précédentes). Lorsqu'un capteur émet une nouvelle valeur, celle-ci est transmise à la fonction **car_control** via les interfaces **set_temperature**, **set_sound**, **set_distance**, **set_light** ou **set_vibration** (et donc, par un appel à la fonction **vm_arduino_handler_set_temperature()**, **vm_arduino_handler_set_sound()**, **vm_arduino_handler_set_distance()**, **vm_arduino_handler_set_light()** ou **vm_arduino_handler_set_vibration()**).

Maintenant que le code fonctionnel est écrit, l'architecture modélisée, il ne nous reste plus qu'à utiliser les outils pour générer le système et déployer le code sur chaque machine pour mettre en œuvre notre drone piloté !

```
/* Functions to be filled by the user (never overwritten by buildsupport tool) */
#include <types.h>
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <getopt.h>

#include "arduino_handler.h"
#include "dataview.h"

#define NB_ORDERS 3

#define DEVICE "/dev/ttyS0" /* Port serie connecte a la carte arduino */
int arduino; /* Descripteur de fichier associé au port serie connecte a la carte arduino */
uint8_t light_value    = 0;
uint8_t sound_value  = 0;
uint8_t dist_value      = 0;

char orders[] = {'L', 'S', 'D'};

int serialport_writebyte( int fd, uint8_t b)
{
    if (write(fd,&b,1)!= 1)
    {
        return -1;
    }
    return 0;
}

void move_left ()
{
   serialport_writebyte (arduino,'L');
  serialport_writebyte (arduinhttp://tldp.org/HOWTO/Serial-Programming-HOWTO/o,40);
   serialport_writebyte (arduino,'E');
}

void move_forward ()
{
   serialport_writebyte (arduino,'F');
   serialport_writebyte (arduino,40);
   serialport_writebyte (arduino,'E');
}

void move_right ()
{
   serialport_writebyte (arduino,'R');
   serialport_writebyte (arduino,40);
   serialport_writebyte (arduino,'E');
}

void move_backward ()
{
   serialport_writebyte (arduino,'B');
   serialport_writebyte (arduino,40);
   serialport_writebyte (arduino,'E');
}

void open_arduino ()
{
    struct termios toptions;

    arduino = open(DEVICE, O_RDWR | O_NOCTTY);
    if (arduino == -1)
    {
        return -1;
    }

    if (tcgetattr(arduino, &toptions) < 0)
    {
        return -1;
    }

    cfsetispeed(&toptions, B9600);
    cfsetospeed(&toptions, B9600);
    toptions.c_cflag          &= ~PARENB;
    toptions.c_cflag          &= ~CSTOPB;
    toptions.c_cflag          &= ~CSIZE;
    toptions.c_cflag           |= CS8;
    toptions.c_cflag          &= ~CRTSCTS;
    toptions.c_cflag           |= CREAD | CLOCAL;
    toptions.c_iflag           &= ~(IXON | IXOFF | IXANY);
    toptions.c_lflag           &= ~(ICANON | ECHO | ECHOE | ISIG);
    toptions.c_oflag          &= ~OPOST;
    toptions.c_cc[VMIN]    = 0;
    toptions.c_cc[VTIME]   = 20;

    if( tcsetattr(arduino, TCSANOW, &toptions) < 0)
    {
        return -1;
    }
}

/* Initialisation de la fonction arduino_handler */
void init_arduino_handler()
{
   open_arduino ();
}
```

```
/* Implementation de la PI activator de la fonction arduino_handler */
void arduino_handler_activator()
{
   asn1SccSound_T       sound_level;
   asn1SccDistance_T    dist;
   asn1SccLight_T         light_level;
   char c;
   uint8_t valid;
   uint8_t val;
   int i;
   c = 'A';
   valid = 0;
   while (valid == 0)
   {
      read (arduino, &c, sizeof (char));
      for (i = 0 ; i < NB_ORDERS ; i++)
      {
         if (c == orders[i])
         {
            valid = 1;
         }
      }
   }
   read (arduino, &val, sizeof (uint8_t));

   switch (c)
   {
      case 'L':
         light_level = val;
         vm_arduino_handler_set_light (&light_level, sizeof (asn1SccLight_T));
         /* Appel de la RI set_light de la fonction car_control */
         break;

      case 'S':
         sound_level = val;
         vm_arduino_handler_set_sound (&sound_level, sizeof (asn1SccSound_T));
         /* Appel de la RI set_sound de la fonction car_control */
         break;

      case 'D':
         vm_arduino_handler_set_distance (&dist, sizeof
(asn1SccDistance_T));
         /* Appel de la RI set_distance de la fonction car_control */
         break;
   }
}
```

Listing 9 : Code de la fonction arduino_handler

3.2 Compilation, assemblage, exécution

À ce point de l'article, nous sommes en possession de tous les éléments nécessaires à l'implémentation du système : modélisation fonctionnelle, architecturale et code fonctionnel. La figure 15 décrit le processus de génération des applications à partir de ces éléments :

1. **Transformation des Data View, Interface View et Deployment View** en un unique modèle AADL : la *Concurrency View*. Cette étape est appelée « transformation verticale ».
2. **Génération de code C à partir de code AADL**. En particulier, cette étape crée le code instanciant et configurant (périphérique, système d'exploitation) les ressources de chaque nœud du système distribué.
3. **Compilation du code généré avec une runtime** (que certains appelleront *middleware* ou intergiciel) et le code applicatif, produisant ainsi le binaire final.
4. **Exécution sur la plate-forme cible** (ou sur la machine de développement).

3.2.1 La transformation verticale : de la description abstraite à la notation AADL

La transformation verticale consiste à traduire Interface et Deployment *view* en un seul modèle AADL qui respecte la sémantique des fonctions décrites et représente les aspects de concurrence (exécution de tâches, réception d'événements, etc.). En effet, au-delà de la simple description fonctionnelle du système, il est nécessaire de créer ses ressources (threads, données partagées, etc.) requises à l'exécution du code utilisateur et d'assurer que ces dernières se comportent comme spécifié (périodicité d'exécution, etc.).

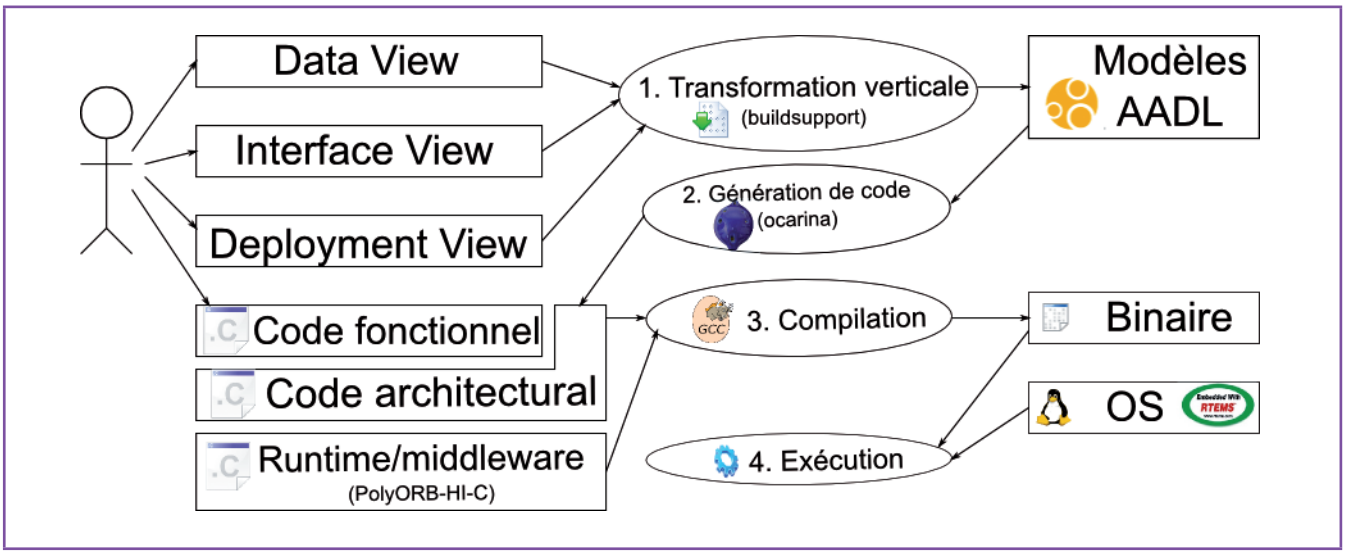


Figure 15 : Processus d'implémentation du système à partir des vues et du code fonctionnel de l'utilisateur

Dans notre chaîne, l'outil **buildsupport** se charge de réaliser cette transformation. Utilisé en ligne de commandes, il prend en paramètres les vues du système et les transforme en un unique modèle AADL. La génération de la *Concurrency View* s'effectue via la commande suivante :

```
buildsupport -aadlv2 -o output -i interfaceview.aadl -d dataview.aadl
-c deploymentview.aadl -onlycv -glue -gw
```

Plusieurs fichiers AADL auront été générés dans le répertoire **output/ConcurrencyView**. Les lecteurs curieux pourront s'intéresser aux détails des mécanismes de la transformation verticale. Cependant, dans notre contexte, nous fournissons une description sommaire de l'architecture générée. Elle est constituée :

- D'une tâche (composant AADL **thread**) pour chaque PI de l'Interface View dont le type correspond à celui de l'interface (cyclique ou sporadique). Les propriétés de l'interface (période, échéance, etc.) sont reportées dans les propriétés AADL du composant généré.
- D'un composant AADL **processor** pour chaque processor board de la Deployment View et associe composants **thread** et **processor** en fonction des associations fonction/processor board.
- D'un composant AADL **device** pour chaque driver de la Deployment View avec ses propriétés de configuration et associé au composant **processor** auquel il est lié dans la Deployment View.
- D'un composant AADL **bus** connectant les composants **device**, décrivant ainsi le lien entre périphériques.

Les mécanismes de transformation entraînent parfois des patrons de génération plus complexes, en particulier lorsque l'on a recours à d'autres types d'interfaces (dites protégées ou non protégées). Cependant, l'idée principale consiste à réunir toutes les informations des vues définies par l'utilisateur en un seul et unique modèle représentant les aspects de distribution et de concurrence du système. Outre le fait que cette vue globale du système facilite sa compréhension, il peut être utilisé à des fins d'implémentation (génération de code) ou de validation (cf. section perspectives). Dans le cadre de cet article, seul l'aspect génération de code est abordé (étape 2 de la figure 15), nous gardons les aspects validation pour d'autres articles !

3.2.2 Génération de code AADL vers C

L'outil **Ocarina** se charge de traduire le modèle AADL produit par **buildsupport** en un code d'implémentation gérant les ressources décrites à haut niveau. En particulier, il :

1. Instancie les tâches du modèle sur chaque processeur ;
2. Dimensionne les ressources par rapport aux besoins du système (taille des buffers, etc.) ;
3. Initialise et configure les périphériques (création de sockets, etc.) ;
4. Génère le point d'entrée de chaque tâche qui appelle le code utilisateur, reçoit et envoie les données en provenance ou à destination des autres tâches du système distribué.

Afin d'assurer une portabilité/maintenabilité et adaptation aux différents OS embarqués, le code généré se doit d'être indépendant de toute API (ex. : POSIX). Pour cela, le code produit par Ocarina est générique et s'appuie sur une *runtime* (ou intergiciel/middleware) qui interface le code généré avec les services de l'OS exécuté. Ainsi, le support de nouveaux OS et/ou architectures s'effectue par une modification de la runtime et non du code généré qui reste indépendant de toute API.

Afin de générer le code implémentant notre système, placez-vous dans le répertoire contenant les fichiers AADL produits par la *vertical transformation* et invoquez la commande suivante :

```
ocarina -r deploymentview.final -f -aadlv2 -g polyorb_hi_c process.
aadl *Thread.aadl ../../interfaceview.aadl ../../deploymentview.aadl
../../dataview.aadl
```

Les options utilisées sont :

- **-r nom_systeme_racine** : spécifie le nom du composant AADL contenant le système principal (appelé aussi système « racine »). Le système racine produit par **buildsupport** se nomme toujours **deploymentview.final**.
- **-f** : charge des fichiers de propriétés spécifiques à AADL et nécessaires à la construction de notre système.
- **-aadlv2** : utilisation de la version 2 du langage AADL.
- **-g polyorb_hi_c** : spécifie que le code généré doit s'interfacer avec la runtime PolyORB-HI-C. Ocarina est capable de générer du code pour d'autres langages et/ou runtime (comme PolyORB-HI-Ada, une runtime écrite en... Ada !). Dans notre cas, nous utiliserons la runtime PolyORB-HI-C, celle-ci s'interfaçant facilement avec Linux.

Une fois la commande exécutée, le code implémentant votre système est stocké dans le répertoire **deploymentview_final**. Le lecteur curieux pourra lire le code produit et constatera sa structure générique :

- Les fichiers implémentant les ressources liées à chaque nœud du système (composant AADL **processor**) sont stockés dans un répertoire séparé.
- Chaque répertoire contient les mêmes fichiers dont la structure est similaire : **main.c** (initialisation), **activity.c** (point d'entrée des tâches), **deployment.c** (configuration et dimensionnement des ressources),...

Maintenant que le code de l'architecture est créé, il est nécessaire de faire entrer en jeu GCC pour le compiler et le lier avec le code fonctionnel pour créer l'implémentation finale (étape 3 de notre figure 15).

3.2.3 Compilation du code fonctionnel et lien avec le code architectural

À cette étape du processus, nous avons tous les éléments sous la forme de fichiers sources :

1. description et implémentation des types de données ;
2. code architectural de chaque nœud ;
3. code applicatif fourni par l'utilisateur et implémenté à partir des squelettes de code.

Il ne nous reste donc plus qu'à assembler toutes ces parties et créer les binaires correspondant à chaque nœud. Cette étape d'assemblage va compiler séparément chaque fichier C avec GCC et les lier ensemble avec le *linker*. Cette opération est réalisée par l'outil **assert-builder-ocarina.py**, qui génère les types de données et le code architectural pour y intégrer le code applicatif. Pour cela, l'outil requiert que le code de chaque fonction soit contenu dans un fichier zip puis passé en paramètre à l'aide de l'option **-C**.

3.2.3.1 Création des archives de chaque fonction

Pour construire l'archive zip de chaque fonction, placez-vous dans le répertoire contenant l'implémentation des fonctions (probablement le répertoire de génération des squelettes), puis invoquez la commande **zip** :

```
zip nom_fonction.zip nom_fonction/*
```

Par exemple, pour la fonction **car_command**, utilisez la commande suivante :

```
zip car_command.zip car_command/*
```

Vous devez créer une archive zip pour chaque fonction. L'archive devra contenir un répertoire contenant le nom de la fonction (convention de l'outil **assert-ocarina-builder.py**). Enfin, pour les plus paresseux, la commande suivante permet de créer tous les fichiers nécessaires si vous l'invoquez dans le répertoire contenant toutes vos fonctions :

```
for v in `find . -maxdepth 1 -mindepth 1 -type d|grep -v .svn`; do zip $v $v/* ; done
```

3.2.3.2 Invocation de l'orchestrateur, création des binaires

Une fois les archives du code fonctionnel créées, invoquez l'outil **assert-builder-ocarina.py** ainsi :

```
USER_LDFLAGS=-lcurses DISABLE_MULTICORE_CHECK=1 \
 assert-builder-ocarina.py  -f -p   -v -o /tmp/mygeneratedsystem \
                 -a dataview.asn -i interfaceview.aadl -c deploymentview.aadl \
                 -C arduino_handler:code/arduino_handler.zip -C keyboard:code/keyboard.zip \
                 -C car_command:code/car_command.zip -C car_controller:code/car_controller.zip
```

Les options suivantes sont utilisées :

- **-p** : spécifie que l'on utilise la runtime C (PolyORB-HI-C) ;
- **-o** : indique le chemin utilisé pour le processus de compilation et le stockage des binaires ;
- **-a** : indique le chemin de la Data View (fichier ASN.1) ;
- **-i** : spécifie le fichier d'Interface View (fichier AADL) ;
- **-c** : spécifie le fichier de Deployment View (fichier AADL) ;
- **-C** : indique l'emplacement de l'archive contenant le code de chaque fonction. La valeur passée à cette option est formatée ainsi : **-C nom_fonction:chemin_archive.zip**. Il faut bien entendu spécifier l'archive contenant chaque fonction, et donc, l'utiliser autant de fois que notre système contient de fonctions.

La variable d'environnement **USER_LDFLAGS** spécifie les bibliothèques externes avec lesquelles les applications générées sont liées, tandis que **DISABLE_MULTICORE_CHECK** supprime certaines vérifications liées à l'exécution du programme dans une machine virtuelle.

3.2.4 Exécution

Une fois le processus de compilation et d'intégration terminé, l'orchestrateur place les binaires dans le sous-répertoire **binaries/** du répertoire spécifié par l'option **-o** lors de son invocation. Dans notre cas, les binaires seront donc stockés dans le répertoire **/tmp/mygeneratedsystem/binaries**.

Il ne vous reste alors plus qu'à copier les binaires sur les machines correspondantes et à les exécuter. Si votre système est correctement configuré (adresse IP des cartes réseau, bonne connexion de la carte arduino, etc.), vous devriez être capable de piloter votre drone au moyen de votre clavier.

3.3 Autres aspects du processus & perspectives

Au travers de cet article, nous avons détaillé l'approche suivie par TASTE pour la création de systèmes critiques. Elle repose sur une spécification du système au moyen de modèles, puis par plusieurs étapes de génération de code, produit les applications de chaque nœud du système distribué. Bien que nous ne les ayons pas détaillées, l'utilisation de modèles, et donc, d'une abstraction du système, offre de nombreuses perspectives dans le processus de conception, les paragraphes suivants présentent quelques possibilités.

3.3.1 Validation du design a priori

Un des principaux avantages apportés par l'utilisation de modèles est la capacité à analyser le système et à détecter ses erreurs au plus tôt dans le processus de développement. Notre méthode permet de décrire tous les aspects du système (fonctions, architecture, configuration, déploiement), il est donc possible de procéder à une évaluation de sa faisabilité avant implémentation. Cela peut revêtir plusieurs formes :

- Vérification sémantique de la cohérence de l'architecture par des analyses de l'intégration des composants matériels (un périphérique pour un bus *spacewire* n'est pas connecté à un bus ethernet)

- Vérification de l'exécution des tâches, de l'absence d'inter-blocage : plusieurs algorithmes d'ordonnancement (tels *Rate Monotonic* **[RATEMON]**) peuvent être validés, et des outils comme Marzhin (intégré à notre chaîne d'outils via **TASTE-CV**, non détaillé dans cet article faute de place) permettent de procéder à de telles vérifications.

- Validation du bon assemblage des composants par l'analyse de leurs propriétés (par exemple, un processor board avec 100Ko de mémoire ne pourra pas accueillir une fonction embarquant 200Ko de données).

Traditionnellement, ce type d'erreur est très difficile à détecter dans un contexte industriel : la taille des systèmes est telle qu'il est compliqué de valider chacun de ses aspects. Ces erreurs sont souvent détectées au cours de tests d'intégration. Or, ceux-ci ont lieu une fois la production du système achevée, et la découverte d'une erreur nécessite alors de réviser à nouveau le code ou le design du système, embarquant alors le développeur dans une modification de son travail initial et à ... une nouvelle série de tests ... Au final, cette phase de test et d'intégration est très coûteuse et peut représenter jusqu'à 70 % du budget d'un projet **[NIST02]**. La validation *a priori* du système constitue donc une plus-value importante, aussi bien en termes de qualité que de temps ou d'argent.

3.3.2 Implémentation de la machine contrôleur sur RTEMS

Pour de nombreuses raisons, les décisions techniques sur les composants utilisés au sein d'un projet peuvent être revues, entraînant alors des changements bien souvent coûteux (et parfois périlleux) : une bonne partie du code déjà produit doit alors être adaptée pour être compatible avec les interfaces et contraintes des nouveaux composants. Par exemple, un code fonctionnel utilisant Linux et POSIX pour la gestion des ressources devra être complètement adapté si le concepteur décide finalement d'utiliser RTEMS **[RTEMS]** et l'interface spécifique de cette plateforme.

Cependant, ce changement n'impacte pas l'aspect « fonctionnel » (quelle que soit la plateforme d'exécution, l'Interface View sera la même) du système en tant que tel : seules les communications avec les couches inférieures (OS, pilotes, etc.) sont impactées.

Notre approche permet justement de gérer ce type de changement tout en affranchissant le développeur d'adapter son code. Un simple changement dans la Deployment View et la modification du type de système d'exploitation affecté à un composant `processor` suffit à adapter le code fonctionnel sur une autre architecture/système d'exploitation. Si vous êtes familier avec RTEMS, nous vous encourageons à changer la propriété du processeur associé au nœud contrôleur, afin de générer un binaire exécutable sur RTEMS. Quelques adaptations seront toutefois nécessaires (en particulier les aspects d'interfaçage avec l'arduino), mais celles-ci sont mineures.

3.3.3 Modification du type de réseau utilisé

Une autre perspective mettant en avant les avantages d'un tel processus de développement réside dans la mise en œuvre de la configuration du système. Par exemple, changer le protocole de communication entre les nœuds d'ethernet à spacewire ne requiert qu'une simple édition de la Deployment View et la modification de trois composants (les cartes réseau – composants de type `driver` - de chaque nœud et des composants `bus` les connectant). De la même manière, la modification de la configuration s'effectue par l'édition des propriétés des composants `driver` (champ ***Config*** associé à un périphérique, cf. figure 12). Cette automatisation de la configuration des périphériques et autres supports de communication affranchit le concepteur du système des efforts d'intégration et de configuration, bien souvent longs et vecteurs d'erreurs.

3.3.4 Ajout de nouvelles touches de contrôle : l'avantage d'ASN.1

Un aspect peu détaillé au cours de cet article, mais constituant l'une des plus-values les plus importantes, réside dans l'utilisation d'ASN.1. Le recours à une description à haut niveau des types, la génération de leurs définitions et la production automatique des fonctions d'encodage/décodage garantissent la cohérence (valeurs, représentation) des données entre les nœuds du système. Ils suppriment toute mauvaise gestion des types et de l'hétérogénéité des processeurs utilisés (gestion de l'endianess, de la taille des mots, etc.), erreurs difficiles à détecter et que l'on retrouve trop souvent dans le domaine de l'embarqué (communications entre architectures hétérogènes, etc.).

À titre d'exemple, dans notre cas d'étude, la gestion d'une nouvelle touche du clavier est très simple et se résume à l'ajout d'un champ au type `Input-T` (cf. listing 3) et à la mise à jour du code applicatif utilisant ce type (fonctions

`keyboard` et `car_command`). Les noms des fonctions empaquetant les données resteront inchangés, évitant de potentielles modifications manuelles du code source.

Conclusion

Cet article a mis en avant la difficulté de conception des systèmes critiques : ils doivent être exempts d'erreurs et respecter des contraintes fortes (ordonnancement, empreinte mémoire, capacité de calcul limitée, etc.). Le recours à des méthodes de développement manuelles devrait donc être évité : la probabilité de la présence d'une faute n'est pas tolérable et doit être détectée au plus tôt dans le processus de conception à l'aide d'outils dédiés.

Nous avons présenté une chaîne d'outils expérimentale pour la création de tels systèmes à partir d'une description de haut niveau (données, fonctions, architecture) et illustré sa mise en œuvre dans le cadre de la création d'un drone autopiloté. Même si ce cas d'étude ne constitue pas un système « critique » en lui-même, il en possède les principales caractéristiques : distribution des fonctions, tâches cycliques et sporadiques ayant des contraintes temporelles, caractérisation des télécommandes et télémétries, etc.

Nous nous sommes limités à la partie implémentation de la chaîne d'outils, celle-ci étant la plus pratique et la plus intéressante pour notre lectorat. De nombreuses fonctionnalités n'ont pu être détaillées, faute de place, comme l'intégration de code provenant de modèles applicatifs (SDL – RTDS, Simulink/Matlab, SCADE ou encore VHDL) et l'assurance de la cohérence des données entre ces éléments grâce à une unique définition des données via l'utilisation d'ASN.1. De plus, le recours à des modèles et à une abstraction du système permet de vérifier les exigences du concepteur, ainsi que plusieurs caractéristiques importantes (telles l'ordonnancement) avant tout effort d'implémentation. Les lecteurs intéressés par ces aspects se tourneront vers la documentation de la chaîne d'outils pour obtenir plus d'informations à ce sujet. ■

Remerciements

Nous tenons à remercier tous les partenaires du projet TASTE et tout particulièrement la société Semantix pour les outils ASN.1 développés au cours du projet, Ellidiss et leur accueil brestois chaleureux, mais également l'ISAE, pour le support d'Ocarina et les grands débats sur les avantages de la sémantique forte d'Ada.

Note

Les lecteurs intéressés souhaitant reproduire le système de drone implémenté pourront se référer au site http://www.safety-critical.eu, qui contient le code du projet avec ses prochaines mises à jour.

Références

[ARDUINO] : http://www.arduino.cc

[ARDUINOFR] : http://arduino.cc/fr/

[SEMANTIX] : http://www.semantix.gr/assert

[ELLIDISS] : Site web de la société Ellidiss éditant les outils graphiques de la chaîne TASTE, http://www.ellidiss.com

[TASTE] : Site du projet ASSERT, section dédiée à la chaîne d'outils TASTE, http://www.assert-project.net/taste

[WIKIASN1] : http://fr.wikipedia.org/wiki/ASN1

[AADL] : http://www.aadl.info

[ASN1REFCARD] : http://www.oss.com/asn1/tutorial/A4Card.pdf

[RATEMON] : http://en.wikipedia.org/wiki/Rate-monotonic_scheduling

[NIST02] : *The Economic Impacts of Inadequate Infrastructure for Software Testing* - http://www.nist.gov/director/planning/upload/report02-3.pdf

[RTEMS] : Plateforme d'exécution temps-réel, http://www.rtems.com

[ASSERT] : Site officiel du projet ASSERT, http://www.assert-project.net

[WIKICURSES] : http://en.wikipedia.org/wiki/Curses_%28programming_library%29

[TLDP-SERIAL] : *Serial Programming HowTo*, http://tldp.org/HOWTO/Serial-Programming-HOWTO/

[AADL-TUTORIAL] : « *The Architecture Analysis and Design Language : An Introduction* », http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm

IMPLÉMENTATION EFFICACE D'ALGORITHMES SUR FPGA

par *Sébastien Bourdeauducq*

Les FPGA ouvrent des possibilités de calculs extrêmement performants, à condition de bien savoir les utiliser. Cet article est le premier d'une série visant à montrer, en partant d'un exemple simple, comment différents choix d'architectures impactent les performances et l'utilisation des ressources disponibles sur le FPGA. Ce choix est primordial mais souvent négligé dans beaucoup de « cores » open source, qui sont parfois des monstres de lenteur et d'inefficacité.

1 Introduction

Nous allons prendre pour exemple celui de l'implémentation sur FPGA de la multiplication de nombres complexes, opération couramment utilisée dans de nombreux algorithmes, en particulier dans le domaine du traitement du signal (un domaine où les FPGA excellent). Petit rappel de mathématiques : nous cherchons à calculer le produit A*B = (a + ib)*(c + id) = a*c - b*d + i(b*c + d*a). Autrement dit, notre circuit prendra pour entrée les nombres (réels) a, b, c, d et calculera deux nombres, a*c - b*d et b*c + d*a correspondant aux parties réelle et imaginaire du résultat. Nous travaillerons avec des entiers signés sur 9 bits, ce qui permettra à notre multiplieur complexe d'être utilisable en pratique (les nombres complexes dont les parties réelle et imaginaire sont toutes les deux positives sont un ensemble assez restreint) et de rappeler comment faire de l'arithmétique signée avec les HDL (Verilog et VHDL).

Le FPGA que nous allons prendre pour exemple dans nos expérimentations est un XC4VLX25 (Xilinx Virtex-4) en *speed grade* 10. Vous pourrez essayer avec d'autres FPGA (pas forcément Xilinx), d'ailleurs, ce qui suit ne s'applique pas qu'aux FPGA, mais à l'ensemble des circuits logiques (ASIC, ...). Nous nous focaliserons sur les FPGA car ils sont bien plus facilement accessibles : des kits de développement complets sont disponibles pour quelques centaines d'euros (ce qui est bien moins cher qu'un masque ASIC) et les logiciels de conception (dont nous allons avoir besoin ici) sont souvent téléchargeables gratuitement, ce qui est loin d'être le cas de leurs homologues pour ASIC.

2 Approche naïve

2.1. Arithmétique en complément à deux

2.1.1 Généralités

Les HDL les plus courants (Verilog et VHDL) permettent de travailler directement en arithmétique en complément à deux, qui est la méthode utilisée par tous les microprocesseurs modernes pour permettre la représentation de nombres négatifs. Nous en rappellerons ici des propriétés importantes :

- Le bit de poids fort est le bit de signe (0 pour un nombre positif et 1 pour un nombre négatif).
- Si le bit de signe est à 0, le nombre se lit comme les nombres « ordinaires ». Si le bit de signe est à 1, la valeur absolue du nombre s'obtient en inversant tous les bits et en ajoutant 1.
- L'addition et la soustraction de deux nombres ayant le même nombre de bits s'effectuent comme pour des nombres positifs.
- Pour additionner ou soustraire deux nombres ayant des nombres de bits différents, il faut tout d'abord effectuer une extension de signe du nombre ayant le moins de bits afin de se ramener au cas précédent. Cela se fait en « recopiant » le bit de signe autant de fois que nécessaire (par exemple, 1011 et 1111011 sont deux représentations de -5).
- La multiplication et la division sont légèrement différentes pour pouvoir traiter des nombres négatifs.

2.1.2 Verilog

Le Verilog permet de marquer des objets **wire** et **reg** comme étant signés à l'aide du mot-clé **signed**, par exemple :

```
wire signed [12:0] a;
reg signed [2:0] b;
```

Verilog effectuera ensuite une opération signée si ses **deux** opérandes sont signés (et uniquement dans ce cas), comme dans les exemples ci-dessous.

```
wire signed [12:0] a;
wire signed [8:0] b;
wire [3:0] c;
wire [12:0] d;

(...) = a + b; // op. signée (avec extension de signe)
(...) = b + c; // op. non signée (correct, pas d'extension de signe nécessaire pour c)
(...) = b * c; // op. non signée (INCORRECT)
(...) = d + b; // op. non signée (INCORRECT)
```

Pour une description plus approfondie de l'arithmétique signée en Verilog et de ses pièges à éviter, je vous invite à lire le document http://www.uccs.edu/%7Egtumbush/published_papers/Tumbush%20DVCon%2005.pdf.

2.1.3 VHDL

L'arithmétique (signée ou non) en VHDL est bien plus complexe qu'en Verilog, et elle n'est pas incluse dans le langage de base, mais par l'intermédiaire d'une bibliothèque. Vous l'aurez deviné : il en existe plusieurs, qui provoquent parfois confusion et incompatibilité. La bibliothèque recommandée pour les designs modernes est **numeric_std**, qui a été standardisée par l'IEEE. Les bibliothèques **std_logic_arith** et **std_logic_unsigned**, issues de l'entreprise de logiciels propriétaires Synopsys, sont également très répandues dans l'industrie, mais nous n'y intéresserons pas.

Une fois la bibliothèque chargée à l'aide d'un **use ieee.numeric_std.all;**, les types **signed** et **unsigned** sont définis et s'utilisent comme **std_logic_vector** :

```
signal a: unsigned(4 downto 0);
signal b: signed(6 downto 0);
```

VHDL autorise ensuite les opérations arithmétiques sur ces signaux. N'étant pas un spécialiste de VHDL, je ne me hasarderai pas à vous expliquer en détail ce qui se passe lorsque vous multipliez par exemple un **signed** et un **unsigned**... mais le typage fort, qui est l'un des principes de VHDL, pourra vous éviter des surprises rencontrées en Verilog.

Davantage de détails pertinents sur l'arithmétique en VHDL se trouvent dans le document http://www.synthworks.com/papers/vhdl_math_tricks_mapld_2003.pdf.

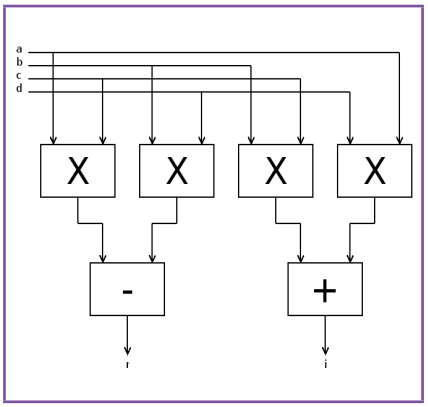
2.1.4 Verilog ou VHDL ?

Certains pourront dire qu'étant donné le temps (extrêmement important) passé à tester et déboguer dans la plupart des projets ASIC ou FPGA et le coût élevé d'un bogue se retrouvant dans un ASIC, VHDL est à recommander car il offre des garde-fous que Verilog n'a pas. D'autres répondront que les erreurs dont VHDL protège sont des erreurs de débutant et que Verilog permet une plus grande productivité chez des développeurs expérimentés grâce à sa concision. À vous de juger... Il est par ailleurs intéressant de constater que VHDL est répandu en Europe alors que c'est Verilog qui fait rage outre-Atlantique. Cette différence semble issue du milieu universitaire, ce qui tend à montrer que l'utilisation de HDL reste majoritairement l'apanage de professionnels.

Dans tous les cas, ces deux langages font la même chose et s'utilisent selon les mêmes principes fondamentaux. Cette série d'articles mettra l'accent sur Verilog tout en donnant des pistes pour que les utilisateurs de VHDL puissent reproduire facilement les expériences décrites dans leur langage préféré.

2.2 Première approche

Nous sommes maintenant armés pour réaliser une première implémentation de l'algorithme de multiplication de nombres complexes. Nous allons faire ceci le plus simplement du monde, en connectant ensemble quatre multiplieurs, un additionneur et un soustracteur.



Implémentation naïve du multiplieur complexe

Le code Verilog correspondant, très simple, est donné ci-dessous. On rappelle au passage que l'opérateur * est synthétisable avec la totalité des outils FPGA modernes et s'implémente automatiquement en utilisant les blocs câblés de multiplication des FPGA en possédant (ce qui est le cas du Virtex-4 pris pour exemple). Les sorties, également sur 9 bits, présentent un risque de débordement (selon les nombres en entrée), mais pour des raisons de simplicité, nous garderons tous les entiers sur 9 bits.

```
module compmult1(
  input signed [8:0] a,
  input signed [8:0] b,
  input signed [8:0] c,
  input signed [8:0] d,

  output signed [8:0] r,
  output signed [8:0] i
);

assign r = a*c - b*d;
assign i = b*c + d*a;

endmodule
```

VHDL n'est pas fondamentalement différent de Verilog. Voici la même chose écrite en VHDL :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compmult1 is
  port(
    a : in signed(8 downto 0);
    b : in signed(8 downto 0);
    c : in signed(8 downto 0);
    d : in signed(8 downto 0);

    r : out signed(8 downto 0);
    i : out signed(8 downto 0)
  );
end entity;

architecture rtl of compmult1 is
begin
  r <= a*c - b*d;
  i <= b*c + d*a;
end architecture;
```

2.3 Compilation

Nous allons maintenant évaluer la performance et l'occupation de ressources du code ci-dessus. Pour cela, nous allons effectuer sa compilation (synthèse, placement et routage) en utilisant les outils de développement Xilinx (ou de votre fabricant de FPGA).

Vous devrez donc, si ce n'est déjà fait, installer la suite de développement FPGA de votre fabricant. Ces dernières sont généralement extrêmement lourdes, boguées, mal écrites et propriétaires, mais elles ont le mérite d'exister, contrairement aux solutions open source. Ce qui suit s'applique à la suite Xilinx (ISE), dont la version *WebPack* est téléchargeable gratuitement.

Plutôt que d'utiliser l'IDE Xilinx (*ISE Project Navigator*) qui est lourd et peu flexible, nous allons utiliser les outils en ligne de commandes à l'aide d'un Makefile. Les étapes traditionnelles de la compilation Xilinx consistent en :

- la synthèse à l'aide de l'outil **xst**, qui consiste à transformer les fichiers Verilog ou VHDL en une série de blocs logiques de bas niveau connectés entre eux (la *netlist*).
- la conversion en un format intermédiaire et l'inclusion d'un éventuel fichier de contraintes UCF (contenant par exemple des affectations de pins FPGA ou des fréquences d'horloges auxquelles le design devra fonctionner) à l'aide de l'outil **ngdbuild**.
- une affectation plus précise des blocs logiques aux ressources disponibles dans le FPGA à l'aide de l'outil **map**.
- le placement-routage des blocs logiques avec l'outil **par**.
- enfin, la conversion en un format lisible par le FPGA à l'aide de **bitgen**.

Tous ces outils ont la fâcheuse manie de créer un fouillis de fichiers temporaires et de logs divers dans le répertoire courant. De plus, les noms de ces fichiers changent de version en version. Afin d'éviter de se retrouver avec un code source noyé dans une centaine de fichiers autogénérés et pour pouvoir supprimer facilement ces derniers (par exemple, pour une redistribution du code source), nous créerons un répertoire **build** dans lequel nous nous placerons avant de lancer chaque outil.

Cela nous donne le **Makefile** reproduit ci-dessous. L'étape **bitgen** n'est pas incluse, car nous ne sommes pas intéressés actuellement par la programmation de composants physiques, mais uniquement par l'évaluation des performances et des ressources utilisées.

```
SRC=compmult1.v # vos fichiers sources ici

all: build/system-routed.ncd

build/system.prj: $(SRC)
  rm -f build/system.prj
  for i in `echo $^`; do \
      echo "verilog work ../$$i" >> build/
      system.prj; \
  done

build/system.ngc: build/system.prj
  cd build && xst -ifn ../run.xst

build/system.ngd: build/system.ngc
  cd build && ngdbuild system.ngc

build/system.ncd: build/system.ngd
  cd build && map -ol high -w system.ngd

build/system-routed.ncd: build/system.ncd
  cd build && par -ol high -w system.ncd
  system-routed.ncd
```

Ce **Makefile** fonctionne avec un script d'options pour **xst** (nommé **run.xst**), qui définit le FPGA utilisé, les noms de fichiers d'entrée et de sortie, et le module Verilog de niveau supérieur :

```
run
-ifn system.prj
-top compmult1
-ifmt MIXED
-ofn system.ngc
-p xc4vlx25-ff668-10
-iob true
```

Si vous utilisez VHDL, remplacez simplement **verilog work** par **vhdl work** dans le **Makefile**.

2.4 Résultat

Lancez **make** et observez le résultat. Parmi les nombreux messages affichés, celui-ci nous intéresse particulièrement :

```
Device Utilization Summary:

  Number of DSP48s           4 out of 48     8%
  Number of External IOBs   55 out of 448   12%
  Number of LOCed IOBs       0 out of 55     0%
```

Ceci signifie que notre design occupe quatre des blocs **DSP48** du FPGA (chacun contenant notamment un multiplieur et un additionneur câblés). Les **IOB** sont simplement les broches d'entrée/sortie du FPGA. C'est cohérent avec la structure de notre code HDL. Mais qu'en est-il de la vitesse ? Pour le savoir, nous utilisons l'outil d'analyse temporelle **trce**, comme ceci :

```
cd build
trce -v 10 system-routed.ncd
less system-routed.twr
```

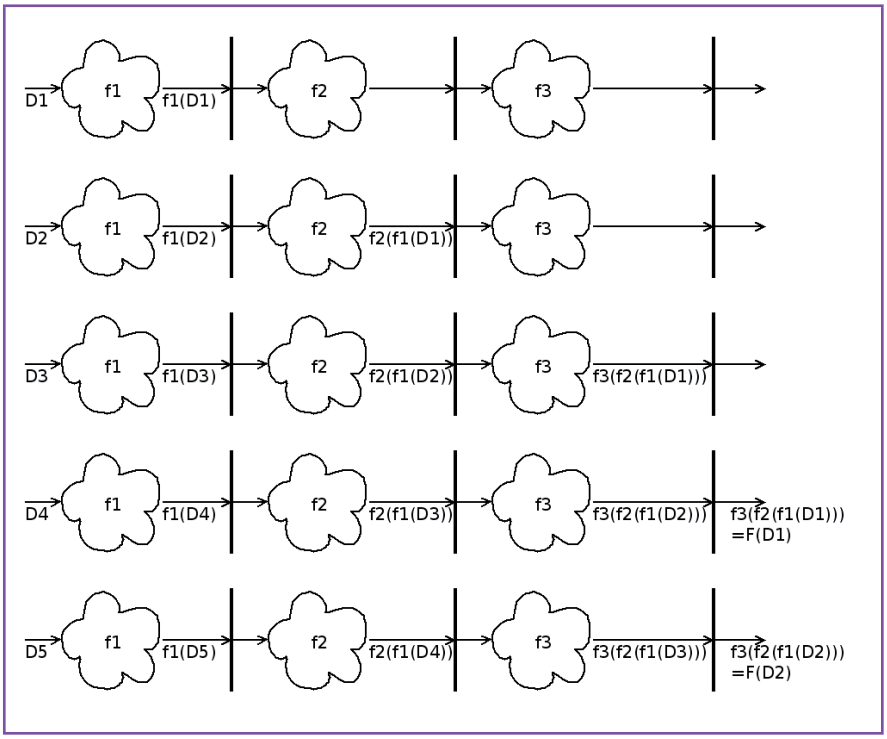
Nous voyons que notre design met 16.764 ns pour stabiliser toutes ses sorties lors de l'application d'une nouvelle entrée. Ce qui signifie que s'il est incorporé dans un système synchrone, sa fréquence d'horloge sera au mieux 1/16.764 ns = 60 MHz (environ)... Autrement dit, notre multiplieur complexe ne pourra pas dépasser les 60 millions d'opérations par seconde. C'est déjà beaucoup, mais cela reste très en dessous de ce qu'un FPGA Virtex-4 peut offrir avec une telle utilisation de ses ressources. Comment améliorer cela ?

3 Pipelining

Une technique courante consiste à découper le circuit traitant les données en circuit plus petits (et donc ayant un temps de propagation plus faible) et en insérant des éléments de stockage (registres ou bascules D) entre chaque morceau (appelés étages). Le fonctionnement du circuit est alors similaire à une chaîne de production. Le premier étage effectue un traitement partiel des données et stocke le résultat. Au front d'horloge suivant, le deuxième étage poursuit le calcul pendant que le premier étage peut traiter une nouvelle donnée, et ainsi de suite. Comme le temps de propagation de chaque étage est inférieur à celui du circuit effectuant le calcul complet (dans le cas idéal, il est divisé par le nombre de registres insérés), on augmente d'autant la bande passante du circuit (c'est-à-dire le nombre de données traitées par seconde).

Cette technique s'appelle *pipelining*. Nous allons donc essayer d'insérer des registres dans notre multiplieur complexe et observer le résultat.

Le code Verilog ci-dessous insère un registre sur chaque entrée des multiplieurs, et deux registres à la sortie de l'additionneur et du soustracteur. Les outils FPGA déplacent ensuite ces registres pour les rapprocher de leur position optimale.



Fonctionnement d'un pipeline. Une fonction F est décomposée en sous-fonctions f1, f2 et f3, qui forment les étages du pipeline. Les barres verticales représentent les registres. Après une période d'amorçage, un nouveau résultat est disponible à chaque cycle d'horloge.

Cela ne fonctionne pas toujours très bien et on obtient parfois de meilleurs résultats en « mettant les mains dans le cambouis » pour placer (physiquement) les registres manuellement, mais ceci sort du cadre de cet article.

```
module compmult1(
  input clk,

  input signed [8:0] a,
  input signed [8:0] b,
  input signed [8:0] c,
  input signed [8:0] d,

  output reg signed [8:0] r,
  output reg signed [8:0] i
);

reg signed [8:0] a0;
reg signed [8:0] b0;
reg signed [8:0] c0;
reg signed [8:0] d0;

reg signed [8:0] r0;
reg signed [8:0] i0;

always @(posedge clk) begin
  a0 <= a;
  b0 <= b;
  c0 <= c;
  d0 <= d;
  r0 <= a0*c0 - b0*d0;
  i0 <= b0*c0 + d0*a0;
  r <= r0;
  i <= i0;
end

endmodule
```

Si vous utilisez VHDL, il n'y a, comme toujours, pas de différence fondamentale avec Verilog : il vous suffira d'utiliser un **process** sensible à l'horloge et d'inférer les registres avec un **if rising_edge(clk) then...** :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compmult1 is
  port(
    clk : in std_logic;

    a : in signed(8 downto 0);
    b : in signed(8 downto 0);
    c : in signed(8 downto 0);
    d : in signed(8 downto 0);

    r : out signed(8 downto 0);
    i : out signed(8 downto 0)
  );
end entity;
```

```
architecture rtl of compmult1 is
signal a0, b0, c0, d0,
  i0, r0 : signed(8 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      a0 <= a;
      b0 <= b;
      c0 <= c;
      d0 <= d;
      r0 <= a0*c0 - b0*d0;
      i0 <= b0*c0 + d0*a0;
      r <= r0;
      i <= i0;
    end if;
  end process;
end architecture;
```

Lançons, comme précédemment, la compilation. Parmi les messages affichés, on se rend compte que le synthétiseur a reconnu les registres de pipelining et les a inclus dans le multiplieur câblé du FPGA :

```
Synthesizing (advanced) Unit <compmult1>.
(...)
Found pipelined multiplier on signal <r0_mult0002>:
 - 1 pipeline level(s) found in a register on signal <a0>.
 Pushing register(s) into the multiplier macro.
 - 1 pipeline level(s) found in a register on signal <c0>.
 Pushing register(s) into the multiplier macro.
 Found pipelined multiplier on signal <i0_mult0003>:
 - 1 pipeline level(s) found in a register on signal <d0>.
 Pushing register(s) into the multiplier macro.
 - 1 pipeline level(s) found in a register on signal <a0>.
 Pushing register(s) into the multiplier macro.
```

En effet, ce bloc multiplieur contient des registres destinés spécialement au pipelining, qui étaient simplement inutilisés (et perdus) dans l'exemple précédent. Cela est confirmé par le rapport d'utilisation de ressources :

```
Device Utilization Summary:

   Number of BUFGs         1 out of 32     3%
   Number of DSP48s        4 out of 48     8%
   Number of External IOBs 55 out of 448   12%
      Number of LOCed IOBs 0 out of 55     0%

   Number of OLOGICs       18 out of 448    4%
```

Quoi de plus par rapport à la version précédente ? Bien peu de choses en réalité. Le **BUFG** est un *buffer* d'horloge global qui, comme son nom l'indique, est partagé dans la totalité d'un design utilisant une même horloge, son « coût » est donc pratiquement nul. Les **OLOGIC** sont des registres dédiés à chaque broche d'entrée/sortie du FPGA. Comme pour les registres des **DSP48**, ils étaient une ressource inexploitée (et inexploitable, car chaque **OLOGIC** ne peut se connecter que sur sa broche qui lui est propre) dans l'exemple précédent.

Dans notre exemple, le pipelining a donc eu un coût quasi-nul sur l'utilisation de ressources. Plus généralement, les architectures internes des FPGA regorgent de registres dédiés qui ne demandent qu'à être exploités, ce qui fait que le pipelining est souvent une technique presque gratuite.

Qu'avons-nous gagné en bande passante ? Le rapport de contraintes d'horloge nous l'indique :

```
------------------------------------------------------------------------
 Constraint                              |    Check    | Worst Case |  Best Case |
                                         |             |    Slack   | Achievable |
------------------------------------------------------------------------
 Autotimespec constraint for clock net clk | SETUP       |         N/A|     5.983ns|
 _BUFGP                                  | HOLD        |     2.349ns|            |
------------------------------------------------------------------------
```

Notre design peut maintenant effectuer 1/5.983 ns = 167 millions d'opérations par seconde, ce qui est près de 2,8 fois la bande passante précédente, tout en gardant le même coût en ressources !

Le pipelining n'aurait-il donc que des avantages ? Hélas, non. Car si c'est une méthode très efficace pour augmenter la bande passante (et la fréquence d'horloge, ce qui a permis au Pentium 4 d'atteindre des sommets), on ne gagne rien sur la latence, voire on y perd légèrement (c'est l'une des raisons pour lesquelles la logique de contrôle des processeurs modernes est si complexe).

Plus précisément, la latence est le temps que met un élément de données pour traverser l'ensemble du circuit de calcul. Quelle est la latence de notre nouveau design ? C'est simple : les trois cycles d'horloges nécessaires pour charger tous les registres, soit 3*5.983 ns = 17.949 ns. Nous avons effectivement perdu par rapport aux 16.764 ns précédentes...

Peut-on améliorer à la fois la bande passante et la latence ? À moins de trouver des sous-calculs indépendants à effectuer en parallèle (ce qui n'est plus possible ici), pas tellement. Les améliorations touchant à la fois la bande passante et la latence sont surtout apportées par le progrès des techniques de fabrication des semi-conducteurs.

En revanche, ce que nous pouvons faire - et ce sera le sujet du prochain article - c'est diminuer le coût en ressources (les **DSP48** peuvent être assez prisés) en partageant le même multiplieur entre différents calculs. Cela se fait, bien sûr, au détriment des performances, mais il peut s'agir d'un choix judicieux afin d'éviter de gaspiller des **DSP48** si l'application ne nécessite pas une bande passante trop élevée pour la multiplication de nombres complexes.

Je vous donne donc rendez-vous dans un prochain numéro, d'ici-là, si vous avez des questions ou des retours sur cet article, n'hésitez pas à me contacter (sebastien@milkymist.org). ■

CONCEPTION ET APPLICATIONS DES LFSR EN VHDL

par Yann Guidon

Les LFSR (registres à décalage à rétroaction linéaire) sont une famille de générateurs de bits pseudo-aléatoires. Ils sont techniquement les plus simples de tous, ce qui explique qu'ils sont couramment employés en informatique et en électronique. Nous allons étudier comment les réaliser en VHDL, en écrivant des unités génériques, portables, simulables et synthétisables, réutilisables dans de nombreuses situations. Plusieurs cas pratiques sont ensuite présentés.

1 Rappels

Les LFSR ont déjà été étudiés dans GNU/Linux Magazine France N° 81 en 2006 **[1]**. Les avantages majeurs de ces structures sont la simplicité de réalisation, ainsi que les propriétés très bien comprises et très utiles des nombres générés. Nous allons nous concentrer sur les aspects pratiques après ces quelques rappels théoriques :

- Les LFSR génèrent des valeurs pseudo-aléatoires, c'est-à-dire qu'elles ont l'air embrouillées pour un œil non averti, mais la séquence est parfaitement déterministe, elle n'est pas aléatoire. C'est même pour cela qu'on les utilise en télécommunications (GPS, GSM, Ethernet, ...).

- Un LFSR peut être décrit ou identifié grâce à son polynôme seul, que nous représentons ici sous forme d'une liste de nombres. En mathématiques classiques, un polynôme de huitième degré (correspondant à un LFSR à 8 bits) est écrit sous la forme $x^8+x^6+x^5+x^4+1$. Pour nous simplifier la vie, nous ne conservons que les exposants, ce qui donne la représentation (8, 6, 5, 4, 1). Cela correspond en électronique aux rangs des bits à prendre en compte pour le calcul du nouvel état du registre à décalage.

- Un LFSR classique ne doit pas être initialisé avec tous les bits à 0, car cet état neutre ne lui permettra pas de passer à un autre état.

- Les seuls polynômes qui nous intéressent sont ceux qui génèrent des séquences de longueur maximale. La séquence de bits est obtenue en lisant l'état d'un des registres du LFSR. Et s'il dispose de N bits, il peut prendre 2^N états différents (l'état avec tous les bits à zéro est interdit, donc cela fait en fait 2^N-1 états). Les polynômes dont il est question ici ont la propriété de faire passer le LFSR par tous les états possibles avant de reboucler, ce qui génère une séquence de 2^N-1 bits.

- Lorsqu'on dispose d'un polynôme valide (générant une séquence de longueur maximale), son polynôme réfléchi dispose des mêmes propriétés et génère la même séquence, mais dans l'ordre inverse. Pour l'obtenir, il faut transformer chaque indice i du polynôme avec i'=N-(i+1). Par exemple, la réflexion du polynôme (8, 6, 5, 4, 1) est (8, 5, 4, 3, 1).

- Il existe deux façons de construire un LFSR : la configuration de Galois et la configuration de Fibonacci. Comme avec les polynômes réfléchis, la séquence en configuration de Galois est inversée par rapport à la configuration de Fibonacci. En pratique, on choisit la configuration qui est techniquement la plus avantageuse.

Attention

Il faut bien garder à l'esprit que les séquences générées par un LFSR sont très prévisibles, elles sont apparemment embrouillées, mais ont des propriétés bien précises et connues. Les LFSR sont donc à éviter absolument pour les applications cryptographiques ou de sécurité.

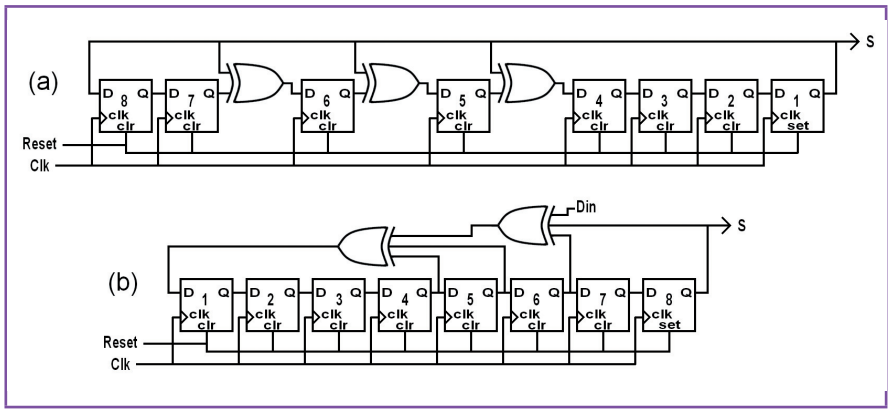


Figure 1 : Un LFSR à 8 bits en configuration de Galois (a) et Fibonacci (b). L'ordre des bits dépend de la convention choisie.

2 À table !

Admettons maintenant que vous ayez besoin d'un générateur de bits pseudo-aléatoire et déterministe, qui n'intervient pas dans un système de sécurité : le LFSR est très probablement fait pour vous. La démarche normale consiste maintenant à déterminer la configuration (Galois ou Fibonacci ?), la valeur initiale du registre, sa taille et le polynôme. C'est ce que nous avons fait dans *GLMF Hors-Série* n° 51 **[2]** pour contrôler les segments de notre afficheur en VHDL.

Le seul point délicat, c'est le polynôme : il dépend directement de la taille du registre, il ne s'invente pas et il n'existe pas de formule simple pour le créer. Il faut alors consulter la littérature pour trouver des tables précalculées, contenant un polynôme adapté à notre cas. Bruce Schneier, dans son *best seller* « Cryptographie appliquée » **[3]**, fournit une telle table. On en trouve aussi dans les liens de la page Wikipédia dédiée aux LFSR **[4]** ou chez Xilinx **[6]**.

Ces tables sont très pratiques lorsqu'on a une application occasionnelle pour laquelle on écrit le LFSR au cas par cas. Elles ont cependant un inconvénient qui n'apparaît pas tout de suite : les polynômes listés n'ont pas un nombre fixe de facteurs. Lorsqu'on veut changer ou ajuster la taille d'un registre à décalage, la complexité logique varie alors de manière imprévisible. Nous verrons plus loin un cas où la linéarité de la complexité est déterminante.

C'est là qu'intervient la table de Roy Ward et Tim Molteno **[5]**. Ils ont réuni en 2007 des polynômes à 2 et 4 facteurs pour des registres de 2 à 786 bits. Les polynômes à deux facteurs n'existent malheureusement pas pour toutes les tailles. Il existe cependant une liste continue avec 4 facteurs, ce qui nous assure la linéarité désirée.

2.1 Version à quatre facteurs

Commençons par exploiter la table. Dans sa forme originale, chaque ligne présente un indice (le nombre de bits du registre à décalage), suivi éventuellement d'un polynôme à 2, puis à 4 facteurs :

```
41    (41, 38)    (41, 40, 39, 38)
```

Si on compare avec l'exemple de polynôme en début d'article, on remarque que le facteur 1 est absent. C'est un artefact mathématique qui signifie : « on reboucle les bits ici » et cela n'a pas d'influence sur le calcul. La valeur 1 est toujours implicite dans la table et n'apparaît pas.

Pour l'instant, nous ignorons les polynômes à deux facteurs, il reste donc l'indice et le polynôme à 4 facteurs. Or nous remarquons aussi que l'indice est le même nombre que le facteur le plus élevé. Pour économiser de la place dans le code source en VHDL, nous pouvons enlever ce facteur car il est redondant. Ainsi, nos polynômes à 4 facteurs sont codés avec seulement 3 nombres, le quatrième étant implicitement donné par la place dans le tableau.

En VHDL, nous créons donc un type de tableau, appelé ici `poly_array_type`, avec deux dimensions. Une dimension sert à stocker un polynôme, l'autre contient les 3 facteurs non implicites du polynôme. La table ainsi créée reste assez conséquente...

Ensuite, codons le registre à décalage. Pour simplifier l'écriture, on préserve la numérotation des bits, commençant par 1 (au lieu de 0 habituellement en informatique). La configuration de Fibonacci est choisie pour deux raisons :

- Elle est la plus facile à coder, tout simplement.
- Électroniquement, elle est aussi un peu plus rapide, car les temps de propagation sont faibles et constants. Le *fanout* (ou la *sortance* en franglais) des registres n'est que de 2 portes, alors que la configuration de Galois oblige une des cellules logiques à envoyer sa valeur à au moins quatre portes logiques, ce qui est plus lent. On peut donc espérer une fréquence de fonctionnement un tout petit peu plus élevée.

Le document original **[5]** donne un exemple avec un LFSR à 8 bits en configuration de Galois, mais nous savons (au pire, depuis avoir lu le rappel) que les polynômes sont aussi valables en configuration de Fibonacci, que nous allons maintenant utiliser sans complexe.

Selon la technologie électronique employée, les performances vont varier. Par exemple, une structure de FPGA classique telle qu'employée par Altera ou Xilinx (à base de cellules contenant une LUT et un registre) peut réaliser des LFSR en configuration de Galois aussi efficaces. C'est juste plus compliqué à écrire de manière paramétrable. Il est recommandé de consulter les notes d'application du constructeur **[6]**.

```
-- file : lfsr4.vhdl : parameterised LFSR generator
-- table from http://www.physics.otago.ac.nz/px/research/
--    electronics/papers/technical-reports/lfsr_table.pdf
-- by Roy Ward, Tim Molteno, "Table of Linear Feedback Shift Registers"
library ieee;  use ieee.std_logic_1164.all;
entity lfsr4 is
  generic(
    size : integer := 8);  -- taille par défaut
  port(
    -- broches de contrôle
    clk, reset, din : in std_logic := '0';
    -- les bits du registre sont tous accessibles :
    lfsr : inout std_ulogic_vector(size downto 1);
    -- valeur d'initialisation (laisser "open" ou connecter à une constante) :
    init_val : in std_ulogic_vector(size downto 1) := (1=>'1', others=>'0');
    -- sortie séparée :
    s : out std_logic );
end lfsr4;
architecture fibonacci of lfsr4 is
  -- taille minimale et maximale
  constant first_poly : integer := 5;
  constant  last_poly : integer := 786;
  -- type d'une table de polynômes :
  type poly_array_type is array(first_poly to last_poly, 0 to 2) of integer;
  -- la table des polynômes :
  constant poly4_array : poly_array_type := (
    (4, 3, 2),       (5, 3, 2),       (6, 5, 4),       (6, 5, 4), -- 5 à 8
    (8, 6, 5),       (9, 7, 6),      (10, 9, 7),      (11, 8, 6), -- 9
   (12, 10, 9),     (13, 11, 9),     (14, 13, 11),    (14, 13, 11), -- 13
  (16, 15, 14),    (17, 16, 13),    (18, 17, 14),    (19, 16, 14), -- 17
  (20, 19, 16),    (19, 18, 17),    (22, 20, 18),    (23, 21, 20), -- 21
  (24, 23, 22),    (25, 24, 20),    (26, 25, 22),    (27, 24, 22), -- 25
  (28, 27, 25),    (29, 26, 24),    (30, 29, 28),    (30, 26, 25), -- 29
  (32, 29, 27),    (31, 30, 26),    (34, 28, 27),    (35, 29, 28), -- 33
  (36, 33, 31),    (37, 33, 32),    (38, 35, 32),    (37, 36, 35), -- 37
  (40, 39, 38),    (40, 37, 35),    (42, 38, 37),    (42, 39, 38), -- 41
  (44, 42, 41),    (40, 39, 38),    (46, 43, 42),    (44, 41, 39), -- 45
  (45, 44, 43),    (48, 47, 46),    (50, 48, 45),    (51, 49, 46), -- 49
  (52, 51, 47),    (51, 48, 46),    (54, 53, 49),    (54, 52, 49), -- 53
  (55, 54, 52),    (57, 53, 52),    (57, 55, 52),    (58, 56, 55), -- 57
  (60, 59, 56),    (59, 57, 56),    (62, 59, 58),    (63, 61, 60), -- 61
  (64, 62, 61),    (60, 58, 57),    (66, 65, 62),    (67, 63, 61), -- 65
  (67, 64, 63),    (69, 67, 65),    (70, 68, 66),    (69, 63, 62), -- 69
  (71, 70, 69),    (71, 70, 67),    (74, 72, 69),    (74, 72, 71), -- 73
  (75, 72, 71),    (77, 76, 71),    (77, 76, 75),    (78, 76, 71), -- 77
  (79, 78, 75),    (78, 76, 73),    (81, 79, 76),    (83, 77, 75), -- 81
  (84, 83, 77),    (84, 81, 80),    (86, 82, 80),    (80, 79, 77), -- 85
  (86, 84, 83),    (88, 87, 85),    (90, 86, 83),    (90, 87, 86), -- 89
  (91, 90, 87),    (93, 89, 88),    (94, 90, 88),    (90, 87, 86), -- 93
  (95, 93, 91),    (97, 91, 90),    (95, 94, 92),    (98, 93, 92), -- 97
 (100, 95, 94),    (99, 97, 96),   (102, 99, 94),   (103, 94, 93), -- 101
 (104, 99, 98), (105, 101, 100),   (105, 99, 98),   (103, 97, 96), -- 105
-- Table abrégée pour les besoins de mise en page
 (776, 767, 761), (775, 762, 759), (776, 771, 769), (775, 772, 764), -- 777
 (779, 765, 764), (780, 779, 773), (782, 776, 773), (778, 775, 771), -- 781
 (780, 776, 775), (782, 780, 771));                                  -- 785
begin
  process (clk, reset)
  begin
    -- initialisation
    if reset = '1' then
      lfsr <= init_val;
    else
      if rising_edge(clk) then
        -- calcul du nouveau bit
        lfsr(1) <= lfsr(size)
              xor lfsr(poly4_array(size,0))
              xor lfsr(poly4_array(size,1))
              xor lfsr(poly4_array(size,2))
              xor din;
        -- décalage :
        lfsr(size downto 2) <= lfsr(size-1 downto 1);
      end if;
    end if;
  end process;
  s <= lfsr(1);
end fibonacci;
```

Ce LFSR est équipé de série avec tous les signaux de configuration et de contrôle possibles. On dispose même de l'intégralité des signaux internes, lisibles si nécessaire, puisque le registre est un port de type **inout**. Le code du process a une structure tout à fait classique pour décrire une bascule D avec réinitialisation. Le décalage du registre est décrit en une seule ligne. Le calcul du nouveau bit prend plus de place, mais comme promis, il ne s'agit que de XOR.

Le port **init_val** est un peu spécial, car il remplit une fonction normalement dédiée à l'interface générique. La valeur d'initialisation est normalement déterminée avant le début de la simulation, pour qu'elle soit considérée comme constante et devienne une entrée *set* ou *reset* après la synthèse. Cependant, puisque la taille du vecteur d'initialisation dépend d'une autre valeur générique, on ne peut pas la mettre à côté. Le seul endroit où cela reste possible et pratique est dans la liste des ports. Cependant, il faut soit laisser ce port ouvert (non connecté), soit le connecter à une constante, sinon le circuit synthétisé risque de ne pas être celui attendu...

J'ai aussi choisi d'ajouter une entrée auxiliaire **Din**. Elle peut rester inutilisée, on la fixe alors à '0' lors de l'instanciation. Elle sert dans les cas où on veut perturber l'état du registre, par exemple dans une application similaire à un CRC, ou pour satisfaire des contraintes de synthèse logique (ce que nous allons voir plus loin).

2.2 Version à deux facteurs

La table créée par Roy Ward et Tim Molteno contient aussi une colonne présentant, lorsqu'il existe, un polynôme à seulement deux facteurs. Cela peut être utile quand on désire minimiser autant que possible la complexité logique pour grapiller quelques femtosecondes et accélérer un circuit.

Cependant, de tels polynômes sont assez épars : seulement 333 polynômes

parmi 785 valeurs. Ils ne permettent donc pas la souplesse permise par les polynômes à 4 facteurs. La table est néanmoins présentée ici avec son code, car elle peut aussi être utile.

En ce qui concerne le code lui-même, il est directement dérivé du précédent. Le premier changement concerne la forme de la table des polynômes : les facteurs sont la taille du registre et un autre nombre. Il n'y a donc plus qu'un seul nombre à stocker, ce qui permet d'utiliser une table simple, à une seule dimension.

L'autre différence est que le polynôme n'existe pas pour toutes les tailles. Il faut donc pouvoir indiquer quels polynômes sont valides, et n'indiquer qu'eux. La table précédente utilisait une énumération par position (l'indice de l'élément étant indiqué par sa position dans la liste), la table suivante utilise une notation explicite où l'indice est indiqué clairement. Tous les autres indices non listés sont initialisés à 0 par la clause **others**.

Enfin, puisque certaines tailles ne sont pas disponibles, il faut les détecter. En ce qui concerne la simulation, c'est simple comme un **assert**. Pour la synthèse... on oublie. Les assertions un peu complexes sont systématiquement ignorées par la plupart des outils, qui détectent cependant que l'indice 0 tombe hors du vecteur (qui commence à 1). On ne peut pas fournir de message expliquant l'erreur, tant pis. La morale est de toujours simuler avant de synthétiser !

```
-- file : lfsr2.vhdl
-- parameterised LFSR generator with 2 factors
library ieee;  use ieee.std_logic_1164.all;

entity lfsr2 is
  generic(
    size : integer := 7);
  port(
    clk, reset, din : in std_logic := '0';
    lfsr : inout std_ulogic_vector(size downto 1);
    init_val : in std_ulogic_vector(size downto 1) := (1=>'1', others=>'0');
    s : out std_logic );
end lfsr2;

architecture fibonacci of lfsr2 is
  type poly_array_type is array(2 to 785) of integer;
  constant poly2_array : poly_array_type := (
  2=>1,     3=>2,     4=>3,     5=>3,     6=>5,     7=>6,     9=>5,    10=>7,
 11=>9,    15=>14,   17=>14,   18=>11,   20=>17,   21=>19,   22=>21,   23=>18,
 25=>22,   28=>25,   29=>27,   31=>28,   33=>20,   35=>33,   36=>25,   39=>35,
 41=>38,   47=>42,   49=>40,   52=>49,   55=>31,   57=>50,   58=>39,   60=>59,
 63=>62,   65=>47,   68=>59,   71=>65,   73=>48,   79=>70,   81=>77,   84=>71,
 87=>74,   89=>51,   93=>91,   94=>73,   95=>84,   97=>91,   98=>87,  100=>63,
103=>94,  105=>89,  106=>91,  108=>77,  111=>101, 113=>104, 118=>85,  119=>111,
-- Table abrégée pour les besoins de mise en page
759=>661, 761=>758, 762=>679, 767=>599, 769=>649, 772=>765, 774=>589, 775=>408,
777=>748, 778=>403, 782=>453, 783=>715, 785=>693, others=>0);

begin
  process (clk, reset)
  begin
    assert poly2_array(size) > 0
      report "Mauvaise taille de LFSR2" severity failure;
    if reset = '1' then
      lfsr <= init_val;
    else
      if rising_edge(clk) then
        lfsr(1) <= lfsr(size)
               xor lfsr(poly2_array(size))
               xor din;
        lfsr(size downto 2) <= lfsr(size-1 downto 1);
      end if;
    end if;
  end process;
  s <= lfsr(1);
end fibonacci;
```

3 Exemple d'utilisation en simulation

Commençons par vérifier le bon fonctionnement du LFSR en créant un petit banc de test. Nous allons simuler le registre et afficher sa sortie pour vérifier, par exemple, la périodicité et la répartition des bits. On sait qu'il doit y avoir quasiment autant de 1 que de 0 en sortie, nous allons donc (faire) compter les bits.

D'abord, il faut instancier notre LFSR4, ce qui se fait en VHDL'93 avec le mot-clé **entity**. On recopie la déclaration des ports et les génériques du code source du LFSR4, et on connecte les entrées/sorties à des signaux spécialement prévus par notre banc de test : ici, nous avons pour cela créé les signaux **result**, **clk** et **reset**.

Ensuite, pour faire tourner la simulation, on contrôle ces signaux, qui indirectement déclenchent le fonctionnement du LFSR. Ce dernier change sa sortie, connectée à **result** qu'on veut observer.

Pour « tirer les ficelles » d'un circuit comme le LFSR4, on crée un process dans lequel on peut écrire des séquences d'opérations. Chaque pas de temps de la simulation se passe souvent en trois étapes : d'abord l'assignation d'une valeur à un signal de contrôle (**reset <= '1';**), suivie d'une attente (par exemple **wait for 1 ns;**) qui met le process en sommeil et déclenche le calcul du nouvel état du LFSR. Enfin, on lit l'état de sortie, ici connecté au signal **result**.

Ce dernier est de type **std_ulogic** et si on veut afficher sa valeur au moyen de l'expression **std_ulogic'image(result)**, le simulateur retourne une chaîne de caractères avec le caractère désiré entouré d'apostrophes (**'1'**). Ce comportement est normal en VHDL, mais n'est pas désiré ici. Dans le programme suivant, la chaîne est donc stockée temporairement et on sélectionne ensuite le caractère du milieu pour l'ajouter (avec **write()**) à la ligne à afficher (la variable **l**).

```
-- fichier : test_lfsr4.vhdl
library ieee;   use ieee.std_logic_1164.all;
library std;    use std.textio.all;
entity test_lfsr4 is end test_lfsr4;
architecture test of test_lfsr4 is
  -- taille du LFSR4 : (2^c)-1 => 255 bits générés par cycle
  constant c : integer := 8;
  signal result, clk, reset : std_ulogic;
begin
 -- instanciation du LFSR4
 dut : entity work.lfsr4
    generic map(size => c)
    port map ( clk => clk,  reset => reset,
               din => '0',  s => result);
  process
    variable i, j, n0, n1 : integer; -- compteurs
    variable l: line;  -- contient une ligne à afficher
    -- variable temporaire pour l'affichage
    variable s: string(3 downto 1);
  begin
    -- séquence d'initialisation
    clk <= '0';
    reset <= '1';    wait for 1 ns;
    reset <:= '0';    wait for 1 ns;
    for j in 1 to 3 loop
      n0 := 0;
      n1 := 0;
      for i in 2 to 2**c loop
        -- simule un cycle d'horloge
        clk <= '1'; wait for 1 ns;
        clk <= '0'; wait for 1 ns;
        -- compte les bits
        if result = '1' then
          n1 := n1 + 1;
        else
          n0 := n0 + 1;
        end if;
        -- récupère la sortie
        s:=std_ulogic'image(result);
        write(l,s(2));
      end loop;
      -- affiche toute la ligne
      writeline(output,l);
      -- affiche le nombre de 0 et de 1 :
      write(l,"0:"&integer'image(n0)&", 1:"&integer'image(n1));
      writeline(output,l);
    end loop;
    wait;
  end process;
end test;
```

La boucle qui effectue un cycle complet du LFSR est répétée trois fois. En superposant les lignes successives, on peut voir qu'elles sont identiques et indirectement conclure que la période (ici, 255 bits avant rebouclage) est correcte.

```
yg$ ghdl -a lfsr4.vhd test_lfsr4.vhdl
test_lfsr4.vhdl:40:16:warning: universal integer bound must be numeric literal
 or attribute
yg$ ghdl -e test_lfsr4
yg$ ./test_lfsr4
00011100010010111000000110010010011011100100000101011011010110010110000111110
10111101011101000100001101100011110011100110001011010010001010010101001110111
11001111011111101001100110101000110000011101010101111100101000010011111111000
10111100011010000001
0:127, 1:128
00011100010010111000000110010010011011100100000101011011010110010110000111110
10111101011101000100001101100011110011100110001011010010001010010101001110111
11001111011111101001100110101000110000011101010101111100101000010011111111000
10111100011010000001
0:127, 1:128
00011100010010111000000110010010011011100100000101011011010110010110000111110
10111101011101000100001101100011110011100110001011010010001010010101001110111
11001111011111101001100110101000110000011101010101111100101000010011111111000
10111100011010000001
0:127, 1:128
```

De plus, on compte le nombre de bits à 0 et à 1. On trouve bien un 1 de plus qu'il n'y a de 0, ce qui correspond au comportement théorique et confirme que le LFSR fonctionne bien.

4 Exemple d'utilisation en synthèse

Pour des raisons personnelles, j'utilise surtout les FPGA de la famille Actel ProASIC3. Ils ne sont pas parfaits, mais leur granularité très fine les rapproche des contraintes qu'on trouve dans la conception des ASIC (d'où leur nom).

Avec cette famille, une porte logique simple ou un registre seront réalisés au moyen d'une tuile (ou *tile* dans la terminologie anglaise d'Actel), une cellule dont les connexions et la fonction sont reconfigurables. Contrairement à la plupart des autres familles de FPGA, cela donne une bonne idée du comportement et des limites d'un système électronique si on le réalise plus tard en technologie *full-custom* ; avec les portes logiques réalisées en dur, non reconfigurables, on réduit la taille et on augmente la vitesse d'un facteur 10 environ, mais les proportions restent approximativement les mêmes.

4.1 Un LFSR

Les tuiles des ProASIC3 peuvent être configurées comme une porte logique à 3 entrées de fonction arbitraire. On peut donc coder au maximum un XOR à 3 entrées. Le LFSR4 combine 5 entrées avec le XOR, il faut donc cascader 2 portes XOR3 pour réaliser la fonction *A xor B xor (C xor D xor Din)*. Cette complexité logique est indépendante de la taille, nous aurons donc toujours 2 tuiles à utiliser avec le LFSR4.

Les registres ont, eux, besoin typiquement de 4 entrées (donnée, horloge, réinitialisation, validation), une entrée auxiliaire et dédiée a donc été ajoutée à la tuile pour éviter d'utiliser deux tuiles dans la plupart des cas (lorsqu'une validation et/ou une réinitialisation est nécessaire). Chaque bit du registre à décalage utilise donc une seule tuile.

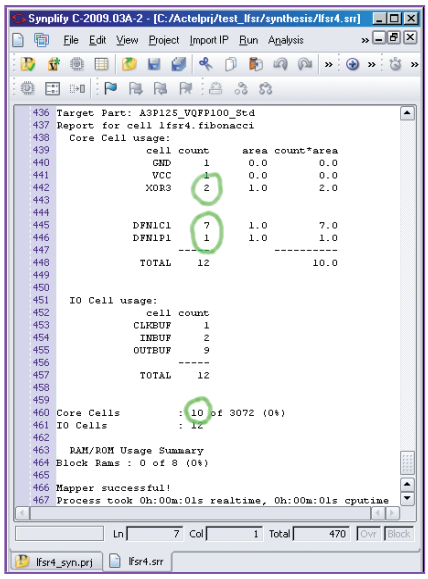


Figure 2 : Rapport de la synthèse du LFSR4 à 8 bits (eh oui, j'ai honte d'utiliser Vista)

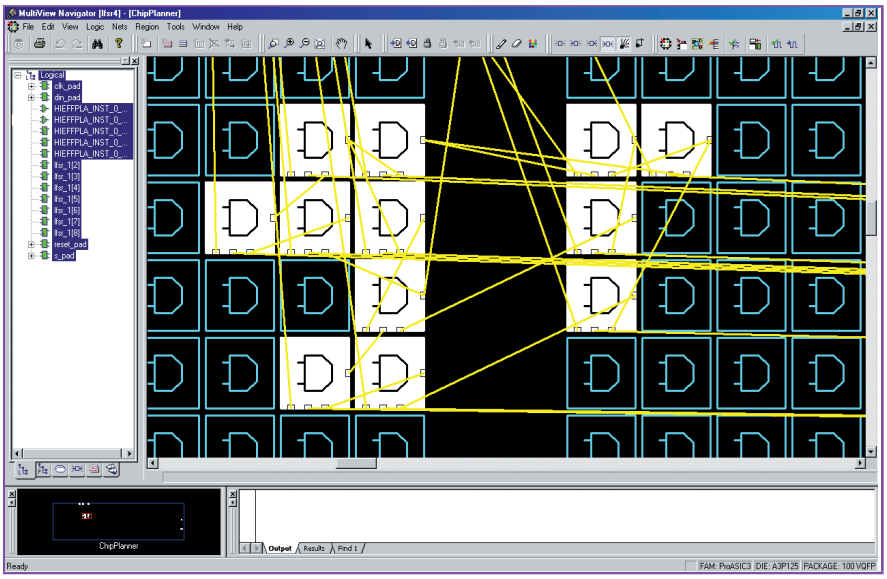


Figure 3 : Placement et routage du LFSR4 dans la matrice du FPGA

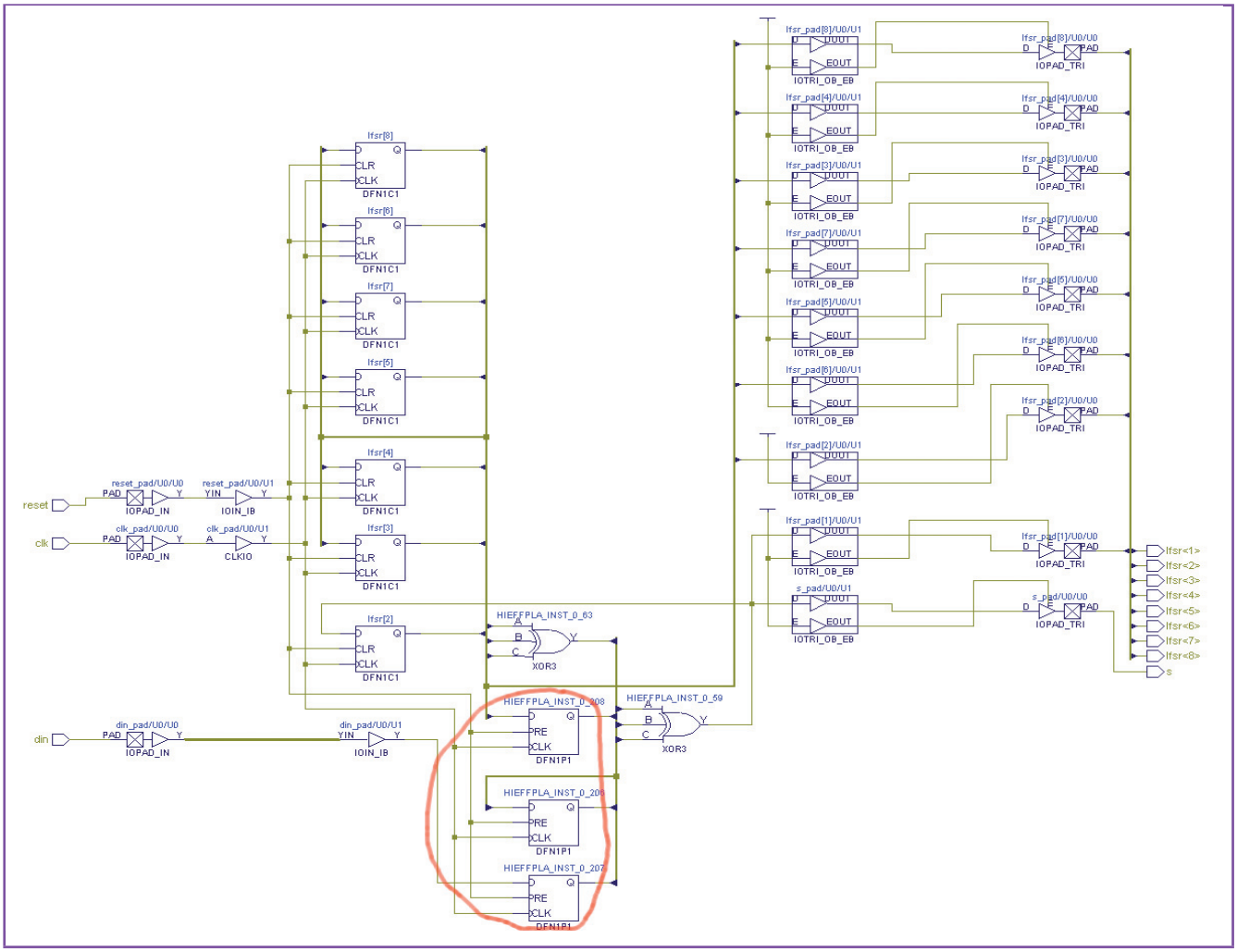


Figure 4 : Schéma électronique correspondant au circuit après optimisations. Les cellules de la moitié droite correspondent aux broches de sorties.

Si on synthétise notre LFSR4 directement, avec la configuration par défaut à 8 bits, nous devrions obtenir une utilisation de 2+8=10 tuiles. La figure 2 montre que le synthétiseur indique bien qu'il utilise 2 portes XOR3, 7 portes DFN1C1 (registre avec initialisation à 0) et 1 porte DFN1P1 (initialisation à 1). Le compte est bon, passons au placement-routage !

Regardons sur la figure 3 les tuiles après l'opération de placement-routage : on en trouve 12 au lieu de 10.

Le logiciel permet d'extraire le schéma électronique après toutes les optimisations. La figure 4 est une copie d'écran qui indique que la cellule DFN1P1 a été répliquée, elle est maintenant en triple exemplaire. C'est très probablement l'effet de toutes les options d'optimisation que j'ai sélectionnées...

En tout cas, sans effort de codage, le logiciel indique que la fréquence de fonctionnement atteint 250 MHz, ce qui est un très bon résultat pour cette famille de composants (proche du maximum), surtout sans soigner ni optimiser la conception.

4.2 Autant de LFSR que possible

Maintenant que nous connaissons mieux le comportement de LFSR4 dans ce FPGA, passons à un cas plus intéressant. J'ai reçu quelques échantillons de A3P125-VQG100 que je voulais tester sommairement. En effet, les vecteurs de test d'Actel sont confidentiels et je ne dispose pas de leur matériel. L'idée est donc de créer un circuit électronique qui utilise au maximum les ressources internes et externes pour vérifier que la plupart des fonctions sont en état de fonctionnement.

En particulier, je voulais tester l'intégrité de la matrice de tuiles ainsi que les broches d'entrées/sorties. L'A3P125 dispose de 3072 tuiles et la version à 100 broches fournit 67 signaux d'entrées/sorties. Si on utilise les broches de contrôle d'un LFSR (horloge, réinitialisation et donnée auxiliaire), on dispose encore de

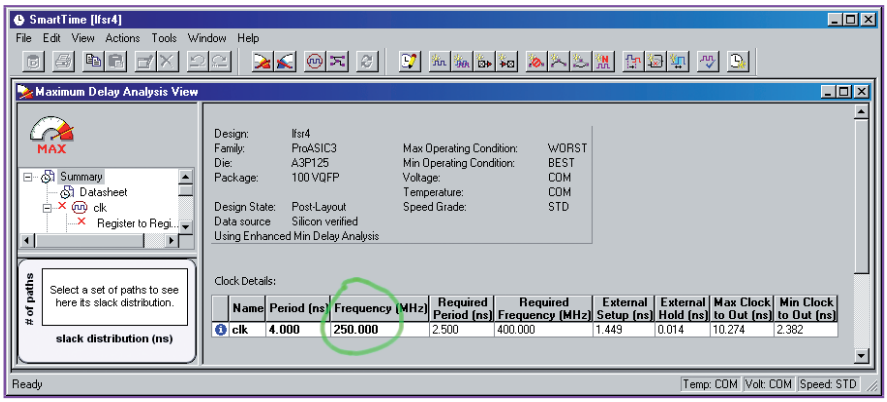


Figure 5 : Le rapport de placement-routage indique la fréquence maximale de fonctionnement.

64 broches. En partageant équitablement toutes les tuiles parmi les 64 broches, on tombe sur le compte exact de 48 tuiles par broche. Comment allons-nous faire pour obtenir exactement cette occupation ?

La réponse est évidente : il suffit de dimensionner précisément un LFSR. En plus, puisque ce circuit peut fonctionner à la limite de la vitesse normale du composant, cela permet de vérifier que l'échantillon peut effectivement atteindre la fréquence indiquée par la documentation. C'est donc un test simple, rapide et efficace, bien que non exhaustif. Une boucle d'instanciation **`for ... generate`** suffit à répliquer notre LFSR autant de fois que désiré.

En pratique, ce n'est pas si simple, car le synthétiseur va toujours tenter de faire son malin en optimisant le circuit. Et dans notre cas, nous créons 64 circuits identiques, alors que va-t-il se passer ? Le synthétiseur va remarquer que les résultats des 64 LFSR sont identiques, il va créer seulement un LFSR et envoyer le résultat à toutes les broches de sortie et juste 1/64ème de la matrice sera occupée.

C'est là que l'entrée auxiliaire Din montre son utilité : elle va nous permettre de chaîner les registres entre eux, la sortie S de l'un allant au Din du registre suivant, ce qui brise le parallélisme et force le synthétiseur à considérer chaque LFSR individuellement. À part cela, rien de remarquable.

```
-- test_a3p125.vhdl

library ieee;
use ieee.std_logic_1164.all;

entity test_a3p125 is
generic ( size : integer := 64); -- nombre de broches à tester
port ( clk, reset, sig_in : in std_ulogic;
    sig_out : inout std_ulogic_vector(size downto 1));
end test_a3p125;
architecture test of test_a3p125 is
  signal sigtmp : std_ulogic_vector(size downto 1);
begin
 lab: for iterateur in 1 to size generate
     dut : entity work.lfsr4
    generic map(size => 46) -- 46+6 = 48, 48*64 = 3072 tiles
    port map (
      clk => clk,
      reset => reset,
      din => sigtmp(iterateur),
      s => sig_out(iterateur),
      lfsr => open);
  end generate lab;

  sigtmp <= sig_out(size-1 downto 1) & sig_in;
end test;
```

Cette fois-ci, j'ai été moins agressif sur les options d'optimisation et aucune porte n'a été répliquée. Avec 46 bits par LFSR4, nous occupons comme prévu 48 tuiles. Le LFSR2 dispose d'un polynôme à 47 bits, ce qui donne aussi 48 tuiles avec le XOR3 nécessaire. Cependant, si on doit réutiliser cette méthode pour tester d'autres composants avec le LFSR2, il y aura probablement moins de chance de tomber sur une valeur exacte.

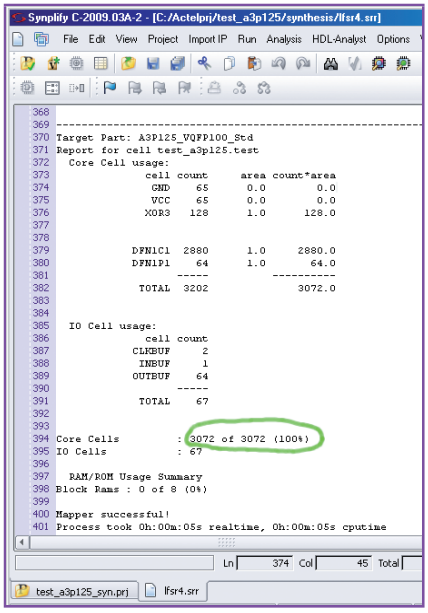


Figure 6 : Le rapport de synthèse confirme que la matrice est bien utilisée à 100 %.

En occupant autant de ressources, on atteint les limites intrinsèques du composant. Par exemple, l'arbre d'horloge interne se retrouve extrêmement chargé, il doit envoyer son signal à quasiment toutes les tuiles !

On ne peut donc pas atteindre les 250 MHz obtenus avec un petit LFSR seul. La fréquence a chuté d'un quart, mais reste encore très bonne : 190 MHz, c'est trois fois plus rapide que le cœur ARM conçu spécialement pour cette famille de FPGA !

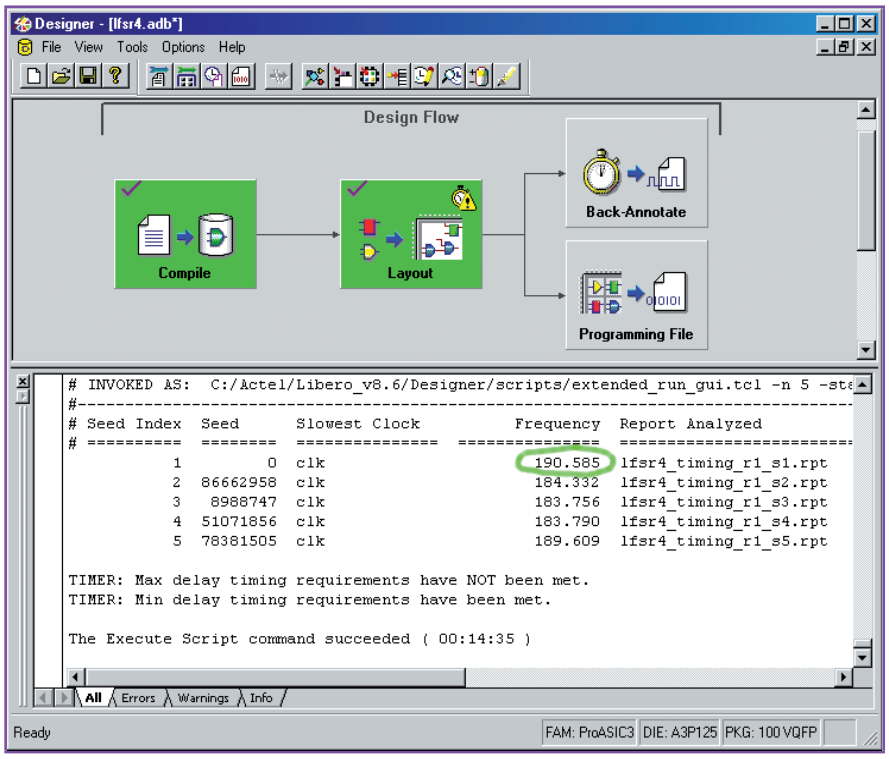


Figure 8 : Cette fois-ci, la vitesse est moins élevée, mais on frise encore les 200 MHz.

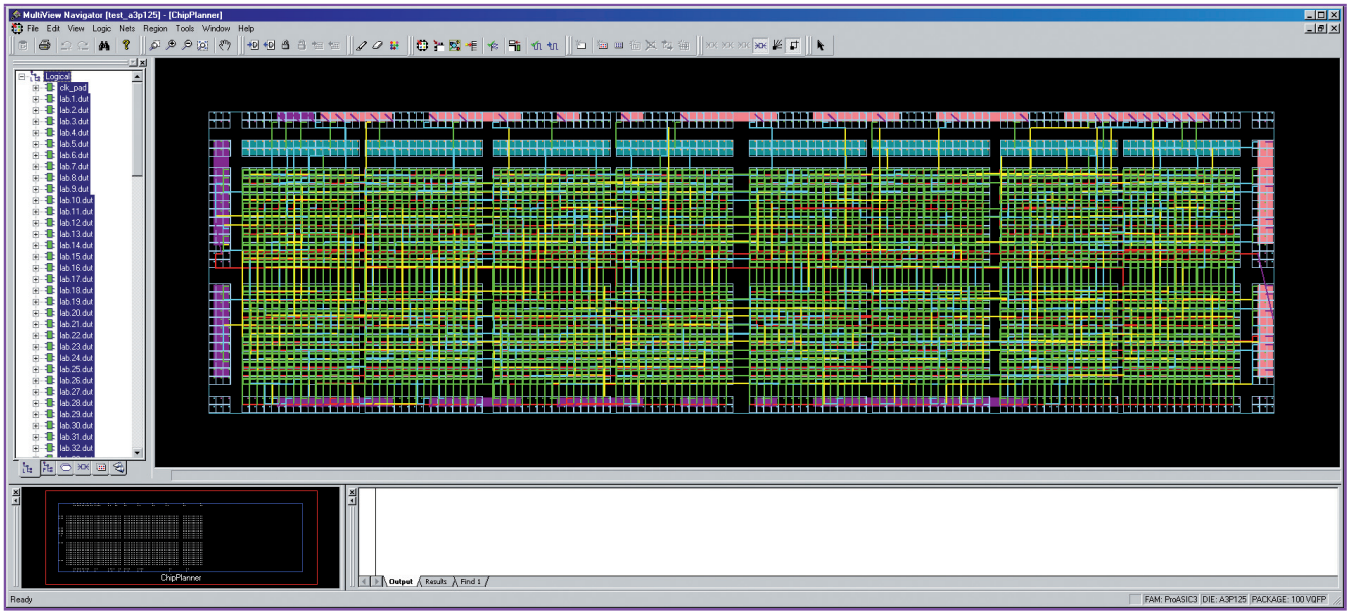


Figure 7 : Toute la matrice est effectivement occupée et le routage s'effectue sans problème.

5 Application industrielle : vérification des circuits intégrés

Un des nombreux défis de la conception d'un circuit électronique numérique intégré moderne est de vérifier que la puce qui sort des fours du fabricant est bien fonctionnelle. Et si possible, il faut effectuer la vérification avant l'intégration dans le boîtier, car ce dernier représente une part importante du coût total et il serait stupide d'y mettre une puce défectueuse...

5.1 Méthodologie de test embarqué

Il y a encore peu de temps, on soumettait encore les circuits à des batteries de tests en usine, au moyen d'appareils sophistiqués et très spécifiques. Cette approche n'est plus gérable en raison du coût énorme de ces appareils, qui doivent être adaptés et reprogrammés pour chaque nouveau type de puce à tester (ce qui a aussi un coût).

Plus problématique encore : la vitesse des signaux traités (aujourd'hui, plusieurs centaines de MHz) ne permet plus d'examiner facilement les propriétés électriques et temporelles des signaux internes de la puce. Les sondes que l'on pose dessus, ou les broches d'entrées/sorties, ne permettent plus des vitesses suffisantes et ce manque de bande passante rend les tests classiques très difficiles.

L'intégration croissante des fonctions permet une nouvelle approche : on peut inclure une partie des tests dans la puce. Le terme consacré est BIST, pour *Built-In Self Test* (autovérification embarquée). Cela peut représenter moins d'un millième à plus de 10 % de surface de silicium de plus, selon la complexité du circuit **[7]**. Les gains de productivité justifient cette surface, car détecter rapidement les puces défectueuses permet d'économiser du temps lors du test final, donc de produire plus de puces à l'heure et d'améliorer la rentabilité.

On ne va donc pas intégrer tout un test exhaustif, mais dégrossir, détecter autant d'erreurs simples que possible. L'objectif est de s'assurer que le circuit fonctionne suffisamment bien à vitesse maximale pour pouvoir charger ensuite des tests supplémentaires et spécifiques. Par exemple, s'il s'agit d'un microprocesseur, on lui fera charger dans un deuxième temps dans sa mémoire des programmes spéciaux pour tester les cas non couverts ou difficiles.

Lorsqu'on effectue un test, on injecte dans le circuit un ensemble de données prédéfinies et on compare la sortie avec d'autres données dépendantes des données d'entrées. Toutes ces données sont appelées vecteurs de test et elles font partie intégrante du processus de conception du circuit. On parle souvent de la méthodologie DFT, *Design For Test*, qui vise à intégrer les tests au plus tôt lors de la création de la puce pour faciliter son industrialisation.

La création de ces vecteurs nécessite des logiciels complexes pour gérer les circuits de plus en plus sophistiqués. La taille de ces vecteurs ne permet pas non plus de les stocker sur la puce elle-même, il faudrait plutôt créer un circuit (simple si possible) qui générerait les vecteurs automatiquement, quitte à avoir des doublons. Au moins, on économise de la surface de silicium et le test est beaucoup plus rapide.

5.2 Tester un circuit grâce à un LFSR

Imaginons que nous devons tester un circuit combinatoire, tel un additionneur prenant en entrée deux nombres de 32 bits. Les deux opérandes totalisent 64 bits, sans compter les bits optionnels tels que le contrôle de la soustraction (qui est dérivée de l'addition) et la retenue entrante.

Si nous voulons tester exhaustivement le circuit, il faudrait fournir 2^{66} vecteurs de tests, nécessitant un espace de stockage irréaliste. On peut éviter le stockage de ces vecteurs en utilisant un compteur sur 66 bits, mais il lui faudrait 8000 ans pour égrener toutes les combinaisons à 1 GHz.

N'oublions pas non plus que certaines fautes dépendent de l'état précédent du circuit (à cause d'effets capacitifs ou d'autres erreurs encore plus sournoises) et ce n'est plus 2^{66} combinaisons qu'il faut tester mais 2^{132} ! Un compteur n'est clairement pas adapté. Par contre, un LFSR possède des propriétés particulières de confusion et de diffusion des bits, ce qui le rend idéal ici.

Dans notre exemple d'additionneur, supposons qu'il y ait un défaut de fabrication affectant la connexion de la porte logique qui calcule le bit de poids fort. En utilisant un compteur, il faudrait attendre au moins 4000 ans pour découvrir le problème. Avec un LFSR, il faudrait juste quelques centaines ou milliers de cycles, car les bits sont rapidement mélangés.

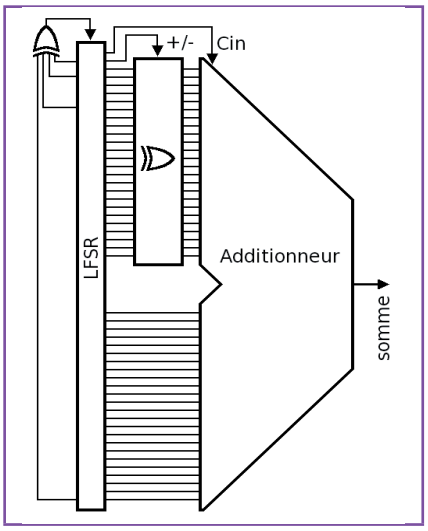


Figure 9 : Un LFSR est connecté à l'entrée d'un additionneur/soustracteur pour lui envoyer des vecteurs de test.

Pour les combinaisons de bits plus complexes, pas de souci non plus : on sait qu'un LFSR génère toutes les séquences, de 1 à N bits identiques contigus. On peut ainsi tester la plupart des combinaisons de bits sans les énumérer une par une.

Le programme VHDL suivant montre comment un LFSR s'approche des propriétés d'une source aléatoire pour détecter la faute que l'on a introduite dans un additionneur-soustracteur. Cette faute dépend d'une combinaison arbitraire de 7 bits en entrée et force un bit en sortie. On la détecte facilement, en moins de mille cycles.

```
library ieee; use ieee.std_logic_1164.all;
              use ieee.numeric_std.all;

entity BIST_addsub is end BIST_addsub;

architecture BIST of BIST_addsub is
  signal result, clk, reset : std_ulogic;
  signal operandes : std_ulogic_vector(66 downto 1);
  signal result_dut, result_ref : std_ulogic_vector(33 downto 1);
begin
  -- notre LFSR générateur de vecteurs de tests
 lfsr : entity work.lfsr4
    generic map(size => 66)
    port map (              -- operandes(1) : add/substract
      clk => clk,           -- operandes(2) : carry in
      reset => reset,       -- operandes(34 downto 3) : substractend
      lfsr => operandes);   -- operandes(66 downto 35) : addend
  -- additionneur/soustracteur de référence
  reference: process(operandes) is
    variable addend, substractend : std_ulogic_vector(34 downto 1);
    variable res : unsigned(34 downto 1);
  begin
    -- fait de la place pour la retenue entrante et sortante :
    addend := '0' & operandes(66 downto 35) & '1';
    substractend := '0' & operandes(34 downto 2);
    -- A-B = A+(-B) = A+(not B + 1) :
    if (operandes(1) = '1') then
      substractend := not substractend;
    end if;
    -- calcule l'addition avec les retenues
    res := unsigned(addend) + unsigned(substractend);
    -- écrit la retenue sortante mais pas entrante
    result_ref <= std_ulogic_vector(res(34 downto 2));
  end process;
  -- additionneur/soustracteur à tester
  faulty: process(operandes) is
  -- pareil que "reference" mais avec une faute conditionnelle injectée
    ...
    if (operandes(34 downto 28) = "0110110") then
      res(33) := '1';  -- force le bit à 1
    end if;
    result_dut <= std_ulogic_vector(res(34 downto 2));
  end process;

  process
    -- nécessaire pour afficher un std_ulogic_vector
    function sulv2txt(s : std_ulogic_vector) return string is
      variable t : string(s'range);
      variable u : string(3 downto 1);
    begin
      for i in s'range loop
        u := std_ulogic'image(s(i));
        t(i) := u(2);
      end loop;
      return t;
    end sulv2txt;
    variable r : std_ulogic_vector(33 downto 1);
  begin
    clk <= '0';   -- initialisation
    reset <= '1';  wait for 1 ns;
    reset <= '0';
    for i in 1 to 100000 loop
      clk <= '1'; wait for 1 ns;
      clk <= '0'; wait for 1 ns;
      r := result_ref xor result_dut;
      -- vérifie si les deux sorties sont identiques
      if (r /= (33 downto 1=>'0')) then
        report integer'image(i) & " : " & sulv2txt(operandes)
            & " - " & sulv2txt(r);
      end if;
    end loop;
    wait;
  end process;
end BIST;
```

Le programme est divisé en quatre parties : le LFSR4 instancié, l'additionneur/soustracteur de référence, celui à tester (contenant la faute), et enfin, le process de contrôle du test. Ce dernier contient une fonction pour convertir le type **`std_ulogic_vector`** en **`string`**, car une telle fonction est absente par défaut en VHDL.

La compilation et l'exécution du code ne suscitent aucun commentaire :

```
yg$ ghdl -a lfsr4.vhdl BIST_addsub.vhdl && ghdl -e BIST_addsub &&
./bist_addsub
BIST_addsub.vhdl:95:9:@1555ns:(report note): 777 :
 10100110100000101000000000000110011011001010101011101110010010100
  - 01000000000000000000000000000000000
(snip 339 lignes)
BIST_addsub.vhdl:95:9:@199713ns:(report note): 99856 :
 11001110010001001001001101101010011011011010100010010101111010011O
  - 01000000000000000000000000000000000
yg$ _
```

Le LFSR distribue les bits progressivement lorsque le registre est initialisé à 1, ce qui permet de tester toutes les combinaisons avec un seul bit à 1, puis plusieurs.

La faute qui a été injectée dépend de 7 bits en entrée, il y a donc une chance sur 2^7=128 qu'elle se produise. Comme c'est une erreur qui force un bit de sortie, il y a donc une chance sur deux pour qu'elle apparaisse (si on attend 1 et on voit 1, on ne voit pas la faute). Notre programme détecte 341 erreurs après 100000 itérations, ce qui est suffisamment proche de la valeur théorique de 100000/(128×2)=390. On peut supposer que lors d'un test sommaire avec dix millions de cycles, ce système détectera une erreur conditionnelle dépendante de $\log_2(10^7/2)$=25 bits (approximativement).

Par rapport à un compteur, un LFSR a encore d'autres avantages : d'abord, la taille du circuit est plus petite, quelques portes XOR comparées à un long circuit de retenue. Ensuite, et c'est une conséquence directe de l'élimination de la retenue, le circuit peut fonctionner beaucoup plus vite ! Enfin, les LFSR génèrent des vecteurs adaptés à la majorité des circuits numériques, leur utilisation ne se limite pas à notre petit exemple d'additionneur.

5.3 Détecter les signes d'une faute matérielle

Un simple LFSR nous permet donc de créer un générateur de vecteurs de tests très simple, très rapide et fournissant une couverture satisfaisante. Mais nous n'avons économisé que la moitié des vecteurs des tests ! Nous devons encore comparer la sortie avec d'autres vecteurs.

Encore une fois, il n'est pas question de les mémoriser, car ils prendraient trop de place, ou de les transférer hors du circuit, car il n'y aurait pas assez de bande passante. On ne peut pas non plus les générer à la volée comme on l'a fait avec

les vecteurs d'entrée, car ce serait trop complexe. En fait, on n'a pas besoin de comparer un à un les vecteurs avec la sortie du circuit à tester, puisqu'on veut juste savoir si la puce a une erreur de fabrication, on se moque de savoir où elle se trouve. On économise du temps de test en combinant les résultats successifs et en comparant cette combinaison avec une signature précalculée.

Cette démarche de compaction des résultats est très similaire à un domaine voisin, celui des télécommunications. On y utilise beaucoup les CRC (*Cyclic Redundancy Check*, l'objet de plusieurs articles dans GNU/Linux Magazine en 2006), qui utilisent les mêmes notions mathématiques que les LFSR.

La différence entre un CRC et un LFSR est qu'un LFSR génère une suite de bits, alors qu'un CRC n'a pas de sortie, mais une entrée qui perturbe l'état du registre à décalage. Cela signifie que l'on peut réutiliser le même code et les mêmes polynômes qu'un LFSR pour générer une signature d'un flux de bits, avec les mêmes avantages. Un CRC favorisera toutefois un polynôme avec un maximum de facteurs pour diffuser autant que possible une perturbation.

Notre code de LFSR dispose déjà d'une entrée auxiliaire **`Din`**, mais elle n'a qu'un bit de large alors qu'il nous en faut 33 en tout. Il faut donc ajouter 32 autres portes XOR pour perturber l'état d'autant de bits du registre à décalage. Le registre résultant n'est plus appelé LFSR mais MISR (*Multiple Input Shift Register*).

Dans un circuit réel, il est souvent difficile de tester certains signaux de contrôle. Il est facile de tester 95 % d'un circuit, mais c'est le dernier pourcent qui nécessite le plus d'efforts. Avec nos LFSR modifiés, on peut augmenter la couverture du test en lisant la valeur de certains signaux internes, en plus des sorties.

En plus, le circuit de test peut être réutilisé pour fournir de nouvelles fonctionnalités au système. Par exemple, si notre circuit est un microprocesseur,

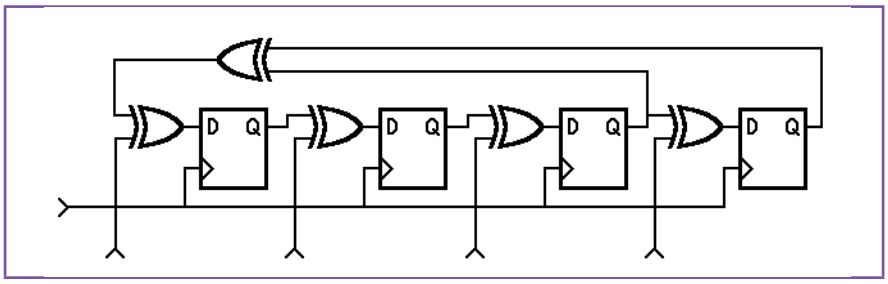


Figure 10 : Schéma d'un MISR à 4 bits, un LFSR dont tous les bits sont aussi affectés par des signaux extérieurs.

le LFSR peut servir comme générateur d'entropie, il fournira alors en prime des valeurs pseudo-aléatoires aux applications. Et pour continuer le test du système, un programme de test complémentaire peut réaliser un LFSR sous forme logicielle.

5.4 Déroulement du test

Tout d'abord, il faut déjà déterminer combien de temps la vérification va durer. On va considérer qu'il s'agit du tout premier test, lorsque le *wafer* est terminé et avant son découpage en petites puces indépendantes. Un bras robotique va successivement placer des sondes sur chaque puce du wafer et lancer notre test embarqué pendant par exemple un dixième de seconde.

Si le circuit est testé à 100 MHz, le LFSR va effectuer dix millions de cycles durant ce bref instant. Cependant, les circuits actuels dépassent le 1 GHz de fréquence interne de fonctionnement alors que les sondes économiques dépassent difficilement 100 MHz. On va donc compter sur les propriétés de confusion et de diffusion des LFSR pour faire en sorte qu'une erreur persiste dans le registre interne.

On n'a alors plus besoin de sortir tous les bits de la puce. Si on teste par exemple un bit sur seize, une sonde économique à 100 MHz suffira pour tester un circuit travaillant à 1,6 GHz. Le testeur robotique compare alors le flux de bits avec celui fourni par un circuit de référence

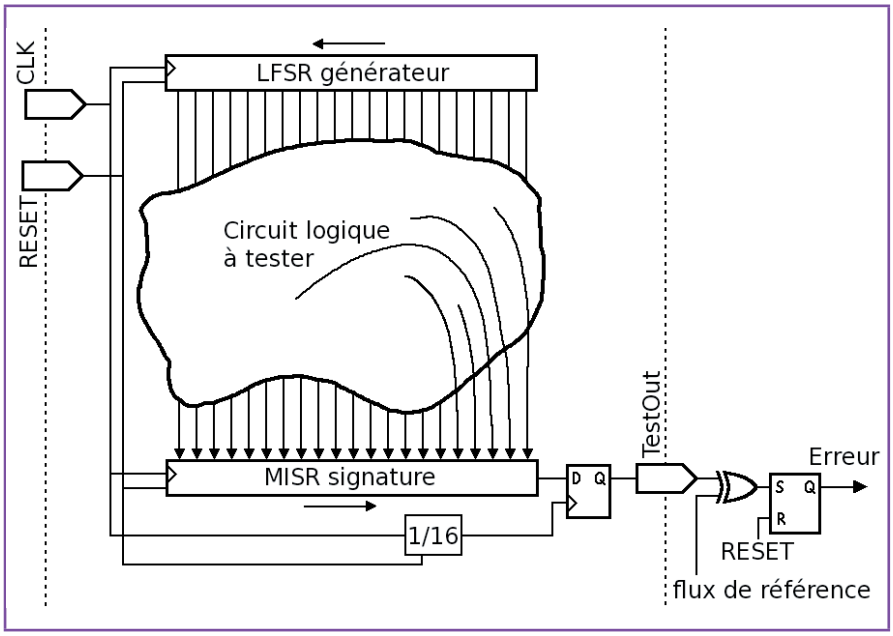


Figure 11 : Diagramme du système de test intégré : le circuit logique est contrôlé par un LFSR générateur, puis un MISR en extrait la signature. Le flux de bits de signature peut être décimé pour faciliter la comparaison avec un système de référence plus lent.

ou un FPGA. Un XOR entre ces deux flux indiquera que la puce à tester est défectueuse lorsque le résultat sera différent de 0.

Il faut absolument effectuer la comparaison et la validation de la sortie du LFSR hors du circuit à tester. En effet, s'il y a un défaut de fabrication dans le circuit de test interne, il peut donner un mauvais résultat, un faux positif.

Autre détail important : pour éviter que les valeurs des LFSR de génération et de signature s'annulent, il faut faire décaler l'un d'eux (la signature, par exemple) dans un sens différent. Le LFSR de signature a été complètement réécrit dans l'exemple suivant :

```
library ieee; use ieee.std_logic_1164.all;
              use ieee.numeric_std.all;
library std;  use std.textio.all;

entity BIST_signature is
end BIST_signature;

architecture BIST of BIST_signature is
  signal result, clk, reset : std_ulogic;
  signal operandes : std_ulogic_vector(66 downto 1);
  signal result_dut, result_ref,
         sign_dut, sign_ref : std_ulogic_vector(33 downto 1);
begin
-- copie du début de BIST_addsub.vhdl : le LFSR4 et les 2 additionneurs

 -- ce process est une version figée et réfléchie de LFSR4 avec 33 bits
 signref : process(result_ref, reset, clk)
  begin
    -- polynôme réfléchi : 33 32 29 27 => 1 2 5 7
    if reset = '1' then
      sign_ref <= (1=>'1', others=>'0');
    else
      if rising_edge(clk) then
        sign_ref <= result_ref
           xor ((sign_ref(1) xor sign_ref(2)
             xor sign_ref(5) xor sign_ref(7))
               & sign_ref(33 downto 2));
      end if;
    end if;
  end process;

-- process identique mais traitant l'additionneur défectueux :
 signdut : process(result_dut, reset, clk)
  begin
    if reset = '1' then
      sign_dut <= (1=>'1', others=>'0');
    else
      if rising_edge(clk) then
        sign_dut <= result_dut
           xor ((sign_dut(1) xor sign_dut(2)
             xor sign_dut(5) xor sign_dut(7))
               & sign_dut(33 downto 2));
      end if;
    end if;
  end process;

  process
    variable r : std_ulogic;
    variable u : string(3 downto 1);
    variable l : line;
 begin
    clk <= '0';    reset <= '1';
    wait for 1 ns; reset <= '0';
    for i in 1 to 100 loop
      for j in 1 to 70 loop -- 7000 pas de temps simulés
        clk <= '1'; wait for 1 ns;
        clk <= '0'; wait for 1 ns;
        r := sign_dut(1) xor sign_ref(1);  -- compare les deux additionneurs
        u := std_ulogic'image(r);
        write(l,u(2));  -- affiche la différence, 1 si erreur détectée
      end loop;
      writeline(output,l); -- affiche toute une ligne de 70 bits
    end loop;
    wait;
  end process;
end BIST;
```

Deux process ont été ajoutés et le process de contrôle se contente d'afficher la différence entre les deux sorties de signature. Le résultat arrive très vite :

```
yg$ ghdl -a lfsr4.vhdl BIST_signature.vhdl && ghdl -e BIST_signature
      && ./bist_signature
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
(snip 7 lignes)
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000001000000000000000000000000000010100
1100000000000000000000010001000000101000000000000000101011000100001111100
0000001000000010000000000001000100101001101010011010001010110101110001 0
1000001000011000011111001101011110010000110110000011111011110010110001 <=
0101110010000001000010001010010001000011010010001101110101011100000010 00
1101100011111001010010000101000101101100011001101010010011011111001101 <=
0110110100010010111010111110110100010001010100011011110011010001001110
...
```

Même en sautant de nombreux cycles, il suffit qu'une seule comparaison échoue pour que le défaut soit détecté, car l'erreur persiste et se propage dans le registre. Lorsqu'on compare le flux de sortie avec le flux de référence, un bit sur deux en moyenne diffère.

Dans le listing ci-dessus, si on ne lit que la dernière colonne de chaque ligne, l'erreur apparaît dès la quatorzième ligne. Ce qui signifie que le test robotisé peut marquer la puce comme défectueuse au bout de seulement 910 cycles d'horloge.

La détection d'une erreur simple dure donc moins d'une microseconde pour une horloge à 1 GHz. L'estimation de la durée d'un test à un dixième de seconde est donc confortable pour notre circuit. En effet, il faudra au moins autant de temps pour effectuer les opérations robotiques : déplacer et repositionner les sondes, établir les contacts des électrodes, alimenter le circuit, ...

Maintenant, supposons que notre circuit contient de nombreuses autres unités qui doivent aussi être testées. Le test va-t-il durer plus longtemps ? Non, car encore une fois, nous pouvons combiner les résultats des différents tests pour éviter d'y passer plus de temps.

Évidemment, un circuit intégré réel (tel un *System On Chip* de dernière génération) est considérablement plus compliqué que notre petit additionneur. Nous avons vu un exemple avec

de la pure logique combinatoire, et les stratégies de test sont beaucoup plus évoluées lorsque la logique séquentielle entre en jeu **[7]**. Cependant, cet article a démontré que les LFSR et leurs dérivés sont des outils indispensables !

Conclusion

Nous avons mis en œuvre des LFSR et créé du code source VHDL permettant leur utilisation dans diverses situations. Nous avons rappelé quelques principes théoriques et vu des exemples pratiques de la vie de tous les jours d'un électronicien. En plus de quelques jolies copies d'écran, nous avons aussi deux exemples de syntaxes pour coder les tableaux en VHDL, ainsi que deux tables (abrégées) de polynômes. Enfin, un des procédés de test des puces électroniques a été démystifié.

Avec tout cela, vous devriez maintenant être en mesure d'utiliser les LFSR vous-même, en logiciel ou en circuiterie logique. Et si vous en avez besoin, les codes sources présentés ici sont disponibles à http://ygdes.com/GHDL/. ■

Références

[1] Guidon (Yann), « Comprendre les générateurs de nombres pseudo-aléatoires », *GNU/Linux Magazine France* n° 81, mars 2006, pp. 64-76, http://www.unixgarden.com/index.php/comprendre/comprendre-les-generateurs-de-nombres-pseudo-aleatoires

[2] Guidon (Yann), « Création d'un afficheur 7 segments avec GHDL », *GNU/Linux Magazine France Hors Série* n° 51, décembre 2010, http://www.ed-diamond.com/feuille_lmhs51/index.html

[3] Schneier (Bruce), « Cryptographie appliquée », Wiley & Son, édition française, 1997, pp. 398-399

[4] http://en.wikipedia.org/wiki/LFSR

[5] Roy Ward, Tim Molteno, « *Table of Linear Feedback Shift Registers* », University of Otago, New Zealand, October 26, 2007, http://www.physics.otago.ac.nz/px/research/electronics/papers/technical-reports/lfsr_table.pdf (lien cassé)

miroir : http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf

[6] Alfke (Peter) « *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators* », Xilinx Corporation Application note XAPP 052, July 7,1996, http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf

[7] « *Testing and testable design of digital systems* », 2002, http://csserver.evansville.edu/~mr56/ece553/Lecture_17.pdf, http://csserver.evansville.edu/~mr56/ece553/Lecture_18.pdf

EXPLORATION DU NETGEAR READYNAS DUO

par Denis Bodor

Les NAS ou Network Attached Storage sont des périphériques de plus en plus présents, aussi bien dans le domaine professionnel que chez le particulier. GNU/Linux est un système utilisé de longue date sur ces fameux disques durs réseau. L'adaptabilité du système en fait un choix pertinent de la part des constructeurs qui souvent ne rediffusent pourtant qu'une partie du code sous la forme de sources « vanilla ». NetGear, comme d'autres, propose une approche sensiblement différente, bien plus orientée vers le développement communautaire. Résultat : une gamme de NAS ouverts et faciles à adapter à ses besoins.

Pour comprendre d'où vient l'intérêt d'un constructeur de NAS pour GNU/Linux, il suffit de revenir à la définition la plus basique de ce type de matériel. Un NAS est un système de stockage centralisé connecté en réseau. En d'autres termes, il s'agit d'une machine, disposant d'une grande quantité d'espace de stockage, mise à disposition via un ensemble de protocoles standards comme CIFS, NFS, FTP, AFP, etc. Il apparaît alors comme évident le choix du noyau Linux et des outils GNU. Tout existe déjà, du support pour le *monitoring* des unités de stockage, en passant par la redondance ou l'implémentation des protocoles de partage. Après adaptation du système, il ne reste plus, ensuite, qu'à fournir une interface utilisateur (généralement de type Web/HTML) et de réunir l'ensemble sous la forme d'un *firmware* homogène. Tout ce que propose Linux est ainsi directement utilisable sous la forme d'un NAS.

Bien entendu, de nombreux constructeurs ont compris cela et proposent des périphériques reposant sur les technologies *open source*. NetGear, LaCie, Synologie, Linksys/Cisco, etc. Tous, par contre, ne proposent pas automatiquement un système modulaire permettant à l'utilisateur avancé d'adapter son matériel. L'administrateur système souhaitant disposer d'une solution personnalisée doit alors soit hacker le firmware, soit construire son NAS à partir d'une configuration de type PC et d'une distribution comme CryptoNAS ou encore FreeNAS reposant sur FreeBSD.

C'est avec le périphérique Linksys NSLU2 (*Network Storage Link for USB 2.0 Disk Drives*) que les besoins de personnalisation se sont faits évidents. Peu après la mise en vente de l'appareil en 2004, une communauté d'utilisateurs/développeurs s'est formée et Linksys respectant les licences utilisées dans le NSLU2 a diffusé les sources du firmware. Plusieurs projets ont vu le jour, générant ainsi une masse énorme de documentations, des firmwares alternatifs comme Unslung ou SlugOS/BE et des analyses poussées et complètes à la fois du matériel et du système original. Les utilisateurs ont ainsi ajouté un nombre de fonctionnalités impressionnant, comme l'intégration de serveurs de messagerie, de solutions UPnP, de clients BitTorrent ou encore de serveurs audio pour iTunes (DAAP). Le NSLU2 a même été transformé en PBX Asterisk et en routeur/firewall. Malheureusement, les limitations techniques du produit et la connectique USB pour les disques, en particulier, réduisaient le champ d'application. En effet, en USB, point de monitoring SMART (*Self-Monitoring, Analysis and Reporting Technology*) et donc peu de chance de voir le produit dans un environnement de production.

Ce qu'il fallait retenir de cette histoire est simple : un constructeur de périphériques peut utiliser le logiciel comme un plus dans ses gammes de produits, mais il reste un vendeur de matériel. En ouvrant les sources du firmware, Linksys a permis au NSLU2 d'être plus qu'une solution figée et clés en main. Les développements qui ont été faits autour de ce matériel ont démultiplié les fonctionnalités et popularisé la solution. Nombreux sont les utilisateurs ayant acquis un NSLU2 dans le seul but d'utiliser Unslug, qu'ils jugeaient plus adapté à leurs besoins.

C'est là quelque chose que NetGear a clairement compris. Fabriquons du matériel, fournissons un firmware de qualité et modulaire et ouvrons les sources. Et bien entendu, la mayonnaise a pris ! La gamme ReadyNAS dispose de près d'une cinquantaine de plugins communautaires permettant d'étendre les fonctionnalités des produits de la gamme, ceux-ci s'ajoutant à la vingtaine de plugins officiels fournis par NetGear ou ses partenaires.

1 ReadyNAS Duo : Hands on

Le modèle utilisé comme base pour cet article est le ReadyNAS Duo RND2000. Il s'agit du boîtier NAS vendu nu et donc livré sans disque. Il faudra donc ajouter un ou deux disques SATA. Nous avons opté pour des disques Samsung SpinPoint F3 1 To pour un coût de revient total d'environ 250 euros. Les disques se placent dans des paniers directement encastrables dans le NAS et le premier démarrage déclenchera la procédure d'initialisation pouvant durer jusqu'à 30 bonnes minutes. Par défaut, le périphérique utilisera un schéma compatible RAID niveau 1 (miroir). Si vous souhaitez utiliser la capacité complète des deux disques en RAID 0 (agrégation), il faudra suivre une procédure spécifique après mise à jour éventuelle du firmware, appelé RAIDiator. L'utilisation de RAID 0 ou 1 est sujette à débat. En effet, un NAS a généralement pour objectif de garantir la sécurité des données tout en servant d'espace de stockage volumineux. Certains diront que le fait de favoriser la taille au détriment de la sécurité est une mauvaise chose. D'autres préciseront que cela dépend totalement de l'utilisation finale du NAS. La configuration par défaut est cependant spécifique à Netgear, utilisant une technologie brevetée appelée X-RAID (pour *eXpandable RAID*). Cette technologie permet de dynamiquement étendre le RAID. Avec un disque, vous n'avez pas de réplication. Ajoutez un disque et vous utilisez la capacité d'un disque mais avec une redondance. Ajoutez encore un disque et vous avez la capacité de deux disques, plus une redondance, etc., etc., etc. Nous avons ici choisi de rester en X-RAID par défaut dans un premier temps, car c'est là l'objet même de l'utilisation du périphérique et que cela ne change en rien les fonctionnalités de personnalisation que nous allons évoquer. Nous reviendrons sur le sujet plus après dans l'article.

Après démarrage effectif du périphérique, celui-ci, naturellement, s'auto-configure avec DHCP, offrant ainsi accès à l'interface d'administration web via https://adresse/admin. Un assistant de configuration vous permet alors de définir vos partages, les utilisateurs, ainsi que les diverses fonctionnalités à activer.

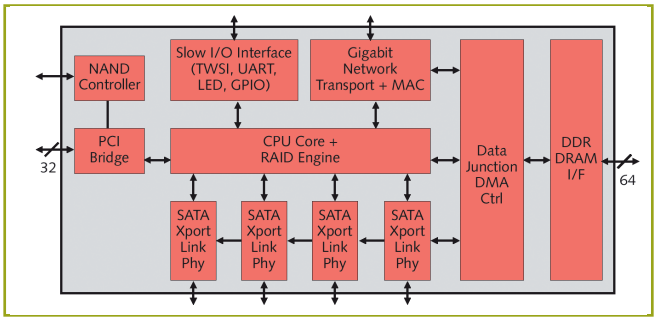
L'interface web du NAS est relativement classique et présente toutes les informations de configuration et d'état concernant le matériel.

Première et agréable surprise ici, on a le plaisir de constater que NFS n'a pas totalement disparu des configurations proposées par les fabricants. NFS et, par là même, les utilisateurs Unix purs et durs, sont généralement les grands oubliés des NAS récents. Merci Netgear de vous rappeler que CIFS/SMB n'est de loin pas le protocole le mieux pensé ! Mais ce n'est pas tout : en plus de CIFS, NFS, AFP, FTP, HTTP et HTTPS, on constate une autre option (je ne peux me retenir de copier ici le nom complet tant c'est agréable) : « RSYNC , un protocole de sauvegardes incrémentales populaire dans le monde Unix et Linux ». Côté services installés par défaut, on trouve le protocole de diffusion audio de la Squeezebox, le streaming pour iTunes (+ Bonjour) et la diffusion UPnP/DLNA. Dans les modules complémentaires, on trouvera le BitTorrent, un service ReadyNAS Photo et une fonction d'accès à distance appelée ReadyNAS Remote.

Première étape après démarrage, la mise à jour du firmware vers RAIDiator 4.1.7 du 12 novembre dernier via l'interface web, avec une récupération automatique de la version par le Net (à condition que votre serveur DHCP ait effectivement envoyé toutes les informations au NAS, bien entendu). On se tourne alors naturellement vers le site http://www.readynas.com pour télécharger le greffon « Enable Root SSH Access ». Il s'agit d'un greffon officiel (non communautaire) permettant d'activer, vous vous en doutez, l'accès SSH au système installé sur le NAS. Après récupération et installation via l'interface web, le port 22 est accessible. Le login à utiliser est **root** et le mot de passe, celui que vous aurez défini dans la phase d'installation du NAS.

C'est le moment de faire véritablement connaissance avec la bête :

```
# cat /etc/issue
RAIDiator 4.1.6 [192.168.0.9]

readynas:~# cat /proc/cpuinfo
cpu             : Infrant Technologics, Inc. - neon version: 0
fpu             : Softfpu
ncpus probed    : 1
ncpus active    : 1
BogoMips        : 186.36
MMU             : version: 0
LP              : HW.FW version: 0.1
FPGA            : fpga000000-0 Configuration: 0
AHB arbitraion  : 7
CPU id          : 0
Switch          : 0
ASIC            : IT3107
```



Le processeur IT3107 d'Infrant Technologies est un SoC RISC spécialement dédié à la construction de NAS et intégrant Gigabit Ethernet, plusieurs interfaces SATA (4 pour l'IT3107 et 2 pour IT3102) et des fonctionnalités RAID matériel (0/1/5).

L'architecture générale du ReadyNAS Duo est assez similaire à celle du NAS600 d'Infrant (même l'interface web est très proche). En en cherchant davantage sur l'IT3107, on découvre qu'il s'agit d'une conception basée sur le softcore LEON1. Preuve, s'il en faut, que là encore, l'open source démontre son adaptabilité et ses avantages. Mais il y a plus intéressant pour nous. En effet, le LEON implémente les instructions SPARC V8 et les registres SPARC courants. En d'autres termes, nous avons affaire à une architecture SPARC, comme le démontre la simple utilisation de la commande **file** :

```
readynas:~# file /bin/ls
/bin/ls: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
for GNU/Linux 2.2.0, dynamically linked (uses shared libs),
stripped
```

Voilà qui est plutôt intéressant en termes de développements exotiques. Mais nous ne sommes pas au bout de nos surprises. Le système est installé sur une partition du disque (**/dev/hdc1** de quelques 2 Go aux trois quarts libre) et les données partagées sont placées sur le point de montage **/c** donnant accès aux données présentes sur **/dev/c/c**. En y regardant de plus près, on constate :

```
readynas:/# lvscan
  ACTIVE            '/dev/c/c' [929.09 GB] inherit
```

```
readynas:/# vgscan
  Reading all physical volumes.  This may take a while...
  Found volume group "c" using metadata type lvm2

readynas:/# pvscan
  PV /dev/hdc5   VG c   lvm2 [929.09 GB / 0    free]
  Total: 1 [929.09 GB] / in use: 1 [929.09 GB] / in no VG: 0 [0   ]
```

Nous avons donc affaire à du LVM2. La partition **hdc5** est un volume physique, compris dans un groupe de volume **c**, lui-même utilisé par un volume logique **c** monté dans **/c**. Mais où est le RAID ? Nous n'avons pas affaire à du RAID logiciel, mais du X-RAID directement contrôlé au niveau noyau/SoC, comme tend à le montrer le contenu de **/proc** :

```
readynas:/# cat /proc/xraid/configuration
VERSION/ID::superblock=(0.1.0),ID=0eb33327.00000000.
    00000000.00000000,create_time=4cdc299e
RAID_INFO::disks_total=2,raid_disks=2,parity_disk=1,
    disks_online=2,disks_working=2,disks_failed=0,spare_disk=0,
    base_disk=0,size=1953108616,update_time=00000000,state=0,
    luns=2,extcmd=1,expandable_bitmap=0x0,lsize=1953108614,
    drive_present=0x3
LOGICAL_DRIVE:0:begin_sector=2,sectors=4096000,raid_level=1,
    status=redundant,initialized=1,dmap=3
LOGICAL_DRIVE:1:begin_sector=4096002,sectors=1949012614,raid_level=1,
    status=redundant,initialized=1,dmap=3
PHYSICAL_DRIVE:0:number=0,device=hdc,major=22,minor=0,raid_id=0,
    state=online,present=1,size=1953108616,r_model=SAMSUNG HD103SJ,
    r_size=1953108616,r_fw=1AJ10001
PHYSICAL_DRIVE:1:number=1,device=hde,major=33,minor=0,raid_id=1,
    state=online,present=1,size=1953108616,r_model=SAMSUNG HD103SJ,
    r_size=1953108616,r_fw=1AJ10001
RUN_PARAMETERS::raid_running=1,last_word=ok,interface_start_at=1,fake=0
RAID_REBUILD::sync=0,logical=0,parity=1,sectors=0,
    total=4294967295,source=3,total_drives=2,auto_sync=1
RAID_P_CHECK::chck=0,current_lun=0,total_luns=2,raid_level=0,err=0,
    current_sector=0,total_sectors=0,report_err=1
RAID_P_INIT::initialized_bitmap=0x3,initialize_error_bitmap=0x0,
    initializing_bitmap=0x0,current_sector=0,total_sector=0
```

En jetant un œil aux scripts de démarrage et à **/etc/init.d/rc3** en particulier, on constate que la configuration et la gestion du X-RAID se font via **/proc/xraid/configuration** avec des commandes comme **echo X_RAID_SYNC_FORCE > /proc/xraid/user_command**. Difficile donc d'envisager de sortir l'un des disques en cas de problème du NAS lui-même pour récupérer les données avec une machine GNU/Linux ou *BSD. Celle-ci ne disposant pas, a priori, d'un noyau intégrant les fonctionnalités adéquates. La connexion du premier disque dur, via un adaptateur USB, affiche effectivement des éléments utiles :

```
Model: SAMSUNG HD103SJ (scsi)
Disk /dev/sdf: 1000GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End     Size    Type      File system  Flags
1       1024B   2097MB  2097MB  primary   ext3
2       2097MB  2359MB  262MB   primary
3       2359MB  1000GB  998GB   extended
5       2359MB  1000GB  998GB   logical               lvm
```

On retrouve la partition système, le swap et la partition étendue contenant une partition logique LVM2 pour nos données. En revanche, le second disque, lui, n'est pas directement utilisable de cette manière :

```
[1809656.673488] scsi 16:0:0:0: Direct-Access SAMSUNG  HD103SJ PQ: 0 ANSI: 2 CCS
[1809656.673881] sd 16:0:0:0: Attached scsi generic sg5 type 0
[1809656.674478] sd 16:0:0:0: [sdf] 1953525168 512-byte logical blocks: (1.00 TB/931 GiB)
[1809656.675351] sd 16:0:0:0: [sdf] Write Protect is off
[1809656.675354] sd 16:0:0:0: [sdf] Mode Sense: 34 00 00 00
[1809656.675357] sd 16:0:0:0: [sdf] Assuming drive cache: write through
[1809656.679635] sd 16:0:0:0: [sdf] Assuming drive cache: write through
[1809656.679640]  sdf: unknown partition table
```

Sans être catastrophique, on constate donc que le X-RAID risque de poser problème, même si, en tant que telle, la solution semble très intéressante (possibilité d'extension du stockage). Le retrait du premier disque provoque un événement parfaitement pris en charge et le NAS continue de remplir son office. La réintégration du disque provoque la synchronisation et le retour à la normale après environ 4h pour 1 To. Nous avons poussé plus loin l'expérience (vicieux nous sommes), en retirant à nouveau le premier disque en pleine synchronisation et en rebootant le NAS. Celui-ci redémarre alors avec le second disque et on retrouve exactement le même schéma de partitions que précédemment (**hdc1**, **hdc2**/**swap**, **hdc3**/**hdc5**). Là, nous réintégrons le premier disque. Un disque **hde** apparaît et la synchronisation démarre. Maintenant, en pleine synchronisation, stoppons le NAS et vérifions le contenu du second disque (je vous ai dit qu'on était vicieux). Résultat : le second disque est toujours illisible via un adaptateur SATA/USB et le premier également, du moins jusqu'à la synchronisation complète sur le NAS. Moralité : comme avec beaucoup de solutions de stockage de ce type, le X-RAID est une fonctionnalité intéressante, mais la défaillance de l'électronique du NAS risque d'impacter l'accès aux données. Ce n'est pas très rassurant, même si c'est toujours plus sûr que les solutions à base de RAID matériel ou les NAS utilisant un RAID totalement propriétaire. Personnellement, je n'aime pas l'idée du « je perds le NAS, je perds forcément les données dans le NAS ».

Netgear met à disposition un outil de monitoring et de configuration appelé RAIDar. En plus de permettre le paramétrage du matériel dans les premières phases d'installation, il offre également la possibilité de surveiller l'état des disques et des divers composants via UDP.

C'est en parcourant les nombreux répertoires à la recherche d'informations diverses que l'on finit par tomber sur l'élément le plus intéressant (à mon avis) : **/var/lib/dpkg** ! Mais ?! Mais alors :

```
readynas:~# dpkg -l
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Cfg-files/Unpacked/Failed-cfg/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: uppercase=bad)
||/ Name         Version             Description
+++-============-===================-==============================
ii  acl          2.2.23-1            Access control list utilities
ii  adduser      3.47                Add and remove users and groups
[...]
ii  zip          2.31-1              Archiver for .zip files
ii  zlib1g       1:1.2.2-4.sarge.2   compression library - runtime
```

À ce stade de la lecture de l'article, certains diront sans doute que ceci était prévisible étant donné la présence d'un greffon APT. Celui-ci permet d'installer la commande **apt-get** offrant la capacité de gestion de paquets sur une distribution Debian. Le fichier **sources.list** intégré au greffon utilise un dépôt **archive.debian.org** pour une distribution Sarge ainsi qu'un dépôt Netgear, principalement pour les paquets de développement qui semblent modifiés par le constructeur pour supporter les spécificités de la plateforme. Quoi qu'il en soit, il devient ainsi possible de compléter l'installation du système via **apt-cache**/**apt-get**. N'oubliez pas, cependant, que l'accès SSH en lui-même est susceptible de vous priver du support de la part du constructeur, ceci est d'autant plus vrai si vous commencez à installer des éléments systèmes via APT.

2 Installation du cross-compilateur

Nous avons affaire à une architecture basée sur SPARC, le réflexe alors est d'envisager l'utilisation d'un cross-compilateur pour faire tout le développement sur du x86 puissant et cibler le NAS. Trouver un environnement de compilation complet à destination de cette plateforme n'est pas chose facile. Heureusement, Julien Lemoine alias speedblue, maintient un dépôt Debian contenant un ensemble de paquets pour Sid parmi lesquels on trouve les **binutils**, **gcc**, **g++**, **glibc-cross**, etc. Il suffira d'ajouter le dépôt dans votre configuration APT avec :

```
# /etc/apt/sources.list.d/cross.list
deb http://debian.speedblue.org ./
deb-src http://debian.speedblue.org ./
```

Attention toutefois ! Netgear semble utiliser des paquets de développement (*headers*) spécifiques. Il faudra donc prévoir de faire un mix de tout cela avec les difficultés qu'on peut imaginer.

Quelques pages du site communautaire de Netgear nous mettent cependant sur la piste de trois autres stratégies de construction de binaires pour les NAS basés sur l'architecture SPARC/Neon/Leon d'Infrant Tech. La première (http://www.readynas.com/?p=146) prend la forme du classique et horrible tarball (http://www.readynas.com/download/development/readynas-cross-3.3.5.tar.gz) à, dixit la documentation, désarchiver « à partir de la racine du système ». Cela aura pour effet de peupler tout un ensemble de sous-répertoires de **/usr** : **bin**, **include**, **lib**,

share et **sparc-linux**. C'est la méthode « sale » qu'on retrouve généralement dans l'embarqué. Personnellement, c'est recalé d'office. J'aime trop mon système pour lui faire quelque chose de ce genre.

Une autre voie possible, également détaillée par le site officiel, est encore plus étonnante. Il s'agit de disposer d'un NAS de développement (entendez par là, un NAS où vous ne comptez pas stocker de données importantes), et d'y installer, en natif, les éléments de développement nécessaires. Il faut pour cela ajouter les greffons APT et **EnableRootSSH** puis, en ligne de commandes, utiliser **apt-get** pour installer les paquets **libc6-dev**, **gcc**, **gdb**, **libtag1-dev** et **uuid-dev**. Vous développerez donc directement sur le système cible. Le site Netgear le précise clairement : *It is recommended that you create and build your add-on right on your ReadyNAS*. Voilà qui n'est pas courant dans le milieu, même s'il faut reconnaître que de nos jours, avec un système pour *smartphone* à la mode faisant fonctionner les applications dans des machines virtuelles Java, on peut presque tout se permettre...

Enfin, dernière solution, et à mon goût la plus adaptée, vous pouvez utiliser un émulateur pour faire fonctionner un système SPARC équivalent à celui du NAS et ainsi développer en natif, mais en disposant de la puissance CPU d'une machine de développement. Pour cela, vous devrez récupérer une image QCOW compressée d'environ 400 Mo sur http://www.readynas.com/download/development/readynas_compile_environment.qcow.gz. Après décompression, le fichier de 1 Go vous servira d'image système pour l'émulateur Qemu. Il vous suffira alors d'utiliser :

```
% qemu-system-sparc -hda \
  readynas_compile_environment.qcow \
  -m 256 -nographic

Configuration device id QEMU version 1 machine id 32
UUID: 00000000-0000-0000-0000-000000000000
CPUs: 1 x FMI,MB86904
Welcome to OpenBIOS v1.0 built on Feb 27 2010 22:57
  Type 'help' for detailed information
[sparc] Booting file 'disk' with parameters ''
Trying disk (/iommu/sbus/espdma/esp/sd@0,0)
Not a bootable ELF image
Not a Linux kernel image
Loading a.out image...
Loaded 7680 bytes
entry point is 0x4000
Jumping to entry point...
SILO Version 1.4.13
boot: [ENTREE]
Uncompressing image...
Loaded kernel version 2.6.18
Loading initial ramdisk (3210249 bytes at
   0x3000000 phys, 0x60000000 virt)...
PROMLIB: obio_ranges 1
Booting Linux...
[...]
debian-sparc login: root
Password:
Linux debian-sparc 2.6.18-6-sparc32 #1
  Fri Dec 12 16:29:52 UTC 2008 sparc GNU/Linux
debian-sparc:~#
```

Le nom d'utilisateur est **root** et le mot de passe **a**. Vous disposerez de 10 Go d'espace pour vos développements tout en bénéficiant de l'infrastructure de votre machine de développement (voir article sur le débogage Qemu dans le présent numéro).

3 Install RAID et non X-RAID

Nous l'avons dit en début d'article, par défaut, le ReadyNAS Duo s'initialise avec une configuration X-RAID. Un autre mode existe cependant, appelé Flex-RAID, permettant de choisir une configuration RAID 0 ou RAID 1 entièrement logiciel. Si le NAS est déjà en service, il vous faudra alors revenir à la configuration d'usine et, par la même occasion, perdre les données des disques. La réinitialisation est, et le mot est faible, relativement pénible jusqu'au moment où l'on comprend la logique utilisée par Netgear. Chose que nous avons durement expérimenté.

Le principe est le suivant : en utilisant le bouton de *reset* placé à l'arrière de l'appareil, il est possible d'influer sur le comportement du *bootloader* **iboot** intégré. Il faut éteindre le NAS, appuyer sur reset à l'aide d'un trombone (ou un objet similaire) et mettre le NAS sous tension. Là, il convient de bien observer les LEDS en façade et en particulier les deux LEDS d'activité des disques (surtout pas le voyant/bouton *power* bleu). Le clignotement vous indique l'action qu'il est possible de faire exécuter au bootloader. La séquence de clignotement est la suivante :

- à 5 secondes : réinstallation du *firmware* depuis la NAND ;
- à 10 secondes : retour à la configuration d'usine ;

Note **À propos des sources**

Vous trouverez sur http://www.readynas.com/?page_id=2324 une liste complète des différentes versions sources sous GPL du firmware RAIDiator. Dans le cas de la version 4.1.7, il s'agit d'un fichier Zip de quelques 430 Mo qui, une fois décompressé, fournit une arborescence de quelques 1,5 Go de sources brutes. Celles-ci proviennent en grande majorité des archives Debian, comme vous le montrera un simple **find . -name control | grep --count debian**. Avec tant d'ouverture du côté de l'interface et de la modularisation, on aurait espéré quelque chose d'un peu plus homogène ou même simplement un **Makefile** ou un script shell permettant de reconstruire RAIDiator. Un peu à l'instar des quelques firmwares de Linksys, comme celui de la caméra Ethernet Cisco WVC54GCA, par exemple.

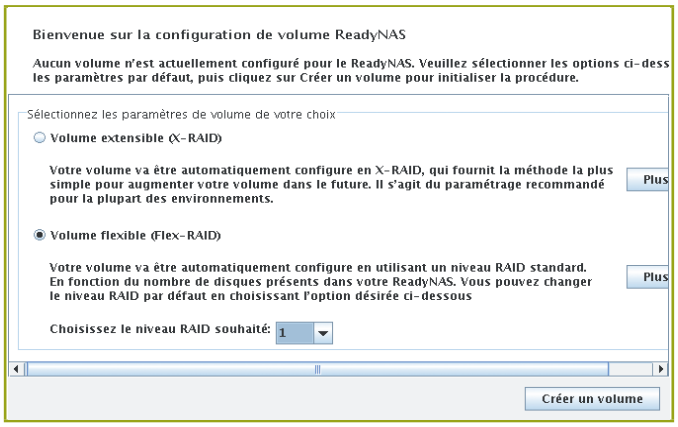
En faisant quelques recherches, on se rend compte que la situation en 2006 était cependant pire que celle d'aujourd'hui. En effet, on retrouve des documentations décrivant comment obtenir un shell **root** et un accès SSH sur le ReadyNAS NV d'Infrant. Les forums dont le contenu est maintenant repris sur http://www.readynas.com/forum regroupent également un certain nombre de discussions autour de l'ouverture du firmware, datant de la même époque. On trouve également quelques retours d'utilisateurs sur http://gpl-violations.org à propos de délais jugés un peu trop longs en ce qui concerne la mise à disposition des sources d'applications en GPL utilisées dans les produits Netgear (2008).

- à 15 secondes : réintsallation du firmware par TFTP ;
- à 20 secondes : réinstallation du firmware par clé USB (http://home.bott.ca/webserver/?p=159).

Ces informations sont vraies pour notre ReadyNas Duo équipé par défaut d'un firmware RAIDiator 4.1.6. Il semblerait cependant que les différentes documentations se contredisent. Certaines parlent de deux clignotements à 5 et 30 secondes (FAQ Netgear) pour une configuration d'usine, alors que d'autres (**README.txt** du **TFTP_Flash_Recovery.zip** également de Netgear) parlent de deux clignotements à 5 et 10 secondes pour une restauration de firmware via TFTP (serveur en 192.168.125.1). La non-disponibilité des sources du bootloader est un obstacle évident pour la clarification de ce point. Dommage.

Dans notre cas, pour un retour à la configuration d'usine, nous avons démarré le NAS avec reset enfoncé jusqu'au second clignotement des deux leds de disque. Là, il faut relâcher reset et le bootloader charge le système pour une réinitialisation de la configuration.

Il n'est pas possible de configurer le mode RAID depuis une interface Web. Il vous faut, pour cela, utiliser l'application Java fournie par Netgear : RAIDar. Communiquant en UDP (port 22081) cette application peut dialoguer avec le NAS dès le début de la séquence de démarrage. L'application ayant détecté le ReadyNAS Duo, un clic sur « Configurer » vous permettra de choisir entre X-RAID et Flex-RAID en spécifiant le niveau de RAID souhaité.



La configuration en RAID logiciel niveau 0 ou 1 est tout à fait possible au prix d'un retour à la configuration d'usine et l'utilisation de RAIDar, l'outil de configuration Java de Netgear.

Une fois toute la phase de réinstallation terminée, on retrouve une configuration classique de RAID logiciel sous GNU/Linux :

Note **Un port console en bonus !**

C'est un réflexe lorsqu'on commence à être un habitué des systèmes embarqués et des *appliances* en tous genres : rechercher un port série pour obtenir une console système. Il faut dire que sur le ReadyNAS Duo, les choses sont plutôt bien faites. Inutile de sortir le fer à souder, le connecteur mâle est directement accessible à l'arrière de l'appareil. Il suffit de retirer un autocollant gris métal situé juste au-dessus du trou pour le bouton reset. On accède ainsi à un connecteur 4 broches avec, de haut en bas, le Vcc en 3.3V, TX, RX et la masse (GND). Bien entendu, il vous faudra passer par un adaptateur TTL/RS232 ou un TTL/USB pour vous connecter, mais ceci fait normalement partie de votre trousse à outils de sysadmin/développeur/hacker embarqué.

La connexion est en 9600 8N1 et vous aurez alors tout loisir d'assister au démarrage verbeux du système et d'obtenir un shell **root** sans activer l'accès SSH. L'autre option est d'accéder au bootloader, mais cela semble une voie sans issue étant donné le manque de documentation et d'aide :

```
Loading ...
Welcome to iboot 1.00a043
built 15:54 2008-10-01
OC[7f]
Reason:
Help:
tftp MAC IP IPsvr file mem
nand
usb
ics v12
ICS reg val
pci v17
reason x
go mem
auto
iboot>
```

iboot semble être également le nom du bootloader d'iPhone, ce qui n'est pas sans poser un certain nombre de problèmes quant aux recherches d'informations sur Internet. On pointera cependant le site http://debugmo.de regroupant quelques pistes intéressantes.

```
readynasduo:~# cat /proc/mdstat
Personalities :
    [raid0] [raid1] [raid5] [raid4]
md1 : active raid1 hde2[1] hdc2[0]
      524224 blocks [2/2] [UU]
md2 : active raid1 hde3[1] hdc3[0]
      974182016 blocks [2/2] [UU]
md0 : active raid1 hde1[1] hdc1[0]
      2047936 blocks [2/2] [UU]
unused devices: <none>

readynasduo:~# lvscan
  ACTIVE            '/dev/c/c' [929.03 GB] inherit

readynasduo:~# vgscan
  Reading all physical volumes.  This may take a while...
  Found volume group "c" using metadata type lvm2

readynasduo:~# pvscan
  PV /dev/md2   VG c   lvm2 [929.03 GB / 0    free]
  Total: 1[929.03 GB]/in use:1[929.03 GB]/in no VG:0[0   ]

readynasduo:~# mount
/dev/md0 on / type ext3 (rw,noatime)
[...]
/dev/c/c on /c type ext3
    (rw,noatime,acl,user_xattr,usrquota,grpquota)
```

Conclusion préliminaire

Oui, « préliminaire » et ce pour deux raisons. La première est que nous n'avons absolument pas abordé la question du développement de greffons et l'intégration dans l'interface web du périphérique. Ici, nous avons préféré mettre l'accent sur l'aspect système du périphérique. Nous reviendrons très certainement sur le sujet dans un prochain article. La seconde raison réside dans le fait qu'un certain nombre de choses restent à faire. Le NAS dispose de trois ports USB, mais en jetant un œil dans **/lib/modules**, on constatera que, bien entendu, très peu de pilotes USB sont présents. Il aurait été intéressant, par exemple, de vérifier la complexité d'un développement noyau, ou du moins de l'intégration de nouveaux pilotes dans le NAS (modules). Je pense tout naturellement aux dongles Wifi, mais également, pourquoi pas, à l'ajout d'un écran/*touchscreen* Mimo utilisant la technologie USB/DisplayLink. L'utilisation de convertisseurs USB/séries est également une possibilité intéressante en termes d'interconnexion avec l'environnement externe du NAS.

Quoi qu'il en soit, je pense que nous avons là un matériel qui repose sur une base solide et possède un certain vécu. Apparemment, autant d'un point de vue de la communication avec des développeurs d'open source que sur le plan technique. J'ai personnellement eu entre les mains un nombre non négligeable de NAS de cette catégorie (utilisation SoHo) et il faut bien avouer que j'ai été globalement et positivement surpris par ce produit Netgear. On aurait cependant souhaité voir encore plus d'ouverture avec un SDK plus travaillé et un accès à une quantité plus importante de sources. Un peu comme ce qu'on trouve dans le monde des routeurs Wifi. Il est à noter d'ailleurs que c'est également Netgear qui est à l'origine de myopenrouter.com, un site communautaire proposant plusieurs firmwares pour les routeurs comme le WNR3500L (de Netgear, bien sûr).

C'est finalement très agréable de voir qu'il existe un réel mouvement de fond des gros constructeurs, non pas seulement vers l'open source, puisque c'est le cas depuis longtemps, mais vers une ouverture pour un développement communautaire. Eh oui, c'est bien là que se trouvent les ressources, la créativité et l'énergie de développement. Donnez plus et vous recevrez plus en retour. C'est tout simple. ■

NOUVELLE FORMULE

GNU/LINUX MAGAZINE N°134

CHEZ VOTRE MARCHAND DE JOURNAUX !

NOUVELLE FORMULE - NOUVELLE FORMULE - NOUVELLE FORMULE

N°134 JANVIER 2011

L 19275 - 134 - F: 6,50 €

GNU LINUX MAGAZINE / FRANCE

Administration et développement sur systèmes UNIX

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

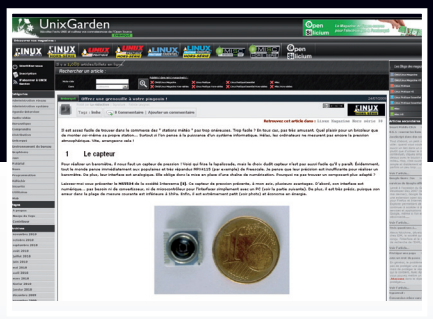
*Sous réserve de toute modification.

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX

JUSQU'AU 28 JANVIER 2011 ET SUR : www.ed-diamond.com

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

Sécurité

Agenda-Interview

Matériel

Web

Jeux

Réfléchir

UnixGarden