N°51 DÉCEMBRE 2010 JANVIER 2011

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

GNU LINUX MAGAZINE / FRANCE HORS-SÉRIE

Administration et développement sur systèmes UNIX

IPHONE / LINUX

Développer des applications iPhone depuis GNU/Linux, c'est possible !

FREEBOX / JAVASCRIPT

Développez des applications pour la Freebox avec les Enlightenment Foundation Libraries

IVY / BUS

Enfin un bus logiciel simple et efficace plus accessible que DBUS

BUILD / X86 & ARM

Premiers pas avec Scratchbox, l'environnement de construction à la base du projet Maemo

PYTHON, ANDROID, C, JAVASCRIPT, IOS, VHDL, ...

HACKS, ÉLECTRONIQUE & EMBARQUÉ

GPS / USB

Magie du GPS et lecture & décodage des données d'un récepteur USB

L 15066 - 51 H - F: 6,50 € - RD

RFID / LAPINS

Exploration et mise en œuvre du lecteur RFID Mir:ror avec GNU/Linux et Python

EPAD / APAD

Prise en main et analyse des clones chinois de l'iPad sous Android

SOMMAIRE

SYSTÈME

PÉRIPHÉRIQUE

HACK

CODE

ABONNEMENT

ÉDITO

C'est incroyable !

Si, si, si. Je vous assure. C'est en allant déjeuner et en marchant tranquillement dans la rue que cette vérité stupéfiante m'est apparue. C'est incroyable !

Vous souvenez-vous du temps où vous rêviez d'avoir « un Linux » dans votre poche ou dans votre sac ? Un vrai système Unix open source que vous pourriez avoir toujours avec vous. Autonome, communicant, réactif… pour lequel vous pourriez développer des applications dans le langage de votre choix.

Eh bien, croyez-le ou non, voilà que ce rêve est devenu une réalité, et ce, dans des proportions plus que sympathiques. Tablettes, *smartphones*, GPS, *Media players/centers* ou téléviseurs sont autant d'exemples bien réels prenant la forme de produits disponibles pour tous via les canaux de diffusion « normaux ». Si cela n'est pas assez, sachez que le monde de l'embarqué pur et dur ne reste pas inactif. Ainsi, des cartes puissantes à base d'ARM9 ou 11 sont maintenant à la portée de toutes les bourses. Il en va de même pour les microcontrôleurs, de plus en plus puissants (AVR, PIC, MSP430), disposant de SDK sous GNU/Linux et d'une communauté sans cesse grandissante.

L'open source dope littéralement la technologie et tire tout un monde vers le haut. Les produits « consumer » se multiplient, les composants et les cartes de développement voient leur coût se réduire et le développement s'ouvre. Tout cela pour qu'au final, le rêve du *geek/nerd* devienne une réalité et que nous puissions tous réutiliser nos connaissances chèrement acquises (en manque de sommeil principalement) à presque toute la technologie qui nous entoure.

On ne s'en rend pas vraiment compte, car cela se met en place et progresse doucement. Mais en prenant un peu de recul et en se replaçant dans le contexte d'il y a seulement 5 ans, cela saute aux yeux : c'est incroyable !

Sans doute moins incroyable et plus inéluctable, une autre nouvelle tient notre rédaction en ébullition depuis quelques mois : la création d'un nouveau magazine appelé *Open Silicium*. Celui-ci s'inscrira dans la même philosophie que *GLMF* et le hors-série que vous tenez présentement dans vos mains. Ce magazine, dont le numéro 1 est à paraître le 24 décembre, explorera, en votre compagnie, le monde de l'embarqué et de l'électronique, sous une optique ouverte, expérimentale et pratique. Préparez vos cross-compilateurs, faites de la place sur vos disques pour les SDK et les sources et faites chauffer les fers à souder ! Open Silicium débarque et compte bien gâcher quelques-unes de vos nuits :)

Denis Bodor

GNU/Linux Magazine est membre de l'APRIL

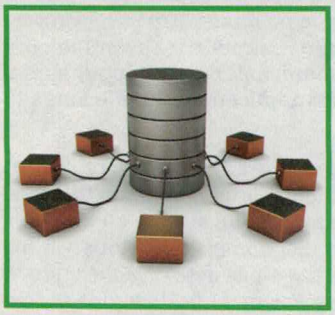
GNU /Linux Magazine France Hors-série
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 67 10 00 20
Fax : 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Secrétaire de rédaction : Véronique Wilhelm
Relecture : Véronique Wilhelm
Conception graphique : Kathrin Troeger

Responsable publicité : Tél. : 03 67 10 00 26
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 :** 1291-78 34
Commission paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 6,50 €

Tokyo et Kyoto Cabinet



Auteur

- Éric Dumas

Bien nombreux sont ceux qui utilisent des bases de données dans leurs applications, mais ne souhaitent pas mettre en place une vraie base de données telle que MySQL, PostGreSQL, etc. Les raisons sont souvent des questions de simplification de la configuration, d'installation, ainsi que le fait que l'on n'a pas besoin de toutes les fonctionnalités d'une vraie base de données (juste un stockage clés/valeurs, par exemple, et pas de base relationnelle). Une autre raison est pour le développement d'applications embarquées qui ont des ressources limitées, et donc, faire tourner un système de base de données est tout simplement impossible.

1 UN PEU D'HISTOIRE...

Historiquement, la plupart des développeurs qui ont eu un besoin de stockage de données ont soit développé leur propre format, soit se sont basés sur BerkleyDb ou Sqlite3. Le principe est assez commun pour toutes ces bases de données : une interface en C (avec souvent des API pour Python, Perl, etc.), un fichier de stockage unique, des performances assez rapides et multiplates-formes. Idéal, par exemple, pour maintenir un cache ou des tables de hachage sur le disque.

ORACLE BERKELEY DB 11g

BerkeleyDb : il s'agit un peu de la base de données historique. Initialement venant de l'université Berkeley, ses auteurs ont fondé la société SleepyCat (1996). Cette société a été rachetée par Oracle en 2006. Cette base s'est concentrée sur le stockage clés/valeurs, ce qui fait des anciennes versions une bonne solution pour de l'embarqué simple (pas de SQL). La dernière version d'OracleBerkelyDb a ajouté le support SQL en se basant sur SqLite.

SQLite

SqLite3 : depuis sa première version en 2000, cette base de données est devenue la base de données favorite de nombreuses applications (Firefox, Skype, etc.) ainsi que pour les applications embarquées. Le fait qu'elle n'utilise que quelques centaines de Ko a fait de cette bibliothèque un choix judicieux pour de très nombreuses applications, et de plus, elle fonctionne très bien en embarqué : petite, rapide, efficace sur des périphériques mobiles. À noter que SQLite permet de créer un vrai schéma de base avec des requêtes SQL. SqLite3 devrait être un choix pour toute application qui a besoin de transactions SQL. À noter que ses performances sont acceptables tant que la taille des données à stocker est raisonnable. Ses limites sont assez bien expliquées dans l'article http://www.sqlite.org/limits.html.

Ces bases de données ont toutes été créées à une époque où les besoins en accès concurrents étaient assez faibles... Leur principale limitation se trouve dès que le volume de données et/ou les accès concurrents sont importants.

2 UNE NOUVELLE GÉNÉRATION VENUE DU PAYS DU SOLEIL LEVANT...

En 2007, Mikio Hirabayashi a publié Tokyo Cabinet, une bibliothèque entièrement écrite en C (un peu moins de 500 K sur un linux 64bits), dont le but est d'offrir une base de données simple (clés/valeurs, pas de SQL), mais très performante, incluant dès sa conception le support des processus légers (*threads*).

L'auteur n'est autre que l'auteur de QDBM, publiée en 2000, qui a posé les jalons de l'amélioration de nos bibliothèques historiques.

D'un point de vue de l'extérieur, TokyoCabinet ressemble à ses prédécesseurs : base de données sous la forme de clés/valeurs (table de hachage, arbre B+, tableau de taille fixe, etc.).

Il n'y a pas de schéma de base. Le principe de base de stocker les données dans un fichier unique n'a pas changé, par contre, les performances, c'est une autre histoire...

Les résultats publiés montrent que les opérations de lecture sont 10 fois plus rapides - l'écriture étant encore plus rapide (x15/x20) - http://1978th.net/tokyocabinet/benchmark.pdf.

Créer des bases de données de grande taille ne pose pas de problème (échelle de plusieurs millions), et les performances de la recherche sont excellentes. Cette bibliothèque est utilisée par des applications haut niveau telles que Mutt (gestion du cache), ou plus bas niveau, telles que le système de fichiers lessfs (système de fichiers avec déduplication). À noter que l'on peut effectuer des requêtes sur le contenu de la base et qu'il existe certaines options d'optimisation telles que la compression à la volée, gestion de bases de grande taille, etc.

Pour être complet, en 2009, le même auteur a publié une nouvelle mouture, KyotoCabinet. Cette version est toujours « jeune » (la 1ère version officielle est arrivée en mai 2010) et il s'agit d'une réécriture en C++ de Tokyo. Les performances sont à peu près les mêmes (même si normalement, des machines avec de nombreux cœurs devraient fonctionner plus vite avec Kyoto). À noter qu'il s'agit d'une bibliothèque en C++ et que l'utilisation de certaines fonctionnalités avancées peuvent poser problème. La taille de la bibliothèque est bien plus importante, donc faites quand même attention...

Voici le tableau de performances qui est extrait du site (voir référence en fin d'article). Les temps exprimés concernent un million d'enregistrements... et les facteurs d'améliorations sont significatifs - cela peut se comparer à l'utilisation d'un cache mémoire tel que memcache.

Name	Description	Write Time	Read Time	File Size
TC	Tokyo Cabinet 1.3.5	0.402	0.334	42,583,208
QDBM	Quick Database Manager 1.8.77	2.779	0.962	56,582,932
NDBM	New Databese Manager 5.1	5.118	3.551	834,003,968
SDBM	Substitute Database Manager 1.0.2	6.277	0.001	621,281,280
GDBM	GNU Database Manager 1.9.3	18.692	3.133	88,137,728
TDB	Trivial Databse 1.0.6	7.219	0.789	52,523,008
CDB	Tiny Constant Database 0.75	0.357	0.371	40,002,048
BDB	Berkeley DB 4.621	9.108	3.109	41,938,944
TC-BT-ASC	B+ tree API of TC (a sending order)	0.707	0.601	32,340,739
TC-BT-RND	B+ tree API of TC (at random)	2.896	2.263	12,532,397
QDBM-BT-ASC	B+ tree API of QDBM (a sending order)	1.404	0.951	40,620,715
QDBM-BT-RND	B+ tree API of QDBM (at random)	6.311	3.816	15,731,675
DBD-BT-ASC	B+ tree API of BDB (a sending order)	1.354	1.451	57,999,360
DBD-BT-RND	B+ tree API of BDB (at random)	3.751	2.415	29,818,880
TC-FIXED	Fixed length API of TC	0.248	0.037	9,002,452

Unit of time is seconds. Unit of size is bytes.

3 UN PEU DE PRATIQUE...

Les sources de la base peuvent être récupérées via http://1978th.net/tokyocabinet/. Tokyo permet de créer plusieurs types d'organisation des enregistrements (le type est à choisir en fonction de ce que l'on veut faire et donc en fonction des performances choisies). La liste ci-dessous est triée par la rapidité des performances :

- tableau de taille fixe : les enregistrements sont effectués via des numéros d'index ;
- table de Hachage : chaque clé doit être unique ;
- arbre B+ : il peut y avoir des clés identiques.

Les API sont fournies en C, Perl, Ruby, Java et Lua (http://fr.wikipedia.org/wiki/Lua). L'exemple qui suit est en C.

Il effectue ces différentes opérations :

1. Il crée une base de données.
2. Il crée un processus léger, qui va de façon assez peu intéressante ajouter des données.
3. En même temps, il crée une grande quantité de processus légers qui vont effectuer des requêtes dans la base et compter le nombre d'enregistrements qui sont retournés.

4. Le résultat est donc différent au fil du temps...

```
#include <tcutil.h>
#include <tctdb.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <pthread.h>

// Usual error and usage display.
static void tkdbError( TCTDB *myDb)
{
  int ecode;
  ecode = tctdbecode(myDb);
  fprintf(stderr, "Error detected: %s\n", tctdberrmsg(ecode));
}

static void usage( const char *s) {
  printf("%s nbTreads\n",s);
  exit(1);
}

// Dumps data into the database.
void *DataWriter(void *myDb)
{
  TCTDB * tdb = (TCTDB*)myDb;
  int64_t dbUid;
  TCMAP * dbColumn;
  int     n = 0;

  /* Consider a db of the form EmailAddress, Name */
  while (n<32000) {
    char name[256];
    sprintf( name , "name%d", n);
    char email[256];
    sprintf( email , "name%d@toto.com", n++);

    dbUid = (long)tctdbgenuid(tdb);
    dbColumn = tcmapnew3("email" , email, "name", name, NULL);

    // Store it
    if (!tctdbput(tdb, (void*)&dbUid, sizeof(dbUid), dbColumn)){
      tkdbError( tdb );
    }
    tcmapdel(dbColumn);
  }

  pthread_exit(NULL);
  return NULL;
}

void *DataReader(void *myDb)
{
  TCTDB * tdb = (TCTDB*)myDb;

  /* Build a query */
  TDBQRY *qry = tctdbqrynew(tdb);
  tctdbqryaddcond(qry, "name", TDBQCSTRINC, "name");
  TCLIST *res = tctdbqrysearch(qry);

  printf("[%d] %d elements\n", (int)pthread_self(), tclistnum(res));

  tclistdel(res);
  tctdbqrydel(qry);

  pthread_exit(NULL);

  return NULL;
}

int main(int argc, char **argv){
  TCTDB *tdb;

  if (argc != 2) {
    usage(argv[0]);
  }
```

```
  /* create the object */
  tdb = tctdbnew();

  /* open the database */
  if(!tctdbopen(tdb, "LinuxMagDb.tct", TDBOWRITER | TDBOCREAT | TDBOLCKNB )){
    tkdbError(tdb);
  }

   // Spawn a creator thread.
   pthread_t thread_creator;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   int ret;
   ret = pthread_create(&thread_creator, &attr, DataWriter, (void *)tdb);
   if (ret){
        printf("ERROR: cannot create a thread (%d)\n", ret);
        exit(-1);
   }

   // And the usual readers.
   int READER_THREADS = atoi(argv[1]);
   pthread_t threads_readers[READER_THREADS];
   int thread_num;
   for(thread_num = 0; thread_num < READER_THREADS; ++thread_num) {
      ret = pthread_create(&threads_readers[thread_num], &attr,
        DataReader, (void *)tdb);
      if (ret){
        printf("ERROR: cannot create a thread (%d)\n", ret);
        exit(-1);
      }
   }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   void *status;
   pthread_join(thread_creator, &status);
   for(thread_num=0; thread_num< READER_THREADS; ++thread_num) {
      ret = pthread_join(threads_readers[thread_num], &status);
      if (ret){
        printf("ERROR: cannot join back a thread (%d)\n", ret);
        }
   }

   pthread_exit(NULL);

  /* close the database */
  if(!tctdbclose(tdb)){
    tkdbError(tdb);
  }

  /* delete the object */
  tctdbdel(tdb);

  return 0;
}
```

On compile ? Un Makefile assez standard...

```
CFLAGS=-O2 -Wall
LDFLAGS=-ltokyocabinet -lpthread
CC=gcc

SRCS=tkcabinetsample.c
OBJS=$(SRCS:.c=.o)
EXECUTABLE=TokyoRun

$(EXECUTABLE) : $(OBJS)
  $(CC) $(OBJS) -o $(EXECUTABLE) $(LDFLAGS)

all: $(EXECUTABLE)

clean:
  rm -rf ${OBJS} ${EXECUTABLE}
```

L'exécution du programme donne le résultat suivant :

```
[eric@gandalf]$ gmake
gcc -O2 -Wall    -c -o tkcabinetsample.o tkcabinetsample.c
gcc tkcabinetsample.o -o TokyoRun -ltokyocabinet -lpthread
[eric@gandalf]$ ./TokyoRun 15
[-1983600880] 2 elements
[-2080377072] 5 elements
[-1973111024] 2 elements
[-1994090736] 3 elements
[-2090866928] 12 elements
[-2101356784] 13 elements
[-2111846640] 17 elements
[1677719312] 19 elements
[1667229456] 20 elements
[1646249744] 25 elements
[1635759888] 35 elements
[-2132826352] 49 elements
[1656739600] 1666 elements
[1625270032] 2491 elements
[-2122336496] 4520 elements
```

Si vous relancez le programme, vous pouvez, de plus, voir l'effet sur les performances ainsi que la taille de la base.

CONCLUSION

Au moment où le système de stockage clé/valeur prend tout son sens (sur disque ou en mémoire), cette avancée en termes de gestion de bases devient particulièrement intéressante, surtout pour des applications qui, par exemple, ont besoin de charger de gros dictionnaires ou des caches.

Une autre utilisation avancée est également pour le traitement de gros volumes de données souvent stockés en base : passer par une telle structure peu permettre des avancées en termes de performance non négligeables, quitte à laisser en base uniquement les données sur lesquelles des requêtes sont nécessaires.

Tokyo permet, pour ceux qui n'ont pas besoin de base relationnelle, d'utiliser une base puissante et rapide tout en pouvant effectuer des requêtes assez avancées. ■

Auteur : Eric Dumas

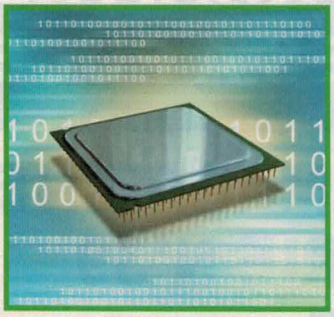
Références

- Sqlite : http://www.sqlite.org/
- BerkeleyDb : http://www.oracle.com/technology/products/berkeley-db/index.html
- QDBM : http://qdbm.sourceforge.net/

Performances :

- http://1978th.net/tokyocabinet/benchmark.pdf
- http://www.mysqlperformanceblog.com/2009/10/19/mysql_memcached_tyrant_part3/
- http://75.134.27.61:8101/wordpress/tokyo-cabinet-is-31-times-faster-than-mysql/
- TokyoCabinet : http://1978th.net/tokyocabinet/
- KyotoCabinet : http://1978th.net/kyotocabinet/kyotoproducts.pdf

La Scratchbox 1.x



La Scratchbox est un environnement de build particulièrement adapté au développement d'applications pour du Linux embarqué. Cet environnement a été développé sous licence GPL par Lauri Leukkunen, principalement pour le projet Maemo de Nokia, la distribution Linux des plates-formes ARM N770 et suivantes. Une version 2 de la Scratchbox, sensiblement différente, est disponible, mais l'auteur ne la maintient pas, et elle ne sera plus évoquée dans cet article. La version 1.x est toujours maintenue par Lauri Leukkunen.

Auteur

- Alain Basty

Les points forts de la *Scratchbox* sont les suivants :

- Possibilité de créer rapidement des cibles pour différents processeurs, principalement x86 et ARM, ainsi que de choisir différentes versions de la libc, grâce aux toolchains déjà disponibles dans les paquets Scratchbox.
- La cross-compilation se fait dans un environnement « chrooté », dans lequel on dispose des mêmes versions des bibliothèques que celles de notre cible. Il n'y aucune ambiguïté possible entre les fichiers de la cible et ceux de la machine hôte.
- On peut exécuter directement dans la Scratchbox des programmes destinés à la cible grâce à une fonctionnalité, la « CPU Transparency », qui s'appuie soit sur un *remote shell*, on doit alors disposer physiquement d'une carte de développement reliée au PC, soit dans QEMU.

Cet article montrera comment installer la dernière version de la Scratchbox 1.x, Hathor, comment compiler et exécuter un premier « Hello World! » sur des cibles x86 puis ARM. Nous passerons ensuite à un projet plus conséquent : la compilation pour une cible ARM de la célèbre Busybox, bien connue dans les environnements embarqués.

1 INSTALLATION

Il faut télécharger et installer les derniers paquets Hathor disponibles ici : http://www.scratchbox.org/download/scratchbox-hathor/. Si votre distribution est basée sur Debian, vous pouvez ajouter une source APT mentionnée sur le site, sinon vous devrez télécharger les « Binary tarballs » disponibles aussi sur le site. La Scratchbox va s'installer dans le répertoire `/scratchbox`. Attention, il semble que l'installation pose problème sur Ubuntu 10.10 et il est nécessaire d'appliquer un fix décrit ici : http://talk.maemo.org/showthread.php?t=51550, avant d'installer les paquets.

Les deux premiers paquets qui nous intéressent sont `scratchbox-libs_1.0.20_i386` et `scratchbox-core_1.0.20_i386`. Si vous êtes sur une distribution Debian, `apt-get install scratchbox-libs scratchbox-core` devrait suffire. Sinon, il faut télécharger les `tgz` correspondants et les décompresser en `root` avec `tar zxf <package> -C /` puis exécuter le script d'installation : `/scratchbox/run_me_first.sh`.

On utilise la Scratchbox un peu comme si on se connectait sur une machine physique : on obtient un shell et on est placé dans son répertoire personnel. Il faut donc que la Scratchbox ait connaissance des utilisateurs et on doit créer des comptes pour chacun d'entre eux. Nous allons créer un premier compte dans la Scratchbox pour l'utilisateur « alain » en `root` : `/scratchbox/sbin/sbox_adduser alain`. Attention, cet utilisateur doit réellement exister sur la machine hôte, le mieux étant donc d'utiliser votre compte utilisateur standard à la place de « alain ». Une fois l'utilisateur créé, on doit se reconnecter à la machine hôte avec cet utilisateur, ou alors exécuter un `su` sur cet utilisateur pour être sûr d'appartenir au groupe `sbox`. La commande `groups` peut être utilisée pour vérifier qu'un utilisateur est bien créé dans la Scratchbox et dispose des droits nécessaires : le groupe `sbox` doit apparaître dans la liste :

```
alain@alain-desktop:~$ groups
alain adm dialout cdrom audio plugdev
lpadmin admin sambashare scanner sbox
```

On peut maintenant se connecter à la Scratchbox avec la commande `scratchbox` ou `scratchbox/login`.

```
alain@alain-desktop:~$ /scratchbox/login

Welcome to Scratchbox, the cross-compilation toolkit!

Use 'sb-menu' to change your compilation target.
See /scratchbox/doc/ for documentation.

[sbox: ~] >
```

2 HELLO WORLD!

Si l'on veut compiler le moindre petit programme, il est nécessaire d'installer au moins une *toolchain* dans la Scratchbox. La première toolchain que nous allons utiliser est la « host-gcc ». Ce n'est pas une véritable toolchain, mais plutôt des liens vers la toolchain naturelle du système hôte. Il faut donc que **gcc** et **g++** soient déjà installés sur le système hôte.

Comme pour les paquets **core** et **libs**, téléchargez et installez le paquet correspondant à **scratchbox-toolchain-host-gcc_1.0.20_i386.*** selon la méthode adaptée à votre distribution.

Nous allons maintenant créer une « target » dans la Scratchbox. Une target est une sorte de système de fichiers qui contient les bibliothèques, fichiers d'include, autres binaires et fichiers de données qui vont nous permettre de compiler, linker, exécuter notre projet. Une *target* est associée à une toolchain et à un environnement d'exécution, directement sur la machine hôte, par réseau sur le hardware relié au PC ou émulé avec QEMU.

Une fois logué dans la Scratchbox, on exécute **sb-menu** et on choisit l'option ***Setup a target*** puis ***Create a new target***, on entre le nom de la nouvelle target, par exemple « host », on choisit la toolchain, dans notre cas « host-gcc », on passe l'écran des « Devkits » en choisissant « DONE », puis « none » pour la « CPU Transparency » et on répond « No » à la question « Do you wish to install files into target? » et « Yes » à « Do you wish to select the target? ».

Et voilà, notre environnement est mis en place, nous verrons par la suite qu'il est aussi facile de créer une target ARM.

Une rapide coup d'œil à notre environnement nous montre la toolchain sélectionnée, un système de fichiers avec les répertoires standards, bien que tous pratiquement vides : aucune bibliothèque installée, pas un utilitaire dans **/bin**, aucun fichier dans notre répertoire personnel.

```
[sbox-host: ~] > sb-conf show
Compiler: host-gcc
Devkits: none
CPU-transparency: none

[sbox-host: ~] > cc --version
host-cc (GCC) 3.3.5 (Debian 1:3.3.5-13)
Copyright (C) 2003 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[sbox-host: ~] > ls /
bin cdrom etc home initrd linuxrc media opt root scratchbox srv targets usr
boot dev floppy host_usr lib login_target.sh mnt proc sbin selinux sys tmp var
```

```
[sbox-host: ~] > ls /bin /etc /home /lib
/bin:
/etc:
/home:
alain
/lib:
```

La Scratchbox fait de la « magie » avec **chroot** et une bonne dose de liens symboliques. Dans la pratique, on se retrouve dans un système de fichiers vide où tous les outils de build restent cependant accessibles. Chaque target a son propre système de fichiers. En effet, les bibliothèques et autres fichiers d'include ou binaires que l'on installe dans une target x86, par exemple, ne coexisteront pas avec leur version ARM. Changer de target, c'est comme se connecter sur une autre machine. L'avantage de cette approche est que les composants que l'on installe dans une target se trouvent donc à leur place « naturelle » dans le système de fichiers, les fichiers d'include dans **/usr/include**, les bibliothèques dans **/lib** et **/usr/lib**, les utilitaires de base dans **/bin**, etc. Cela peut éviter de gros problèmes de compilation pour certains composants dont le système de build n'a pas été bien pensé pour de la cross-compilation et qui rechignent à aller chercher les bibliothèques et includes ailleurs que dans les répertoires standards.

Le répertoire personnel de l'utilisateur de Scratchbox est un cas particulier : il est partagé entre toutes les targets de cet utilisateur. C'est ici que nous allons placer les sources de nos projets, qui seront donc accessibles par toutes nos targets.

Allons-y, dans le shell de la Scratchbox, copions le programme exemple :

```
cd ; mkdir -p src/hello ; cp /scratchbox/packages/hello.c src/hello
cd src/hello
```

On va créer un petit **Makefile** qui va nous permettre de bien séparer les binaires générés pour les différentes targets dont nous nous servirons dans cet article :

```
[sbox-host: ~/src/hello] > cat >Makefile
OBJSDIR := objs/$(shell sb-conf current)

$(OBJSDIR)/hello : hello.c
  mkdir -p $(OBJSDIR) 2>/dev/null
  $(CC) -o $@ $<

clean:
  -rm -rf $(OBJSDIR)
```

La macro **$(shell sb-conf current)** permet de retrouver le nom de la target courante, pour l'instant « host ». Un petit **make** et notre premier programme est disponible dans **objs/host/hello**.

```
[sbox-host: ~/src/hello] > file objs/host/hello
objs/host/hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.0, dynamically linked (uses shared libs), not stripped
```

Pas de mystère ici, son exécution remplit bien le contrat !

3 PREMIERS PAS SUR ARM

La première des choses à faire est de choisir et installer une toolchain ARM. Sur le site de téléchargement de la Scratchbox, plusieurs toolchains sont disponibles. Les toolchains ARM utilisent les versions « lite » du compilateur CodeSourcery, c'est ce que signifie la partie « cs20YY-qQ-NN » dans le nom des paquets. On peut aussi remarquer la version de la libc associée : **glibc** ou **eglibc** déclinées en différentes versions.

Selon le projet sur lequel vous travaillez, il vous faudra choisir la bonne toolchain, voire en créer une nouvelle basée sur les versions des outils imposés pour votre développement. Le site de la Scratchbox propose de la documentation sur la création de sa propre toolchain.

Dans cet article, nous utiliserons la toolchain **scratchbox-toolchain-arm-linux-cs2009q3-67** disponible dans le dépôt Debian ainsi qu'en *tarball*. Une fois téléchargée et installée, on peut créer une nouvelle target, ARM cette fois-ci.

Une fois logué dans la Scratchbox, on exécute **sb-menu** et on choisit l'option ***Setup a target*** puis ***Create a new target***, on entre le nom de la target, par exemple « arm », on choisit la toolchain, dans notre cas « arm-linux-cs2009q3-67 », on passe l'écran des « Devkits » en choisissant « DONE », puis « none » pour la « CPU Transparency » et on répond « No » à « Do you wish to extract a rootstrap on the target? », « No » à la question « Do you wish to install files into target? », et « Yes » à « Do you wish to select the target? ».

Dans cette nouvelle target ARM, le système de fichiers est vierge, mis à par notre répertoire personnel, dans lequel nous avons nos sources. Compilons et vérifions le programme généré :

```
[sbox-arm: ~] > cd src/hello ; make
mkdir -p objs/arm 2>/dev/null
cc -o objs/arm/hello hello.c
[sbox-arm: ~/src/hello] > file objs/arm/hello
objs/arm/hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for
GNU/Linux 2.6.16, dynamically linked (uses shared libs), not stripped
[sbox-arm: ~/src/hello] > objs/arm/hello
/scratchbox/tools/bin/misc_runner: SBOX_CPUTRANSPARENCY_METHOD not set
```

Nous avons bien créé un exécutable ARM. Par contre, nous ne pouvons pas l'exécuter et avons une erreur **SBOX_CPUTRANSPARENCY_METHOD** *not set*.

Nous allons donc installer le paquet QEMU qui va nous permettre d'émuler l'exécution de code ARM sur notre machine hôte x86. Il faut télécharger et installer le paquet **scratchbox-devkit-qemu_0.12.50-0sb3_i386.*** par la méthode habituelle. Nous devons maintenant modifier notre target ARM : lancer **sb-menu**, puis sélectionner la target « arm », répondre aux questions comme lors de la création de la target sauf pour « Select devkits », où l'on ajoutera « qemu» avant de faire « DONE » et pour « Select CPU-transparency method », où l'on choisira « qemu-arm-sb ».

Si on essaie d'exécuter notre programme, on a maintenant une erreur bien plus parlante :

```
[sbox-arm: ~] > src/hello/objs/arm/hello
/lib/ld-linux.so.3: No such file or directory
```

Notre programme se lance, mais bien sûr, il lui manque des bibliothèques essentielles pour s'exécuter puisque le système de fichiers de notre target ARM est totalement vide. Installons donc les fichiers nécessaires en lançant encore une fois **sb-menu** et en sélectionnant « Install files to a target ». Après avoir choisi la target ARM, un écran apparaît avec un certain nombre de paquets pour la target. La sélection par défaut est bonne et il suffit d'appuyer sur ***OK***. On remarque au passage un paquet pour la libc et des fichiers par défaut pour le répertoire **/etc**. La Scratchbox va installer la libc correspondant à la toolchain sélectionnée, ainsi que des fichiers nécessaires à son fonctionnement, comme **/etc/passwd** ou **/etc/groups**. Un petit coup d'œil dans les répertoires **/lib** et **/etc** nous rassure, nous pouvons donc réessayer notre premier programme ARM.

```
[sbox-arm: ~] > src/hello/objs/arm/hello
hello world
```

Excellent, ça marche !

Un petit mot sur l'utilisation de QEMU dans le cas bien précis de la Scratchbox : seul le code ARM du programme et des bibliothèques est émulé. Une fois que l'on passe en mode Kernel, c'est-à-dire pour tout appel système, c'est le code x86 du noyau de l'hôte qui s'exécute. Indépendamment de la Scratchbox, QEMU est plutôt utilisé de manière très différente pour émuler un hardware complet et booter/exécuter un OS à part entière.

L'utilisation de QEMU dans la Scratchbox pose quelques contraintes :

- Vu que l'émulation QEMU utilise le noyau du système hôte, cette méthode d'exécution n'est pas applicable pour du code très dépendant du noyau, car il y a peu de chance que la cible et l'hôte partagent la même version de noyau.
- On ne peut pas utiliser gdb pour déboguer un programme ARM dans QEMU, évidemment il est toujours possible de le faire dans une target x86.

4 LA BUSYBOX

Il est tant de peupler le **/bin** de notre target ARM avec les commandes de base Linux.

La Busybox, bien connue de tous les aficionados de l'embarqué, regroupe en un seul programme, **/bin/busybox**, un ensemble d'utilitaires de base Linux, comme le code de **ls**, **ifconfig**, **gzip** et bien d'autres. La compacité et l'optimisation de son code font de la Busybox un choix très adapté pour les environnements hardware embarqués, où l'utilisation raisonnée des ressources physiques reste un élément fondateur de la conception des systèmes.

La configuration de la Busybox est assez similaire à celle d'un noyau Linux : un **make config** qui pose des questions pour inclure ou exclure des fonctionnalités, un **make**, et c'est parti. Évidemment, vue la diversité des options disponibles, le système de build de la Busybox offre aussi une interface graphico-textuelle bien plus adaptée pour l'utilisateur : **make menuconfig**, ça vous rappelle quelque chose ? Bien, mais cette interface utilisateur utilise la non moins célèbre bibliothèque **ncurses**, qui n'est pas installée dans notre target ARM ! Impossible donc de l'utiliser pour l'instant.

Nous allons d'abord devoir installer **ncurses**. Assez facile : on la récupère sur le site GNU, on la compile et l'installe. Je vous passe les détails du fichier **INSTALL** de **ncurses** et voilà :

```
cd ~/src
wget http://ftp.gnu.org/pub/gnu/ncurses/ncurses-5.7.tar.gz
tar zxvf ncurses-5.7.tar.gz
cd ncurses-5.7
./configure --with-shared --without-normal --without-debug
make
file ./test/ncurses
./test/ncurses
make install.progs
make install.includes
make install.libs
make install.data
```

Joie ! La version ARM de **ncurses** est installée dans notre target ARM, le programme de test est exécutable et fonctionnel grâce à QEMU !

Bien, la Busybox maintenant. Elle existe en plusieurs versions et pour cause : de nombreux utilitaires Linux utilisent les *headers* du noyau et là encore, un choix est nécessaire pour prendre la version qui correspond à votre toolchain et votre noyau. Dans cet article, j'ai choisi une version qui correspond à la toolchain ARM téléchargée. Simplement :

```
cd ~/src
wget http://www.busybox.net/downloads/busybox-1.13.4.tar.bz2
tar jxvf busybox-1.13.4.tar.bz2
cd busybox-1.13.4
make HOSTCC=host-gcc HOSTCXX=host-g++ menuconfig
make CONFIG_PREFIX=/ install
file /bin/busybox
```

Bravo, Busybox compilée, installée, un petit système Linux ARM est prêt à l'emploi. Il est facile de cross-compiler dans la Scratchbox.

Certains composants, basés en majorité sur les *Autotools*, doivent générer un programme et l'exécuter avant de passer à la compilation du composant lui-même, ceci afin de vérifier la présence d'une fonctionnalité sur le système cible. Que se passe-t-il dans une Scratchbox avec target ARM ? Le programme de test est compilé (en ARM donc), et grâce à l'émulation QEMU, il s'exécute sans erreur dans l'environnement cible émulé. Les Autotools sont contents et la compilation a lieu comme un charme.

5 QUELQUES ASTUCES

Quelques petits trucs pour ceux qui veulent aller plus loin :

- Il n'est pas possible depuis un shell Scratchbox d'utiliser un éditeur de programmes « moderne » (à part **vi** bien sûr), mais il est facile d'accéder aux fichiers de votre compte Scratchbox depuis le compte sur le système hôte : essayez, dans un shell hôte, **gedit /scratchbox/users/$USER/home/$USER/src/hello/hello.c**.
- Il est utile de créer un lien symbolique entre le répertoire des sources de l'utilisateur Scratchbox et celui du compte lui correspondant sur la machine hôte : **cd ~ ; ln -sf /scratchbox/users/$USER/home/$USER/src**. Cela permet d'utiliser les mêmes noms de fichiers sources à l'intérieur et à l'extérieur de la Scratchbox, ce qui est bien pratique pour l'édition des fichiers, mais aussi plus tard pour le débogage symbolique.

CONCLUSION

Il reste énormément de choses à dire sur la Scratchbox, notamment comment l'utiliser avec un *hardware* relié au PC, comment déboguer, installer des binaires ARM pré-compilés des dépôts Debian, comment installer des bibliothèques graphiques comme GTK, utiliser des *rootstraps*...

Je vous laisse découvrir tout cela sur le site officiel. ■

Auteur : Alain Basty

Références

- Le site officiel de la Scratchbox : http://www.scratchbox.org/
- Le développement Maemo : http://maemo.org/development/
- Le fix pour l'installation de la Scratchbox sur Ubuntu 10.10 : http://talk.maemo.org/showthread.php?t=51550
- CodeSourcery : http://www.codesourcery.com/sgpp/lite_edition.html
- EGLIBC : http://www.eglibc.org/home
- Busybox : http://www.busybox.net/
- *Free Software Directory/Ncurses* : http://directory.fsf.org/project/ncurses/
- *Autotools* : http://fr.wikipedia.org/wiki/Autotools

L'ePad/aPad, un clone d'iPad

Quelques semaines à peine après l'annonce et la mise en vente de la tablette Apple, on a vu apparaître sur les sites d'enchères en ligne plusieurs clones chinois se vantant de fournir quasiment les mêmes fonctionnalités que l'iPad, à un tarif deux voire trois fois moindre. Ne résistant pas à la curiosité d'étudier la chose plus en détail, nous avons acquis deux exemplaires de ces clones pour analyser la bête.

Auteur

- Denis Bodor

Entrons dans le vif du sujet en précisant qu'il ne faut pas vous attendre à un matériel de grande qualité en achetant ce genre de produits. En effet, bien que d'une finition tout à fait correcte, l'ePad/aPad chinois vaut son prix, mais pas un sous de plus (et encore). Ce n'est pas tant la robustesse qu'il faut mettre en cause, mais plutôt le choix des composants utilisés. L'écran tactile résistif est d'une lenteur affligeante et le processeur peine à fournir la puissance nécessaire au fonctionnement optimal de l'environnement, même avec une configuration revue à la baisse. Cette dissonance, nous la retrouvons également entre le produit lui-même et les intitulés des annonces de vente comme « 10.2'' Google Android 2.1 Tablet Ebook Reader 3D 3G WIFI ». La notion de 3D est très subjective (bien que le SoC d'InfoTM Micro Electronics supporte OpenGL ES) et la 3G est tout simplement une option pour peu que l'on connecte une clé USB 3G/GPRS/GSM.

En revanche, les caractéristiques techniques fonctionnelles sont au rendez-vous si on les compare avec celles de la tablette Apple. Les petites mains chinoises ont assemblé un produit basé sur un processeur ARM11 Zenithink ZT-180 (un iMAPx200) à 1 Ghz disposant de :

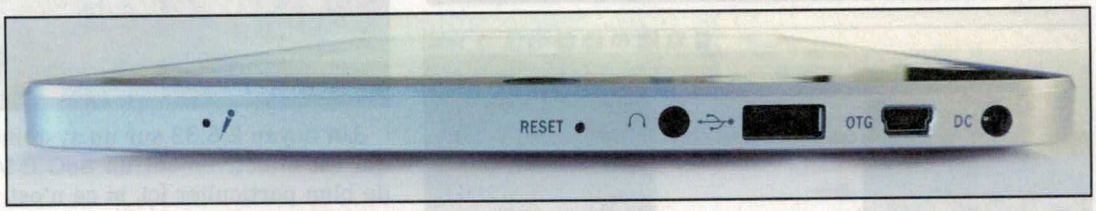
- 256 Mo de mémoire SDRAM ;
- 2 Go de Flash NAND ;
- un écran 10"2 tactile 1024×600 ;
- une interface Ethernet (via un connecteur spécifique avec adaptateur fourni) ;
- une connectivité wifi (un adaptateur Ralink rt3070 USB soudé à la carte d'expansion) ;
- un lecteur de cartes micro SD/MMC jusqu'à 32 Go ;
- un port USB OTG (*On The Go*) ;
- un port USB hôte ;
- un micro intégré ;
- un port audio jack pour haut-parleurs en plus des haut-parleurs intégrés (d'environ 1 cm sur 1,5 cm).

À l'ouverture du colis en provenance directe de Chine, le premier choc provient très certainement du logo utilisé sur le coffret du produit. Non, vous ne rêvez pas, il s'agit bien du logo d'Internet Explorer !! En Chine, il semblerait que la notion de copyright soit très subjective… tout comme celle de publicité mensongère.

Voilà pour ce qui est des spécifications. Celles-ci sont généralement reprises correctement sur les sites de vente, ce qui n'est généralement pas le cas du descriptif logiciel. Il est, en effet, souvent question de « Built-in a dozen of apps (MSN, Skype Google Talk, Facebook, Youtube, Gmail) », de « You can get numerous applications from Android Market » ou encore de « Provide a fantastic multimedia and Internet experience ». Les trois affirmations sont souvent fausses. Aucune application Google n'est installée par défaut. Vous ne pouvez pas avoir accès au *Market* de Google avec le *firmware* fourni de base et le surf est tout aussi pénible que sur n'importe quel *smartphone*.

sous Android 2.1

La tablette est livrée sous Android 2.1 et dès l'allumage (après bricolage du chargeur si celui-ci vous a été livré dans une configuration US ou UK), on se retrouve avec l'environnement Google qui semble fonctionnel. Après un petit tour des applications installées et une configuration du Wifi, ainsi que de quelques autres éléments (clavier, date/heure, locales, etc.), on se rend rapidement compte qu'on est loin de la vélocité du plus petit smartphone du marché.



L'aPad est riche en possibilités de connexion, contrairement à la tablette dont le design a été copié. De gauche à droite, micro, bouton de reset (trombone obligatoire), sortie audio, port USB hôte, USB OTG (ADB) et connecteur d'alimentation. Non visible sur la photo, l'aPad dispose également d'une connectique Ethernet et d'un slot pour carte micro-SDHC.

Évitons le descriptif sur ton de pseudo-test en vous faisant grâce des commentaires comme « la prise en main est rapide, on se sent vite en confiance ». C'est un périphérique Android de type tablette avec tout ce que le système Google fournit classiquement dans sa version de base.

1 SOYONS INTRUSIFS

Le fait de prendre en main une plate-forme Android présente un certain nombre d'avantages. Parmi eux, nous avons la quasi-certitude de pouvoir utiliser nos acquis en la matière et donc nos connaissances de l'utilisation des outils de développement fournis par Google. Une chose qu'il faut savoir à propos des périphériques (tablette et smartphones) Android est le fait qu'il est possible d'obtenir un shell (parfois **root**) via l'utilitaire **adb** livré avec le SDK Google. Ceci s'avère vrai dans 90 % des cas et les rares exceptions sont généralement le fait d'opérateurs de téléphonie mobile soucieux d'empêcher les utilisateurs de trop jouer avec le matériel qu'ils ont pourtant payé rubis sur ongle (si vous ne voyez pas de qui je parle, cherchez entre le rouge et le jaune à la section Samsung et *recovery mode*).

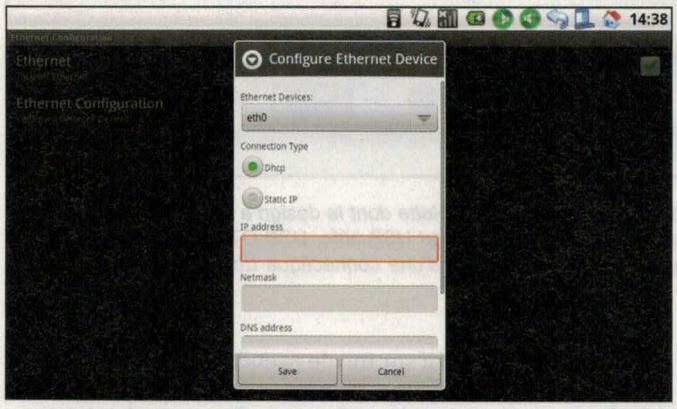
Un petit tour dans les préférences, tout d'abord, pour vous assurer de la bonne prise en charge de l'interface Ethernet. Une icône sur le « bureau » vous permet d'accéder directement aux préférences système. La configuration de l'interface se fait via une entrée située tout en bas de la liste des éléments de configuration. Ceci ressemble à un ajout quelque peu brouillon, mais nous ne sommes pas au bout de nos surprises. Ceci laisse sans doute déjà présager de la conclusion de cet article (et vous avez bien raison). Nous avons ici configuré un serveur DHCP sur le LAN afin que l'ePad se voit attribuer une adresse automatiquement. Fort heureusement pour vous, nous avons commandé deux de ces choses auprès de deux marchands différents via un site d'enchères en ligne. À la connexion du second périphérique, nous trouvons dans les journaux système du serveur DHCP une information relativement inquiétante : plusieurs occurrences de la même adresse MAC, à savoir **00:1b:fc:eb:92:fa**, sous la forme de requêtes provenant de plusieurs interfaces du serveur DHCP ! Après vérification, il s'avère effectivement que les deux appareils possèdent la même adresse. Édifiant ! Pour corriger le problème, il vous faudra (plus tard) utiliser la commande **ip** ainsi :

```
ip link set eth0 address XX:XX:XX:XX:XX:XX
ip link set eth0 broadcast XX:XX:XX:XX:XX:XX
```

Une fois le réseau activé, la tablette est en mesure de se connecter à Internet, et vous, par la même occasion, êtes en mesure d'utiliser cette connectivité pour accéder à Android. Malheureusement, n'espérez pas user d'une commande **ssh** ou **telnet**, car aucun serveur n'est installé. D'ailleurs, aucun port n'est ouvert, ce qui règle la question. Il faut donc passer par le connecteur OTG (USB *On The Go*) via un simple câble USB type A (PC) vers mini-B (tablette), qui n'est pas livré avec l'aPad, mais que vous possédez sans doute déjà (c'est le type classique utilisé pour les disques 2"1/2 externes, par exemple).

Assurez-vous que dans l'interface des préférences, menu ***Applications/Développement***, l'option ***Débogage USB*** soit bien activée. Dès la connexion, un périphérique 0bb4:0c01 apparaît. Il est possible que vous soyez obligé de modifier les règles **udev** afin de faciliter l'accès non **root** au périphérique, selon votre distribution. Ceci n'est que temporaire puisque, justement, le but est d'utiliser l'outil **adb** du SDK Android via le réseau. Pour ce faire, lancez tout d'abord la commande **adb** pour lister les périphériques Android détectés :

```
% sudo ./adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
0123456789ABCDEF        device
```



La configuration du port Ethernet, pourtant peu courant sur les périphériques Android, est parfaitement prise en charge dans l'interface de configuration. Elle propose un paramétrage manuel ou automatique via DHCP. C'est là qu'on aura la suprise de constater que tous les aPad Zenithink semblent utiliser la même adresse MAC !

Dans la continuité du « on a vite fait une tablette Android », vous remarquerez que le numéro du périphérique est assez représentatif. Il est identique avec nos deux exemplaires de l'aPad. **adb** détecte bien notre tablette, nous pouvons donc communiquer avec le serveur de débogage et lui demander d'écouter le réseau (port 5555) plutôt que le port USB OTG :

```
./adb -d tcpip 5555
restarting in TCP mode port: 5555
```

Dès lors, nous pouvons accéder au débogueur via l'adresse IP de la tablette et connecter notre démon **adb** à la machine :

```
./adb connect 192.168.10.174:5555
connected to 192.168.10.174:5555
```

Nous sommes maintenant en mesure de procéder à de nombreuses manipulations, et ce, avec une accessibilité relativement efficace (réseau). **adb** nous permet ainsi d'opérer des changements sur le ou les systèmes de fichiers du périphérique Android, de copier et récupérer des fichiers, mais aussi d'obtenir un shell. Ce dernier point va nous permettre d'en apprendre un peu plus sur la bête :

```
% ./adb shell

# uname -a
Linux localhost 2.6.32.9 #1816 Wed Oct
  13 10:43:19 CST 2010 armv6l GNU/Linux

# cat /proc/cpuinfo
Processor       : ARMv6-compatible processor rev 5 (v6l)
BogoMIPS        : 1005.97
Features        : swp half thumb fastmult vfp edsp java
CPU implementer : 0x41
CPU architecture: 6TEJ
CPU variant     : 0x1
CPU part        : 0xb36
CPU revision    : 5
Hardware        : IMAPX200
Revision        : 0000
Serial          : 0000000000000000

# free
             total         used         free       shared      buffers
  Mem:       188828       181520         7308            0          164
Swap:            0            0            0
Total:       188828       181520         7308
```

Un noyau 2.6.32 sur un système ARM11 (ARMv6 rev5) qui est effectivement un SoC IMAPX200 d'InfoTM : rien de bien particulier ici, si ce n'est que certaines annonces accrocheuses pour ces aPad/ePad parlent sans complexe d'ARM Cortex A8. Notez que matériellement, il ne semble exister qu'un seul modèle chez Zenithink. En revanche, le firmware existe en plusieurs versions, mais nous en reparlerons plus loin dans l'article.

Côté points de montage et systèmes de fichiers, rien d'exceptionnel pour une plate-forme Android :

```
# mount
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
tmpfs /sqlite_stmt_journals tmpfs rw,relatime,size=4096k 0 0
/dev/block/mtdblock0 /system yaffs2 ro,relatime 0 0
/dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev,relatime 0 0
/dev/block/mtdblock2 /cache yaffs2 rw,nosuid,nodev,relatime 0 0
/dev/block/mtdblock3 /nand yaffs2 rw,nosuid,nodev,relatime 0 0
/dev/block//vold/179:1 /sdcard vfat rw,dirsync,nosuid,nodev,
  noexec,relatime,uid=1000,gid=1015, fmask=0702,dmask=0702,
  allow_utime=0020,codepage=cp936,iocharset=cp936,
  shortname=mixed,utf8,errors=remount-ro 0 0

# df
/dev: 94412K total, 12K used, 94400K available (block size 4096)
/sqlite_stmt_journals: 4096K total, 0K used, 4096K available (block size 4096)
/system: 262144K total, 101060K used, 161084K available (block size 4096)
/data: 393216K total, 49884K used, 343332K available (block size 4096)
/cache: 102400K total, 4616K used, 97784K available (block size 4096)
/nand: 1306624K total, 5168K used, 1301456K available (block size 4096)
/sdcard: 7975424K total, 224K used, 7975200K available (block size 32768)

# cat filesystems
[...]
nodev   inotifyfs
nodev   devpts
        ext3
        ext2
        cramfs
nodev   ramfs
        vfat
        msdos
nodev   nfs
        romfs
        yaffs
        yaffs2
```

On remarquera le support de NFS, ce qui pourra être intéressant pour procéder à quelques essais et expérimentations futurs. Autre élément curieux, le point de montage **/nand** constitué d'un système de fichiers YAFFS2 de plus d'1 Go est inutilisé, alors que n'avons que peu d'espace pour les applications (**/system/app**).

Toujours côté espace de stockage, nous en apprenons un peu plus sur la mémoire Flash (NAND comme on peut s'en douter) en consultant la table MTD :

```
# cat mtd
dev:    size   erasesize  name
mtd0: 10000000 00080000 "system"
mtd1: 18000000 00080000 "userdata"
mtd2: 06400000 00080000 "cache"
mtd3: 4fc00000 00080000 "Local-disk"
```

Point de MTD pour le stockage de la configuration U-Boot, on peut donc présager que le reflashage d'un système GNU/Linux pour l'embarqué risque d'être délicat. Revenons aux périphériques avec :

```
# cat /proc/bus/input/devices
I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="gpio-keys"
P: Phys=gpio-keys/input0
S: Sysfs=/devices/platform/gpio-keys/input/input0
U: Uniq=
H: Handlers=kbd event0
B: EV=3
B: KEY=40000000 100040 0 a000000 0

I: Bus=0019 Vendor=0000 Product=0000 Version=0000
N: Name="imapx200_keybd"
P: Phys=
S: Sysfs=/devices/platform/imapx200_keybd/input/input1
U: Uniq=
H: Handlers=kbd event1
B: EV=100003
B: KEY=10000 0 0 0 0 0 0 0 40000 0 40000800 80d680 3f ff3fffff fffffffd

I: Bus=0018 Vendor=0000 Product=0000 Version=0000
N: Name="TSC2007 Touchscreen"
P: Phys=2-0048/input0
S: Sysfs=/devices/virtual/input/input2
U: Uniq=
H: Handlers=mouse0 event2
B: EV=b
B: KEY=400 0 0 0 0 0 0 0 0 0 0
B: ABS=1000003

# cat fb
0 imapfb
1 imapfb

# cat /proc/asound/devices
  0: [ 0]   : control
  1:        : sequencer
16: [ 0- 0]: digital audio playback
24: [ 0- 0]: digital audio capture
33:        : timer
```

GPIO, framebuffer, contrôleur audio, ... tout est pris en charge par l'IMAPX200, qui est un SoC, rappelons-le. Ceci signifie une chose très importante : en dehors du Wifi (qui est une clé USB rt3070 soudée) et de la dalle tactile (contrôlée par un TSC2007 de Ti), les périphériques sont pris en charge par des pilotes Linux spécifiques. Précisons de suite que la possibilité d'accéder aux sources de ces pilotes relève de l'utopie la plus totale. Il ne s'agit pas d'un problème de licence, la difficulté est bien au-delà. Alors qu'Android est une plate-forme open source, tout comme Linux et les outils GNU, la notion de licence, de droits d'auteurs et de *copyright* ne semble avoir qu'une existence purement théorique dans certains pays. Eh oui, comme le précise un utilisateur dans un des nombreux forums consacrés aux *hacks* autour de ces tablettes Android, comment peut-on espérer d'une société qu'elle respecte la GPL alors même qu'elle ne se prive pas d'utiliser le logo de Microsoft Internet Explorer pour le « e » de son ePad ? Ce n'est tout simplement pas le même monde. Oubliez donc ce doux rêve où vous compiliez un noyau pour la tablette ZT-180.

2 UN PEU PLUS DE GOOGLE DANS MON APAD ?

Précisons d'emblée que les indications qui vont suivre sont données à titre purement démonstratif, informatif et explicatif. Il ne s'agit en aucune manière de contrevenir aux politiques et aux choix stratégiques des différents acteurs du marché de la téléphonie mobile et de l'embarqué grand public. Il semble, en effet, que Google, pour l'heure, ne considère pas les tablettes comme étant des périphériques adaptés pour Android. Il en résulte le fait que seuls les smartphones (« with Google » ou non) aient, par exemple, accès au Market place Google contenant quelques 100000 applications gratuites et payantes. L'état des lieux est relativement facile à appréhender dans son ensemble : ce qui fait vendre un smartphone plutôt qu'un autre n'est plus tant ses caractéristiques techniques (elle se valent plus ou moins toutes), mais les applications que l'utilisateur peut installer. Ceci n'est pas nouveau et 3com l'avait déjà compris en son temps pour les PDA Palm. Plus il y a d'applications disponibles pour une plate-forme, plus les utilisateurs sont tentés de se diriger vers cette dernière. Ainsi, l'accès au Market place Google est un élément clé pour un constructeur. Pour qu'un périphérique puisse bénéficier d'un accès, il doit répondre à un certain nombre de caractéristiques dont, par exemple, la présence d'un accéléromètre, d'un magnétomètre, d'un GPS, etc. Ceci explique que la Dell Streak ou encore les tablettes Archos ne disposent pas de l'application Market.

De ce fait, le « jeu » favori des utilisateurs/bidouilleurs est devenu l'installation et l'utilisation forcées de l'application Market. Avec les aPad, de nombreuses techniques existent et, plus suspect, le Market est déjà installé avec l'un des derniers firmwares disponible. Nous sommes là dans un univers qui n'est pas sans rappeler celui des copies illégales de jeux vidéo d'il y a quelque 15 ou 20 ans : forum flashy, pseudos de W4rR10r, téléchargements depuis des sites anonymes pleins de pubs, etc.

La plupart des astuces fournies par ces biais prennent la forme de scripts et de binaires sortis d'on ne sait où, qui sont

mis à disposition principalement pour la plate-forme Windows. En y regardant de plus près, il s'avère souvent qu'il s'agit simplement de l'utilisation tout à fait classique de l'outil principal du SDK Android : **adb**. À titre d'exemple, voici une partie des commandes du script RPM-Script v1.2 permettant de mettre à jour l'aPad de manière à obtenir un système plus complet et plus orienté Google.

La première étape consiste à remonter le système de fichiers **/system** contenant, comme son nom l'indique, le cœur du système Android, normalement accessible en lecture seule :

```
./adb remount
remount succeeded
```

Le script, ensuite, se borne à nettoyer l'ensemble des éléments qui ne présentent pas grand intérêt aux yeux du créateur (anonyme) du script :

- Suppression de SlideME, un Market place alternatif regroupant largement moins d'applications que celui de Google, mais proposant également les applications payantes et gratuites :

```
#About to delete SlideME 2 (Who needs it with working Market)
./adb shell rm /system/app/sam2.apk
```

- Suppression de l'outil de benchmark supposé peu fiable :

```
#About to delete Performance BenchMark (Don't Trust!)
./adb shell rm /system/app/softweg.hw.performance.apk
```

- Suppression du gestionnaire de fichiers au bénéfice d'une installation ultérieure d'Astro :

```
#About to delete OI FileManager (Astro is better)
./adb shell rm /system/app/FileManager.apk
```

- Idem pour la mini suite bureautique par défaut du bénéfice de QuickOffice :

```
#About to delete OfficeSuite Viewer (use QuickOffice GApp)
./adb shell rm /system/app/OfficeSuite.Viewer.apk
```

- Suppression du GoogleMaps inclus par défaut pour réinstallation via le Market place Google :

```
#About to delete included GoogleMaps (Use GApps version)
./adb shell rm /system/app/google_maps_ver4.3.0.apk
```

- Élimination de l'application de chat MSN (en chinois) :

```
#About to delete chinese app "Hi MSN" (Many IM alternatives)
./adb shell rm /system/app/himsn.apk
```

- Idem pour le tueur de processus. Il faut savoir, en effet, qu'Android gère lui-même la fermeture des applications en fonction des processus utilisés et des ressources à disposition pour le système. Ainsi, une application non utilisée ne sera pas nécessairement tuée. Il n'y a pas d'option de fermeture dans les applications (sauf exception). Malheureusement, parfois, il est nécessaire de forcer la fermeture des applications non utilisées afin de récupérer des ressources pour une application plus gourmande ou, tout simplement, rendre le système plus véloce. TasKiller est un outil permettant un contrôle plus fin de l'exécution des tâches. L'auteur du script semble ici préférer AutoKiller :

```
#About to delete chinese version of TasKiller (use AutoKiller)
./adb shell rm /system/app/and_com.tni.TasKillerFull2.4.1
```

- Suppression du *player* Real sans doute installé sans accord par le fabricant de la tablette. L'auteur du script suppose une réinstallation via le Market :

```
#About to delete RealPlayer (install newest from Market)
./adb shell rm /system/app/RealPlayer.apk
```

- L'émulateur de terminal intégré ne permet pas une utilisation avec le clavier virtuel Android, mais uniquement via la connexion d'un clavier USB (point de Bluetooth dans le périphérique). Cette application est donc totalement inutile dans 80 % des cas :

```
#About to delete Terminal (only works with USB keyboard)
./adb shell rm /system/app/Term.apk
```

- Android 2.1 Cupcake intègre le support des fonds d'écran animés via une application spécifique. Les performances de l'aPad ne permettent cependant pas de profiter de ce type de fonctionnalités, sauf pour éventuellement mettre le système à genoux ou vider la batterie très rapidement :

```
#About to delete Live Wallpapers (no work, too slow, batt drain)
./adb shell rm /system/app/VisualizationWallpapers.apk
```

- Le script se poursuit ainsi en éliminant bon nombre d'applicatifs afin de libérer de l'espace sur le système de fichiers **/system** :

```
#About to delete Flip Calendar/Clock Widgets
./adb shell rm /system/app/calendar_clock.apk

#About to delete Real Calc (better stuff in Market)
./adb shell rm /system/app/realcalc.apk

#About to delete NotePad (better stuff in Market)
./adb shell rm /system/app/NotePad.apk
#About to delete AlarmClock (DeskClock is better)
./adb shell rm /system/app/AlarmClock.apk

#About to delete E-Mail (I only use GMail)
./adb shell rm /system/app/Email.apk

#About to delete Music (RealPlayer is better)
./adb shell rm /system/app/Music.apk

#About to delete iReader
./adb shell rm /system/app/ireaderv1.0.8.1_android.apk

#About to delete RSSReader
./adb shell rm /system/app/RSSReader.apk
```

Après avoir fait le ménage, il est temps d'installer quelques applications qui permettront à la tablette chinoise d'être plus efficace et plus complète. Le script utilise alors une copie locale d'un certain nombre de binaires et paquets pour compléter l'installation, toujours avec **adb** :

```
# GoogleAPPS
./adb shell rm /system/app/GoogleSearch.apk
./adb shell rm /system/app/Provision.apk
./adb push system/etc/permissions /system/etc/permissions
./adb push system/framework /system/framework
./adb push system/lib/libgtalk_jni.so /system/lib
./adb push system/lib/libinterstitial.so /system/lib
./adb push system/app /system/app
./adb push system/bin /system/bin
./adb push system/xbin /system/xbin
./adb shell sync
./adb shell chmod 4755 /system/xbin/su
./adb shell ln -s /system/xbin/su /system/bin/su
./adb shell chmod 777 /system/bin/busybox
./adb shell /system/bin/busybox --install -s /system/bin
./adb shell chmod 777 /system/build.prop /system/app/* \
  /system/lib/* /system/framework/* /system/etc/permissions/*
./adb shell chown system.system /system/build.prop \
/system/app/* /system/lib/* /system/framework/* \
/system/etc/permissions/*
./adb shell setprop gsm.sim.operator.numeric 310260
./adb shell setprop gsm.operator.numeric 310260
./adb shell setprop gsm.sim.operator.iso-country us
./adb shell setprop gsm.operator.iso-country us
./adb shell setprop gsm.operator.alpha T-Mobile
./adb shell setprop gsm.sim.operator.alpha T-Mobile
```

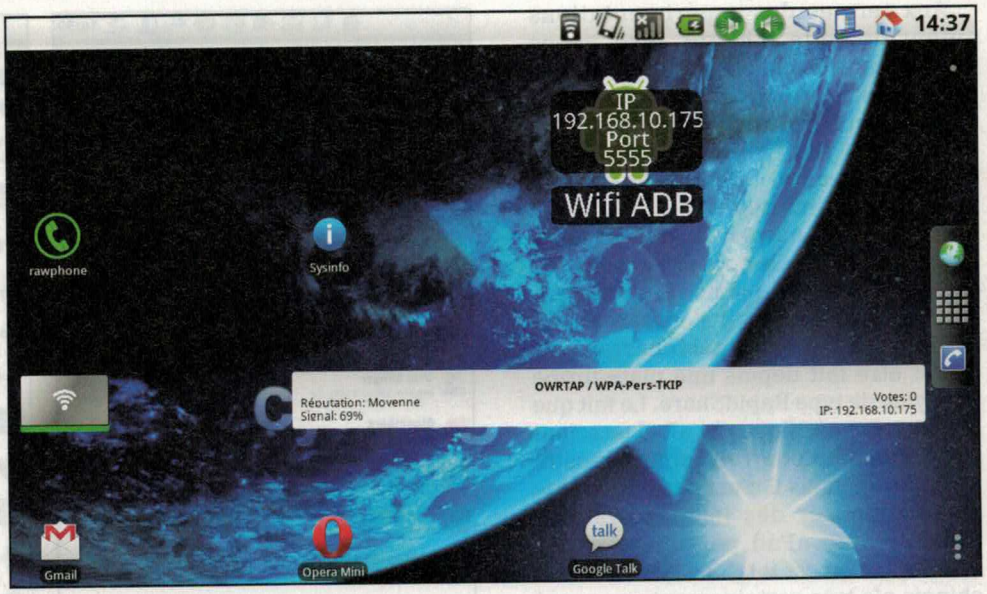
La directive **push** permet d'envoyer des fichiers au système Android. On remarque ici une mise à jour d'éléments critiques comme des bibliothèques, des fichiers de configuration ou encore des propriétés (**[set|get]prop**). Concernant ces dernières, l'auteur du script spécifie divers éléments permettant de rendre compatible la tablette avec le Market place Google. Il s'agit ici de faire croire que le périphérique est un téléphone mobile en spécifiant un opérateur de téléphonie mobile et une localisation aux USA. Il semblerait, en effet, que certaines applications ne soient pas disponibles pour tous les pays et que l'application Market trie et filtre les applications que l'utilisateur peut voir et installer. Cette partie du script est clairement une manière de tromper le Market place pour le forcer à installer les applications. C'est là, cependant, une pratique courante de la part de certains utilisateurs de tablettes, mais également de smartphones. Il existe même une application permettant de procéder à ces changements automatiquement et gérant une collection de profils d'opérateurs.

Le script, ensuite, va vider les données du Dalvik, la machine virtuelle Java, pour forcer une réexécution du processus de configuration initial :

```
./adb shell rm -r /data/dalvik*
./adb reboot
```

Comme le précisent les commentaires dans le script, cette dernière étape de *reboot* ne fonctionne pas. La tablette reste sur le message d'extinction sans passer à l'acte. Il faut alors éteindre le matériel de force en appuyant sur le bouton d'allumage et en le maintenant enfoncé une dizaine de secondes. Au relâchement du bouton, la tablette s'éteint enfin. Une autre solution est d'utiliser un trombone pour procéder à un *reset* via l'orifice situé à côté de la prise audio (ne pas confondre avec les trous situés sur le haut de la tablette permettant l'ouverture du boîtier).

Le redémarrage nécessite quelques précautions. En effet, on assiste à un premier démarrage d'Android (comme ce serait le cas dans l'AVD du SDK) et vous êtes invité à toucher l'androïde pour compléter la procédure. IL NE FAUT SURTOUT PAS LE FAIRE. Le système n'est pas encore complètement démarré et vous devez attendre patiemment 5 à 10 bonnes minutes. Ensuite, touchez les quatre coins du fond noir en commençant par le coin supérieur gauche et dans le sens des aiguilles d'une montre. Ceci aura pour effet de calibrer l'écran. Observez l'animation présentant un doigt touchant l'androïde. Après cette opération, il se placera au centre du personnage, confirmant la calibration. Ce n'est qu'alors que vous pourrez poursuivre le démarrage.



Après reflashage, personnalisation et installation de plusieurs applications, l'aPad devient une plate-forme d'expérimentation intéressante et éventuellement une solution pour jeter un coup d'œil à ses mails. N'en attendez cependant pas plus, tant l'ensemble est d'une lenteur déprimante.

À ce stade, vous ne pourrez pas vous connecter au Market place. En effet, il vous faut un Android ID. Là, beaucoup de flou entoure ce fameux identifiant. Si on s'en tient tout simplement à la seule source officielle (la documentation d'Android), cette valeur est décrite comme : « A 64-bit

number (as a hex string) that is randomly generated on the device's first boot and should remain constant for the lifetime of the device. ». Cet ID permet donc d'identifier une « entité Android ». Bien entendu, les forums cités précédemment sont peuplés d'utilisateurs adorant les raccourcis, les conclusions hâtives et la vulgarisation extrême au détriment de l'exactitude technique. On y parle ainsi de l'Android ID comme étant le IMSI (*International Mobile Subscriber Identity* ou numéro unique international de souscripteur/client mobile) provenant de la carte SIM ou du IMEI (*International Mobile Equipment Identity* ou identifiant unique de l'équipement mobile) du téléphone mobile. L'emplacement de stockage de l'ID en question est également très variable selon les pseudo-documentations, mais dans tous les cas, il s'agit d'un fichier au format SQLite placé dans `/data/data/com.google.android.googleapps/databases/`.

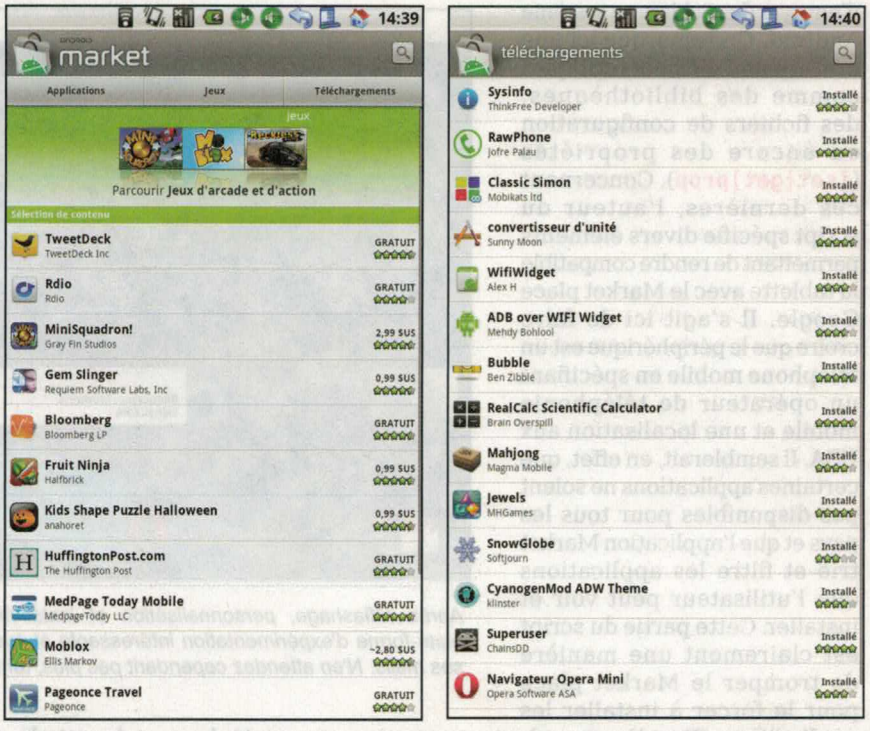
Pour obtenir un Android ID à partir de zéro, l'une des méthodes décrites sur le Web (et reprise un peu partout avec les mêmes captures) consiste à utiliser un AVD (*Android Virtual Devices*) qui n'est autre qu'une session de l'émulateur livrée avec le SDK Android. La tâche consiste simplement à créer un nouvel AVD, utiliser une image système bricolée récupérée sur un site de téléchargement, lancer le système Android et aller chercher le fameux ID dans le fichier `/data/data/com.google.android.googleapps/databases/gls.db` (certaines documentations parlent du fichier `accounts.db`).

Il faut ensuite se connecter au shell de la tablette via `adb` pour utiliser la commande `sqlite3` afin de spécifier la valeur du champ `androidId` de la table `meta` du fichier `gls.db`. Par la même occasion, on s'empressera de purger les données dans `/data/data/com.android.vending/cache/*`.

Après redémarrage (inexplicable mais conseillé par les différentes sources documentaires), on dispose alors d'un périphérique capable de se connecter au Market, mais les résultats ne sont guère à la hauteur des attentes d'un utilisateur, même très tolérant. Bien que la connexion soit possible, le processus d'installation des applications est très aléatoire. Les tentatives d'installation échouent 4 fois sur 5 et il n'est pas rare de devoir faire un tour dans les préférences système pour vider le cache de quelques services en cours d'exécution. Avec insistance, on arrive toutefois à disposer d'applications utilitaires fonctionnelles, mais le résultat est presque toujours navrant. Après des heures de boucles installation-réinstallation-échec-réinstallation-essai, la seule vraie utilité du matériel réside dans l'accès `adb` par le wifi, la navigation web, la consultation de ses mails et la découverte de l'environnement Android en lui-même.

3 UN FIRMWARE OFFICIEL AVEC LE MARKET ?

Aux termes des expérimentations autour de cette tablette, nous avons trouvé un message posté sur un site de développeurs Android présentant une nouvelle version du firmware pour l'aPad Zenithink ZT-180. Celle-ci, décrite comme plus performante, intégrait également une version directement utilisable du Market place Android. D'où vient cette mise à jour ? Difficile de le dire clairement puisque, là encore, même si l'utilisateur semblait affirmer qu'il s'agissait d'une version officielle, le téléchargement devait être fait depuis une plate-forme anonyme de type RapidShare. Le fait que les sources spécifiques à l'aPad ne soient pas disponibles pourrait confirmer cette affirmation, mais il n'est pas impossible de recomposer des images système à partir d'un aPad personnalisé. Là encore, le flou est total. Le constructeur chinois n'a très certainement pas passé d'accord avec Google pour la validation de l'aPad comme plate-forme ayant accès au Market, et ce en contradiction totale avec la politique de la firme de Mountain View. Pour Google, semble-t-il, seul Gingerbread (Android 3.0) et les versions qui suivront, Honeycomb (3.5 ?) et Icecream (??) seront considérées comme des systèmes adaptés aux tablettes.



La dernière version en date du firmware proposé par le constructeur chinois intègre l'application Market de Google. Comme le montre cette capture, l'application fonctionne relativement bien sur un écran de cette taille, mais l'installation des applications pose souvent problème. Il est fort peu problable que Google ait autorisé l'accès à son Market place pour ce périphérique. Ce qui ne semble pas déranger le constructeur le moins du monde...

Cependant, ceci constitue l'occasion de présenter ici une méthode de mise à jour sans doute moins aléatoire que la précédente. Bien entendu, le maître mot dans ces opérations de *reflashage* est souvent « Windows ». Tous les *howto*, pages de wiki et forums parlent de **burntool.exe** et ce n'est qu'après une recherche approfondie qu'on trouvera l'outil adéquat pour GNU/Linux : zt180-flash 0.1, développé par Yuri Kozlov et reposant sur la **libusb**. Heureusement, ce petit outil, qui n'est pas sans rappeler **usbpush**, livré avec d'autres plates-formes embarquées (**qt2410_boot_usb** des cartes Armzone QT2410 ou de la mini2440 de FriendlyARM), composé de trois fichiers sources C, se compile sans problème via le **Makefile** livré. On obtient ainsi trois binaires, **usbpush**, **usbread** et **usbstate** directement utilisables (en **root**), via le script shell suivant :

```
UBOOT=FW2/u-boot.img
KERNEL=FW2/zImage
SYSTEM=FW2/system.img
USERDATA=FW2/userdata.img


USBPUSH=./usbpush
USBSTATE=./usbstate
USBREAD=./usbread
#-------------------------
function die {
        echo $1
        exit 1
}


[ -f ${USBSTATE} ] || die "usbstate not found"
[ -f ${USBPUSH} ]  || die "usbpush not found"
echo "Searching files..."
[ -f ${UBOOT} ]    || die "u-boot not found"
[ -f ${KERNEL} ]   || die "kernel not found"
[ -f ${SYSTEM} ]   || die "system not found"
[ -f ${USERDATA} ] || die "userdata not found"
echo "Checking device state..."
${USBSTATE} 1 || die ""
echo "Running u-boot..."
${USBPUSH} ${UBOOT} 0 0x88 0xffffffff
echo "Waiting for device reconnected..."
sleep 3
echo "Checking device state..."
${USBSTATE} 0x1000000 || die ""
#exit 0
echo "Flashing u-boot..."
${USBPUSH} ${UBOOT}    0x00010200 0 0x40200000
echo "Flashing kernel..."
${USBPUSH} ${KERNEL}   0x00010300 0 0x40200000
echo "Flashing system..."
${USBPUSH} ${SYSTEM}   0x00010400 0 0x40200000
echo "Flashing userdata..."
${USBPUSH} ${USERDATA} 0x00000500 0 0x40200000
echo "Checking device state..."
${USBREAD} 0x6000000f ||
```

Les quatre fichiers **.img** sont extraits du fichier **download_1024x600_1013.rar** constituant la mise à jour. Pour appliquer ces modifications, il vous faudra placer la tablette dans un mode particulier en la démarrant en maintenant le bouton en façade enfoncé. L'aPad démarre alors en mode *recovery* permettant l'accès à la mémoire NAND avec les trois outils de Yuri via la connexion USB (évitez les hubs entre la tablette et le PC et optez pour une connexion directe) :

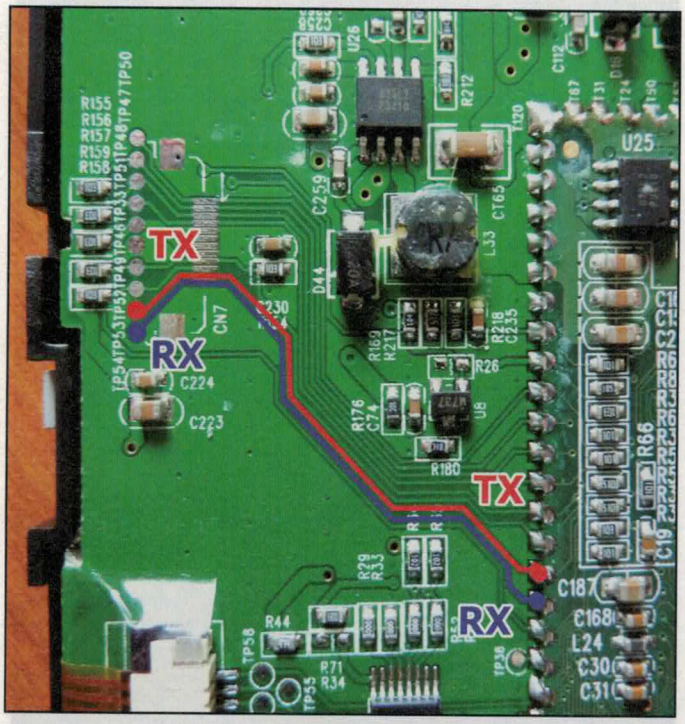
```
% ./flash.sh
Searching files...
Checking device state...
Return code: 0
Transferred: 4
Data: 1
Running u-boot...
csum = 0x19b4
send_file: addr = 0xffffffff, len = 0x0004e3d0
.........................
Waiting for device reconnected...
Checking device state...
Return code: 0
Transferred: 4
Data: 16777216
Flashing u-boot...
csum = 0x19b4
send_file: addr = 0x40200000, len = 0x0004e3d0
.........................
Flashing kernel...
csum = 0x776f
send_file: addr = 0x40200000, len = 0x002d3450
.....................................
Flashing system...
csum = 0x4142
send_file: addr = 0x40200000, len = 0x0614cd00
.........................
Flashing userdata...
csum = 0xbb15
send_file: addr = 0x40200000, len = 0x0021a500
...TTTTTTTTTTTTT.....................
Checking device state...
Return code: 0
Transferred: 4
Data: 100663311
Done
```

Voilà bien l'une des premières surprises agréables qu'on peut avoir avec ce matériel : un outil en GPL permettant d'opérer une manipulation qui fonctionne sur l'aPad. Un vrai miracle. En dehors de cela, le système obtenu sur la tablette, après redémarrage (reset physique), correspond plus ou moins à ce qu'on aurait obtenu avec la méthode précédente. Le système semble plus rapide au premier abord, mais arrive rapidement à ses limites au bout de quelques minutes d'utilisation.

On notera cependant le gain obtenu de par cette manipulation : nous disposons d'un outil pour mettre à jour la NAND si le périphérique est complètement bloqué (« briqué/brické » comme aiment le dire les utilisateurs de smartphones et de forums à la mode). En dehors de cela, la conclusion est toujours la même et n'est pas des plus brillantes.

4 ET SI ON SOUHAITE INSTALLER UN AUTRE SYSTÈME ?

Aux vues de ce qu'il est possible d'obtenir de l'aPad et si l'on ne se satisfait pas vraiment d'un système qui se résume à une plate-forme bas de gamme pour l'expérimentation Android, une idée vient tout naturellement à l'esprit : « Et si on utilisait un vrai GNU/Linux embarqué ? ». Si tel est votre état d'esprit, vous devrez faire preuve de courage et de ténacité. La première étape consiste à prendre la main sur le processus de démarrage. Pour ce faire, rien de mieux qu'un accès direct au *bootloader* via une console série.



Après ouverture de l'arrière de la tablette, on découvre une carte maîtresse de taille réduite soudée sur un support offrant l'accès à la connectique externe. Cette carte dispose d'une console série permettant d'avoir un accès direct au bootloader. On préférera cependant se brancher sur la carte support, plus facile d'accès en utilisant des TP (Test Points) 53 et 54. Les tensions utilisées sont 0-3.3V. Ne connectez donc pas ces lignes directement à un port RS232 d'un PC. Si nécessaire, il est possible de récupérer une masse et un Vcc (tension d'alimentation) aux bornes du régulateur de tension U23.

Le SoC iMAPx200 dispose, comme tout système embarqué qui se respecte, d'une console série permettant de prendre la main en cas de gros problème. L'illustration ci-contre montre comment connecter une liaison série sur la carte principale de l'aPad. Dans le modèle de test, l'absence d'un composant à proximité nous a permis d'éviter une soudure directe sur la carte principale et d'utiliser les TP (*Test Points*) 53 et 54. Attention ! Les niveaux des signaux sont en 0-3.3V, vous devrez utiliser un adaptateur USB/série compatible pour la connexion. Un peu de dextérité et un bon fer à souder et vous serez à même de connecter **cu**, **minicom** ou **screen** (115200 8N1). Ceci après avoir ouvert le boîtier, bien entendu ! Pour cela, remarquez les deux petits trous sur la partie supérieure du périphérique (dont un est peut-être couvert par un sticker de garantie/qualité). Pour ouvrir l'aPad, vous devrez insérer une fine tige métallique (type trombone déplié, modèle MacGyver standard) dans ces trous et appliquer une ferme pression tout en vous aidant de la gravité pour détacher la coque arrière de la tablette. Ce faisant, vous déplacez les clips à ressort maintenant l'écran en place et celui-ci se désolidarise. La manipulation est délicate, mais ce n'est rien comparé à l'opération inverse (vous comptiez vraiment le refermer ?).

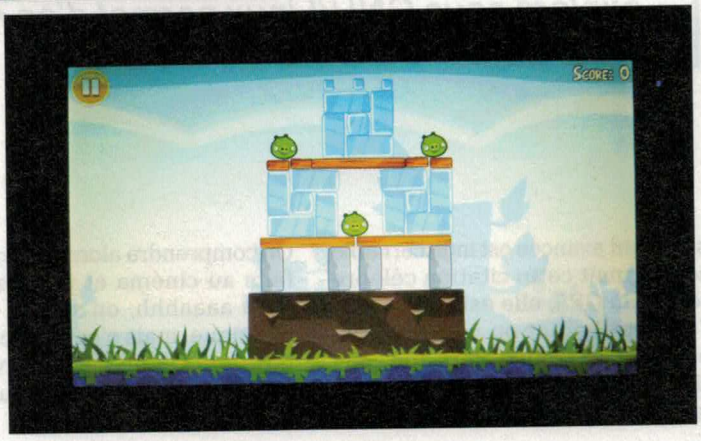
Une fois la connexion établie via la liaison série, vous aurez le privilège d'obtenir un affichage semblable à celui-ci :

```
U-Boot 2009.08 (Aug 25 2010 - 10:35:38) for
    android 1008MHz ddr126 256M 4L fast

CPU :  iMAPx200 @ 1008 MHz
FCLK:  1008 MHz, HCLK: 126 MHz, PCLK:  63 MHz
SDRAM: 256 MB
MMU :  Enabled!!
NAND:  2048 MiB
*** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
MMC:   IMAP_SDHCI: 0
Net:   GMAC
GPBDAT is 0xf , 0x20e10010
Bootup U1 successfully!!
GPBDAT 2 is 0xf
Enter usb updata mode: *0x4fff0000 = 0x1f75dfe5
Hit any key to stop autoboot:  0
oem_bootl : args = console=ttySAC1,115200
    rdinit=/init mem=192M android.ril=ttyUSB1


NAND read: device 0 offset 0x500000, size 0x300000
3145728 bytes read: OK
## Booting kernel from Legacy Image at 40007fc0 ...
   Image Name:   Linux-2.6.31.6
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2959460 Bytes =  2.8 MB
   Load Address: 40007fc0
   Entry Point:  40008000
   Verifying Checksum ... OK
   XIP Kernel Image ... OK
OK


Starting kernel ...
Uncompressing Linux.................done, bootin
g the kernel.
Linux version 2.6.32.9 (wwang@android)
    (gcc version 4.3.3 (GCC) ) #1705
    Mon Aug 30 11:39:57 CST 2010
```

```
CPU: ARMv6-compatible processor [4117b365]
    revision 5 (ARMv6TEJ), cr=00c5387f
CPU: VIPT nonaliasing data cache,
    VIPT nonaliasing instruction cache
Machine: IMAPX200
Ignoring unrecognised tag 0x00000000
Ignoring unrecognised tag 0x54410008
Memory policy: ECC disabled, Data cache writeback
CPU IMAPX200 (id 0x13ab2000)
IMAP Clocks, (c) 2009 Infotm Micro Electronics
IMAPX200: PLL CLOCKS, APLL 1008.000 MHz,
    DPLL 1296.000 MHz, EPLL 480.000 MHz
IMAPX200: CPU_CLK 1008.000 MHz, HCLK 126.000 MHz, PCLK 63.000 MHz
IMAP Power Management, Copyright 2010 Infotm
[...]
```

Grosse déception ! Le bootloader U-Boot installé n'est pas pleinement opérationnel/configuré. En réalité, il se charge d'un minimum de manipulations consistant à initialiser la mémoire et le système, puis à charger le noyau Linux et l'exécuter. Il sera cependant possible de « faire avec » en utilisant U-Boot tel que livré ainsi que le noyau Linux.

Sur cette base, nous pourrions construire un système complet avec Buildroot pour tout ce qui concerne les outils *userland*. La tâche est conséquente, mais une chaîne de compilation GCC ARM EABI serait suffisante. À noter que des annonces ont été faites par quelques utilisateurs sur le Web, mentionnant le portage de distributions comme OpenWrt ou Ubuntu. Cependant, rien ne permet d'étayer la véracité de ces annonces (*dump*, capture, vidéo) qui ont été faites dans un milieu où le prestige des beaux parleurs tient plus dans de vides affirmations que dans d'éventuelles démonstrations techniques bien documentées. L'utilisation, de concert, des outils de Yuri avec cette version *light* d'U-Boot constitue un point de départ intéressant, mais rien ne semble avoir été réellement fait pour l'instant. Rappelons que les sources les plus importantes du système, les pilotes de périphériques (audio, Ethernet, i2c, framebuffer) ne sont pas disponibles et qu'à moins de procéder à une analyse des binaires, on ne peut reposer que sur ce que fournit Zenithink.



Il existe plusieurs manières d'installer des applications sur un périphérique Android. Ici, nous avons l'un des jeux les plus populaires du moment, Angry Birds, installé manuellement avec adb. L'aPad et ses ressources ne s'en sortent finalement pas trop mal pour peu que l'on fasse le ménage dans les processus en cours d'exécution.

CONCLUSION : POUR EXPÉRIMENTER UNIQUEMENT !

La conclusion de cette petite excursion dans le monde de l'aPad est décevante. La plate-forme matérielle, bien que bas de gamme, permettrait un certain nombre d'expérimentations intéressantes si un peu plus d'informations étaient diffusées par le constructeur. Force est de constater que l'ouverture n'a jamais été un élément important concernant la distribution de ce matériel. L'objectif est clairement de diffuser un clone raté de la tablette Apple en jouant sur la popularité grandissante du système développé par Google. C'est la masse de « premiers achats » qui intéresse le fabricant et les réseaux de distributeurs. C'est bien dommage, car à l'aide d'un simple SVN/Git/CVS et quelques pages d'un wiki officiel, une communauté de développeurs pourrait immédiatement se créer et faire évoluer le produit rapidement.

Au lieu de cela, l'utilisateur curieux et soucieux d'en apprendre un peu plus sur le développement et la composition d'applications Android aura mieux fait de se tourner vers un simple smartphone d'occasion, plus stable, plus véloce, presque plus ouvert, et surtout, finalement, plus économique. À titre d'exemple, un simple coup d'œil sur un site d'enchères en ligne et l'on constate qu'un « 10.2 Zenithink ePad/aPad Google Android 2.1 Tablet PC » se vend à quelques 140 euros (+ 30 euros de port) contre 110 euros (+ 10 euros de port) pour un HTC Tatoo d'occasion. Une différence de 50 euros en faveur d'un smartphone disposant d'un accès légal au Market place et d'une documentation en ligne bien plus riche (si on fait abstraction des forums en chinois pour l'aPad).

En quelques mots, si vous souhaitez explorer Android, optez pour un smartphone d'occasion ou attendez de véritables tablettes (sous Android 3.0 ou 3.5). L'aPad n'est un choix intéressant que si vous cherchez le challenge et voulez relever le défi du portage d'un système GNU/Linux sur le périphérique Zenithink. ■

Auteur : Denis Bodor

Lire et décoder les informations

Tout le monde sait ce qu'est un GPS ou du moins tout le monde croit le savoir. Pour le commun des mortels, il s'agit d'un périphérique permettant de savoir, tout simplement, où l'on se trouve et ainsi de retrouver son chemin. Le GPS, pour Global Positioning System, est en réalité bien plus qu'un équipement, mais un système complet. Son fonctionnement exploré sous GNU/Linux permet d'en révéler toute la beauté et la magie.

Auteur

- Denis Bodor

« Toute technologie suffisamment avancée est indiscernable de la magie ». Tout le monde connaît cette citation célèbre d'Arthur C. Clarke. Dans le cas du GPS, elle est sans doute plus vraie qu'ailleurs. Qu'un petit boîtier ou un téléphone sache à quelques mètres près où il se trouve alors que vous n'en avez pas la moindre idée relève réellement de la magie pour qui n'en connaît pas le principe de fonctionnement. On comprendra alors les raccourcis malheureux généralement faits au cinéma et parfois dans la vie courante, du type : « Ah aaaahhh, on sait où il se trouve, il a un GPS ! ». Un classique, mais c'est le récepteur GPS qui connaît sa propre position. On ne peut trouver quelqu'un simplement parce qu'il possède un récepteur sur lui, sauf si le récepteur en question émet un signal.

1 PRINCIPES DE FONCTIONNEMENT

Bien entendu, il ne s'agit pas de magie, mais le fonctionnement global du système de positionnement, les principes mis en œuvre et les paramètres pris en compte ont vraiment quelque chose de fantastique. Le GPS se compose de 31 satellites NAVSTAR en orbite autour de la Terre. Ceux-ci doivent être au nombre minimum de 24 en orbite à 20200 km d'altitude pour que le système soit fonctionnel. Chaque satellite émet des données appelées code C/A et code P. Le premier est librement utilisable par le domaine civil alors que le second est réservé à un groupe limité d'utilisateurs, dont les entités militaires. Bien entendu, les données code P permettent une plus grande précision (jusqu'au millimètre) que le code C/A.

Le principe de fonctionnement est basé sur l'information transmise par les satellites embarquant une horloge atomique. Grossièrement, ils ne font que transmettre l'heure de l'émission. Par extrapolation et calcul, le récepteur peut déduire la distance qui le sépare du satellite en se basant sur la vitesse de propagation du signal. La position d'un satellite est prévisible et chacun d'eux s'identifie par un code. En effet, les satellites, vus du sol, reprennent la même position dans le ciel au bout d'un jour sidéral.

Il est plus juste de parler de pseudo-distance lorsqu'il s'agit de mesurer l'espace séparant l'observateur (le périphérique de réception GPS) d'un satellite. En effet, le calcul se base sur le temps de propagation du signal par calcul de la différence entre l'heure exacte actuelle et celle présente dans le message du satellite. Le signal se propageant à la vitesse de la lumière, une erreur de 0.000001 seconde correspond à une distance erronée de 300 mètres. Pour obtenir la distance véritable, l'observateur doit faire une estimation du décalage entre son horloge et celle des satellites. Pour cela, il doit obtenir les informations de 4 satellites au moins. Une fois le décalage calculé, il peut déterminer sa position. Pour procéder à ces deux opérations, le périphérique, que vous preniez sans doute pour un simple gadget, doit résoudre un système d'équations à quatre inconnues (trois inconnues pour la position du récepteur, plus le décalage de son horloge par rapport au temps GPS).

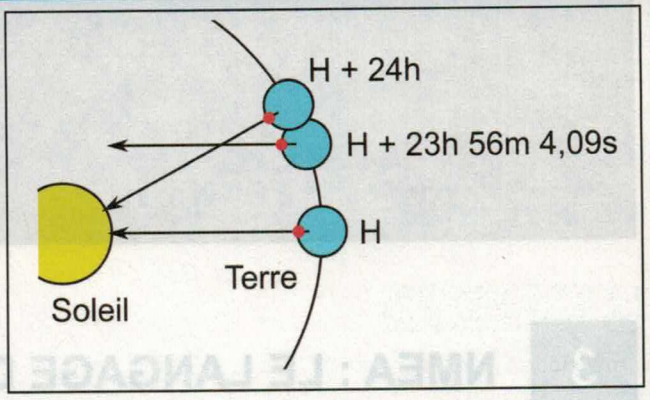
Une fois connecté, le petit périphérique acheté quelques 20 dollars sur un site d'enchères en ligne calcule en permanence et recherche à résoudre de manière continue une équation lui permettant d'interpréter les informations qu'il reçoit. Mais ce n'est pas tout et c'est là que tout devient extraordinaire. Non seulement vous voici en train d'écouter des informations provenant de l'espace, mais en plus, vous devez prendre en compte les principes de la relativité restreinte et la relativité générale.

Eh oui, le satellite se déplace à une vitesse de l'ordre de 14 000 km/h. À cette vitesse, la relativité restreinte implique que le temps ne s'écoule pas de la même manière

GPS

Jour sidéral vs jour solaire

Cette unité de temps ne correspond pas à la notion de jour telle que nous la connaissons, où un jour de 24 heures correspond à une rotation de la Terre pour qu'un habitant revienne exactement à la même position en face du Soleil. On devrait alors parler de jour solaire. Cependant, pour que cet événement se produise, la Terre aura fait plus qu'un tour sur elle-même car elle tourne autour du Soleil et non pas seulement sur elle-même. La simple rotation de la Terre par rapport aux étoiles est appelée jour sidéral. Le temps nécessaire pour cette rotation complète est de 23 heures 56 minutes et 4,09 secondes. Le temps formant la différence avec nos jours (solaires) de 24 heures correspond au temps de rotation de la Terre pour que la même position à la surface terrestre revienne face au Soleil (la différence correspond à un angle de 1 degré environ).



qu'à la surface terrestre. La relativité générale, quant à elle, nous permet de savoir que la gravité terrestre à laquelle est soumis le récepteur au sol impacte également l'écoulement du temps. Heureusement, ces données sont connues et calculables. Mieux encore, elles le sont par le satellite lui-même de manière à corriger les données pour qu'elles s'avèrent exactes une fois réceptionnées au sol.

Un nombre de quatre satellites est un minimum pour obtenir une position, mais les récepteurs sont capables d'utiliser plus de données en provenance de plus de satellites pour affiner les calculs et corriger les erreurs. Ainsi, plus de satellites sont suivis, meilleurs seront l'interprétation des données et le calcul de la position exacte au sol, et ce, en trois dimensions (longitude, latitude et altitude).

2 RÉCEPTION GPS SOUS LINUX

Les récepteurs GPS existent sous de nombreuses formes, allant de l'ordinateur embarqué dans une voiture au GPS de randonné robuste et léger, en passant par le module de réception intégré dans la plupart des *smartphones*. Toutes ces solutions matérielles, cependant, sont généralement totalement intégrées dans une *appliance*. Difficile, voire impossible alors, de lire directement les données décodées émises par les satellites.

Il faut se tourner, pour cela, vers un autre type de matériel : le récepteur simple. Celui-ci peut prendre deux formes différentes. Soit vous optez pour un module comme ceux distribués par Sparkfun, que vous pourrez intégrer à vos montages, soit vous vous dirigez vers la solution « customer », à savoir l'achat d'un simple récepteur USB pour 20 à 30 dollars. Les modules offrent un niveau d'intégration supérieur en raison de leur aspect compact et du fait qu'ils peuvent intégrer des solutions de stockage de plus ou moins grande capacité. Si vous êtes plutôt intéressé par l'exploration et l'expérimentation, en revanche, mieux vaudra vous diriger vers le périphérique USB. Dans les deux cas, les récepteurs GPS transmettent leurs données via une interface série (généralement à 4800 bps). Les modules le font en TTL alors que les périphérique USB utilisent tout simplement un port série virtuel.

Ainsi, à la connexion d'un récepteur GPS comme celui utilisé pour cet article, vous verrez apparaître tout simplement un nouveau port sur votre système :

```
[93720.352069] usb 2-1: new full speed USB device using uhci_hcd and address 4
[93720.510605] usb 2-1: New USB device found, idVendor=067b, idProduct=2303
[93720.510619] usb 2-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[93720.510629] usb 2-1: Product: USB-Serial Controller
[93720.510637] usb 2-1: Manufacturer: Prolific Technology Inc.
[93720.511132] usb 2-1: configuration #1 chosen from 1 choice
[93720.512844] pl2303 2-1:1.0: pl2303 converter detected
[93720.524821] usb 2-1: pl2303 converter now attached to ttyUSB0
```

Le type d'adaptateur varie en fonction du périphérique utilisé (*Prolific Technology*, FTDI, etc.), mais le résultat est le même. Dès la connexion, le module se met à émettre des messages que vous pourrez consulter avec **cu**, **minicom** ou encore le sympathique **screen**. Tout ce que vous avez à faire, c'est diriger votre outil sur le port nouvellement créé et régler la vitesse sur 4800 bps pour voir apparaître les messages en provenance de l'espace :

```
$GPRMC,091756.114,A,4815.4196,N,00726.5176,E,0.00,0.00,221010,,,A*62
$GPGGA,091757.114,4815.4194,N,00726.5176,E,1,04,4.1,174.0,M,47.9,M,,0000*5A
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPRMC,091757.114,A,4815.4194,N,00726.5176,E,0.00,0.00,221010,,,A*61
$GPGGA,091758.114,4815.4195,N,00726.5176,E,1,04,4.1,174.3,M,47.9,M,,0000*57
```

```
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPGSV,3,1,09,25,79,045,,30,72,305,35,29,62,215,17,12,43,087,*78
$GPGSV,3,2,09,31,36,308,40,02,31,062,29,14,25,241,15,04,08,033,24*7D
$GPGSV,3,3,09,09,05,148,*41
$GPRMC,091758.114,A,4815.4195,N,00726.5176,E,0.00,0.00,221010,,,A*6F
$GPGGA,091759.114,4815.4204,N,00726.5177,E,1,04,4.1,175.6,M,47.9,M,,0000*58
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPRMC,091759.114,A,4815.4204,N,00726.5177,E,0.00,0.00,221010,,,A*64
$GPGGA,091800.114,4815.4206,N,00726.5177,E,1,04,4.1,176.1,M,47.9,M,,0000*5D
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPRMC,091800.114,A,4815.4206,N,00726.5177,E,0.70,0.00,221010,,,A*62
$GPGGA,091801.114,4815.4205,N,00726.5176,E,1,04,4.1,176.2,M,47.9,M,,0000*5D
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPRMC,091801.114,A,4815.4205,N,00726.5176,E,0.72,0.00,221010,,,A*63
$GPGGA,091802.114,4815.4205,N,00726.5176,E,1,04,4.1,176.5,M,47.9,M,,0000*59
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
$GPRMC,091802.114,A,4815.4205,N,00726.5176,E,0.86,0.00,221010,,,A*6B
$GPGGA,091803.114,4815.4205,N,00726.5176,E,1,04,4.1,176.6,M,47.9,M,,0000*5B
$GPGSA,A,3,30,31,02,04,,,,,,,,,5.2,4.1,3.1*35
```

Notez au passage que le périphérique USB n'est finalement rien d'autre qu'un module de réception GPS interfacé avec un convertisseur série TTL vers USB. En d'autres termes, pour peu que le périphérique puisse être démonté facilement, vous pouvez alors directement utiliser le récepteur avec un montage si vous court-circuitez le convertisseur. Bien entendu, vous ne connaissez pas l'applicabilité de cette solution avant d'avoir le récepteur en main et une vue du circuit interne. Il arrive qu'après ouverture, bien que les journaux système parlent d'un convertisseur pl2303, il ne soit pas possible d'accéder aux composants (résine, cache en céramique, etc.). Nous nous en tiendrons ici à l'utilisation via le port USB d'un PC, mais sachez que la communication série devrait être parfaitement possible avec un AVR, par exemple.

3 NMEA : LE LANGAGE DU GPS

Comme nous venons de le voir, le récepteur GPS diffuse un flot d'informations dans un format standardisé (mais non normalisé et non ouvert, voir http://www.nmea.org/content/nmea_standards/nmea_083_v_400.asp). Il s'agit du standard NMEA 0183 (*National Marine Electronics Association*) qui combine les spécifications à la fois pour l'aspect électronique et pour le format de données. NMEA ne se limite pas aux récepteurs GPS, il est également utilisé pour plusieurs équipements de marine, comme les sonars, les gyrocompas ou encore les anémomètres. D'un point de vue matériel, le standard indique une communication unidirectionnelle de type série à 4800 bps, 8 bits de données, pas de parité et un bit de stop (8N1).

Au niveau format des données, voici quelques spécifications :

- Chaque message débute par le symbole **$** et se termine par CRLF (**0x0D0A**).
- Les cinq caractères suivant le **$** permettent d'identifier l'équipement émetteur du message sur deux caractères (**GP** pour *Global Positioning system receiver*) et le type de message sur les trois autres caractères.
- Les données qui suivent cet en-tête sont organisées en champs délimités par une virgule.
- Lorsqu'une donnée n'est pas disponible, le champ est vide.
- Le premier caractère après la liste des champs est un astérisque (*).
- Immédiatement après * est spécifiée une somme de contrôle en notation hexadécimale. Le calcul est simple, il s'agit d'un XOR de tous les caractères présents entre **$** et *. Cette somme de contrôle est optionnelle pour certains types de messages, mais obligatoire pour d'autres.

Le format de données NMEA, bien que largement utilisé, est un format propriétaire. Vous pourrez acheter les spécifications complètes sur le Web. Sachez cependant que certains fabricants ajoutent leurs propres spécifications en supplément de NMEA 0183 en vue de fournir un certain nombre d'informations supplémentaires sur leurs récepteurs. Fort heureusement, côté open source, un certain nombre de travaux ont permis d'en apprendre davantage sur le sujet et de développer des applications à même d'interpréter les messages NMEA (ainsi que certaines de leurs extensions).

Le document de référence fait partie de la documentation de **gpsd**. Il a été écrit par Eric S. Raymond et est librement consultable à http://gpsd.berlios.de/NMEA.txt.

Ce document référence un grand nombre de types de messages, mais tous les équipements ne les utilisent pas dans leur ensemble. De plus, il faut savoir que beaucoup de récepteurs d'entrée de gamme utilisent un jeu réduit de messages, et parfois, ne respectent pas le standard à la lettre. Avec un petit module comme celui servant d'exemple pour cet article, les messages se résument aux types suivants :

3.1 RMC : Recommended Minimum Navigation Information

Ce sont les informations minimums permettant de déterminer la position. Exemple :

```
$GPRMC,134517.000,A,4815.4256,N,00726.5176,E,0.00,172.23,251010,,,A*62
```

On retrouve le type de message (**RMC**) suivi de l'heure GMT (**134517** pour 13h45h17s), de l'état (**A** comme actif ou **V** comme *void*), de la latitude (48° 15.4256' Nord), de la longitude (7° 26.5176' Est), de la vitesse (ici zéro), etc. Les deux derniers champs avant la somme de contrôle, vides ici, permettent de connaître la déclinaison magnétique qui pourra alors être utilisée pour calibrer/corriger une boussole électronique.

3.2 GGA : Global Positioning System Fix Data

GGA est une autre source d'information de positionnement, permettant également d'en savoir plus sur l'état de la réception et la validité des informations. Voici un exemple de message **GGA** :

```
$GPGGA,134518.000,4815.4256,N,00726.5176,E,1,07,1.7,189.2,M,47.9
,M,,0000*54
```

En plus des informations déjà présentes dans un message **RMC**, on peut ainsi connaître la qualité des données en position 6. Ici, **1** correspond à une donnée d'origine GPS. D'autres valeurs sont possibles en fonction des récepteurs pouvant améliorer la précision en utilisant, par exemple, des stations au sol pour affiner le positionnement (DGPS). Le septième champ est également très intéressant puisqu'il nous indique le nombre de satellites utilisés pour le positionnement (ici **07**). Suit la précision du positionnement en mètres, avec ici 1,7 mètres, ce qui est tout à fait excellent pour un récepteur posé derrière une fenêtre au rez-de-chaussée d'un immeuble.

Avec suffisamment de satellites, la pertinence de l'information fournie permet non seulement un positionnement 2D précis, mais également 3D (4 satellites minimum). Le neuvième champ nous permet ainsi de connaître l'altitude calculée du récepteur. On remarquera cependant que ce type de données est normalement corrélé avec un altimètre utilisant la pression atmosphérique avec les GPS de randonnées. En parlant de précision...

3.3 GSA : GPS DOP and active satellites

GSA est un type de message plus intéressant pour l'utilisateur curieux. Il permet, en effet, de savoir quels satellites sont effectivement utilisés pour le positionnement. DOP signifie *Dilution Of Precision*. Il s'agit d'un indicateur nous renseignant sur l'impact de la position du satellite dans la précision des données que nous obtenons. En effet, si tous les satellites utilisés se trouvent dans une zone réduite, la précision du calcul sera moins importante que s'ils sont équitablement répartis dans le ciel.

Un message **GSA** ressemble à ceci :

```
$GPGSA,A,3,21,16,07,19,03,15,06,,,,,2.3,1.5,1.7*3B
```

Nous apprenons ici que le récepteur est en mode automatique pour le choix du positionnement 2D ou 3D (1er champ). Nous savons également que la qualité de réception est suffisante pour un positionnement 3D (**3** dans le second champ). Suivent ensuite les PRN des satellites utilisés (**21**, **16**, **07**, **19**, **03**, **15** et **06**). Vous conviendrez avec moi que c'est là quelque chose de captivant. Le PRN 21, par exemple, est un satellite GPS IIR-9 lancé le 31 mars 2003 et il fait partie du Block IIR. Moins anecdotique, les derniers champs nous informent de la valeur DOP positionnelle (3D), horizontale et verticale. La notion de DOP est héritée du système de positionnement Loran-C toujours en activité, mais considérée comme une solution de repli en cas de problème avec le GPS. La valeur DOP idéale est 1 ; plus elle est basse, plus la précision est grande. En dessous de 2, elle peut être considérée comme excellente ; jusqu'à 5, elle est bonne. À 10, elle devient moyenne et au-delà de 20, elle est médiocre.

GSA nous fournit des informations intéressantes en plus d'être utiles, mais la suite l'est encore plus.

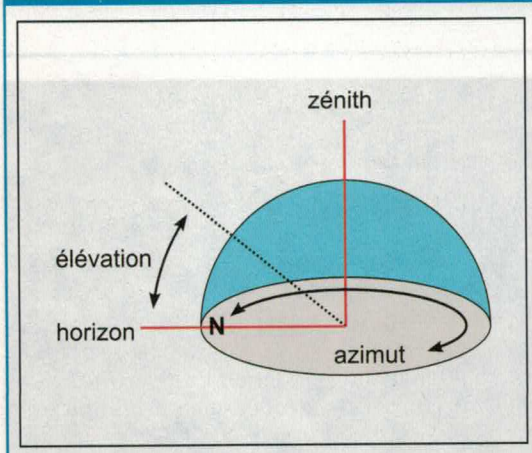
3.4 GSV : Satellites in view

GSV nous apporte ce que notre curiosité nous pousse à vouloir connaître : mais où sont les satellites ? Dans les messages de type **GSA**, on remarque qu'il est possible de lister quelques 12 satellites. On retrouve ici cette valeur, mais divisée en plusieurs messages. En effet, la notion de groupe de messages intervient lorsqu'il n'est pas possible de stocker les informations dans un seul.

Exemple :

```
$GPGSV,3,1,12,06,81,307,27,03,65,295,26,16,51,191,22,18,48,087,22*7C
$GPGSV,3,2,12,22,46,143,19,21,34,062,28,19,32,291,27,24,20,239,*77
$GPGSV,3,3,12,07,13,310,38,15,08,050,22,08,04,341,19,26,04,019,26*71
```

Élévation et azimut



Ces deux valeurs permettent de définir l'emplacement d'un objet (satellite ou astre) dans le ciel. L'élévation est l'angle formé entre l'horizon et la position verticale de l'objet dans le ciel. Le zénith étant défini comme la position à la verticale de l'observateur, il se trouve à une élévation de 90°. L'angle formé entre le zénith et la position de l'objet est appelé angle de zénith.

Horizontalement, cette fois, pour repérer l'objet, on utilise comme point de référence le nord géographique. L'azimut est mesuré par l'angle entre ce point de référence et la position de l'objet dans le sens des aiguilles d'une montre. Ainsi, l'Est se trouve par exemple à 90° d'azimut, le Sud à 180° et l'Ouest à 270°.

Ces deux valeurs permettent de repérer n'importe quel objet sur la voûte céleste. Une représentation 2D est souvent utilisée, sous la forme d'un disque dont le centre est l'observateur et où l'élévation est représentée par la distance entre le centre (90°) et le périmètre (0°). Voir capture de **xpgs**.

Nous avons là trois messages **GSV**. Le premier précise dans le premier champ le nombre total de messages composant le groupe, puis son numéro dans ce dernier (**1**). À la troisième position, on retrouve une indication sur le nombre total de satellites « en vue ». Attention, il ne faut pas confondre ici « en vue » avec « utilisés » (messages **GSA**). Le choix et l'utilisation des satellites est l'affaire du récepteur et ce, en fonction de ses capacités. Après cette valeur, nous avons une succession de données pour chaque satellite : un PRN, l'élévation en degrés, l'azimut en degrés et le SNR. Cette dernière valeur définit le rapport signal/bruit. Plus il est important, plus la réception est bonne. Il est possible d'avoir au maximum 4 satellites ainsi listés par message pour un total de 12 pour un groupe. Le PRN pour *Pseudo Random Noise* est une suite de modulation binaire se répétant sur 267 jours. Un segment d'une semaine est unique à chaque satellite et change de façon hebdomadaire. Le PRN dont il est question ici est en réalité un nombre permettant d'identifier un satellite « en poste ».

Il est relativement aisé de lire tous ces messages à l'aide d'un simple code en Python, par exemple, via le module Python Serial. Les messages étant parfaitement standardisés, il n'y a aucune surprise quant à leur interprétation. Exemple :

```
import serial
import string
import signal
import sys

gps = serial.Serial('/dev/ttyUSB0', 4800, timeout=1)

def signal_handler(signal, frame):
        print 'Exiting.'
        gps.close()
        sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)

def convertStr(s):
        try:
                ret=float(s)
        except ValueError:
                ret=0
        return ret

while 1:
        line = gps.readline()
        datablock = line.split(',')

        if line[0:6] == '$GPRMC':
                latitude_in = convertStr(datablock[3])
                longitude_in = convertStr(datablock[5])
                if datablock[4] == 'S':
                        latitude_in = -latitude_in
                if datablock[6] == 'W':
                        longitude_in = -longitude_in
                latitude_degrees = int(latitude_in/100)
                latitude_minutes = latitude_in - latitude_degrees*100

                longitude_degrees = int(longitude_in/100)
                longitude_minutes = longitude_in - longitude_degrees*100

                latitude = latitude_degrees + (latitude_minutes/60)
                longitude = longitude_degrees + (longitude_minutes/60)
                print line.rstrip('\n')
                print "Longitude :%s  Latitude:%s" %
(longitude,latitude)
```

Ceci nous affichera des lignes comme :

```
Longitude :7.44190166667  Latitude:48.2570433333
```

Ce simple programme, inspiré du code de démonstration en GPL de Jaroslaw Zachwieja, vous permettra très simplement, via les messages **RMC**, de produire un fichier KML pour une utilisation dans Google Earth ou Google Map (**http://earth.google.com/kml/2.0**). C'est là l'un des principaux « bons côtés » de NMEA : l'utilisation avec un système embarqué ou un microcontrôleur est aisée et ne nécessite pas énormément de ressources. Un *tracker* GPS pourra être construit très facilement à l'aide de quelques composants, comme un Atmel AVR, une carte SD et éventuellement un afficheur LCD 16*2.

4 UNE SOLUTION PLUS SIMPLE POUR L'EMBARQUÉ : GPSD

La lecture des messages NMEA implique l'utilisation d'un outil ou d'un code sur la même machine que celle où est connectée le récepteur. De plus, dans le cas d'un outil « maison », vous devrez décoder vous-même les messages. Heureusement, **gpsd** règle ces deux petits problèmes en proposant un démon se chargeant de l'écoute des messages d'un ou plusieurs récepteurs GPS et mettant à disposition un service réseau (TCP 2947 par défaut).

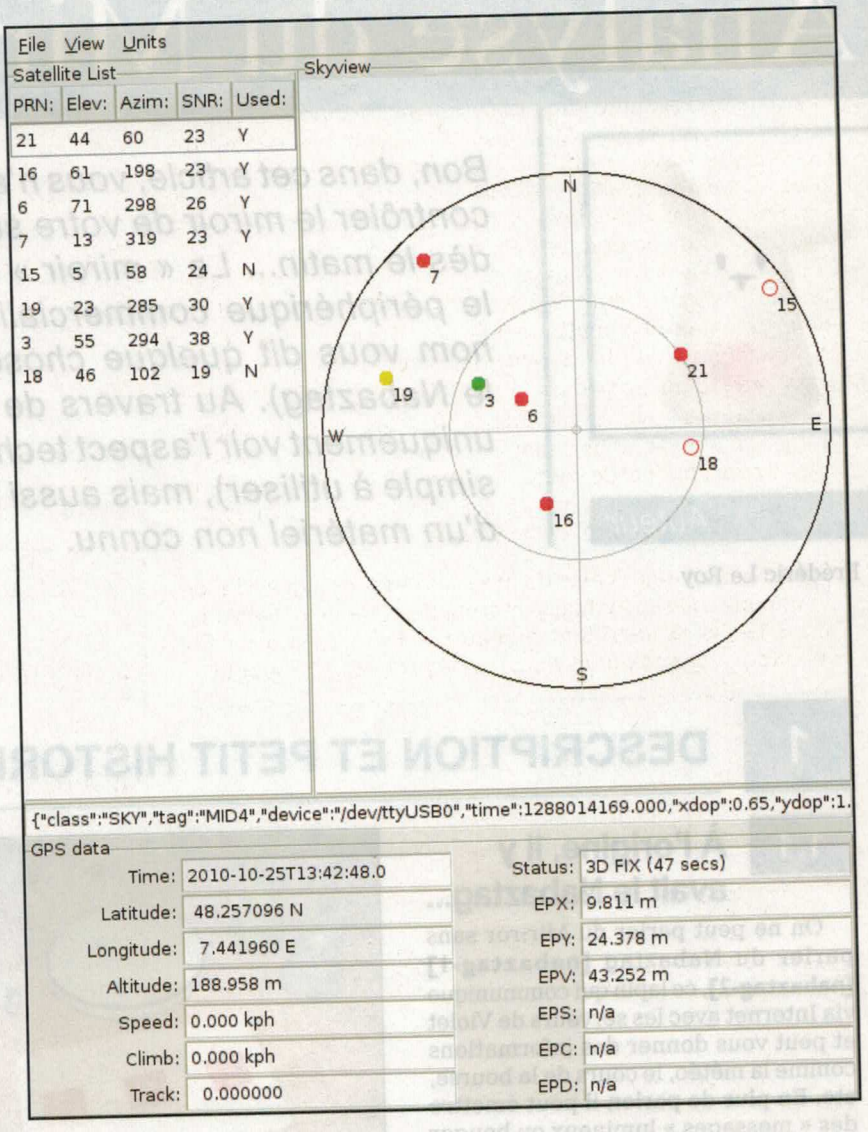
La prise en charge des récepteurs est automatique. On choisira généralement de laisser le système configurer seul l'ensemble (via **udev**) et éventuellement lancer directement **gpsd** lors de la connexion d'un récepteur. Cependant, à des fins de test, on utilisera tout d'abord le démon ainsi :

```
% sudo /usr/sbin/gpsd -n -G  -S 4000 -N -D 2 /dev/ttyUSB0
gpsd: launching (Version 2.95)
gpsd: listening on port 4000
gpsd: running with effective group ID 0
gpsd: running with effective user ID 0
gpsd: opening GPS data source type 3 at '/dev/ttyUSB0'
gpsd: speed 4800, 8N1
gpsd: attempting USB device enumeration.
gpsd: 1d6b:0002 (bus 1, device 1)
gpsd: 1d6b:0001 (bus 2, device 1)
gpsd: 1d6b:0001 (bus 3, device 1)
gpsd: 1d6b:0001 (bus 4, device 1)
gpsd: 1d6b:0001 (bus 5, device 1)
gpsd: 0ac8:c33f (bus 1, device 4)
gpsd: 0eef:480e (bus 2, device 2)
```

```
gpsd: 0a5c:219b (bus 4, device 2)
gpsd: 067b:2303 (bus 3, device 2)
gpsd: vendor/product match with 091e:0003 not found
gpsd: speed 9600, 8O1
gpsd: speed 4800, 8N1
gpsd: gpsd_activate(): opened GPS (fd 6)
gpsd: NTPD ntpd_link_activate: 1
gpsd: /dev/ttyUSB0 identified as type Generic NMEA
(0.641568 sec @ 4800bps)
```

L'option **-n** permet au démon de récupérer en permanence les messages NMEA, même si aucun client réseau n'est connecté, afin que le récepteur ne passe pas en mode veille. **-G** nous permet de préciser au démon de se mettre à l'écoute de connexions sur toutes les interfaces réseau (par défaut, seul le *localhost* est utilisé). **-S 4000** nous permet de spécifier le port TCP 4000 pour l'écoute en lieu et place du 2947 par défaut. Les deux options **-N** et **-D 2** sont utiles pour les phases d'expérimentation en demandant respectivement au démon de rester en premier plan et d'utiliser un niveau de débogage nous permettant d'en savoir un peu plus sur ce qui se passe. En fin de ligne, on précise en argument le périphérique à utiliser.

Dès lors, le démon sera à l'écoute sur le port 4000 et sera accessible via un protocole qui lui est propre, mais parfaitement documenté. De nombreuses applications open source reposent sur **gpsd** car les bibliothèques existent et l'interprétation des messages NMEA s'en trouve centralisée. Il faut également savoir que le code de **gpsd** est audité et testé régulièrement. Le démon est livré avec un exemple très sympathique écrit en Python/GTK : **xgps**. Il ne s'agit pas d'une application de cartographie, mais simplement de visualisation des informations mises à disposition du démon.



On retrouve alors à l'écran la liste des satellites en vue (avec une valeur SNR) et utilisés, ainsi que les informations de positionnement. Une représentation 2D de la position des satellites complète le tout de manière fort agréable. Notez que **gpsd** peut également être utilisé comme base de temps NTP. Vous ne disposez sans doute pas d'une horloge atomique, mais chaque satellite de la constellation GPS en possède une. Vous pouvez donc relativement simplement coupler la réception GPS avec votre configuration serveur actuelle comme horloge de référence pour tout votre réseau. Précisons que les dernières versions de **gpsd** supportent D-BUS et sont capables de diffuser les informations de positionnement à toutes les applications du bus logiciel.

Enfin, si la question vous a traversé l'esprit, le site est relativement explicite concernant un portage sous l'OS de Redmond : « *No, we don't support Windows — get a better operating system* ».

CONCLUSION

Nous venons de faire un tour d'horizon des possibilités pratiques et ludiques gravitant autour de l'utilisation d'un récepteur GPS. Ceci ouvre la voie à bon nombre d'expérimentations et de codes à réaliser : représentation de la position des satellites via différentes bibliothèques graphiques, enregistrement de traces GPS, *tracking*, construction d'un système autonome de découverte de points d'accès Wifi, interfaçage avec des microcontrôleurs, etc. Libre à vous d'explorer tout cela via le décodage NMEA ou avec le protocole **gpsd**. ■

Analyse du Mir:ror

Auteur

- Frédéric Le Roy

Bon, dans cet article, vous n'allez pas apprendre de magie et contrôler le miroir de votre salle de bain pour vous embellir dès le matin... Le « miroir » que nous allons étudier ici est le périphérique commercialisé par la société Violet (si le nom vous dit quelque chose, vous devez déjà connaître le Nabaztag). Au travers de cet article, nous n'allons pas uniquement voir l'aspect technique (ce périphérique restant simple à utiliser), mais aussi la manière d'aborder l'analyse d'un matériel non connu.

1 DESCRIPTION ET PETIT HISTORIQUE DE LA « BÊTE »

1.1 À l'origine, il y avait le Nabaztag...

On ne peut parler du Mir:ror sans parler du Nabaztag **[nabaztag-1] [nabaztag-2]**, ce lapin qui communique via Internet avec les serveurs de Violet et peut vous donner des informations comme la météo, le cours de la bourse, etc. En plus de parler, il peut émettre des « messages » lumineux ou bouger ses oreilles. Il s'agit donc d'un objet communiquant, et ici, nous tombons dans le monde de l'Internet des objets **[internet-des-objets]**. Le gros point négatif de ce lapin est sa dépendance vis-à-vis des serveurs de la société Violet. En effet, pour bénéficier des services (météo, bourse, etc.), le lapin doit être connecté aux serveurs de Violet qui lui fournissent les données (et permettent également des interactions entre les lapins). Pas de bras, pas de chocolat... Plus d'Internet, plus de lapin...

1.2 Une pincée de libre dans le civet...

Heureusement pour le libre, une troupe de lapins décida de se rebeller et de s'affranchir des serveurs de Violet. Le projet OpenNab fut né **[opennab]**. Ce projet permet d'utiliser le lapin sans être dépendant des serveurs de Violet. Je ne suis par contre pas vraiment sûr que ce projet fonctionne avec les derniers lapins... Mais tout ceci est une autre histoire, notre sujet aujourd'hui étant le Mir:ror.

1.3 Un lapin radio amateur

Une seconde version du lapin fut lancée en 2006 : le Nabaztag:tag. Ce lapin est équipé (entre autres) d'un lecteur de puces RFID **[rfid]**. Afin d'utiliser ce lecteur, la société Violet a commercialisé également des Ztamp:s et des Nano:ztag. Les premiers sont des petits timbres de couleurs équipés d'une puce RFID. Ils peuvent être collés sur des objets et ainsi, lorsqu'on passe un objet devant le lapin, une action se réalise (encore faut-il avoir programmé une action). Les seconds sont des petits lapins aux oreilles détachables et contenant également une puce RFID. De la même manière, en passant ces lapinous devant un lapin, une action sera exécutée. Chaque Ztamp ou Nano:ztag a un identifiant unique, ce qui leur permet d'avoir chacun une fonction différente.

1.4 Et le mir:ror, on en parle un jour ou pas ?

Venons-en au Mir:ror **[mirror]**. Ce miroir se présente sous la forme d'une sorte de dessous de verre design agrémenté d'un câble usb. Ce miroir fait, contrairement au lapin, une seule chose, mais le fait bien : il s'agit en fait d'un lecteur RFID (et c'est tout).

Ce lecteur RFID peut être utilisé avec les Ztamp:s et les Nano:ztag dont nous avons parlé précédemment. D'ailleurs il est fourni avec 3 Ztamp:s (que nous appellerons timbres par la suite) et 2 Nano:ztag (les lapinous pour ceux qui ne suivent pas).

Le miroir, comme son cousin le lapin, est conçu pour être utilisé avec les serveurs de Violet, ce qui pour moi est un critère de non-achat. Nous allons donc voir dans cet article comment utiliser cet objet depuis Linux sans dépendre de qui que ce soit.

1.5 Une dernière parenthèse avant la partie intéressante

Avant de continuer, je tiens quand même à vous parler du projet Tuxdroid **[tuxdroid-1][tuxdroid-2]**. Il s'agit également d'un objet communiquant, que l'on peut comparer au Nabaztag sur de nombreux points, mais ce manchot-là a l'avantage d'être à l'image de Tux et surtout d'être bien plus orienté vers le monde du libre :).

2 C'EST BIEN BEAU TOUT ÇA, MAIS ÇA VA ME SERVIR À QUOI CE MIROIR ?

L'utilisation que vous pouvez faire du miroir ne dépend que de vous, mais voici quelques exemples en vrac :

- Vous avez un timbre collé sur votre porte-clés. Vous rentrez chez vous, posez vos clés sur le miroir et une source sonore vous diffuse les nouvelles du jour puis les rendez-vous du lendemain.
- Votre enfant de 2 ans veut écouter son CD préféré. Pour cela, il prend le boîtier du CD sur lequel est collé un timbre et le pose sur le miroir. Votre *media center* (libre de préférence) préféré va lancer automatiquement la liste de lecture correspondant à ce CD.
- Vous avez envie d'une ambiance romantique, vous posez le lapinou rose sur le miroir. Vos volets se ferment, la lumière se tamise et votre media center vous met une musique romantique.
- etc.

Bon, mes exemples sont très orientés domotique, mais je dois bien avouer que j'ai acheté un Mir:ror dans ce but aussi :). À vous de trouver les usages qui vous correspondent le mieux.

3 ANALYSE DE LA BÊTE

3.1 Comment ça fonctionne ?

Branchons le miroir à notre ordinateur. Il émet une petite musique et le pourtour se met à clignoter. Si on le retourne, il émet une autre musique et le pourtour s'éteint. En le remettant dans le bon sens, il émet à nouveau la première musique et le pourtour se remet à clignoter. Il semble donc que le fait de le retourner le met en veille (nous appellerons cette position la position passive).

Approchons un timbre du miroir en mode actif : une petite musique et un changement de la fréquence de clignotement nous apprend que le timbre est détecté. Éloignons le timbre du miroir : une nouvelle musique nous signale la sortie du timbre du champ de détection du miroir. Le clignotement reprend son rythme initial. Le même comportement est observé avec un lapinou.

3.2 Qu'en dit notre manchot ?

Notre manchot de test est une distribution Ubuntu 9.10. Le test a aussi été réalisé sous une Debian Squeeze.

Au branchement du périphérique, la commande **dmesg** nous indique :

```
[1330852.592096] usb 7-1: new full speed USB device using uhci_hcd and
address 4
[1330852.768579] usb 7-1: configuration #1 chosen from 1 choice
[1330852.778543] generic-usb 0003:1DA8:1301.0007: hiddev96,hidraw1: USB
HID v1.00 Device [Violet Mirror] on usb-0000:00:1d.1-1/input0
```

Le miroir est donc bien détecté. Au vu du message, il semble que le périphérique soit accessible en tant que **/dev/hidraw1** sur mon poste. Vérifions ceci :

```
$ ls -l /dev/hidraw1
crw-rw---- 1 root root 252, 1 2010-02-13 17:13 /dev/hidraw1
```

L'heure de création du fichier **/dev/hidraw1** correspond bien à l'heure à laquelle j'ai branché le miroir. Ouf ! Notons ici que seul root a les droits en lecture/écriture sur ce fichier. Je laisse à chacun d'entre vous gérer ce problème à votre manière pour que tout utilisateur puisse y accéder ;)

3.3 Étude du flux

Maintenant que nous savons où regarder, jetons un œil à ce qui se passe dans notre discussion avec le périphérique. Soyons fous et utilisons directement **cat** :

Pour l'action d'approcher un timbre puis de le retirer, nous obtenons :

```
$ cat /dev/hidraw1
#◆###R◆08#◆###R◆08
```

Il semble que ça ne soit pas une si bonne idée que ça : le flux n'est pas textuel. Passons à l'outil suivant, **hexdump**, et réalisons la même action :

```
$ hexdump /dev/hidraw1
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
00010e0 0102 0000 d008 1a02 5203 30c1 0038 0000
00010f0 0000 0000 0000 0000 0000 0000 0000 0000
*
0001540 0202 0000 d008 1a02 5203 30c1 0038 0000
0001550 0000 0000 0000 0000 0000 0000 0000 0000
*
```

Ici, le résultat est beaucoup plus probant. Il semble que les données émises par le miroir soient structurées par paquets de 16 bits et on voit déjà certaines redondances entre l'approche et l'éloignement du timbre. Nous pouvons donc déjà supposer ici que la première partie correspondrait à une action et la seconde à un identifiant de la puce. Avec ce point de départ, nous allons réaliser un petit script Python pour poursuivre notre analyse.

3.4 Première étape en Python : lecture du flux

Le script suivant lit en boucle le périphérique en mode bloc et affiche les données dès qu'elles ne sont pas nulles :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import binascii

# lecture du fichier en lecture avec le mode binaire
mirror = open("/dev/hidraw1","rb")
while 1:
    # on lit 16 byte
    donnee = mirror.read(16)
    # si la donne est non nulle
     if donnee != '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00':
        # affichage des données hexadécimales de manière lisible humainement
        print binascii.hexlify(donnee)
```

Utilisons ce script et testons plusieurs actions. Nous obtenons ces résultats :

On voit directement que les deux premiers bytes sont liés à l'action. C'est pour ça que je les ai directement séparés des autres bytes dans le tableau.

Nous avons donc les actions suivantes :

- **0105** : retournement du miroir ;
- **0104** : on remet le miroir dans le bon sens ;
- **0201** : on approche une puce RFID ;
- **0202** : une puce RFID sort du champ de détection du miroir.

Dans le cas des actions **0105** et **0104**, les bytes suivants sont toujours nuls. Dans le cadre des actions **0201** et **0202**, les bytes suivants semblent représenter l'identifiant de la puce RFID. Avec les puces que j'ai à disposition, certains bytes semblent valoir tout le temps 00 ou être identiques d'une puce à une autre. Dans le doute, et comme il n'y aura pas d'impacts, nous prendrons les 14 bytes en tant qu'identifiant de la puce RFID (qui peut le plus peut le moins :)).

Maintenant que nous avons décrypté le « Da Mir:ror Code », nous allons pouvoir améliorer notre script Python.

3.5 Seconde étape en Python : exécuter une commande en fonction du timbre ou du lapinou positionné

Nous allons améliorer notre script de manière à ce qu'il exécute les fichiers suivants en fonction des cas :

- retournement du miroir : **mirror-off.sh** ;
- remise dans le bon sens du miroir : **mirror-on.sh** ;
- approche d'une puce RFID : **rfid-<ifdentifiant>-in.sh** ;
- sortie d'une puce RFID du champ du lecteur : **rfid-<identifiant>-out.sh**.

Ainsi, les quatre actions pourront être gérées et nous pourrons programmer pour chaque Nano:ztag et Ztamp une action lorsqu'on les pose ou qu'on les retire.

Voici le code du script :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import binascii
import os

# noms des fichiers ou prefixes/suffixes pour les puces
mirror_on = "./mirror-on.sh"
mirror_off = "./mirror-off.sh"
```

Action	Bytes 1 à 2	Bytes 3 à 16
Retournement du miroir pour le mettre en mode passif	0105	00000000000000000000000000
Retournement du miroir pour le remettre en mode actif	0104	00000000000000000000000000
En mode actif, approche du lapinou gris	0201	000008d0021a0352c13038000000
En mode actif, éloignement du lapinou gris	0202	000008d0021a0352c13038000000
En mode actif, approche du lapinou vert	0201	000008d00218c10916a8a9000000
En mode actif, éloignement du lapinou vert	0202	000008d00218c10916a8a9000000
En mode actif, approche du timbre	0201	000008d00218c10916df50000000
En mode actif, éloignement du timbre	0202	000008d00218c10916df50000000

```
rfid_in_1 = "./rfid-"
rfid_in_2 = "-in.sh"
rfid_out_1 = "./rfid-"
rfid_out_2 = "-out.sh"

# lecture du fichier en lecture avec le mode binaire
mirror = open("/dev/hidraw1","rb")
while 1:
    # on lit 16 byte
    donnee = mirror.read(16)
    # si la donne est non nulle
    if donnee != '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00':
        # affichage des données hexadécimales de manière lisible humainement
        donnee_hex = binascii.hexlify(donnee)
        print "Donnée : %s" % donnee_hex
        # si action sur le miroir
        if donnee[0] == '\x01':
            # miroir en mode actif
            if donnee[1] == '\x04':
                print "Lancement de %s" % mirror_on
                os.system(mirror_on)
            # miroir en mode actif
            elif donnee[1] == '\x05':
                print "Lancement de %s" % mirror_off
                os.system(mirror_off)
        # si action liée à une puce RFID
        if donnee[0] == '\x02':
            # valeur de l'identifiant
            rfid_id = donnee_hex[4:]
            # puce en approche
            if donnee[1] == '\x01':
                print "Lancement de %s%s%s" % (rfid_in_1, rfid_id, rfid_in_2)
                os.system(rfid_in_1+rfid_id+rfid_in_2)
            # puce en sortie de champ
            if donnee[1] == '\x02':
                print "Lancement de %s%s%s" % (rfid_out_1, rfid_id, rfid_out_2)
                os.system(rfid_out_1+rfid_id+rfid_out_2)
```

J'ai créé les fichiers **mirror-on.sh**, **mirror-off.sh**, ainsi que les 2 fichiers **rfid-000008d00218c10916a8a9000000-in.sh** et **rfid-000008d00218c10916a8a9000000-out.sh** pour le lapinou vert en faisant en sorte que chacun de ces scripts shell affiche une ligne de texte. Voici ce que donne le test.

Tout d'abord, en mettant le miroir à l'envers, puis en le remettant à l'endroit :

```
Donnée : 0105000000000000000000000000000000
Lancement de ./mirror-off.sh
OFF!
Donnée : 0104000000000000000000000000000000
Lancement de ./mirror-on.sh
ON!
```

Maintenant, en posant puis retirant le lapinou vert :

```
Donnée : 0201000008d00218c10916a8a9000000
Lancement de ./rfid-000008d00218c10916a8a9000000-in.sh
Lapin vert en approche!
Donnée : 0202000008d00218c10916a8a9000000
Lancement de ./rfid-000008d00218c10916a8a9000000-out.sh
Le vert quitte le navire!
```

Puis avec le lapinou gris ::

```
Donnée : 0201000008d0021a0352c13038000000
Lancement de ./rfid-000008d0021a0352c13038000000-in.sh
sh: ./rfid-000008d0021a0352c13038000000-in.sh: Aucun fichier ou dossier de ce type
Donnée : 0202000008d0021a0352c13038000000
Lancement de ./rfid-000008d0021a0352c13038000000-out.sh
sh: ./rfid-000008d0021a0352c13038000000-out.sh: Aucun fichier ou dossier de ce type
```

Et voilà, notre script de gestion du miroir fonctionne :)

3.6 Dernière étape : finaliser

La dernière étape de l'utilisation du miroir, eh bien, je vous la laisse :). Elle va comprendre les actions suivantes :

- automatiser le démarrage de ce script ;
- créer vos propres scripts pour réagir en fonction des actions sur le miroir ou des puces RFID ;
- éventuellement améliorer le script que nous avons développé au cours de cet article. En effet, il existe plusieurs axes d'amélioration que nous n'avons pas vus ici, car j'ai choisi de me concentrer uniquement sur le Mir:ror et non sur la réalisation d'une vraie bibliothèque digne de ce nom en Python.

4 AVANT DE TOURNER LA PAGE...

Avant que vous ne tourniez la page pour lire le prochain article de cette merveilleuse revue, je vais vous dévoiler deux secrets à propos du miroir :

1. La norme ISO qu'utilise le lecteur est la norme ISO 14443A **[iso-14443a]**.
2. Il est possible d'acheter des puces RFID respectant cette norme et donc reconnues par le Mir:ror sans devoir forcément passer par la société Violet. Il en existe par exemple sous forme de porte-clés, ce qui peut être plus pratique que de coller un Ztamp sur un porte-clés.

Et pour finir, je tiens juste à préciser que le support du Mir:ror sera intégré dès les premières versions dans la solution libre de domotique Domogik (une première version devrait sortir avant la fin du premier semestre, mais ceci est une autre histoire ;)) ■

Auteur : Frédéric Le Roy

Références

- **[nabaztag-1]** http://www.nabaztag.com/fr/
- **[nabaztag-2]** http://fr.wikipedia.org/wiki/Nabaztag
- **[internet-des-objets]** http://fr.wikipedia.org/wiki/Internet_des_objets
- **[opennab]** http://opennab.sourceforge.net/
- **[rfid]** http://fr.wikipedia.org/wiki/Radio-identification
- **[mirror]** http://www.violet.net/_mirror-le-premier-lecteur-rfid.html
- **[tuxdroid-1]** http://fr.wikipedia.org/wiki/Tux_Droid
- **[tuxdroid-2]** http://www.kysoh.com/tux-droid
- **[iso-14443a]** http://en.wikipedia.org/wiki/ISO/IEC_14443

Développement pour iPod Touch sous GNU/Linux :

Auteurs

- G. Goavec-Mérou,
- J.-M Friedt

*Les iPod Touch et iPhone sont probablement les systèmes embarqués les plus largement disponibles auprès du grand public. Bien que le qualificatif « embarqué » ne soit associé qu'à leur autonomie, faible encombrement et absence de ports de communication, la puissance de calcul est compatible avec celle nécessaire pour exécuter un environnement Un*x dans lequel un développeur sous GNU/Linux ou *BSD sera familier. Nous proposons dans cette présentation, après avoir libéré son iPod Touch de l'emprise de Cupertino, d'exploiter une chaîne de compilation croisée libre exploitant les bibliothèques propriétaires mais gratuites d'Apple, pour développer nos propres applications. Nous nous intéresserons en particulier à l'accès aux périphériques matériels, qu'il s'agisse d'une liaison Bluetooth pour communiquer par une liaison sans fil avec des périphériques tels que la brique LEGO NXT, ou des accéléromètres. Ces développements passent nécessairement par la maîtrise de l'Objective-C 2.0, dont nous démontrerons la fonctionnalité comme langage de développement sous GNU/Linux lors de l'exploitation du compilateur LLVM.*

1 INTRODUCTION

L'*iPod Touch* est un ordinateur populaire sur lequel le développement ne peut se qualifier d'embarqué que du fait de la taille et de l'autonomie de la plate-forme. La puissance de calcul est en effet digne de tout ordinateur : basé sur un processeur d'architecture ARM (Cortex A8) cadencé à 600 MHz, l'iPod Touch que nous utiliserons pour nos développements propose 256 MB de RAM et un support de stockage de masse non volatil de plusieurs gigaoctets **[1]**. Le matériel nécessaire à l'exploitation des protocoles de communication sans fil wifi et Bluetooth est implémenté. Le système d'exploitation exécuté sur cette plate-forme - basé sur un noyau Mach intégré dans une version de MacOS X aux performances bridées pour s'ajuster à la puissance de calcul réduite par rapport aux autres plates-formes proposées par Apple - fournit un environnement Unix familier au lecteur. Nous avons expérimenté avec les versions 3.1.2 et 3.1.3 des *firmwares* imposés par Apple.

Notre objectif est d'exploiter cette plate-forme pour nous familiariser avec le développement sur ce système embarqué, en suivant le cheminement classique du développement sur une cible basée sur un processeur autre que celui de la plate-forme de développement : installation et mise en œuvre de la chaîne de compilation croisée, transfert du fichier binaire contenant la version exécutable de l'application sur le système embarqué, exécution et validation du bon fonctionnement. Cependant, le cas de l'iPod Touch va, dans un premier temps, se distinguer des projets n'exploitant que des outils libres (buildroot, par exemple) du fait de l'origine propriétaire de la plate-forme cible **[2]**. En effet, dans la continuité de sa philosophie de réduire le nombre de développeurs accédant aux ressources de ses ordinateurs, Apple bride la capacité d'exécuter des binaires issus de sources autres qu'iTunes. Notre première tâche consiste donc à libérer l'iPod de cette contrainte afin d'étendre la gamme de programmes exécutés par l'ordinateur aux programmes que nous développerons au cours de cette présentation (*jailbreak*).

application à la communication par liaison Bluetooth

2 LIBÉRER SON IPOD TOUCH

La configuration imposée par Apple est de n'autoriser que l'exécution de programmes issus de son dépôt iTunes. Cela signifie notamment que le développeur doit s'enregistrer (en échange d'une contribution financière) auprès d'Apple, et que la gamme des applications accessibles est celle mise à disposition après censure par les gestionnaires d'iTunes. Ainsi, une vaste gamme d'applications, et notamment un serveur OpenSSH pour se connecter par wifi à l'iPod, sont inaccessibles.

Dans l'incessante lutte stérile entre les développeurs de logiciels propriétaires et les *hackers* curieux de découvrir les outils en leur possession, un outil a été développé pour libérer l'iPod de la contrainte d'iTunes : Spirit est un outil fonctionnant sous tout système d'exploitation largement accessible - et notamment GNU/Linux - afin de patcher le système d'exploitation fourni par Apple sur iPod. Il s'agit d'une méthode qualifiée de *untethered*, signifiant que la modification n'est pas à refaire à chaque réinitialisation de l'iPod : le patch est appliqué en mémoire non volatile et s'applique jusqu'à la prochaine mise à jour du système d'exploitation. Notez qu'il s'agit uniquement de l'application d'un correctif sur le système d'exploitation existant : la version du système d'exploitation n'est pas modifiée par cette opération.

Spirit s'obtient à http://spiritjb.com et s'applique par l'exécution du *daemon* `usbmuxd` suivi de `spirit`. Noter que Spirit suppose qu'il s'applique sur un système d'exploitation vierge de toute modification antérieure sur l'iPod : nous conseillons donc de restaurer son iPod sur une version 3.1.3 du firmware (induisant la perte de toutes les données personnelles lors de la mise à jour) avant d'appliquer la procédure de libération. Par ailleurs, Spirit nécessite l'installation des bibliothèques permettant la communication avec iPod (libimobile).

Une fois l'iPod libre, un outil familier aux utilisateurs de Debian GNU/Linux est un gestionnaire de paquets nommé Cydia, installé automatiquement par Spirit. Une alternative, compatible avec cydia, que nous préférons pour des raisons d'ergonomie, est rock, disponible sous forme de paquet dans cydia. Dans tous les cas, les outils classiques de gestion des paquets `apt-*` sont fournis. Notre première tâche est donc d'étoffer la gamme de logiciels disponibles sur l'iPod : une console nommée MobileTerminal fournira un environnement proche de l'xterm classique (Fig. 1), et donne accès à tous les outils en ligne de commandes Unix disponibles dans divers paquets que nous laissons le soin au lecteur d'explorer (catégories *Networking* pour le serveur OpenSSH et les services associés inetutils, catégorie *Utilities* pour les applications classiques en console que sont *Core Utilities*, et évidemment *Development*, qui donne accès à *Developer Commands*, `diff` et autres outils fondamentaux). Bien que le compilateur libre `llvm-gcc` à destination de l'architecture ARM soit disponible comme paquet fonctionnel sur iPod, nous allons nous restreindre à la méthode classique de cross-compilation qui permet de séparer la phase de développement sur un PC classique (notamment muni d'un clavier, interface plus simple à manier que l'écran tactile pour coder), et l'exécution sur la plate-forme cible (ici l'iPod).

3 INSTALLATION DE LA CHAÎNE DE COMPILATION CROISÉE ET DES BIBLIOTHÈQUES PROPRIÉTAIRES APPLE

Le projet code.google.com/p/iphonedevonlinux propose d'exploiter le compilateur `llvm-gcc` - et plus exactement la branche LLVM **(1)**, qui est la seule à implémenter les méthodes d'Objective-C 2.0 (section 5) - à destination de la cible ARM comme outil libre d'une part, et les bibliothèques propriétaires fournies par Apple dans son outil XCode d'autre part, pour générer un environnement de travail capable de compiler un programme exécutable sur iPod. Le script `toolchain.sh` se charge a priori de rapatrier toutes les informations disponibles, mais une lecture de ce code et la compréhension des diverses étapes n'est pas inutile pour le faire fonctionner **(2)**. En effet, l'évolution permanente des archives nécessaires et la diversité des installations GNU/Linux exploitées rendent parfois le script inefficace.

Nous supposerons que l'installation de la chaîne de compilation croisée se déroule dans le répertoire `$IPOD`.

L'obtention de l'archive de compilation des outils - et notamment du script `toolchain.sh`, s'obtient depuis le dépôt `svn checkout http://iphonedevonlinux.googlecode.com/svn/trunk/ iphonedevonlinux-read-only`. Partant des instructions fournies à http://code.google.com/p/iphonedevonlinux/wiki/Installation, nous nous inspirons de `toolchain.sh` pour :

1. Rapatrier manuellement l'archive contenant les bibliothèques propriétaires du site de développeurs Apple **(3)** - disponible dans leur *Software Development Kit* (SDK) nommé XCode - après s'être enregistré

(gratuitement !) auprès d'Apple **(4)** pour obtenir un accès à leurs outils de développement propriétaires. À la date de rédaction de ce document (août 2010), l'archive disponible est celle pour le firmware 3.2.3 de l'iPod à destination de Snow Leopard, nommée **xcode_3.2.3_and_ios_sdk_4.0.1.img**. Ce fichier est placé dans le répertoire **$IPOD/files** de l'environnement de compilation.

2. Le script **toolchain.sh** s'attend à une archive avec une certaine structure, qui change avec les versions successives de XCode mises à disposition par Apple. Nous allons donc extraire manuellement les outils nécessaires depuis cette archive, en nous plaçant dans **$IPOD/files**, par :

```
mount -t hfsplus -o loop ../tmp/xcode_3.2.3_and_ios_sdk_4.0.1.img mnt
xar -xf mnt/Packages/iPhoneSDK3_2.pkg Payload cat Payload | zcat | cpio -id
```

De cette façon, nous obtenons un répertoire **Platforms** contenant les bibliothèques nécessaires à XCode 3.2.3.

3. Récupérer le firmware de son système embarqué - dans notre cas un iPod Touch 3G - à http://www.theiphonewiki.com/wiki/index.php?title=Firmware, et le placer dans le répertoire **files/firmware**. Bien que la version 3.2 du firmware ne soit pas disponible, nous obtiendrons une chaîne de compilation croisée fonctionnelle en téléchargeant la version 3.1.2, et ce, même si la version exécutée sur l'iPod Touch est 3.1.3.

4. L'exploitation de ce firmware nécessite une clé de décryptage. Celle-ci est disponible sur la page web accessible en cliquant sur le nom du *build* de firmware sélectionné, et est renseignée dans le champ **VFDecrypt**. Nous renseignons la variable **DECRYPTION_KEY_SYSTEM** dans le script **toolchain.sh** avec cette chaîne de caractères.

5. Une fois tous ces fichiers rapatriés et placés dans les bons répertoires, il ne reste plus qu'à lancer **sh ./toolchain.sh** pour débuter la compilation de **llvm-gcc** à destination de l'iPod : les archives et patchs sont automatiquement rapatriés. Ainsi, autant la phase de mise en place des outils propriétaires est fastidieuse car la nomenclature des fichiers change avec les versions de XCode et des firmwares, autant la phase de compilation des outils libres, et notamment de **llvm-gcc**, ne pose aucun problème.

6. En fin de compilation, on prendra encore soin de ne pas effacer les sources et archives intermédiaires qui ont été nécessaires. En effet, il nous faut encore déplacer deux répertoires d'une archive vers l'environnement de travail, faute de quoi les bibliothèques nécessaires à l'édition de liens (et notamment **libobjc**) ne seront pas accessibles :

```
cp -r $IPOD/sdks/iPhoneOS3.1.2.sdk/System $IPOD/toolchain/sys
cp -r $IPOD/sdks/iPhoneOS3.1.2.sdk/usr/lib $IPOD/toolchain/sys/usr
```

7. Enfin, on ajoutera le répertoire contenant les exécutables, **$IPOD/toolchain/pre/bin/** à son **PATH**.

Selon les distributions de GNU/Linux, la compilation peut buter sur des en-têtes implicites qu'il faut expliciter : toute référence manquante à printf ou fonctions d'entrée/sortie en général sont dues à l'absence de l'inclusion des fichiers d'en-têtes **stdio.h** et **stdlib.h** dans les sources en C++. Au final, un espace libre de plus de 8,5 GB est nécessaire sur le disque pour compléter cette installation.

Une validation simple de leur fonctionnement consiste en la compilation des exemples dans le répertoire **apps/**. Un exemple de la compilation de HelloToolchain fonctionnelle est :

```
$ make
arm-apple-darwin9-gcc -c src/HelloToolchain.m -o HelloToolchain.o
arm-apple-darwin9-gcc -lobjc -bind_at_load -framework Foundation -framework
CoreFoundation -framework UIKit -w -o HelloToolchain HelloToolchain.o
```

qui génère le répertoire **HelloToolchain.app** contenant un script intermédiaire nécessaire à l'exécution sur iPod (**HelloToolchain**) et le binaire proprement dit (**HelloToolchain_**).

Un point fondamental à ne pas négliger après avoir généré l'exécutable est la phase de signature : bien que nous ayons libéré notre iPod de la nécessité de se limiter aux programmes accessibles sur iTunes, il faut néanmoins signer les fichiers binaires afin de les rendre exécutables. Un outil disponible aussi bien sur iPod que sous GNU/Linux est **LDID** :

- Afin de signer son binaire sur la plate-forme de compilation, nous récupérons l'outil approprié à http://svn.telesphoreo.org/trunk/data/ldid, qui se compile par **g++ -I . -o util/ldid{,.cpp} -x c util/{lookup2,sha1}.c** et nécessite l'accès à un utilitaire de la chaîne de compilation défini par **export CODESIGN_ALLOCATE=$IPOD/toolchain/pre/bin/arm-apple-darwin9-codesign_allocate**.
- Installer le paquet **ldid** sur son iPod depuis **cydia** ou **rock**, et signer le programme exécutable depuis l'iPod après transfert du binaire depuis l'hôte.

Dans tous les cas, la signature du programme **prog** s'obtient par **ldid -S prog**. Dans le cas des applications graphiques qui contiennent un *wrapper* sous forme de script shell et un exécutable sous forme de binaire dont le nom finit par **_**, nous ne signerons que le binaire.

L'ensemble des codes sources des programmes présentés dans ce document, compilables au moyen de cette chaîne de compilation croisée, est disponible à www.trabucayre.com/lm/lm_ipod_sources.tgz ainsi qu'à jmfriedt.free.fr.

4 PREMIER PAS : APPLICATION EXPLOITANT L'INTERFACE POSIX

Nous avons auparavant installé une console nommée MobileTerminal. Cet outil fournit une interface en mode texte et donc l'accès aux commandes classiquement disponibles sous Un*x. Le corollaire évident est la capacité à exploiter la compatibilité POSIX de iPod OS pour générer des programmes n'exploitant pas les fonctions graphiques de l'iPod mais

uniquement l'interface en mode texte : ces programmes seront soit exécutables depuis la console, soit depuis un lien SSH au travers d'une liaison wifi entre un ordinateur et l'iPod.

Exemple de programme exploitant l'API POSIX :

```
#include <stdio.h>

int main(void)
{
        printf("Ipod : Hello World\n");
        return 0;
}
```

Une fois compilé pour iPod au moyen de :

```
arm-apple-darwin9-gcc -c -o helloWorld.o helloWorld.c
arm-apple-darwin9-gcc -bind_at_load -w -o hello helloWorld.c
```

suivi de l'installation automatique par :

```
@scp -rp hello root@$(IP_IPOD):/usr/local/bin
@ssh root@$(IP_IPOD) "cd /usr/local/bin ; ldid -S hello; killall SpringBoard"
```

nous obtenons le résultat présenté sur la figure 1 lors de l'exécution du programme dans la console MobileTerminal. Bien que dans la suite de ce document, toutes les figures soient des captures d'écran de l'iPod (outil **uishoot** exécuté depuis une session SSH), tous les exemples ont été exécutés sur une plate-forme matérielle iPod Touch 3G et non sur un émulateur : les captures d'écran au lieu de photos ont pour vocation d'améliorer la qualité graphique des illustrations.

Nous avons donc atteint un stade où nous sommes capables de compiler une application en mode texte, écrite en C, compatible POSIX, au moyen d'une chaîne de compilation croisée fonctionnelle sur un environnement de développement libre.

Au-delà de la démonstration d'un exemple trivial d'affichage, le support de la norme POSIX implique la capacité à compiler et exécuter tout programme ne nécessitant pas d'interface graphique sur iPod. Un cas concret est la cross-compilation du

Figure 1 : Exécution d'une application POSIX dans le MobileTerminal de l'iPod Touch, application compilée au moyen de la toolchain mise en place sur plate-forme x86.

simulateur de circuits électroniques SPICE, et sa version libre ngspice, dont les sources sont disponibles à ngspice.sourceforge.net. La configuration de la cross-compilation se fait classiquement par **./configure --host=arm-apple-darwin9**, mais quelques ajustements sont nécessaires aux fichiers résultants :

- dans le fichier **config.h**, désactiver les définitions de **rpl_malloc** et **rpl_realloc** ;
- dans le fichier d'en-tête **src/include/ngspice.h**, définir la fonction de vérification si un résultat est infini : **#define finite isnormal** qui n'est définie que si la condition **_MSC_VER** est vérifiée.

Une fois ces corrections effectuées, **make** exploite la chaîne de compilation pour générer un certain nombre d'exécutables dans le sous-répertoire **src**, dont les plus utiles

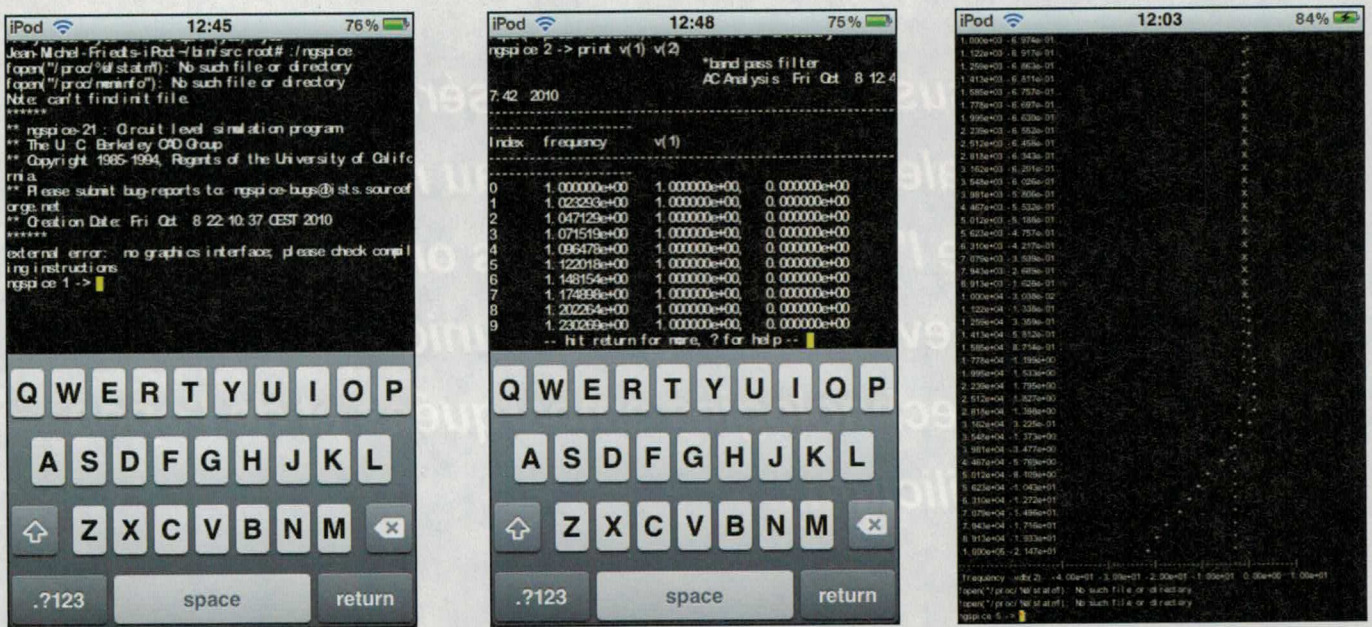


Figure 2 : Exécution de ngspice sur iPod Touch dans le MobileTerminal : à gauche, le lancement du logiciel de simulation du comportement de circuits électroniques, et au milieu, la solution pour deux potentiels d'un circuit après simulation. Malgré l'absence d'interface graphique pour SPICE, la sortie en ASCII au moyen de asciiplot est exploitable.

sont **ngspice** et **nutmeg**. Nous vérifions sur un exemple trivial le bon fonctionnement du simulateur, qui doit évidemment se contenter d'une sortie dans un fichier pour exploitation ultérieure, l'interface graphique n'étant pas portable sur iPod (Fig. 2). Le calcul est néanmoins pertinent puisque la fonction d'affichage **asciiplot** fournit une sortie exploitable sur un terminal en mode texte, et la redirection du résultat (**print variable > fichier**) est fonctionnelle.

Quelques subtilités subsistent quant au portage d'applications POSIX sur iPod Touch. Mentionnons par exemple l'application **sattrack** (Fig. 3), une application de prévision de passage de satellites réellement utile **[10]** sur ce support pour identifier les objets mobiles visibles dans le ciel après le coucher du soleil - qui nécessite quelques modifications pour être exploitable :

- modifier le compilateur **CC=arm-apple-darwin9-gcc** dans le **Makefile**, avec la configuration FreeBSD, pour compiler sur architecture ARM ;
- corriger le bogue de l'an 2000 en modifiant la ligne 272 de **sattime.h** (remplacer 100 par 200 dans la comparaison) ;
- modifier l'appel aux signaux **SIGALARM** de la fonction **milliSleep()** dans **sattime.c** pour exploiter la fonction **usleep(1000*msec)** qui fonctionne sur iPod au lieu d'une gestion manuelle du *timer*. Ce dernier point semble impliquer que la gestion des signaux n'est pas compatible POSIX sur iPod.

On obtient alors une application utile et fonctionnelle : en copiant l'intégralité du répertoire **Sattrack** dans **/var/mobile** (utilisateur par défaut) ou **/var/root**, le logiciel sera capable de retrouver ses fichiers de données, et notamment **data/cities.dat** (noter dans ce fichier que les longitudes à l'Est du méridien 0 sont négatives) et **data/default0.dat** pour la configuration par défaut (ville d'observation, satellite d'intérêt, paramètres orbitaux) qui évite l'insertion fastidieuse de ces informations par le clavier du MobileTerminal.

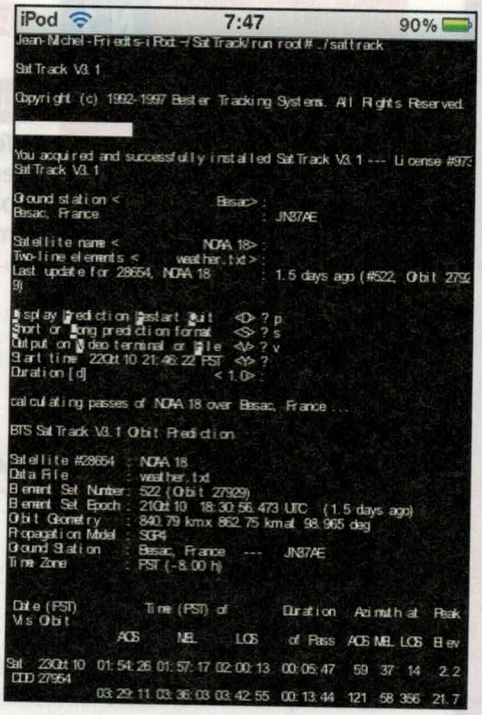


Figure 3 : Sattrack - un logiciel de prévision de passage de satellites open source jusqu'à sa version 3.1.6 malgré une licence très restrictive - fonctionne en mode texte (MobileTerminal) sur iPod avec la gestion de la position du curseur au moyen des caractères de contrôle compatibles avec un terminal VT100. Cette application permet d'identifier les satellites visibles dans le ciel après le coucher du soleil, ou de prédire le prochain passage d'un satellite météorologique. Dans cet exemple, nous avons obtenu les paramètres orbitaux les plus récents sur www.celestrak.com et calculé les dates de passage du satellite en orbite basse polaire NOAA 18.

Une application fonctionnant dans un terminal ne répond cependant pas aux exigences de la majorité des utilisateurs d'iPod et iPhone qui s'attendent à des applications graphiques. L'accès aux ressources matérielles des plates-formes Apple, qu'il s'agisse de l'interface graphique ou de périphériques tels que les accéléromètres, nécessite de passer par les bibliothèques propriétaires fournies avec XCode. Pour ce faire, un langage dédié est imposé : Objective-C **[3] [4]**.

5 ACCÈS AUX RESSOURCES DE L'IPOD : OBJECTIVE-C ET COCOA

5.1 L'objectiveC

L'utilisation du C a permis de valider l'ensemble des étapes nécessaires au développement sur iPod, mais la ligne de commandes ne répond pas nécessairement aux besoins de l'utilisateur, et ce langage ne permet pas d'exploiter les bibliothèques de développement d'interfaces graphiques.

En effet, les interfaces graphiques pour plates-formes Apple utilisent la bibliothèque COCOA, qui étant écrite en *Objective-C*, va nous contraindre à appréhender ce langage. La présentation qui suit n'est pas un cours complet, mais juste le strict nécessaire pour comprendre les applications qui suivront. Il existe bon nombre de documents, que ce soit sur Internet ou sur papier, pour aller plus en avant dans cet apprentissage **[5] [6]**.

Objective-C est un langage orienté objet. Plutôt que reprendre son historique complet qui se trouve un peu partout sur le Web **(5)**, contentons-nous de mentionner que son développement est issu d'une société nommée NextStep qui justifie le préfixe NS de nombreux objets fournis par les bibliothèques, et qu'un ensemble d'outils libres sous GNU/Linux exploitant les fonctionnalités de ce langage se retrouve dans GNUStep et l'environnement de fenêtrage Window Maker.

Ce langage dispose d'un *garbage collector* permettant la suppression automatique des objets qui ne sont plus référencés par d'autres. Pour ce faire, chaque instance possède un compteur qui permettra, lorsqu'il arrive à 0, de supprimer l'objet. Lors de la création, il est mis à 1, ainsi que quand la méthode **retain** est appelée (généralement lorsqu'un objet reçoit un pointeur sur l'autre objet). Le compteur est décrémenté par l'appel à sa méthode **release** (à la destruction d'un objet ayant une référence dessus, ou lorsque l'objet n'est plus nécessaire dans l'application).

Comme pour le C++, ce langage utilise un fichier d'en-tête ainsi qu'un fichier d'implémentation.

5.1.1 Le fichier d'en-tête

Ce fichier, classiquement, porte une extension en **.h**. Il peut contenir deux types :

- le type protocol (**@protocol**), qui correspond au type interface en Java ou à une classe virtuelle pure en C++. Il sert à définir les signatures pour une ou plusieurs méthodes sans en fournir la mise en œuvre. Une classe déclarée comme implémentant un protocole devra donc contenir le corps des méthodes du protocole visé ;
- le type interface (**@interface**), qui peut être considéré comme la vue externe d'une classe.

```
#import <objc/objc.h>
#import <objc/Object.h>

@interface my_class : Object
{
        int nb;
}
- (void)sayHello;
@property (nonatomic,assign) int nb;
@end
```

Listing 1 : Exemple d'un fichier d'en-tête

Le listing 1 présente un en-tête simple :

- l.1-l.2, **import** est l'équivalent en C de **include**. Cette instruction s'assure en plus que le fichier ne soit pas inclus plus d'une fois.
- l.4, une interface nommée **my_class** est définie. Elle hérite de la classe **Object**.
- l.5 est défini un membre pour cette classe. Comme dans la plupart des langages orientés objets, chaque variable a une certaine portée, permettant ainsi d'autoriser ou d'interdire leur modification depuis l'extérieur de l'instance de l'objet. Par défaut, ils sont de type **protected**, mais peuvent être :
  - privés (**@private**) ;
  - protégés (**@protected**) ;
  - publics (**@public**).
- l.8, définit une méthode. Là encore, il y a une différence vis-à-vis du C : une méthode en Objective-C est décrite de la façon suivante :

```
-/+(type_de_retour) nom:(type_param1)param1 nom_param2:(type_param2) param2;
```

Le signe devant le type de retour définit si la méthode est d'instance (signe **+**) ou de classe (signe **-**).

- l.9, l'instruction **@property** a pour but d'éviter au développeur la contrainte d'avoir à coder les méthodes d'accès aux attributs d'une classe. En effet, afin de donner la possibilité depuis l'extérieur d'un objet de lire et/ou de modifier la valeur d'une variable, le développeur se voit contraint d'utiliser la technique des accesseurs, avec les *getter* pour lire une variable, et les *setter* pour la modifier. Dans le cas de l'Objective-C, l'instruction **@property** va permettre, au moment de la compilation, de générer ces deux types de méthodes, selon les attributs fournis entre parenthèses. Dans le cas présent, l'accès à ce membre se fait d'une manière non atomique (**nonatomic**), c'est-à-dire sans se préoccuper d'un accès concurrent à la méthode et la valeur passée à la méthode sera copiée dans **nb**. Il existe d'autres options telles que **retain** pour l'accès aux objets, qui ne fera pas une copie mais affectera une référence à l'objet passé à la méthode, ainsi que l'incrémentation du nombre de pointeurs sur celui-ci.

5.1.2 Fichier d'implémentation

Ce fichier possède une extension en **.m**.

Il a pour rôle de fournir les implémentations pour l'ensemble des méthodes définies dans l'interface, ainsi que celles de protocoles si la classe en implémente.

```
#import "my_class.h"
@implementation my_class:Object
@synthesize nb;
- (void)sayHello {
        printf("Hello, %d!\n", nb);
}
@end
```

Listing 2 : Exemple d'un fichier d'implémentation : fichier my_class.m

L'ensemble des méthodes qui doivent être implémentées sont englobées par un **@implementation nom_classe** et un **@end**.

Le mot-clé **@synthesize** l.3 est le pendant de **@property**. Il peut être considéré comme une macro de génération du getter et du setter, le contenu dépendant des instructions spécifiées par **@property**.

5.1.3 Exemple d'utilisation d'une classe

```
#import <objc/objc.h>
#import <objc/Object.h>
#import "my_class.h"

int main(void)
{
        my_class *mc = [[my_class alloc] init];
        [mc setNb:1];
        [mc sayHello];
        mc.nb=2;
        [mc sayHello];
        return 0;
}
```

Listing 3 : Instanciation et utilisation d'une classe : fichier main.m

Le listing 3 présente une fonction principale (`main`) classique qui instancie un objet de type `my_class` (défini dans le listing 2).

Les lignes 8 à 11 font des appels à des méthodes de l'objet `mc`. L'appel à une méthode se fait en donnant le nom de l'objet suivi d'un `:` et de son paramètre, dans le cas où plusieurs paramètres sont fournis, chacun des suivants se fera en donnant le nom du paramètre suivi d'un `:` puis de la variable ou l'objet.

Les lignes 8 et 10 sont identiques : la première appelle explicitement la fonction automatiquement générée à la compilation, alors que la ligne 10 utilise implicitement cette même fonction.

5.1.4 Compilation sur GNU/Linux

Il est possible sur un poste sous GNU/Linux de compiler et d'exécuter du code écrit en Objective-C. Ceci se fait à l'aide de LLVM compilé pour plate-forme x86 avec le support pour Objective-C2.0 (cette configuration n'est pas disponible par défaut sous Debian).

Dans le cas de l'exemple précédemment présenté, la compilation se fera au moyen de la ligne suivante, le résultat étant un binaire fonctionnel sous GNU/Linux :

```
llvm-gcc main.m my_class.m -lobjc
```

dont le résultat est :

```
gwe@linux objc_example $ ./objc_example
Hello, 1!
Hello, 2!
```

Bien que fonctionnelle, l'implémentation disponible sous LLVM n'est pas entièrement compatible avec la version fournie par Apple. Il n'est, par exemple, pas possible d'utiliser les `property` avec le mot-clé `retain` sur un objet type `NSString` sans avoir modifié la classe pour y ajouter les méthodes nécessaires. À n'en pas douter, les versions ultérieures de LLVM combleront ces lacunes.

5.1.5 Listener et Delegate

Avant de passer à des applications relatives à l'iPod, il est encore nécessaire de présenter une notion qui, sans être spécifique au langage, est omniprésente dans la plupart des applications graphiques et dont la compréhension sera nécessaire par la suite pour l'exploitation de certaines classes liées au Bluetooth.

Imaginons que nous réalisions une classe `A` qui va contenir une instance d'un `NSString`. Depuis la classe `A`, il sera possible de remplir ou de changer le contenu de la chaîne de caractères ou d'en connaître la taille, i.e. d'établir une relation unidirectionnelle de type maître-esclave.

Si au lieu d'avoir une chaîne de caractères, notre classe `A` contenait une instance d'une autre classe implémentant un quelconque protocole ou tout simplement un bouton, notre classe pourrait modifier des informations, mais ne serait jamais en mesure d'être avertie des événements reçus par l'instance qu'elle contient. La seule solution serait de régulièrement interroger l'instance afin de vérifier qu'il n'y a pas de message en attente.

Le développement sur plates-formes Apple se base généralement sur un découpage de l'application entre plusieurs types de classes :

- celles qui gèrent l'interface graphique ou des composants graphiques particuliers ;
- celles qui gèrent le stockage et la mise en forme des données et qui sont par extension la source des données de l'affichage ;
- celles qui contrôlent le comportement des sources de données et interagissent avec l'interface graphique afin de recevoir les événements, comme l'appui sur un bouton, ou qui forcent la remise à jour de l'interface.

C'est dans cette optique que l'Objective-C dispose des notions de `Listener` et de `Delegate`.

Dans ce concept, le premier (`Listener`) sert à recevoir les informations ou les événements issus d'une classe ou d'une base de données, par exemple. Le second sert à gérer certains comportements spécifiques qui ne sont pas codés en dur dans la classe par souci de généricité.

Pour ce faire, la classe ayant besoin de faire remonter des informations va contenir une instance d'un objet de type non déterminé, mais implémentant un protocole particulier. Ainsi, la classe `A` précédemment codée peut, en s'inscrivant (en passant un pointeur sur elle-même) auprès de la nouvelle classe, recevoir les informations qui jusqu'alors n'étaient pas trivialement disponibles. L'utilisation d'un pointeur générique, mais qui implémente un certain protocole, permet l'exploitation de la classe événementielle sans pour autant la lier en dur à la classe englobante en donnant un type particulier qui empêcherait la réutilisation ultérieure du code.

5.2 Application graphique sur iPod

La maîtrise de la programmation en Objective-C permet d'aborder le développement d'interfaces graphiques sur iPod. À titre de premier exemple, le listing 2 présente la création d'un champ de texte contenant le message classique du premier exemple, afin de générer l'application illustrée sur la figure 4.

Le point d'entrée d'une application graphique sur iPod est exactement le même que pour une application, à savoir une fonction `main` :

```
int main(int argc, char **argv) {
    NSAutoreleasePool *autoreleasePool = [
        [ NSAutoreleasePool alloc ] init
    ];

    int returnCode = UIApplicationMain(argc, argv, @"HelloCocoa",
    @"HelloCocoa");
    [ autoreleasePool release ];
    return returnCode;
}
```

Listing 4 : Chargement d'une application graphique : source HelloCocoa_main.m

- Les lignes 2 à 4 présentent l'initialisation du garbage collector qui aura la charge de la suppression des objets qui ne seront plus utilisés dans l'application.
- La ligne 6 présente le chargement de l'application graphique en fournissant le nom de la classe qui constitue le cœur de l'application. Cette fonction est bloquante tant que l'application n'a pas quitté.
- La ligne 7 force la destruction de l'objet **autoreleasePool**.

Le reste du traitement se passe au niveau de la classe dont le nom a été fourni à **UiApplicationMain**.

```
@implementation HelloCocoa

- (void)applicationDidFinishLaunching: (UIApplication *) application {
    window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
    bounds]] autorelease];

    CGRect windowRect = [ [ UIScreen mainScreen ] applicationFrame ];
    windowRect.origin.x = windowRect.origin.y = 0.0f;

    window.backgroundColor = [UIColor whiteColor];

    CGRect txtFrame = CGRectMake(50,150,150,150);
    UITextView *txtView = [[UITextView alloc] initWithFrame:txtFrame];
    txtView.text = @"Hello Cocoa";
    UIFont *font = [UIFont boldSystemFontOfSize:18.0];
    txtView.font = font;

    [ window  addSubview: txtView ];
    [txtView release];
    [window makeKeyAndVisible];
}

@end
```

Listing 5 : Code source permettant de générer une interface graphique contenant un unique champ de texte : source HelloCocoa.m.

Lorsqu'une application est lancée et que son chargement en mémoire est fini, la fonction **applicationDidFinishLaunching** est appelée, c'est donc le point de départ pour la création de l'interface graphique :

- À la ligne 4 est créée la fenêtre : bien que l'iPod ne soit pas multifenêtre, il est nécessaire de définir cette zone dans laquelle il sera ensuite possible de dessiner.
- La ligne 9 attribue la couleur blanche au fond de l'interface.
- La ligne 11 crée un rectangle positionné aux coordonnées x=50, y=150 et de taille 150*150.
- Les lignes 12 à 15 configurent un champ de texte qui aura comme taille et position celle définie par le rectangle précédemment cité, un contenu par défaut (l.13) défini à **Hello Cocoa** (**@** précise que la chaîne est de type **NSString**), et une police (l.14 et 15) de 18 en gras.
- Une fois les composants graphiques de l'interface instanciés et configurés, il ne reste plus qu'à les insérer dans la fenêtre (l.17), le compteur sur l'objet **txtView** est décrémenté (l.18) afin que le garbage collector puisse faire son œuvre lors de la destruction de la fenêtre à la fermeture de l'application.
- Finalement, l.19, la commande **makeKeyAndVisible** est appelée. Cette fonction va lancer l'affichage de la fenêtre ainsi que la boucle qui gère les événements tels que l'appui sur l'écran et la fermeture de l'application.

Une fois le code réalisé, il faut le compiler et l'installer sur l'iPod (listing 6).

```
[...]
LDFLAGS=          -lobjc -bind_at_load -framework Foundation \
                  -framework CoreFoundation -framework UIKit
[...]
bundle: HelloCocoa
        @mkdir -p HelloCocoa.app
        @cp HelloCocoa HelloCocoa.app/HelloCocoa
        @cp Info.plist HelloCocoa.app
install: bundle
        @ssh root@$(IP) "cd /Applications/HelloCocoa.app && rm -R * ||
        echo 'not found' "
        @scp -rp HelloCocoa.app root@$(IP):/Applications
        @ssh root@$(IP) "cd /Applications/HelloCocoa.app ; ldid -S
        HelloCocoa_; killall SpringBoard"
```

Listing 6 : Makefile type pour la compilation d'une application graphique

Par rapport à l'exemple POSIX, le Makefile est un peu plus complexe :

- Il est nécessaire d'ajouter des frameworks pour les bibliothèques graphiques lors de l'édition de liens (l.2 et 3).
- L'application exploitable n'est pas simplement un binaire mais un répertoire dont l'extension est en **.app** (l.6), qui contient à la fois le binaire (l.7), mais aussi un fichier en **.plist** (l.8) qui définit diverses informations telles que la version et le nom de l'application, mais aussi si l'application est en plein écran (pas de barre en haut de l'écran).
- Lors de l'installation, ce sous-répertoire doit se trouver dans le répertoire **Applications** (l.11 et 12), et il est nécessaire de forcer le redémarrage de **Springboard** (qui gère le bureau) pour que la nouvelle application soit prise en compte et son icône affichée.

Au terme de la création, de la compilation et de l'installation de notre application, nous pouvons voir (non sans une certaine fierté) une fenêtre telle que sur la figure 4.

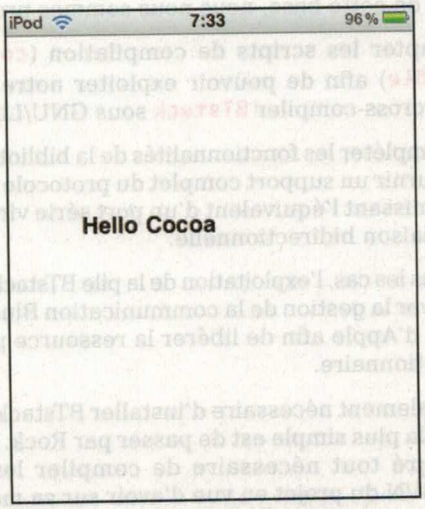


Figure 4 : Exemple d'interface graphique contenant un unique champ de texte

6 LIAISON BLUETOOTH

Une pile libre de liaison Bluetooth **[7]** a été développée, principalement par M. Ringwald, et son code source mis à disposition à http://code.google.com/p/btstack (Fig. 5). Cette pile est compatible avec les API POSIX et COCOA, tel que nous l'illustrerons ci-dessous, mais surtout donne accès à toutes les étapes de gestion des paquets et donc permet d'étendre la capacité de communication de l'iPod au-delà des périphériques qualifiés de compatibles par Apple.

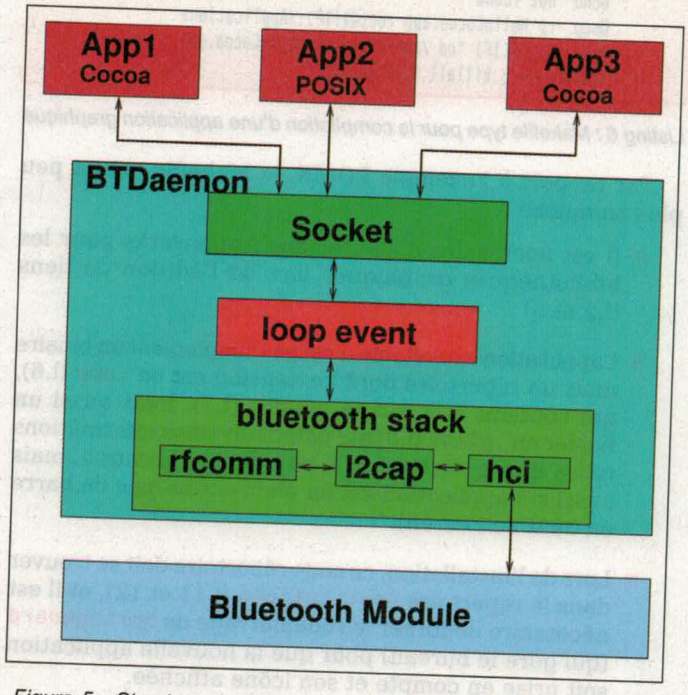


Figure 5 : Structure de BTstack, fournissant BTDaemon chargé de l'interface entre la matériel (géré par le système d'exploitation d'Apple) et les applications utilisateurs, ainsi que les bibliothèques de gestion des trames Bluetooth - nommées BTStackManager - pour COCOA.

Partant de cette base, nous nous sommes proposés :

- d'adapter les scripts de compilation (**configure** et **Makefile**) afin de pouvoir exploiter notre *toolchain* pour cross-compiler **BTstack** sous GNU/Linux ;
- de compléter les fonctionnalités de la bibliothèque afin de fournir un support complet du protocole RFCOMM - fournissant l'équivalent d'un port série virtuel - pour une liaison bidirectionnelle.

Dans tous les cas, l'exploitation de la pile BTstack nécessite de désactiver la gestion de la communication Bluetooth par le système d'Apple afin de libérer la ressource pour notre propre gestionnaire.

Il est également nécessaire d'installer BTstack sur iPod : la solution la plus simple est de passer par Rock. Toutefois, il est malgré tout nécessaire de compiler les sources issues du SVN du projet en vue d'avoir sur sa machine les bibliothèques qui seront nécessaires pour la compilation de ses propres applications.

La procédure est très classique :

- Télécharger les sources **svn checkout http://btstack.googlecode.com/svn/trunk**.
- Dans le répertoire nouvellement créé, faire un **./bootstrap.sh** suivi d'un :

```
./configure --target=iphone --with-sdk-version=SDK_Vers \
--with-gcc-version=GCC_Vers \
--prefix=/somewhere/ipod/btstack \
--with-developer-path=/somewhere/ipod/toolchain/ \
--enable-launchd
```

pour configurer les sources (au moment de la rédaction, nous utilisons 3.2 pour GCC_Vers et 4.2.1 pour SDK_Vers). Le point important est le **--enable-launchd**, qui va permettre de ne lancer **BTDaemon** que lorsqu'une application en a besoin, **launchd** ayant un rôle équivalent à **inetd** pour les connexions réseau entrantes. Il est envisageable de devoir modifier certaines occurrences de sdk en minuscules vers des SDK en majuscules lors de l'exploitation du script **configure** destiné à MacOS sous GNU/Linux.

- Une fois la configuration finie, un simple **make && make install** suffit pour compiler et installer les bibliothèques et les binaires. En cas de problème concernant libstdc++, vérifier le lien symbolique dans **$IPOD/toolchain/sys/usr/lib** : la bibliothèque **libstdc++.dylib** doit pointer sur **libstdc++.6.dylib**, sinon créer le lien symbolique manuellement.

Bien que nous ayons recompilé **BTDaemon**, nous sommes surtout intéressés par la disponibilité des bibliothèques associées à **BTstack**, et nous nous contentons d'exploiter le paquet **cydia** ou **rock** de **BTstack** pour placer une version fonctionnelle de **BTDaemon** et des scripts associés aux emplacements appropriés de l'arborescence. Pour ceux souhaitant ne pas utiliser la version disponible par Cydia/Rock, le script **package.sh** contenu dans le SVN permettra de générer le **.deb** ou sera source d'inspiration pour une installation totalement manuelle.

Dans le cas de la modification de **BTDaemon** en lui-même, le remplacement par une version personnelle se fera par écrasement du binaire (**BTDaemon**) et de la bibliothèque (**libBTstack.dylib**) sur l'iPod par les nouvelles versions, puis par l'exécution des commandes :

```
/bin/launchctl unload /Library/LaunchDaemons/ch.ringwald.BTstack.plist
/bin/launchctl load /Library/LaunchDaemons/ch.ringwald.BTstack.plist
```

afin de s'assurer que la nouvelle version sera à l'avenir bien prise en compte.

6.1 Premiers essais : RFCOMM

RFCOMM **(6)** est un protocole de communication au-dessus de Bluetooth pour créer une liaison de type point à point rappelant le port série virtuel. Bien que d'un débit réduit, il répond à la majorité de nos besoins pour exploiter

l'iPod comme terminal d'affichage et de contrôle de systèmes autonomes à consommation réduite, ce protocole est notamment implémenté dans les convertisseurs RS232-Bluetooth Free2Move **(7)** et dans la brique LEGO NXT **(8)**, que nous exploiterons pour illustrer nos développements.

Afin de valider le bon fonctionnement de la liaison, nous nous proposons d'effectuer quelques échanges simples depuis la console MobileTerminal ou, plus simple à manipuler, depuis une connexion SSH. Nous supposerons l'ordinateur personnel équipé d'une interface Bluetooth fonctionnelle (dans notre cas, un eeePC équipé d'un convertisseur USB-Bluetooth Belkin F8T012). Les adresses Bluetooth des périphériques s'obtiennent par `hcitool scan` pour identifier l'iPod, et `hcitool dev` pour obtenir l'identifiant du PC (argument `MAC_BT` qu'il faudra fournir à RFCOMM fonctionnant sur l'iPod). Une fois ces identifiants acquis, la liaison est établie en effectuant sur le PC les commandes :

```
sdptool add SP
rfcomm listen 4
```

et, sur iPod :

```
rfcomm -a MAC_BT -c 1
cat < /tmp/rfcomm0
```

La connexion est établie lorsqu'un canal est défini par les messages `Connection from MAC_IPOD to /dev/rfcomm1` sur le PC (en supposant une liaison sur le canal 1 - option `-c` de `rfcomm` sur iPod) et `Got 1 credits, can send!` sur iPod. Le message est expédié depuis le PC par un `echo Message > /dev/rfcomm1` (Fig. 6).

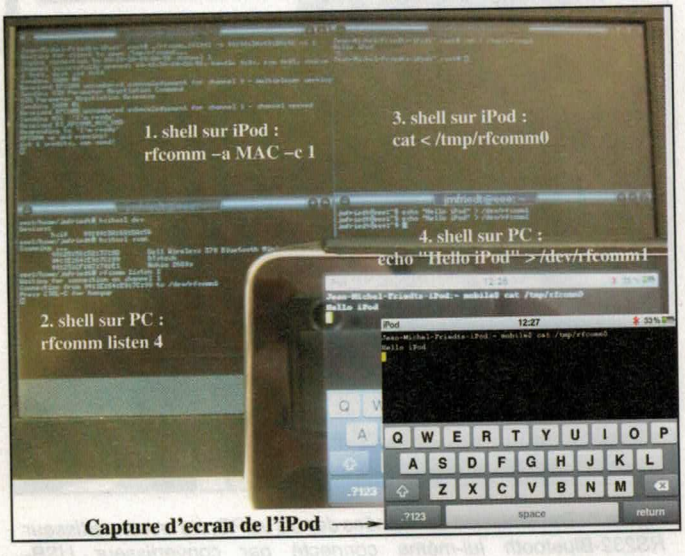


Figure 6 : Transmission de données depuis un shell sur PC, via un convertisseur USB-Bluetooth, vers iPod, au moyen des outils rfcomm exécutés sur les deux interlocuteurs.

Ce mode de communication, depuis le shell, permet d'exploiter l'iPod comme enregistreur et afficheur de trames, mais ne répond pas nécessairement à la flexibilité d'utilisation attendue sur iPod. Nous nous proposons donc de développer une application dédiée, en mode graphique, capable de rechercher les périphériques Bluetooth et d'afficher les informations qui en sont reçues.

6.2 Gestion des trames Bluetooth dans une application POSIX

Une première prise en main de la pile Bluetooth consiste en la mise en œuvre d'une application en console similaire à rfcomm, permettant d'afficher les messages transmis par un port série virtuel accessible au travers de `/tmp/rfcomm0`. Le principal inconvénient de cette approche tient en la gestion explicite des trames reçues :

1. Dans un premier temps, nous déclarons la compatibilité de cette application avec la norme POSIX par `run_loop_init(RUN_LOOP_POSIX);`.
2. La bibliothèque initialisant l'interface Bluetooth est sollicitée par `bt_open();`.
3. Nous enregistrons un gestionnaire des trames par `bt_register_packet_handler(packet_handler);` : l'argument est une fonction `void packet_handler(uint8_t , uint16_t , uint8_t*, uint16_t)` qui va traiter chaque paquet reçu par l'interface Bluetooth pour en interpréter le contenu (transactions associées aux différentes couches du protocole, L2CAP, HCI ou RFCOMM). Cette fonction se comportera comme un gestionnaire d'interruption (ISR, *Interrupt Service Routine*), appelée pour chaque message.
4. La liaison est initialisée par `bt_send_cmd(&btstack_set_power_mode,HCI_POWER_ON );`.
5. Finalement, la boucle de gestion des applications POSIX, l'équivalent de la boucle infinie dans la fonction `main()` pour un microcontrôleur, est exécutée par `run_loop_execute();`.

Nous retrouvons donc une structure classique de développement sur système embarqué, avec des événements gérés par une ISR et une boucle principale en attente de ces messages, ici gérée par le système d'exploitation de l'iPod. Le cœur de la gestion des trames, la fonction `packet_handler`, n'est pas explicitée ici : il s'agit d'une machine à états qui, en fonction de la nature du paquet reçu (`switch (packet_type) { ...`), gère la communication ou interprète le contenu en vue de son exploitation par l'application utilisateur qui doit donc s'insérer dans ce flux de gestion.

Cette stratégie est significativement plus complexe à mettre en œuvre - de par la gestion explicite des messages par la fonction `packet_handler` - que la stratégie exploitant Objective-C et l'environnement COCOA : dans ce cas, l'ISR est convertie en bibliothèque (`BTstackManager`) qui peut rester cachée à l'utilisateur. Dans la version POSIX d'un exemple d'application mis en œuvre pour simplement afficher les messages qui transitent, la fonction `packet_handler` nécessite environ 200 lignes de code, la majorité pour gérer le protocole, tandis que la seule fonction qui nous concerne en tant qu'utilisateur de la liaison Bluetooth se résume à :

```
if (packet[1] == BT_RFCOMM_UIH && packet[0] == ((RFCOMM_CHANNEL_ID<<3)|1)){
   packet_processed++;
   credits_used++;
   int written = 0,length = size-4,start_of_data=3;
   while (length) {                          // write data to FIFO (/tmp/rfcomm0)
      if ((written = write(fifo_fd, &packet[start_of_data], length)) == -1)
         {printf("Error writing to FIFO\n");}
      else {length -= written;}
   }
}
```

6.3 Débogage Bluetooth

Les premières tentatives d'utilisation de **BTstack** avec un convertisseur série bluetooth ont présenté des problèmes : à une vitesse de 115200 bauds, l'application sur iPod était en mesure de recevoir correctement les trames, mais si le Free2Move était configuré en 9600 bauds, l'application ne recevait plus la moindre trame.

Afin de déverminer une application Bluetooth, il est nécessaire d'utiliser **hcidump** (du package **bluez-hcidump** sur Debian). Cet outil permet d'afficher en direct les trames reçues ou envoyées depuis l'ordinateur sur lequel il est lancé (droits **root** obligatoires), de les enregistrer, ou de faire une analyse « post-mortem » du flux.

Sur iPod, **BTDaemon** enregistre automatiquement dans un fichier (**/var/tmp/hci_dump.pklg**) l'ensemble du trafic. Ainsi, après transfert de ce fichier de log sur le PC, il est possible grâce à la ligne :

```
hcidump -rVa hci_dump.pklg
```

de voir tout ce qui s'est produit et donc de connaître la cause de l'échec dans la communication, tel que présenté ci-après :

```
> ACL data: handle 12 flags 0x02 dlen 17
> ACL data: handle 12 flags 0x01 dlen 1
      L2CAP(d): cid 0x0040 len 14 [psm 3]
            RFCOMM(s): PN RSP: cr 0 dlci 0 pf 0 ilen 10 fcs 0xaa mcc_len 8
            dlci 2 frame_type 0 credit_flow 14 pri 0 ack_timer 0
            frame_size 100 max_retrans 0 credits 0
```

Listing 7 : Exemple de dump d'une trame L2CAP fragmentée

Le champ **flag** (*Packet Boundary Flag* dans les spécifications) avec une valeur de **0x02**, à la ligne 1, correspond au début d'un message. Deux cas sont possibles :

- La taille du contenu moins la taille d'un en-tête (4 octets), contenu dans la première trame, correspond à la taille annoncée pour le message (paramètre **len** l.3), alors le paquet n'est pas fragmenté.
- La taille est inférieure, ou un morceau du message est déjà stocké, alors cette trame est un fragment d'un message plus grand.

Si le flag est à **0x01**, alors la trame est la fin d'un message fragmenté. Dans cet exemple, un paquet de taille totale de 14 octets a été découpé en deux paquets, un de 17 octets et un de 1 octet, le premier paquet contenant l'en-tête de 4 octets.

Après analyse, il apparaît que les trames sont fortement fragmentées, ce qui est une caractéristique classique pour le Bluetooth (comme pour d'autres protocoles). En se référant à la documentation du convertisseur, nous avons appris que celui-ci augmentait la fragmentation lors de l'utilisation d'un débit faible afin de réduire les *timeout*. Après échanges de mails avec M. Ringwald, il est apparu que la fragmentation et la reconstruction de paquets n'étaient pas encore implémentées dans **BTstack**. Nous avons donc proposé un correctif qui a été exploité pour le support en question.

6.4 Affichage de données transmises par Bluetooth

Une seconde application consiste en la réception de trames Bluetooth contenant une valeur à afficher à l'écran en mode graphique. Dans ce cas, nous retirons toute authentification pour obtenir le résultat exposé sur les figures 11 et 12. Deux points méritent notre attention pour atteindre ce résultat :

- La liaison Bluetooth avec un balayage des périphériques accessibles pour éviter de devoir explicitement taper l'adresse MAC (équivalent de **hcitool scan** sous GNU/Linux).
- La création d'une fenêtre graphique et la capacité à y afficher un point à une position représentative de la valeur reçue.

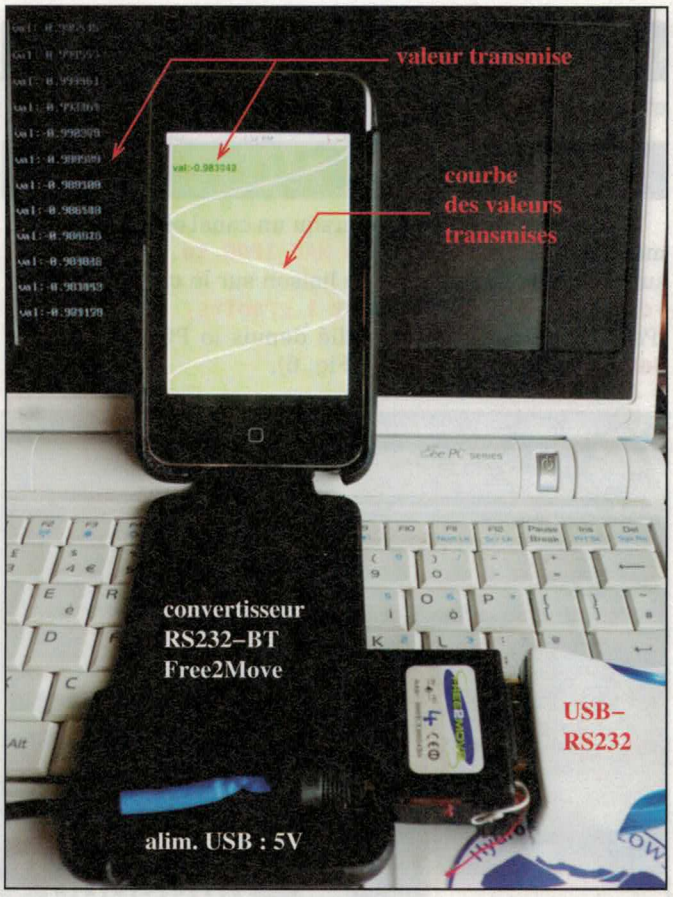


Figure 7 : Transmission de données depuis un PC, via un convertisseur RS232-Bluetooth lui-même connecté par convertisseur USB-RS232, pour affichage sur iPod. Cette application est représentative d'une application dans laquelle les données sont acquises par un microcontrôleur et transmises par liaison Bluetooth grâce au convertisseur Free2Move, les données étant émises selon un format val:valeur au travers du port série.

Afin de pouvoir exploiter la communication Bluetooth au travers de **BTstack**, il existe deux possibilités :

- Créer une classe « maison » qui aurait pour but d'encapsuler la machine d'états présentée plus tôt. Cette classe exploitant la boucle événementielle POSIX.

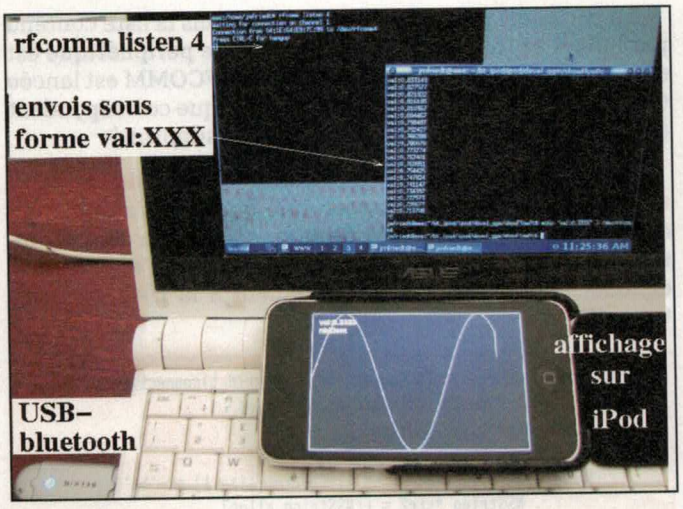


Figure 8 : Transmission de données depuis un PC, via un convertisseur USB-Bluetooth, au moyen d'un programme dédié à l'affichage graphique sur iPod avec émission de valeurs fournies depuis le shell sur PC. Comme nous l'avons vu auparavant, la liaison est établie sur le PC par rfcomm listen et les données transmises selon un format val:valeur au travers de /dev/rfcomm1.

Cette solution, bien que tentante, présente une difficulté, car la boucle POSIX lancée par `run_loop_execute` est bloquante et concurrente de la boucle événementielle de l'interface graphique : il n'est donc plus possible au niveau de l'application de lancer la fonction `[window makeKeyAndVisible];`, rendant donc impossibles l'affichage et l'utilisation de l'interface graphique. La solution possible est d'exécuter la boucle POSIX dans un autre *thread*, mais cette solution entraîne des contraintes pour les remises à jour des éléments de l'interface et ajoute une seconde boucle à l'application.

- Exploiter les bibliothèques disponibles dans le SVN de `Btstack`, exploitant la boucle COCOA pour la gestion des événements Bluetooth. Cette solution est sans doute la mieux adaptée vis-à-vis de nos objectifs.

Nous allons présenter en premier lieu les deux bibliothèques qui vont être utilisées par la suite.

6.4.1 BTstackManager

Cette première classe est la plus importante. Son rôle consiste à encapsuler tous les aspects liés à la communication et aux actions du Bluetooth (recherche de périphériques, demande d'informations, authentification, etc.).

Elle va permettre aux classes développées par l'utilisateur d'exploiter le Bluetooth sans avoir les mêmes contraintes que dans l'exemple précédent. Les nouvelles classes devront s'inscrire auprès de celle-ci afin de pouvoir envoyer des commandes et d'être averties de certains événements tels que l'arrivée d'un paquet RFCOMM, la découverte d'un nouveau périphérique, etc.

Son utilisation est particulièrement simple :

```
bt = [BTstackManager sharedInstance];
 [bt setDelegate:self];
 [bt addListener:self];
```

- l.1, une instance de `BTstackManager` est créée. Pour être plus explicite, l'appel à cette méthode de classe va créer un objet du type `BTstackManager` s'il n'existe pas, et dans le cas contraire, va se contenter de renvoyer le pointeur sur l'objet préexistant. Il ne peut donc y avoir qu'une seule instance de la classe `BTstackManager` pour toute l'application.

- l.2-3, l'instance de la classe qui contient le code du listing se déclare à la fois comme un *delegate* et comme un *listener*. Un pointeur sur un objet (dans le cas présent, l'objet courant au travers de self) est inscrit. Le delegate aura pour rôle d'avertir d'une requête de code PIN reçue d'un périphérique distant et de la réception d'un paquet. Le listener fournira l'annonce de divers événements liés à la tentative d'activation du Bluetooth, de ceux liés à la découverte d'un périphérique et de la création d'un canal de communication (L2CAP, RFCOMM).

Pour qu'une classe puisse être en mesure de s'inscrire en tant que delegate et/ou listener, il faut qu'au niveau de la déclaration de l'interface, elle s'annonce comme implémentant le ou les protocoles :

```
@interface ma_class: UIApplication <BtstackManagerDelegate,BTstackManagerListener>
{
[...]
```

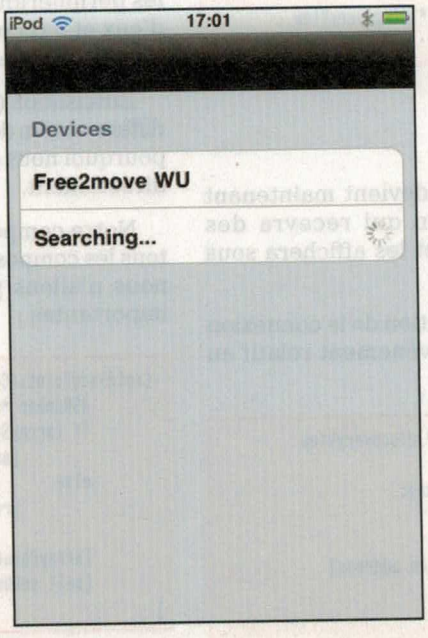


Figure 9 : Fenêtre de choix d'un périphérique

Ensuite, dans l'implémentation de la classe, il sera nécessaire d'ajouter les méthodes listées dans `BTCocoaTouch/include/BTstack/BTstackManager.h` du SVN.

À l'heure actuelle, la classe `BTstackManager` fournie dans le dépôt du projet ne dispose pas de l'implémentation complète du protocole RFCOMM, il est donc nécessaire d'appliquer le patch **(9)**.

6.4.2 Découverte des périphériques

La découverte et l'affichage des périphériques Bluetooth disponibles se font grâce à `BTDiscoveryViewController`. Cette classe a pour fonction d'afficher une table contenant les périphériques distants et de générer un événement lorsque l'utilisateur en a sélectionné un.

Du point de vue du développeur, cette classe présente la même simplicité d'utilisation que la précédente. La relation entre les événements Bluetooth et **BTDiscoveryViewController** se fait au travers du protocole **BTStackManagerListener**, qui va permettre de recevoir diverses informations, dont la découverte d'un nouveau périphérique et ses informations (nom, type, ...). La classe faite par le développeur qui va recevoir les événements de la classe **BTDiscoveryViewController** devra implémenter le protocole **BTDiscoveryDelegate**.

Son utilisation au niveau de l'application se limite à ces quelques lignes pour l'initialisation :

```
inqView =[[BTDiscoveryViewController alloc] init];
[inqView setDelegate:self];
nav =[[UINavigationController alloc] initWithRootViewController
:inqView];

[window addSubview:nav.view];
[bt addListener:inqView];
```

- la première ligne crée une nouvelle instance de cette classe ;
- dans le cas présent, la classe courante va recevoir les informations telles que l'annonce du choix de l'utilisateur (ligne 2) ;
- ligne 5, l'élément graphique est inséré dans la fenêtre ;
- ligne 6, l'instance reçoit un pointeur sur bt qui est une instance de **BtStackManager**.

À l'ajout du protocole ad-hoc dans la définition de l'interface

```
@interface ma_class: UIApplication <[...],BTDiscoveryDelegate>{
```

et à l'ajout de (au moins) cette méthode pour l'annonce de la sélection :

```
-(BOOL) discoveryView:(BTDiscoveryViewController*)discoveryView
willSelectDeviceAtIndex:(int)deviceIndex
```

6.4.3 Exemple

Avec tous les éléments en main, il devient maintenant possible de réaliser une application qui recevra des informations par la liaison Bluetooth et les affichera sous la forme d'une courbe.

La première étape consiste en la création de la connexion avec le périphérique, à l'arrivée de l'événement relatif au choix de l'utilisateur :

```
-(BOOL) discoveryView:(BTDiscoveryViewController*)discoveryView
willSelectDeviceAtIndex:(int)deviceIndex {
    selectedDevice = [bt deviceAtIndex:deviceIndex];
    BTDevice *device = selectedDevice;
    [bt stopDiscovery];
        [bt createRFCOMMConnectionAtAddress:[device address]
        withChannel:11 authenticated:NO];
        [nav.view removeFromSuperview];
    return NO;
}
```

Le périphérique choisi est récupéré dans la liste contenu par l'objet **bt** (ligne.2-3). La recherche de périphérique est stoppée (ligne.4). Puis la connexion en RFCOMM est lancée (ligne.5). Et finalement, l'élément graphique correspondant à la liste des périphériques trouvés est supprimé.

Il existe une fonction nommée **rfcommConnection CreatedAtAddress**, qui a pour rôle d'avertir l'application de la réussite ou de l'échec de la création de la connexion, mais dans le cas présent, nous considérons qu'elle ne peut échouer et nous passons directement à la réception des trames :

```
-(void) rfcommDataReceivedForConnectionID:(uint16_t)connectionID
withData:(uint8_t *)packet ofLen:(uint16_t)size{
        memcpy(tmp+nb,packet,size);
        nb+=size;
        if ((packet[size-1] == '\0')||(packet[size-1] == '\n')){
                NSString *str2 = [[NSString alloc]
                initWithCString:(char*)tmp+4 encoding : 1];
                CGFloat val = [str2 doubleValue];
                [affSin addPoint: val];
                nb = 0;
        }
}
```

Au même titre qu'un paquet L2CAP peut être fragmenté, il est possible qu'une trame RFCOMM puisse parvenir à l'application en plusieurs morceaux. C'est pourquoi les caractères reçus sont concaténés dans un *buffer* (l.2-3) tant que le caractère fin de chaîne ou retour à la ligne n'est pas détecté (l.4). Lorsque la trame est complète, elle est convertie en un **NSString**, qui est un objet Objective-C permettant le stockage et la manipulation de chaîne de caractères (l.5), le contenu est ensuite converti en un flottant (l.6) pour être finalement passé à l'instance de notre composant graphique (l.7) en charge de l'affichage de la courbe.

À ce stade, l'application est parfaitement apte à détecter les périphériques Bluetooth présents, à se connecter à l'un d'eux et à recevoir les données de celui-ci. Il ne reste donc plus qu'à coder notre composant graphique.

L'utilisation (instanciation et insertion) de ce composant ne diffère en rien de ce qui a pu être présenté précédemment, c'est pourquoi nous allons nous focaliser sur son implémentation directement.

Notre composant va hériter de la classe **UIView**, comme tous les composants disponibles en Cocoa. De ce composant, nous n'allons présenter que les deux méthodes les plus importantes :

```
-(void)addPoint:(CGFloat) point {
        NSNumber *nb = [NSNumber numberWithFloat: point];
        if (arraySize == nbPoints)
                [arrayPoints removeObjectAtIndex:0];
        else
                arraySize ++;

        [arrayPoints addObject:nb];
        [self setNeedsDisplay];
}
```

Listing 8 : Fonction en charge de la réception et du stockage de nouvelles données

Cette fonction (listing 8) appelée lors de la réception d'une trame complète se charge de stocker la nouvelle information dans une liste. Comme nous souhaitons faire défiler la courbe au fur et à mesure de l'arrivée de points, si le nombre de points déjà reçus dépasse un seuil (l.3), le premier point est supprimé (l.4) ; dans le cas contraire, le nombre de points reçus est juste incrémenté (l.6), puis le point est inséré à la fin de la liste (l.8) et un événement annonçant que le contenu de la fenêtre est obsolète est envoyé (l.9).

Cette dernière commande va avoir pour effet de forcer le rafraîchissement du contenu (listing 9) :

```
-(void)drawRect:(CGRect)rect
{
        CGRect imgRect = [self bounds];
        CGContextRef context = UIGraphicsGetCurrentContext();
        [[UIColor whiteColor] set];

        CGFloat orig=imgRect.size.width/2;
        CGFloat x=imgRect.size.width/2;
        CGFloat decalY = imgRect.size.height/(nbPoints-1.0);

        CGContextSetRGBFillColor (context, 0, 0, 0, 1);
        CGContextSetRGBStrokeColor(context,1,1,1,1);

        CGContextFillRect (context, imgRect);
        UIRectFrame(imgRect);
        int i;
        CGContextBeginPath (context);
        NSNumber *nb, *nb1;

        for (i=1;i<arraySize;i++) {
                nb = [arrayPoints objectAtIndex:i];
                nb1 = [arrayPoints objectAtIndex:i-1];
            CGContextMoveToPoint(context, x+[nb1 floatValue]*orig,
(i-1)*decalY);
            CGContextAddLineToPoint(context, x+[nb
floatValue]*orig,i*decalY);
        }
        CGContextStrokePath(context);
}
```

Listing 9 : Fonction d'affichage de la courbe à l'écran

La fonction **drawRect**, héritée de la classe **UIView**, est exécutée lors de l'affichage de l'interface graphique (au chargement de l'application) ou lorsqu'il est nécessaire de rafraîchir l'affichage de l'interface.

- Cette fonction ne va redessiner que la zone qu'elle couvre (à l'instar du rectangle fourni au bouton dans le premier exemple), la première opération nécessaire est d'obtenir ce rectangle (l.3).
- Comme nous allons dessiner dans cette zone, il est nécessaire ensuite d'obtenir un **CGContextRef** (l.4), c'est à travers cet objet qu'il sera possible d'afficher une couleur de fond et la courbe résultante de la réception de données.
- Les valeurs issues de la communication peuvent être négatives ou positives, ainsi il sera nécessaire de positionner l'origine de notre ordonnée au milieu de notre rectangle (l.7-8) et de définir le pas en abscisse (l.9).
- L'étape suivante, avant de commencer à dessiner, consiste à configurer la couleur du fond (l.11) et du stylo (l.12), ainsi qu'à lui donner la zone de dessin (l.14).
- Puis de s'assurer que le contexte est vide (l.17).
- Et nous pouvons enfin commencer à dessiner (l.20-25). Il est nécessaire d'avoir deux points pour dessiner une ligne (l.21-22), de se positionner aux coordonnées du premier (l.23) et de tracer la ligne vers le suivant (l.24).
- Il ne reste plus qu'à forcer l'affichage de l'ensemble de la courbe (l.26).

Le résultat est tel que présenté figure 8.

6.5 Contrôle d'une brique LEGO NXT

L'exemple précédent se contentait de recevoir des informations sur l'iPod et d'afficher le contenu, sous forme textuelle ou graphique. Nous allons maintenant proposer d'établir une liaison bidirectionnelle, dans laquelle l'iPod complète l'acquisition d'informations par l'émission d'ordres. Nous prendrons pour prétexte le contrôle d'un robot formé autour d'une brique LEGO NXT. Cette brique contient un processeur ARM7 exécutant un interpréteur de commandes, une interface graphique et une pile de communication Bluetooth. Dans le cas qui va nous intéresser ici, nous nous contenterons de l'interpréteur de commande fourni par défaut par LEGO, sans nécessiter l'écriture de programmes dédiés au NXT, tâche qui sortirait du cadre de cette présentation.

Plusieurs points doivent être maîtrisés en vue d'accomplir cette réalisation :

- Afin de réaliser nos premiers tests sous GNU/Linux, nous devons être capables d'authentifier la connexion entre NXT et un PC muni d'un adaptateur USB-Bluetooth. Il faut pour cela exécuter **bluetooth-agent "1234"** (disponible depuis la version *testing* de Debian/GNU dans le paquet **bluez**).
- Ayant identifié que le protocole de communication est RFCOMM (port série virtuel sur liaison Bluetooth), il nous reste à connaître les trames permettant la communication avec la brique NXT. De nombreux projets visant en l'exploitation libre de cette brique LEGO existent : nous nous sommes en particulier inspirés de LEGO::NXT en Perl **(10)** et de la *toolbox* Matlab issue de RWTH à Aachen **(11)**. Nous allons uniquement décrire ici quelques trames qui répondent à nos besoins : émettre un son (pour valider la connexion), activer un moteur ou recevoir une mesure d'un capteur :

  - ouverture de la connexion Bluetooth :

```
char nxt_addr[18] = "00:16:53:0E:39:D3";
  statusf = write(bt_socket,sendf, sizef);
  statusf = read(bt_socket,receber,sizeof(receber));

  bt_socket = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
  addr.rc_family = AF_BLUETOOTH;
  addr.rc_channel = (uint8_t) 1;
  str2ba( nxt_addr, &addr.rc_bdaddr);

  status = connect(bt_socket, (struct sockaddr *)&addr, sizeof(addr));
```

- Toutes les trames transmises par Bluetooth suivent le même format incluant la longueur de la phrase, sa nature (index de commande) et ses arguments, décrits à **[8]**. Ainsi, pour émettre un son :

```
char beep[8]={0x06,0x00,0x80,0x03,0xf4,0x01,0xf4,0x01};
write(bt_socket,beep,sizeof(beep));
```

- Pour faire tourner un moteur, on enverra le message :

```
// SETOUTPUTSTATE PORT POWER ON+REGUL SPEED REG TURN_RATIO RUNNING
char msg_fwd[11]={0x09,0x00,0x00,0x04,0x00,0x20,1+4,1,0x00,0x20,0};
```

avec 0x20 la vitesse de rotation.

- Le cas du télémètre à ultrasons est un peu plus compliqué, car il s'agit d'un périphérique I2C qui nécessite une initialisation et un protocole de communication dédié **[9]**. Dans un premier temps, la brique NXT est informée de la nature du capteur connecté au premier port (tableau `msg_rcv1`) et requiert une conversion en continu (tableau `msg_rcv2`)

```
// SETINPUTMODE PORT I2C_powered_9V RAW
  char msg_rcv1[7]={0x05,0x00,0x00,0x05,0x00,0x0b,0x00};
// LSWRITE PORT TX_LEN RX_LEN SENTENCE (0x41 0x02=continuous mode)
  char msg_rcv2[10]={0x08,0x00,0x00,0x0f,0x00,0x03,0x00,0x02,0x41,0x02};
```

- Nous itérons ensuite des requêtes de résultats en émettant successivement les contenus des tableaux `msg_rcv0` et `msg_rcv` :

```
// LSWRITE PORT TX_LEN RX_LEN READ_REG_0
  char msg_rcv0[9]={0x07,0x00,0x00,0x0f,0x00,0x02,0x08,0x02,0x42};
  char msg_rcv[5]={0x03,0x00,0x00,0x10,0x00}; // LSREAD PORT
```

Ayant validé la liaison Bluetooth authentifiée (LEGO considère probablement utile d'inculquer la notion de sécurité informatique dès le plus jeune âge) depuis GNU/Linux, ces fonctions sont portées à l'iPod pour transformer ce dernier en interface de communication bidirectionnelle.

La démonstration finale que nous nous sommes proposés de développer est le contrôle du robot fourni comme exemple de base par LEGO au moyen des signaux issus des accéléromètres de l'iPod Touch.

Figure 10 : La brique NXT reçoit des commandes de l'iPod Touch générées à partir des signaux lus sur ses accéléromètres : ainsi, pencher l'iPod dans une direction incite le robot à se déplacer dans cette direction.

Afin d'exploiter au mieux les périphériques mis à disposition par l'iPod Touch, il nous reste à accéder aux composants probablement les plus ludiques de cette plate-forme : les accéléromètres. Dans la tradition Apple, nous ne nous soucierons pas de connaître les détails de l'accès au matériel : des méthodes sont mises à disposition par les bibliothèques propriétaires pour obtenir les informations (un nombre en virgule flottante représentant l'inclinaison de l'iPod Touch dans une des 3 directions, en radians) : après une phase d'initialisation

```
/* Initialisation */
UIAccelerometer *accel = [UIAccelerometer sharedAccelerometer];
/* for callback function */
accel.delegate = self;
/* timer for event */
accel.updateInterval = 0.03f;
```

dans laquelle nous obtenons un pointeur sur l'instance qui encapsule le pilotage des accéléromètres (l.2), nous inscrivons la classe courante auprès de cet objet (l.4) afin de recevoir les données qui seront fournies avec un intervalle configuré en l.6.

Ainsi, toutes les 0.3 secondes, les informations sont récupérées

```
-(void)accelerometer:(UIAccelerometer*)acel didAccelerate:(UIAcceleration
  *)aceler
{
    vx = aceler.x;
    vy = aceler.y;
    vz = aceler.z;
       [...]
}
```

sous forme d'une structure `acceler.{x,y,z}` dont les éléments sont les trois angles d'inclinaison qui nous intéressent.

Finalement, afin d'authentifier la liaison entre l'iPod Touch et la brique NXT (fonction fournie par `bluetooth-agent` sous GNU/Linux), nous ajoutons la séquence de commandes suivante à l'initialisation de la liaison Bluetooth :

1. création du canal L2CAP :

```
bt_send_cmd(&hci_write_authentication_enable, 1);
```

2. lors de la requête pour la clé PIN, nous répondons par :

```
bt_send_cmd(&hci_pin_code_request_reply, &event_addr, 4, "1234");
```

Tout comme sous GNU/Linux, les séquences de commandes restent celles présentées plus haut, mais cette fois dans le formalisme imposé par Objective-C :

```
char msg_fwd_v[30]={0x0d,0x00,0x80,0x04,0x00,0x00,1+4,1,0x00,0x20,0,0,0,0,0
                   ,0x0d,0x00,0x80,0x04,0x02,0x00,1+4,1,0x00,0x20,0,0,0,0,0};
```

qui fournit la trame de base pour contrôler les moteurs :

```
- (void)accelerometer:(UIAccelerometer *)acel didAccelerate:(UIAcceleration *)aceler
{
      char vx = (char)(aceler.x*100);
      char vy = (char)(aceler.y*100);
      msg_fwd_v[5] =  msg_fwd_v[20] = vx;
      if (vy <-0)      msg_fwd_v[5] = vx+vy;
      else if (vy > 0) msg_fwd_v[20] = vx-vy;
      [bt sendRFCOMMPacketForChannelID: 1
            packet: (uint8_t *) msg_fwd_v
            len:sizeof(msg_fwd_v)];
}
```

Il est à noter que dans le cas de la version iPod, les deux commandes pour contrôler les deux moteurs sont envoyées dans le même message : dans le cas contraire, les moteurs n'ont pas un comportement normal, donnant un déplacement en crabe. Il est donc possible de concaténer les commandes pour les expédier dans un unique message Bluetooth (l.8 à 10).

La conclusion de ce développement est un robot chenillé contrôlé au moyen de l'iPod qui obtient ses ordres depuis les accéléromètres et rend ainsi le pilotage plus « intuitif » qu'au moyen de boutons de commande (Fig. 10).

CONCLUSION

Nous avons démontré la capacité à développer sur iPod Touch en exploitant exclusivement des outils open source et ce, afin de créer des programmes en mode texte (exploitant l'interface POSIX) ou en mode graphique (exploitant l'interface COCOA). Une fois l'iPod libéré, l'accès à l'intégralité de ses fonctionnalités, et notamment le système de fichiers, s'obtient au moyen d'une liaison SSH qui permet notamment de transférer nos propres exécutables sur cette plate-forme. Au-delà de la liaison wifi, nous avons exploité l'interface sans fil Bluetooth au travers d'une pile libre - BTstack - rendue compilable au moyen des outils de cross-compilation exécutés sous GNU/Linux. Cette pile Bluetooth a été complétée d'un certain nombre de fonctionnalités pour la gestion du mode RFCOMM, pour utiliser l'iPod comme interface utilisateur pour un affichage graphique et stockage des informations acquises par un capteur capable de communiquer par cette interface, et ainsi étendre la gamme des périphériques accessibles depuis l'iPod.

Notons finalement que, bien que la méthode de libération par Spirit et la toolchain présentée ici permette de générer des binaires fonctionnels sur iPad, les applications graphiques n'exploitent que la résolution d'un iPod (320*480 pixels) au lieu de l'intégralité de l'écran de l'iPad. Par ailleurs, seule la version la plus récente (version 472) de MobileTerminal disponible à code.google.com/p/mobileterminal est fonctionnelle sur iPad, fournissant une interface confortable (clavier de taille raisonnable) en complément de l'interaction au travers d'un serveur SSH. ■

Auteur : Gwenhaël Goavec-Merou

Gwenhaël Goavec-Merou est en thèse sur le traitement d'images sur FPGA à la fois dans le département temps-fréquence de l'institut FEMTO-ST et au sein d'une entreprise.

Auteur : Jean-Michel Friedt

Jean-Michel Friedt est ingénieur dans une société privée, hébergée par le département temps-fréquence de l'institut FEMTO-ST à Besançon, et membre de l'association pour la diffusion de la culture scientifique et technique Projet Aurore.

Références

- **[1]** http://www.apple.com/ipodtouch/specs.html pour une version incomplète des spécifications techniques, Wikipédia et les références associées fournissant les valeurs numériques fournies dans le texte.
- **[2]** É. Vautherin, Développer pour l'iPhone et l'iPad - Le guide du SDK, Créez vos applications pour l'App Store, Dunod (2010), ne présente que peu d'intérêt pour le développeur sous GNU/Linux, mais illustre quelques concepts sur les objets disponibles pour la réalisation des interfaces graphiques.
- **[3]** Y. Le Guen, « Programmation Objet : Objective-C », *GNU/Linux Magazine France* n°9 (2000), disponible à http://chl.be/glmf/www.linuxmag-france.org/old/lm9/objectiveC.html
- **[4]** Y. Le Guen, « Programmation Objet : Objective-C, 2ème partie », *GNU/Linux Magazine France* n°10 (2000), disponible à http://chl.be/glmf/www.linuxmag-france.org/old/lm10/objectiveC.html
- **[5]** A.M. Duncan, *Objective-C pocket reference*, O'Reilly Media (2002)
- **[6]** Un point de départ pour appréhender le langage à http://www.siteduzero.com/tutoriel-3-5208-la-programmation-mac.html
- **[7]** X. Garreau, « Bluetooth, installation et utilisation », *GNU/Linux Magazine France* n°78 (2005), disponible à http://www.unixgarden.com/index.php/comprendre/bluetooth-installation-et-utilisation
- **[8]** *LEGO Mindstorms NXT Direct Commands v1.00* (2006), disponible à http://www.microframeworkprojects.com/images/d/df/LEGO_MINDSTORMS_NXT_Direct_commands.pdf
- **[9]** *LEGO Mindstorms NXT Ultrasonic Sensor I2C Communication Protocol v1.00* (2006), disponible à http://www3.wooster.edu/physics/jacobs/220/Appendix_7_Ultrasonic_Sensor_I2C_communication_protocol.pdf, ou pour une version plus lisible en C : http://stackoverflow.com/questions/1967978/lego-mindstorm-nxt-cocoa-and-hitechnic-sensors
- **[10]** S. Guinot, J.-M. Friedt, « La réception d'images météorologiques issues de satellites : utilisation d'un système embarqué », *GNU/Linux Magazine France Hors-Série* n°24 (2006)

Notes

- **(1)** http://llvm.info
- **(2)** http://www.saurik.com/id/4
- **(3)** developer.apple.com/technologies/xcode.html
- **(4)** http://developer.apple.com
- **(5)** http://en.wikipedia.org/wiki/Objective-C
- **(6)** http://code.google.com/p/btstack/wiki/RFCOMM
- **(7)** www.free2move.se/, produits F2M01C1i et Uncord pour une liaison Bluetooth-RS232
- **(8)** http://mindstorms.lego.com, technologie NXT
- **(9)** http://www.trabucayre.com/lm/btstackmanager_rfcomm.patch
- **(10)** http://nxt.ivorycity.com/
- **(11)** http://www.mindstorms.rwth-aachen.de/documents/downloads/doc/troubleshooting.html

C : Retour sur les qualificateurs

En C, const et volatile sont souvent considérés comme des sous-types de données, l'un étant l'opposé de l'autre. Ces deux affirmations sont parfaitement fausses. Il en découle en général une utilisation plus ou moins aléatoire aussi bien dans le développement userland, kernel ou pour des éléments comme des microcontrôleurs. Petite mise au point sur le rôle de ces gentilles bestioles.

const et **volatile** ne sont pas des types de données comme **char**, **int**, **long** ou **unsigned**. Ces derniers sont des spécificateurs de type. On notera au passage que **int** implique **signed** et inversement. Un **int** est par défaut signé et un **signed** par défaut un **int**. De ce fait, **signed int** n'est rien d'autre qu'un pléonasme.

const et **volatile** sont des qualificateurs. Ils permettent au développeur d'informer le compilateur sur la manière de traiter une variable. On remarquera avec amusement que le terme « qualificateur » est dans le dictionnaire de l'académie française, mais qu'il définit un membre d'un tribunal de l'inquisition chargé de qualifier un crime. Il est vrai qu'en C, on risque facilement le bûcher...

Concernant l'opposition de **const** et **volatile**, nous verrons dans la suite que ces deux qualificateurs ne portent absolument pas sur la même chose et que **const volatile int ma_variable** est parfaitement viable, autorisé et tantôt bien utile.

1 CONST

Pour bien ancrer le sens de **const** dans son esprit, il suffit de l'associer avec le mot « constant », synonyme d'invariabilité. Une variable (sic) déclarée avec **const** est donc quelque chose en lecture seule. Mais il ne s'agit pas nécessairement de son contenu. Le qualificateur **const** sert à préciser au compilateur une notion d'invariabilité. Reste encore à comprendre sur quoi porte cette dernière.

Le cas le plus courant d'utilisation de **const** est sans le moindre doute ceci :

```
const int toto = 42;
```

Si vous tentez de modifier **toto**, le compilateur ne manquera pas de vous signaler votre erreur :

```
% make
gcc -O3 -Wall -c -o base.o base.c
base.c: In function 'main':
base.c:12: error: assignment of read-only variable 'toto'
make: *** [base.o] Erreur 1
```

En revanche, bien que cela soit techniquement possible, il est « mal » de générer un pointeur qui ne pointe pas vers des données **const** à partir d'un **const**. Exemple d'horreur :

```
int i = 42;
const int *ptoto;

ptoto = (int *)&i;
printf("%d\n", *ptoto);

i=25;
printf("%d\n", *ptoto);
int *plop = (int *)ptoto;
*plop = 75;
printf("%d\n", *ptoto);
```

La variable **i** est un simple **int** classique, variable... normal. Nous déclarons un pointeur sur un **int** constant (en lecture seule donc). À la troisième ligne, nous commençons à tricher et utilisons le pointeur vers **i** pour donner une adresse à **ptoto**. Plus loin, si nous changeons **i**, le contenu de ***ptoto** change également. Nos données ne sont plus en lecture seule. Idem pour le comble de l'illisibilité avec la déclaration/initialisation de **plop**. Souvenez-vous, **const** est là pour placer un contenu en lecture seule. En jouant avec les pointeurs pour contourner le problème, vous vous mettez vous-même en danger. À noter, d'ailleurs, que l'absence du cast de **&i** ne provoquera pas même un *warning* avec GCC (version 4.4.3), même en présence de l'option **-Wall**. l'initialisation de ***plop** en revanche déclenchera :

```
base.c: In function 'main':
base.c:19: warning: initialization discards
  qualifiers from pointer target type
```

Mais ce n'est pas tout. Comme précisé en début de section, **const** indique que quelque chose est en lecture seule. Mais il peut s'agir du pointeur lui-même. Remplacez simplement :

```
const int *ptoto;
```

par :

```
int *const ptoto;
```

Immédiatement, le compilateur vous signale le problème : **error: assignment of read-only variable ptoto**. Inversement, avec un **int *const ptoto = &i**, un ***ptoto = 666;** ne posera aucun problème. Le pointeur est en lecture seule,

const et volatile

mais le contenu de la variable peut être changé à souhait. Remarquez que ceci n'est pas vrai sans initialisation de **ptoto**, ce qui démontre bien que jouer avec **const** et les pointeurs n'est généralement pas une bonne chose et que mieux vaut s'abstenir de contourner ou détourner l'objet même du qualificateur.

2 VOLATILE

volatile est sans doute plus simple dans son utilisation alors qu'il l'est moins dans son concept. Ce qualificateur ne sert pas à placer une variable en lecture/écriture, du moins, pas exactement. Le problème qui se pose ici touche à l'optimisation du code par le compilateur.

Soit le code suivant :

```
struct devregs{
  unsigned short contstat;
  unsigned short data;
};

#define BASEADDR ((struct devregs *)0xffff0010)

unsigned int readdata(unsigned devnum){
  struct devregs *device = BASEADDR + devum;

  /* boucle d'attente */
  while((device->csr & 1) == 0);

  /* erreur ? */
  if(device->csr & ERROR){
    return(0xffff);
  }

  return((device->data) & 0xff));
}
```

C'est un cas relativement classique lorsqu'on touche au matériel, par exemple, ou dans la programmation de microcontrôleurs. Lire une adresse mémoire dont les données sont modifiées par ailleurs est fortement lié à l'utilisation d'interruption. On retrouvera souvent ce type de code dans les pilotes de périphériques. Mais ce code a de très fortes chances de ne pas fonctionner et de provoquer une attente infinie au niveau du **while()**. En effet, les techniques d'optimisation utilisées par les compilateurs font que ce dernier aura aussitôt fait de se rendre compte que la boucle lit en permanence la même adresse mémoire. Il va donc prendre la valeur s'y trouvant et la placer dans un registre afin d'accélérer le traitement de la boucle.

Malheureusement, ceci rend notre code défectueux, car nous savons que la valeur à cette adresse peut changer et nous attendons même qu'elle le fasse. Le compilateur non. **volatile** nous permet de préciser ce fait au compilateur qui ne fera, alors, pas d'optimisation. Ici, il y a deux manières de voir les choses, soit on précise le qualificateur pour les variables dans la structure :

```
struct devregs{
  unsigned short volatile contstat;
  unsigned short volatile data;
};
```

soit on le fait dans la déclaration de la variable contenant la structure en question :

```
volatile struct devregs *device = BASEADDR+devnum;
```

Si une structure est qualifiée avec **const** ou **volatile**, tous les membres de cette structure seront également qualifiés lors de la déclaration. Il est même possible de mixer avec **const** en rendant ce pointeur en lecture seule :

```
volatile struct devregs *const device = BASEADDR+devnum;
```

Vous ne pouvez pas, cependant, utiliser le qualificateur **volatile** dans la définition de la structure car, là, c'est la structure déclarée qui sera qualifiée et non le **struct** lui-même. Ceci est donc inopérant :

```
volatile struct devregs{
  unsigned short contstat;
  unsigned short data;
} *device;

struct devregs *newdevice;
```

newdevice ne sera pas qualifié de **volatile** car, dans la déclaration de **device**, c'est bel et bien **device** qui est qualifié et non **devregs**. En revanche, vous pouvez utiliser un **typedef** pour cela :

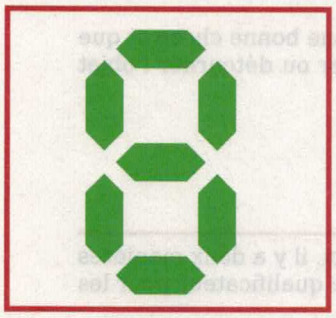
```
struct devregs{
  unsigned short contstat;
  unsigned short data;
};
typedef volatile struct devregs vdev;

vdev *device;
struct struct devregs *newdevice;
```

device est **volatile** mais **newdevice** ne l'est pas.

POUR CONCLURE

Après ce bref rappel des deux qualificateurs sans doute les plus importants du C, et les plus trompeurs, la conclusion qui s'impose naturellement à nous est très simple. Les mécanismes mis en place par un langage pour faciliter la vie du programmeur sont parfois aussi ceux qui peuvent la compliquer. Cependant, en dehors du cas particulier de la modification d'une valeur à une adresse mémoire spécifique, les problèmes n'apparaissent que lorsqu'on tente le diable en jouant littéralement avec ces mécanismes. ■

Création d'un afficheur 7 segments



Auteur

■ Yann Guidon

Après avoir disséqué le langage VHDL, son implémentation par GHDL et l'extension graphique du numéro de septembre [1], passons à la pratique ! On en avait presque oublié que tous ces efforts pour accéder au framebuffer sont motivés par le besoin de Laura d'afficher confortablement les chiffres de sa montre. Maintenant que nous savons contrôler les pixels de l'écran lors d'une simulation, l'affichage est (quasiment) un jeu d'enfant ! Profitons-en pour découvrir de nouveaux aspects bien pratiques du langage VHDL, tout en faisant un peu de programmation graphique.

1 RAPPELS

Pour utiliser le code de cet article, il est nécessaire de disposer du code présenté en septembre. Il est disponible librement à http://ygdes.com/GHDL/fb/ et ne fonctionne qu'avec GHDL sous GNU/Linux. Il pourrait aussi fonctionner sous des BSD et dérivés, mais pas sous Windows.

Pour pouvoir accéder à l'écran (et plus précisément, au *framebuffer*), il ne faut pas oublier au début de notre code source les incantations magiques qui rendent le paquetage accessible :

```
library work;
use work.fb_ghdl.all;
```

Cela initialise automatiquement le framebuffer. Nous disposons alors d'un tableau appelé **pixel**, accessible en lecture et écriture. Chaque élément du tableau est un pixel codé au moyen d'un nombre entier de 32 bits (de type **integer**), ce qui semble être la configuration par défaut des cartes graphiques sous Linux de nos jours. Pour écrire un pixel de valeur 0x123456 aux coordonnées x=24, y=42, il suffit d'écrire le code suivant :

```
pixel(42, 24) := 16#123456#;
```

Le VHDL n'utilise pas la notation du C pour les valeurs hexadécimales, mais au moins, il y en a une. L'inversion des indices (**y** en premier et **x** en second) est le seul détail important dont il faut se souvenir en permanence.

Les dimensions de l'écran sont fournies dans les constantes **fbx** et **fby**. Tout accès à des coordonnées en dehors de ces limites provoquera un arrêt de la simulation.

Pour tous les détails concernant le code lui-même ou le framebuffer (sa configuration et son utilisation en C), tout a déjà été expliqué dans ce magazine.

2 SÉLECTION DE LA COULEUR

Notre afficheur virtuel va être connecté à des fils virtuels en VHDL, du type **std_logic** et dont la définition se trouve dans le paquetage **std_logic_1164**. Ce type peut prendre une valeur parmi neuf, représentant différents états :

```
-- extrait de std_logic_1164.v93
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

Syntaxiquement, il s'agit d'une énumération : **'U'** va prendre la valeur entière 0, **'X'** prendra 1, etc. Par exemple, chaque fois que GHDL verra le caractère **'W'** dans un contexte où il s'attend à une valeur du type **std_logic**, il va la remplacer en interne par la valeur 5.

avec GHDL

Notre afficheur virtuel doit donc tenir compte des neuf états ci-dessus et changer la couleur du segment. Dans la pratique, nous rencontrons surtout trois cas : `'0'`, `'1'` et `'U'`, les autres ne se produisant que lorsque le code ne fonctionne pas bien (ce qui leur vaut leur couleur gris foncé). `'U'` se produit normalement au début de la simulation, lorsque le système est en cours d'initialisation.

La conversion d'un état `std_logic` vers une couleur (représentée par un entier) étant réalisée par divers morceaux de code, il est logique d'en faire une fonction. Elle est très simple et contient seulement un branchement à choix multiples, ce qui illustre l'utilisation de la syntaxe `case ... when ...` (l'équivalent du `switch ... case ...` en C) :

```
function sul2rgb(v: std_ulogic) return integer is
begin
  case v is
    when '1'    => return 16#4fff4f#; -- vert clair
    when '0'    => return 16#003000#; -- vert foncé
    when 'U'    => return 16#ff9400#; -- orange
    when others => return 16#8b8b8b#; -- gris foncé
  end case;
end sul2rgb;
```

Il y a une autre manière de réaliser cette conversion simple, puisque le nombre d'états possibles en entrée est très limité : on peut utiliser une table au lieu d'une fonction. L'utilisation est inchangée et la vitesse d'exécution est meilleure :

```
type sul2int_array is array (std_ulogic) of integer;
constant sulv2int : sulv2int_array := (
  16#ff9400#, -- 'U'
  16#8b8b8b#, -- 'X'
  16#003000#, -- '0'
  16#4fff4f#, -- '1'
  16#8b8b8b#, -- 'Z'
  16#8b8b8b#, -- 'W'
  16#8b8b8b#, -- 'L'
  16#8b8b8b#, -- 'H'
  16#8b8b8b#  -- '-'
);
```

La syntaxe du VHDL est assez sophistiquée, mais quand on la connaît, on s'aperçoit qu'elle a été conçue pour faire des raccourcis très pratiques. D'abord, on voit que l'indice dans le tableau peut être autre chose qu'un nombre : VHDL accepte les énumérations telles `std_logic`. Ainsi, il n'est pas nécessaire de transformer explicitement un fil en sa représentation interne, ce qui éclaircit le code.

Ensuite, le code précédent utilisait une notation par position, où la valeur d'une constante est assignée directement à l'élément du même indice dans la liste. On peut aussi utiliser une autre notation où les indices sont explicites (la position dans la liste n'est plus utilisée) et on peut même désigner le reste des éléments avec le mot-clé `others` :

```
type sul2int_array is array (std_ulogic) of integer;
constant sulv2int : sulv2int_array := (
  'U'    => 16#ff9400#,
  '0'    => 16#003000#,
  '1'    => 16#4fff4f#,
  others => 16#8b8b8b# -- économise 5 lignes
);
```

Pour obtenir la couleur en fonction de la valeur du fil, on n'a plus qu'à appeler la fonction ou consulter la table, ce qui se code exactement de la même manière :

```
variable c: integer;
c := sul2rgb(v);
-- ou
c := sul2int(v);
```

3 AFFICHAGE D'UN... POINT

Notre premier élément graphique est le plus simple possible. C'est le point décimal, ou les deux points utilisés pour séparer les minutes des secondes sur une horloge digitale. Et pour se simplifier encore la vie, nous n'allons pas afficher de rond, mais faire un point carré (attention, un mathématicien français se cache dans ce paragraphe).

```
procedure segment_point(x,y,h: integer; v: std_ulogic) is
  variable i,j, c: integer;
begin
  -- convertit la couleur :
  c := sul2int(v);
  -- carré :
  for j in y to y+h loop
    for i in x to x+h loop
      pixel(j,i) := c;
    end loop;
  end loop;
end segment_point;
```

Les coordonnées du point supérieur gauche sont données par `x` et `y`, `h` est la largeur et la hauteur du carré. `i` et `j` sont les itérateurs pour `x` et `y`. C'est le minimum à comprendre pour faire de la programmation graphique.

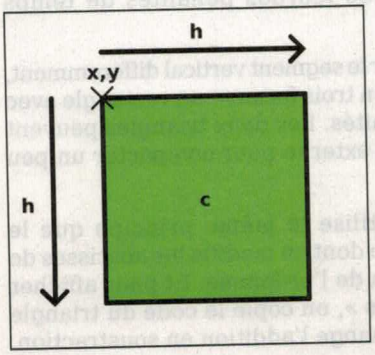


Figure 1 : Coordonnées et paramètres pour afficher le carré

4 AFFICHAGE D'UN SEGMENT HORIZONTAL

Après ce petit échauffement des neurones, passons à un élément plus élaboré : un segment horizontal. Il serait très simple de le représenter comme un rectangle, mais ça manque de style car la plupart des afficheurs utilisent une forme allongée, avec les bouts biseautés à 45 degrés.

La contrainte principale dont il faut tenir compte pour que le code soit efficace est que les accès à la mémoire doivent être (autant que possible) à des adresses consécutives et croissantes. Donc il faut balayer la forme à remplir de gauche à droite en priorité.

On peut considérer l'hexagone allongé représentant notre segment, comme deux trapèzes accolés contre leur bord le plus long. Donc pour afficher le segment, il faut savoir afficher un trapèze... Pas de souci : un trapèze à 45 degrés est un rectangle un peu modifié. En ajustant au fur et à mesure les coordonnées de début et de fin de la ligne horizontale, on déforme facilement le rectangle pour le transformer en trapèze.

Pour profiter de la symétrie, on exécute deux fois la boucle de la ligne horizontale, une fois pour chaque trapèze, en additionnant et en soustrayant l'itérateur principal (**j**). Cette technique a l'avantage de créer un code source compact, mais la ligne du centre est écrite deux fois. Pour contourner cela, on commence l'itération de **j** à 1, pour sauter la ligne double, cette dernière étant affichée une seule fois par un autre balayage horizontal supplémentaire.

```
procedure segment_horizontal(x,y,w,h: integer; v: std_ulogic) is
  variable i,j,t, c: integer;
begin

 c := sul2int(v);
   -- la ligne centrale, affichée une seule fois
 for i in x to x+h loop
   pixel(y,i) := c;
 end loop;

 for j in 1 to w loop
   -- boucle sur la ligne du dessus
   -- j modifie l'abscisse de début et de fin de la ligne
   -- ainsi que l'ordonnée, pour faire un angle à 45 degrés
   t := y-j;
   for i in (x+j) to (x+h-j) loop
     pixel(t,i) := c;
   end loop;
   -- boucle sur la ligne du dessous
   -- (identique mais avec une autre ordonnée)
   t := y+j;
   for i in (x+j) to (x+h-j) loop
     pixel(t,i) := c;
   end loop;
 end loop;
end segment_horizontal;
```

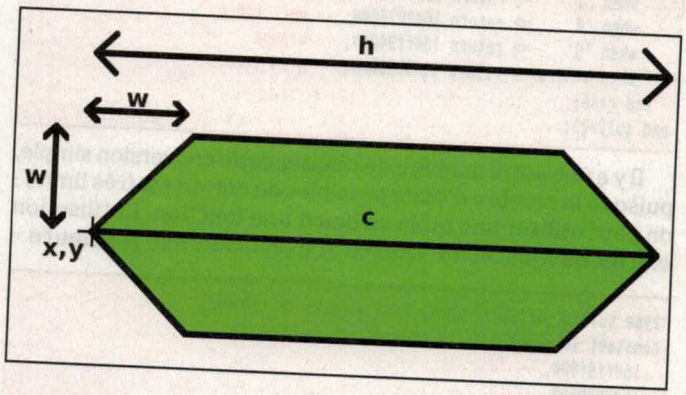


Figure 2 : Coordonnées et paramètres pour afficher les deux trapèzes constituant le segment horizontal.

5 AFFICHAGE D'UN SEGMENT VERTICAL

Pour le segment vertical, on pourrait très bien repartir du code horizontal et échanger les paramètres **x** et **y**. Or la règle est de balayer l'écran par lignes et non pas par colonnes, ce qui inflige de lourdes pénalités de temps d'accès à la mémoire.

Nous devons donc traiter le segment vertical différemment, en découpant l'hexagone en trois formes : un rectangle avec deux triangles aux extrémités. Les deux triangles peuvent réutiliser la même boucle externe pour compacter un peu le code.

Afficher un triangle utilise le même principe que le trapèze : c'est un rectangle dont on modifie les abscisses de début et de fin en fonction de l'ordonnée. Et pour afficher un triangle « la tête en bas », on copie le code du triangle « la tête en haut » et on change l'addition en soustraction.

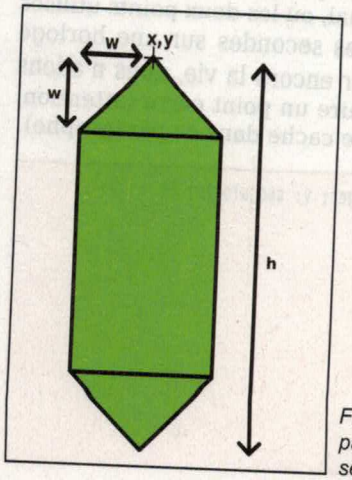


Figure 3 : Coordonnées et paramètres pour afficher le segment vertical

```
procedure segment_vertical(x,y,w,h: integer; v: std_ulogic) is
  variable i,j, c: integer;
begin
  c := sul2int(v);

  for j in 0 to w loop
    -- triangle supérieur
    for i in (x-j) to (x+j) loop
      pixel(y+j,i) := c;
    end loop;
    -- triangle inférieur
   for i in (x-j) to (x+j) loop
      pixel(y+h-j,i) := c;
    end loop;
  end loop;

  -- rectangle
  for j in y+w to y+h-w loop
    for i in x-w to x+w loop
      pixel(j,i) := c;
    end loop;
  end loop;

end segment_vertical;
```

6 ET UN SEGMENT OBLIQUE !

Celui-ci n'est pas nécessaire pour afficher les nombres, mais je tiens à le réaliser puisque ça permet de faire des lettres et surtout son codage utilise une technique un peu particulière et très intéressante. En effet, grâce à celle-ci, nous pouvons afficher un hexagone avec juste une double boucle imbriquée, sans recourir à des symétries ou aux astuces précédentes.

Lorsqu'on regarde la forme à générer, on peut la voir comme un petit carré qui se décale sur la diagonale d'un grand carré. Mais ce qui nous intéresse le plus, c'est les abscisses de début et de fin de chaque ligne horizontale (puisqu'il faut accéder au framebuffer ligne par ligne).

Appelons **a** l'abscisse de gauche (le début du balayage de la ligne) et **b** l'abscisse à droite (la fin). On voit que **b** démarre avec une valeur **w** et augmente progressivement jusqu'à atteindre l'abscisse maximale (égale à **h**). On code cela au moyen d'une instruction conditionnelle (**if**) : on incrémente **b** seulement si sa valeur est inférieure à **h**.

Le même principe est utilisé pour la variable **a** : elle commence à augmenter lorsque le nombre de lignes dépasse **w**. Un autre **if** va détecter cette condition et autoriser ou non l'incrémentation de **a** à chaque ligne. C'est ainsi que nous ne codons que deux boucles imbriquées, mais avec deux **if** entre les deux.

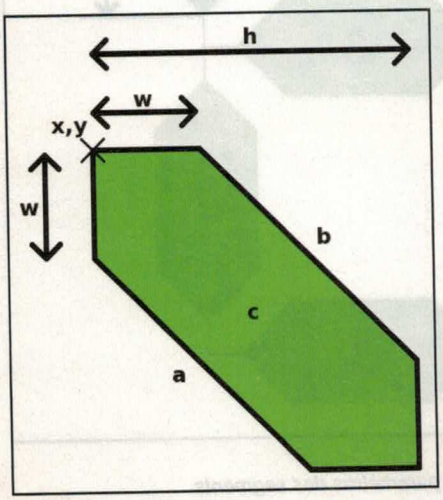


Figure 4 : Coordonnées et paramètres pour afficher le segment oblique

```
procedure segment_oblique(x,y,w,h: integer; v: std_ulogic) is
  variable i, j, a, b, c: integer;
begin
  a := 0;
  b := w;
  c := sul2int(v);

  for j in 0 to h loop
    -- affiche une ligne horizontale
    for i in a+x to b+x loop
      pixel(y+j,i) := c;
    end loop;

    -- avance le début :
    if j>w then a := a+1; end if;
    -- avance la fin :
    if b<h then b := b+1; end if;
  end loop;
end segment_oblique;
```

Ce code ne fonctionne que dans une direction (en bas à droite). Pour réaliser la même chose, mais vers le haut, on pourrait recopier la fonction et transformer les incrémentations de **y** en décrémentations. Mais dupliquer du code juste pour cette petite adaptation, ce n'est pas bien ! Il est préférable d'indiquer la direction comme un paramètre de la fonction et d'ajouter ce paramètre au lieu d'incrémenter ou décrémenter de manière figée.

Le paramètre de direction prendra la valeur 1 pour aller vers le bas (ce qui équivaut à l'incrémentation) ou -1 (décrémentation) pour aller vers le haut. Ces valeurs sont ensuite définies comme constantes pour éviter de se mélanger les pinceaux en codant. Les quelques petites modifications suivantes évitent donc de doubler la taille du code :

```
constant oblique_haut : integer := -1;
constant oblique_bas  : integer :=  1;

procedure segment_oblique(x,y,w,h,dir: integer; v: std_ulogic) is
  .....
    -- passe à la ligne du dessous
    y := y+dir;
  .....
```

Il ne faut pas non plus oublier de corriger l'épaisseur du trait car il est incliné à 45 degrés. Normalement, il devrait être plus épais d'un facteur 1,414 (racine carrée de 2), un rapport de 3/2 convient souvent.

La figure 5 montre le résultat que l'on peut obtenir en associant les différents segments que nous venons de concevoir. Les primitives et autres constantes sont réunies dans le *package* appelé **segment**.

Figure 5 : Avec nos segments, nous pouvons maintenant créer plein de motifs !

7 CONNEXION DES SEGMENTS

Nous disposons des primitives graphiques qu'il reste à assembler, configurer et connecter aux fils de la simulation.

- L'assemblage consiste ici à réunir nos primitives (les segments) pour constituer l'afficheur virtuel. Elles devront être placées à des coordonnées relatives bien précises pour former un chiffre correct.
- La configuration consiste à modifier les paramètres d'affichage pour adapter le graphisme à nos désirs. Par exemple, on voudrait régler les coordonnées, la largeur et la hauteur des segments de l'afficheur.
- La connexion consiste à déclencher la fonction appropriée lorsque le fil correspondant change d'état, le tout dans une structure VHDL adéquate.

La connexion va s'effectuer en utilisant la syntaxe présentée dans ces pages en juin **[2]** avec la fonction **observe**. Elle exploite le fait que dans une section de code concurrent (dans une architecture mais hors d'un process), on peut automatiquement créer un process en écrivant la fonction directement. Elle sera appelée à chaque fois qu'un des paramètres change.

```
architecture observe of seg7 is
begin
  segment_horizontal(x, y, longueur, épaisseur, seg(0));
  -- un changement d'état de seg(0) provoquera l'exécution de cette procédure
  ...
end observe;
```

Pour éviter de surcharger le simulateur, il faut s'assurer que chaque segment est réaffiché uniquement lorsque la valeur du fil change, pas à chaque fois qu'un des fils de l'afficheur change. C'est pourquoi l'afficheur prend la forme d'une entité VHDL (une sorte de composant réutilisable) dans laquelle on peut placer des process et du code concurrent (nos fonctions d'affichage, donc). Chaque segment sera donc contrôlé par une fonction écrite sous forme concurrente, afin que tous les segments soient indépendants les uns des autres.

Ensuite, il faut assembler les segments. C'est une étape manuelle très répétitive (modification du code, compilation, exécution, vérification et on recommence) nécessaire pour obtenir les bonnes coordonnées. Pour éviter de tout reprendre à chaque fois qu'on veut changer un paramètre (tels l'écartement des segments, la longueur, etc.), le code a été écrit dès le départ en intégrant ces paramètres, en évitant d'utiliser toute constante. Ainsi, on peut adapter l'afficheur à toutes les applications futures.

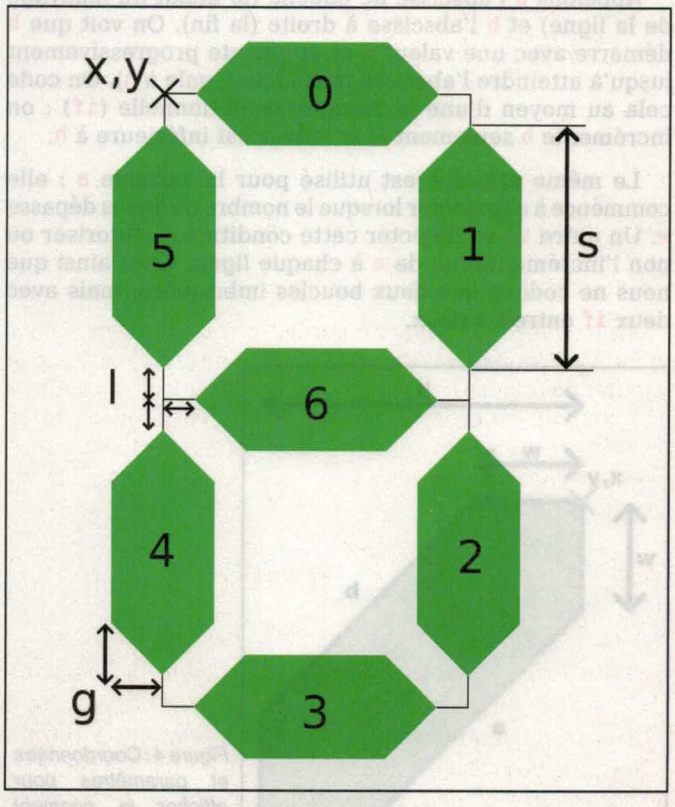


Figure 6 : Placement et paramètres des segments

En ce qui concerne la configuration, les paramètres de rendu graphique sont fournis à l'entité grâce à l'interface bien pratique des génériques. Ils sont figés au début de la simulation et peuvent même disposer d'une valeur par défaut, qui convient dans la plupart des cas. Il ne faut pas oublier de mettre à jour les coordonnées par contre.

L'autre interface, celle plus connue des ports, accepte en entrée les sept fils de commande, réunis sous forme d'un vecteur. Le code complet de l'afficheur 7 segments est fourni ci-dessous.

```
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.segment.all;

entity seg7 is
  generic(
    x : integer;       -- abscisse du coin supérieur gauche
    y : integer;       -- ordonnée
    s : integer := 70; -- longueur des segments
    g : integer := 7;  -- largeur des segments
    l : integer := 3   -- espacement entre les segments
  );
  port(seg : in std_ulogic_vector(6 downto 0));
end seg7;

architecture observe of seg7 is
begin
  segment_horizontal(x+l    ,y             ,g,s, seg(0));
  segment_vertical  (x+l+s+l,y+l           ,g,s, seg(1));
  segment_vertical  (x+l+s+l,y+l+s+l+l     ,g,s, seg(2));
  segment_horizontal(x+l    ,y+l+s+l+l+s+l,g,s, seg(3));
  segment_vertical  (x      ,y+l+s+l+l     ,g,s, seg(4));
  segment_vertical  (x      ,y+l           ,g,s, seg(5));
  segment_horizontal(x+l    ,y+l+s+l       ,g,s, seg(6));
end observe;
```

Cela fonctionne maintenant très bien, mais il manque juste un petit détail : notre code ne vérifie pas la validité des paramètres génériques. Il faut par exemple s'assurer que la longueur d'un segment est supérieure à son épaisseur, ou d'autres conditions dont la combinaison n'aurait aucun sens, ou entraînerait un échec de la simulation.

L'interface générique ne permet pas d'effectuer un test d'intervalle, du moins directement. On ne peut pas non plus ajouter des clauses **assert** en plein milieu du bloc « entity ». On pourrait créer une fonction dont le résultat fournirait la valeur vérifiée, mais ce serait une solution désespérée, il est possible de faire mieux.

En effet, j'ai découvert récemment que le VHDL autorise aussi l'écriture d'un process (contenant du code séquentiel, donc des **assert**) dans le bloc **entity**. Ce n'est possible qu'avec un **passive process** (process passif), c'est-à-dire qu'il n'effectue pas d'assignation de signal, puisque ce type de process est exécuté une fois, juste avant le début de la simulation. Il ne peut donc pas recevoir ou envoyer de signaux, c'est juste un confort syntaxique et sémantique supplémentaire pour la simulation.

Dans notre cas, ce process passif est la solution idéale pour ajouter des vérifications qui seront effectuées une fois pour toutes, puisque les coordonnées et les paramètres de l'affichage ne changeront pas ensuite. On peut y mettre autant de code que l'on veut pour tester des conditions aussi complexes que nécessaires.

Pour s'assurer que le process est écrit après la partie générique, on met le process après le mot-clé **begin**, que l'on utilise que rarement.

```
use work.fb_ghdl.all; -- on utilise fbx, fby...

entity seg7 is
  generic(
    ...
  );
  port(seg : in std_ulogic_vector(6 downto 0));
begin
  process is
  begin
    -- 1 < g < s
    assert 1 < g                 report "segment trop fin" severity failure;
    assert g < s                 report "segment trop épais" severity failure;
    -- 0 < l < s
    assert 0 < l           report "espacement trop faible" severity failure;
    assert l < s           report "espacement trop grand"  severity failure;
    -- 3 < s < fby
    assert 3 < s              report "segment trop long"  severity failure;
    assert s < fby            report "segment trop court" severity failure;
    -- l < x < fbx1-(s+l)
    assert l < x          report "abscisse trop à gauche" severity failure;
    assert x < fbx1-(s+l) report "abscisse trop à droite" severity failure;
    -- l < y < fby1-(s+l+l+s+l)
    assert l < y             report "ordonnée trop haute" severity failure;
    assert y < fby1-(2*s+3*l) report "ordonnée trop basse" severity failure;

    -- à ne pas oublier, comme pour tout process sans liste de sensibilité :
    wait;
  end process;
end seg7;

architecture observe of seg7 is
begin
  ...
```

8 EXEMPLE D'UTILISATION

Nous disposons maintenant de suffisamment d'éléments pour mettre en œuvre un afficheur complet, qui permettra plus tard de simuler une montre. Cela consiste d'abord à créer une nouvelle architecture VHDL où on met en place quatre chiffres et deux points. Pour les chiffres, on instancie quatre exemplaires de l'entité **seg7** que nous venons d'écrire.

Pour les points, on utilise la fonction **segment_point()** en syntaxe concurrente (comme dans le corps de l'architecture du chiffre à 7 segments).

Ensuite, pour faire « vivre » notre circuit, il faut lui fournir des informations à afficher, et cela à une vitesse suffisamment lente pour être distinguée par l'utilisateur.

Pour ralentir la simulation, nous réutilisons le code présenté dans ces pages en juin **[2]**. Nous instancions une entité `work.rt_clk`, un générateur d'horloge (à peu près) synchronisé avec l'horloge de l'ordinateur, que nous configurons avec l'intervalle désiré, ce qui rafraîchit l'affichage 10 fois par seconde.

Pour fournir des informations, j'ai choisi de créer des données pseudo-aléatoires (je ne vais quand même pas mâcher le travail de conception de la montre de Laura). Et il y a un moyen très facile pour réaliser cela avec un circuit électronique ou un programme informatique, du moment qu'on ne s'attend pas à une séquence imprévisible : c'est le registre à décalage à rétroaction linéaire, ou simplement LFSR pour les initiales du terme anglais original (*Linear Feedback Shift Register*), un type de système qui a déjà été étudié dans ce magazine en 2006 **[3]**.

Quand je dis facile, c'est qu'il n'est mathématiquement pas possible de faire plus simple : il suffit d'effectuer un XOR entre quelques bits. Le code utilise une configuration de Fibonacci : on décale tous les bits d'un cran et on remplit le bit libéré avec le résultat du XOR entre certains des autres bits.

Tout l'enjeu réside dans le choix des bits à utiliser, et cela touche des notions qui ont fait l'objet de plusieurs articles dans ce magazine. Heureusement, on dispose de tables telles que celle créée par Roy Ward et Tim Molteno **[4]**. Ils donnent un exemple d'utilisation avec une configuration de Galois, et nous savons qu'on peut utiliser directement les mêmes paramètres (les indices des bits à XORer entre eux) avec la configuration de Fibonacci.

Dans notre cas, nous avons besoin de 29 bits, 28 pour les 4 chiffres à 7 segments, plus un autre bit pour les deux points. La table **[4]** nous donne une possibilité en utilisant un XOR seulement entre les indices 29 et 27, mais cela ne donne pas un bon effet visuel car les bits ne sont pas suffisamment vite mélangés. On va donc se servir de l'autre possibilité, les indices 29, 28, 27, 25, qu'il faut décrémenter car nos vecteurs de bits commencent à l'indice 0. Et voilà, nous avons un petit générateur de bits brouillés, ce qui donne un effet qui rappelle le générique d'une série américaine racontant la journée d'un agent antiterroriste. Eh oui, on peut aussi faire cela en VHDL !

Figure 7 : Cet afficheur est complet, mais les segments sont contrôlés pseudo aléatoirement.

Pour faire fonctionner ce code, il faut disposer du code source de 2 packages décrits dans les mois précédents **[1] [2]**.

```
-- fichier : test_seg.vhdl
-- créé dim. juin 20 10:35:28 2010 (whygee@f-cpu.org)
-- version jeu. sept. 30 22:23:00 CEST 2010

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.rt_utils.all; -- l'horloge temps réel
use work.segment.all;  -- l'afficheur à 7 segments

entity test_7seg is
  -- pas d'entrées-sorties
end test_7seg;

architecture test of test_7seg is
  -- contrôle de l'horloge temps réel
  signal clk_rt, stop_rt: std_ulogic:='0';
  -- les fils à afficher :
  signal segs : std_ulogic_vector(28 downto 0);
begin

 -- horloge temps réel pour ralentir la simulation
  horloge : entity work.rt_clk
    generic map(ms => 50) -- 1/(2x50ms)=10 Hz
    port map(clk => clk_rt, stop => stop_rt);

  rt: process
    variable i, j : integer;
  begin
    -- initialisation
    segs <= (0=>'1', others=>'0');

    -- fait clignoter les segments pseudo-aléatoirement
    for i in 0 to 400 loop   -- (on ne va pas y passer la journée non
plus)
      wait until rising_edge(clk_rt);

      -- LFSR à 29 bits
      segs <= segs(27 downto 0) & (segs(28) xor segs(27) xor segs(26)
xor segs(24));
    end loop;

    stop_rt <= '1'; -- arrête l'horloge pour terminer la simulation
    wait;
  end process;

  -- l'affichage :

  display1: entity work.seg7
    generic map(x=>50, y=>50)    port map(seg => segs( 6 downto 0));
  display2: entity work.seg7
    generic map(x=>170, y=>50)   port map(seg => segs(13 downto 7));
  segment_point(275,85, 15, segs(14));
  segment_point(275,155,15, segs(14));
  display3: entity work.seg7
    generic map(x=>320, y=>50)   port map(seg => segs(21 downto 15));
  display4: entity work.seg7
    generic map(x=>440, y=>50)   port map(seg => segs(28 downto 22));
end test;
```

Et pour compiler et lancer le code, il faut encore quelques incantations magiques, expliquées dans les articles précédents (dont le script suivant est dérivé) :

```
#!/bin/bash
gcc -c fb_ghdl.c -o fb_ghdl_c.o &&
gcc -c rt_functions.c &&
ghdl -a fb_ghdl.vhdl rt_clk.vhdl segment.vhdl test_7seg.vhdl &&
ghdl -e -Wl,fb_ghdl_c.o -Wl,rt_functions.o test_7seg &&
./test_7seg
rm *.o work-obj93.cf
```

CONCLUSION

Cet article a détaillé l'écriture d'un paquetage de code VHDL qui permet de réaliser un afficheur 7 segments. Son utilisation est totalement transparente puisqu'il suffit d'instancier l'entité de l'afficheur puis y connecter les fils pour obtenir un affichage fonctionnel.

Comme d'habitude, les codes sources présentés ici sont disponibles à **http://ygdes.com/GHDL/**. ■

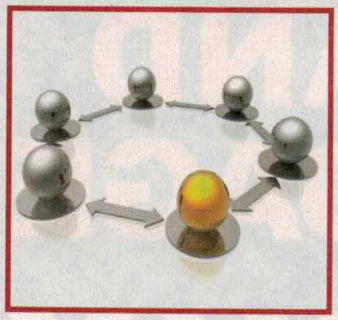
Références

- **[1]** Guidon (Yann), « Affichage graphique avec le framebuffer et GHDL », *GNU/Linux Magazine France* n°130, septembre 2010, http://www.ed-diamond.com/produit.php?ref=lmag130
- **[2]** Guidon (Yann), « Simulation à vitesse réelle avec GHDL », *GNU/Linux Magazine France* n°128, juin 2010, pp. 50-67, http://www.ed-diamond.com/produit.php?ref=lmag128
- **[3]** 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
- **[4]** 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

Auteur : Yann Guidon

- Electronique, musique et informatique en folie^Wliberté

Ivy, un bus logiciel simple et



Lorsqu'on veut faire communiquer un ensemble d'applications, l'une des solutions les plus évidentes est celle consistant à utiliser un bus logiciel. Un très bon exemple d'implémentation est D-Bus, généralement en place sur la quasi-totalité des ordinateurs desktop GNU/Linux, qu'ils soient sous GNOME ou KDE. Mais D-Bus n'est pas toujours la solution idéale pour le développeur. Ici, nous traiterons d'une implémentation reposant sur une API bien plus simple et plus rapide à utiliser. C'est Ivy.

Auteur

- Denis Bodor

Posons clairement les choses. Si l'on veut mettre en place des applications communicantes ou ajouter un peu de code dans ce sens à une application, D-Bus est souvent un chemin qu'il est difficile d'emprunter. Même si son principe de fonctionnement est simple, D-Bus nécessite la mise en place d'un démon et le respect de toute une logique d'enregistrement auprès du bus. De ce fait, l'API ne permet pas vraiment de rapidement obtenir un résultat fonctionnel. Ce que propose D-Bus est sans doute parfaitement adapté aux rouages d'un environnement de bureau, mais souvent trop « décalé » ou trop lourd pour mettre en œuvre quelque chose se rapprochant davantage des IPC.

Ivy propose une alternative intéressante de bus logiciel (*message bus* ou *message-oriented middleware* en anglais) en visant la simplicité avant tout :

- pas de démon ;
- pas de système d'enregistrement lourd ;
- un fonctionnement réseau implicite et transparent ;
- un tri des messages par chacun des clients du bus via des expressions rationnelles ;
- une API simple ;
- un support pour de nombreux langages (C, C++, Java, Perl, Caml, Python, Ruby, Ada, C#).

Ivy, développé et utilisé depuis 1996 par le CENA (Centre d'Etudes de la Navigation Aérienne), vise donc le *rapid development* et la facilité de mise en œuvre. Le bus est entièrement distribué et tire son inspiration (utilisation d'expressions rationnelles et *broadcast* des messages) de KoalaTalk (Cédric Beust / INRIA), lui-même s'inspirant du ToolTalk de Sun. La version 3 du protocole date de 1996 et celui-ci a très peu changé depuis lors. Ivy est utilisé aussi bien au CENA pour le développement des interfaces utilisateurs et les simulateurs de contrôle du trafic aérien, qu'au niveau européen à l'ECC (*Eurocontrol Experimental Center*) ou encore avec des logiciels propriétaires de gestion des bateaux de pêche. Ivy est, en effet, sous licence LGPL.

Le CENA de la DTI de la DSNA dans la DGAC ?

Le CENA fait partie du domaine R&D de la DTI (Direction de la Technique et de l'Innovation) de la DSNA (Direction des Services de la Navigation Aérienne), qui est le fournisseur de services du contrôle aérien au sein de la DGAC (Direction Générale de l'Aviation Civile). Preuve est faite qu'il n'y a pas qu'en informatique où on aime les acronymes. CQFD.

1 INSTALLATION ET PREMIER CODE

Des paquets binaires sont disponibles au téléchargement sur le site officiel (http://www2.tls.cena.fr/products/ivy/) pour de nombreuses distributions GNU/Linux (Debian, Mandriva, Red Hat, etc.) ainsi que des versions installables pour d'autres systèmes (Mac OS X, Windows, Solaris). Pour Debian, vous devrez installer les paquets `ivy-c_3.11.4_i386.deb` et `ivy-c-dev_3.11.4_i386.deb` afin de disposer des bibliothèques ainsi que des fichiers d'en-tête adéquats. Notez que des dépendances à des paquets comme `tcl8.4` existent en raison de la présence de « pont » vers les fonctions `mainloop` de différent *toolkits* comme Xt, GTK+/Glib ou encore TCL/Tk.

Autre point intéressant, un profil de *package* pour `pkg-config` est automatiquement installé sous le nom `ivy-glib`.

souple

La compilation pourra ainsi se faire plus facilement via un `pkg-config ivy-glib --libs --cflags` même si certains chemins utilisés sont un peu étranges, comme `-I/usr/local/include`. Ces paquets, du moins pour Debian, vous fourniront une base pour développer et essayer Ivy, mais force est de constater qu'il sera plus raisonnable, dans le cas d'une utilisation en production, de récupérer les sources de la bibliothèque via SVN et de reconditionner le paquet source (voire en créer un nouveau). On pourra ainsi obtenir quelque chose de bien plus léger reposant alors uniquement sur la bibliothèque PCRE3 et la glibc.

Une fois les paquets binaires officiels installés, vous pouvez vous lancer dans votre première application Ivy qui sera, vous vous en doutez, simplissime.

- On inclut tout d'abord les en-têtes classiques ainsi que celui d'Ivy. Notez qu'il est de bon ton d'aller faire un tour dans `/usr/include/ivy` car bon nombre de fonctions ne sont pas clairement documentées :

```
#include <getopt.h>
#include <Ivy/ivy.h>
#include <Ivy/ivyloop.h>
```

- Suivent ensuite les définitions des fonctions de *callback* avec, dans l'ordre, une fonction `HelloCallback` destinée à être appelée lors de la réception du message `Hello`. Le prototype d'une fonction callback Ivy est simple. En argument, on trouve un pointeur vers les informations sur l'émetteur du message ayant déclenché l'appel, un autre pointeur sur des données « utilisateurs », un compteur d'arguments, et enfin, un pointeur sur un tableau de pointeurs de chaînes contenant les arguments, le même principe que pour une fonction `main()` classique. Notre fonction utilise une fonction `IvySendMsg` permettant d'envoyer un message (broadcasté) sur le bus :

```
void HelloCallback (IvyClientPtr app,
  void *data, int argc, char **argv)
{
  IvySendMsg ("Bonjour");
}
```

- Nous procédons de même pour un second callback afin de répondre au message `Bye` qui retournera également un message sur le bus avant de stopper la boucle principale d'Ivy avec `IvyStop()`. Nous verrons plus loin dans l'article qu'il est également possible de reposer sur une boucle externe comme celle de GTK+, Xt ou Ecore :

```
void ByeCallback (IvyClientPtr app,
  void *data, int argc, char **argv)
{
  IvySendMsg ("Time to quit");
  IvyStop();
}
```

- Nous arrivons à `main()` qui, pour répondre aux conventions d'Ivy, doit impérativement analyser le contenu d'un éventuel argument `-b` ou `-bus` spécifiant le bus à utiliser. Si cet argument n'est pas utilisé, c'est le contenu de la variable d'environnement `IVYBUS` qui sera évalué à son tour. Enfin, c'est le bus `127:2010` (réseau 127.255.255.255, port 2010) qui sera utilisé par défaut :

```
int main (int argc, char**argv)
{
  const char* bus = 0;
  char c;

  while ((c = getopt (argc, argv, "b:")) != EOF) {
    switch (c) {
      case 'b':
        bus = optarg;
        break;
    }
  }

  if (!bus)
    bus = getenv ("IVYBUS");
```

- Enfin, nous trouvons l'initialisation du bus et sa mise en œuvre avec `IvyInit`. Les arguments sont, dans l'ordre, le nom de l'application sur le bus, un pointeur vers la chaîne de caractères du premier message envoyé automatiquement, un pointeur vers le callback appelé en cas d'arrivée d'un nouveau logiciel sur le bus et un argument utilisateur, et enfin, un point vers le callback appelé quand une application quitte le bus et son argument utilisateur. Les quatre derniers arguments peuvent être `NULL` si l'on n'utilise pas ces fonctionnalités. Il en va de même avec le message initial sachant que, dans ce cas, on ira à l'encontre de la convention du protocole. Le démarrage du bus se fait tout simplement par `IvyStart()`. L'argument passé est le bus qu'on souhaite utiliser. Si ce paramètre est `NULL`, la variable `IVYBUS` sera utilisée ou, par dépit, `127:2010` :

```
  IvyInit ("IvyLefinnois", "IvyLefinnois ready",
    NULL, NULL, NULL, NULL);
  IvyStart (bus);
```

- Nous installons ensuite nos deux callbacks. Nous associons directement, en troisième argument, une expression rationnelle pour procéder au filtrage. Le second argument est, une fois encore, un pointeur sur d'éventuelles données utilisateurs à passer au callback (cf `*data`). Il s'agira souvent d'un pointeur sur une structure complète contenant une configuration ou des données modifiées par ailleurs (ou rien du tout) :

```
  IvyBindMsg (HelloCallback, NULL, "^Hello$");
  IvyBindMsg (ByeCallback, NULL, "^Bye$");
```

- Tout est prêt. Il ne nous reste plus qu'à entrer dans la boucle dont nous sortirons avec notre appel à **IvyStop()** depuis le callback **ByeCallback()** :

```
  IvyMainLoop();
  return(EXIT_SUCCESS);
}
```

La compilation se fera très simplement avec :

```
% gcc -O2 -Wall-o firstivy firstivy.c -livy
```

L'essai de notre première application se fera en lançant le moniteur de bus **ivyprobe** installé avec le paquet **ivy-c** :

```
% ivyprobe '(.*)'
Broadcasting on network 127.255.255.255, port 2010
```

L'argument **(.*)** est, vous l'aurez compris, une expression rationnelle permettant de filtrer les messages (ici, nous les acceptons tous). Notez que l'absence d'argument ne provoque pas la prise en compte de tous les messages, mais d'aucun. Encore une fois, par convention, l'option **-b** ou la variable d'environnement **IVYBUS** nous permettent de spécifier le bus utilisé.

Nous lançons ensuite notre application :

```
% ./firstivy
Broadcasting on network 127.255.255.255, port 2010
```

Immédiatement, le terminal dans lequel a été lancé le moniteur nous affiche la notification de connexion ainsi que le message initial. Nous en profitons pour envoyer à notre tour un message sur le bus :

```
IvyLefinnois connected from localhost
IvyLefinnois sent  'IvyLefinnois ready'
Hello
-> Sent to 1 peer
```

Côté application « maison », on nous informe de la réception du message avec **Got Hello msg**. Le callback est appelé et on retourne le message adéquat :

```
IvyLefinnois sent  'Bonjour'
Bye
-> Sent to 1 peer
IvyLefinnois sent  'Time to quit'
IvyLefinnois disconnected from localhost
```

Nous venons de réitérer avec le message **Bye** provoquant la fin de l'application et la déconnexion au bus. Notre premier essai d'Ivy est un succès. Nous sommes loin de la complexité de D-Bus. N'est-ce pas ? En spécifiant un **IVYBUS** différent, comme **192.168.10:2010**, notre bus « sort » alors du *localhost* et nos différents clients n'y voient strictement aucune différence. Les messages circulent ainsi via le port UDP/2010.

2 FONCTIONNALITÉS AVANCÉES

Nous venons de voir l'utilisation la plus basique d'Ivy. Il est cependant possible de pousser un peu plus loin sans vraiment pénaliser le développement rapide. Dans un premier temps, nous pouvons modifier notre callback **HelloCallback** de manière à ce qu'il puisse accepter un éventuel argument :

```
void HelloCallback (IvyClientPtr app,
   void *data, int argc, char **argv)
{
  const char* arg = (argc < 1) ? "" : argv[0];
  printf("Got Hello msg\n");
  IvySendMsg ("Bonjour %s", arg);
}
```

Encore une fois, Ivy montre sa simplicité de mise en œuvre puisque nous procédons exactement comme avec une classique fonction **main()**. Il vous suffira de correctement structurer vos expressions rationnelles pour assurer la bonne prise en charge et le traitement de vos messages et arguments.

Ivy met également à disposition des fonctions utilitaires supplémentaires qui peuvent être intéressantes si vous ne souhaitez pas lier votre application avec d'autres bibliothèques. Ainsi, la gestion de *timers* est intégrée, permettant d'appeler à intervalle régulier des fonctions. L'implémentation d'un timer se fera ainsi :

```
void TimerCallback(TimerId id,
   void *data, unsigned long delta)
{
  printf("Tic\n");
}
```

Là encore, diverses données utilisateurs peuvent être passées via ***data**. À noter également qu'un delta (écart entre l'instant où la fonction est appelée et celui où elle devait l'être) est automatiquement passé en argument sous la forme d'un **long** non signé exprimé en millisecondes. L'installation d'un nouveau timer se fera ainsi :

```
TimerId firsttimer;
firsttimer = TimerRepeatAfter(TIMER_LOOP,
  500, TimerCallback, NULL);
```

Ainsi, dès l'entrée dans la boucle principale Ivy, **TimerCallback** sera appelé toutes les demi-secondes (500ms). Le premier argument de **TimerRepeatAfter** est le nombre de répétitions de l'appel. Ici, **TIMER_LOOP** (défini à **-1**) permet un nombre de répétitions infini.

Enfin, en ne reposant que sur Ivy, il est fort probable que votre application doive également gérer d'autres entrées/sorties (fichiers, console, réseau, etc.). Pour cela, la bibliothèque permet de mettre en place des canaux (*channels*) à l'instar de ce que proposent des toolkits comme GTK+/Glib avec **g_io_channel_unix_new()**. C'est alors Ivy qui se charge de gérer les événements survenant sur le descripteur de fichiers utilisé. La mise en place d'un nouveau canal se fera via :

```
Channel IvyChannelAdd (fd, DelFD, NULL, ReadFD, NULL);
```

Les fonctions **DelFD** et **ReadFD** seront implémentées ainsi (si **fd** vaut **0** pour **stdin**, par exemple) :

```
void DelFD (Channel channel, void *data)
{
  IvyChannelRemove (channel);
}

void ReadFD (Channel channel,
  HANDLE fd, void *data)
{
  char buf[4096];
  char *line;
  char *cmd;

  line = fgets(buf, 4096, stdin);
  if(!line)
  {
    printf("Ligne vide... Bye\n");
    IvyChannelRemove (channel);
    IvyStop();
    return;
  }
  cmd = strtok (line, "$: \n");
  printf (":> %s\n", line);
  if(strcmp (cmd, "die") == 0)
  {
    printf("DIE !!!!\n");
    IvyStop();
  }
}
```

Nous avons ici un exemple d'utilisation d'`IvyChannelRemove()` permettant de stopper la gestion sur le descripteur en passant en paramètre le canal concerné. Notez au passage que la documentation présente sur le site officiel comporte une erreur puisque le prototype donné pour `IvyChannelAdd()` est :

```
Channel IvyChannelAdd (HANDLE fd,
void* data,
ChannelHandleDelete handle_delete,
ChannelHandleRead handle_read);
```

Voici pour l'utilisation d'Ivy dans le cadre d'un développement d'application en ligne de commandes sans grande prétention. Les fonctions utilitaires proposées permettent d'assurer un minimum de fonctionnalités. En ce qui me concerne, l'utilisation d'Ivy sera plus poussée puisque j'entends utiliser ce bus afin de faire communiquer, *over-UDP*, différents modules domotiques « faits maison » à base de Fonera, cartes PC/104 et autres. Certains de ces modules, construits autour d'un ARM9/s3c2440, doivent accueillir une IHM graphique via un écran LCD 3.5" tactile. La programmation du *framebuffer* n'étant pas ma tasse de thé (et ce n'est pas faute d'avoir essayé), l'utilisation d'un canvas et/ou d'un toolkit paraît évidente. Là, une problématique spécifique se met en place. Celle consistant à faire travailler, de concert, `IvyMainLoop()` et la boucle principale du canvas/toolkit.

3 INTÉGRATION DANS UNE MAIN LOOP EXISTANTE : L'EXEMPLE EVAS/ECORE

De base, le paquet `ivy-c_3.11.4` installe non seulement `libivy.so.3`, mais également trois autres bibliothèques dynamiques. Chacune d'elles correspond à une `mainloop` spécifique :

- `libgivy.so.3` pour Glib et une utilisation pour le développement d'applications graphiques GTK+ ;
- `libtclivy.so.3` pour Tcl ;
- `libxtivy.so.3` pour Xt et donc Motif et Xaw.

Les sources SVN des bibliothèques C intègrent également le support pour GLUT, le toolkit OpenGL. Je vous ferai grâce ici de l'exemple d'utilisation avec GTK+, par exemple, puisque celui-ci est parfaitement détaillé dans la documentation officielle d'Ivy. Sachez que ceci se limite à utiliser exactement le même code que précédemment, à l'exception de l'appel à `IvyMainLoop()`, remplacé par le classique `gtk_main()`. On prendra simplement soin de lier son programme à la bonne bibliothèque.

Bien entendu, les développeurs d'Ivy n'ont pas développé de bibliothèques pour tous les toolkits imaginables. En revanche, ils fournissent une méthode permettant de procéder soi-même à ce développement. Pour cela, tout ce que nous avons à faire est de définir ou redéfinir un certain nombre de fonctions :

- `IvyChannelInit`, contenant l'initialisation avant l'entrée dans la boucle principale ;
- `IvyChannelStop`, regroupant les opérations à effectuer avant de quitter la boucle ;
- `IvyChannelAdd`, permettant de lier les canaux du toolkit utilisé par Ivy ;
- `IvyChannelRemove`, pour retirer cette gestion.

La logique de « canal » est identique à celle vue précédemment, et pour cause, puisque nous devons implémenter les fonctions en question. Le cœur d'Ivy se chargera de gérer les communications réseau via des *sockets* Unix et donc des descripteurs de fichiers qu'il nous suffit de surveiller. La logique consiste donc à laisser Ivy gérer les sockets et à utiliser les fonctions du toolkit de son choix pour déclencher des fonctions de callback qui, elles, passeront les données à Ivy.

Avec GTK+/Glib, par exemple, cette gestion de *file handle* consiste, dans notre fonction `IvyChannelAdd`, à utiliser le descripteur de fichiers reçu en argument pour créer un *GIOChannel*. On ajoute ensuite deux points de surveillance avec `g_io_add_watch`, déclenchant l'appel aux fonctions qu'Ivy nous aura fournies, en indiquant un pointeur vers les données à traiter.

Une structure `_channel` est utilisée pour stocker les informations non seulement pour Ivy (adresses des fonctions de gestion lecture/écriture/destruction), mais également pour notre implémentation.

EFL, Ecore, Evas...

Ecore et Evas sont respectivement deux bibliothèques chargées de la gestion d'événements et de la manipulation de canvas dans les EFL (*Enlightenment Foundation Libraries*). Constituant non seulement, avec Edje, Embryo ou encore Eina, la base logicielle d'Enlightenment e17, les EFL permettent également des développements indépendants. On les retrouve, par exemple, dans le monde de l'embarqué (Canola, OpenMoko, OpenInkPot), mais également dans le multimédia avec GeexBox ou dans votre salon si vous disposez d'une Freebox HD. Le *binding* JavaScript pour les EFL, appelé Elixir, sert en effet de base pour le support de jeux sur la « *box* » en question.

En fonction des toolkits, la logique et la structure du code pourront être très différentes. Ainsi, avec Ecore, nous ne disposons pas de la possibilité de dispatcher les types d'événements directement dans l'API comme c'est le cas avec GTK+/Glib. Nous devons donc appeler une seule fonction de callback qui se chargera de faire le tri.

Pour cela, nous commençons par définir notre structure **_channel** ainsi :

```
struct _channel {
  // le handler Ecore
  Ecore_Fd_Handler *id_fdh;
  // données à transmettre
  void *data;
  // les trois fonctions de gestion
  ChannelHandleDelete handle_delete;
  ChannelHandleRead handle_read;
  ChannelHandleWrite handle_write;
  // hack pour le handle en écriture
  int enablewritehandle;
  // stockage du descripteur
  HANDLE fd;
};
```

Ensuite vient la fonction la plus conséquente avec, dans un premier temps, l'allocation de la mémoire pour notre structure **Channel** :

```
Channel IvyChannelAdd(HANDLE fd, void *data,
                      ChannelHandleDelete handle_delete,
                      ChannelHandleRead handle_read,
                      ChannelHandleWrite handle_write
                      )
{
  Channel channel;

  channel = (Channel)malloc( sizeof(struct _channel) );
  if ( !channel )
  {
    printf("malloc failed !\n");
    exit(EXIT_FAILURE);
  }
```

Nous initialisons immédiatement cette structure avec les informations à notre disposition directement fournies par Ivy :

```
  channel->handle_delete = handle_delete;
  channel->handle_read = handle_read;
  channel->handle_write = handle_write;
  channel->data = data;
  channel->fd = fd;
  channel->enablewritehandle=0;
```

Vient ensuite la partie où Ecore intervient et où nous mettons en place la fonction de callback de gestion avant de conclure :

```
    channel->id_fdh = ecore_main_fd_handler_add (fd,
      ECORE_FD_READ |
      ECORE_FD_ERROR |
      ECORE_FD_WRITE,
      MyEcoreFdCB, channel, NULL, NULL);
  }
  return channel;
}
```

ecore_main_fd_handler_add est relativement simple à utiliser et prend en arguments le descripteur de fichier, des drapeaux (*flags*) utilisés en guise de masque pour filtrer les événements ainsi qu'une fonction callback et un pointeur sur les données à lui transmettre. Ne vous y trompez pas, les drapeaux ne font office que de filtres et vous ne pouvez pas utiliser plusieurs fois **ecore_main_fd_handler_add** pour un même descripteur. Cela ne fonctionne pas comme le **g_io_add_watch** de GTK+/Glib.

Notre fonction callback de tri ressemblera alors à ceci :

```
int MyEcoreFdCB(void *ldata, Ecore_Fd_Handler *fdh)
{
  Channel channel = (Channel)ldata;

  if ( ecore_main_fd_handler_active_get(fdh, ECORE_FD_READ) )
    (*channel->handle_read)(channel,
      ecore_main_fd_handler_fd_get(fdh), channel->data);
  if ( ecore_main_fd_handler_active_get(fdh, ECORE_FD_ERROR) )
    (*channel->handle_delete)(channel->data);
  if ( ecore_main_fd_handler_active_get(fdh, ECORE_FD_WRITE)
       && channel->enablewritehandle )
    (*channel->handle_write)(channel,
      ecore_main_fd_handler_fd_get(fdh), channel->data);
  return TRUE;
}
```

On teste avec **ecore_main_fd_handler_active_get** quel drapeau est levé et on appelle la fonction fournie par Ivy en argument de **IvyChannelAdd** en lui passant la structure, le descripteur de fichier récupéré avec **ecore_main_fd_handler_fd_get()** et les données. Les fonctions stockées dans **handle_read**, **handle_delete** et **handle_write** ne prennent pas toutes les mêmes arguments. On notera également le test de **channel→enablewritehandle**, permettant ou non d'activer l'appel à **handle_write**.

En effet, deux fonctions supplémentaires doivent être implémentées, mais semblent absentes de la documentation :

- **IvyChannelAddWritableEvent** ;
- **IvyChannelClearWritableEvent**.

Elle permettent respectivement l'ajout et la suppression des canaux pour l'écriture et prennent toutes deux en argument un pointeur sur une structure **_channel**. Ces deux fonctions, ici, changent simplement la valeur de **enablewritehandle** pour satisfaire le test dans le callback. Ces fonctions n'étant pas documentées, nous avons pris exemple, tout simplement, sur l'implémentation GTK+/Glib utilisant **g_io_add_watch**/**g_source_remove**.

Les autres fonctions à implémenter sont :

- **IvyChannelInit**, honteusement copiée des autres implémentations :

```
void IvyChannelInit(void) {
  if ( channel_initialized ) return;
  /* fixes bug when another
     app coredumps */
  channel_initialized = 1;
}
```

- **IvyChannelRemove**, permettant de nettoyer la prise en charge par Ecore :

```
void IvyChannelRemove( Channel channel )
{
  if (channel->handle_delete)
     (*channel->handle_delete)( channel->data );
  ecore_main_fd_handler_del(channel->id_fdh);
}
```

- Et **IvyChannelStop**, qui, également reprise des autres implémentations, est un peu surprenante :

```
IvyChannelStop ()
{
  /* To be implemented */
}
```

La méthode la plus simple pour intégrer ce support Ecore dans Ivy est tout simplement de modifier directement les données dans les paquets sources de la distribution qui nous concerne (ici Debian). En modifiant ainsi sensiblement le contenu du répertoire **src/** ainsi que des fichiers **Makefile** et **debian/***, nous pouvons obtenir directement un paquet binaire adapté à nos nouveaux besoins. Une autre solution est d'écraser (*override*) les appels aux fonctions Ivy, soit en intégrant nos fonctions directement dans notre application, soit en utilisant **LD_PRELOAD**. Vous conviendrez que la première solution est sans doute la plus élégante, outre le fait de « nettoyer » les sources originales des dépendances à TCL, GTK+, etc. Et pourquoi pas en profiter pour recréer un paquet source plus à jour vis-à-vis de debhelper (voir l'article sur la construction de paquets Debian dans *GLMF* HS 48).

En guise de test, il nous suffit de construire une application simple reposant sur Ecore, Evas et/ou Edje, tout en utilisant classiquement Ivy, comme dans notre premier exemple. Inutile bien entendu de faire appel à **IvyMainLoop()**, le cœur du programme reposant alors sur la boucle **ecore_main_loop_begin()**.

L'exemple d'intégration donné ici sera facilement adaptable à n'importe quel toolkit. Certes, le gros du travail est déjà fait avec GTK+, Tcl, Glut, etc., mais des besoins spécifiques peuvent toutefois rendre cela nécessaire.

CONCLUSION : LES LIMITES D'IVY

Ivy n'est pas conçu, à l'origine, pour les réseaux publics ou LAN avec accès externe. En conséquence, aucun mécanisme de sécurisation des échanges n'est prévu. Implémenter un réseau d'applications (interfaces, capteurs, services, etc.) utilisant Ivy consiste à mettre en place des composants sûrs et contrôlables dans un environnement qui l'est tout autant. Des directives du type **IvySendDieMsg(app)**, permettant de tuer une application via la couche Ivy, sont donc très dangereuses. Il sera ainsi possible pour une personne malintentionnée de forger un paquet UDP dans ce sens ou, bien plus simplement, de développer quelques lignes de code pour tuer l'intégralité des applications présentes sur le bus logiciel.

De la même manière et pour les mêmes raisons, aucune implémentation n'existe en ce qui concerne l'authentification des applications sur le bus. On pourra bien entendu développer cela du côté applicatif en mettant en place un agent chargé d'authentifier les clients du bus et de faire part du statut (autorisé ou non) à l'ensemble des applications connectées. Ceci aura pour effet de rendre critique un élément du bus et donc de perdre l'aspect décentralisé initial. Enfin, rejoignant la thématique de la sécurité, l'ajout d'une couche de chiffrement, via OpenSSL, par exemple, reste possible mais aucunement planifié à l'heure actuelle.

Comprenez bien qu'il ne s'agit là aucunement de critiques, mais de limitations inhérentes au cahier des charges de ce bus logiciel. La page officielle du projet nous informe qu'une évolution du protocole serait en préparation, reposant sur XML. Il nous a été confirmé, cependant, que ceci n'était pas vraiment à l'ordre du jour, Ivy répondant aux besoins de ses créateurs dans le cadre lui étant dédié actuellement.

Ivy brille par sa simplicité et son efficacité en RAD. On pourra contourner ses limitations via l'utilisation d'un VPN ou, pour les plus courageux, via l'évolution de l'implémentation et du protocole. Ceci aura pour effet de créer un *fork*, mais aussi et surtout de complexifier l'ensemble au risque de perdre le principal avantage de la solution originale. Il s'avérera alors judicieux de se demander si l'utilisation de D-BUS ne serait pas une meilleure solution. À chaque besoin son outil open source après tout... ■

Auteur : Denis Bodor

Développez avec des logiciels

Depuis maintenant un peu moins d'un an, Free offre la possibilité d'exécuter sur la Freebox des applications écrites en JavaScript utilisant les Enlightenment Foundation Libraries pour réaliser l'interface graphique.

Auteurs

- Cédric Bail
- Pierre-François Hugues

1 ENLIGHTENMENT ET LES EFL

Les EFL, *Enlightenment Foundation Libraries*, sont les bases d'un *toolkit* développé par l'équipe du projet Enlightenment depuis un peu plus de 10 ans, principalement pour le gestionnaire de fenêtres E17. Le projet E17 visant à fournir un gestionnaire de fenêtres à la fois joli, léger et rapide a donc mis de très gros efforts dans la réalisation d'une base saine, légère et rapide. Les EFL ont évolué dans le temps et sont maintenant destinées à la réalisation de tout type d'application ; à tel point qu'aujourd'hui, l'objectif est d'en faire un concurrent dans l'embarqué face aux poids lourds tels que QT ou GTK.

C'est ainsi que les EFL viennent de passer en version alpha, ce qui garantira une compatibilité API/ABI la plus longue possible dans le temps. La version 1.0 devrait être disponible d'ici peu, et l'objectif est que la version 2.0 (permettant éventuellement de « casser » l'ABI) ne sorte pas avant plusieurs années. Avec cette sortie, les EFL entrent définitivement dans la cour des grands et cassent un peu plus ce mythe de *vaporware*. D'ailleurs, pour le clin d'œil, Free a inclus *Duke Nukem 3D* dans le même *firmware* mettant à disposition **Elixir**, ce *framework* JavaScript qui utilise les EFL.

Les EFL ont été découpées en morceaux logiques pour en faciliter la compréhension et la maintenance :

Eina est la base de toutes les EFL. Une bibliothèque de types de données qui essaye de rendre le plus simple possible la manipulation efficace de ces types. Dans le cadre du projet Elixir et sur la Freebox, celle-ci n'est pas du tout exposée aux utilisateurs car le JavaScript a déjà, contrairement au C, ces fonctionnalités dans le langage.

Eet fournit une mécanique robuste et efficace pour sérialiser et enregistrer des données dans un fichier ou les envoyer sur le réseau. Cela inclut n'importe quelle structure mémoire, des images ou encore les scripts JavaScript d'Elixir.

Evas est un canvas graphique *stateful*. C'est réellement le cœur des EFL car c'est Evas qui est en charge de la gestion et de l'affichage des objets sur le canvas graphique. C'est cette bibliothèque qui optimise toutes les requêtes au système graphique pour tirer au mieux parti des performances du matériel.

Ecore est chargé de gérer tous les événements, qu'ils soient liés aux fenêtres graphiques (redimensionnement, déplacement, ...) ou au réseau.

Edje permet quant à elle de définir indépendamment du code de l'application son *look & feel*. Cela permet, par exemple, d'aussi bien définir le *layout* des menus d'une application que les animations d'un *sprite*. L'intérêt étant de pouvoir avoir plusieurs versions d'un thème s'adaptant le mieux possible au terminal cible, une véritable abstraction entre le code et l'interface

Sur ces bases vient se poser **Elixir**, qui expose directement l'API C des EFL en JavaScript grâce à la bibliothèque SpiderMonkey du projet Mozilla. Ce qui fait de cette solution un framework entièrement libre et ouvert, portable et intégrable facilement sur des plates-formes même très limitées. Pour pouvoir bénéficier d'un maximum d'exemples, de documentation et d'aide, il a été préféré de ne pas faire une API orientée objet : cela permet de réutiliser quasi directement les exemples en C.

2 ELIXIR

Il existe plusieurs manières de récupérer les EFL et SpiderMonkey. Ainsi, il existe un PPA pour Ubuntu, de Cédric Schieli, `ppa:cschieli/freebox-elixir`, des paquets sont aussi disponibles dans AUR pour Arch et il est bien entendu possible de tout compiler à la main en partant des sources. Pour ce dernier cas, nous vous laisserons consulter la documentation disponible sur http://code.google.com/p/freebox-elixir/wiki/InstallerElixir. Free met aussi à disposition

libres sur la Freebox !

une image pour machine virtuelle VirtualBox à l'adresse : ftp://ftp.free.fr/pub/elixir/Elixir_virtualbox.tar.gz. Ceci doit permettre de démarrer à développer sans trop perdre de temps sur des détails de compilation et d'installation.

Essayons donc, pour commencer, de réaliser un classique *Hello World!*. Tout d'abord, il faut charger les *bindings* nécessaires à notre exemple, c'est-à-dire Evas, Ecore et Ecore_Evas. Chaque module pouvant exister de manière optionnelle, il est considéré comme étant une bonne pratique de vérifier la valeur de retour de la fonction **elx.load**.

```
var test = true;

test &= elx.load("evas");
test &= elx.load("ecore");
test &= elx.load("ecore-evas");
```

Maintenant que les *bindings* sont activés, il faut initialiser les EFL, créer une fenêtre, ajouter le texte au canvas, puis enfin, attendre que l'utilisateur presse sur une touche avant de quitter. Il va donc falloir utiliser Ecore puis Evas. Regardons ce que cela donne :

```
var background;
var obj;

ecore_init();
ecore_evas_init();

// Création de la fenêtre
var ee = ecore_evas_new(null, 0, 0, 720, 576, "name=Hello World;");
// Obtention du canvas lié à la fenêtre
var evas = ecore_evas_get(ee);

// Définition des contraintes sur le canvas
evas_image_cache_set(evas, 10 * 1024 * 1024);
evas_font_path_prepend(evas, "/.fonts/");
evas_font_cache_set(evas, 512 * 1024);

// Création d'un background noir opaque
obj = evas_object_rectangle_add(evas);
evas_object_resize(obj, 720, 576);
evas_object_color_set(obj, 0, 0, 0, 255);
evas_object_show(obj);
background = obj;

// Création de l'objet texte
obj = evas_object_text_add(evas);
evas_object_text_text_set(obj, "Hello World!");
evas_object_text_font_set(obj, "Vera", 30);
evas_object_text_style_set(obj, EVAS_TEXT_STYLE_SOFT_SHADOW);
evas_object_text_shadow_color_set(obj, 128, 128, 128, 255);
evas_object_color_set(obj, 128, 64, 0, 180);
evas_object_resize(obj, 200, 50);
evas_object_move(obj, 100, 100);
evas_object_show(obj);
```

Comme Evas est un canvas *stateful*, nous créons les objets qui seront affichés à l'écran, puis on en change les propriétés pour obtenir le résultat voulu. Par contre, vous noterez la présence d'un rectangle noir explicitement défini pour le fond. Il est nécessaire, car Evas n'a pas de notion d'initialisation du fond d'écran avec une valeur quelconque. Donc, sans cet objet, Evas ne saurait pas quoi mettre comme contenu dans les pixels composant le fond. Vous pouvez tester en désactivant juste l'appel à **evas_object_show**, qui correspond au fond d'écran (par défaut, les objets ne sont pas visibles). Ce bout de code a aussi introduit deux types d'objets gérés par Evas : les rectangles et le texte. Evas gère aussi des images, des blocs de texte, des tables, des listes fixes (appelées *box*) et des objets composés de toutes ces primitives (appelés *smart objects*).

Ajoutons maintenant la gestion du clavier pour quitter l'application :

```
   evas_object_event_callback_add(background, EVAS_CALLBACK_KEY_UP,
key_up_cb, null);
   evas_object_focus_set(background, 1);

   ecore_evas_show(ee);
   ecore_main_loop_begin();
```

Voilà ! Nous avons maintenant attaché une *callback* sur l'objet **background** et défini qu'il recevrait les événements clavier puisqu'il possède le focus (**evas_object_focus_set**). Enfin, la fenêtre a été affichée juste avant de démarrer la boucle principale du programme. C'est cette boucle qui déclenchera au moment opportun la mise à jour du rendu à l'écran et transmettra les événements clavier à Evas sans bloquer l'interactivité de l'application. Détaillons donc le contenu de la fonction **key_up_cb** :

```
function key_up_cb(data, e, obj, event)
{
   switch (event.keyname)
     {
      case "b":
      case "Red":
      case "equal":
      case "Stop":
      case "Home":
      case "Escape":
      case "Start":
         ecore_main_loop_quit();
         break;
     }
}
```

On notera qu'ici, le langage JavaScript nous aide en permettant de directement faire un *switch* sur des chaînes de caractères, ce qui simplifie grandement le code. On précisera que **ecore_main_loop_quit** va demander à Ecore de quitter la boucle principale, mais ne quittera pas brutalement le programme. On va donc revenir à la fonction d'initialisation, juste après l'appel à **ecore_main_loop_begin**, et on en profitera pour tout désinitialiser proprement en détruisant les objets du canvas ainsi que la fenêtre et en « coupant » les EFL utilisées dans l'exemple.

```
evas_object_del(obj);
evas_object_del(background);

ecore_evas_free(ee);

ecore_evas_shutdown();
ecore_shutdown();
```

Voici donc un exemple simple mais peu animé (ou plutôt très statique !). Nous allons maintenant ajouter du tonus à ce texte en le faisant rebondir d'un bout à l'autre de l'écran. Pour cela, nous allons utiliser un *timer* un peu spécial : un *animator*. Utiliser un timer classique dans cet exemple ne comportant qu'une seule animation n'aurait pas vraiment montré de problème, mais lorsqu'on a beaucoup d'objets en mouvement, on veut que leurs déplacements soient synchronisés. Pour cela, il faut synchroniser les timers entre eux, ce qui n'est pas vraiment pratique. C'est ainsi que l'on fait appel à un type spécial de timers, qui vont toujours s'exécuter ensemble au même rythme, et ainsi, permettre à l'animation d'une frame d'être toujours complète. Commençons donc par l'initialiser juste après avoir créé le texte :

```
obj.dx = +20;
obj.dy = +10;
obj.x = 100;
obj.y = 100;
obj.w = 200;
obj.h = 50;
obj.constrain = { x: 0, y: 0, w: 720, h: 576 };
ecore_animator_add(anim_cb, obj);
```

La première chose que l'on remarque ici, c'est que nous définissons des propriétés supplémentaires à l'objet texte. Le problème que l'on tente de résoudre par cette démarche est que faire des allers-retours entre le monde en C et le JavaScript est coûteux. Pour éviter ceci, on utilise aussi longtemps que possible les informations connues par le JavaScript, puis on copie la dimension de l'objet texte et on lui ajoute également une propriété de vitesse, dx et dy. Avec cela, on est prêt à lancer l'animation, il ne reste plus qu'à coder **anim_cb**.

```
function anim_cb(obj)
{
   var x;
   var y;

   x = obj.x + obj.dx;
   y = obj.y + obj.dy;

   if (x + obj.w > obj.constrain.x + obj.constrain.w || x < obj.constrain.x)
     {
        obj.dx = - obj.dx;
        x += 2 * obj.dx;
     }

   if (y + obj.h > obj.constrain.y + obj.constrain.h || y < obj.constrain.y)
     {
        obj.dy = - obj.dy;
        y += 2 * obj.dy;
     }
   evas_object_move(obj, x, y);
   obj.x = x;
   obj.y = y;

   return true;
}
```

Le code de l'animation est plutôt simple : on déplace l'objet à la vitesse dx/dy en gérant juste un rebond/changement de direction lorsqu'on atteint un des bords de l'écran. Vous pouvez assez facilement ajouter d'autres objets et (par exemple) changer la couleur avec **evas_object_color_set**.

Avec cet exemple, vous avez vu comment fonctionne un canvas stateful et comment on peut réaliser des animations. C'est la base de toute interface, mais cela deviendra vite laborieux de devoir déclarer chaque objet à la main dans le JavaScript. Cela ne permet pas non plus de séparer le design du code et ralentit l'adaptation de l'interface aux plates-formes cibles. C'est pour cela que le projet Enlightenment a développé la technologie Edje qui est là pour adresser tous ces problèmes. Edje mériterait bien plus qu'un article, mais pour cette fois-ci, nous allons juste l'utiliser pour changer le rectangle du fond d'écran et en faire un *layout* simple. Avant d'utiliser Edje, il ne faut pas oublier de charger son binding à l'aide d'un **elx.load("edje")**. Ensuite, nous allons créer un fichier **elixir.edc** qui va contenir notre layout :

```
collections {
   group {
      name: "main";
      parts {
         part {
            name: "background";
            type: RECT;
            mouse_events: 0;

            description {
               state: "default" 0.0;
               rel1.relative: 0.0 0.0;
               rel2.relative: 1.0 1.0;
               color: 0 0 100 255;
            }
         }
      }
   }
}
```

Le principe d'un fichier Edje est de décrire différents groupes d'objets qui pourront s'instancier dans un unique objet Edje dans le JavaScript. Ici, on ne crée qu'un seul groupe, ne contenant qu'un seul objet, **background**, qui prend toute la surface de l'objet Edje et aura du bleu foncé comme couleur. Il est à noter qu'il est préférable d'exprimer les coordonnées d'un objet de manière relative dans un fichier Edje. Maintenant que nous avons notre fichier de descriptions, il faut le compiler dans sa forme finale grâce à la commande **edje_cc -v elixir.edc elixir.edj**. À partir de ce fichier Edje, nous pouvons instancier un objet Edje pour le fond d'écran en remplaçant les appels à **evas_object_rectangle_add** et **evas_object_color_set** par :

```
obj = edje_object_add(evas);
edje_object_file_set(obj, "elixir.edj", "main");
```

Et voilà un superbe fond bleu ! Nous pouvons un peu améliorer ceci en limitant la zone d'animation avec un deuxième rectangle, un peu à la manière d'une zone de jeu. Pour cela, ajoutons une deuxième **part** à notre fichier Edje :

```
part {
   name: "constrain";
   type: RECT;
   mouse_events: 0;

   description {
      state: "default" 0.0;
      rel1.relative: 0.1 0.1;
      rel2.relative: 0.7 0.9;
      color: 0 0 180 255;
   }
}
```

Il reste à prendre en compte les caractéristiques de cet objet dans notre JavaScript en changeant juste la ligne **`obj.constrain = { x: 0, y: 0, w: 720, h: 576 }`** par **`obj.constrain = edje_object_part_geometry_get(background, "constrain")`**. Après recompilation du fichier Edje, l'animation se fait maintenant dans un cadre plus limité. On peut facilement modifier les coordonnées de cette zone et voir les résultats du JavaScript varier. Mais il y a quand même un défaut : le point de départ du texte n'est pas forcément dans la zone d'animation, ce qui peut provoquer de petits inconvénients. Pour cela, une légère modification résoudra le problème :

```
obj.x = obj.constrain.x + 10;
obj.y = obj.constrain.y + 10;
evas_object_move(obj, obj.x, obj.y);
```

Il reste une dernière notion à aborder dans Edje : les animations. Dans les objets que nous avons décrits jusqu'à présent, nous n'avons spécifié qu'une description par *défaut,* qui est l'état initial d'une *part* lors du chargement d'un Edje. En en créant plusieurs, nous pouvons définir les bases d'une animation :

```
part {
   name: "animation";
   type: RECT;
   mouse_events: 0;

   description {
      state: "default" 0.0;
      rel1.relative: 0.75 0.1;
      rel2.relative: 0.85 0.2;
      color: 0 0 255 255;
   }
   description {
      state: "right" 0.0;
      rel1.relative: 0.85 0.1;
      rel2.relative: 0.95 0.2;
      color: 255 0 0 255;
   }
   description {
      state: "bottom" 0.0;
      rel1.relative: 0.75 0.8;
      rel2.relative: 0.85 0.9;
      color: 0 255 0 255;
   }
}
```

Il faut maintenant décrire la logique de l'animation grâce à une série de programmes qui seront déclenchés lors de l'affichage de l'objet après une période d'attente aléatoire :

```
programs {
   program {
      name: "beginning";
      source: "";
      signal: "show";
      in: 0.1 1.0;
      action: STATE_SET "right" 0.0;
      transition: LINEAR 0.5;
      target: "animation";
      after: "nextstep";
   }
   program {
      name: "nextstep";
      action: STATE_SET "bottom" 0.0;
      transition: SINUSOIDAL 0.8;
      target: "animation";
      after: "laststep";
   }
   program {
      name: "laststep";
      action: STATE_SET "default" 0.0;
      transition: ACCELERATE 0.8;
      target: "animation";
      after: "beginning";
   }
}
```

Le premier programme (**beginning**) démarrera lorsqu'il recevra le signal **show** après un temps d'attente entre 0,1 et 1,1 seconde. **show** est un signal envoyé automatiquement par Edje dès l'affichage de la première frame, mais il est possible d'envoyer des signaux depuis le JavaScript avec **edje_object_signal_emit**, ce qui permet d'intégrer complètement l'Edje au JavaScript. Nous allons donc ajouter une **part** dont l'animation sera contrôlée directement par le JavaScript et qui changera d'état entre visible et invisible en fonction de l'appui sur une touche du clavier.

```
   program {
      source: "js";
      signal: "toggle";
      filter: "feedback" "default";
      action: STATE_SET "transparent" 0.0;
      transition: LINEAR 0.5;
      target: "feedback";
      after: "back";
   }
   program {
      source: "js";
      signal: "toggle";
      filter: "feedback" "transparent";
      action: STATE_SET "default" 0.0;
      transition: LINEAR 0.5;
      target: "feedback";
      after: "back";
   }
   program {
      name: "back";
      action: SIGNAL_EMIT "toggle" "end";
   }
```

On utilise ici deux nouvelles possibilités de Edje. Tout d'abord, chaque programme interceptant les signaux ne sera déclenché que si la **part** *feedback* est dans l'état spécifié dans le second paramètre. Cela évite de devoir coder tous les états possibles dans le JavaScript et permet d'ajouter facilement de nouveaux états intermédiaires. Enfin, pour permettre des animations de temps variable, il faut notifier au JavaScript quand elles se terminent. C'est à cela que sert le **SIGNAL_EMIT**. Les signaux envoyés par Edje peuvent être captés de manière identique dans un programme et en JavaScript. Pour cela, il suffit d'enregistrer une *callback* pour un signal particulier à la création de l'objet Edje : **edje_object_signal_callback_add(obj, "toggle", "end", toggle_end_cb, null)** dans notre exemple. La fonction gérant le signal va être très simple dans notre cas :

```
function toggle_end_cb(data, obj, emission, source)
{
   obj.toggling = false;
}
```

Il manque juste un petit morceau de code pour envoyer le signal depuis le JavaScript dans le *handler* de clavier :

```
      case "x":
         if (!obj.toggling)
           {
              edje_object_signal_emit(obj, "toggle", "js");
              obj.toggling = true;
           }
         break;
```

Et voilà, ça fonctionne ! Rien de bien compliqué pour un exemple qui commence à bouger. Il y a encore beaucoup d'autres fonctionnalités à voir dans Edje ; on peut toutes les retrouver dans la documentation à l'adresse http://docs.enlightenment.org/auto/edje/edcref.html. Notre première application est presque finie, il ne reste plus qu'à en faire un seul fichier pour être déployé plus facilement. Pour cela, nous nous appuyons sur le format de fichier de Edje qui utilise Eet et qui peut intégrer plusieurs sections différentes et indépendantes. Vous pouvez voir les sections composant un fichier Edje de la manière suivante :

```
$ eet -l elixir.edj
edje_source_fontmap
edje_sources
edje/file
edje/collections/0
$
```

Nous allons donc simplement insérer le fichier JavaScript dans une section utilisée par Elixir à l'aide de la commande **eet -i elixir.edj elixir/main elixir.js**. Il suffit maintenant d'exécuter **elixir elixir.edj** ou de déposer le fichier sur le disque dur de la Freebox via son serveur FTP, ou encore de l'envoyer via http://factory.free.fr sur le Freestore. Bien entendu, tout ce code d'exemple et bien plus sont disponibles à l'adresse : http://code.google.com/p/freebox-elixir/source/browse/#svn/trunk/exemples/articles/introduction. ■

Auteur : Cédric Bail

Utilisateur GNU/Linux depuis 1997. Mainteneur d'Elixir, Eina et Eet pour une petite société française de set top box.

Auteur : Pierre-François Hugues

Pousseur de wagonnets au fond de la mine.