Développement pour iPod Touch sous GNU/Linux : application à la communication par liaison Bluetooth

G. Goavec-Mérou, J.-M Friedt

6 novembre 2010

Les iPod Touch et iPhone sont probablement les systèmes embarqués les plus largement disponibles auprès du grand public. Bien que le qualificatif d'"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 de 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 plateforme. 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 sont implémentés. Le système d'exploitation exécuté sur cette plateforme – 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 plateformes 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 plateforme 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 plateforme 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 plateforme 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 que 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*).

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 de Apple, et que la gamme des applications accessibles est celle mise à disposition après censure par les gestionnaires de iTunes. Ainsi, une vaste gamme d'applications, et notamment un serveur ssh 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. Noter 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 commande 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 plateforme 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 de 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

[1] http://llvm.info
[2] http ://www.saurik.com/id/4
[3] developer.apple.com/technologies/xcode.html
[4] http://developer.apple.com

`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 la champ *VFDecrypt*. Nous renseignons la variable `DECRYPTION_KEY_SYSTEM` dans le script `toolchain.sh` avec cette chaîne de caractères.
5. une fois tous ces fichier 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 entêtes 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'entêtes `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 plateforme 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 Fig. 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 plateforme 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.

FIG. 1 – Exécution d'une application POSIX dans le `MobileTerminal` de l'iPod Touch, application compilée au moyen de la toolchain mise en place sur plateforme x86.

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 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 résultant :

- dans le fichier `config.h`, désactiver les définitions de `rpl_malloc` et `rpl_realloc`
- dans le fichier d'entête `src/include/ngspice.h`, définir la fonction de vérification si un résultat est infini : `#define finite isnormal` qui n'est sinon 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 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 néanmois 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.

FIG. 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 (droite) au moyen de `asciiplot` est exploitable.

Quelques subilités 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 [3] sur ce support pour identifier les objets mobiles visibles dans le ciel après le coucher du soleil – qui nécessite quelques modifications pour être exploitable :

1. modifier le compilateur `CC=arm-apple-darwin9-gcc` dans le `Makefile`, avec la configuration FreeBSD, pour compiler sur architecture ARM,
2. corriger le bug de l'an 2000 en modifiant la ligne 272 de `sattime.h` (remplacer 100 par 200 dans la comparaison),
3. 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`.

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 plateformes 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 [4, 5].

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 commande ne répond pas nécessairement aux besoins de l'utilisateur, et ce langage ne permet pas d'exploiter les bibliothèques de développements d'interfaces graphiques.

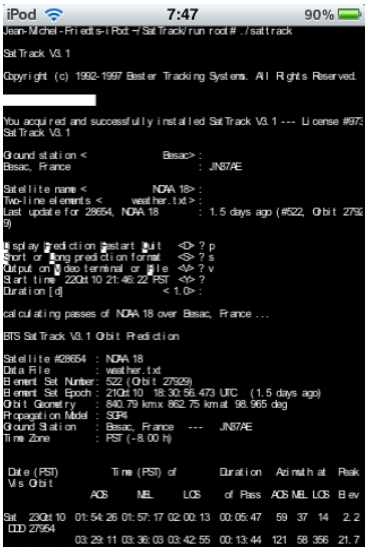

FIG. 3 – Sattrack – un logiciel de prévision de passage de satellites opensource jusqu'à sa version 3.1.6 malgré une license très restrictive – fonctionne en mode texte (`MobileTerminal`) sur iPod avec la gestion de la position du curseur au moyen des caractères de contrôle compatibles avec un terminal 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.

En effet, les interfaces graphiques pour plateformes 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 avant dans cet apprentissage [6, 7].

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 reférence dessus, ou lorsque l'objet n'est plus nécessaire dans l'application).

Comme pour le C++ ce langage utilise un fichier d'entête ainsi qu'un fichier d'implémentation.

5.1.1 Le fichier d'entête

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.

Le listing 4 présente un entête 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 définie un membre pour cette classe. Comme dans la plupart des languages orientés objets, chaque variable a une certaine portée, permettant ainsi d'autoriser ou d'interdire leur

[5] http://en.wikipedia.org/wiki/Objective-C

```
1#import <objc/objc.h>
 #import <objc/Object.h>
3
 @interface my_class : Object
5{
   int nb;
7}
 - (void)sayHello;
9@property (nonatomic,assign) int nb;
 @end
```

FIG. 4 – Exemple d'un fichier d'entête

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 :

```
1  -/+(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'accés 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 *setters* 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 fournit 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 accés conccurent à la méthode et la valeur passée à la méthode sera copiée dans nb. Il existe d'autres options telles que *retain* pour l'accés 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`.

Ce fichier 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.

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

FIG. 5 – 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.6, 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

Le listing 6 présente une fonction principale (`main`) classique qui instancie un objet de type `my_class` (défini dans le code Fig. 5).

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 fournies chacun des suivants se fera en donnant le nom du paramètre suivi d'un " :" puis de la variable ou l'objet.

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

5int main(void)
 {
7  my_class *mc = [[my_class alloc] init];
   [mc setNb:1];
9  [mc sayHello];
   mc.nb=2;
11 [mc sayHello];
   return 0;
13}
```

FIG. 6 – Instanciation et utilisation d'une classe : fichier `main.m`

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 plateforme 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 :

```
1gwe@linux objc_example $ ./objc_example
 Hello, 1!
3Hello, 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 modifier 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 interoger l'instance afin de vérifier qu'il n'y a pas de message en attente.

Le développement sur plateformes 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 donnée, par exemple. Le second sert pour gérer certains comportements spécifiques qui ne sont pas codés en dur dans la classe par soucis 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 code source 2 présente la création d'un champ de texte contenant le message classique du premier exemple, afin de générer l'application illustrée sur la Fig. 7.

Le point d'entrée d'une application graphique sur iPod est exactement la même que pour une application à savoir une fonction main (listing 1) :

```
1 int main(int argc, char **argv) {
      NSAutoreleasePool *autoreleasePool = [
3         [ NSAutoreleasePool alloc ] init
      ];
5
      int returnCode = UIApplicationMain(argc, argv, @"HelloCocoa", @"HelloCocoa");
7     [ autoreleasePool release ];
      return returnCode;
9 }
```

Tab. 1 – Chargement d'une application graphique : source `HelloCocoa_main.m`.

- les lignes de 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 fournie à *UiApplicationMain*.

```
  @implementation HelloCocoa
2
 - (void)applicationDidFinishLaunching: (UIApplication *) application {
4     window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] →
          ↪autorelease];

6     CGRect windowRect = [ [ UIScreen mainScreen ] applicationFrame ];
      windowRect.origin.x = windowRect.origin.y = 0.0f;
8
      window.backgroundColor = [UIColor whiteColor];
10
      CGRect txtFrame = CGRectMake(50,150,150,150);
12    UITextView *txtView = [[UITextView alloc] initWithFrame:txtFrame];
      txtView.text = @"Hello Cocoa";
14    UIFont *font = [UIFont boldSystemFontOfSize:18.0];
      txtView.font = font;
16
      [ window  addSubview: txtView ];
18    [txtView release];
      [window makeKeyAndVisible];
20 }

22 @end
```

Tab. 2 – 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 (listing 2) :

- à la ligne 4 est créée la fenêtre : bien que l'iPod ne soit pas multi-fenêtre, 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 3).

```
1 [...]
 LDFLAGS= -lobjc -bind_at_load -framework Foundation \
3     -framework CoreFoundation -framework UIKit
  [...]
5 bundle: HelloCocoa
    @mkdir -p HelloCocoa.app
7   @cp HelloCocoa HelloCocoa.app/HelloCocoa
    @cp Info.plist HelloCocoa.app
9 install: bundle
    @ssh root@$(IP) "cd /Applications/HelloCocoa.app && rm -R * || echo 'not found'→
        ↪ "
11  @scp -rp HelloCocoa.app root@$(IP):/Applications
    @ssh root@$(IP) "cd /Applications/HelloCocoa.app ; ldid -S HelloCocoa_; killall→
        ↪ SpringBoard"
```

Tab. 3 – 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 défini 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 7.

6 Liaison Bluetooth

Une pile libre de liaison Bluetooth [8] a été développée, principalement par M. Ringwald, et son code source mis à disposition à http://code.google.com/p/btstack (Fig. 8). 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.

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

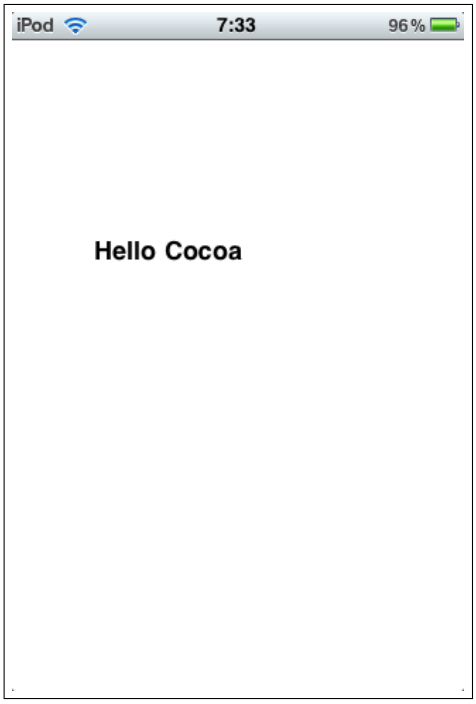

FIG. 7 – Exemple d'interface graphique contenant un unique champ de texte.

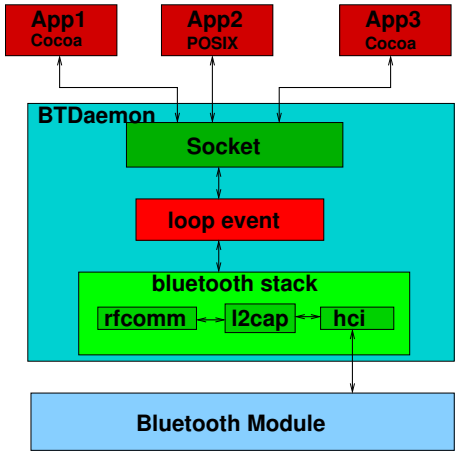

FIG. 8 – Structure de `BTstack`, fournissant BTDaemon chargé de l'interface entre la matériel (géré par le système d'exploitation d'Apple) et les applications utilisateur, ainsi que les bibliothèques de gestion des trames Bluetooth – nommées BTStackManager – pour COCOA.

- 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 occurences de `sdk` en miniscules 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 identifiant 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. 9).

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,

[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

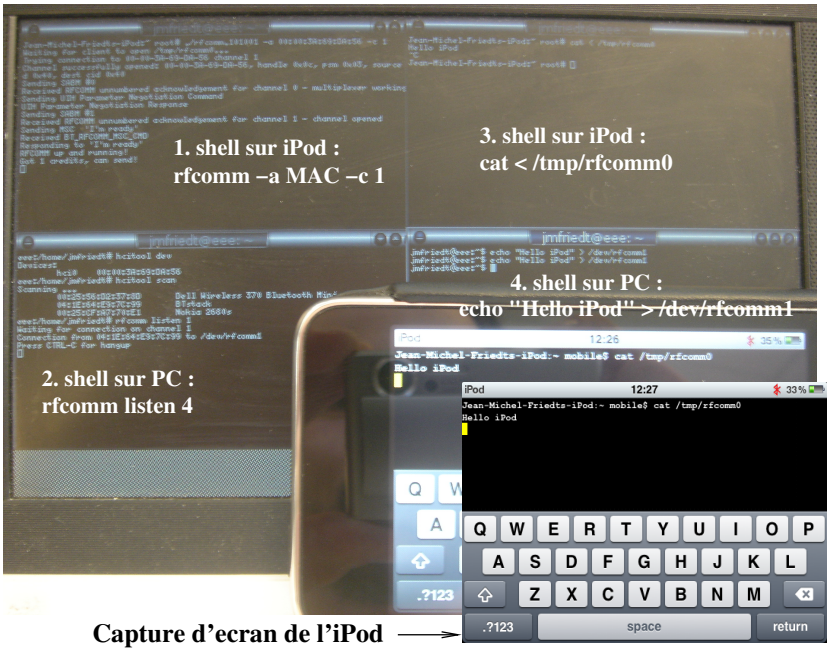

FIG. 9 – Transmission de données depuis un shell sur PC, *via* un convertisseur USB-Bluetooth, vers iPod, au moyen des outils `rfcomm` exécutés sur les deux interlocuteurs.

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 évènements 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 Debuggage 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 fig.10.

```
1> ACL data: handle 12 flags 0x02 dlen 17
 > ACL data: handle 12 flags 0x01 dlen 1
3  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
5     dlci 2 frame_type 0 credit_flow 14 pri 0 ack_timer 0
      frame_size 100 max_retrans 0 credits 0
```

FIG. 10 – 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 entête (4 octets), contenue 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'entête 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'était pas encore implémentée 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 Figs. 11 et 12. Deux points méritent notre attention pour atteindre ce résultats :

- 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.

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. 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 impossible 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.
- d'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.

FIG. 11 – 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.

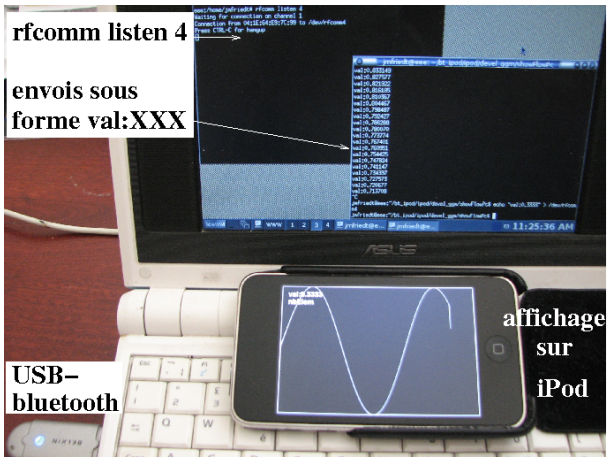

FIG. 12 – Transmission de données depuis un PC, *via* un convertisseur USB-Bluetooth, au moyen d'un programme dédié à l'affichage graphique sur iPod avec émission de valeurs fournies depuis le shell sur PC. Comme nous l'avons vu auparavant, la liaison est établie sur le PC par `rfcomm listen` et les données transmises selon un format `val :`valeur au travers de `/dev/rfcomm1`.

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 est qu'au niveau de la déclaration de l'interface elle s'annonce comme implémentant le ou les protocoles :

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

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

A l'heure actuelle, la classe *BTstackManager* fournie dans le dépot du projet ne dispose pas de l'implémentation compléte 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 fait 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.

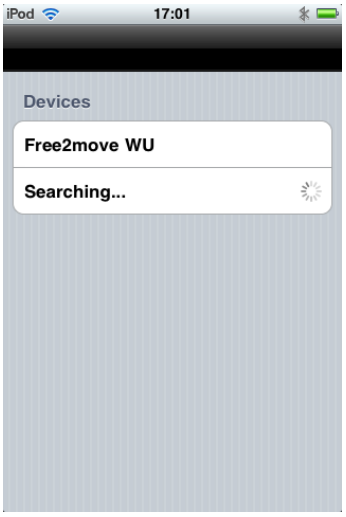

FIG. 13 – Fenêtre de choix d'un périphérique

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*.

[9] http://www.trabucayre.com/lm/btstackmanager_rfcomm.patch

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

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

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

- la première ligne créer une nouvelle instance de cette classe,
- dans le cas présent la classe courante va recevoir les informations tel 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*.

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

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

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

```
-(BOOL) discoveryView:(BTDiscoveryViewController*)discoveryView
2          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 :

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

le périphérique choisie 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 supprimés.

Il existe une fonction nommée

```
rfcommConnectionCreatedAtAddress
```

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 :

```
1-(void) rfcommDataReceivedForConnectionID:(uint16_t)connectionID withData:(→
    ↪uint8_t *)packet ofLen:(uint16_t)size{
  memcpy(tmp+nb,packet,size);
3  nb+=size;
   if ((packet[size-1] == '\0')||(packet[size-1] == '\n')){
5    NSString *str2 = [[NSString alloc] initWithCString:(char*)tmp+4 encoding : →
        ↪1];
     CGFloat val = [str2 doubleValue];
7    [affSin addPoint: val];
     nb = 0;
9  }
}
```

Au même titre qu'un packet *L2CAP* peut être fragmenté, il est possible qu'une trame *RFCOMM* puisse parvenir à l'application en plusieurs morceaux. C'est pourquoi les caractères reçut sont concaténés dans un buffer (l.2-3) tant que le caractère fin de fin de chaîne ou un 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 convertie 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.

A 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 put 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];
}
```

TAB. 4 – Fonction en charge de la réception et du stockage de nouvelles données

Cette fonction (listing 4) 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 point, si le nombre de points déjà reçu dépasse un seuil (l.3), le premier point est supprimé (l.4), dans le cas contraire le nombre de points reçut 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 5) :

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

7   CGFloat orig=imgRect.size.width/2;
    CGFloat x=imgRect.size.width/2;
9   CGFloat decalY = imgRect.size.height/(nbPoints-10);

11  CGContextSetRGBFillColor (context, 0, 0, 0, 1);
    CGContextSetRGBStrokeColor(context,1,1,1,1);
13
    CGContextFillRect (context, imgRect);
15  UIRectFrame(imgRect);
    int i;
17  CGContextBeginPath (context);
    NSNumber *nb, *nb1;
19
    for (i=1;i<arraySize;i++) {
21    nb = [arrayPoints objectAtIndex:i];
      nb1 = [arrayPoints objectAtIndex:i-1];
23      CGContextMoveToPoint(context, x+[nb1 floatValue]*orig, (i-1)*decalY);
        CGContextAddLineToPoint(context, x+[nb floatValue]*orig,i*decalY);
25  }
    CGContextStrokePath(context);
27}
```

TAB. 5 – 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 fournit 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 issus 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 context 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 12.

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 celà 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 [9]. 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 I^2C qui nécessite une initialisation et un protocole de communication dédié [10]. Dans un premier temps, la brique NXT est informée de la nature du capteur connectée 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
```

[10] http://nxt.ivorycity.com/
[11] http://www.mindstorms.rwth-aachen.de/documents/downloads/doc/troubleshooting.html

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.

FIG. 14 – 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 plateforme : 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 cette objet (l.4) afin de recevoir les données, qui seront fourni avec un interval configuré en l.6.

Ainsi tous 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. 14).

7 Conclusion

Nous avons démontré la capacité à développer sur iPod Touch en exploitant exclusivement des outils opensource 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 fichier, s'obtient au moyen d'une liaison `ssh` qui permet notamment de transférer nos propres exécutables sur cette plateforme. 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.

Références

[1] http://www.apple.com/ipodtouch/specs.html pour une version incomplète des spécifications techniques, wikipedia 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] S. Guinot, J.-M Friedt, *La réception d'images météorologiques issues de satellites : utilisation d'un système embarqué*, GNU/Linux Magazine France Hors Série **24** (2006)

[4] Y. Le Guen, *Programmation Objet : Objective-C*, GNU/Linux Magazine France **9** (2000), disponible à http://chl.be/glmf/www.linuxmag-france.org/old/lm9/objectiveC.html

[5] Y. Le Guen, *Programmation Objet : Objective-C, 2ème partie*, GNU/Linux Magazine France **10** (2000), disponible à http://chl.be/glmf/www.linuxmag-france.org/old/lm10/objectiveC.html

[6] A.M. Duncan, *Objective-C pocket reference*, O'Reilly Media (2002)

[7] un point de départ pour appréhender le langage à http://www.siteduzero.com/tutoriel-3-5208-la-programmation-mac.html

[8] X. Garreau, *Bluetooth, installation et utilisation*, GNU/Linux Magazine France **78** (2005), disponible à http://www.unixgarden.com/index.php/comprendre/bluetooth-installation-et-utilisation

[9] *LEGO Mindstorms NXT Direct Commands* v1.00 (2006), disponible à http://www.microframeworkprojects.com/images/d/df/LEGO_MINDSTORMS_NXT_Direct_commands.pdf

[10] *LEGO Mindstorms NXT Ultrasonic Sensor I^2C 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

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.

Jean-Michel Friedt est ingénieur dans une société privée, hébergé 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.