# PROJET DE MASTER, SEMESTRE HIVER 2007–2008

# PÉRIPHÉRIQUES BLUETOOTH : CONNECTIQUE PC ET ROBOTIQUE

Christophe Richon, MA–IN 3

BIRG Laboratory

École Polytechnique Fédérale de Lausanne

Supervision : Professeur Eduardo Sanchez et assistant Pierre–André Mudry

18 Janvier 2008

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

“Et maintenant, tu as les dents bleues”
La chanson du dimanche, épisode 2 saison 1

Photo : Pierre runnique de Harald Bluetooth, érigée en 965 à Jelling, capitale du Danemark. Crédits : anonyme.

# TABLES DES MATIÈRES

# INTRODUCTION

Les consoles de jeux ont longtemps été reléguées au rang de simples jouets. En effet, lors de l'apparition des premières consoles populaires en 1983 (comme la NES de Nintendo ou la Master System de Sega), les techniques informatiques et électroniques utilisées n'étaient pas autant accessibles qu'aujourd'hui. Les composantes électroniques étaient spécifiques à ces consoles, le format des jeux (cartouche) unique et les périphériques de contrôle étaient très différents des joysticks ou des claviers utilisés par l'informatique.

La tendance s'est aujourd'hui radicalement inversée : les consoles sont des ordinateurs spécialisés dans le domaine du jeu ce qui rapproche le domaine des consoles de celui de l'informatique. De plus, la technologie des consoles n'est plus une petite partie de l'investissement des sociétés fabriquant les composantes (Intel, IBM, Sony). À titre d'exemple, le Cell[1], processeur conçu par IBM, Sony et Toshiba est l'un des plus puissants processeurs (jusqu'à 205 GFLOPS en 32 bits) à disposition du particulier. L'époque où la puissance des ordinateurs était déterminée par le dernier Pentium de chez Intel et de sa puissance en GHz est dépassée : les ordinateurs ont maintenant toute la puissance nécessaire à une utilisation quotidienne. Intel cherche donc à plus se diversifier[2] et le temps du processeur central à utilité générale est révolu[3] : l'industrie tend aujourd'hui vers la conception de processeurs spécialisés, mathématique, physique ou encore graphique. L'industrie des consoles de jeux est donc actuellement un des principaux moteurs d'évolution et de recherche de l'informatique.

À titre d'exemple, le laboratoire LACAL[4] de l'École Polytechnique Fédérale de Lausanne[5] a récemment fait l'acquisition de 220 consoles Playstation 3 pour concevoir une ferme de calcul. Elle est utilisée pour des projets de cryptographie[6] (*Cryptanalytic application of PS3 cluster*), de l'algorithmique (*Develop and compare fast long integer arithmetic packages for Sony Playstation 3*) ou encore des calculs de prédictions (*Predicting the winner of the 2008 US Presidential Elections using a Sony Playstation 3*).

De notre côté, nous allons développer les outils nécessaires afin de pouvoir utiliser le matériel du domaine des consoles de jeu dans les domaines de l'informatique et de la robotique. Dans le cadre de ce projet, nous étudierons les périphériques de contrôle des consoles actuelles afin de pouvoir les utiliser dans différents contextes comme la robotique, l'intégration de détecteurs de mouvement ou encore la technologie sans fil Bluetooth. Ces divers éléments font ainsi des manettes de jeu de dernière génération de véritables concentrés de technologie que nous nous proposons d'étudier. Avant cela,

---

1 Kevin Krewell, "Cell Moves Into the Limelight – ISSCC Begins the Rollout of Cell Architecture", Microprocessor Report, February 2005

2 McGregor J., "Let the Processor Wars Begin, Intel Takes Aim with New Products and a New Strategy" The Insider's Guide to Microprocessor Hardware, July 2007

3 The MPR Staff, " Trends in General–Purpose Processors, Microprocessor Forum Panel Speculates on Future Direction" The Insider's Guide to Microprocessor Hardware, November 2007

4 Laboratory for cryptologic algorithms : lacal.epfl.ch

5 www.epfl.ch

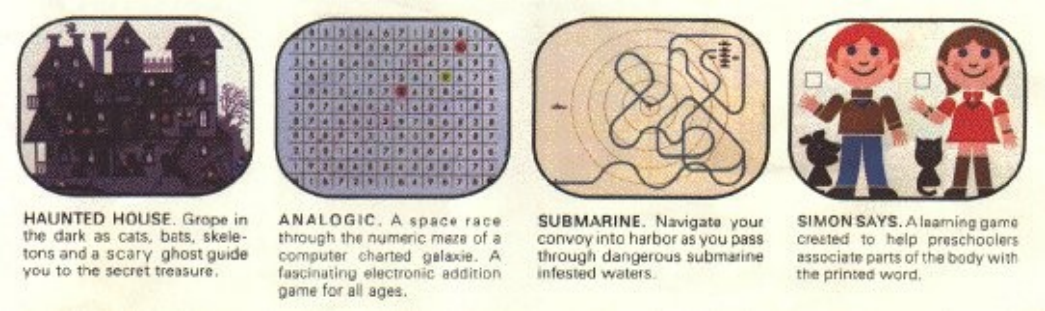
6 Projets dont la liste et les détails se trouvent à cette url : http://people.epfl.ch/arjen.lenstra

afin de comprendre comment le domaine des consoles en est arrivé à cette convergence vers le domaine informatique, nous allons réviser l'historique des consoles pour mieux introduire le propos.

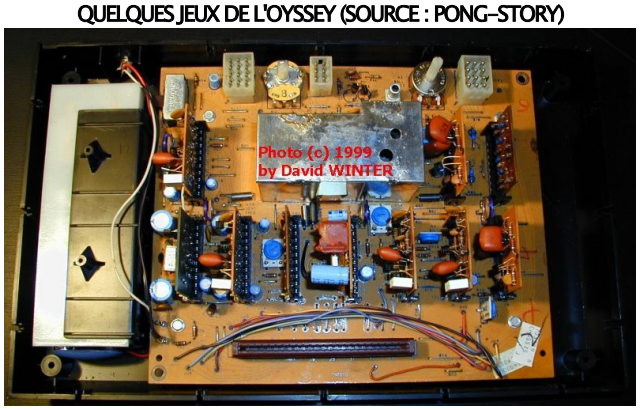
## Historique

Afin de comprendre au mieux leur évolution, il faut avoir à l'esprit les diverses générations de consoles qui ont profité, tout comme les ordinateurs, des progrès technologiques. Ces étapes sont beaucoup plus évidentes dans le domaine des consoles, car une nouvelle console apparaît généralement tous les 1–2 ans, alors que l'évolution du domaine informatique est constante. Sept générations se sont succédé de 1972 à aujourd'hui.

Ainsi, l'une des toutes premières consoles était l'Odyssey[7], qui ne contenait ni mémoire, ni processeur: les jeux étaient en fait des circuits électroniques câblés. La totalité du système et les jeux fonctionnaient avec des circuits logiques et la toute première version ne supportait pas le son. Le jeu le plus célèbre de cette console était Pong, jeu simulant un match de tennis où on dirige une barre verticale de chaque côté de l'écran.




QUELQUES JEUX DE L'OYSSEY (SOURCE : PONG-STORY)




LA CARTE-MÈRE (SOURCE : PONG-STORY)

La caractéristique technique faisant le pont entre la 1re et la 2e génération est l'apparition des processeurs et des cartouches à mémoire : les jeux sont désormais stockés en ROM sous forme binaire et non plus sous la forme de circuits électroniques dont la capacité variait entre 2 et 32 KO. La mémoire vive a également fait son apparition, mais vu son prix elle ne dépassait pas 1KO. Comme console de cette époque, se déroulant entre 1978 et 1982, nous pouvons citer l'Intellivision (1980), la

[7] http://www.pong-story.com/odyssey.htm

Vectrex (pouvant générer des dessins vectoriels) et l'Atari 2600. Durant cette même période, les développeurs se multiplient, les jeux envahissent le marché et le public se trouve saturé de jeux vidéos: s'ensuivra alors un grand crash du marché des jeux vidéos[8]. Il faut savoir qu'aux USA, les jeux invendus en magasin doivent être remboursés par le fournisseur ou remplacés par d'autres jeux. Vu la quantité massive de jeux édités durant cette période, les invendus s'accumulent. De plus, les premiers ordinateurs personnels font leur apparition et concurrencent les consoles de salon. Résultat, beaucoup de maisons d'édition doivent stopper leur production de jeux vidéos.

En 1983, le Japon dépasse les US dans le domaine de la production des jeux vidéos et Nintendo sort sa première console, souvent considérée comme étant la première console de jeux vidéos : la NES, ou Famicom en Japonais. C'est la 3e génération de consoles de jeux, dont les processeurs avaient en principe une architecture 8 bits. Malgré le format propriétaire des cartouches utilisées, il existait déjà des outils permettant de détourner l'utilisation première des jeux : des adaptateurs matériels autorisaient les utilisateurs à altérer le flux de données du jeu afin de tricher[9].


UNE CARTOUCHE ACTION REPLAY POUR TRICHER SUR AMIGA(SOURCE : WIKIMEDIA COMMONS)

La 4e génération des consoles dès 1987 (composée de la SNES de Nintendo, la NEC PC ou la Sega MegaDrive, toutes consoles 16 bits) voit apparaître les premiers périphériques ajoutés aux consoles, comme les lecteurs CD. Cependant, ces médias ne sont encore que faiblement diffusés et il faut attendre le début de la distribution massive des ordinateurs personnels (de type x86) pour voir ce type de périphérique ajouté à la majorité des consoles de la génération. Les consoles 32 et 64 bits (Ultra64 de Nintendo ou la Saturn) de 5e génération apparaissent durant cette période, lors de la diffusion massive des premiers médias optiques. Elles ont naturellement profité de cette technologie, à commencer par Sony et sa Playstation sortie vers 1995 qui comporte un lecteur CD optique normal double-vitesse en standard qui permette aussi de lire des CD audios.

[8] R. DeMaria & J. Wilson., *High Score!: The Illustrated History of Electronic Games* (2e édition), McGraw-Hill Osborne, New York, 2005.

[9] Par exemple le « Pro Action Replay », de chez Datel (http://www.datel.co.uk/).


SEGAMEGADRIVE (SOURCE : WIKIMEDIA COMMONS)

Cette introduction de médias de jeux communs avec le monde des ordinateurs a été un premier pas vers une convergence de ces deux milieux et le début des tentatives d'analyse et d'exécution de code "personnel" sur ces consoles. En effet, vu que les CD étaient lisibles et gravables par un ordinateur, il n'y avait qu'un pas pour faire tourner ses propres CD sur la console. Des émulateurs utilisant directement le support de données CD commencent à se développer, rapprochant un peu plus le monde des consoles de celui des ordinateurs.


LA NINTENDO GAMECUBE, VERSION PLATINIUM (SOURCE : WIKIMEDIA COMMONS)

Suivent les consoles de la 6e génération (128 bits en général) dont les membres sont la PlayStation 2, la XBOX de Microsoft, la Gamecube de Nintendo et bien entendu la Dreamcast de Sega. Ces consoles de 6e génération ont toutes une particularité. Tout d'abord, la Xbox de Microsoft (2002) est le premier ordinateur camouflé sous une coque plastique de console. En effet, elle contient un Intel Pentium III 733 MHz, un processeur largement diffusé dans les ordinateurs personnels. Elle a également une carte graphique de chez nVidia et une carte son également fabriquée par nVidia. De plus, c'est la première console à fournir par défaut un disque dur intégré et les ports destinés aux manettes sont des ports USB déformés supportant néanmoins des clés USB ou des claviers. Ensuite, la PlayStation 2, dotée d'un lecteur DVD, introduit dans les salons des lecteurs DVD vidéos avant la large diffusion des premiers lecteurs DVD vidéo. Sony mise ainsi sur une console technologiquement comparable à ses concurrentes, mais en plus qui, à long terme, permet de remplacer les futurs lecteurs DVD de salon. À noter également que des systèmes Linux peuvent être exécutés sur ces machines. Avant la sortie de

ces deux consoles, une console trop peu connue était vendue : la Dreamcast de Sega. Cette console innovait sur de nombreux points : première console avec manette à gâchettes analogiques, lecteur GD rom propriétaire, unité mémoire servant également de mini console de jeux dans la manette et également première console dotée d'un modem (56 ou 33.6k selon les régions) permettant de surfer à l'aide de clavier/souris branchés sur la console. En bref, une excellente console dotée d'une ludothèque de qualité, enterrée par le succès médiatique de sa concurrente de chez Sony, et qui marqua la fin du développement de console vidéo par Sega.

LA DREAMCAST DE SEGA ET SA MANETTE/CONSOLE (SOURCE : WIKIMEDIA COMMONS)

Récemment sont apparues les consoles de 7e génération : la Xbox 360, la Nintendo Wii et la Sony Playstation 3. Les consoles se sont dotées de disques durs, de lecteur de cartes standards, de ports USB, de connexion internet WiFi… bref, la différence avec les ordinateurs n'est plus que visuelle et nominative. De plus, les consoles ont monté en puissance de manière fulgurante, soutenues par l'industrie informatique. La Playstation 3 est dotée du Cell qui introduit le concept de parallélisation dans le monde des consoles avec ses 8 *cores* DSP spécifiques et la Xbox 360 d'un triple *core* à deux voies, transformant ces consoles en calculateur massif parallélisé.

Les contrôleurs de jeux ont également subi des modifications innovantes, particulièrement pour la Wiimote, la manette de la Wii : Wireless (Bluetooth), elle est également dotée d'un accéléromètre, d'un haut–parleur et d'un capteur CMOS afin de pouvoir directement pointer l'écran, qu'il soit cathodique, LCD ou rétroprojecteur.

Les possibilités offertes par les contrôleurs de jeux de la dernière génération sont innombrables et, étant donné la connectique Bluetooth utilisée, la conception d'un adaptateur logiciel ou matériel pour les utiliser sur un ordinateur, ou du matériel ad hoc, devient tout à fait envisageable. Ainsi, s'il y a possibilité de les rendre compatibles avec un ordinateur ou un module Bluetooth hardware embarqué, on aurait à disposition un périphérique de contrôle de haute technologie à faible coût ce qui sera là le but de notre projet.

# BUTS DU PROJET

Notre étude se concentrera sur les périphériques Sixaxix (Playstation 3) et Wiimote (Wii), dans le but d'adapter leur connectique à un usage informatique. Notre étude préliminaire a montré que la Wiimote était conçue de manière standard et connectable sur la majorité des *stack*s Bluetooth (Windows et Linux) du marché, ce qui rend une étude approfondie redondante. Il reste donc la tâche d'étudier le comportement de la Sixaxis. Celle-ci ne permet au premier abord qu'un branchement par USB et limité au niveau des fonctionnalités. L'étude de son protocole de connexion Bluetooth et de fonctionnement permettra de concevoir des interfaces pour la brancher sur une machine Win32 (Windows XP), Unix (Linux) et également sur du hardware embarqué. Nous possédons au laboratoire du Birg[10] un module robotique nommé YAMOR[11] (en photo ci-dessous), qui est constitué d'une interface Bluetooth, d'une FPGA et d'un moteur, ce qui lui permet de se mouvoir de manière autonome lorsqu'il est connecté à d'autres modules semblables. Ce YAMOR nous servira d'interface Bluetooth hardware pour mettre au point un protocole universel de connexion pour les modules Bluetooth de développement.
Au final, étant donné l'ampleur des domaines touchés, nous aurons à disposition du matériel ad hoc et les interfaces logicielles nécessaires à sa connexion sur n'importe quelle plateforme, software ou hardware.

[10] Biologically inspired robotics group : http://birg.epfl.ch
[11] http://birg.epfl.ch/page53469.html

# PÉRIPHÉRIQUES DE JEUX

## Wiimote

### Introduction

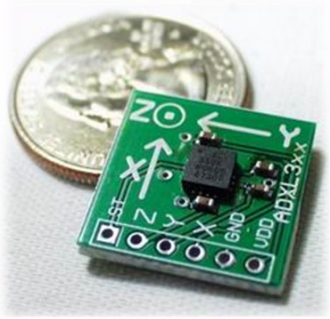

IMAGES PUBLICITAIRES DES FONCTIONNALITÉS DE LA WIIMOTE (SOURCE : HTTP://WII.NINTENDO.COM/)

La Wiimote est un périphérique de jeu au design très particulier. Au lieu d'être tenue à deux mains comme toute manette de jeu normale, celle-ci est tenue à une main. La fiche technique nous annonce un détecteur de mouvement, un système de pointage à l'écran (que nous allons étudier plus loin), un haut-parleur et un moteur désaxé pour obtenir un retour de force. En ce qui concerne les contrôles, cette manette dispose de 7 boutons, dont une gâchette et une croix directionnelle. Le design est conçu de telle sorte que l'on puisse tenir la Wiimote verticalement (comme sur les photos), pour avoir un accès facile à la croix et au bouton sous le pouce et à la gâchette sous l'index. Une autre manière de la tenir s'exécute avec deux mains, de manière horizontale pour avoir la croix directionnelle sous le pouce gauche et les deux boutons "1" et "2" disponibles pour la main droite, comme présentée ci-dessous avec un renfort :

GRIP HORIZONTAL POUR LA WIIMOTE (SOURCE : HTTP://WWW.WIISWORLD.COM)

### Caractéristiques techniques

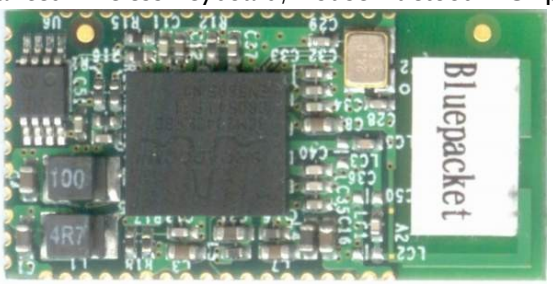
La détection des mouvements est réalisée avec un accéléromètre :




(SOURCE DE LA PHOTO : HTTP://WWW.SPARFUNK.COM)

Il s'agit d'un ADXL330 d'Analog Device[12]. Il a une consommation de 320µA, pour 3V et un champ de détection de –3G à 3G sur les 3 axes. Son prix à la pièce est d'environ 11 dollars[13] pour une taille de 0.7"x0.7".

Ensuite vient le module Bluetooth, qui est un module standard propre aux souris et aux claviers Bluetooth, le BCM2042 Advanced Wireless Keyboard/Mouse Bluetooth® Chip de Broadcom[14] :




(SOURCE DE LA PHOTO : WWW.ALIBABA.COM)

Ce chip est fourni avec un profil HID et une *stack* Bluetooth complète. Pour plus de détails sur la technologie Bluetooth, veuillez vous référer au chapitre Bluetooth. Ce contrôleur Bluetooth contient un processeur 8051 intégré et supporte la version 2 du protocole Bluetooth.

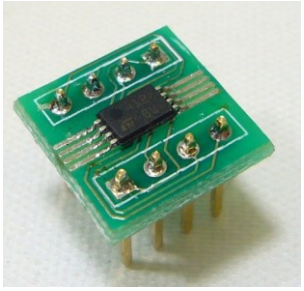
Ensuite nous avons un contrôleur audio, composante relativement peu diffusée dans des contrôleurs de jeux. L'idée est néanmoins bonne pour une meilleure immersion du joueur. Techniquement, il s'agit du BH7824FVM de Rohm, désigné particulièrement pour les téléphones portables. Il a une consommation de pointe de 500mw et se vend aux alentours de 1.56 dollar.

Après cela nous avons également de la mémoire intégrée, qui permet de sauvegarder quelques données utiles.

[12] Source de la photo : http://www.sparkfun.com

[13] Tous les prix présents ici sont tirés de / http://www.digikey.com

[14] www.broadcom.com/products/Bluetooth/Bluetooth-RF-Silicon-and-Software-Solutions/BCM2042



MÉMOIRE INTÉGRÉE (SOURCE : HTTP://WWW.SPARKFUN.COM)

Cette mémoire EEPROM est de type M24128 128kbit, qui revient environ à 2 dollars pièce.

Pour terminer sur les composantes importantes, il reste le capteur CMOS qui permet à la Wiimote de pointer à l'écran. Il s'agit d'un PixArt's CMOS *sensor*, d'une résolution de 1Mpixel.

En résumé, la Wiimote est un concentré de technologies relativement onéreuses. Sans capteur CMOS, on peut acquérir pour 110 dollars un contrôleur du même type avec accéléromètre et support Bluetooth[15], alors qu'une Wiimote peut être acquise pour environ 60 dollars. Il est bien évident que l'on compare du matériel de développement en vente à la pièce à du matériel construit à des milliers d'exemplaires, mais de notre point de vue le résultat est le même, car nous ne souhaitons que quelques-uns de ces périphériques pour le développement et les tests. De plus, tous les composants sont standards et liés à l'équipement des claviers, des téléphones portables ou des manettes de jeux, ce qui en fait un périphérique simple à utiliser en toutes circonstances, car les *datasheets* sont à disposition et le Bluetooth est conçu pour être utilisé pour des claviers ou des souris.

### Accessoires

La Wiimote possède un port d'extension propriétaire sur lequel on peut brancher des périphériques supplémentaires. Mise à part la *Sensor bar* dont nous parlons plus loin, la majorité des périphériques se branchent sur ce port d'extension, comme le Nunchuk, le Classic Controller, le Wii Fitness Board ou encore une guitare.

[15] http://www.sparkfun.com/commerce/product_info.php?products_ID=254


WIIMOTE, VUE DE TOUTES SES FACES. ON REMARQUE LE PORT D'EXTENSION (EN BAS) ET LE RÉCEPTEUR INFRAROUGE (EN HAUT) (SOURCE : HTTP://WII.NINTENDO.COM)

## *SENSOR BAR*

LA *SENSOR BAR* ET SON CÂBLE (SOURCE : HTTP://WII.NINTENDO.COM)

La *Sensor bar* n'est pas un récepteur comme le nom semble l'indiquer mais un émetteur infrarouge. La Wiimote possède un capteur CMOS à l'avant (avec filtre infrarouge), qui lui permet de suivre jusqu'à 4 points infrarouges. Cela permet de calculer son déplacement et d'afficher son positionnement à l'écran. Pour cela, il faut une source infrarouge située au-dessus ou au-dessous de l'écran, d'où l'utilité de cette barre. Elle est alimentée directement par la console, mais il est tout à fait possible de s'en procurer une à pile ou d'en construire une manuellement avec quelques diodes infrarouges.

L'INTÉRIEUR DE LA *SENSOR BAR*, AVEC SUR LA GAUCHE ET LA DROITE LES ENSEMBLES DE *LEDS* INFRAROUGES (SOURCE : WIILI.ORG)

La *sensor bar* est conçue d'origine pour être fixe, néanmoins le capteur de la Wiimote peut être utilisé à d'autres fins si on inverse le processus, c'est-à-dire en utilisant la Wiimote comme point fixe pour détecter des points infrarouges mouvants. Johnny Chung Lee[16] a conçu divers dispositifs permettant d'effectuer du *tracing* d'objets devant un écran, créé un *touchboard* interactif uniquement à l'aide d'une Wiimote et d'un stylo infrarouge et récemment a mis au point un casque pour détecter la

[16] Ph.D. Graduate Student, Human-Computer Interaction Institute Carnegie Mellon University

position de la tête dans le cadre de projets de réalités virtuelles[17]. Ces projets ont un très fort potentiel et le lecteur est invité à visionner les vidéos de démonstration de ce doctorant.

## NUNCHUK

NUNCHUK SEUL (SOURCE : HTTP://WII.NINTENDO.COM)

NUNCHUK CONNECTÉ À LA WIIMOTE : TENUE TYPIQUE (SOURCE : HTTP://WII.NINTENDO.COM)

Le Nunchuk, lorsqu'il est connecté à une Wiimote, permet non seulement d'occuper la main gauche de l'utilisateur, mais ajoute également un joystick analogique, deux gâchettes et surtout, un 2e accéléromètre 3 axes, ce qui monte le nombre d'axes de libertés à 10. Celui-ci envoie ses informations directement à la Wiimote via le port propriétaire (dont le protocole est I2C) qui les transmet ensuite à la console.

[17] http://www.cs.cmu.edu/~johnny/projects/wii/

## WII BALANCE BOARD

(SOURCE : HTTP://WII.NINTENDO.COM)

Le Wii Balance Board n'est pas encore distribué en Europe. Cet appareil un peu étrange est en fait une balance évoluée, dont toute la surface est criblée de capteurs de pression. Elle détecte ainsi le moindre mouvement de l'utilisateur qui est debout (ou dans d'autres positions) dessus.

Outre son potentiel élevé pour les jeux vidéos, il pourrait parfaitement s'adapter au contrôle de certains robots adaptés à ce périphérique ou à concevoir des simulations personnes–machines en imitant ou détectant le mouvement d'un humain sur ses jambes.

# Sixaxis


LA MANETTE SIXAXIS (SOURCE : HTTP://WWW.PLAYSTATION3.COM/)

## Introduction

La manette Sixaxis de Sony tire son nom de son accéléromètre 3 axes. Ce périphérique, contrairement à celui de Nintendo, ne contient que des modules propriétaires ce qui rend son analyse très difficile. Ceci est tout à fait compréhensible étant donné que Sony fabrique lui-même les composantes nécessaires à la conception de ce genre d'appareil, il n'allait donc pas faire appel à une entreprise externe alors qu'il peut le faire lui-même. L'étude de ses caractéristiques techniques se limitera donc ici à l'aspect externe du périphérique.

## Caractéristiques

Cette manette est caractérisée par trois points principaux : le sans-fil, les boutons analogiques et l'accéléromètre. En effet, celle-ci dispose d'un module Bluetooth lui permettant de se connecter à la console, qui elle-même peut supporter jusqu'à 7 périphériques Bluetooth. Elle a une batterie (interne) qui peut être rechargée via la prise USB à l'avant. De plus, une version *rumble* est sortie récemment, dotée d'un retour de force à l'aide de moteurs désaxés.
Ensuite, l'accéléromètre et tous les boutons analogiques permettent d'avoir une multitude d'axes de libertés : chaque bouton a une réponse analogique d'une précision de 10 bits, idem pour les deux joysticks et pour la croix directionnelle. Cela monte le nombre d'axes à 17, ce qui permet d'avoir un contrôle très poussé et très précis.

# Conclusion

Comme on a pu le remarquer durant ce chapitre, les manettes des deux concurrents sont très semblables et très différentes à la fois. Avec la Wiimote, c'est l'intégralité du périphérique qui est adapté pour s'axer autour des nouvelles fonctionnalités, avec un port d'extension pour la faire évoluer. Tout est axé sur le pointage et la gestuelle, étant donné qu'on peut effectuer des mouvements des deux mains, ce qui permet dans l'absolu de reproduire les mouvements des bras. En contre-partie, elle est assez pauvre au niveau des boutons, que cela soit au niveau du nombre ou de la précision de ceux-ci puisqu'aucun d'entre eux n'est analogique. La Wiimote est bien plus adaptée pour simuler des mouvements humanoïdes à l'aide des bras. La Sixaxis, en revanche, conserve l'idée des anciennes manettes, que l'on tient à deux mains et sans extension possible. L'accéléromètre n'est que secondaire dans le design final et ne permet pas autant de variantes acrobatiques qu'avec une Wiimote-Nunchuk. En revanche, les boutons sont nombreux et le tout analogique permet de remplacer les mouvements de la concurrente par une précision lors des différentes pressions. Elle est donc parfaitement adaptée au contrôle de véhicules ou d'animaux.

Enfin, la Wiimote se démarque par son utilisation de composantes standard : accéléromètre, chip Bluetooth de clavier, chip audio de téléphone portable, alimentation à base de piles et on verra plus tard que même au niveau du protocole de communication nous avons un comportement normalisé alors que la Sixaxis peut poser quelques problèmes.

# LE PROTOCOLE BLUETOOTH


## Introduction

Étant donné que la grosse majorité du projet se déroule en compagnie de ce protocole, une étude et une bonne compréhension de celui-ci sont indispensables et c'est là le sujet de ce chapitre.
Il est relativement aisé de trouver des informations sur le protocole Bluetooth dans les bibliothèques ou sur internet. La référence officielle est même libre au téléchargement sur le site de l'organisation Bluetooth, toutefois ce protocole est très complexe à étudier. En effet, pour passer dans le domaine du Wireless, il ne suffit pas d'enlever les câbles et de les remplacer par une liaison radio. De plus, le protocole Bluetooth est compliqué, car des applications sont disponibles à chaque niveau de la *stack* : il ne suffit pas de connaître la dernière couche pour maîtriser toutes les utilisations du Bluetooth. Enfin, le protocole Bluetooth ne peut pas être complètement compris uniquement par la théorie et seule la pratique permet de saisir les fondements et l'utilisation de telle ou telle couche de la *stack*. Le but de ce dossier est donc d'apporter au lecteur notre expérience et les connaissances récoltées sur le Bluetooth tout au long de ce projet afin de lui faciliter la compréhension de la suite du projet et lui fournir une solide base théorique dans le domaine.

## Pourquoi le Bluetooth ?

Quand une nouvelle technologie apparaît, il est toujours judicieux de se demander pourquoi elle émerge. Certaines sont une évolution logique de ce qui existe déjà, d'autres ne sont que des arguments pour vendre du matériel neuf et certaines veulent combler des lacunes. C'est ici le cas du Bluetooth, dans le domaine de la transmission sans fil. Nous n'allons pas argumenter le pour et le contre de la communication avec ou sans fil sous peine de transformer ce dossier en traité philosophique. Concentrons-nous plutôt sur ce qui existe déjà en matière de sans-fil pour les périphériques électroniques, à savoir le Wireless (802.11) et l'infrarouge.

### Wireless 802.11 (WiFi)

Le Wireless est originellement une norme permettant de mettre en place un WLAN, un Wireless Local Arena Network, autrement dit un réseau local d'ordinateurs. Le WiFi nécessite une *stack* TCP/IP pour fonctionner, ce qui rend le WiFi tout à fait adéquat pour un réseau local. En revanche, s'il s'agit de faire de petits transferts de fichier, la mise en place de tout le protocole est inutile et coûteuse. De plus, connecter des périphériques HID comme une souris ou un clavier en WiFi serait une absurdité, autant au niveau des implémentations à effectuer, mais surtout au niveau de la consommation. Une puce WiFi d'ordinateur portable consomme entre 700–1300 mW et 100 mW au repos. Des puces à basse consommation commencent à être conçues, pouvant baisser la consommation à 270 mW (comme le BCM4326 AirForce One Ultra–Low Power 802.11b/g[18]), mais cela reste assez conséquent pour de petits périphériques comme des claviers, des robots, des souris ou des téléphones portables.

### Infrarouge

L'infrarouge est très simple, au contraire du WiFi. Il consomme peu et les protocoles de transfert sont très faciles à comprendre. En revanche, l'infrarouge est tout de suite mis sur la touche en ce qui concerne la vitesse de transmission et la portée. En effet, il faut compter maximum 10cm de distance pour que la transmission s'effectue et il est indispensable d'être aligné et en vue. Impossible d'avoir son téléphone portable dans la poche et son oreillette infrarouge… d'où l'apparition du Bluetooth pour combler les lacunes de ces deux protocoles.

### Bluetooth

Le Bluetooth est le remède à tous ces maux : fréquence à 2.4Ghz (comme le WiFi), donc passe–muraille. Plus besoin d'être en vue et la portée est beaucoup plus élevée que 10cm. La *stack* Bluetooth est considérablement simplifiée par rapport à la *stack* TCP/IP pour autoriser toutes sortes de transmission (échange de fichier, envoi asynchrone de périphérique de contrôle, transmission audio en streaming, etc.) et la consommation est faible et sélectionnable selon la puissance désirée :

- Classe I : 100 mW, portée jusqu'à 100 mètres
- Classe II : 2.5 mW, portée jusqu' 10 mètres
- Classe III : 1 mW, portée jusqu'à 1 mètre[19]

Les classes II sont les plus communes, car pour une très légère augmentation de la puissance on obtient une bonne portée maximale et il faut tenir compte des interférences et des éventuels obstacles. De plus, le protocole est évolutif pour supporter des nouveaux types de périphériques.

---

[18] http://www.broadcom.com/products/Wireless-LAN/802.11-Wireless-LAN-Solutions/BCM4326

[19] Remarque : ces spécifications proviennent du site officiel Bluetooth.com. On peut trouver ces spécifications un peu partout sur Internet et dans la littérature avec à chaque fois des valeurs fantaisistes ou différentes, comme 25 mètres pour le Classe II ou 10cm pour le Classe I. Que le lecteur ne soit donc pas étonné de trouver d'autres valeurs pour les classes de puissance.

En ce qui concerne la vitesse de transfert, on peut attendre du 1 Mbs avec le Bluetooth 1.2 jusqu'à 3 Mbs avec la version 2.0 + EDR (*Enhance Data Rate*), ce qui est largement suffisant pour des transferts de petits fichiers ou du streaming audio. Si on veut un plus gros débit, le Wireless est plus adéquat ou il suffit d'attendre les prochaines normes Bluetooth qui permettront d'atteindre 20Mbs.

## Un peu d'histoire

Harald 1er Blåtand fut le roi du Danemark en 940. Son nom signifiait "dent bleue". Il imposa le christianisme, combattit beaucoup et unifia les tribus de son pays. Une pierre fut érigée à Jelling sur laquelle les runes suivantes sont gravées :

HARALD CHRISTIANIZED THE DANES
HARALD CONTROLLED DENMARK AND NORWAY
HARALD THINKS NOTEBOOKS AND CELLULAR PHONES SHOULD COMMUNICATE SEAMLESSLY [20]

Sans transition aucune, le fabricant suédois Ericsson créa le protocole Bluetooth en 1994 en s'inspirant de son nom et de sa rune, en déclarant que le protocole Bluetooth sert à unifier les périphériques comme Harald unifia les tribus. Le nom définitif fut fixé en 1998 par le SIG, le Bluetooth Special Interest Group, comprenant entre autres Sony Ericsson, IBM, Nokia ou encore Intel. Voici un petit historique, tiré du site Bluetooth.com :

- 1999 : Les spécifications Bluetooth 1.0 sont publiées.
- 2000 : Le premier téléphone mobile avec le premier casque–micro et la première carte PC dotés de Bluetooth sont mis sur le marché. Des prototypes de souris, ordinateurs portables et clés USB sont présentés en démonstration.
- 2001 : Première imprimante, premier ordinateur portable et premier kit mains libres pour voiture, doté de reconnaissance vocale.
- 2002 : Les périphériques se multiplient : premier duo clavier–souris, premier GPS et caméra numérique. Le nombre de produits certifiés Bluetooth se monte à 500. Année importante, la IEEE approuve les spécifications Bluetooth 802.15.1.

[20] http://developers.sun.com/mobility/midp/articles/bluetooth1/ Il est bien clair que les runes gravées sur cette pierre ne disent pas ceci, mais l'idée paraît toutefois bonne.

- 2003 : Premier lecteur mp3 Bluetooth (bien que l'utilité du Bluetooth soit assez obscure pour ce genre d'appareil) et le monde médical se voit doté d'un système Bluetooth. La norme 1.2 est adoptée par le Bluetooth SIG.

- 2004 : Les spécifications 2.0 + EDR (*Enhanced Data Rate*) sont acceptées par le Bluetooth SIG. Le premier casque stéréo (avec qualité audio améliorée, auparavant la qualité audio était celle d'un téléphone).

- 2005–2006 : Le Bluetooth commence à s'intégrer un peu n'importe où : montre, cadre à photo, radio–réveil…

- 2007 : Première télévision Bluetooth et surtout, spécification 2.1 + EDR annoncée et disponible sur le site Bluetooth.

Il est important de noter que les spécifications 2.0 et 2.1 sont entièrement rétrocompatibles avec la spécification 1.1, qui est encore largement distribuée et disponible dans de multiples périphériques.
Ainsi, on voit que le Bluetooth est une technologie encore toute jeune qui est apparue quasiment en même temps que le premier Wireless utilisable, à savoir le 802.11a. Bien que ces technologies paraissent encore un peu fraîches et instables, l'évolution qu'elles ont suivie depuis une dizaine d'années promet encore de belles innovations.

## Connexions Bluetooth

Pour bien comprendre comment fonctionne le Bluetooth, nous allons examiner tout d'abord les plus bas niveaux radios pour progressivement remonter la *stack*, de manière à analyser une connexion Bluetooth depuis le plus bas niveau, ce qui permet de mieux comprendre les bases d'une connexion Bluetooth.

### Le Wireless Bluetooth

Le Bluetooth est une technologie sans fil opérant à la fréquence de 2.4Ghz. Plus précisément, 79 canaux de 1Mhz de 2.402Ghz à 2.480Ghz. À noter que dans certains pays (comme la France), le nombre de canaux est réduit à 23 pour s'adapter aux règlements des fréquences du pays. Afin d'éviter au mieux les interférences sur les différentes fréquences, les périphériques Bluetooth sautent ("*hopping*") de fréquence à un taux pseudoaléatoire. Par souci de précision, il est à noter que la modulation standard utilisée est GSFK (Gaussian Frequency Shift Key)[21] pour toutes les transmissions Bluetooth 1.1 compatibles. En ce qui concerne les modules supportant Bluetooth 2.0/2.1 avec EDR (Enhanced Data Rate), seuls les codes d'accès et les en–têtes de transmission sont envoyés en GSFK.

[21] http://www.lgm.fr/ : *Le protocole Bluetooth*, p. 11

Les séquences de synchronisation ainsi que les données (*payload*) sont transmises avec une modulation de type PSK[22] (*Phase-shift keying*) à un taux de 2 (ou 3) Mbs, ce qui permet d'augmenter la vitesse de transmission tout en conservant au besoin la compatibilité avec les anciens protocoles.

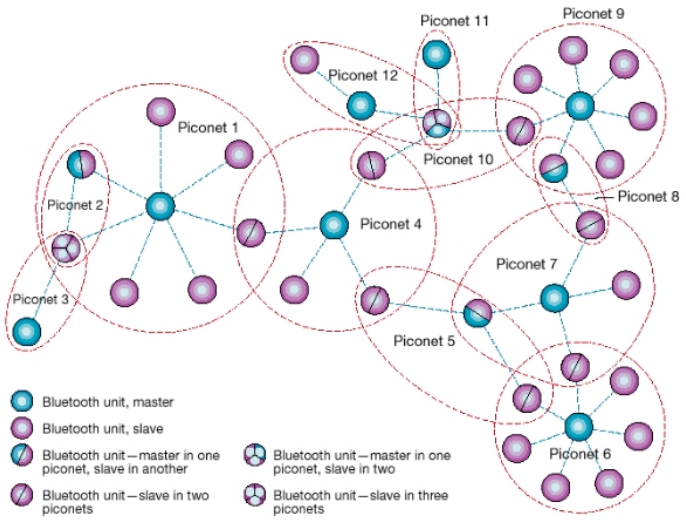
### L'organisation dans Bluetooth

Maintenant que les bases électroniques sont posées, comment sont gérés les différents périphériques Bluetooth pour qu'ils soient synchronisés et connectés ? Il faut savoir que les données sont transmises selon des *time-slots* de 0.625 ms et selon une horloge précise qui est en fait un compteur de 28 bits, ce qui lui permet de durer environ une journée. La synchronisation des périphériques au sein d'un même réseau (appelé *Piconet*) est gérée par le *master* du réseau. Les différences fondamentales entre un *master* et un *slave* sont les suivantes : les *slave*s se synchronisent sur le *master*, le *master* initialise (ce qui s'appelle *to page*) les connexions et les *slaves* sont connectés uniquement au *master*, c'est-à-dire que seul le *master* peut gérer plusieurs connexions, au nombre de 7. Le rôle peut être changé à la fin de la procédure de connexion pour différentes raisons. Lorsqu'un périphérique se connecte à un réseau, il est par définition *master* puisqu'il initialise la connexion. C'est le cas lorsqu'on connecte son téléphone portable à un ordinateur pour transférer des photos : c'est le téléphone qui démarre la connexion et qui est donc *master*. Cependant, l'ordinateur a besoin de rester *master* pour gérer d'autres périphériques, d'où un changement de rôle.

### Problèmes de hiérarchie

Le changement de rôle est une solution à la plupart des problèmes, mais pas tous. En effet, on se retrouve parfois (souvent même) avec des connexions où les deux périphériques veulent rester *master* car ils ont chacun des *slaves* connectés à eux lors de la communication. Exemple : un PDA, connecté à un clavier et une imprimante Bluetooth, veut se connecter à un ordinateur, doté lui-même d'une souris et d'un clavier Bluetooth. Pour résoudre ce genre de problème, un périphérique peut porter plusieurs casquettes et même être connecté à plusieurs piconet : il s'agit du *scatternet*. Ce type de réseau est constitué de plusieurs piconet, avec les *masters* de ces piconets qui peuvent être en même temps *slaves* dans les autres réseaux. Voici un schéma très complet (et très complexe) illustrant parfaitement ce propos :

[22] http://www.tech-faq.com/lang/fr/psk.shtml




SOURCE : WWW.ACME.ORG

Comme on peut le voir, un réseau Bluetooth peut être très complexe, contrairement à un simple réseau Wireless de type infrastructure où ce genre de configuration nécessiterait l'utilisation de ponts. Une autre utilité importante du *scatternet* est la limitation de 7 périphériques connectés à un *master* : si on souhaite brancher plus d'appareils, le *scatternet* est la solution. À présent, intéressons-nous un peu plus à la méthode de connexion des périphériques.

### Découvertes et découvrable

Deux choses sont indispensables dans le monde sans fil pour que 2 entités se "voient" : une doit être découvrable et l'autre doit chercher à la découvrir. Cela peut paraître stupide dans un monde filaire ou radio mono-fréquence, mais ici nous sommes en Wireless avec des dizaines de canaux disponibles sur lesquels on peut émettre.

Ainsi, un périphérique doit se placer dans un mode particulier (*discovery mode*) dans lequel il va émettre des paquets contenant des informations le concernant comme son adresse MAC, sa classe de périphérique et son décalage d'horloge (*clock offset*). Les raisons pour lesquelles un périphérique n'est pas en permanence en mode *discovery* sont nombreuses : il y a tout d'abord la question de la sécurité. En effet, un périphérique en permanence découvrable est une cible facile pour diverses attaques hostiles. Ensuite, ce mode est coûteux en énergie, ce qui est inadapté pour des périphériques portables fonctionnant sur batterie. Il existe un autre mode, appelé *limited discovery mode* qui ne rend le périphérique découvrable que pendant une durée limitée (1 minute en principe), ce qui évite à l'utilisateur d'oublier de remettre son appareil en mode invisible.

Du côté du *master* (celui qui veut se connecter), il doit se mettre en *inquiry mode*. Celui-ci créé alors des *trains*[23] qui se chargent de *scanner* des paquets de 24 fréquences (de manière séquentielle). La

[23] En anglais dans le texte

valeur minimale pour découvrir un périphérique est donc de 1.25ms : on part du principe que le message d'*inquiry* est émis à la même fréquence et au même instant que le *slave* scanne les *inquiry*. Le *slave* reçoit le message et lui répond : pour mémoire, le time-slot est de 0.625ms, donc un aller-retour prend 1.25ms. Ceci est fortement improbable, surtout qu'un *slave* attend en général un nombre de *slots* aléatoire (compris entre 0 et 1023) avant de répondre. Le temps moyen a été fixé à 10.24s, afin que le *master* collecte suffisamment d'informations. En pratique, 3-5 secondes sont suffisantes, bien que ces valeurs puissent dépendre énormément de l'alignement des horloges des deux appareils et de leur état. Le temps maximum suggéré par les spécifications HCI est de 1 minute, bien que 30.72 secondes soient généralement suggérées comme temps maximal.
Reste que même 10 secondes, c'est long, et en plus si l'environnement est bruité, rien ne garantit que les périphériques puissent se "voir" dans le temps déterminé à cause de paquets corrompus.
Toutefois, des solutions ont été proposées lors de diverses études afin de réduire ce temps de connexion. La toute première est l'enregistrement des périphériques connus pour une connexion directe. Cela permet d'éviter de lancer un *inquiry*, mais si l'appareil distant ne répond pas, une dizaine de secondes peuvent être perdues avant d'avoir un message d'erreur. D'autres méthodes plus ou moins élégantes ont été étudiées, comme une aide infrarouge[24], des simplifications du protocole[25] ou la mise en place d'une "couche de rendez-vous"[26].

### Connexion[27] HCI

Une fois un périphérique découvert, il faut le connecter à l'aide de l'adresse MAC obtenue lors de la découverte et en utilisant (si possible) le *clock offset* afin d'être immédiatement synchronisé avec. Cette procédure est nommée *to page*. Le *page time* peut aussi prendre un certain temps pour que les deux appareils se synchronisent, entre 0.0025 seconde au minimum jusqu'à 2.56 secondes au maximum.
Intéressons-nous d'abord à la procédure de connexion et aux paquets envoyés. Tout le bas niveau est géré par les drivers HCI, Host Controller Interface. Ces drivers HCI peuvent être décomposés en une partie BaseBand et une partie LinkManager (appelé LMP), bien que le LinkManager soit en fait une partie de la couche BaseBand qui se charge de toutes les opérations de bases. Le LinkManager gère les liens (d'où le nom) et se charge donc des connexions.
Ainsi, la première opération à effectuer après avoir fait toutes les découvertes nécessaires est une connexion HCI. Ainsi, un paquet de demande de connexion HCI commence par les bytes 0x5 0x4, suivi de l'adresse MAC du destinataire (en *little-endian*), le type de paquet et des autres données. Le dernier octet précise si le périphérique demandant la connexion autorise un changement de rôle pour devenir *slave*.

---

[24] Ryan Woodings, Derek Joos, Trevor Clifton, Charles D. Knutson, *Rapid Heterogeneous Connectio nEstablishment : Accelerating Bluetooth Inquiry Using IrDA,* Brigham Youn University, 2005

[25] Brian S. Peterson, Rusty O. Baldwin, Jeffrey P. Kharoufeh, *A Specification Bluetooth Inquiry Simplification*, United State Air Force Institue of Technology, 2004

[26] Frank Siegemund, Michael Rohs, *Rendezvous Layer Protocols for Bluetooth-Enabled smart Devices*, Institute of Technology (ETH) Zurich, 2004

[27] Référence officielle sur les connexion HCI : "Core System Package [Host volume], part E" in *Bluetooth Specification version 2.1 +EDR* [vol 2], p. 346

Exemple :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 05 04 | 0D | AD 32 65 CD 1A 03 | 18 CC | 01 00 59 | 53 | 01 |

1 : Demande de connexion
2 : Longueur de ce qui suit (13 bytes)
3 : Adresse MAC *little endian*
4 : Type de paquets supportés
5 : Valeurs *Page Scan Repetition Mode, Page Scan Period Mode* et *Page Scan Mode*. Ces valeurs sont spécifiques au périphérique sur lequel on veut se connecter et sont transmises lors du *discovery*.
6 : *Clock offset* (décalage d'horloge) par rapport à notre horloge. Ce décalage est également transmis lors du *discovery* pour accélérer la connexion.
7 : Autorisation de passer en mode *slave* après la connexion

La demande de connexion reçoit en retour soit une confirmation de la connexion, soit une déconnexion :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 04 | 0B | 00 | 20 10 | AD 32 65 CD 1A 03 | 01 | 00 |

1 : Réponse de connexion
2 : Longueur de la réponse
3 : Connexion acceptée
4 : ID de la connexion
5 : Adresse MAC *little-endian*
6 : Type de connexion (ACL ou SCO)
7 : Cryptage désactivé

### ACL ou SCO ?

Les connexions de type ACL (*Asynchronous Connection Less*) sont des connexions asynchrones. Les connexions peuvent en effet être symétriques (entre 108.8 et 432.6 kbps) ou asymétriques (721kbps, 57.6kbps). Un *slave* ne peut avoir qu'une seule connexion ACL ouverte et on s'en sert en général pour communiquer ensuite avec la couche supérieure (L2CAP, qui sera étudié au chapitre suivant).
Les connexions de type SCO (Synchronous Connection Oriented) servent principalement pour l'audio et pour des connexions symétriques. Les kits-oreillettes sont un excellent exemple. Un *slave* peut supporter jusqu'à 3 connexions SCO et il est important de noter que les paquets SCO ne sont jamais

retransmis (comme le protocole UDP). C'est pourquoi ils sont typiquement utilisés pour les applications de "streaming" audio.

### Déconnexion

Une déconnexion peut être envoyée n'importe quand et par n'importe qui (*slave* ou *master*). Un seul paquet est nécessaire pour cela :

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 05 | 04 | 0 | 20 | 10 | 0b |

1 : Déconnexion (un *slave* peut se déconnecter sans demande)
2 : longueur de ce qui suit (4 bytes)
3 : Déconnexion effective
4 : ID de la déconnexion (la même que lors de la connexion)
5 : raison de la déconnexion, ici la raison est *ACL connection already exists*[28]

### Format des paquets

Jusqu'à maintenant, nous avons vu deux genres de paquets de données au niveau HCI : paquets ACL et SCO. Or, ils peuvent être de plusieurs types[29] afin de s'adapter aux besoins de l'application. Les principales différences entre ces différents paquets se situent au niveau du CRC, de la taille des informations transmissibles, de l'encodage utilisé et du nombre de time *slots* occupés. Les périphériques Bluetooth s'accordent lors de la connexion ACL sur le type de paquets utilisés, car les versions 2–X et 3–X ne sont supportées que par la spécification 2.0 du protocole Bluetooth.

### Et ensuite ?

Une fois une connexion physique HCI établie, on peut alors l'utiliser soit pour ouvrir une connexion audio, soit pour passer le contrôle à des couches supérieures afin d'ouvrir d'autres connexions plus spécifiques.

## La *stack*

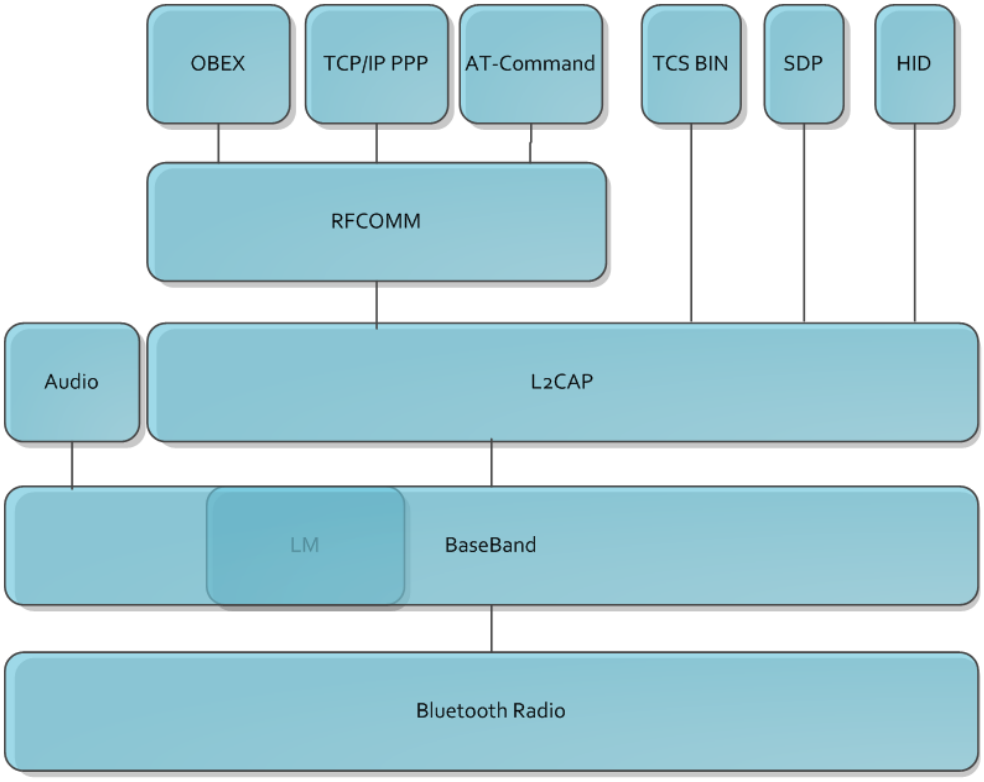
### Introduction

Quand on parle de *stack* (pile), on pense automatiquement à la *stack* TCP/IP. Le principe reste le même, sauf que la *stack* Bluetooth est assez particulière car elle permet une très grande adaptabilité et donc peut fournir un grand nombre de fonctionnalités tout en conservant la base du protocole.

---

[28] Liste des codes d'erreur : "Core System Package [Host volume], part E" in *Bluetooth Specification version 2.1 +EDR [vol 3]*, p. 494

[29] Liste exhaustive des différents paquets : "Core System Package [Host volume], part E" in *Bluetooth Specification version 2.1 +EDR [vol 3]*, p. 122




*STACK* BLUETOOTH

La *stack* Bluetooth est sujette à de nombreuses interprétations de la part de ceux qui l'expliquent. Pour éviter tout malentendu ou mal-information, nous nous baserons uniquement sur la spécification officielle pour la définir dans ce qui suit.

### HCI, Baseband et LMP

Les niveaux les plus bas du protocole, que nous avons brièvement abordés, et que l'on regroupe sous le terme HCI, est composé des 2 éléments suivants :

- Baseband : il s'agit de la toute première couche qui se situe au-dessus des contrôleurs radios *hardware*. Elle permet de lancer des *discovery*, de faire des connexions ACL/SCO et de gérer la sécurité (pin, authentification). Elle comprend le LM, utilisant le LMP (Link Manager Protocol).

- Link Manager (LM) : Cette sous-couche de Baseband se charge de tout ce qui est gestion de connexion, déconnexion. Il est également responsable de la gestion des types de paquets.

Ainsi, le HCI (Host Controler Interface) est un regroupement des drivers *Baseband*, fournissant une interface unifiée pour commander à la puce Bluetooth.
À présent que les termes sont clairement définis, passons aux couches supérieures.

### L2CAP[30]

La couche L2CAP se connecte directement au-dessus du Baseband, via une connexion ACL (pas de SCO pour le L2CAP). L2CAP signifie *Logical Link Control and Adaptation Protocol*. C'est la couche qui s'occupe de multiplexer les connexions et les services, de fragmenter les paquets au besoin et se charge aussi du Qos (*quality of service)* si nécessaire.
Lors de l'ouverture d'un canal de connexion L2CAP, on spécifie 3 paramètres : Le SCID (*Source Canal* ID), le DCID (*Destination Canal ID*) et le PSM (*Protocol Service Multiplexer*). Le PSM est souvent considéré comme un canal, ce qui n'est pas faux, mais c'est une source de malentendus, car on peut le confondre avec les SCID et les DCID. Le PSM ne correspond pas à une fréquence ou à un port physique : il sert uniquement à préciser quel service est utilisé par cette connexion, alors que les SCID et les DCID peuvent être considérés comme des ports d'écoute et d'écriture.
La liste complète des PSM n'est désormais disponible que pour les membres du SIG[31]. Néanmoins, en voici les premières valeurs à titre d'information :

- 0x0001 : Service *Discovery* Protocol
- 0x0003 : RFCOMM
- 0x0005 : Telephony Control Protocol
- 0x0007 : TCS cordless
- 0x000F : BNEP
- 0x0011 : HID Control
- 0x0013 : HID Interrupt

Les 1000 premières valeurs sont réservées, le reste est à disposition pour les applications personnelles. En ce qui concerne les CID, le 01 est réservé à l'ouverture, fermeture et test des connexions L2CAP et le 02 au mode *connectionless* (non connecté). Ce mode sert principalement à envoyer des paquets de *broadcast* (diffusion large), c'est pourquoi nous allons nous concentrer sur comment réaliser une connexion au niveau L2CAP.

---

30 "Core System Package [Host volume], part A : Logical Link Control and Adaptation Protocol Specification" in *Bluetooth Specification version 2.1 +EDR [vol 3]*, p. 13

31 Rappel : le SIG est le *special interest group*, organisation regroupant les principaux acteurs du monde Bluetooth contribuant à son développement.

EXEMPLE :

Après avoir ouvert une connexion HCI en mode ACL, une connexion L2CAP se décompose en 2 étapes principales : connexion et double configuration. Voici les étapes en détail :

1. Demande de connexion L2CAP
2. Acceptation de la connexion L2CAP
3. Demande de configuration (*Master*)
4. Réponse de configuration (*Slave*)
5. Demande de configuration (*Slave*)
6. Réponse de configuration (*Master*)
7. Transmission du contrôle à un niveau plus élevé de la *stack*
8. Déconnexion

Au niveau matériel, les paquets L2CAP sont encapsulés dans des paquets ACL, dont l'en–tête est constitué d'un *handle* de connexion et de la longueur des données. Une fois cette encapsulation éliminée, voici les codes des paquets L2CAP :

Demande de connexion :

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 08 00 | 01 00 | 02 01 | 04 00 | 00 11 | 40 00 |

1. Taille de ce qui suit
2. Numéro du canal de commande
3. Demande de connexion et identificateur
4. Taille de ce qui suit
5. Identificateur du PSM
6. Canal ouvert en local (SCID)

Réponse :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0c 00 | 01 00 | 03 | 01 | 08 | 00 30 | 00 40 | 00... |

1. Taille de ce qui suit
2. Numéro du canal de commande
3. Réponse à une connexion
4. Identificateur de commande
5. Taille de ce qui suit
6. Canal distant ouvert (DCID)
7. Canal local ouvert (SCID)
8. Indique que tout s'est bien déroulé

Demande de configuration :

| | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 08 00 | 01 00 | 04 02 | 04 00 | 00 30 | 00... |

L'en-tête est identique à la demande de connexion jusqu'au 04 02, qui correspond à une demande de configuration et son ID :

1. Demande de configuration et ID de la commande
2. Taille de ce qui suit
3. DCID
4. Aucune configuration particulière demandée

Réponse de configuration :

| 1 | 2 | 3 |
|---|---|---|
| 0D 00 01 00 | 05 02 | 06 00 40 00 00 00 00 |

1. En-tête standard
2. Réponse de configuration (05) et ID de la commande (02)
3. Taille, SCID et acceptation de toutes les options

Une fois ceci fait, les périphériques peuvent communiquer en L2CAP avec le format standard :

| 1 | 2 | 3 |
|---|---|---|
| 03 00 | 30 00 | 01 02 03 |

1. Taille
2. Canal
3. Data quelconques

Voyons à présent les différentes couches pouvant se greffer au-dessus du protocole L2CAP. Celles-ci n'étant pas nécessaires à la compréhension de la suite du projet, la description ne sera pas autant détaillée que pour le L2CAP et le HCI.

### SDP

Le SDP, Service *Discovery* Protocol, est une base de données disponible dans les périphériques Bluetooth (pas tous, ceci n'est pas obligatoire) à l'intérieur de laquelle on trouve tous les détails concernant le périphérique, ses fonctions et comment s'y connecter.
Cette base de données contient une liste de services et à chaque service sont attribuées des informations de configuration et de connexion. Lorsqu'un périphérique fait une recherche sur le réseau, il peut effectuer une requête SDP sur les périphériques découverts pour examiner si ceux-ci sont compatibles avec la fonction désirée (téléphonie, imprimante, échange de données...).
Il est important de noter qu'une découverte SDP se fait au-dessus d'une connexion L2CAP, via le PSM 1. Cela signifie qu'une connexion L2CAP supplémentaire est nécessaire pour découvrir les services disponibles. Celle-ci sera suivie d'une autre connexion L2CAP, cette fois-ci sur le PSM adéquat.

### RFCOMM

Le service RFCOMM est bien particulier et apparaît souvent lors du développement de logiciel Bluetooth. Il s'agit en fait simplement d'une émulation de connexion série RS232. Il se greffe au-dessus de L2CAP et utilise le PSM 3. Une fois une connexion RFCOMM ouverte, on dispose de 60 ports séries au maximum via lesquels on peut directement communiquer comme si les périphériques étaient câblés à l'aide d'une liaison série.
Ce service est donc très utile pour fournir un mode *bypass*, c'est-à-dire un mode qui est transparent et qui permet de communiquer "comme si" il n'y avait pas de Bluetooth mais juste un câble (ou plutôt, 60 câbles).
Il est important de remarquer que malgré ces connexions séries, celles-ci ne permettent pas de communiquer en série avec la puce Bluetooth mais uniquement avec le périphérique distant.

### Conclusion

L2CAP, SDP et RFCOMM sont les bases de la *stack* Bluetooth, bien que le L2CAP seul est nécessaire pour y greffer ensuite divers profils, que nous allons survoler dans le chapitre suivant.

## Les profils

Un profil est une surcouche à la *stack* standard Bluetooth. Il doit décrire sa dépendance envers les autres profils (qui sont empilables), les formats utilisés ainsi que les parties de la *stack* Bluetooth utilisées.

Voici une liste (non exhaustive) des profils décrits sur le site de Bluetooth[32]:

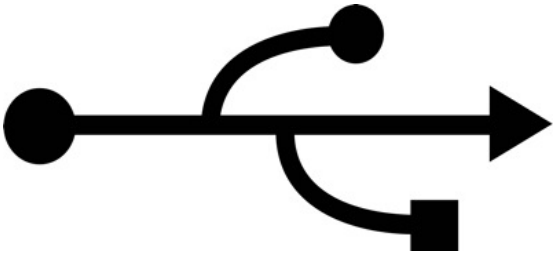
| | |
|---|---|
| Advanced Audio Distribution Profile (A2DP) | Audio / Video Control Transport Protocol (AVCTP) |
| Audio / Video Distribution Transport Protocol (AVDTP) | Audio/Video Remote Control Profile (AVRCP) |
| Basic Imaging Profile (BIP) | Basic Printing Profile (BPP) |
| BNEP | Common ISDN Access Profile (CIP) |
| Cordless Telephony Profile (CTP) | Dial-up Networking Profile (DUN) |
| Extended Service *Discovery* Profile (ESDP) | Fax Profile (FAX) |
| File Transfer Profile (FTP) | Generic Access Profile (GAP) |
| General Audio/Video Distribution Profile (GAVDP) | Generic Object Exchange Profile (GOEP) |
| Hands-Free Profile (HFP) | Hard Copy Cable Replacement Profile (HCRP) |
| Headset Profile (HSP) | Human Interface Device Profile (HID) |
| Intercom Profile (ICP) | Object Exchange (OBEX) |
| Object Push Profile (OPP) | Personal Area Networking Profile (PAN) |
| RFCOMM | Service *Discovery* Protocol (SDP) |
| Service *Discovery* Protocol (SDP) | SIM Access Profile (SAP) |
| Serial Port Profile (SPP) | Synchronization Profile (SYNC) |
| Telephony Control Specification (TCS-Binary or TCP) | Telephony Control Specification (TCS-Binary or TCP) |
| WAP Over Bluetooth Profile (WAP) | |

Le lecteur est invité à se référer à cette page pour de plus amples informations à propos des profils, étant donné que le profil particulier qui nous intéresse ici est le BTHID, ou Human Interface Device Profile (HID). Il peut également trouver une référence détaillée de tous ces profils sur le site de l'organisation Bluetooth[33].

[32] http://www.bluetooth.com/Bluetooth/Learn/Works/Profiles_Overview.htm

[33] http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/

## Le BTHID, ou le Bluetooth-USB



### Introduction

Parmi tous les profils existants et qui décrivent les types d'application utilisant le Bluetooth, nous allons maintenant nous intéresser au HID. En effet, les périphériques de contrôle (joystick, joypad, Wiimote, Sixaxis) s'apparentent à des périphériques HID, qui comprennent les souris, les claviers, les joysticks et une multitude d'autres périphériques comme les pistolets à scanner les code-barres. Le HID est indispensable à l'interaction personne-machine, car il est le protocole définissant les règles de cette communication. Le HID est la norme qui nous permet d'utiliser les souris, les claviers et tous les périphériques autorisant un être humain à contrôler une machine.

### HID[34]

Un périphérique compatible HID a l'avantage qu'il ne nécessite en général pas de pilotes de périphérique (*drivers*) particuliers : il s'agit d'un standard et la norme HID permet d'utiliser l'appareil avec les fonctionnalités basiques. Un clavier compatible HID sera toujours utilisable, même s'il dispose de 40 boutons spéciaux programmables (qui ne seront pas accessibles).

La classe HID comporte basiquement un ensemble de périphériques utilisés par les humains pour contrôler des systèmes informatiques : souris, trackball, claviers, boutons, pavé numérique, manettes de jeux, etc.

Pour transférer des données, on utilise des *Report Descriptor*. On les emploie pour envoyer/recevoir des données (*input/output report*) ou des données de configuration (*feature report*). Un *report* contient ensuite diverses sections avec les informations spécifiques du périphérique : état d'un bouton, valeur d'un déplacement, etc. À noter qu'un appareil HID peut être activé dans deux modes : le mode *report*, qui permet d'avoir toutes les fonctionnalités HID possibles et le mode *boot*, qui met le clavier ou la souris en mode de fonctionnement minimal pour une compatibilité optimale. Ce mode n'est disponible que pour les souris et les claviers (bien qu'on puisse utiliser une manette en mode clavier). Pour la souris, trois boutons sont supportés ainsi que les axes x et y. Le clavier, lui, supporte 104 touches. C'est grâce à ces mécanismes que l'on peut connecter des souris et des claviers sur un ordinateur via USB sans utiliser de *drivers* particuliers dans la plupart des cas.

[34] http://www.USB.org/developers/hidpage/

Ainsi, ce HID brièvement expliqué est originellement implémenté sur USB, mais étant donné la prolifération des périphériques sans fil et le Bluetooth étant la solution standard, le HID a été adapté au Bluetooth.

### BTHID

Le HID adapté au Bluetooth est, dans l'absolu, identique. Toutefois, il nécessite une sous-couche adaptative pour qu'il s'intègre dans Bluetooth.

La première chose importante à noter est que le BTHID se connecte sur la couche L2CAP de la *stack* Bluetooth. C'est un point très important, car cela signifie que si l'on ne peut pas accéder à la couche L2CAP mais qu'à des couches supérieures, il est impossible d'utiliser le HID. Le deuxième point à garder à l'esprit c'est que l'USB et le Bluetooth sont fondamentalement différents sur un point : la présence ou non d'un fil. Cela inclut des problèmes de sécurité, de bande passante, d'autonomie ou encore de connexions, auxquels on ne doit pas forcément faire attention en filiaire.

En ce qui concerne la connexion, il faut que les deux périphériques connaissent les paramètres de connexions l'un de l'autre. Il existe des standards qui peuvent être utilisés comme le *Boot Mode* qui permet d'activer le périphérique comme une souris ou un clavier standard. Sinon, toutes les informations nécessaires à son fonctionnement et à sa connexion sont disponibles dans un enregistrement du SDP (service *discovery* protocol). L'hôte doit donc y quérir les informations nécessaires à l'établissement d'une connexion avec le périphérique avant la connexion proprement dite.

Ensuite, étant donné que le périphérique Bluetooth n'est pas connecté à l'ordinateur via un câble, il doit fonctionner sur batterie, ce qui implique d'être strict sur la consommation et d'implémenter un système de mise en veille du périphérique. Bluetooth fournit des standards pour économiser l'énergie (*sniff mode, park mode*) et il convient de les utiliser à bon escient : un clavier Bluetooth ne doit pas rester tout le temps allumé, mais il doit répondre rapidement à une sollicitation extérieure après une mise en veille.

De plus, vu la nature *Wireless* du Bluetooth, la sécurité n'est pas à négliger. Si un ordinateur accepte que n'importe quel clavier se connecte, des problèmes peuvent se poser. Ceci reste néanmoins du ressort de la machine hôte et non pas du périphérique. À moins d'un scénario catastrophique, personne ne va placer d'analyseur Bluetooth à moins de 10 mètres d'un ordinateur pour en capter les données.

Enfin, un appareil doit être simple à connecter. Sur une souris ou une manette, on ne dispose souvent que d'un seul bouton de connexion qui doit se charger d'effectuer toutes les opérations de connexion à notre place. De plus, la reconnexion doit être rapide : s'il faut attendre 10 secondes chaque fois qu'on veut utiliser sa souris, elle va rapidement être délaissée par l'utilisateur. Il est donc important que les périphériques aient déjà en mémoire l'appareil sur lequel se connecter. Ceci est un point à ne pas négliger et qui explique pourquoi les nombreux périphériques HID existants

sont tous (à quelques rares exceptions) fournis avec un dongle Bluetooth USB configuré spécifiquement à cet usage. Cela permet :

- D'être certain de la compatibilité entre le périphérique et son hôte : en effet, les *stack*s logicielles sont très différentes et le profil HID n'est parfois pas fourni. Dans l'absolu, seule la stack Microsoft pose de sérieux problèmes de compatiblité avec les périphériques HID.
- De simplifier le protocole de connexion : le périphérique a déjà en mémoire l'adresse du dongle Bluetooth sur lequel se connecter et celui–ci est configuré pour accepter rapidement des requêtes de la part du périphérique donné
- D'être sûr que l'utilisateur possède bien un module Bluetooth (ce qui, encore aujourd'hui, n'est pas toujours le cas)

C'est là une grosse tare du Bluetooth et de son adaptation HID : si l'on souhaite une utilisation simple et aisée, il est impératif de fournir un dongle spécifique, qui ne pourra même pas être utilisé comme dongle Bluetooth standard. Au contraire, si l'on veut fournir à l'utilisateur une plus grande marge de manoeuvre, il faut avoir beaucoup de chance pour que sa *stack* logicielle soit compatible avec le périphérique et il faut également s'attendre à des plaintes des utilisateurs ne possédant pas de dongle Bluetooth et ayant mal lu l'emballage… La compatibilité entre les *stack*s logicielles (spécifiquement sous Windows) est le deuxième gros point noir de Bluetooth. Il y a à peine quelques années, les dongles Bluetooth étaient vendus avec une liste de téléphones Bluetooth compatibles, car le standard n'en était absolument pas un et souffrait de grosses incompatibilités.
Aujourd'hui tout ceci s'est amélioré, mais le domaine du HID est encore à ses balbutiements en ce qui concerne son implémentation logicielle sur certaines stacks, ce qui oblige les constructeurs à fournir leur propre matériel. Et même lorsque les appareils sont parfaitement compatibles, des problèmes apparaissent parfois, comme nous allons le voir dans le chapitre suivant qui traite des manettes de jeux connectées sous des *stack* Bluetooth standards sous Windows et Linux.

# COMMUNICATION DES PÉRIPHÉRIQUES

## Introduction

Maintenant que les bases ont été posées, nous allons pouvoir nous intéresser au fonctionnement des périphériques de jeux Bluetooth lorsqu'ils sont en présence d'un module Bluetooth qui ne vient pas d'une console : un ordinateur par exemple. C'est là que commence le travail d'analyse pour comprendre comment ces appareils fonctionnent et quelles sont leurs réactions face à du matériel étranger : sont-ils compatibles, découvrables, connectables, reconnus, fonctionnels ? Et pour cela nous allons également étudier comment ces manettes se connectent à leur console respective.

## La Wiimote et l'informatique

### Première approche

La Wiimote est assez semblable aux périphériques Bluetooth HID courants. En effet, ceux-ci sont en général dotés d'un bouton qui sert à les rendre découvrables et connectables ou d'un bouton qui initie une connexion automatique vers un périphérique hôte pré-enregistré. En ce qui concerne la Wiimote, plusieurs choses sont à remarquer :

- Elle peut allumer la console en pressant sur la touche Power de la manette. Cette touche Power sert également à éteindre la console et à déconnecter la manette.

- Lors de l'allumage de la console, la Wiimote n'est pas automatiquement connectée : il s'agit donc probablement d'un broadcast ou d'une initialisation de connexion qui enclenche la console. Par la suite, une pression sur un bouton quelconque de la manette lui permet de se connecter à la console à condition que la manette ait été enregistrée sur la console. Une manette étrangère peut allumer une Wii, mais pas s'y connecter.

- Pour connecter de manière permanente des manettes à la console, il faut utiliser les boutons de synchronisation disponibles sur la console et la manette. Ceci permet vraisemblablement à la console d'enregistrer l'adresse MAC de la manette afin de la connecter ensuite automatiquement.

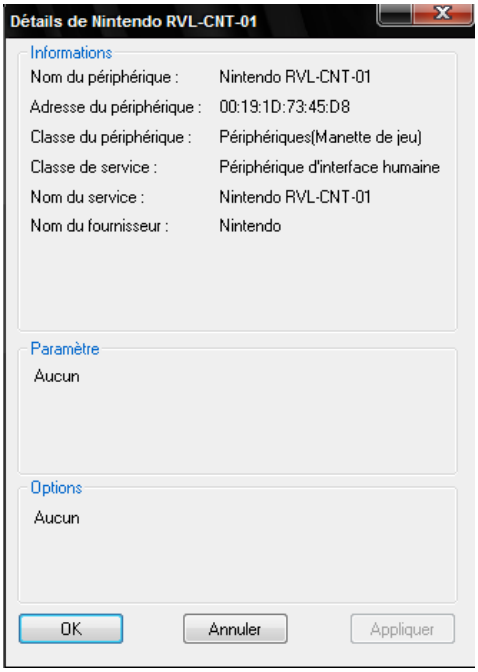
- Pour connecter de manière temporaire des manettes à la console, la Wiimote dispose d'un mode spécial où elle désynchronise tous ses périphériques pour ensuite accepter n'importe quelle connexion. On peut le faire via le bouton de synchronisation ou en pressant les touches 1+2.

Nous pouvons donc en déduire plusieurs choses. Tout d'abord, lorsque la manette est mise en mode "synchronisation" à l'aide des touches 1+2 ou de synchronisation, elle entre en mode découvrable limité durant quelques secondes. Cela signifie que la Wiimote peut rester passive en attente d'une connexion. De plus, la Wiimote dispose de deux manières pour se pairer : soit via les touches 1+2, soit via le bouton sync à côté des piles. L'expérimentation a démontré que l'apairage sur une console via 1+2 est temporaire, alors que via le bouton *sync* l'apairage est permanent. Sur un ordinateur, cela n'a aucune incidence. Est-ce que cela implique que la manette enregistre l'adresse MAC de la console ? C'est peu probable, vu qu'une manette quelconque peut allumer une Wii aléatoire. Enfin, l'allumage de la Wii est assez particulier : la pression du bouton Power allume la Wii, quelle que soit la manette utilisée. Ensuite en revanche, il faut appuyer sur un bouton quelconque pour que la manette se connecte à la console (d'où l'utilité de l'écran d'avertissement avec le "Appuyez sur le bouton A").

### Sur une machine Windows

Pour brancher un appareil Bluetooth sur un ordinateur via un dongle standard, la méthode est toujours la même pour les périphériques non enregistrés : il faut mettre le périphérique en mode découvrable, le chercher depuis l'interface fournie par la *stack* de son fabricant et initialiser une connexion. Par la suite un SDP (service *discovery* protocol) se fait du côté de l'ordinateur pour apprendre quels services sont fournis par le périphérique afin de charger les profils adéquats et, si nécessaire, les drivers nécessaires au bon fonctionnement de cet appareil.

Quatre *stack*s ont été testées : Widcomm, Bluesoleil, Microsoft et Toshiba. Pour toutes, le même cheminement a été utiilsé : tentative de découverte de la manette, connexion puis vérification des drivers installés. Pour toutes les *stack*s, le comportement a été identique ou presque. La Wiimote a été découvrable par toutes et un SDP a pu être effectué :



C'est une bonne chose : Nintendo a pris la peine d'inscrire ces informations dans le registre SDP. Ce n'est nullement obligatoire et rarement utile si l'on envisage d'utiliser des manettes Bluetooth sur du matériel connu, comme la Wii, à moins que la Wiimote n'effectue un SDP lors de l'apairage avec du nouveau matériel pour être certain de se brancher sur une Wiimote. Reste que cela nous est très utile, car les différentes *stack*s ont pu ainsi détecter de quel périphérique il s'agit et s'y connecter correctement en chargeant les drivers HID de manette de jeu. Une fois ceci fait, un filtrage des données provenant de la manette permet de l'utiliser pour diverses fonctions via un logiciel adapté (comme Glovepie[35]) pour en faire une télécommande, un clavier ou une manette de jeux.
On remarque également que lors d'une connexion à un ordinateur, le bouton power permet également d'éteindre la Wiimote. Cela signifie que l'ordre d'extinction ne vient pas de la Wii, mais bien de la Wiimote.

### Sur un ordinateur Linux

Le gros avantage d'un ordinateur tournant sous Linux est qu'on peut accéder directement au matériel sans avoir besoin de passer par le DDK[36] totalement incompréhensible pour le commun des mortels. De plus, on dispose d'un outil d'analyse, `hcidump`[37] qui permet de lire les paquets HCI transitant entre le module et le système d'exploitation.
Sans grande surprise, la manette est découvrable, mais vu les outils plus évolués dont on dispose, plus d'informations peuvent être récupérées. Voici les informations disponibles à l'aide de la commande `hcitool info 00:19:1D:73:45:D8` :

```
Requesting information ...
        BD Address:  00:19:1D:73:45:D8
        Device Name: Nintendo RVL-CNT-01
        LMP Version: 1.2 (0x2) LMP Subversion: 0x229
        Manufacturer: Broadcom Corporation (15)
        Features: 0xbc 0x02 0x04 0x38 0x08 0x00 0x00 0x00
                <encryption> <slot offset> <timing accuracy> <role switch>
                <sniff mode> <RSSI> <power control> <enhanced iscan>
                <interlaced iscan> <interlaced pscan> <AFH cap. slave>
```

Ceci est conforme au matériel contenu dans la Wiimote. Nous pouvons toutefois retenir avec intérêt que la puce supporte le cryptage, ce qui est dans le fond normal, car supporté par tous les chips Bluetooth. Néanmoins, cela ne signifie pas que Nintendo l'ait activé de manière volontaire pour un usage futur.

---

[35] http://carl.kenner.googlepages.com/glovepie_download

[36] *Drivers Development Kit* est un ensemble de librairies et de codes permettant de développer des *drivers* (pilotes de périphériques) sur un système Win32

[37] *Hcidump* est un logiciel fourni avec les utilitaires de la *stack* Bluez permettant de visualiser les paquets Bluetooth transitant sur un odinateur : http://www.bluez.org

Poursuivons par un SDP (`sdptool browse`) :

```
Browsing 00:19:1D:73:45:D8...
Service RecHandle: 0x0
Service Class ID List:
  "SDP Server" (0x1000)
Protocol Descriptor List:
  "L2CAP" (0x0100)
    PSM: 1
  "SDP" (0x0001)
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
Profile Descriptor List:
  "" (0x0100)
    Version: 0x0100

Service Name: Nintendo RVL-CNT-01
Service Description: Nintendo RVL-CNT-01
Service Provider: Nintendo
Service RecHandle: 0x10000
Service Class ID List:
  "Human Interface Device" (0x1124)
Protocol Descriptor List:
  "L2CAP" (0x0100)
    PSM: 17
  "HIDP" (0x0011)
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
Profile Descriptor List:
  "Human Interface Device" (0x1124)
    Version: 0x0100

Service RecHandle: 0x10001
Service Class ID List:
  "PnP Information" (0x1200)
Protocol Descriptor List:
  "L2CAP" (0x0100)
    PSM: 1
  "SDP" (0x0001)
Profile Descriptor List:
  "PnP Information" (0x1200)
    Version: 0x0100
```

Ces informations sont intéressantes à la lumière des connaissances acquises dans le chapitre précédent. On remarque que 3 services sont à disposition : le SDP Server (qui fournit ces informations), un service PNP (Plug and Play) d'informations et au milieu, le service HID et toutes les informations nécessaires à son utilisation comme le protocole utilisé (L2CAP), le PSM utilisé (17) et divers autres paramètres comme le nom et l'identificateur. En plus de tout cela, on peut obtenir le HID descriptor qui permet d'avoir toutes les informations pour *parser* (filtrer) les paquets envoyés par la Wiimote. Une analyse complète est disponible à l'adresse http://www.wiili.org/index.php/Wiimote.

### Conclusion

Ainsi, la Wiimote est un périphérique relativement simple à adapter malgré sa provenance du monde des consoles de jeux. Celle-ci peut se connecter aussi simplement qu'un clavier ou qu'une souris USB. La principale difficulté est de décoder les trames HID provenant de la Wiimote et de concevoir un protocole de connexion pour les *stack*s sans profile HID comme Bluez. Tout ceci a été déjà fait de nombreuses fois et tous les résultats peuvent être consultés sur le site de référence http://www.wiili.org.

## La Sixaxis et l'informatique

### Première approche

La manette Sixaxis de la Playstation 3 est très différente de la Wiimote, autant au niveau physique qu'au niveau connectique. Ainsi, sur la Sixaxis on dispose d'un port USB permettant, à première vue, de recharger la manette. Nous verrons plus loin que cette connexion USB est utile pour de nombreuses autres choses. Sinon, nous ne disposons que d'un seul et unique bouton pour interagir avec les connexions de la console, comme sur les périphériques HID normaux (claviers ou souris) : c'est le bouton *PS* dans notre cas. En voici les utilités :

- Allumer la console
- Connecter une manette secondaire sur la console
- Accéder à un menu permettant d'éteindre la manette/console
- Sortir des jeux

Vu que son utilisation pour accéder au menu d'extinction de la console ou de fermeture des jeux n'est pas instantanée, c'est-à-dire qu'il faut ensuite faire une sélection et la confirmer, on peut en déduire que son utilisation dans ces cas est purement logicielle. Cela signifie que l'appui sur ce bouton n'envoie pas de commande particulière à la console comme le ferait le bouton Power de la Wiimote.

### Sur une machine Windows

Comme lors de l'étude de la Wiimote, toutes les *stack*s citées ci-dessus ont été testées. Le protocole est le même : pression sur la touche PS de la Sixaxis puis tentative de découverte du périphérique.
La première chose que l'on remarque est que l'appui constant sur le bouton PS ne garde pas la manette dans son mode "connectable". Il faut répéter continuellement les pressions et parfois la manette ne s'éteint plus du tout (*crash* du système interne ?) et il est alors nécessaire de la *reseter* à l'aide du petit bouton caché à son dos. Contrairement à ce qu'on pouvait espérer en revanche, la manette n'est absolument pas découvrable. Cela signifie soit que la manette "sait" chez qui se connecter, soit que la manette se met en attente de connexion et que la console "sait" et cherche constamment des périphériques à connecter. La principale question ici reste la même quel que soit le

scénario : comment la console ou la manette connaît l'adresse MAC de l'autre lorsque les appareils sont neufs ?

Immédiatement on pense à la connexion USB et étant donné la présence de prise USB également sur les ordinateurs, une connexion via ce biais peut être instructive. La connexion câblée de la Sixaxis sous Windows mène par défaut à l'installation d'un périphérique inutilisable, car les drivers ne sont pas adéquats, mais il faut savoir que Sony a fourni un drivers (Windows) et un patch (Linux) pour utiliser sa manette en USB comme contrôleur de jeux. Étant donné que le drivers fourni pour Windows n'est pas ouvert et ne permet pas plus de connecter la manette en Bluetooth, l'étude se tourne rapidement dans le domaine Linux pour y trouver des pistes.

### Sur une machine Linux

Sur Linux, de très nombreuses choses ont déjà été faites et les *stack*s sont variées et multiples. Nous utiliserons donc principalement Bluez, l'implémentation Bluetooth officielle pour Linux.

Comme dit plus haut, en ce qui concerne la connexion USB, Sony a fourni un patch au contenu intéressant :

```
USB_control_msg(dev, USB_rcvctrlpipe(dev, 0),
+                                HID_REQ_GET_REPORT,
+                                USB_DIR_IN | USB_TYPE_CLASS |
+                                USB_RECIP_INTERFACE,
+                                (3 << 8) | 0xf2, ifnum, buf, 17,
+                                USB_CTRL_GET_TIMEOUT);
```

Il est intéressant de remarquer ici la présence d'un GET_REPORT, de type HID. Cette commande permet d'activer la manette afin qu'elle initialise l'envoi de données. Un patch similaire existe pour le Bluetooth, trouvé sur http://pabr.org, le site de Pascal (nom inconnu) très intéressé dans le développement Bluetooth de la Sixaxis sur ordinateurs et systèmes embarqués :

```
+static void enable_sixaxis(int csk) {
+       static const unsigned char msg[] = {
+               0x53 /*HIDP_TRANS_SET_REPORT | HIDP_DATA_RTYPE_FEATURE*/,
+               0xf4,  0x42, 0x03, 0x00, 0x00
+       };
+       write(csk, msg, sizeof(msg));
```

Avec comme test pour appeler cette fonction :

```
+       if ( req.vendor==0x054C && req.product==0x0268 )
+               enable_sixaxis(csk);
```

Ainsi, la commande SET_REPORT avec comme paramètres 0xf4, 0x42, 0x03, 0x00, 0x00 sert d'activateur pour que la manette envoie ses données, comme nous l'avons remarqué plus haut.

Une fois qu'on a vérifié avoir ce patch (qui est par défaut dans la distribution de Bluez depuis la version 3.19), il reste encore un élément inconnu : comment la manette sait-elle où se connecter ? C'est là qu'intervient la connexion USB et le logiciel *sixpair* : en effet, la connexion USB sert à recharger la batterie, mais également à enregistrer l'adresse MAC de la console dans la mémoire de la manette afin

qu'elle opère une connexion directe lors de l'appui du bouton PS. Une fois ceci fait, la connexion s'opère automatiquement et il reste à lancer le serveur HID hidd patché pour que la manette se connecte :

```
hidd --server -nocheck -d
```

Ensuite, un dump des données entrantes et sortantes en conservant uniquement les données HID ressemble à ceci (à l'aide de `hcidump -x`) :

```
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8E 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
```

Voici une étude détaillée d'un log de type `hcidump` (`hcidump -x`) qui nous fournit les données décodées des trames HCI transitant entre le système et le dongle Bluetooth. Cette connexion a été créée sur la Playstation 3, dotée d'un système Linux nous permettant de reproduire le comportement de la Sixaxis avec la console étant donné que le module Bluetooth est le même :

```
1970-01-01 01:00:00.1180091604 > HCI Event: Connect Request (0x04) plen 10
    bdaddr 00:19:C1:xx:xx:xx class 0x000508 type ACL
1970-01-01 01:00:00.1180091604 < HCI Command: Accept Connection Request (0x01|0x0009) plen 7
    bdaddr 00:19:C1:xx:xx:xx role 0x01
    Role: Slave
1970-01-01 01:00:00.1180091604 > HCI Event: Command Status (0x0f) plen 4
    Accept Connection Request (0x01|0x0009) status 0x00 ncmd 1
1970-01-01 01:00:00.1180091605 > HCI Event: Connect Complete (0x03) plen 11
    status 0x00 handle 43 bdaddr 00:19:C1:xx:xx:xx type ACL encrypt 0x00
```

Tout d'abord, une connexion est initialisée au niveau HCI par la manette. Aucun paramètre particulier n'est utilisé, le cryptage est désactivé et la manette est toujours *master* vu qu'elle a initialisé la connexion.

```
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 12
    L2CAP(s): Connect req: psm 17 scid 0x0040
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 16
    L2CAP(s): Connect rsp: dcid 0x0040 scid 0x0040 result 0 status 0
      Connection successful
```

Une fois la connexion ACL ouverte, une connexion L2CAP est immédiatement ouverte sur le PSM 17(0x11), à savoir le canal *HID-command*. Cela confirme donc nos suppositions comme quoi la manette est utilisée comme périphérique HID.

```
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 12
    L2CAP(s): Configuration req: dcid 0x0040 flags 0x00 clen 0
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 18
    L2CAP(s): Configuration rsp: scid 0x0040 flags 0x00 result 0 clen 4
      Success
      MTU 672
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 16
    L2CAP(s): Configuration req: dcid 0x0040 flags 0x00 clen 4
      MTU 64
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 14
    L2CAP(s): Configuration rsp: scid 0x0040 flags 0x00 result 0 clen 0
      Success
```

Après toutes connexions L2CAP, un *handshake* de configuration doit être effectué. Il se déroule en deux parties, chacune composée d'une *configuration request* et d'une *configuration response* de part et autre de la connexion.

```
1970-01-01 01:00:00.1180091605 < HCI Command: Write Link Policy Settings (0x02|0x000d) plen 4
    handle 43 policy 0x0f
    Link policy: RSWITCH HOLD SNIFF PARK
1970-01-01 01:00:00.1180091605 > HCI Event: Command Complete (0x0e) plen 6
    Write Link Policy Settings (0x02|0x000d) ncmd 1
    status 0x00 handle 43
```

Nous avons à ce moment une intervention de la part de l'hôte, qui exige un changement de rôle pour redevenir un *master*. C'est une opération tout à fait normale et justifiée qui se déroule lors de chaque connexion de la part d'un périphérique, car l'hôte ne veut/peut pas rester *slave* sinon il ne pourrait plus supporter plusieurs connexions.

```
1970-01-01 01:00:00.1180091605 > HCI Event: Connection Packet Type Changed (0x1d) plen 5
    status 0x00 handle 43 ptype 0x3318
    Packet type: DM1 DH1 2-DH3 2-DH5 3-DH3 3-DH5
```

Ici se déroule un événement qui peut paraître insignifiant : la manette demande un changement de type de paquets de transport. C'est une opération tout à fait normale, mais qui a son importance, car nous verrons plus loin que si l'hôte ne supporte pas certains paquets liés à la version 2.0 du protocole Bluetooth, la manette n'envoie que la moitié des informations par la suite.

```
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 12
    L2CAP(s): Connect req: psm 19 scid 0x0041
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 16
    L2CAP(s): Connect rsp: dcid 0x0041 scid 0x0041 result 0 status 0
      Connection successful
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 36
    L2CAP(s): Configuration req: dcid 0x0041 flags 0x00 clen 24
      QoS 0x02 (Guaranteed)
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 18
    L2CAP(s): Configuration rsp: scid 0x0041 flags 0x00 result 0 clen 4
      Success
      MTU 672
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 16
    L2CAP(s): Configuration req: dcid 0x0041 flags 0x00 clen 4
      MTU 64
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 14
    L2CAP(s): Configuration rsp: scid 0x0041 flags 0x00 result 0 clen 0
      Success
```

Après des requêtes de noms et de *features* que nous sautons ici, une connexion L2CAP sur le psm 19, soit le HID-interrupt est demandée par la manette, suivi d'un *handshake* de configuration.

```
1970-01-01 01:00:00.1180091605 < ACL data: handle 43 flags 0x02 dlen 10
    L2CAP(d): cid 0x0040 len 6 [psm 17]
      HIDP: Set report: Feature report
      F4 42 03 00 00
```

Enfin, voici la commande essentielle à l'activation de la manette : un *Set Report HID* de type *Feature* avec comme informations : F4 42 03 00 00. Sans cette commande, la manette se déconnecte d'elle-même après quelques secondes.

```
1970-01-01 01:00:00.1180091605 > HCI Event: Number of Completed Packets (0x13) plen 5
    handle 43 packets 1
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 5
    L2CAP(d): cid 0x0040 len 1 [psm 17]
```

```
      HIDP: Handshake: Successful
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8E 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
```

La manette a accepté la commande et commence l'envoi de donnée HID de manière continue :

```
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8E 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8E 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8D 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 07 01 EA 01 9B 00 02
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 85 82 8D 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8D 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
1970-01-01 01:00:00.1180091605 > ACL data: handle 43 flags 0x02 dlen 54
    L2CAP(d): cid 0x0041 len 50 [psm 19]
      HIDP: Data: Input report
      01 00 00 00 00 00 84 82 8D 7C 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
      81 02 08 01 EB 01 9B 00 02
...
```

Les données sont envoyées sous la forme de *Input Report* et ont une longueur de 54. Le décodage complet de ces paquets est effectué dans la section "Étude de la Sixaxis". Maintenant que nous avons plus d'informations et d'outils à disposition, nous pouvons étudier le portage des outils Linux sous Windows afin de pouvoir connecter la manette par Bluetooth sous Windows.

### Portage des outils Linux sous Windows

Nous avons adapté l'utilitaire *sixpair* pour une machine tournant sous Windows afin de fournir à tous les utilisateurs un outil permettant d'appairer une manette Sixaxis avec un module Bluetooth quelconque. Malheureusement, la *stack* Bluetooth officiel Bluez ne peut être adaptée aussi aisément que cet outil étant donné que les drivers Bluetooth fournis sous Windows sont fermés et non documentés. Il faut donc se contenter des *stack*s déjà existantes et essayer d'ouvrir une connexion à l'aide de la manette tout en analysant le contenu des connexions en cas de problème pour en comprendre le fonctionnement au besoin, comme nous allons l'étudier dans le chapitre suivant.

# IMPLÉMENTATION DE BLUETOOTH

## Introduction

Le but de ce chapitre est de fournir un résumé des diverses fonctionnalités des *stack*s Bluetooth tournant sous Windows et Linux pour le grand public, c'est-à-dire les *stack*s fonctionnant avec des dongles Bluetooth standards du commerce et non pas des dongles Bluetooth de développement comme celui dont nous disposons sur Yamor et que nous étudierons plus tard.
En effet, l'un des buts de ce travail est d'étudier la possibilité de concevoir un logiciel à l'aide des kits de développement fournis par les concepteurs de *stack* Windows afin d'y brancher la manette. Vu que les SDK fournis pour Windows ont tous une approche et une utilisation différente, plusieurs semaines ont été nécessaires à toutes les tester afin d'être certain de n'avoir omis aucune possibilité. Le lecteur peut légitimement se demander pourquoi nous avons analysé *toutes* les *stack*s Bluetooth commerciales sous Windows. La raison est simple : après l'essai d'une ou deux *stack*s, nous avons remarqué que le codage d'une interface de connexion Sixaxis sous Windows est très limitée puisque les kits de développement standard ne fournissent que très peu d'interfaces pour le développeur et qu'il y a une probabilité que cette conexion ne soit pas possible avec des dongles Bluetooth standards. D'où l'extension de notre étude à la majorité des *stack*s existantes.
Nous terminerons ce chapitre par une brève description des outils disponibles sous Linux et de la possibilité de s'en inspirer pour concevoir un drivers Bluetooth unique et générique sous Windows.

## Widcomm

### Introduction

Au début de ce travail, le premier choix de la *stack* à utiliser était d'une grande importance : nous étions conscients que le travail que l'on souhaitait effectuer était potentiellement impossible à faire sur certaines *stack*s et/ou avec plus ou moins de difficultés. Il est utile de remarquer que la compréhension du protocole Bluetooth et du fonctionnement de la manette a évolué tout au long du projet et que les connaissances disponibles au début étaient très maigres et ne permettaient de faire qu'un choix approximatif. Toutefois, le SDK Widcomm est désormais gratuit (il coûtait autrefois 1'000$), fourni avec une documentation structurée (un fichier .pdf d'une centaine de pages), beaucoup d'exemples et un template pour les outils de développement les plus courants. De plus, il donne accès à la couche L2CAP, ce qui est relativement bas et donc bon signe pour pouvoir coder ses propres commandes. Toutes ces raisons ont poussé le choix à tester celle-ci en premier.

### Spécifications du SDK[38]

Actuellement à la version 6.1.0.1501, il est compatible avec Windows 98SE, Windows ME, Windows 2000 et XP. Il fonctionne également sous Windows Vista mais au-dessus de la *stack* Microsoft, ce qui limite les fonctionnalités puisque, comme nous le verrons plus loin, la *stack* Microsoft est plus limitée que la Widcomm au niveau de la liberté de manoeuvre.
Le SDK fourni des interfaces pour la couche L2CAP, SDP et RFCOMM ainsi que des profils pour l'impression, l'audio, Lap, Dun, FTP, Spp et Obex. Du côté de la *stack*, tous les profils standards Bluetooth sont supportés, dont le HID et l'audio haute qualité.
Le langage utilisé est le C++ et les librairies sont fournies pour Visual C++, Visual .NET et Visual Studio.

### Démarches de tests

Avant de se lancer dans l'élaboration d'un logiciel de connexion, il convient de tester la *stack* "nue". Étant donné le support du profile HID, la connexion devrait se dérouler sans problème et pourtant, bien que la manette soit visible par le périphérique (vu que l'on a configuré au préalable l'adresse MAC adéquate), la connexion se termine après quelques secondes sans avoir installé le moindre driver.
L'analyse d'un log de connexion nous permettra d'éclaircir un peu le problème (cf. annexes A, premier log). On remarque que le profile HID est bien chargé, mais le SET REPORT n'étant pas envoyé, la manette se déconnecte après quelques secondes. La meilleure supposition à ce sujet est que la *stack* cherche les informations de connexions dans le SDP de la manette, mais celui-ci est absent. Elle n'a donc pas à sa disposition les informations nécessaires et tente simplement de connecter le périphérique en *Boot Mode,* ce qui ne suffit pas à la manette pour s'activer et cela entraîne une déconnexion. Dans le cadre de cette *stack*, il va donc falloir intervenir soit au niveau HID, soit au niveau L2CAP pour ouvrir une connexion manuellement.
Le niveau HID n'étant pas accessible avec le SDK, la seule possiblité qui s'offre à nous est la gestion d'une connexion L2CAP, nous allons donc effectuer un *discovery* pour tester la visibilité de la manette, suivi de l'ouverture d'une connexion L2CAP et l'attente de la connexion.
Au niveau des classes, nous disposons d'une classe de base CBtIf qui permet de gérer les *inquiry,* les *discovery* ainsi que les habituelles fonctions de connexions disponibles en LM et également quelques fonctions de connexions audio. Une fonction `StartDiscovery()` permet de lancer la découverte de périphérique. La fonction virtuelle `OnDiscoveryComplete()` (donc à redéfinir) est appelée lorsque la découverte est complète. Cela permet d'être certain d'avoir découvert un maximum de périphérique en laissant tourner le processus durant le temps adéquat. Pour lire les résultats, plusieurs fonctions sont à notre disposition: soit `GetLastDiscoveryResult()`, soit `ReadDiscoveryRecords()` ou encore `OnDeviceResponded()`, qui permet d'obtenir des informations comme le nom, l'adresse MAC ou les services supportés par le périphérique découvert.

[38] http://www.broadcom.com/products/bluetooth_sdk.php

En ce qui nous concerne, par exemple :

```
void OnDeviceResponded(BD_ADDR bda, DEV_CLASS devClass, BD_NAME bdName, BOOL bConnected) {
        printf("Trouve \n");
        printf("BD_NAME : %s\n",bdName);
        printf("Adresse : 0x%x%x%x%x%x%x\n",bda[0],bda[1],bda[2],bda[3],bda[4],bda[5]);

        if(!strcmp((char*)bdName,"PLAYSTATION(R)3 Controller")) {

                printf("Joy ps3 detecte\n");
                joy_discovery = true;
                for(int i=0;i<6;i++) {
                        joy_add[i]=bda[i];
                }
        }
```

Cela permet de vérifier si la manette Sixaxis a été détectée (et donc correctement configurée pour se connecter sur notre dongle Bluetooth) et d'en enregistrer l'adresse MAC afin d'accepter la connexion L2CAP plus loin. Le résultat est concluant : la Sixaxis est reconnue et son adresse MAC retenue.

En ce qui concerne les connexions L2CAP, nous avons deux classes à disposition : CL2CapIf et CL2CapConn. La classe CL2CapIf sert à assigner un psm (`AssignPsmvalue()`) sur la connexion que l'on souhaite créer ou écouter. On doit également l'enregistrer à l'aide de la fonction `Register()` et fixer un niveau de sécurité (`SetSecurityLevel()`). Une fois tout ceci effectué (chaque opération est obligatoire), la classe CL2CapConn permet d'effectuer toutes les opérations possibles au niveau L2CAP: écoute, acceptation de connexion, rejet, connexion, configuration, déconnexion, écriture, changement de rôle et également les opérations pour les connexions audio.

Il suffit de décider si nous serons un serveur ou un client, c'est–à–dire si l'on va initier la connexion ou l'attendre : dans le cas de la Sixaxis, nous sommes un serveur. Pour la Wiimote, il faudrait au contraire être un client.

Il ne reste donc plus qu'à appeler la fonction Listen() et à attendre que la fonction virtuelle (redéfinie) `OnIncomingConnection()` soit déclenchée et d'y appeler la fonction d'acceptation pour ensuite utiliser les fonction `OnConnected()` et `OnDataReceived()` pour communiquer avec la manette :

```
void OnIncomingConnection() {
        printf("Incoming\n");
        incoming = true;
        printf("Accepting : %d \n", this->Accept());
}
```

Malgré cette beauté théorique, la fonction `OnIncomingConnection` n'est jamais déclenchée, malgré la réussite de l'appel de toutes les autres fonctions. En vérifiant le log, on remarque qu'il est rigoureusement identique à celui que nous avons sans notre serveur. Après de très nombreuses vérifications, spécialement au niveau du service à enregistrer lors de l'attribution du PSM (à savoir : CBtIf::guid_SERVCLASS_HUMAN_INTERFACE;) pour trouver les erreurs, un client a été codé en Python (via Pybluez dont nous parlerons plus tard) sous Linux pour sa rapidité de mise en oeuvre afin de vérifier si le serveur était correctement lancé. Ce script sert de client qui se connecte à notre serveur avec comme mission de simuler une connexion L2CAP provenant de la Sixaxis.

Concrètement, nous utilisons ce script pour initialiser une connexion L2CAP en direction de notre serveur. Sous Pybluez, nous pouvons configurer le port du *socket*, c'est-à-dire le PSM. En utilisant un PSM quelconque supérieur à 1000, la connexion est déclenchée et acceptée. En revanche, si on utilise le PSM 17 ou 19, rien ne se passe, exactement comme auparavant. La réponse se trouve à la page 39 du SDK Programmer's Guide : "*Values between 3 and 0x1000 should be used with care to avoid conflicts with other reserved PSM values*". Et en effet, que le serveur soit actif ou pas, on peut remarquer sur les logs de connexion que les connexions HID sont tout de même acceptées et gérées : cela signifie donc que le profil HID est réservé par la *stack* pour sa propre gestion, avec impossibilité de la désactiver. Rappelons les différents niveaux où l'on peut intervenir pour connecter la Sixaxis :

- HID : géré par la *stack* et impossible de lui prendre la main
- L2CAP : accessible, mais le service HID étant réservé, on ne peut gérer des connexions à ce niveau
- HCI : indisponible par la *stack*

### Conclusion

La *stack* fournie par Widcomm est très complète : elle supporte la majorité des profils Bluetooth existants et son SDK est clair, bien documenté et fourni avec beaucoup d'exemples. Néanmoins, malgré son apparente universalité, il est très restreint d'un certain point de vue, car il n'autorise que l'utilisation des profils pour lesquels il fournit des interfaces, à savoir l'impression, l'audio, Lap, Dun, FTP, Spp et Obex. L'accès au L2CAP est autorisé, mais uniquement pour des applications personnelles: impossible de gérer un profil autremment qu'en laissant la *stack* s'en charger. Cela limite grandement l'avantage d'avoir accès au L2CAP si cela ne permet que d'avoir accès à des logiciels personnels qui communiquent uniquement entre eux. D'un côté c'est compréhensible pour ne pas faire de conflits avec les profils déjà existants, mais d'un autre cela limite la marge de manoeuvre du codeur qui aurait souhaité écrire, par exemple, un driver maison pour un clavier Bluetooth récalcitrant dont on aurait perdu le dongle propriétaire.

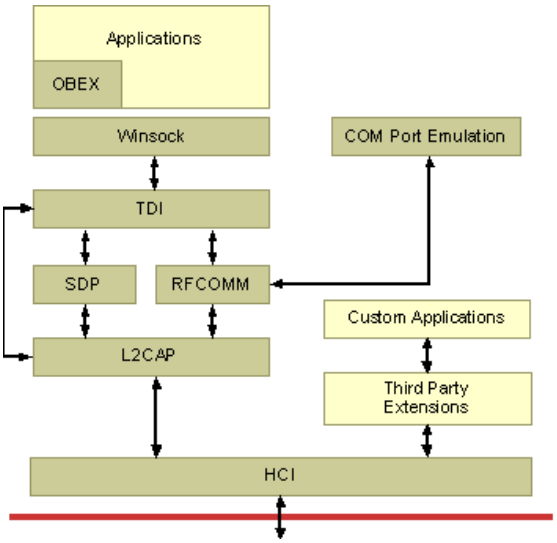
# Microsoft

## Introduction

La gestion du Bluetooth est relativement récente chez Microsoft en ce qui concerne les ordinateurs étant donné qu'elle n'est apparue qu'avec Windows XP Service Pack 2. Toutefois, le Bluetooth est également géré pour les appareils portables comme Windows CE et Windows Mobile, systèmes qui disposent d'une interface Bluetooth très différente de celle fournie avec Windows XP et/ou Windows Vista. En effet, les versions mobiles sont beaucoup plus complètes.

## Spécifications

Le développement Bluetooth à l'aide des *stack*s Microsoft peut se faire de diverses manières : Sous Windows XP/Vista à l'aide de fonctions Bluetooth standards de connexion[39], sous Windows XP/Vista à l'aide de *sockets*[40], sous Windows Vista via le DDK (*drivers development kit*)[41] et enfin sous Windows *Embedeed* à l'aide du Windows *Embedded Source Tools for Bluetooth Technology*[42]. Toutes ces versions ont leurs fonctionnalités propres. Sous Windows CE, voici les couches disponibles de la *stack* par l'API de développement[43] :



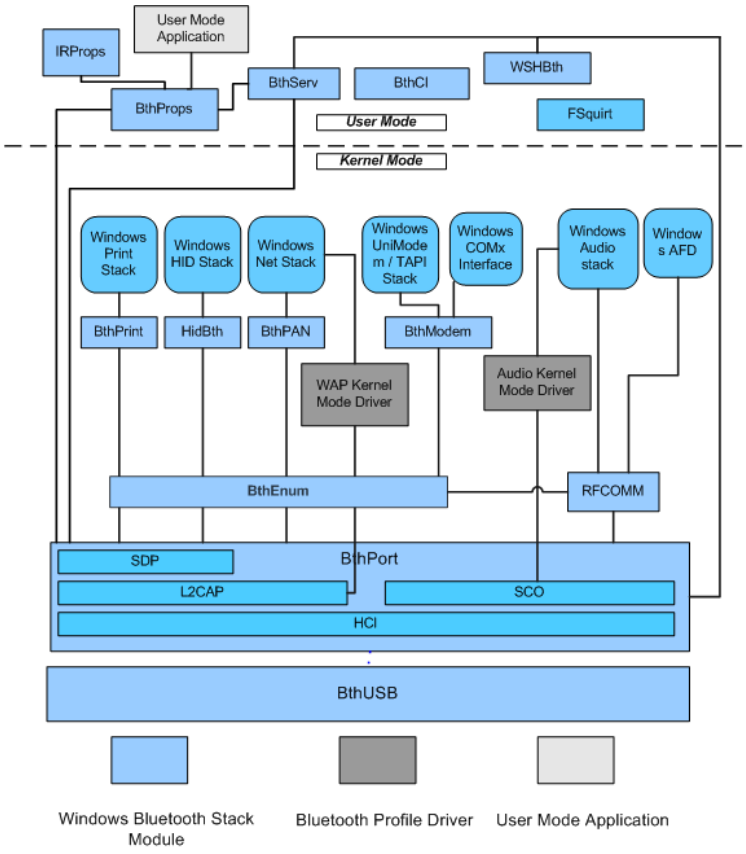

(SOURCE : HTTP://MSDN2.MICROSOFT.COM)

Les parties brunes sont fournies par la *stack* et utilisables par le programmeur. On remarque que tous les niveaux sont accessibles : HCI, L2CAP et le tout via Winsock au besoin. De plus, un Wiki officiel a été conçu ainsi qu'une série d'outils facilitant le développement[44]. Tout ceci est parfait, mais nous n'en

[39] http://msdn2.microsoft.com/en-us/library/aa362942(VS.85).aspx
[40] http://msdn2.microsoft.com/en-us/library/aa363059(VS.85).aspx
[41] http://msdn2.microsoft.com/en-us/library/bb870474.aspx
[42] http://msdn2.microsoft.com/en-us/embedded/aa714533.aspx
[43] http://msdn2.microsoft.com/en-us/library/ms863323.aspx
[44] http://channel9.msdn.com/wiki/default.aspx/Channel9.BluetoothDevelopment

avons pas besoin car la Sixaxis aurait une utilité très limitée sur un appareil portable et doté de Windows CE.

Sous Windows XP comme indiqué précédemment, le développement est possible via des interfaces Bluetooth standards ou via des *sockets* (Winsock). Voici le graphique définissant la *stack* Bluetooth complète sous Windows XP :

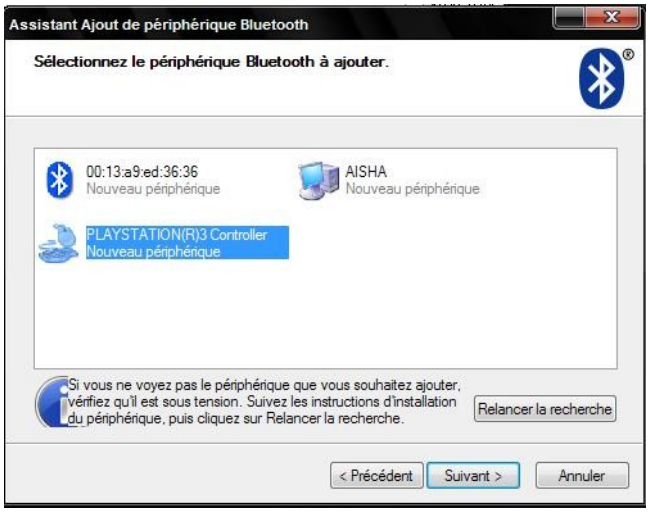



(SOURCE : HTTP://MSDN2.MICROSOFT.COM)

Via les interfaces Bluetooth normales et les *sockets*, nous ne pouvons accéder qu'au *User Mode Application*. En ce qui concerne le Kernel Mode, il faut se tourner vers les drivers et les outils fournis pour les développer comme nous le verrons plus tard.

### Connexion avec la *stack* seule

Avant de se lancer dans la conception et le codage d'un serveur, il semble évident de tester une connexion uniquement avec la *stack* pour étudier son comportement et voir si elle peut gérer seule la connexion.




Après configuration de la manette pour se connecter à notre périphérique, la *stack* Microsoft détecte la manette ainsi que son nom, comme ci-dessus.

Une fois la Sixaxis découverte, le résultat d'une connexion est assez surprenant : Windows installe un drivers de périphérique HID générique et indique la manette comme "connectée" alors qu'elle est éteinte.


Inutile d'ajouter que la Sixaxis est totalement inutilisable, malgré des pressions répétitives sur le bouton PS durant la connexion. En étudiant un *log* de cette connexion, on remarque que la *stack* ne fait rien de spécial : elle accepte la connexion de la Sixaxis *uniquement* quand on lui demande de s'y connecter (il faut donc être synchronisé), gère les configurations L2CAP et s'arrête là. Voyons donc ce que nous propose le SDK.

### Via les interfaces Bluetooth

Les premiers essais ont été faits via les interfaces "normales" fournies par Microsoft. Pour l'utiliser, l'installation du Platform SDK for Windows[45] est nécessaire, le SDK étant fourni avec Visual Studio étant incomplet. En ce qui concerne les fichiers d'en-tête, ws2bth.h Bthsdpdef.h et BluetoothAPIs.h sont indispensables.

En ce qui concerne le langage, il s'agit de C, donc aucun objet mais des structures sont utilisées. Comme auparavant, nous commencerons par effectuer un *discovery* pour détecter la Sixaxis, son adresse MAC et vérifier qu'elle est correctement configurée.

[45] http://www.microsoft.com/msdownload/platformsdk/sdkupdate/XPSP2FULLInstall.htm

Pour cela, une structure WSA (*Web Services Architecture*) est nécessaire pour "feuilleter" l'interface Bluetooth et les divers périphériques découverts (code complet en annexe) :

```
WSADATA wsaData;
WSAStartup(MAKEWORD(iReqWinsockVer,0),&wsaData)
```

En plus de cela, il nous faut une structure pour récupérer les résultats du *discovery* :

```
PWSAQUERYSET            pWSAQuerySet = NULL;
pWSAQuerySet->dwNameSpace = NS_BTH;
pWSAQuerySet->dwSize = ulPQSSize;
```

On remarque que c'est dans le dwNameSpace qu'on indique que nous souhaitons utiliser le Bluetooth et non pas une autre interface réseaux. En effet, le *discovery* via WSADATA est identique à une énumération de périphériques réseaux. On précise ensuite les informations que l'on veut récupérer :

```
//
//On ne veut que les devices, pas les services
//
 ulFlags = LUP_CONTAINERS;
 //
 // Que le nom
 //
 ulFlags |= LUP_RETURN_NAME;
 //
 // Et l'adresse
 //
 ulFlags |= LUP_RETURN_ADDR;
//
// Et le COD (class of device)
//
ulFlags |= LUP_RETURN_TYPE;
```

Enfin, on lance le *discovery* proprement dit avec tous nos paramètres :

```
WSALookupServiceBegin(pWSAQuerySet, ulFlags, &hLookup)
```

Et on itère sur les différents périphériques découverts :

```
WSALookupServiceNext(hLookup, ulFlags, &ulPQSSize, pWSAQuerySet)
```

En principe, la première itération est un échec car la mémoire allouée à ulPQSSize est insuffisante. Il convient donc d'intercepter l'erreur et de réallouer la mémoire nécessaire, retournée dans le même paramètre. Les données sont ensuite disponibles dans pWSAQuerySet et ses différents membres (lpszServiceInstanceName par exemple).

Une fois le *discovery* réussi, il est obligatoire de passer à une interface *socket* étant donné que l'interface de base ne fournit aucune méthode pour gérer les connexions (elle ne gère que la partie LM).

### Winsock

L'avantage des *sockets* pour le programmeur est qu'ils sont gérés de manière identiques quels que soit le protocole ou le système d'exploitation utilisé, à quelques détails près. La première chose à faire est de créer la *socket* avec les bons paramètres afin d'accéder aux services souhaités :

```
LocalSocket = socket(AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM);
```

Le problème est que nous souhaitons un *socket* L2CAP, pas RFCOMM. Néanmoins, il n'est jamais mentionné dans la documentation d'autres *sockets* que RFCOMM.
En revanche, il est indiqué[46] : *To create a socket using Bluetooth, use the following settings:*

- *The af parameter of the socket function is always set to AF_BTH for Bluetooth sockets.*
- *The type parameter of the socket function is always SOCK_STREAM; SOCK_DGRAM sockets are not supported by Bluetooth.*
- *For the protocol parameter, BTHPROTO_RFCOMM is the supported protocol.*

Il semblerait donc qu'il ne soit pas possible d'ouvrir des *sockets* RFCOMM et il est bien clair qu'on ne peut accéder au L2CAP via une connexion RFCOMM, car parfois on pourrait confondre les ports ouverts sur une machine par le protocole RFCOMM comme étant des ports de communication directe avec le dongle : il n'en est rien, ces ports servent uniquement à communiquer avec d'autres périphériques se connectant en RFCOMM. Même en tentant d'utiliser des *sockets* en BTHPROTO_L2CAP :

```
socket( AF_BTH, SOCK_DGRAM, BTHPROTO_L2CAP )
```

, nous avons un message d'erreur qui indique que le protocole n'est pas supporté. Après quelques recherches, il semble que ce problème soit assez récurrent et connu pour la *stack* Bluetooth Microsoft mais qu'il n'y ait aucun moyen de passer outre[47]. Il est toutefois surprenant qu'il n'existe quasiment aucune documentation à ce sujet, bien que certaines rumeurs indiquent qu'il serait possible d'accéder au L2CAP mais uniquement sous Windows Vista[48]. Analysons cette nouvelle voie de recherches.

[46] http://msdn2.microsoft.com/en-us/library/aa362910(VS.85).aspx
[47] http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=2227688&SiteID=1
[48] Mark S., *USB/1394 on the PC*, WinHEC presentation, 2005

### Via le DDK et Windows Vista[49]

Le fait que le L2CAP ne soit disponible que sous Windows Vista rend le développement d'une application autorisant la connexion de la Sixaxis beaucoup moins attractive. En effet, cela indiquerait déjà que la connexion est impossible sous Windows XP, qui reste encore aujourd'hui majoritairement installé et utilisé pour le développement (du moins par rapport à Vista). Néanmoins, aucune piste ne doit être négligée. Sur MSDN (Microsoft Developer Network), dans la section *Windows Driver Kit/Device and Driver Technologies/Bluetooth* se trouvent des informations sur l'utilisation de la *stack* Bluetooth via le DDK. La documentation fait référence à des *Profile Drivers* mais sans indiquer comment en coder. De plus il est clairement précisé que "*IHVs[50] must use Windows Vista to develop their profile drivers because earlier versions of Windows, including Windows XP with SP2, do not currently support profile driver development*"[51]. Il est donc clair que le développement ne peut s'effectuer que sous Windows Vista. Le développement ne peut en revanche pas se dérouler sous Visual Studio mais directement en ligne de commande via le WDK, disponible sur http://connect.microsoft.com/ (sur inscription). Il faut ensuite se rendre sous "My participation" et sélectionner le WDK. Attention, il est indispensable de prendre la version beta pour Windows Server 2008, sinon le WDK ne sera pas fourni avec un environnement de développement pour Windows Vista. Une fois le WDK installé, la référence Bluetooth fournit une liste de fonctions disponibles, dont des fonctions pour ouvrir des connexions L2CAP. Toutefois, malgré toutes ces fonctions à disposition, aucune information ni exemple n'est fourni pour expliquer une quelconque implémentation. En effet, il s'agit d'un kit de développement de drivers : cela signifie que l'on développe un drivers lié à du matériel et non pas un simple logiciel. De plus, le développement de Profile Driver sous Windows Vista est récent : la preuve en est que sa gestion a été ajoutée à la *stack* Widcomm en novembre 2007. Cela signifie non seulement que cette technologie est fraîche, mais surtout que la documentation pour le grand public est encore lacunaire et incomplète. Aucun exemple n'est disponible et chaque semaine, quelques lignes ou pages sont ajoutées au MSDN. Après quelques semaines de tests et de développement, il semble clair que cette technologie est trop récente pour être utilisée sans un investissement en temps énorme. De plus, la priorité est de fournir des interfaces de développement fonctionnelles sur n'importe quelle plateforme, pas uniquement Windows (YAMOR par exemple), c'est pourquoi cette partie a été mise en attente pour se concentrer sur les autres *stack*s disponibles et le travail restant.

---

[49] Vatsal B., *Bluetooth Advance In Windows Vista and Beyond*, WinHEC presentation, 2006

[50] IVH se réfère ici à un concepteur et vendeur de matériel informatique

[51] http://msdn2.microsoft.com/en-us/library/aa938557.aspx

## Conclusion

La *stack* Microsoft n'est clairement pas la plus fournie et la plus complète à utiliser. Microsoft est arrivé tardivement dans le monde du Bluetooth et cela se répercute sur son développement et sa *stack*. L'obligation de ne coder qu'en RFCOMM en est une preuve claire : en RFCOMM, la seule possibilité est de communiquer avec nos propres applications ou avec un téléphone portable, ce qui est bien faible comparativement à toutes les possibilités qu'offre le Bluetooth. Malgré l'apparente complexité du schéma de sa *stack*, très peu de modules sont disponibles pour le développeur non professionnel. Aucune possibilité d'accéder au minidriver HID, pas moyen d'accéder au L2CAP alors que les prototypes des *sockets* sont disponibles dans les fichiers en–tête. Avec l'arrivée de Windows Vista on pouvait espérer une amélioration, mais à première vue Microsoft en a profité pour rendre obligatoire le passage par sa *stack*, ce qui oblige les développeurs de *stack* Bluetooh à s'adapter à celle de Microsoft, ce qui les empêche de fournir plus de fonctionnalités. L'accès hypothétique à la couche L2CAP en mode driver n'est pour le moment qu'au stade de théorie et une interface de codage accessible au public n'est clairement pas encore sur le marché du développement Bluetooth. Ainsi, la *stack* Microsoft a beaucoup de potentiel, mais qui va prendre encore du temps à se développer.

# IVT Bluesoleil

## Introduction

Bluesoleil est le principal concurrent de Widcomm. Plus complet que la *stack* Microsoft et plus simple d'emploi, elle a été la première *stack* à supporter complètement la connexion de la Wiimote sous Windows et à juste titre : les concepteurs de cette *stack* ont inclu la gestion de ce périphérique lors des mises à jour de leur logiciel. Tout comme la *stack* Widcomm, celle-ci n'est fournie que lors de l'achat d'un dongle Bluetooth ou elle peut être achetée seule. Contrairement à la *stack* Widcomm en revanche, on peut en installer une version de démo qui permet de transférer jusqu'à 5mo de données par connexion avant de devoir réinitialiser la connexion, ce qui permet d'en avoir une réelle utilisation en version d'évaluation.


L'INTERFACE DE BLUESOLEIL

IVT Corporation fournit également des utilitaires de développement pour les développeurs hardware en Bluetooth comme la *stack* Bluelet[52], réservée aux professionnels :

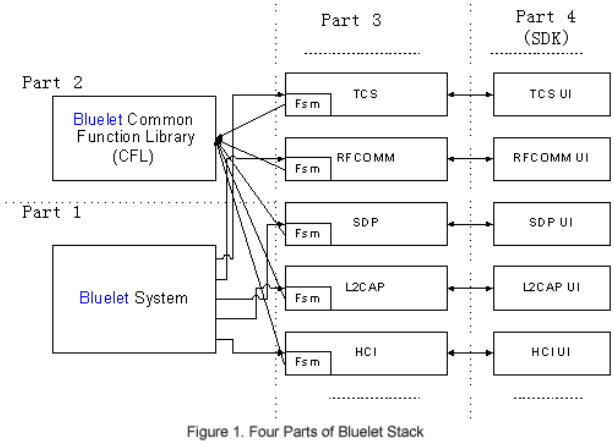



Figure 1. Four Parts of Bluelet Stack

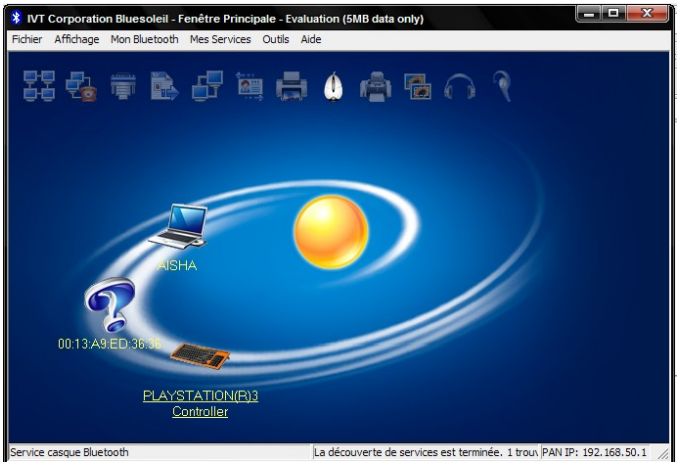
(SOURCE : HTTP://WWW.IVT.COM)

[52] http://www.ivtcorporation.com/products/stack/index.php

Néanmoins, nous devrons nous contenter de l'édition gratuite et standard du SDK : API 0.83 for BlueSoleil 2.0.0.0.0 Release.[53]

### *Stack* seule

Comme pour les *stack*s précédentes, il serait dommage de réinventer la roue si la Sixaxis peut se connecter directement sur la *stack* seule. L'essai a donc été tenté de nombreuses fois et voici le résultat:




Le périphérique est bien découvrable, ainsi que son nom et sa classe (périphérique de type HID, représenté par un clavier). Malgré cela, une connexion directe échoue et le log (disponible en annexe) nous apprend que la *stack* Bluesoleil ne fait rien de plus que d'accepter les connexions L2CAP, ne reconnaissant pas le périphérique et ne sachant pas comment le gérer.

### Étude du SDK

Le SDK est fourni avec une documentation sommaire (35 pages) regroupant les principales fonctions disponibles pour le programmeur, en revanche, il n'est nulle part fait mention des fonctionnalités complètes disponibles pour le SDK.

Si l'on commence par la traditionnelle tentative de connexion de la Sixaxis uniquement avec la *stack*, elle se termine par un échec de connexion. Malheureusement, lorsqu'on consulte la documentation du SDK on se rend compte que les choix de connexions sont quasiment inexistants :

*Typical API Calling Sequence:*

1. `BT_InitializeLibrary`
2. `BT_RegisterCallback`
3. `BT_InquireDevices`
4. `BT_PairDevice`
5. `BT_BrowseServices`
6. `BT_ConnectService`

[53] http://www.bluesoleil.com/download/index.asp?topic=bluesoleil_sdk#

7. `BT_DisconnectService`
8. `BT_UninitializeLibrary`

Rien n'indique une quelconque possibilité d'envoi de données, de connexion HCI, L2CAP ou RFCOMM : les fonctions se limitent au LM et à une fonction de connexion simple. En résumé, le SDK de Bluesoleil est une méthode pour utiliser la *stack* graphique en programmation. Cela n'est pas étonnant, étant donné que le SDK est indiqué pour la version 2.0 de Bluesoleil alors que la dernière version de Bluesoleil est la 5.0. et que le SDK pour cette version est payant.

### Conclusion sur les *stack*s Windows commerciales

Les trois *stack*s Bluetooth Windows disposant d'un SDK ont donc toutes été testées. La première chose à remarquer est que les kits de développement sont totalement indépendants des fonctionnalités fournies par la *stack*. Widcomm et Bluesoleil fournissent un grand nombre de profils avec leur *stack*, en revanche on ne peut quasiment pas y accéder via le SDK, et en ce qui concerne Bluesoleil, aucune opération supplémentaire ne peut être effectuée par rapport à la *stack*. D'autres *stack*s commerciales existent, comme la Toshiba pour ne citer qu'elle, mais aucune autre ne fournit de SDK, du moins disponibles au public ou pour l'éducation. Il faut toutefois remarquer qu'il existe des SDK Bluetooth professionnel, fournissant des interfaces pour utiliser les plus basses couches du protocole comme le HCI et L2CAP comme le SDK de MecelAp[54]. De plus, il n'est pas impossible que les concepteurs de Bluesoleil ou Widcomm ajoutent prochainement la gestion de la Sixaxis à leur *stack*, étant donné que nous les avons contactés par mail pour leur indiquer la méthode afin de rendre leur *stack* compatible. En plus de ces possibilités, nous avons encore les *stacks* Bluetooth Java à notre disposition, que nous nous proposons d'étudier afin de compléter cette étude.

[54] http://www.mecel.se

## Java : Electricblue, Harald, JavaBluetooth et Com one

### Introduction

De manière assez surprenante, les *stack*s Java sont assez nombreuses, dont de nombreux kits de développement. Javablutooth.com en répertorie un certain nombre avec leurs particularités :

| Nom | Javax.bluetooth support | Javax.comm support | Java Platforms | OS | Prix |
|---|---|---|---|---|---|
| Blipnet | Non | Non | J2SE | Win-32, Linux | 1200$ |
| JB-20 | Oui | Oui | J2SE | Win-32 | 150$ |
| Possio Px 30 | Oui | Non | J2ME | Win-32, Linx | 900$ |
| Atinav | Oui | Oui | J2ME, J2SE | Win-32, Linux, Pocket PC | Non-indiqué |
| Avetana | Oui | Oui | J2SE | Win-32, MAC OS X, Linux | 25$ |
| Blue Cove | Oui | Oui | J2SE | Win-32 | Gratuit (libre) |
| Electric Blue | Oui | Oui | J2SE | WinXP SP2 | 15$ |
| Harald | Non | Non | Javax.comm nécessaire | Plusieurs | Gratuit |
| JavaBluetooth.org | Oui | Non | Javax.comm nécessaire | Plusieurs | Gratuit |
| Rococo | Oui | Oui | J2ME, J2SE | Linux, Palm OX | 2500 $ |

Les trois premières se démarquent par leur prix car il s'agit de kits de développement complets, comportant également du matériel de développement Bluetooth. Étudions en détail les diverses propriétés de ces *stack*s.

### Javax.bluetooth support et JSR-82[55]

JSR-82 (java specification request n° 82) n'est rien d'autre que la librairie java officielle pour le Bluetooth, créée et standardisée par la communauté java en 2002. Cette librairie a été spécialement conçue pour les téléphones portables et les appareils dotés des caractéristiques suivantes :

- 512 KO de mémoire disponible
- Connexion bluetooth
- Conforme aux spécifications JSR-000030 J2ME Connected, Limited Device Configuration[56]

L'inconvénient de cette librairie est qu'elle est conçue spécialement pour les appareils portables et surtout qu'elle est ancienne. Ceci pose spécifiquement problème, car certains petits PSM (< 100) ont été bloqués pour des usages futurs non implémentés, ce qui pose certains problèmes comme le fait que le HID n'est pas supporté, car ces PSM sont réservés pour un usage futur. Cela nous permet immédiatement d'éviter les *stack*s Java supportant le JSR-82 car malgré le support du SDP, du RFCOMM et du L2CAP, tout comme avec la *stack* Widcomm il est impossible d'accéder aux ports nécessaires pour le HID.

### Javax.comm support

Le paquetage *comm* sous Java sert à gérer les communications par port série. Il n'est plus supporté sous Windows et son téléchargement n'est plus disponible officiellement. En effet, ce paquetage va à l'encontre de la mentalité Java qui consiste à être portable et malgré cela, le paquet *comm* fournissait (par exemple pour Windows) une librairie dynamique pour gérer les connexions. Toutefois, il est encore possible de disposer de ce paquetage sous Windows pour effectuer nos tests.

### Electricblue

Il n'était pas raisonnable de tester toutes les *stack*s Bluetooth Java, néanmoins il est tout à fait possible de ne choisir que celles qui sont potentiellement utilisables pour nos besoins. La *stack* Electriblue est donc notre premier choix :

- Elle n'exige pas javax.obex
- Elle est déclarée comme supportant les connexions L2CAP
- Elle est gratuite à l'essai

Toutefois, la documentation précise que le L2CAP n'est supporté que "au-dessus de la *stack* Widcomm". De plus, elle supporte les *stack*s Microsoft et Widcomm. Qu'est-ce que cela signifie ? Simplement que cette *stack* n'est qu'une abstraction fonctionnant au-dessus d'une *stack* préexistante.

[55] http://www.jcp.org/en/jsr/detail?ID=82

[56] http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html

Cette abstraction permet néanmoins toutefois de gérer les connexions Bluetooth via des *sockets* même avec la *stack* Widcomm.
Malgré quelques soupçons sur ce soi-disant accès au L2CAP, des tests ont été effectués sur une *stack* Microsoft et Widcomm. Les résultats sont prévisibles :
Au-dessus de la *stack* Microsoft, l'ouverture d'un *socket* de type L2CAP renvoie une erreur de "protocole non défini"
L'ouverture d'un *socket* au-dessus de la *stack* Widcomm est un succès, mais comme auparavant, la *stack* prend le dessus par rapport à notre programme pour gérer elle-même les connexions L2CAP
Electricblue est donc une aide au développement pour le Bluetooth, fournissant une interface conviviale et facile à gérer. Toutefois, elle n'apporte aucune fonctionnalité supplémentaire par rapport aux *stack*s sur lesquelle son fonctionnement se base.

### Harald, JavaBluetooth

Les *stack*s Harald et JavaBluetooth sont quasiment identiques. Elles annoncent toutes les deux qu'elles disposent d'un accès à la couche HCI, L2CAP et RFCOMM. Toutefois, javax.obex est nécessaire à leur bon fonctionnement et il convient d'ouvrir un port sur le dongle Bluetooth.
Après analyse des sources (entièrement disponibles), ces *stack*s fournissent une véritable interface dès le niveau HCI. Toutefois, un problème se pose dès lors qu'il faut ouvrir un port sur le dongle Bluetooth: effectivement, comme précisé plus haut, les ports installés lors de la connexion d'un dongle Bluetooth ne sont aucunement des ports de connexion directe avec le dongle : il s'agit simplement de ports fournis pour le protocole RFCOMM et ne permettent donc aucunement de communiquer directement avec le dongle.
Il faut rappeler ici que les dongles Bluetooth peuvent communiquer via un driver HCI tournant sur différents protocoles de communication : BCSP, H2, H4 et H5, signifiant respectivement un protocole réservé aux puces CSR (Cambridge Silicon Radio), HCI sur USB, HCI sur UART et HCI sur UART 3 câbles. Dans notre cas, un port ouvert en protocole H4 est nécessaire au bon fonctionnement de ces *stack*s, autrement dit une connexion directe avec les drivers du plus bas niveau du dongle Bluetooth. Comme nous le verrons plus loin, c'est par ce moyen que la *stack* Linux fonctionne. Sous Windows en revanche, il est impossible de trouver un moyen d'ouvrir un port de cette manière, sauf apparemment à l'aide de la *stack* Com one que nous allons analyser.

### Com one

La *stack* Com one est une petite *stack* française peu connue, dont aucun SDK n'est fourni et dont aucune information particulière ne nous avait frappés. Toutefois, en cherchant à accéder à la couche HCI sous Windows, ce qui reste notre dernière possibilité pour connecter la Sixaxis sous Windows, nous avons trouvé ce sujet sur les forums de OSDir.com :

"The COM1 *stack* used to allow direct HCI access (H4 interface) if opening as a serial port the special file declared in HKLM\SYSTEM\CurrentControlSet\Enum\USB\Vid_xxxx&Pid_xxxx\something_goes_here \Device Parameters\SymbolicName (replace Vid_xxxx&Pid_xxxx with the ones from your device)"[57]

(Par Xavier Garreau (www.xgarreau.org), 2004)

Après installation (forcée) d'un des nombreux drivers Com One, on remarque effectivement que dans la base de registre des valeurs de type *SymbolicName* sont apparues. En revanche, Com One n'y fait aucune allusion et, surtout, l'ouverture d'un port sur ce *SymbolicName* est impossible. Des mails ont été envoyés à X. Garreau et aux développeurs de la *stack* Com One, restés pour le moment sans réponse. Reste que malgré la plausibilité de cette méthode, il semble surprenant qu'une *stack* à faible diffusion, mais toutefois grand public, fournisse une méthode aussi pratique et utile aux développeurs alors que les *stack*s plus diffusées mondialement ne le font pas. Reste à attendre la réponse de Com One.

## *Stack* Linux Bluez

### Introduction

Si l'on se tourne du côté de Linux, le choix d'une *stack* est rapidement fait: Bluez est la *stack* officielle fournie désormais avec les dernières mises à jour du Kernel. On peut légitimement se demander avec quel niveau de compatibilité cette *stack* supporte les dongles Bluetooth standard et quelles interfaces de programmation elle fournit pour coder nos propres applications. Pour faciliter le codage, il existe Pybluez, qui est une interface Python pour Bluez, qui permet de coder rapidement et efficacement de petites applications.

[57] http://ml.osdir.com/linux.bluez.devel/2004-03/msg00122.html

### Spécifications[58]

Voici les spécifications de cette *stack* :

- Implémentation modulaire
- Système multi-processing et multi-threadé
- Support pour de multiples périphériques Bluetooth
- Interface standard de type *socket* à tous les niveaux
- Support de tous les niveaux de sécurité

En ce qui concerne les modules fournis, nous avons :

- Noyau Bluetooth
- Couches L2AP et SCO audio
- RFCOMM, BNEP, CMTP, HIDP
- HCI UART, USB, PCMCIA et périphériques virtuels
- Utilitaires de configuration, de test, de décodage et d'analyse

Cette stacl existe sur les processeurs Intel et AMD x86, AMD64, EM64T, SUN SPARC (32, 64 bits), PowerPC (32, 64 bits), Intel StrongARM et Xscale, Hitachi/Renegsas SH processors et sur les Motoral Dragonball. Enfin, elle fournit un support pour les distributions Linux Debian GNU, Ubuntu, Fedora Core/Red Hat, OpenSuSe/SuSE, Mandrake. Bien évidemment, Bluez tourne sur quasiment n'importe quelle autre distribution Linux.
La philosophie de Bluez consiste à fournir un noyau disposant des fonctionnalités indispensables (HCI, L2CAP, SCO) ainsi que des utilitaires de configurations, de diagnostics et un serveur pour le profil HID. Les autres profils ne sont pas fournis, mais des applications tierces le permettent. Cette architecture a un énorme avantage, c'est que l'utilisateur module sa *stack* Bluetooth comme il le souhaite et aucun profil n'est imposé. Le serveur HIDD sert par exemple à gérer les connexions de type HID, mais il faut le lancer à la main et on peut le désactiver, l'adapter ou coder le sien.

### Connexion de la Sixaxis sous Linux, HowTo

Pour utiliser son dongle Bluetooth, il faut commencer par l'activer (up) et le rendre connectable et découvrable (page et scan = piscan) :

```
hciconfig hci0 up piscan
```

Ensuite, il faut appairer la Sixaxis :

```
./sixaxis
```

Il doit trouver automatiquement l'adresse MAC du dongle local. Sinon, cela signifie qu'il y a eu un problème d'activation.
Une fois ceci fait, il y a le choix selon ce qu'on veut faire. On peut lancer le serveur HIDD :

```
hidd --server --nocheck -n
```

[58] http://www.bluez.org/

Le gros inconvénient de cette méthode est que les données HID sont ensuite passées au *kernel* Linux qui, selon son implémentation, mappe quelques touches sur un périphérique joystick. Cela ne supporte ni les accéléromètres, ni les touches analogiques, en résumé rien d'utilisable, mais utile pour le développement, car on peut analyser les trames à l'aide d'un hcidump :

```
hcidump -x
```

En effet, on peut étudier les trames Bluetooth provenant de la manette et en analyser les champs (cf. le chapitre "Étude de la Sixaxis"). Sinon on peut utiliser le logiciel sixaxis (cf. la section "Logiciels et démonstrations") qui permet de connecter un ou plusieurs Sixaxis à l'ordinateur et à mapper n'importe quel contrôle très facilement à l'aide de son interface en C. Il supporte également le *rumble*, les touches analogiques, les accéléromètres et les LEDS.

### Pybluez

Pybluez, comme la *stack* Java Electricblue, est une abstraction tournant par-dessus une autre *stack*. En l'occurrence, Pybluez tourne au-dessus de Bluez. Il en existe une version Windows, tournant sur les *stack*s Microsoft et Widcomm. Sans grande surprise il ne permet pas d'avoir plus de fonctionnalités, ce qui signifie que les *sockets* L2CAP sont impossibles avec la *stack* Microsoft et le HID est réservé à la *stack* Widcomm. Ce qui n'est pas le cas sous Linux, où le langage Python permet de coder en moins de 10 lignes une interface de connexion pour la Sixaxis. Quelques exemples sont fournis dans l'annexe "Logiciels".

## Conclusion

Après une analyse de la Sixaxis, les besoins pour l'utiliser ont été clairement définis : soit un accès direct au driver HCI pour coder soi-même toutes les connexions, soit un accès à la couche L2CAP avec accès au service HID, soit un accès à la couche BTHID pour envoyer la commande d'activation. Sous Linux, on peut y accéder au niveau HCI (mais cela est inutile et fastidieux étant donné qu'on a à disposition la couche L2CAP) ou L2CAP grâce à l'interfaçage *socket*. Sous Windows, les besoins sont identiques, mais malgré la multitude de *stack*s fournies, aucune ne permet d'accéder, soit à la couche désirée, soit d'y effectuer toutes les opérations voulues. Aucun SDK à part Widcomm ne permet d'atteindre la couche L2CAP, qui est pourtant indispensable pour coder des applications développées. Quand on parle de Bluetooth avec une personne non initiée, elle fait toujours allusion à son téléphone portable et à un échange de fichiers entre celui-ci et son ordinateur. Nous avons un peu l'impression que les développeurs de SDK se sont concentrés sur cette fonctionnalité et ont fourni une interface dans cette optique : sous la *stack* Microsoft, nous n'avons qu'un accès au RFCOMM, qui est le protocole utilisé pour envoyer des fichiers sur un téléphone portable ou pour en transférer une connexion internet. Sous Widcomm également : la couche L2CAP est disponible, mais uniquement pour quelques applications personnelles (comme un *chat*, seule fonctionnalité pour laquelle aucun profil n'est fourni). En revanche le RFCOMM et le transfert de fichier sont parfaitement disponibles, tout comme les connexions audios afin de transférer ses appels sur son ordinateur. Les périphériques HID

Bluetooth commencent doucement à entrer dans les moeurs, reste à espérer que les concepteurs de *stack* vont suivre le mouvement et inclure cette gestion dans leur SDK. Toutefois, il est probablement trop tard étant donné que tous les périphériques HID utilisent à présent des dongles Bluetooth propriétaire : il est à présent impossible de connecter une souris Bluetooth sur un dongle Bluetooth standard et donc il est impossible de développer une quelconque application Bluetooth avec. Une idée à développer serait de concevoir une interface de drivers HCI universels pour Windows, comme Bluez le fait. Nous en reparlerons dans la section “travaux futurs”.

# YAMOR ET BLUETOOTH HCI

## Introduction

À présent que le monde des dongles grand public a été analysé sous toutes ses formes et dans tous les domaines, il convient d'étudier les dongles dits "de développement". En effet, un des buts de ce projet est d'utiliser la Sixaxis (ou la Wiimote) sur un circuit embarqué contenant un module Bluetooth. Selon les modules Bluetooth, plusieurs possibilités de développement s'offrent à nous : on peut toujours coder directement en HCI, ce qui reste assez compliqué, mais cela a l'avantage d'être universellement compatible. En effet, les drivers HCI sont les drivers les plus communément fournis juste au-dessus des drivers radio. Moins souvent, les drivers Baseband sont également à disposition, mais cela reste sans intérêt vu que la couche HCI est le dénominateur commun de tous les développements Bluetooth. Une autre possibilité et d'utiliser l'OS (*Operating System*) fourni *on-chip* et d'analyser les différentes possibilités d'interaction Bluetooth qu'il procure.
Pour nos tests et notre étude, le module Zeevo a été choisi. En fait, il n'a pas vraiment été choisi, mais étant donné sa grande disponibilité au laboratoire, ce choix paraissait judicieux pour éviter de devoir concevoir complètement une architecture on-chip pour effectuer des tests.

### YAMOR[59]

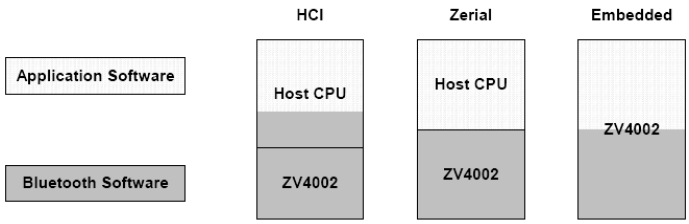
Ce module Zeevo est fourni sur YAMOR, qui est un robot modulaire développé au BIRG. Il possède un servomoteur qui permet à un axe de bouger sur un angle de 180°. L'intérêt de connecter la Sixaxis sur ce module est quasiment nul, étant donné que ce robot est autonome. Notre intérêt réside ici dans le module Zeevo pour fournir une interface de connexion utilisable ensuite sur d'autres robots qui nécessiteront un contrôle humain.

## Interface Zeevo

Zeevo est aujourd'hui la propriété de Broadcom[60]. L'inconvénient étant que Broadcom n'a pas poursuivi son développement et donc que les logiciels, les drivers et la documentation ne sont pas à jour depuis son rachat. Étant donné que le module Zeevo date de 2003, cela peut mener à certaines instabilités du système et à de possibles incompatibilité, étant donné que le module tourne en Bluetooth 1.1. Zeevo est fourni avec deux *firmware*s : Zerial ou HCI.

[59] http://birg.epfl.ch/page53469.html
[60] http://www.broadcom.com/press/release.php?ID=682189

De plus une version *embeded* (embarqué) peut être conçue, dont voici les différentes possibilités :

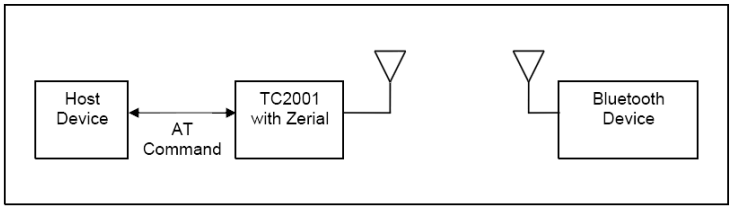



(SOURCE : ZEEVO USER'S GUIDE)

Nous nous intéresserons particulièrement à la version Zerial et HCI, l'*embedded* n'étant pas spécialement intéressant vu que Zeevo n'est utilisé que dans le cadre de Yamor. En effet, le prochain robot en développement au BIRG disposera d'un autre dongle de développement Bluetooth sur lequel il suffira d'adapter la version HCI.

## Zerial

Zerial est le nom du *firmware* fourni avec un système d'exploitation intégré. Environ 70% des sources sont disponibles afin de générer un *firmware* personnalisé. Notre but ici est de profiter de la disponibilité d'un OS pour connecter le Sixaxis au module Bluetooth de Zeevo. Pour ce faire, nous avons à disposition deux sources principales : la documentation très étoffée de Zerial et le code source. Nous n'allons pas nous attarder sur les détails nécessaires à la bonne compréhension de cet OS pour nous concentrer sur les possibilités Bluetooth. Zerial est conçu d'origine pour remplacer de manière transparente une connexion série entre deux périphériques Zeevo. En pratique, une simple commande en indiquant l'adresse MAC du périphérique sur lequel se connecter suffit à entre en "bypass" mode et simule une connexion série.




(SOURCE : ZEEVO USER'S GUIDE)

Pour utiliser Zerial de cette manière, une liste de commandes sont disponibles[61] à envoyer par connexion série. Ces commandes fournissent la possibilité de configurer l'interface, de lancer des *discovery* et de se connecter à un périphérique distant. À première vue donc, il y a possibilité de lancer une connexion via ces commandes; toutefois, il est clairement évident que cette connexion ne permettra pas de se brancher sur un périphérique autre qu'un 2e Zeevo. Si on étudie la trace d'une connexion sur une machine Linux, on voit que Zerial fait une requête SDP pour vérifier s'il s'agit bien d'un 2e Zeevo disposant du profil SPP (Serial Port Profil) et se déconnecte dans la négative. Le SPP se

[61] *Zerial Interface Reference Guide*

situant au-dessus du RFCOMM, il n'est pas possible de modifier la connexion afin d'accéder au HID, étant donné qu'il faut se trouver au niveau L2CAP. Toutefois, la première idée est d'étudier le code source pour trouver la trace de la création de cette connexion, car nous disposons d'une grande majorité des sources ainsi que d'outils permettant de compiler un nouveau *firmware*.

### Connexion Zerial

Une connexion Zerial se déroule en plusieurs étapes. La commande de connexion est envoyée via une interface AT (port série), qui intercepte les commandes et appelle les fonctions adéquates. La fonction de connexion créé un signal contenant un *flag* indiquant au système d'exploitation qu'il veut établir une connexion SPP vers l'adresse MAC fournie en paramètre. Cette connexion SPP est ensuite effectuée via une librairie contenant les quelques profils supportés par Zerial. Afin de créer une connexion HID, il convient alors de fourni à Zerial un profil HID adéquat et il étant donné que les sources des profils ne sont pas fournies (il s'agit de librairies), la conception d'un profil HID n'est pas envisageable. D'ailleurs, si l'on étudie la documentation fournie avec le kit d'évaluation, très peu de documents traitent du Bluetooth : la majorité concerne le système d'exploitation et sa gestion. La documentation concernant le Bluetooth se limite à l'explication du fonctionnement des profils fournis.
Au final, malgré le temps investi pour étudier et comprendre le *firmware* Zerial, il est impossible de l'utiliser pour une quelconque utilité autre que celle de base. D'après Rico Möckel[62], concepteur d'un système *scatternet* sur les robots YAMOR, il est impossible d'accéder à la couche L2CAP avec Zerial. C'est donc une raison suffisante de s'intéresser au second *firmware*, à savoir HCI, vers lequel nous nous sommes tourné. Il fournit moins de facilités de codage Bluetooth mais il permet, théoriquement, d'effectuer toutes les opérations Bluetooth souhaitées.

## HCI

### Introduction

La gestion d'une connexion Bluetooth via le HCI est le plus bas niveau qu'un codeur puisse atteindre. En effet, le HCI est une légère surcouche aux drivers Baseband, qui eux transmettent les commandes sous forme électrique. Cela signifie qu'un drivers HCI permet de faire *tout ce qu'on veut* avec un module Bluetooth. Cela signifie qu'à l'aide d'un drivers HCI, nous pouvons transformer un dongle Bluetooth en n'importe quel autre dongle Bluetooth embarqué ou spécialisé : les commandes HCI permettent même de modifier l'adresse MAC. Le HCI est donc le meilleur driver auquel un programmeur peut avoir accès en ce qui concerne le Bluetooth, même si sa gestion n'est pas aussi simple que les *sockets* L2CAP.
La gestion HCI est indispensable pour toute *stack* Bluetooth et beaucoup d'interfaces existent pour en simplifier l'emploi (comme les *stack*s Java). Toutefois, il a été décidé ici de coder directement en HCI

[62] http://birg.epfl.ch/page53075.html

sans utiliser une quelconque aide comme les *sockets* pour le rendre plus confortable : le but ici est de savoir précisément et à tout moment qu'est-ce qui se passe au niveau du dongle Bluetooth. La réussite d'une connexion Sixaxis en HCI est un travail ardu, mais s'il réussit, nous avons la certitude d'avoir totalement compris le processus de connexion et de pouvoir le simplifier pour n'importe quelle couche supplémentaire.

Le HCI de Zeevo n'est fourni avec aucune documentation et à juste titre étant donné que la spécification Bluetooth contient toutes les informations[63]. Le *firmware* HCI fournit une interface H4, ce qui signifie que les commandes HCI sont envoyées via une liaison série UART. Un logiciel permettant de tester et d'interagir un peu avec la couche HCI est disponible pour Zeevo : il s'agit de bthost, qui est un excellent logiciel d'apprentissage pour comprendre la couche HCI.

### La connexion Sixaxis via HCI

Nous allons donc indiquer ici quelles commandes n'importe quel dongle Bluetooth reçoit et doit envoyer en protocole H4 (HCI sur Uart) pour la connecter et la garder connectée. Les ← indiquent des paquets entrants et sortants en direction du dongle Bluetooth. Un paquet entrant ne provient pas de la Sixaxis, mais du dongle Bluetooth. Il peut donc s'agir de données interprétées par HCI et provenant de la Sixaxis ou de données venant seulement du dongle Bluetooth qui nous informe de l'état des commandes.

**Remarques importantes** : La documentation officielle Bluetooth ne fournit que la signification des différents paquets ainsi que des schémas indiquant les protocoles de connexion. En revanche, il n'existe pas de modèle de connexion HCI au niveau H4 et le code qui suit n'en est pas un : ce code est exclusivement réservé à la Sixaxis. En effet, une connexion HCI et L2CAP comportent selon le standard plus d'interactions, comme des modifications de paquets, des demandes de noms, etc. Nous avons conservé ici le strict minimum pour que la connexion réussisse dans le 100% des cas. L'ordre est évidemment obligatoire, spécifiquement pour les *commands status* qui peuvent théoriquement être reçues dans une plage de temps variable, mais dans notre cas elles sont réceptionnées précisément à ces moments. Il est bien évident que le protocole qui suit peut être copié et adapté tel quel et permet, avec un driver HCI, de connecter une Sixaxis sur n'importe quel appareil.

**← Write scan enable**

Fonction : Activer la connectabilité et la découverte de notre périphérique

Code : 0x01,0x1A,0x0C,0x01,0x03

**→ Write scan enable acknowledge**

[63] Cf. Chapitre sur le protocole Bluetooth

Fonction : Réception de la confirmation de la part du module Bluetooth
Code : 0x04, 0x0F, 0x04, 0x02, 0x01, 0x09, 0x04

**→ ACL connection request**
Fonction : Une requête de connexion de type ACL est reçue.
Code : 0x4, 0x4, 0x0A, **0xF8, 0x62, 0xE2, 0xC1, 0x19, 0x00**, 0x00, 0x00, 0x00,0x01
Particularité : En gras, l'adresse MAC en *little-endian* de la provenance de la connexion. À sauvegarder pour répondre correctement à cette requête.
**← ACL connection accept**
Fonction : Réponse positive à la requête de connexion ACL.
Code : 0x01,0x09,0x04,0x07,**0xF8,0x62,0xE2,0xC1,0x19,0x00**,0x01
Particularité : En gras, l'adresse MAC en *little-endian* du périphérique dont on accepte la connexion.
**→ Command status**
Fonction : Indique l'état de la commande envoyée.
Code : Variable
Particularité : Le code fait une longueur de 7 .
**→ ACL connection complete**
Fonction : La connexion ACL est effective.
Code :
0x4, 0x3,0x0B,0x00,0x00,0x00,**0xF8,0x62,0xE2,0xC1,0x19,0x00**,0x01,0x00
Particularité : Encore une fois, en gras, l'adresse MAC du périphérique connecté.
**→L2Cap connection request, psm 11**
Fonction : Une connexion de type L2Cap est demandée
Code : 0x02, 0x2c, 0x20, 0x0c, 0x00, 0x08, 0x00, 0x01, 0x00, 0x02, 0x01, 0x04, 0x00, 0x11, 0x00, 0x44, 0x00
Particularités : les 4 premiers bytes comportent un en-tête HCI, un handle et la longueur totale du paquet (0x0c, 0x00). Ensuite nous avons le paquet L2CAP, avec la taille totale en en-tête (0x0c, 0x00) et la taille du *payload* (0x08, 0x00), le canal de commande (0x01, 0x00). Le 0x02 suivant correspond à une demande de connexion L2CAP, suivi de son identificateur (0x01). Le 0x04 correspond à la taille des paramètres (0x04, 0x00), suivi enfin du PSM de la connexion (0x11, 0x00) et du CID (0x44, 0x00). Dans la réponse le PSM et le CID seront nécessaires.

**← L2Cap connection response, psm 11**
Fonction : Acceptation de la connexion L2CAP
Code : 0x02, 0x2c, 0x20, 0x10, 0x00, 0x0c, 0x00, 0x01, 0x00, 0x03, 0x01, 0x08, 0x00, 0x40, 0x44, 0x00, 0x00, 0x00, 0x00, 0x00
Particularités : comme avant, les 4 premiers bytes sont l'en-tête HCI (0x02, 0x2c, 0x20) suivi des deux tailles (0x10, 0x00 et 0x0c, 0x00) et du canal (0x01, 0x00). Puis le paquet L2CAP : 03 indique une

acceptation de connexion, suivi de l'ID et de la taille des paramètres (0x08, 0x00). Vient ensuite le CID de la source (0x00, 0x40), nécessaire par la suite et le CID de la cible (0x00, 0x44), nécessaire aussi, mais su d'origine puisque c'est nous qui l'avons envoyé en premier. Les zéros qui suivent indiquent que nous ne souhaitons aucune option particulière.

**→L2Cap configuration request**

Fonction : Demande de configuration avec paramètres

Code : 0x02, 0x2c, 0x20, 0x0c, 0x00, 0x08, 0x00, 0x01, 0x00, **0x04**, 0x02, 0x04, 0x00, 0x40, 0x00, 0x00, 0x00

Particularités : après l'en–tête HCI, le 0x04 indique une demande de configuration, avec un identifieur (0x02). Suivent la taille des paramètres (0x04), le CID de la source (0x00, 0x40) et enfin des bytes flags indiquant qu'aucun configuration particulière n'est souhaitée.

**←L2Cap configuration response**

Fonction : Réponse à une demande de configuration en indiquant nos options.

Code : 0x02, 0xc, 0x20, 0x0e, 0x00, 0x0a, 0x00, 0x01, 0x00, **0x05**, 0x02, 0x06, 0x00, 0x44, 0x00, 0x00, 0x00, 0x00, 0x00

Particularités : Le code de réponse est le 0x05, suivi de l'ID, de la taille, du CID destinataire et des réponses. Les 0x00 indiquent l'acceptation de tous les paramètres.

**← L2Cap configuration request**

Fonction : Idem que plus haut

Code : 0x02, 0x2c, 0x20, 0x10, 0x00, 0x0c, 0x00, 0x01, 0x00, **0x04**, 0x01, 0x08, **0x00, 0x44**, 0x00, 0x00, 0x00, 0x01, 0x02, 0x040, 0x00

Particularités : Comme citées plus haut. Il est important de reporter le CID correct (ici, 0x00, 0x44). Le reste est à définir selon les désirs du programmeur.

**→L2Cap configuration response**

Fonction : Idem que plus haut.

Code : Idem que plus haut.

Particularités : La réponse a une taille de 19 bytes.

Les commandes qui suivent ont le même schéma que plus haut car une 2ème connexion L2Cap est ouverte :

**→L2Cap connection request, psm 13**

Fonction : Une connection de type L2Cap est demandée

Code : 02 2C 20 0C 00 08 00 01 00 02 03 04 **00 13 00 45** 00

Particularités : Le PSM vaut 0x00 0x13 cette fois et le CID est à retenir (0x00, 0x45, variable).

**→ Command status**

Fonction : Indique l'état de la commande envoyée.

Code : Variable

Particularité : Le code fait une longueur de 8 cette fois.

**← L2Cap connection response, psm 13**

Fonction : Acceptation de la connexion L2CAP

Code : 02 2C 20 10 00 0C 00 01 00 03 03 08 **00 41 00 45** 00 00 00 00

Particularités : Ne pas oublier les CID, sources et destinations (0x00, 0x41 et 0x00, 0x45). Le premier est arbitraire, c'est au programmeur de le choisir.

**→L2Cap configuration request QOS**

Fonction : Demande de configuration avec paramètres

Code : 02 2C 20 24 00 20 00 01 00 04 04 1C 00 41 00 00 00 03 16 00 02 00 00 00 00 00 00 00 00 00 00 00 00 F2 2B 00 00 FF FF FF FF

Particularités : La demande est particulièrement longue car du QOS (quality of service) est demandé. Inutile de s'affoler, car aucune information importante n'est à retenir.

**←L2Cap configuration response**

Fonction : Réponse à une demande de configuration en indiquant nos options.

Code :

02 2C 20 0E 00 0A 00 01 00 05 04 06 **00 45** 00 00 00 00 00

Particularité : Le CID est important et doit être identique à la valeur fournie

**← L2Cap configuration request**

Fonction : Idem que plus haut

Code :

02 2C 20 10 00 0C 00 01 00 04 02 08 00 45 00 00 00 01 02 40 00 Particularités : Aucune, comme avant c'est par pure politesse qu'on envoie une configuration request à la Sixaxis.

**→L2Cap configuration response**

Fonction : Idem que plus haut.

Code : Idem que plus haut.

Particularités : La réponse a une taille de 19 bytes.

**→ Command status**

Fonction : Indique l'état de la commande envoyée.

Code : Variable

Particularité : Le code fait une longueur de 8 cette fois.

**←HID SET REPORT**

Fonction : Active la Sixaxis

Code : 02 2C 20 0A 00 06 **00 44** 00 53 F4 42 03 00 00

Particularité : Le CID est important et doit être identique à la valeur fournie

Suivent les *Reports* de la Sixaxis, dont la taille varie entre 50 (Bluetooth 2.0) et 23 bytes (Bluetooth 1.1). Il faut noter que deux command status interviendront quelques secondes après la connexion, il faut en tenir compte pour l'alignement des données.
Une fois ceci effectué sur notre module YAMOR, les données Bluetooth HID arrivent de manière régulière.

### Taille des paquets

Une fois la procédure de connexion décrite plus haut exécutée sur le module YAMOR, les données sont envoyées régulièrement à notre module. Toutefois, les paquets envoyés n'ont pas la taille de 50 bytes habituelle, mais 23 bytes et seule les valeurs numériques de la manette sont envoyées, à savoir :

- Tous les boutons
- Les deux joysticks en analogique
- Pas d'accéléromètre ni aucune valeur analogique pour les boutons

La Sixaxis se comporte donc comme la manette d'une Playstation 1, qui a un *design* identique, mais aucune valeur analogique sur les boutons.
Après différents tests et analyses, nous avons découvert que ces paquets sont envoyés lors de l'utilisation d'un dongle Bluetooth de type 1.1. Seuls les dongles Bluetooth 2.0 ou 2.1 permettent de recevoir les paquets complets. La raison se situe dans le format des paquets : la norme 2.0 introduit des paquets EDR qui permettent d'envoyer des paquets de taille supérieure, sans perte de bande passante. La norme 1.1 de Bluetooth ne supportant pas ces paquets, ceux-ci sont envoyés dans un format compatible. Il est toutefois assez étrange que les concepteurs de la Sixaxis aient envisagé la connexion de la Sixaxis sur un module Bluetooth 1.1 alors qu'elle n'est conçue que pour se connecter sur un module Bluetooth 2.0.

### Conclusion

Ainsi, le passage à du matériel de développement et à des drivers de bas niveau nous a permis de réussir l'opération souhaitée depuis le début de cette étude, à savoir la connexion de la Sixaxis sur un autre périphérique que la Playstation 3 et avec nos propres interfaces. De plus, le fait d'être parvenu à effectuer cette connexion avec la plus basse couche de Bluetooth prouve que notre compréhension du protocole Bluetooth du fonctionnement de la Sixaxis est correcte. Désormais, nous sommes en mesure de concevoir rapidement une interface de connexion pour la Sixaxis à l'aide d'une interface HCI, L2CAP ou BTHID. De la sorte, nous pouvons fournir une liste de pré requis afin de sélectionner des modules Bluetooth de développement dont nous serons certains de la compatibilité avec la Sixaxis et de mettre à disposition du futur développeur un mode d'emploi bref et détaillé de la procédure à suivre pour la connecter. Ceci fait, nous pouvons désormais nous concentrer sur les derniers détails de la Sixaxis, à savoir l'étude complète des paquets qu'elle envoie et des commandes qu'on peut lui envoyer.

# ETUDE DE LA SIXAXIS

## Introduction

Ce projet avait originellement deux faces : une orientée pratique, souhaitant fournir des outils pour connecter la Sixaxis sur ordinateur ou sur de vrais robots. La 2ème, orientée recherche, avait pour but d'analyser la Sixaxis, son comportement et étudier comment l'activer, la faire réagir et à la rigueur l'émuler.Il s'agit ici de reverse-engineering, où il s'agit de comprendre la volonté des concepteurs de la Sixaxis et de la PS3 afin d'en détourner le fonctionnement pour notre propre usage. Bien entendu, il faut savoir que chercher avant de se lancer dans l'étude, et nous sommes ici particulièrement intéressés de :

- Comprendre à quelles informations correspond chaque champ des paquets HID envoyés par la Sixaxis
- Comprendre quelles commandes envoyer à la Sixaxis pour l'activer, l'éteindre, activer son *rumble* et gérer l'affichage des LEDS de la manette
- Étudier comment la Sixaxis allume/éteins la PS3 et s'y connecte pour comprendre comment l'émuler

Mais tout d'abord, il nous faut trouver des outils pour notre étude et les adapter à la console Playstation 3.

## Analyse des paquets HID

### Procédure d'analyse

Pour étudier les paquets envoyés par la Sixaxis, étant donné qu'une connexion est possible directement avec un ordinateur, un *dump* (à l'aide de hcidump) permet d'étudier librement les paquets envoyés. En revanche, les commandes d'initialisation de la console ou d'extinction sont indisponibles. De plus, impossible de connaître les paquets envoyés par la console par ce moyen. Il a donc fallu trouver une autre méthode pour capter ces paquets directement à la volée. Un analyseur USB aurait été envisageable, avec la Sixaxis connectée à la console, mais les commandes sont probablement différentes et l'USB ne nous intéresse pas vu que nous voulons une connexion Wireless. Il reste donc l'utilisation d'un analyseur Bluetooth.

L'inconvénient d'un analyseur Bluetooth réside dans sa synchronisation avec les appareils analysés.

Cette synchronisation n'est en principe pas un problème avec des interfaces Bluetooth configurables.

En quoi ces méthodes de synchronisation peuvent poser problème ? Dans le cas où les périphériques dont nous souhaitons effectuer une analyse ne sont pas visibles, autrement dit, en mode *inquiry*. C'est le cas non seulement de la Playstation 3, qui est invisible, mais également de la Sixaxis. Les raisons en sont les suivantes : en ce qui concerne la Sixaxis, il lui est inutile d'être découvrable, étant donné que

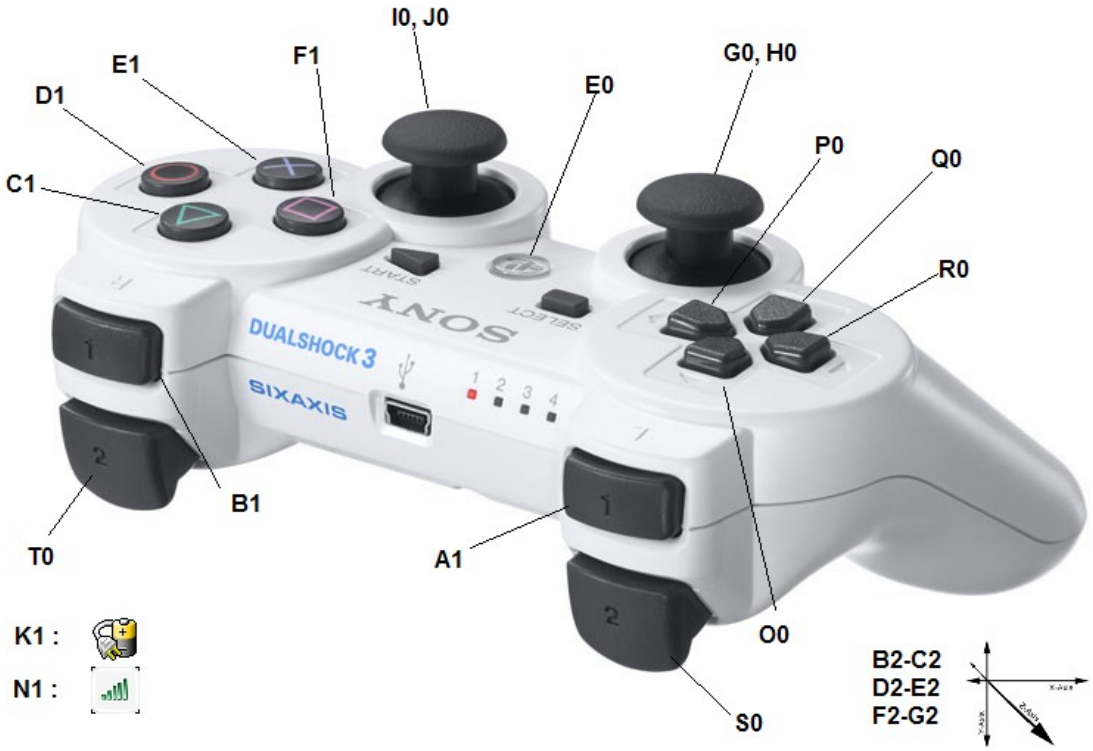
c'est elle qui initie la connexion. De plus, cela nécessite une consommation d'énergie totalement inutile pour un périphérique portable. Du côté de la PS3, son invisibilité est également parfaitement compréhensible : tout d'abord pour des raisons de sécurité, il n'est pas nécessaire qu'elle soit visible, étant donné que la Sixaxis connaît son adresse et s'y connecte. En ce qui concerne la connexion d'autres périphériques, comme des casques-micro, la PS3 peut également rester invisible, vu qu'il lui suffit de rechercher les périphériques à connecter qui eux, seront visibles. Ceci est en revanche ennuyeux pour notre analyseur, car nous ne pouvons plus qu'utiliser une seule méthode pour se synchroniser, à savoir la méthode page *master*.
Toutefois, cette méthode ne fonctionne que de manière aléatoire, car la PS3 refuse toute connexion ne provenant pas d'une de ces manettes enregistrées. Nous verrons cela dans la partie Émulation Sixaxis, mais cela signifie pour le moment que la console refuse la connexion de notre analyseur et lui renvoie donc un décalage d'horloge (*clock offset*) de 0x00, ce qui mène à une désynchronisation (ou plutôt à une non-synchronisation) très fréquente et donc à aucun paquet analysé. Néanmoins, à force de persévérance, les paquets nécessaires à notre étude ont été enregistrés.

### Paquets sortants

Les paquets sortants de la Sixaxis font une taille de 50 bytes. Il est clair que certains bytes sont probablement réservés pour un usage futur, néanmoins 50 bytes est une valeur assez conséquente pour transférer des informations à propos de l'état des boutons d'une manette.
Voilà un paquet HID typique envoyé par la Sixaxis (les en-têtes ont été éliminés, seules restent les informations HID) :

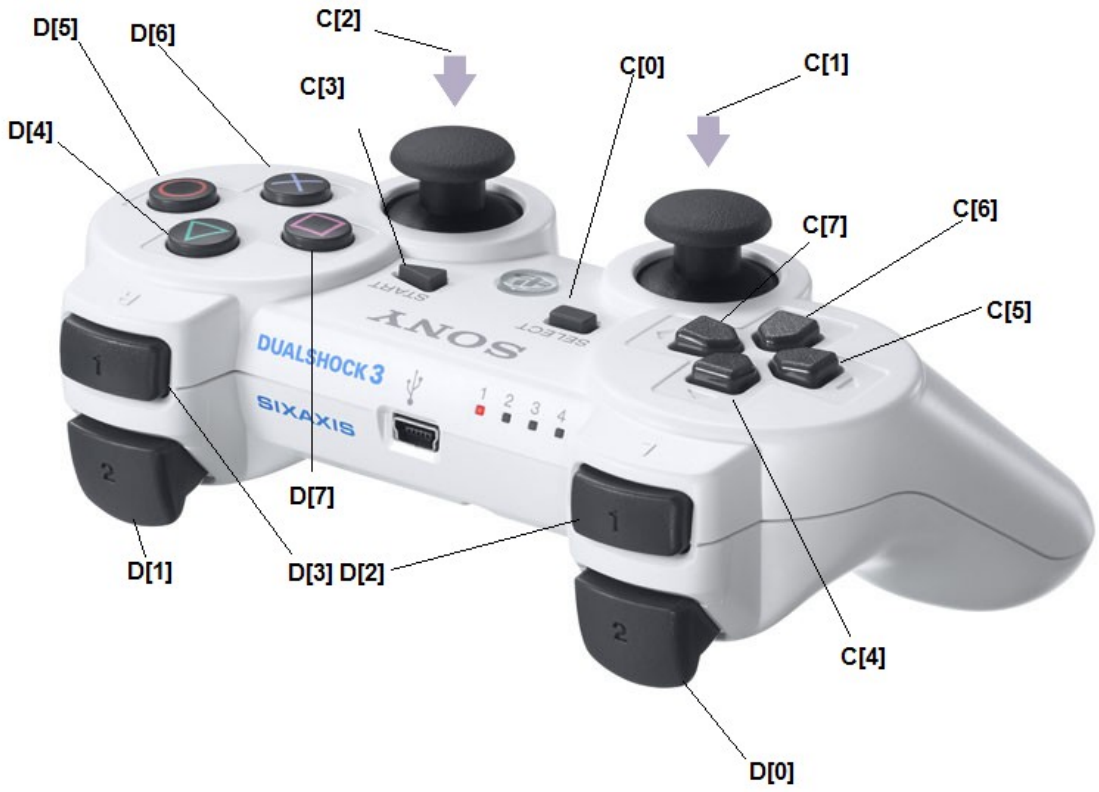
```
01 00 00 00 00 00 84 82 8D 7C 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 03 05 14 00 00 00 00 23 0F 77 01
81 02 07 01 EA 01 9B 00 02
```

Après diverses analyses, beaucoup de tests et d'hypothèses, voici à quoi ces informations correspondent (dans l'ordre). Le format est le suivant : la première colonne correspond à 1 byte. Si ce byte est décomposé en plusieurs bits d'information, une 2ème colonne est appondue de cette manière xx::xx: définition du premier bit. Les x correspondent à des valeurs variables, (x¦y) un bit ayant deux valeurs observées et 00 ou d'autres chiffres à des valeurs fixes.




MAPPAGE ANALOGIQUE

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 01 | 00 | 00 | 00 | 00 | 00 | 84 | 82 | 8E | 7C | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 1 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 03 | 05 | 14 | 00 | 00 | 00 | 00 | 23 | 0F | 77 | 01 |
| 2 | 81 | 02 | 08 | 01 | EB | 01 | 9B | 00 | 02 | | | | | | | | | | | |




**MAPPAGE NUMERIQUE**

Développement de C0 et D0 en bits :

C0 = C[0], C[1], C[2], C[3], C[4], C[5], C[6], C[7]

D0 = D[0], D[1], D[2], D[3], D[4], D[5], D[6], D[7]

Détails :

01 : Inconnu

00 : Inconnu. Il s'agit probablement, avec le byte précédent, d'un identificateur de périphérique, permettant de différencier les Sixaxis d'un autre type de contrôleur.

xx :: xx: touche select

xx: joystick gauche clique. En effet, on peut "cliquer" sur les deux joysticks à disposition

xx: joystick droite clique

xx: joystick start

xx: croix up numérique

xx: croix right numérique

xx: croix down numérique

xx: croix left numérique

xx :: xx: grande gâchette gauche numérique. Le manette est effectivement dotée de 4 gâchettes, dont deux grandes permettant une gestion plus fine de l'appui, du moins pour les valeurs analogiques.

xx: grande gâchette droite numérique

xx: petite gâchette gauche numérique

xx: petite gâchette droite numérique

xx: triangle numérique

xx: rond numérique

xx: croix numérique

xx: carré numérique

xx : Bouton PS. De manière surprenante, le bouton PS a un byte complet à sa disposition alors qu'il n'est que numérique.

00 : Inconnu
xx : joy_x gauche. Les joysticks n'ont que des valeurs analogiques.
xx : joy_y gauche
xx : joy_x droite
xx : joy_y droite
00 : Inconnu
00 : Inconnu
00 : Inconnu
00 : Inconnu
xx : croix up analogique. Tous les boutons sauf start, select, PS et joystick_clique ont une valeur numérique (sur 1 bit) et une valeur analogique (sur 1 byte)
xx : croix right analogique
xx : croix down analogique
xx : croix left analogique
xx : grande gâchette g analogique
xx : grande gâchette d analogique
xx : petite gâchette g analogique
xx : petite gâchette d analogique
xx : triangle analogique
xx : rond analogique
xx : croix analogique
xx : carré analogique
00 : Inconnu
00 : Inconnu
00 : Inconnu
03 : Inconnu
0x : batterie. La batterie a une valeur variant en 05 et 00. 05 correspond aux 3 barres pleines, 04 à 2 barres, 03 à 1 barre, 02 à 1 barre clignotante et 01 à une batterie vide.
14 : Inconnu
FF : Inconnu
xx : force du signal. De manière surprenante, une valeur indiquant la force du signal Blutooth est inclue dans les paquets HID. Cette valeur est mise à jour toutes les 5 secondes et varie sur les 255 valeurs de son byte.
00 : Inconnu
00 : Inconnu
23 : Inconnu
xx : Inconnu (FC ¦30)
77 : Inconnu
01 : Inconnu
81 : Inconnu
xxxx : accéléro z. Les accéléomètres ont des valeurs codées sur 2 bytes. Ceci est légitime, car la pleine amplitude des accéléromètres n'est quasiment jamais utilisées durant un jeu normal, c'est pourquoi, afin de conserver la même sensibilité que les joysticks, ils sont codés sur 2 bytes : 1 byte étant réservé aux "zones extrêmes" et 1 byte pour les valeurs utilisables
xxxx : accéléro x
xxxx : accéléro y
00 : Inconnu
02 : Inconnu

### Paquets entrants

Comme nous avons pu le voir précédemment, la Sixaxis envoit beaucoup de données, toutefois, il est également possible qu'elle en reçoive pour configurer son *rumble* ou ses LEDS.
Voici un paquet que la Sixaxis peut accepter (mis en forme avec numérotation de ligne et explications):

1: 0x52,0x01,0x00, 0x00, 0x00, 0xff, 0x72, 0x00, 0x00, 0x00,0x00,
2: 0x02, Activation des LEDS
3: 0x00, 0x00, 0x00, 0x00, 0x00, Gestion LED 4
4: 0x00, 0x00, 0x00, 0x00, 0x00, Gestion LED 3
5: 0x00, 0x00, 0x00, 0x00, 0x00, Gestion LED 2
6: 0xff, 0x27, 0x10, 0x32, 0x32, Gestion LED 1
7: 0x00, 0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

La première ligne contient l'en-tête HID ainsi que des valeurs permettant d'activer le rumble (si la manette en est dotée). Le byte 5 (en énumérant depuis 0) configure la durée du rumble (0xff l'active de manière permanente) alors que le byte 6 règle la puissance de la vibration.
La ligne 2 contient un byte d'activation pour les LEDS. Le 7e bits (en comptant selon le modèle *Big-Endian,* c'est-à-dire de gauche à droite) enclenche la première LEDS, le 6e bits la 2e, le 5e la 3e et le 4e la 4e. Les lignes 3-6 contrôlent ensuite les paramètres des LEDS, à savoir la ligne 6 pour la première LED, la ligne 5 pour la 2e LED, la ligne 4 pour la 3e et la ligne 3 pour la 4e.
Ces paramètres de contrôles sont les suivants (dans l'ordre) : le premier byte contrôle le timer, c'est-à-dire le temps après lequel la LED s'enclenche. Le 2e byte contrôle la période, donc la fréquence du cycle extinction-allumage de la LED. Le 3e byte contrôle un décalage par rapport aux autres LEDS. Enfin, les bytes 4 et 5 gèrent respectivement la durée d'extinction de la LED et la durée d'allumage.
Ce type de paquet peut être envoyé à n'importe quel moment durant la connexion de la Sixaxis. La seule contrainte est que selon le protocole HID, le module Bluetooth doit attendre la confirmation (*ack*) de la commande de la part de la Sixaxis. Cet *ack* peut prendre quelques ms, il est donc inutile et contre-productif d'envoyer des commandes en continu, car aucune ne serait acceptée étant donné qu'une commande envoyée avant la réception d'un *ack* annule la commande précédente. Néanmoins, pour une utilisation standard, il est inutile de vouloir changer la configuration des LEDS ou du *rumble* toutes les ms et il est déjà surprenant d'avoir à disposition autant de paramètres pour configurer les LEDS alors que dans le cadre du jeu, seul l'allumage de la LED correspondante au numéro du joueur importe.
Ainsi, nous avons à notre disposition le protocole de connexion, celui de déconnexion (qui consiste juste à se déconnecter au niveau Bluetooth) ainsi que les paquets entrants et sortants de la Sixaxis. Il devient donc envisageable d'émuler la manette à l'aide d'un dongle Bluetooth quelconque pour simuler la connexion d'une Sixaxis sur la Playstation 3.

## Émulation Sixaxis

### Utilité

L'émulation d'un système peut avoir plusieurs utilités. La première est, bien évidemment, de remplacer le système par un autre que l'on considère plus adéquat ou plus pratique. C'est le cas des émulateurs de console de jeux ou de systèmes d'exploitation (machines virtuelles) : on émule un système par praticité pour les avoir sur un ordinateur. Une autre utilité permet de démontrer que l'on maîtrise techniquement l'appareil émulé. En effet, sans les connaissances complètes du fonctionnement, il est impossible de l'émuler correctement. Dans notre cas, nous avons un emploi supplémentaire à l'émulation de Sixaxis : son remplacement par d'autres périphériques de contrôle.

Pour bien comprendre cela, il faut se mettre dans le contexte des jeux vidéos. En effet, de nombreux joueurs de consoles sont également des joueurs sur ordinateurs. La majeure différence entre ces deux mondes réside souvent dans le contrôle du jeu. Les joueurs PC sont en général habitués à utiliser un clavier et une souris, ce qui implique qu'ils sont très souvent mal à l'aise avec une Sixaxis dans les mains. L'avantage de l'émulation d'une Sixaxis serait donc de concevoir un boîtier sur lequel on pourrait brancher un clavier et une souris tout en émulant, du côté de la console, le fonctionnement d'une Sixaxis.

### Conception

Étant donné que nous sommes en possession des paquets entrants et sortants de la Sixaxis, il ne reste que peu d'inconnues pour l'émuler à l'aide d'un module Bluetooth. Il reste en effet à savoir comment allumer la console et l'éteindre.

Pour cela, un log de l'allumage de la console PS3 nous apprend qu'aucune commande spéciale n'est utilisée : la Sixaxis initialise simplement une connexion HCI et la console s'allume. En ce qui concerne l'extinction, aucune commande non plus : pour éteindre la console, il faut :

- Appuyer durant 2 secondes sur le bouton PS
- Choisir l'option "éteindre la console"
- Confirmer avec la touche X

Tout ceci est donc géré de manière logicielle : il suffit d'émuler ce comportement en envoyant les paquets HID décodés plus haut et la console s'éteint.

### En pratique

Pour nos tests, nous avons utilisé le module Bluetooth du robot YAMOR à notre disposition. L'expérimentation ne peut être menée jusqu'à son terme, car notre module Bluetooth ne supportant que la norme 1.1 ne nous permet pas d'envoyer des paquets de taille correcte pour émuler la Sixaxis.

Toutefois, si nous parvenons à initialiser une connexion en direction de la console, le reste du travail ne consiste qu'à envoyer les paquets HID des commandes que l'on veut envoyer.
La première tentative est d'initialiser une connexion HCI en direction de la console. On remarque que celle-ci refuse la connexion pour des raisons de sécurité. Après réflexion et quelques tests, il nous est apparu que le seul moyen d'identification que possède la Playstation 3 se situe au niveau de l'adresse MAC des manettes. Bien entendu, les commandes HCI permettent de modifier celle-ci et la connexion est alors réussie et *la console s'allume* :

Demande de connexion HCI :

```
[16:01:45] HCI Command Send .....................................1
       packet sent: 01 05 04 0D 36 36 ED A9 13 00 08 00 00 00 00 00 00
       UART TL Header:              [1]  01   - HCI Command
       OpCode:                      [2]  0405 - CreateConnection
       Parameter Length:            [1]  0D
[16:01:39] HCI Event Received:
       Packet Received:
       04 0E 04 01 01 FC 00
       Event: CommandCompleteEvent
       UART Tl Header:              [1]  04
       Event Code:                  [1]  0E
       ParamLen:                    [1]  04
       Num HCI Command Packets:     [1]  01
       OpCode:                      [2]  FC01
```

Réponse de la console :

```
[16:01:45] HCI Event Received:
       Packet Received:
       04 03 0B 0E FF FF 36 36 ED A9 13 00 01 00
       Event: ConnectionCompleteEvent
       Status:                      [1]  0E   - host rejected due to security reasons
       Connection Handle:           [2]  FFFF
       BD_ADDR:                     [6]  00 13 A9 ED 36 36
       Link_Type:                   [1]  01
       Encryption Mode:             [1]  00
```

Changement d'adresse MAC :

```
[16:01:58] HCI Command Send .....................................1
       packet sent: 01 01 FC 06 55 4A 36 C1 19 00
       UART TL Header:              [1]  01   - HCI Command
       OpCode:                      [2]  FC01 - WriteBdAddr
       Parameter Length:            [1]  06
```

Nouvelle connexion :

```
[16:02:06] HCI Command Send .....................................1
       packet sent: 01 05 04 0D 36 36 ED A9 13 00 08 00 00 00 00 00 00
       UART TL Header:              [1]  01   - HCI Command
       OpCode:                      [2]  0405 - CreateConnection
       Parameter Length:            [1]  0D
```

Réussie :

```
[16:02:06] HCI Event Received:
        Packet Received:
        04 03 0B 00 00 00 36 36 ED A9 13 00 01 00
        Event: ConnectionCompleteEvent
        Status:                         [1]  00
        Connection Handle:              [2]  0000
        BD_ADDR:                        [6]  00 13 A9 ED 36 36
        Link_Type:                      [1]  01
        Encryption Mode:                [1]  00
```

Il est important de remarquer que la console ne s'allume que si elle tournait sous l'OS de Sony auparavant. En effet, si elle exécutait Linux auparavant, le dongle Bluetooth est éteint et ne peut donc pas réagir à la demande de connexion. Donc lorsque la console est en veille, le module Bluetooth reste allumé. Cet allumage signifie également que l'on peut émuler une Sixaxis du moment que l'on connaît l'adresse MAC d'une des Sixaxis–. Donc, lors du branchement USB entre la Sixaxis et la console, non seulement la Sixaxis enregistre la MAC de la console pour s'y connecter, mais également et surtout la console enregistre celle de la Sixaxis pour ne permettre que des connexions autorisées.
Voici donc une liste des prérequis pour notre boîtier d'émulation Sixaxis :

- Un module Bluetooth 2.0 ou 2.1, fourni soit avec des drivers HCI, soit avec un accès à la couche L2CAP et le moyen de changer son adresse MAC
- Deux (minimum) prises USB : pour y brancher la Sixaxis et copier son adresse MAC, pour y brancher un clavier et une souris et, pour plus de confort, pour pouvoir brancher notre boîte sur un ordinateur afin de la configurer (pour changer le *mappage* des touches par exemple)
- Un microcontrôleur (ou plus) contrôlant la connexion Bluetooth, la connexion à la Sixaxis pour copier son adresse MAC (protocole USB), les connexions HID provenant du clavier et de la souris (protocole USB également) et également pour gérer la traduction des contrôles provenant du clavier/souris en paquets HID Sixaxis
- De la mémoire (EEPROM ou autre) pour sauvegarder l'adresse MAC de la Sixaxis et/ou éventuellement les préférences de l'utilisateur
- Une batterie ou une source d'alimentation externe pour alimenter le module Bluetooth, le microcontrôleur et les prises USB

Bien évidemment, en plus des connaissances Bluetooth fournies ici, il est nécessaire de connaître le protocole USB et de développer les softwares embarqués pour le microcontrôleur pour concevoir cette boîte d'émulation.

## Les dernières inconnues

Bien évidemment, étant donné que nous sommes dans le domaine de la recherche, tout n'a pas pu être découvert ou analysé. En voici un bref résumé :

- Certains paquets HID entrants ou sortants ont des bytes inconnus, qui ont souvent la valeur 0 (on peut donc en déduire qu'ils sont réservés pour un usage futur) ou des valeurs fixes.
- Pourquoi la Sixaxis est-elle rétrocompatible avec le HID 1.1 et surtout, pourquoi est-ce que (apparemment), le formattage des paquets est conçu pour obtenir néanmoins toutes les valeurs de la manette malgré la taille réduite du paquet ?
- Pourquoi est-ce que l'on a un contrôle aussi étendu pour les LEDS alors qu'elles ne servent qu'à clignoter ou à s'allumer de manière fixe ?

# CONCLUSION

Ce projet de diplôme a été riche en découvertes, en nouvelles connaissances et en frustrations. Bien que le sujet fut clairement défini, sa réalisation changeait chaque jour selon les découvertes effectuées et le matériel à disposition. Cela nous a permis d'apprendre à être très flexibles et à assimiler de nouvelles technologies très rapidement, tout en restant concentré sur la tâche à accomplir et à ne pas se disperser sur des sujets annexes.

Au final, ce sujet est totalement différent de ce qui était initialement prévu. Nous pensions que les SDK sous Windows nous permettraient de connecter rapidement la Sixaxis sous Windows (1 à 2 semaines étaient prévues), mais force était de constater que ces SDK ne nous permettaient pas du tout de faire ce que l'on désirait. Cela nous a tout de même fourni une expérience et une compréhension du Bluetooth et de ces différentes *stack*s. Cette expérience est partagée ici avec le lecteur qui pourra se référer à ce travail pour le choix de la *stack* sur laquelle il souhaite éventuellement travailler, tout en lui fournissant des exemples et un mode d'emploi. En plus de cela, le lecteur a à sa disposition une base théorique et des exemples d'utilisation du protocole Bluetooth, qui est actuellement le protocole utilisé pour le contrôle à distance de périphérique.

Ce Bluetooth, qui nous semblait encore très instable au début, se révèle être quasiment au point. Il a l'avantage d'être innovant et de n'avoir copié son fonctionnement sur quasiment aucun protocole existant. Sans Bluetooth, la connexion Wireless de plusieurs périphériques à un ordinateur serait impossible ou nécessiterait de très nombreux modules radios prioritaires, aveugles les uns des autres. Le seul problème réside dans les interfaces logicielles, spécialement sous Windows. Espérons que les différentes *stack*s pourront se coordonner pour fournir une interface HID compatible afin d'éviter la prolifération de dongles Bluetooth propriétaires, comme nous le vivons en ce moment.

Ensuite, la Sixaxis, qui était encore relativement inconnue et inutilisée, est à présent en grande partie connue et nous avons fourni toute une série d'interfaces et de procédures pour la connecter et l'utiliser à son plein potentiel. Ceci était déjà fait pour la Wiimote, mais la Sixaxis fournit des fonctionnalités et une ergonomie dont la Wiimote ne dispose pas.

Enfin, ce projet fournit des outils. Des outils pour comprendre et utiliser le Bluetooth, des outils pour utiliser les *stack*s Bluetooth Windows/Linux, des outils pour concevoir une interface de connexion Sixaxis sur un système quelconque, des outils pour utiliser la Sixaxis sur ordinateur ou encore des outils pour gérer une interface Bluetooth comme on le souhaite. En effet, ce projet aura démontré qu'avec le protocole Bluetooth, tant qu'on peut accéder à la couche HCI et que la connexion n'est pas cryptée, on fait ce que l'on veut et on émule n'importe quel appareil. Car malgré tous ses défauts, le Bluetooth reste une norme avec une spécification disponible pour tous et avec à disposition une interface permettant d'effectuer toutes les opérations existantes sur ce protocole. En informatique, cette qualité est rare et recherchée.

## Travaux futurs

Avec tous ces outils à disposition, le nombre d'applications potentielles n'est limité que par l'imagination des développeurs ou les besoins des laboratoires. Toutefois, en voici une liste à titre d'exemple :

- Drivers Sixaxis sous Windows. La méthode est fournie, les *stack*s disponibles, il ne reste qu'à décider si l'investissement pour l'achat des *stack*s compatibles pour cette opération est rentable pour son utilisation.

- Émulateur Sixaxis pour Playstation 3. Cela serait plus utile dans un cadre commercial que celui de la recherche, mais cela reste une application très intéressante de cette étude.

- Branchement Sixaxis sur module de développement Bluetooth. Cela a déjà été fait et une méthode universelle est fournie, cela comporte tout de même un investissement en temps car les modules sont différents.

- Drivers HCI unifiés sous Windows. Sous Linux, avec Bluez, n'importe quel module Bluetooth peut être utilisé et dirigé à l'aide de commande HCI et L2CAP. Ceci est normal, vu que les drivers HCI doivent suivre la spécification Bluetooth et donc être compatibles. Sous Windows, un drivers unifié HCI permettrait à tous les développeurs d'utiliser le Bluetooth à son plein potentiel sans devoir s'encombrer d'API inadéquates.

# ANNEXES

## A. *Logs* de connexion

Les *logs* sont relativement conséquents pour être intégrés directement au texte du rapport, c'est pourquoi ils sont rassemblés ici. La totalité des *logs* est disponible au format électronique et seuls les plus intéressants sont répertoriés ici.

### Sixaxis sur *stack* Widcomm, sans serveur ni demande de connexion

Les points importants sont les suivants : la connexion des deux PSM L2CAP se fait correctement, y compris les demandes de configuration. Toutefois, à la ligne 14:38:22.312, on remarque que le périphérique HID est inconnu (dû au SDP lacunaire), ce qui engendre immédiatement une déconnexion car la commande d'activation est inconnue.

```
14:38:22.109 L2CAP RECV Command.  Name: L2C Connection Request (0x02) ID: 0x03, Len: 4
14:38:22.109                PSM : 0x13
14:38:22.109          Source CID : 0x2ba
14:38:22.109 --
14:38:22.109 L2CAP - LCID: 0x0041  st: CLOSED  evt: PEER_CONNECT_REQ
14:38:22.109 Security Manager: l2cap_access_req PSM:19 Handle:42 State:0 Flags:0x88 Required:0x80
14:38:22.109 btm_sec_execute_procedure: Required:0x80 Flags:0x88 State:0
14:38:22.109 Security Manager: access granted
14:38:22.109 L2CAP - LCID: 0x0040  st: OPEN  evt: SECURITY_COMPLETE
14:38:22.109 L2CAP - LCID: 0x0041  st: TERM_W4_SEC_COMP  evt: SECURITY_COMPLETE
14:38:22.109 L2CAP - Calling Connect_Ind_Cb(), CID: 0x0041
14:38:22.109 l2cif_connect_ind PSM:19 lcid:0x41
14:38:22.109 hidh_l2cif_connect_ind interrupt
14:38:22.109 BTKRNLL2CAPConnectRsp lcid:0x41
14:38:22.109 BTKRNLL2CAPConfigReq lcid:0x41
14:38:22.109 L2CA_ConnectRsp()  CID: 0x0041  Result: 0  Status: 0
14:38:22.109 L2CA_ConnectRsp()  BDA: 00-19-c1-e2-62-f8
14:38:22.109 L2CAP - LCID: 0x0041  st: W4_L2CA_CON_RSP  evt: UPPER_LAYER_CONNECT_RSP
14:38:22.109 L2CAP SENT Command.  Name: L2C Connection Response (0x03) ID 0x03, len 8
14:38:22.109           Destination CID : 0x41
14:38:22.109             Source CID : 0x2ba
14:38:22.109               Result : 0x0
14:38:22.109               Status : 0x0
14:38:22.109 --
14:38:22.109 L2CA_ConfigReq()  CID: 0x0041
14:38:22.109 L2CAP - LCID: 0x0041  st: CONFIGURATION  evt: UPPER_LAYER_CONFIGURATION_REQ
14:38:22.109 L2CAP SENT Command.  Name: L2C Configuration Request (0x04) ID 0x02, len 12
14:38:22.109           Destination CID : 698 (0x02ba)
14:38:22.109         Flags (last request) : 0x0000
14:38:22.109           Option type 1 : a0 02
14:38:22.109           Option type 2 : ff ff
14:38:22.109 --
14:38:22.156 L2CAP RECV Command.  Name: L2C Configuration Request (0x04) ID: 0x04, Len: 28
14:38:22.156           Destination CID : 65 (0x0041)
14:38:22.156         Flags (last request) : 0x0000
14:38:22.156           Option type 3 : 00 02 00 00 00 00 00 00 00 00 00 00 00 00 f2 2b
14:38:22.156                0 : 00 00 ff ff ff ff
14:38:22.156 --
```

14:38:22.156 L2CAP - LCID: 0x0041 st: CONFIGURATION evt: PEER_CONFIGURATION_REQ
14:38:22.156 L2CAP - Calling Configuration_Req_Cb(), CID: 0x0041
14:38:22.156 l2cif_configuration_ind lcid:0x41
14:38:22.156 BTKRNLL2CAPConfigRsp lcid:0x41
14:38:22.156 hidh_l2cif_configuration_ind conn_flags:0xe
14:38:22.156 L2CA_ConfigRsp() CID: 0x0041 Result: 0
14:38:22.156 L2CAP - LCID: 0x0041 st: CONFIGURATION evt: UPPER_LAYER_CONFIGURATION_RSP
14:38:22.156 L2CAP SENT Command. Name: L2C Configuration Response (0x05) ID 0x04, len 30
14:38:22.156 Source CID : 698 (0x02ba)
14:38:22.156 Flags (last response) : 0x0000
14:38:22.156 Result : 0 (0x0000)
14:38:22.156 Option type 3 : 00 02 00 00 00 00 00 00 00 00 00 00 00 00 f2 2b
14:38:22.156 0 : 00 00 ff ff ff ff
14:38:22.156 –
14:38:22.171 L2CAP RECV Command. Name: L2C Configuration Response (0x05) ID: 0x02, Len: 6
14:38:22.171 Source CID : 65 (0x0041)
14:38:22.171 Flags (last response) : 0x0000
14:38:22.171 Result : 0 (0x0000)
14:38:22.171 –
14:38:22.171 L2CAP - LCID: 0x0041 st: CONFIGURATION evt: PEER_CONFIGURATION_RSP
14:38:22.171 L2CAP - Calling Configuration_Rsp_Cb(), CID: 0x0041
14:38:22.171 l2cif_configuration_cfm lcid:0x41
14:38:22.171 hidh_l2cif_configuration_cfm conn_flags:0x1e
14:38:22.171 –
14:38:22.171 SENT Command to HCI. Name: HCI_Write_Link_Supervision_Timeout (Hex Code: 0x0c37 Param Len: 4)
14:38:22.171 Parameters
14:38:22.171 Connection Handle : 42 (0x002a)
14:38:22.171 Timeout (625us units) : 11200 (0x2bc0)
14:38:22.171 –
14:38:22.171 Sending HID Connect status 255

14:38:22.187 –
14:38:22.187 RCVD Event from HCI. Name: HCI_Command_Complete (Hex Code: 0x0e Param Len: 6)
14:38:22.187 Parameters
14:38:22.187 Num HCI Cmd Packets : 1 (0x01)
14:38:22.187 Cmd Code : 0x0c37 (HCI_Write_Link_Supervision_Timeout)
14:38:22.187 Status : Success (0x00)
14:38:22.187 Connection Handle : 42 (0x002a)
14:38:22.187 –
14:38:22.187 –
14:38:22.187 SENT Command to HCI. Name: HCI_QoS_Setup (Hex Code: 0x0807 Param Len: 20)
14:38:22.187 Parameters
14:38:22.187 Connection Handle : 42 (0x002a)
14:38:22.187 Flags (reserved) : 0 (0x00)
14:38:22.187 Service Type : 2 (0x02)
14:38:22.187 Token Rate : 0 (0x00000000)
14:38:22.187 Peak Bandwidth : 0 (0x00000000)
14:38:22.187 Latency : 10000 (0x00002710)
14:38:22.187 Delay Variation : 4294967295 (0xffffffff)
14:38:22.187 –
14:38:22.187 –
14:38:22.187 RCVD Event from HCI. Name: HCI_Command_Status (Hex Code: 0x0f Param Len: 4)
14:38:22.187 Parameters
14:38:22.187 Status : Success (0x00)
14:38:22.187 Num HCI Cmd Packets : 1 (0x01)
14:38:22.187 Cmd Code : 0x0807 (HCI_QoS_Setup)
14:38:22.187 –
14:38:22.218 –
14:38:22.218 RCVD Event from HCI. Name: HCI_QoS_Setup_Complete (Hex Code: 0x0d Param Len: 21)
14:38:22.218 Parameters

14:38:22.218 Status : Success (0x00)
14:38:22.218 Connection Handle : 42 (0x002a)
14:38:22.218 Flags (reserved) : 0 (0x00)
14:38:22.218 Service Type : 2 (0x02)
14:38:22.218 Token Rate : 0 (0x00000000)
14:38:22.218 Peak Bandwidth : 0 (0x00000000)
14:38:22.218 Latency : 10000 (0x00002710)
14:38:22.218 Delay Variation : 4294967295 (0xffffffff)
14:38:22.218 –
14:38:22.312 HID Connected not installed unknown device: BD_ADDR:00:19:c1:e2:62:f8
14:38:22.312 BTKRNLL2CAPHidDisconnect psm:0x11
14:38:22.312 L2CA_DataWrite() CID: 0x0040 Len: 1
14:38:22.312 L2CAP - LCID: 0x0040 st: OPEN evt: UPPER_LAYER_DATA_WRITE
14:38:22.312 HID Control SENT - Control code:Virtual Cable Unplug
14:38:22.312 hidh_control_device ctrlType:5 lcid:0x40
14:38:22.312 CBtComIf::DelyedDeviceDevFoundCB: bConnected=1 bda=f862e2c11900
14:38:22.312 CBtComIf::DelyedDeviceAdd: updateing record. bda=f862e2c11900
14:38:22.390 L2CAP RECV Command. Name: L2C Disconnection Request (0x06) ID: 0x05, Len: 4
14:38:22.390 Destination CID : 0x41
14:38:22.390 Source CID : 0x2ba
14:38:22.390 –
14:38:22.390 L2CAP - LCID: 0x0041 st: OPEN evt: PEER_DISCONNECT_REQ
14:38:22.390 L2CAP - Calling Disconnect_Ind_Cb(), CID: 0x0041 Conf Needed
14:38:22.390 l2cif_disconnect_ind lcid:0x41 ack:1
14:38:22.390 BTKRNLL2CAPDisconnectRsp lcid:0x41
14:38:22.390 L2CA_DisconnectRsp() CID: 0x0041
14:38:22.390 L2CAP - LCID: 0x0041 st: W4_L2CA_DISC_RSP evt: UPPER_LAYER_DISCONNECT_RSP
14:38:22.390 L2CAP SENT Command. Name: L2C Disconnection Response (0x07) ID 0x05, len 4
14:38:22.390 Destination CID : 0x41
14:38:22.390 Source CID : 0x2ba
14:38:22.390 –
14:38:22.390 L2CAP RECV Command. Name: L2C Disconnection Request (0x06) ID: 0x06, Len: 4
14:38:22.390 Destination CID : 0x40
14:38:22.390 Source CID : 0x2b9
14:38:22.390 –
14:38:22.390 L2CAP - LCID: 0x0040 st: OPEN evt: PEER_DISCONNECT_REQ
14:38:22.390 L2CAP - Calling Disconnect_Ind_Cb(), CID: 0x0040 Conf Needed
14:38:22.390 l2cif_disconnect_ind lcid:0x40 ack:1
14:38:22.390 BTKRNLL2CAPDisconnectRsp lcid:0x40
14:38:22.390 L2CA_DisconnectRsp() CID: 0x0040
14:38:22.390 L2CAP - LCID: 0x0040 st: W4_L2CA_DISC_RSP evt: UPPER_LAYER_DISCONNECT_RSP
14:38:22.390 L2CAP SENT Command. Name: L2C Disconnection Response (0x07) ID 0x06, len 4
14:38:22.390 Destination CID : 0x40
14:38:22.390 Source CID : 0x2b9
14:38:22.390 –
14:38:22.734 –
14:38:22.734 RCVD Event from HCI. Name: HCI_Disconnection_Complete (Hex Code: 0x05 Param Len: 4)
14:38:22.734 Parameters
14:38:22.734 Status : Success (0x00)
14:38:22.734 Connection Handle : 42 (0x002a)
14:38:22.734 Reason : 19 (0x13)
14:38:22.734 Other End Terminated Connection: User Ended Connection
14:38:22.734 –
14:38:22.734 btm_sco_acl_removed -> bda 00:19:c1:e2:62:f8
14:38:22.734 btm_sco_acl_removed -> bda 00:19:c1:e2:62:f8
14:38:22.734 SetBusyLevel action:2 level:0

### Sixaxis sur *stack* Microsoft, sans serveur

Le *log* est ici différent du précédent, car un outil différent a été utilisé pour l'obtenir. Il n'y a rien d'intéressant à remarquer, si ce n'est que la *stack* Microsoft accepte la connexion lorsque l'on tente d'y connecter la Sixaxis. Les réponses de configuration standard s'y trouvent également, mais la gestion s'arrête là et aucun profile HID n'est chargé :

| | | | | | | |
|---|---|---|---|---|---|---|
| *Master* | 44 | 0x0001 | HID_Control | Connection request | 0x0281 | |
| *Slave* | 44 | 0x0001 | | Connection response | 0x0000 | 0x0000 |
| *Master* | 45 | 0x0001 | HID_Control | Connection request | 0x0282 | |
| *Slave* | 45 | 0x0001 | | Connection response | 0x0000 | 0x0000 |
| *Master* | 46 | 0x0001 | HID_Control | Connection request | 0x0283 | |
| *Slave* | 46 | 0x0001 | | Connection response | 0x0000 | 0x0000 |
| *Master* | 47 | 0x0001 | HID_Control | Connection request | 0x0284 | |
| *Slave* | 47 | 0x0001 | | Connection response | 0x0000 | 0x0000 |
| *Master* | 48 | 0x0001 | HID_Control | Connection request | 0x0285 | |
| *Slave* | 48 | 0x0001 | | Connection response | 0x0000 | 0x0000 |
| *Master* | 47 | 0x0001 | HID_Interrupt | Connection request | 0x029c | |
| *Slave* | 47 | 0x0001 | | Connection response | 0x029c | 0x0000 |
| *Slave* | 47 | 0x0001 | | Connection response | 0x029c | 0x0041 |
| *Slave* | 47 | 0x0001 | | Configure request | | 0x029c |
| *Master* | 47 | 0x0001 | | Configure request | | 0x0041 |
| *Slave* | 47 | 0x0001 | | Configure response | 0x029c | |
| *Master* | 47 | 0x0001 | | Configure response | 0x0041 | |
| *Master* | 47 | 0x0001 | | Disconnection request | 0x029c | 0x0041 |
| *Master* | 47 | 0x0001 | | Disconnection request | 0x029c | 0x0041 |
| *Slave* | 47 | 0x0001 | | Disconnection request | 0x0040 | 0x029b |
| *Master* | 47 | 0x0001 | | Disconnection response | 0x0040 | 0x029b |

### Sixaxis sur *stack* Bluesoleil, sans serveur

La *stack* Bluesoleil a un comportement semblable à celle de chez Microsoft, si ce n'est qu'elle accepte par défaut les connexions L2CAP entrantes moyennant quelques réglages. Aucun profile HID n'est toutefois chargé.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Master* | 42 | 0x0001 | HID_Control | Connection request | 0x02f9 | |
| *Slave* | 42 | 0x0001 | | Connection response | 0x02f9 | 0x0040 |
| *Master* | 43 | 0x0001 | HID_Control | Connection request | 0x02fa | |
| *Slave* | 43 | 0x0001 | | Connection response | 0x02fa | 0x0041 |
| *Master* | 44 | 0x0001 | HID_Control | Connection request | 0x02fb | |
| *Slave* | 44 | 0x0001 | | Connection response | 0x02fb | 0x0042 |
| *Master* | 45 | 0x0001 | HID_Control | Connection request | 0x02fc | |
| *Slave* | 45 | 0x0001 | | Connection response | 0x02fc | 0x0043 |
| *Master* | 46 | 0x0001 | HID_Control | Connection request | 0x02fd | |
| *Slave* | 46 | 0x0001 | | Connection response | 0x02fd | 0x0044 |
| *Master* | 47 | 0x0001 | HID_Control | Connection request | 0x02fe | |
| *Slave* | 47 | 0x0001 | | Connection response | 0x02fe | 0x0045 |
| *Master* | 48 | 0x0001 | HID_Control | Connection request | 0x02ff | |
| *Slave* | 48 | 0x0001 | | Connection response | 0x02ff | 0x0046 |
| *Master* | 42 | 0x0001 | HID_Control | Connection request | 0x0300 | |
| *Slave* | 42 | 0x0001 | | Connection response | 0x0300 | 0x0047 |
| *Master* | 43 | 0x0001 | HID_Control | Connection request | 0x0301 | |
| *Slave* | 43 | 0x0001 | | Connection response | 0x0301 | 0x0048 |
| *Master* | 44 | 0x0001 | HID_Control | Connection request | 0x0302 | |
| *Slave* | 44 | 0x0001 | | Connection response | 0x0302 | 0x0049 |
| *Master* | 45 | 0x0001 | HID_Control | Connection request | 0x0303 | |
| *Slave* | 45 | 0x0001 | | Connection response | 0x0303 | 0x004a |
| *Master* | 46 | 0x0001 | HID_Control | Connection request | 0x0304 | |
| *Slave* | 46 | 0x0001 | | Connection response | 0x0304 | 0x004b |
| *Master* | 47 | 0x0001 | HID_Control | Connection request | 0x0305 | |
| *Slave* | 47 | 0x0001 | | Connection response | 0x0305 | 0x004c |
| *Master* | 48 | 0x0001 | HID_Control | Connection request | 0x0307 | |
| *Slave* | 48 | 0x0001 | | Connection response | 0x0307 | 0x004e |
| *Master* | 49 | 0x0001 | HID_Control | Connection request | 0x0308 | |

# B. Codes exemples

## Sixpair pour Win32

```
/*
 * sixpair.c
 * Code originale disponble sur www.pabr.org
 * Ceci est une adptation pour système WIN32
 * Ce code sert à appairer une manette Sixaxis
 * au module Bluetooth dont l'adresse MAC est passe
 * en paramètre.
 *
 * Nécessite la librairire libusb pour Windows
 */

#include <string.h>
#include <io.h>
#include <stdio.h>
#include "usb.h"

#define VENDOR 0x054c
#define PRODUCT 0x0268

#define USB_DIR_IN 0x80
#define USB_DIR_OUT 0

void fatal(char *msg) { perror(msg); exit(1); }

void show_master(usb_dev_handle *devh, int itfnum) {
  int res;
  unsigned char msg[8];
  printf("Current Bluetooth master: ");
  res = usb_control_msg    (devh, USB_DIR_IN | USB_TYPE_CLASS | USB_RECIP_INTERFACE,
     0x01, 0x03f5, itfnum, (void*)msg, sizeof(msg), 5000);
  if ( res < 0 ) { perror("USB_REQ_GET_CONFIGURATION"); return; }
  printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
         msg[2], msg[3], msg[4], msg[5], msg[6], msg[7]);
}

void set_master(usb_dev_handle *devh, int itfnum, int MAC[6]) {
  char msg[8]= { 0x01, 0x00, MAC[0],MAC[1],MAC[2],MAC[3],MAC[4],MAC[5] };
  int res;
  printf("Setting master bd_addr to %02x:%02x:%02x:%02x:%02x:%02x\n",
         MAC[0], MAC[1], MAC[2], MAC[3], MAC[4], MAC[5]);
  res = usb_control_msg
    (devh,
     USB_DIR_OUT | USB_TYPE_CLASS | USB_RECIP_INTERFACE,
     0x09,
     0x03f5, itfnum, msg, sizeof(msg),
     5000);
  if ( res < 0 ) fatal("USB_REQ_SET_CONFIGURATION");
}

void process_device(int argc, char **argv, struct usb_device *dev,
                    struct usb_configuration_descriptor *cfg, int itfnum) {
  int MAC[6], have_MAC=0, res;

  usb_dev_handle *devh = usb_open(dev);
  if ( ! devh ) fatal("usb_open");

  /* usb_detach_kernel_driver_np(devh, itfnum);

  res = usb_claim_interface(devh, itfnum);
  if ( res < 0 ) fatal("usb_claim_interface"); */

  show_master(devh, itfnum);

  if ( argc >= 2 ) {
    if ( sscanf(argv[1], "%x:%x:%x:%x:%x:%x",
                &MAC[0],&MAC[1],&MAC[2],&MAC[3],&MAC[4],&MAC[5]) != 6 ) {

      printf("usage: %s [<bd_addr of master>]\n", argv[0]);
      exit(1);
    }
  }
```

```
     else {
    FILE *f = NULL; //popen("hcitool dev", "r");
    if ( !f ||
         fscanf(f, "%*s\n%*s %x:%x:%x:%x:%x:%x",
                &MAC[0],&MAC[1],&MAC[2],&MAC[3],&MAC[4],&MAC[5]) != 6 ) {
      printf("Unable to retrieve local bd_addr from `hcitool dev`.\n");
      printf("Please enable Bluetooth or specify an address manually.\n");
      exit(1);
    }
    //pclose(f);
  }

  set_master(devh, itfnum, MAC);

  usb_close(devh);
}

int main(int argc, char *argv[]) {
  struct usb_bus *busses;
  struct usb_bus *bus;
  int found;
  usb_init();
  if ( usb_find_busses() < 0 ) fatal("usb_find_busses");
  if ( usb_find_devices() < 0 ) fatal("usb_find_devices");
  busses = usb_get_busses();
  if ( ! busses ) fatal("usb_get_busses");

  found = 0;


  for ( bus=busses; bus; bus=bus->next ) {
    struct usb_device *dev;
    for ( dev=bus->devices; dev; dev=dev->next) {
      struct usb_configuration_descriptor *cfg;
      for ( cfg = dev->configuration;
            cfg < dev->configuration + dev->descriptor.bNumConfigurations;
            ++cfg ) {
        int itfnum;
        for ( itfnum=0; itfnum<cfg->bNumInterfaces; ++itfnum ) {
          struct usb_interface *itf = &cfg->interface[itfnum];
          struct usb_interface_descriptor *alt;
          for ( alt = itf->altsetting;
                alt < itf->altsetting + itf->num_altsetting;
                ++alt ) {
            if ( dev->descriptor.idVendor == VENDOR &&
                 dev->descriptor.idProduct == PRODUCT &&
                 alt->bInterfaceClass == 3 ) {
              process_device(argc, argv, dev, cfg, itfnum);
              ++found;
            }
          }
        }
      }
    }
  }

  if ( ! found ) printf("No controller found on USB busses.\n");
  return 0;

}
```

## Serveur Sixaxis avec *stack* Microsoft

```
#include <winsock2.h>
#include <ws2bth.h>
#include <strsafe.h>
#include <initguid.h>
#include <stdio.h>
#include "Bthsdpdef.h"
#include "BluetoothAPIs.h"
#include <windows.h>


/*
* Serveur Bluetooth pour Sixaxis
* Etant donné que l'on ne peut pas ouvrir de socket L2CAP,
* ce code ne fonctionne pas avec la Sixaxis mais peut servir
* d'excellent exemple d'utilisation du SDK Bluetooth de Microsoft.
```

```
*
* Nécessite les librairies Bthprops.lib WS2_32.Lib
*/

const int iReqWinsockVer = 1;
DEFINE_GUID(g_guidServiceClass, 0xb62c4e8d, 0x62cc, 0x404b, 0xbb, 0xbf, 0xbf, 0x3e, 0x3b, 0xbb, 0x13,
0x74);


int main() {

        WSADATA wsaData;
        PWSAQUERYSET     pWSAQuerySet = NULL;
        ULONG ulFlags = 0;
        HANDLE hLookup = 0;
        int suivant = 1;
        ULONG ulPQSSize = sizeof(WSAQUERYSET);


        // Allocation de la mémoir de pWSAQuerySet

        pWSAQuerySet = (PWSAQUERYSET) HeapAlloc(GetProcessHeap(),
                HEAP_ZERO_MEMORY,
                ulPQSSize);

        if ( NULL == pWSAQuerySet ) {
                printf("!ERROR! | Unable to allocate memory for WSAQERYSET\n");
        }


        pWSAQuerySet->dwNameSpace = NS_BTH;
        pWSAQuerySet->dwSize = ulPQSSize;

        //starting wsa
        if (WSAStartup(MAKEWORD(iReqWinsockVer,0),&wsaData)==0) {

                printf("Wsa started \n");
        }
        else {

                printf("Wsa failed \n");
        }

        printf("Discovery on %d \n",BluetoothEnableDiscovery(NULL,true));

        // On ne veut que les devices, pas les services
        ulFlags = LUP_CONTAINERS;

        // Que le nom
        ulFlags |= LUP_RETURN_NAME;

        // Et l'adresse
        ulFlags |= LUP_RETURN_ADDR;

        // Et le COD (class of device)
        ulFlags |= LUP_RETURN_TYPE;

        if (WSALookupServiceBegin(pWSAQuerySet, ulFlags, &hLookup)!=0) {

                printf("Erreur wsalookupservicebegin: [%d] \n", WSAGetLastError());
                exit(2);
        }
        else printf("WsalookupServiceBegin lance \n");
        printf("Memoire a dispo : %d \n",ulPQSSize);

        int iResult=0;
        while(suivant){
                // Recherche des (un seul en fait) périphériques
                if (0==WSALookupServiceNext(hLookup, ulFlags, &ulPQSSize, pWSAQuerySet))
                {
                        if (pWSAQuerySet->lpszServiceInstanceName != NULL ) {

                                printf("DEVICE FOUNDED \n");
                                wprintf(pWSAQuerySet->lpszServiceInstanceName);
                                printf("\nCOD : %d \n",pWSAQuerySet->lpServiceClassId->Data1);
                                printf("\n");
                                printf("-----------------------------------\n");
                        }
                }

                else {
```

```
                if ( WSA_E_NO_MORE == ( iResult = WSAGetLastError() ) ) { //No more data
                        printf("Plus de device\n");
                        suivant = 0;
                }
                else if ( WSAEFAULT == iResult ) {

                        printf("Memoire insuffisante, reallocation : %d\n",ulPQSSize);
                        HeapFree(GetProcessHeap(), 0, pWSAQuerySet);
                        pWSAQuerySet = (PWSAQUERYSET) HeapAlloc(GetProcessHeap(),
                                HEAP_ZERO_MEMORY,
                                ulPQSSize);

                        if ( NULL == pWSAQuerySet ) {
                        printf("!ERROR! | Unable to allocate memory for WSAQERYSET\n");
                                suivant = 0;
                        }
                }

        }
}
if (WSALookupServiceEnd(hLookup)!=0) {

        printf("WSALookupServiceEnd failed : %d \n",WSAGetLastError());

}
else printf("WSALookupServiceEnd successfull\n");
//#########################################
// SDP PART
//#########################################

pWSAQuerySet->dwSize=sizeof(WSAQUERYSET);

//#########################################
//SOCKET PART
//#########################################

SOCKET LocalSocket = INVALID_SOCKET;
SOCKADDR_BTH     SockAddrBthLocal = {0};
LPCSADDR_INFO    lpCSAddrInfo = NULL;
SOCKET                   ClientSocket = INVALID_SOCKET;

lpCSAddrInfo = (LPCSADDR_INFO) HeapAlloc( GetProcessHeap(),
        HEAP_ZERO_MEMORY,
        sizeof(CSADDR_INFO) );
if ( NULL == lpCSAddrInfo ) {
        printf("!ERROR! | Unable to allocate memory for CSADDR_INFO\n");
        exit(2);
}

LocalSocket = socket( AF_BTH, SOCK_DGRAM, BTHPROTO_L2CAP );
if ( INVALID_SOCKET == LocalSocket ) {
        printf("Creation de socket failed :  [%d]\n", WSAGetLastError());
        exit(3);
}
else printf("Socket cree\n");

//Configuration et Bind du socket
SockAddrBthLocal.addressFamily = AF_BTH;
SockAddrBthLocal.btAddr = 0;
//SockAddrBthLocal.serviceClassId  = NUL_LGUID;
SockAddrBthLocal.port = BT_PORT_ANY;

if ( SOCKET_ERROR == bind(LocalSocket,
        (struct sockaddr *) &SockAddrBthLocal,
        sizeof(SOCKADDR_BTH) ) ) {
        printf("Bind failed = [0x%X]. WSAGetLastError=[%d]\n", LocalSocket, WSAGetLastError());
                exit(2);
}
else printf("Bind ok\n");
//
// CSADDR_INFO
//
lpCSAddrInfo[0].LocalAddr.iSockaddrLength = sizeof( SOCKADDR_BTH );
lpCSAddrInfo[0].LocalAddr.lpSockaddr = (LPSOCKADDR)&SockAddrBthLocal;
lpCSAddrInfo[0].RemoteAddr.iSockaddrLength = sizeof( SOCKADDR_BTH );
lpCSAddrInfo[0].RemoteAddr.lpSockaddr = (LPSOCKADDR)&SockAddrBthLocal;
lpCSAddrInfo[0].iSocketType = SOCK_STREAM;
lpCSAddrInfo[0].iProtocol = BTHPROTO_RFCOMM;

if ( SOCKET_ERROR == listen(LocalSocket, SOMAXCONN) ) {
```

```
                printf("=CRITICAL= | listen() call failed w/socket = [0x%X]. WSAGetLastError=[%d]\n",
LocalSocket, WSAGetLastError());
                exit(2);
        }
        else printf("Listening...");

        //Accepting mode
        if ( INVALID_SOCKET == ( ClientSocket = accept(LocalSocket, NULL, NULL) ) ) {
                printf("=CRITICAL= | accept() call failed. WSAGetLastError=[%d]\n", WSAGetLastError());
                exit(2);
        }
        else printf("and accept.\n");
        if (WSACleanup()!=0) {

                printf("Cleanup failed\n");
        }
        else printf("Cleanup complete\n");
        return 0;

}
```

## Serveur Bluetooth avec *stack* Widcomm

```
#include <stdio.h>
#include <windows.h>
#include "Btwlib.h"

/*
* Comme le code précédent, celui-ci ouvre un serveur
* écoutant sur un PSM donné. Cette-fois, la connexion est de type
* L2CAP et est parfaitement fonctionnelle avec des PSM > 1000
*
* Nécessite les librairies btwdsdklib.lib ws2_32.lib version.lib
*/

UINT8 JOY_NAME[128] = "PLAYSTATION(R)3 Controller";
bool discovery = false;
bool joy_discovery = false;
bool incoming = false;
UINT8 joy_add[6];
static char m_serviceName[BT_MAX_SERVICE_NAME_LEN+1] = "HID";

GUID service_guid = CBtIf::guid_SERVCLASS_HUMAN_INTERFACE;
//static GUID service_guid = CBtIf.guid_SERVCLASS_NAP;
//GUID service_guid = 0x13;


int data_bidon = 0xDEADBEEF;
UINT16 length = 32;
UINT16 length_written;


class CBprout: public CBtIf {
        void OnDeviceResponded(BD_ADDR bda, DEV_CLASS devClass, BD_NAME bdName, BOOL bConnected) {
                /*printf("Trouve \n");
                printf("BD_NAME : %s\n",bdName);
                printf("Adresse : 0x%x%x%x%x%x%x\n",bda[0],bda[1],bda[2],bda[3],bda[4],bda[5]);*/

                if(!strcmp((char*)bdName,"PLAYSTATION(R)3 Controller")) {

                        printf("Joy ps3 detecte\n");
                        joy_discovery = true;
                        for(int i=0;i<6;i++) {
                                joy_add[i]=bda[i];
                        }
                }
        }

        void OnInquiryComplete(BOOL success, short num_responses) {
                printf("Inquiry complete : %d \n",num_responses);
                printf("Joy ps3 adresse :
0x%x%x%x%x%x%x\n",joy_add[0],joy_add[1],joy_add[2],joy_add[3],joy_add[4],joy_add[5]);
        }

        void OnDiscoveryComplete(UINT16 nRecs, long lResultCode){
                printf("Discovery Complete \n");
                if (lResultCode==WBT_SUCCESS) {
```

```
                        printf("Discovery complete\n");
                        discovery=true;
                }
                else printf("Discovery failed\n");
        }
};


class CL2conn: public CL2CapConn {
public:
        CL2conn()
        {
                printf("Prout\n");
        }

        void OnIncomingConnection() {
                printf("Incoming\n");
                incoming = true;
                printf("Accepting : %d \n",this->Accept());
        }

        void OnConnected() {
                printf("YIPEE Connected\n");
        }

        void OnConnectPendingReceived() {
                printf("Connected pending\n");
        }

        void OnDataReceived(void *p_data, UINT16 length) {
                printf("Data received \n");
                printf("Data : %p\n",p_data);
        }

        // Client and server may provide a method to be notified
        // when a connection is disconnected.
        //
        void OnRemoteDisconnected (UINT16 reason){
                printf("remote disconntected\n");
        }
};


void callback_func()
{
        printf("Prout\n");
}

int main() {

        CBprout plop;
        CL2CapIf base_lf;
        CL2conn base_conn;
        CSdpService service_sdp;
        SDP_RETURN_CODE sdp_response;

        /*printf("Starting inquiry...\n");
        if(!plop.StartInquiry()) {
        printf("Start inquiry failed\n");
        }*/

        //printf("Waiting on ps3 joy discovery...\n");
        //while(!joy_discovery) {}
        //printf("Ps3 joy discovered, launching server\n");
        //printf("Setting timeout to 0\n");
        //if(!plop.SetLinkSupervisionTimeOut(joy_add,0)) {
        //      printf("SetLinkSupervision Timeout failed\n");
        //fds
        //};
        //UINT16 psm = 0x13;

        //if(!base_lf.AssignPsmValue((GUID*)&service_guid),psm)     {
        if(!base_lf.AssignPsmValue(((GUID*)&service_guid),0x11)){
                printf("Psm assignment error \n");
        }
        else printf("Psm ok \n");

        printf("Psm : 0x%x \n",base_lf.GetPsm());
```

```
        if (!base_lf.Register()) {
                printf("Register error \n");
        }
        else printf("Register ok \n");

        if (!base_lf.SetSecurityLevel(m_serviceName, BTM_SEC_NONE, true)) {
                printf("Security error \n");
        }
        else printf("Security ok \n");

        if (!base_conn.Listen(&base_lf)) {
                printf("Listening error \n");
        }
        else printf("Listening... \n");

        unsigned char sixaxis_add[] = {0x0, 0x19, 0xc1, 0xe2, 0x62, 0xf8};

        while(!incoming) {

        }

        printf("Connection accepted \n");

        while(true);

        return 0;

}
```

## *Stack* java : *discovery*

```
import java.io.IOException;
import java.util.Vector;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;

/*
* Ce code est un exemple trouvé sur internet qui effectue un discovery.
*/
/**
 * Class that discovers all bluetooth devices in the neighbourhood
 * and displays their name and bluetooth address.
 */

public class BluetoothDeviceDiscovery implements DiscoveryListener{


        private static Object lock=new Object();

        //vector containing the devices discovered

        private static Vector vecDevices=new Vector();

        //main method of the application

        public static void main(String[] args) throws IOException {

        //create an instance of this class

        BluetoothDeviceDiscovery bluetoothDeviceDiscovery=new BluetoothDeviceDiscovery();
        //display local device address and name
        LocalDevice localDevice = LocalDevice.getLocalDevice();
        System.out.println("Address: "+localDevice.getBluetoothAddress());
        System.out.println("Name: "+localDevice.getFriendlyName());
        //find devices
        DiscoveryAgent agent = localDevice.getDiscoveryAgent();
        System.out.println("Starting device inquiry...");
        agent.startInquiry(DiscoveryAgent.GIAC, bluetoothDeviceDiscovery);
        try {
                synchronized(lock){
                        lock.wait();
                }
        }
        catch (InterruptedException e) {
```

```
                e.printStackTrace();
        }
        System.out.println("Device Inquiry Completed. ");

        //print all devices in vecDevices
        int deviceCount=vecDevices.size();
        if(deviceCount <= 0){
                System.out.println("No Devices Found .");
        }
        else{
                //print bluetooth device addresses and names in the format [ No. address (name) ]
                System.out.println("Bluetooth Devices: ");
                for (int i = 0; i <deviceCount; i++) {
                        RemoteDevice remoteDevice=(RemoteDevice)vecDevices.elementAt(i);
                        System.out.println((i+1)+". "+remoteDevice.getBluetoothAddress()+"
("+remoteDevice.getFriendlyName(true)+")");
                }

        }

        }//end main

        //methods of DiscoveryListener

        /**
         * This call back method will be called for each discovered bluetooth devices.
         */
        public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {

                System.out.println("Device discovered: "+btDevice.getBluetoothAddress());
                //add the device to the vector
                if(!vecDevices.contains(btDevice)){
                        vecDevices.addElement(btDevice);
                }
        }


        //no need to implement this method since services are not being discovered

        public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {

        }
        //no need to implement this method since services are not being discovered
        public void serviceSearchCompleted(int transID, int respCode) {
        }
        /**
         * This callback method will be called when the device discovery is
         * completed.
         */
        public void inquiryCompleted(int discType) {
                synchronized(lock){
                        lock.notify();
                }
                switch (discType) {
                case DiscoveryListener.INQUIRY_COMPLETED :
                        System.out.println("INQUIRY_COMPLETED");
                        break;
                case DiscoveryListener.INQUIRY_TERMINATED :
                        System.out.println("INQUIRY_TERMINATED");
                        break;
                case DiscoveryListener.INQUIRY_ERROR :
                        System.out.println("INQUIRY_ERROR");
                        break;
                default :
                        System.out.println("Unknown Response Code");
                break;
                }
        }//end method
}//end class
```

## *Stack* java : serveur

```
package bluerichon;

import java.io.IOException;
import java.util.Vector;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
```

```
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.L2CAPConnectionNotifier;
import javax.bluetooth.L2CAPConnection;
import javax.bluetooth.UUID;
import javax.microedition.io.Connection;
import javax.microedition.io.Connector;

/*
* Ceci est également du code exemple, mais adapté pour concevoir
* mais adapté pour servir de serveur Bluetooth. Il supporte les sockets
* selon la stack installée (Microsoft : RFCOMM, Widcomm : RFCOMM et L2CAP)
*/

/**
 * Class that discovers all bluetooth devices in the neighbourhood
 * and displays their name and bluetooth address.
 */

public class DiscoveryTester implements DiscoveryListener{


        private static Object lock=new Object();

        //vector containing the devices discovered

        private static Vector vecDevices=new Vector();

        //main method of the application

        public static void main(String[] args) throws IOException {

                //L2CAPConnectionNotifier notifier =  (L2CAPConnectionNotifier)
Connector.open("btl2cap://localhost:3B9FA89520078C303355AAA694238F07;ReceiveMTU=512;TransmitMTU=512");
                L2CAPConnectionNotifier notifier =  (L2CAPConnectionNotifier)
Connector.open("btl2cap://localhost:" + new UUID(256).toString() + ";ReceiveMTU=512;TransmitMTU=512");
                System.out.println("prout");
                L2CAPConnection connect = notifier.acceptAndOpen();

                System.out.println("prout2");

                //create an instance of this class

                DiscoveryTester bluetoothDeviceDiscovery=new DiscoveryTester();
                //display local device address and name
                LocalDevice localDevice = LocalDevice.getLocalDevice();
                System.out.println("Address: "+localDevice.getBluetoothAddress());
                System.out.println("Name: "+localDevice.getFriendlyName());
                //find devices
                DiscoveryAgent agent = localDevice.getDiscoveryAgent();

                System.out.println("Starting device inquiry...");
                agent.startInquiry(DiscoveryAgent.GIAC, bluetoothDeviceDiscovery);

                try {
                        synchronized(lock){
                                lock.wait();
                        }
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }
                System.out.println("Device Inquiry Completed. " + agent.selectService(new UUID(0x0100),
ServiceRecord.NOAUTHENTICATE_NOENCRYPT, true));
                //print all devices in vecDevices
                int deviceCount=vecDevices.size();
                if(deviceCount <= 0){
                        System.out.println("No Devices Found .");
                }
                else{
                        //print bluetooth device addresses and names in the format [ No. address (name)
]
                        System.out.println("Bluetooth Devices: ");
                        for (int i = 0; i <deviceCount; i++) {
                                RemoteDevice remoteDevice=(RemoteDevice)vecDevices.elementAt(i);
                                System.out.println((i+1)+". "+remoteDevice.getBluetoothAddress()+"
("+remoteDevice.getFriendlyName(true)+")");
                        }
```

```
                }

        }//end main

        //methods of DiscoveryListener

        /**
         * This call back method will be called for each discovered bluetooth devices.
         */
        public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {

                System.out.println("Device discovered: "+btDevice.getBluetoothAddress());
                //add the device to the vector
                if(!vecDevices.contains(btDevice)){
                        vecDevices.addElement(btDevice);
                }
        }


        //no need to implement this method since services are not being discovered

        public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {

        }
        //no need to implement this method since services are not being discovered
        public void serviceSearchCompleted(int transID, int respCode) {
        }
        /**
         * This callback method will be called when the device discovery is
         * completed.
         */
        public void inquiryCompleted(int discType) {
                synchronized(lock){
                        lock.notify();
                }
                switch (discType) {
                case DiscoveryListener.INQUIRY_COMPLETED :
                        System.out.println("INQUIRY_COMPLETED");
                        break;
                case DiscoveryListener.INQUIRY_TERMINATED :
                        System.out.println("INQUIRY_TERMINATED");
                        break;
                case DiscoveryListener.INQUIRY_ERROR :
                        System.out.println("INQUIRY_ERROR");
                        break;
                default :
                        System.out.println("Unknown Response Code");
                break;
                }
        }//end method
}//end class
```

## Pybluez : serveur et client, exemple (Python)

### Client :

```
#!/usr/bin/python

import bluetooth
import sys
import thread
import time

bd_addr = "00:02:5B:01:16:2C"

sock=bluetooth.BluetoothSocket(bluetooth.L2CAP)
port = 0x1001

sock.connect((bd_addr, port))
sock.send("hello!")

sock.close
```

### Serveur :

```
import bluetooth

server_sock=bluetooth.BluetoothSocket( bluetooth.L2CAP )

port = 0x1001
```

```
server_sock.bind(("",port))
server_sock.listen(1)

client_sock,address = server_sock.accept()
print "Accepted connection from ",address

data = client_sock.recv(1024)
print "received [%s]" % data

client_sock.close()
server_sock.close()
```

## Pybluez : connexion Sixaxis

```
#!/usr/bin/python

import bluetooth
import thread
import time

# Sockets necessaires prout HID (2)
control_socket = bluetooth.BluetoothSocket(bluetooth.L2CAP)
data_socket = bluetooth.BluetoothSocket(bluetooth.L2CAP)


# Fonction pour parser l'hexa en String
def hexdump(data):
        hexdump = ""
        for byte in data:
                hexdump += str(byte).encode("hex").upper() + " "
        return "(" + hexdump[0:2] +")" + hexdump[2:-1]

# Thread pour prendre les data entrantes
def receivethread():

        data_socket.settimeout(0.1)
        while 1:
                try:
                        #data = data_socket.recv(23)
                        data = clientD_socket.recv(50)
                        print "Received : " + hexdump(data)
                except bluetooth.BluetoothError:
                        pass
        data_socket.close()
        command_socket.close()

# Fonction pour envoyer des data sur le control_socket
def send_control(data2):
        data = ""
        for datum in data2:
                data += chr(datum)
        clientC_socket.send(data)
        print "Sent on control :    " + hexdump(data)

# Fonction pour envoyer des data sur le data_socket
def send_data(data2):
        clientD_socket.send("salut")
        print "Sent on data    :     " + hexdump(data)

portA = 0x11
portB = 0x13

control_socket.bind(("",portA))
data_socket.bind(("",portB))

control_socket.listen(1)
data_socket.listen(1)

clientC_socket,address = control_socket.accept()
print "Connection C established  from ",address
clientD_socket,address = data_socket.accept()
print "Connection D established from ", address


print "Sending SET_REPORT #2"
send_control([0x53,0xf4,0x42,0x03,0x00,0x00])
receivethread()
```

## *Stack* avec *socket* L2CAP (Bluez) : connexion Sixaxis

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <pthread.h>

#include "sixaxis-status.h"
static uint16_t INT_PSM = 0x13;
static uint16_t COM_PSM = 0x11;
struct sixaxis_LED_rumble* rumble;
int client_command;


void getHID(int *socket) {
        struct sixaxis_status status;

        int i = 0;
        printf("Listener thread created!\n");
        int k = 0;

        while(1) {
                read(*socket, &status, 50);
                // ICI, faire ce qu'on veut des données HID de la sixaxis
        }
}

int main(void){
        pthread_t in_thread, out_thread;
        int server_data,server_command,client_data;
        unsigned char buff[256];
        struct sockaddr_l2 addr_ctl;
        struct sockaddr_l2 addr_int;
        server_data = socket(AF_BLUETOOTH,SOCK_SEQPACKET, BTPROTO_L2CAP);
        server_command = socket(AF_BLUETOOTH,SOCK_SEQPACKET,BTPROTO_L2CAP);
        if (!(server_data && server_command)) {
                printf("Error opening server socket \n");
                exit(-1);
        }

        addr_ctl.l2_family = AF_BLUETOOTH;
        addr_ctl.l2_psm = htobs(COM_PSM) ;
        bacpy(&addr_ctl.l2_bdaddr,BDADDR_ANY);
        addr_int.l2_family = AF_BLUETOOTH;
        addr_int.l2_psm = htobs(INT_PSM);
        bacpy(&addr_int.l2_bdaddr,BDADDR_ANY);
        int addrlen1=sizeof(addr_ctl);
        int addrlen2=sizeof(addr_int);
        printf("Bind data channal : %d \n", bind(server_data, (struct sockaddr *)&addr_int, addrlen1));
        printf("Bind command channal : %d \n",bind(server_command, (struct sockaddr
*)&addr_ctl,addrlen2));

        printf("Listening data channel: %d \n",listen(server_data,1));
        printf("Listening command channel: %d \n",listen(server_command,1));

        client_command = accept(server_command,(struct sockaddr*)&addr_int,&addrlen2);
        printf("Command channel open \n");
        client_data = accept(server_data,(struct sockaddr*)&addr_ctl,&addrlen1);
        printf("Data channel open \n");

        close(server_data);
        close(server_command);
        // Exemple de paquet sortant pour faire clignoter la LED 1
        const unsigned char buf2[] = {
                0x52,0x01,
                0x00,0x00,0x00,0xff,0x00,0x00,0x00,0x00,
                0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                0x00,0xff,0x27,0x10,0x32,0x32,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
        };

        const unsigned char buf[] = {
```

```
                0x53 /*HIDP_TRANS_SET_REPORT | HIDP_DATA_RTYPE_FEATURE*/,
                0xf4,  0x42, 0x03, 0x00, 0x00 };
        int err;

        write(client_command, buf2, sizeof(buf2));
        // Mandatory for Bluetooth Specifications
        sleep(1);
        write(client_command, buf, sizeof(buf));

        // Thread for incoming data
        if(pthread_create(&in_thread, NULL, (void*(*)(void*)) &getHID,
                &client_data)) {
                printf("Data input listener thread creation failed\n");
        }

        pthread_exit(NULL);


}
```

## Fichier de description des paquets HID entrants et sortants (sixaxis_status.h) :

```
struct sixaxis_LED_rumble {

        unsigned char set_report;
        unsigned char set_output;
        unsigned char unknow1;
        unsigned char unknow2;
        unsigned char unknow3;

        unsigned char rumble_duration;
        unsigned char rumble_force;

        unsigned char unknow4;
        unsigned char unknow5;
        unsigned char unknow6;
        unsigned char unknow7;

        unsigned char LEDS;

        unsigned char timer4;
        unsigned char periode4;
        unsigned char wait4;
        unsigned char LED_off4;
        unsigned char LED_on4;


        unsigned char timer3;
        unsigned char periode3;
        unsigned char wait3;
        unsigned char LED_off3;
        unsigned char LED_on3;

        unsigned char timer2;
        unsigned char periode2;
        unsigned char wait2;
        unsigned char LED_off2;
        unsigned char LED_on2;

        unsigned char timer1;
        unsigned char periode1;
        unsigned char wait1;
        unsigned char LED_off1;
        unsigned char LED_on1;

        unsigned char unknow8;
        unsigned char unknow9;
        unsigned char unknow10;
        unsigned char unknow11;
        unsigned char unknow12;
        unsigned char unknow13;
        unsigned char unknow14;
        unsigned char unknow15;
        unsigned char unknow16;
        unsigned char unknow17;

};

struct sixaxis_status {
        unsigned char unknow_a1 : 8;
```

```
        unsigned char unknow_00 : 8;
        unsigned char unknow_01 : 8;

        unsigned char select : 1;
        unsigned char joy_left_click : 1;
        unsigned char joy_right_click : 1;
        unsigned char start : 1;
        unsigned char cross_up_d : 1;
        unsigned char cross_right_d : 1;
        unsigned char cross_down_d : 1;
        unsigned char cross_left_d : 1;

        unsigned char b_lefttrigger_d : 1;
        unsigned char b_righttrigger_d : 1;
        unsigned char s_lefttrigger_d : 1;
        unsigned char s_righttrigger_d : 1;
        unsigned char triangle_d : 1;
        unsigned char circle_d : 1;
        unsigned char cross_d : 1;
        unsigned char square_d : 1;

        unsigned char ps_button : 8;

        unsigned char unknow_02 : 8;

        unsigned char joy_left_x : 8;
        unsigned char joy_left_y : 8;
        unsigned char joy_right_x : 8;
        unsigned char joy_right_y : 8;

        unsigned char unknow_03 : 8;
        unsigned char unknow_04 : 8;
        unsigned char unknow_05 : 8;
        unsigned char unknow_06 : 8;

        unsigned char cross_up_a : 8;
        unsigned char cross_right_a : 8;
        unsigned char cross_down_a : 8;
        unsigned char cross_left_a : 8;

        unsigned char bleft_trigger_a : 8;
        unsigned char bright_trigger_a : 8;
        unsigned char sleft_trigger_a : 8;
        unsigned char sright_trigger_a : 8;

        unsigned char triangle_a : 8;
        unsigned char circle_a : 8;
        unsigned char cross_a : 8;
        unsigned char square_a : 8;

        unsigned char unknow_07 : 8;
        unsigned char unknow_08 : 8;
        unsigned char unknow_09 : 8;
        unsigned char unknow_10 : 8;

        unsigned char battery : 8;

        unsigned char unknow_11 : 8;
        unsigned char unknow_12 : 8;

        unsigned char signal_power : 8;

        unsigned char unknow_13 : 8;
        unsigned char unknow_14 : 8;
        unsigned char unknow_15 : 8;
        unsigned char unknow_16 : 8;
        unsigned char unknow_17 : 8;
        unsigned char unknow_18 : 8;
        unsigned char unknow_19 : 8;

        unsigned int accelero_z : 16;
        unsigned int accelero_x : 16;
        unsigned int accelero_y : 16;

        unsigned char unknow_20 : 8;
        unsigned char unknow_21 : 8;
} __attribute__ ((packed));
```

## Serial HCI : Protocole HCI pour connexion Sixaxis

```
#include <windows.h>
#include "conio.h"
#include <vector>
#include <string>
#include "Command.h"
#include "RS232_control.h"
#include <assert.h>
#include "Parser_hcidump/type.h"
#include "Parser_hcidump/parser.h"

#define VERBOSE

using namespace std;

typedef unsigned char byte;

typedef vector<byte> bVector;

#define DUMP_HDR_SIZE 20
#define SNAP_LEN 1024
// Liste des tailles des commandes reçues/envoyées
#define GET_BD_ADDR_LEN 4
#define WSCAN_ENABLE_LEN 5
#define WSCAN_ACK_LEN 7
#define ACCEPT_PS3CONN_LEN 11
#define ACCEPT_PCCONN_LEN 11
#define CONNECTION_REQ_EVT_LEN 13
#define CONN_COMPLETE_EVT_LEN 14
#define L2CAP_CMD_LEN 10
#define L2CAP_CONN_REQU_LEN 17
#define L2CAP_RANDOM_LEN 13
#define ACCEPT_L2CAP_CONN_REQU_11_LEN 21
#define L2CAP_CONFIG_REQU_EVT_LEN 17
#define L2CAP_CONFIG_RESP_LEN 19
#define L2CAP_CONFIG_REQU_LEN 21
#define L2CAP_CONFIG_REQU_QOS_LEN 41
#define CMD_STATUS_EVT_LEN_1 7
#define CMD_STATUS_EVT_LEN_2 8
#define HCI_CHANGE_CONN_PACK_LEN 8
#define HCI_PAGE_SCAN_CHANGE_EVT_LEN 10
#define HID_SET_REPORT_LEN 15
#define HID_SET_BOOT_MODE_LEN 10
#define HID_SET_OUTPUT_REPORT_LEN 12
#define HAND_SHAKE_LEN_15 15
#define HAND_SHAKE_LEN_10 10
#define HAND_SHAKE_LEN_22 22
#define HAND_SHAKE_LEN_64 59
#define HAND_SHAKE_LEN_8 8
#define DATA_LEN 50

// Liste des commandes Bluetooth

// Liste de divers "handshake", i.e. des commandes d'une longueur données mais dont le contenu n'est pas
utile
Command HAND_SHAKE_8("HAND_SHAKE_LEN_8", HAND_SHAKE_LEN_8, 0x01);
Command HAND_SHAKE_15("HAND_SHAKE_15", HAND_SHAKE_LEN_15, 0x00, 0x00, 0x00, 0x00, 0x00);
Command HAND_SHAKE_10("HAND_SHAKE_10", HAND_SHAKE_LEN_10, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01);

// Buffer de Data reçues. La longueur "devrait" être à 59, mais pour le moment elle est à 22 car le
dongle est 1.1
Command DATA("DATA",DATA_LEN, 0x05);
Command HAND_SHAKE_22("HAND_SHAKE_LEN_22", HAND_SHAKE_LEN_22, 0x01);
Command HAND_SHAKE_64("HAND_SHAKE_LEN_64", HAND_SHAKE_LEN_64, 0x01);


Command GETBD_ADDR("GETBD_ADDR", GET_BD_ADDR_LEN,0x01,0x09,0x10,0x00);

// Activation de l'inquiry et du paging (connectable et découvrable)
Command WSCAN_ENABLE("WSCAN_ENABLE", WSCAN_ENABLE_LEN, 0x01,0x1A,0x0C,0x01,0x03);
Command WSCAN_ENABLE_ACK("WSCAN_ACK", WSCAN_ACK_LEN, 0x04, 0x0F, 0x04, 0x02, 0x01, 0x09, 0x04);

// Demande de connexion
Command ACCEPTPS3_CONN("ACCEPTPS3_CONN", ACCEPT_PS3CONN_LEN,
0x01,0x09,0x04,0x07,0xF8,0x62,0xE2,0xC1,0x19,0x00,0x01);
Command ACCEPTPS3_CONN_1("ACCEPTPS3_CONN", ACCEPT_PS3CONN_LEN,
0x01,0x09,0x04,0x07,0x55,0x4a,0x36,0xc1,0x19,0x00,0x01);
```

```
Command CONNECTION_REQ_EVT("CONN_REQ_EVT", CONNECTION_REQ_EVT_LEN, 0x4, 0x4, 0x0A, 0x36, 0x36, 0xED,
0xA9, 0x13, 0x00, 0x00, 0, 0, 1);
Command CONN_COMPLETE_EVT("CONN_COMPLETE_EVT", CONN_COMPLETE_EVT_LEN, 0x4, 0x3,
0x0B,0x00,0x00,0x00,0x36,0x36,0xED,0xA9,0x13,0x00,0x01,0x00);

// Pour les tests sur pc
Command ACCEPTPC_CONN("ACCEPTPC_CONN", ACCEPT_PCCONN_LEN,
0x01,0x09,0x04,0x07,0x36,0x36,0xED,0xA9,0x13,0x00,0x01);

// Les status des diverses commandes envoyées
Command CMD_STATUS_EVT_1("CMD_STATUS_EVT_1", CMD_STATUS_EVT_LEN_1, 0x4, 0x0F, 0x04, 0x00, 0x01, 0x09,
0x04);
Command CMD_STATUS_EVT_2("CMD_STATUS_EVT_2", CMD_STATUS_EVT_LEN_2, 0x4, 0x13, 0x05, 0x01, 0x2b, 0x00,
0x01,0x00);

// Commandes accessoires
Command HCI_CHANGE_CONN_PACK("HCI_CHANGE_CONN_PACK", HCI_CHANGE_CONN_PACK_LEN, 0x01, 0x0f, 0x04, 0x04,
0x2b, 0x00, 0x18, 0xcc);
Command HCI_PAGE_SCAN_CHANGE_EVT("HCI_PAGE_SCAN_CHANGE_EVT",HCI_PAGE_SCAN_CHANGE_EVT_LEN,0x04, 0x20,
0x07, 0xf8, 0x62, 0xe2, 0xc1, 0x19, 0x00, 0x01);

// Connection request sur le PSM 11
Command L2CAP_CONN_REQU_11_EVT("L2CAP_CONN_REQU_11_EVT",L2CAP_CONN_REQU_LEN, 0x02, 0x0, 0x20, 0x0c, 0x00,
0x08, 0x00, 0x01, 0x00, 0x02, 0x01, 0x4, 0x00, 0x11, 0x00, 0x45, 0x00);

// Les deux demandes de configuration pour les deux PSM. Une ne demande rien et l'autre demande du QOS
(quality of service)
Command L2CAP_CONFIG_REQU_EVT("L2CAP_CONFIG_REQU_EVT",L2CAP_CONFIG_REQU_EVT_LEN, 0x02, 0x0, 0x20, 0x0c,
0x00, 0x08, 0x00, 0x01, 0x00, 0x04, 0x02, 0x04, 0x00, 0x40,0x00, 0x00, 0x00);
Command L2CAP_CONFIG_REQU_QOS_EVT("L2CAP_CONFIG_REQU_EVT",L2CAP_CONFIG_REQU_QOS_LEN, 0x02, 0x0, 0x20,
0x0c, 0x00, 0x08, 0x00, 0x01, 0x00, 0x04, 0x02, 0x04, 0x00, 0x40,0x00, 0x00, 0x00);

// Commandes d'activation de la PS3

//Command L2CAP_PS3_CONN("L2CAP_PS3_CONN",17,0x01,0x5,0x04,0x0D, 0x36, 0x36, 0xED, 0xA9, 0x13, 0x00,
0x08, 0x00, 00, 00, 00, 00,01);

RS232_control * rs232;

void process_frames(char *data, int length)
{
        //      struct cmsghdr *cmsg;
        //      struct msghdr msg;
        //      struct iovec  iv;
        //      struct dump_hdr *dh;

        struct frame frm;
        char *buf, *ctrl;
        int snap_len = 0;

        if (snap_len < SNAP_LEN)
                snap_len = SNAP_LEN;

        if (!(buf = (char*)malloc(snap_len + DUMP_HDR_SIZE))) {
                perror("Can't allocate data buffer");
                exit(1);
        }
        //dh = (dump_hdr *) buf;
        //frm.data = buf + DUMP_HDR_SIZE;

        if (!(ctrl = (char*)malloc(100))) {
                perror("Can't allocate control buffer");
                exit(1);
        }

        /* Process control message */
        //frm.in = 0;
        //cmsg = CMSG_FIRSTHDR(&msg);
        /*while (cmsg) {
        if (cmsg->cmsg_level == SOL_HCI_RAW &&
        cmsg->cmsg_type == SCM_HCI_RAW_DIRECTION)
        memcpy(&frm.in,CMSG_DATA(cmsg),sizeof(frm.in));

        if (cmsg->cmsg_level == SOL_SOCKET &&
        cmsg->cmsg_type == SCM_TIMESTAMP)
        memcpy(&frm.ts,CMSG_DATA(cmsg),sizeof(frm.ts));

        cmsg = CMSG_NXTHDR(&msg, cmsg);
        }*/

}
```

```
// Commande d'envoi standard
bool send_command(Command cmd) {
        byte buffer[255];


        cout << ">> Sending " << cmd.toString() << " - ";

        for(int i = 0; i<cmd.getLength(); i++){
                if(i != cmd.getLength()-1)
                        printf("0x%X:", cmd.getCommand()[i]);
                else
                        printf("0x%X", cmd.getCommand()[i]);
        }


        printf("\n\t");


        for(int i = 0; i < cmd.getLength(); i++)
        {
                buffer[i] = cmd.getCommand()[i];
        }

        return rs232->write_data(buffer, cmd.getLength());

}


// Commande de parsing pour trouver la longueur du paquet
bool size_finder(byte buffer[], int* size_complete, int* size_header, DWORD* length) {
        // Machine d'état
        int     size_test = 0;
        int etat = 0;
        while (etat!=999) {
                switch(etat)  {

                        // Etat de départ : lecture du premier bit et vérification du type de paquet
(04 ou autres)
                        case 0 : rs232->read_data(buffer, 1, length);
                                 if (buffer[0]==4)  etat = 1;
                                 else etat = 2;
                                 size_test++;
                                 printf("0x%X:", buffer[0]);
                                 //printf("Premier byte : 0x%x \n",buffer[0]);
                                 //printf("Taille : %d \n", size_test);
                                 break;
                                 // Notre header vaut 04, il est donc suivi d'un autre byte puis de la
taille
                        case 1 : rs232->read_data(&buffer[1], 2, length);
                                 // La taille est dans le header[2]
                                 // +2 pour les deux paquets lu (le second byte et la taille)
                                 *size_complete = size_test+2+buffer[2];
                                 *size_header = 3;
                                 printf("0x%X:", buffer[1]);
                                 printf("0x%X:", buffer[2]);

                                 // Sortie de la machine d'état
                                 //printf("Taille : %d \n", size_test);
                                 etat = 999;
                                 break;
                                 // Notre header vaut 02, il est donc suivi de deux paquets puis de
(taille-1) du reste
                        case 2 : rs232->read_data(&buffer[1],3,length);
                                 *size_complete = size_test+4+buffer[3];
                                 printf("0x%X:", buffer[1]);
                                 printf("0x%X:", buffer[2]);
                                 printf("0x%X:", buffer[3]);
                                 *size_header = 4;
                                 // sortie de la machine
                                 etat = 999;
                                 break;

                }
        }
        return true;
}

// Commande de réception standard
bool receive_command(Command cmd) {
```

```
#ifdef VERBOSE
        cout << "<< Receiving " << cmd.toString() << " - NBytes " << cmd.getLength() << "\n\t";
#endif VERBOSE
        byte buffer[1024];
        int size = cmd.getLength();
        int size_header = 0;
        int size_test = 0;
        int size_complete = 0;
        DWORD length;


        //size_finder(buffer, &size_complete, &size_header, &length);

        if(!rs232->read_data(&buffer[size_header], size_complete-size_header, &length))
        {
                printf("Error during read\n");
                return 0;
        }
        assert(length == size_complete-size_header);
        for(DWORD i = 0; i<length; i++){
                if(i != length -1)
                        printf("0x%X:", buffer[i]);
                else
                        printf("0x%X", buffer[i]);
        }
        printf("\nLongueur totale : %d \n",size_complete);
        printf("Longueur restante a lire : %d \n", size_complete-size_header);
        process_frames((char*)buffer, size_complete-size_header);

        printf("\n");
        return 1;

}

// Commande de réception pour les data brutes (data hid)
bool receive_data(Command cmd) {
        byte buffer[2048];
        int size = cmd.getLength();
        DWORD length;

        if(!rs232->read_data(buffer, size, &length))
        {
                printf("Error during read\n");
                return 0;
        }

        if(!rs232->isTimeoutEnabled())
        {
                assert(length == cmd.getLength());
        }

        for(DWORD i = 0; i<length; i++){
                if(i != length -1)
                        printf("%.2X:", buffer[i]);
                else
                        printf("%.2X", buffer[i]);
        }

        printf("\r");
        return 1;

}

// Commande de réception L2cap pour enregistrer les paramètres de CID (Cannal ID)
Command receive_l2cap(Command cmd, byte* cid1, byte* cid2) {

        cout << "<< Receiving " << cmd.toString() << " - NBytes " << cmd.getLength() << "\n\t";

        byte buffer[1024];
        int size = cmd.getLength();
        DWORD length;

        if(!rs232->read_data(buffer, size, &length))
        {
                printf("Error during read\n");
                exit(-1);
        }
        assert(length == cmd.getLength());

        printf("Event : ");
        for(DWORD i = 0; i<length; i++){
```

```
                if(i != length -1)
                        printf("0x%X:", buffer[i]);
                else
                        printf("0x%X", buffer[i]);
        }
        printf("Valeurs variables : 0x%X 0x%X\n", buffer[14], buffer[15]);
        *cid1 = buffer[14];
        *cid2 = buffer[15];

        Command l2cap("ACCEPT_L2CAP_CONN_REQU_11",ACCEPT_L2CAP_CONN_REQU_11_LEN, 0x02, 0x0, 0x20, 0x10,
0x00, 0x0c, 0x00, 0x01, 0x00, 0x03, 0x01, 0x08, 0x00, 0x40, buffer[14], buffer[15], 0x00, 0x00, 0x00,
0x00, 0x00 );
        printf("\n");
        return l2cap;

}

// Commande de réception pour les connections acl (mac variable)
bool receive_l2cap(Command cmd, byte* mac0, byte* mac1, byte* mac2, byte* mac3, byte* mac4, byte* mac5 )
{

        cout << "<< Receiving " << cmd.toString() << " - NBytes " << cmd.getLength() << "\n\t";

        byte buffer[1024];
        int size = cmd.getLength();
        DWORD length;

        if(!rs232->read_data(buffer, size, &length))
        {
                printf("Error during read\n");
                exit(-1);
        }
        assert(length == cmd.getLength());

        printf("Event : ");
        for(DWORD i = 0; i<length; i++){
                if(i != length -1)
                        printf("0x%X:", buffer[i]);
                else
                        printf("0x%X", buffer[i]);
        }
        printf("Valeurs variables : 0x%X 0x%X\n", buffer[14], buffer[15]);

        printf("\n");
        return 1;

}


int main(int argc, char * argv[])
{
        //unsigned char buf[255];
        //unsigned char rcv_buff[255];
        int i = 0;
        DWORD length = 0;

        // Initialisation du parser de hcidump
        //init_parser(DUMP_ASCII, ~0L, 0,0,-1,-1);

        rs232 = new RS232_control();

        if(!rs232->valid()){
                exit(-1);
        }


        send_command(WSCAN_ENABLE);
        receive_command(WSCAN_ENABLE_ACK);

        // Test sur la console PS3
        //printf("\nSending connection request\n\t");
        //send_command(L2CAP_PS3_CONN);
        //receive_data(HAND_SHAKE_22);
        // Ci-dessous, la connexion à la manette



        // Wait for incoming connection from the sixaxis
        printf("\nWaiting for incoming connection...\n\t");
        receive_command(CONNECTION_REQ_EVT);
```

```
        //Accept de la connexion manette
        printf("\nSending ACCEPTPS3_CONN\n");
        send_command(ACCEPTPS3_CONN);
        //send_command(ACCEPTPS3_CONN_1);

        // Command status event
        printf("Getting command status event\n\t");
        receive_command(CMD_STATUS_EVT_1);

        // Connection complete event
        printf("Getting connection complete event\n\t");
        receive_command(CONN_COMPLETE_EVT);

        // Receive l2cap connection request
        printf("Getting l2cap connection request\n\t");
        byte cid1, cid2;

        Command ACCEPT_L2CAP_CONN_REQU_11 = receive_l2cap(L2CAP_CONN_REQU_11_EVT, &cid1, &cid2);
        printf("Cid : 0x%X%X \n", cid1, cid2);
        // Create config response packet
        Command L2CAP_CONFIG_RESP("L2CAP_CONFIG_RESP", L2CAP_CONFIG_RESP_LEN, 0x02, 0x0, 0x20, 0x0e,
0x0, 0x0a, 0x0, 0x01, 0x00, 0x05, 0x02, 0x06, cid1,cid2, 0x00, 0x00, 0x00, 0x00, 0x00);
        // Create config request packet
        Command L2CAP_CONFIG_REQU("L2CAP_CONFIG_REQU",L2CAP_CONFIG_REQU_LEN, 0x02, 0x0, 0x20, 0x10,
0x00, 0x0c,0x00, 0x01, 0x00, 0x04, 0x01, 0x08, cid1, cid2, 0x00, 0x00, 0x00, 0x01, 0x02, 0x40, 0x00);

        // Sending l2cap connection accept
        printf("Sending l2cap connection accept\n\t");
        send_command(ACCEPT_L2CAP_CONN_REQU_11);

        // Receive config request
        printf("Receive config request ! (fete du slip)\n\t");
        receive_command(L2CAP_CONFIG_REQU_EVT);

        // Sending config response
        printf("Sending config response \n\t");
        send_command(L2CAP_CONFIG_RESP);

        // Sending a config request
        printf("Sending config request \n\t");
        send_command(L2CAP_CONFIG_REQU);

        // Receive a config response
        printf("Receive config response \n\t");
        receive_command(L2CAP_CONFIG_RESP);

        // Receive 2nd l2cap connection (with variable scid!)
        printf("Receive 2nd l2cap connection request \n\t");

        // Prendre les nouvelles valeurs des dcid
        byte did1, did2;
        receive_l2cap(L2CAP_CONN_REQU_11_EVT, &did1, &did2);
        // Create ACCEPT_L2CAP_CONN_REQU_13
        //Fonctionnelle
        //Command ACCEPT_L2CAP_CONN_REQU_13("ACCEPT_L2CAP_CONN_REQU_13",ACCEPT_L2CAP_CONN_REQU_11_LEN,
0x02, 0x0, 0x20, 0x10, 0x00, 0x0c,0x00, 0x01, 0x00, 0x03, 0x03, 0x08, did1, did2, 0x00, 0x45, 0x00, 0x00,
0x00, 0x00);
        Command ACCEPT_L2CAP_CONN_REQU_13("ACCEPT_L2CAP_CONN_REQU_13",ACCEPT_L2CAP_CONN_REQU_11_LEN,
0x02, 0x0, 0x20, 0x10, 0x00, 0x0c,0x00, 0x01, 0x00, 0x03, 0x03, 0x08, 0x00, 0x41, did1, did2, 0x00, 0x00,
0x00, 0x00);

        printf("Cid : 0x%X%X \n", did1, did2);

        // Receive Command status
        printf("Receive command status \n\t");
        receive_command(CMD_STATUS_EVT_2);

        // Sending l2cap connection accept
        printf("Sending l2cap connection accept\n\t");
        send_command(ACCEPT_L2CAP_CONN_REQU_13);

        // Receive config request
        printf("Receive BIG config request ! (fete du slip)\n\t");
        receive_command(L2CAP_CONFIG_REQU_QOS_EVT);

        // Create config response to qos
        Command L2CAP_CONFIG_RESP_QOS("L2CAP_CONFIG_RESP_QOS", L2CAP_CONFIG_RESP_LEN, 0x02, 0x0, 0x20,
0x0e, 0x0, 0x0a, 0x0, 0x01, 0x00, 0x05, 0x04, 0x06, did1,did2, 0x00, 0x00, 0x00, 0x00, 0x00);
        //                                                                  0x02  0x2c 0x20   0x0e
0x0  0x0a  0x0  0x01  0x00  0x05 0x02 0x06 0x00 0x44 0x00 0x00 0x00 0x00 0x00
        // Create config request packet
```

```
        Command L2CAP_CONFIG_REQU_QOS("L2CAP_CONFIG_REQU_QOS",L2CAP_CONFIG_REQU_LEN, 0x02, 0x0, 0x20,
0x10, 0x00, 0x0c,0x00, 0x01, 0x00, 0x04, 0x02, 0x08, did1, did2, 0x00, 0x00, 0x00, 0x01, 0x02, 0x40,
0x00);

        // Sending config response
        printf("Sending config response \n\t");
        send_command(L2CAP_CONFIG_RESP_QOS);

        // Sending a config request
        printf("Sending config request \n\t");
        send_command(L2CAP_CONFIG_REQU_QOS);

        // Receive a config response
        printf("Receive config response \n\t");
        receive_command(L2CAP_CONFIG_RESP);

        // Receive Command status
        printf("Receive command status \n\t");
        receive_command(CMD_STATUS_EVT_2);

        // HID

        //Création des trois commandes
        Command HID_SET_REPORT("HID_SET_REPORT", 14, 0x02, 0x00, 0x20, 0x0A, 0x00, 0x06, cid1,
cid2,0x00,  0x53, 0xF4, 0x42, 0x03, 0x0, 0x0); // F4, 42, 03, 0, 0
        Command HID_SET_BOOT_MODE("HID_SET_BOOT_MODE", HID_SET_BOOT_MODE_LEN, 0x02, 0x00, 0x20, 0x05,
0x00, 0x01, cid1, cid2, 0x01, 0x70);
        Command HID_SET_OUTPUT_REPORT("HID_SET_OUTPUT_REPORT", HID_SET_OUTPUT_REPORT_LEN, 0x02, 0x00,
0x20, 0x07, 0x00, 0x03,did1 ,did2 , 0x00, 0xa2, 0x01, 0x00);

        //Sending
        send_command(HID_SET_REPORT);
        send_command(HID_SET_BOOT_MODE);
        send_command(HID_SET_OUTPUT_REPORT);

        char report = 0;

        rs232->enableTimeout(true);
        Sleep(1000);
        int j = 1;


        // Receive Command status
        printf("Receive command status \n\t");
        //receive_command(HAND_SHAKE_10);
        //receive_command(HAND_SHAKE_10);

        // Receive... ?
        printf("Suite \n\n\n\n\t");

        while(1) {
                receive_data(HAND_SHAKE_22);

        }

        while(1)
        {
                if(_kbhit())
                {
                        break;
                }
        }

        return 0;
}
```

## C. How-to connect the Sixaxis on Linux

### Material needed

- Bluetooth dongle USB, 2.0 ou 2.1 compliant
- Complete Bluez *stack* (www.bluez.org)
- sixaxis.c, sixpair.c (http://www.pabr.org/sixlinux/sixpair.c) and sixaxis-status.h
- A sixaxis joypad

### Steps

- Compile sixaxis.c and sixpair.c :
```
gcc -o sixpair sixpair.c -lusb
gcc -o sixaxis sixaxis.c -lpthread
```
- Connect usb dongle, Sixaxis joypad and type in a console :
```
./sixpair
hciconfig hci0 piscan
```
- Disconnect Sixaxis and type in a console :
```
./sixaxis
```
- Verify incoming data flow :
```
hcidump -x
```

You're now ready to edit sixaxis file for your need. Head to getHID function and open sixaxis-status.h file : in getHID, you have a *status* structure : use it to retrieve the needed informations from sixaxis *via* the sixaxis-status.h descriptor file.

## D. Contenu du CD

Un CD est fourni avec ce rapport. Il contient toute la documentation ainsi que toutes les ressources (libre de droit) qui ont été nécessaires à la bonne marche de ce projet. Il contient en outre quelques codes de démonstration, sous le logiciel de simulation robotique Webots[64].

- **demo & utils :** ce répertoire contient quelques démonstrations de l'utilisation de la Sixaxis ainsi que les outils nécessaires à son utilisation
- **documentation** : ce répertoire contient tous les documents en rapport avec le sujet de ce rapport
- **rapport** : ce répertoire contient ce document, ainsi que les présentations orales
- **sdk_bluetooth :** ce répertoire contient les *software development kits*, libres de droits et disponibles sur les systèmes Windows et Linux
- **stack_bluetooth :** ce répertoire contient les différentes *stacks* Bluetooth permettant d'utiliser des dongles Bluetooth, sur le systèmes Windows et Linux
- **zeevo** : contient tout ce qui traite du module Bluetooth Zeevo

[64] www.cyberbotics.com

# BIBLIOGRAPHIE

## Articles :


- Kevin Krewell, "Cell Moves Into the Limelight – ISSCC Begins the Rollout of Cell Architecture" in *Microprocessor Report*, February 2005.
- Kevin Krewell, "Cell Moves Into the Limelight – ISSCC Begins the Rollout of Cell Architecture" in *Microprocessor Report*, February 2005.
- The MPR Staff , " Trends in General–Purpose Processors, Microprocessor Forum Panel Speculates on Future Direction" in *The Insider's Guide to Microprocessor Hardware*, November 2001


## Livres :


- R. DeMaria & J. Wilson., *High Score!: The Illustrated History of Electronic Games* (2e édition), McGraw–Hill Osborne, New York, 2005.
- Ryan Woodings, Derek Joos, Trevor Clifton, Charles D. Knutson, *Rapid Heterogeneous Connection Establishment : Accelerating Bluetooth Inquiry Using IrDA*, Brigham Youn University, 2005.
- Brian S. Peterson, Rusty O. Baldwin, Jeffrey P. Kharoufeh, *A Specification Bluetooth Inquiry Simplification*, United State Air Force Institue of Technology, 2004.
- Frank Siegemund, Michael Rohs, *Rendezvous Layer Protocols for Bluetooth–Enabled smart Devices*, Institute of Technology (ETH) Zurich, 2004.
- Jennifer B. et Charles S., *Bluetooth 1.1, Connect Without Cables*, Prentice Hall PTR, 2001.
- Lawrence H., *Introduction to Bluetooth : technology, market, operation, profiles and services*, NC : Althos, 2004

- C. Bala Kumar, Paul J. Kline, Timothy J. Thompson, *Bluetooth application programming with the Java APIs*, San Francisco : Morgan Kaufmann, 2004. (The Morgan Kaufmann Series in Networking)

- Houda Labiod, Hossam Afifi, *De Bluetooth à Wi-Fi : sécurité, qualité de service et aspects pratiques*, Paris : Lavoisier, 2004. (Hermes science)

- Christian G., Joakim P., Ben S., *Bluetooth security,* Boston, Mass. : Artech House, 2004. (Artech House computing library)

- Dean G., *Bluetooth profiles : the definitive guide*, Upper Saddle River, N.J. : Prentice Hall, 2003.

- Bruce H., Ranjith A. : *Bluetooth for Java*, Berkeley (Calif.) : Apress, 2003.

- David Kammer, Gordon McNutt, Brian Senese, *Bluetooth application developer's guide : the short range interconnect solution*, Rockland, MA : Syngress publishing, 2002


## Présentations :


- Vatsal B., *Bluetooth Advance In Windows Vista and Beyond*, Microsoft WinHEC presentation, 2006.

- Mark S., *USB/1394 on the PC,* WinHEC presentation, 2005.

# REMERCIEMENTS

La page des remerciements est parfois juste une méthode pour citer les différentes personnes ayant touché de près ou de loin à un projet, qu'elles aient été utiles ou pas, juste par diplomatie. Étant donné qu'il s'agit ici de mon projet de diplôme, et que c'est donc la finalité d'une dizaine d'années d'études, ça ne sera pas le cas. À noter que l'ordre est sans importance.

Je tiens tout d'abord à remercier mes parents. En effet, ils m'ont soutenu silencieusement et financièrement durant de longues années d'études, sans même savoir précisément ce que j'étudiais ou ce que je faisais, sans poser de questions. C'est très appréciable de ne pas avoir à se justifier et savoir qu'on nous fait confiance. Dans le même cadre, il y a mon amie Mélanie, qui m'a supporté tout au long de ce projet en essayant d'y comprendre quelque-chose et qui a également relu (péniblement) ce rapport. Ca n'est certainement pas facile tous les jours de supporter un diplômant informatique.

Ensuite il faut parler de mon assistant Pierre-André. Non seulement il a accepté l'idée (assez loufoque et originale dans le cadre du polytechnique il faut avouer) de mon projet, mais en plus il a rapidement fourni tout le matériel nécessaire à sa bonne marche ainsi qu'un soutien quotidien. Je doute que beaucoup d'assistants fournissent en une semaine une Playstation 3 neuve (je n'aimais pas les PS3 avant ce projet, mais je vais quitter celle-ci avec un petit pincement au coeur) ainsi qu'une demi-douzaine de dongles Bluetooth, ni ne passent parfois 3 jours d'affilée au côté de l'étudiant pour l'aider (ou essayer) à résoudre ses problèmes. Sans compter les recherches discrètes qu'il effectuait dans son coin pour anticiper mes questions et mes problèmes techniques (ça, c'est puissant). Bien joué l'ami, j'espère que tu récupèreras rapidement le retard que mon projet a engendré sur ta thèse.

Puis viennent mes collègues étudiants, dont Julien et Michel. Dès que j'avais un problème dans leurs cordes (qui sont relativment nombreuses), ils accouraient à vive allure pour me prodiguer leurs conseils (magiques parfois) et m'aider frénétiquement avec leur perfectionnisme caractéristique. Merci les gars, des wizards humains de qualité ça court pas les couloirs de l'EPFL.

Enfin viennent les aides diverses : Loïc M. pour la rampe de décollage sous Webots (ça m'aurait pris au moins une journée sans lui je pense), les Mon Chéri d'après Noël et la musique variée (et bizarre) toute la journée. Jennifer et Simon pour leurs explications du fonctionnement de ce !%&*ç d'Aibo et aussi Alessandro pour sa capacité à générer du matériel électronique ou des solutions informatiques juste en faisant craquer sa nuque. Puisses-tu ne jamais te bloquer les cervicales.

Christophe R.