MARS / AVRIL / MAI 2015 N°14

LE MAGAZINE 100 % TECHNIQUE, 100 % PRATIQUE, 100 % EMBARQUÉ !

Open Silicium

MAGAZINE

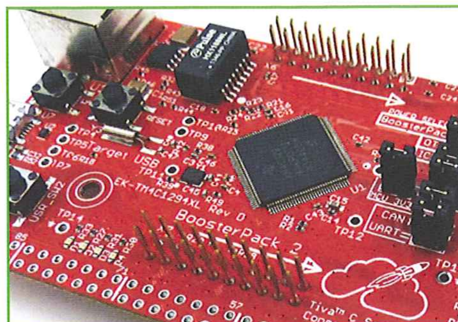
INFORMATIQUE
OPEN SOURCE
EMBARQUÉ
INDUSTRIEL ET R&D

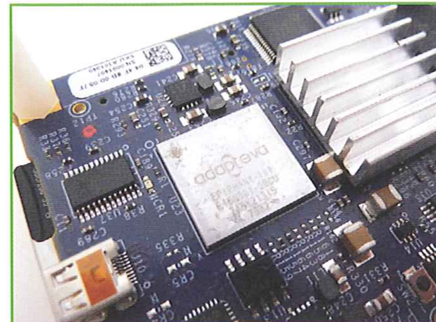
CORTEX M4 / TI

Développement sur plateforme Tiva-C Connected LaunchPad avec les bibliothèques et la ROM TivaWare p.04

16 CŒURS / ARM

Prise en main et programmation de la carte Adapteva Parallella et de son coprocesseur Epiphany p.12



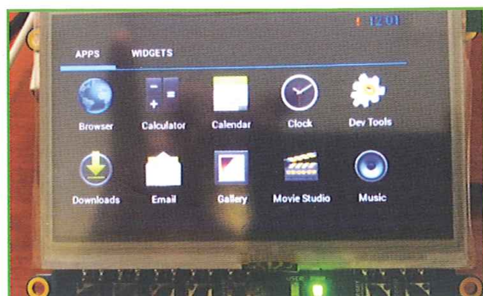


USB / DEVICE

Comprendre et utiliser l'infrastructure Linux Gadget USB pour créer des périphériques USB composites p.28

BBB / ANDROID

Personnaliser le support AOSP 4.3 de la BeagleBone Black pour prendre en charge un écran LCD p.66



ACME / LOW-COST

Mise en œuvre du SoM Arietta SAM9G25 et exploration de la configuration par device tree p.60

DEBIAN / CROSS

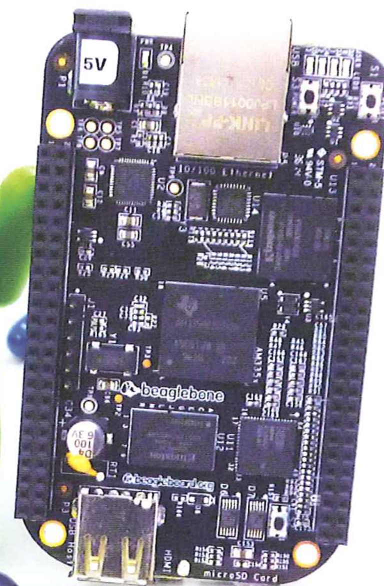
Installer en toute simplicité un environnement de compilation croisée sous Debian GNU/Linux p.56

ANDROID / INDUSTRIEL

ÉTENDEZ LE SUPPORT MATÉRIEL D'ANDROID ! p.44

...pour prendre en charge vos périphériques

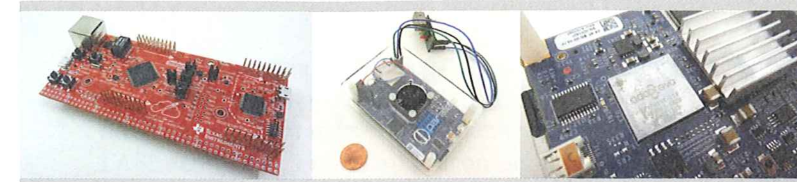
- **Noyau : intégrer le pilote pour le matériel**
- **Système : ajouter le module HAL et le service**
- **API : rendre le service accessible dans le SDK**
- **Application : développer et tester le support**



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

Maker Faire® Paris
BIENVENUE DANS LE FUTUR !
Démonstrations
Conférences
Ateliers
M
2 & 3 Mai
Foire de Paris
Paris Expo
Porte de Versailles
Make:
leFabShop
LEROY MERLIN ...et vos envies prennent Vie!
FOIRE DE PARIS
HACKABLE MAGAZINE
DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

SOMMAIRE N°14



LABO & TESTS

04 Tiva EK-TM4C1294XL LaunchPad — Denis BODOR

12 Parrallella Board : un pas vers le parallel computing — Denis BODOR

22 Parallella : développement C avec le SDK Epiphany — Denis BODOR

REPÈRES

28 L'Infrastructure Linux Gadget USB — Éric LACOMBE

38 L'Infrastructure Linux Gadget USB : le framework Composite — Éric LACOMBE

EN COUVERTURE

44 Extension de l'API Android — Pierre FICHEUX

SYSTÈME

56 Cross-compilation simple sous Debian GNU/Linux — Denis BODOR

60 ACME Arietta G25 : un module ARM9 compact et low cost — Denis BODOR

MOBILITÉ

66 AOSP pour BeagleBone Black — Pierre FICHEUX

DOMOTIQUE

72 Électronique et domotique : partie 5 : Programmation : Premiers signes de vie — Nathael PAJANI

ABONNEMENTS/COMMANDES

15/16 Abonnements multi-supports

27 Offres spéciales professionnels

SUIVEZ LES DERNIÈRES ACTUALITÉS DE VOTRE MAGAZINE SUR :

FACEBOOK : https://www.facebook.com/editionsdiamond

TWITTER : https://twitter.com/open_silicium

Open Silicium Magazine est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

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

Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali, Caroline Massing

Responsable publicité : Black Mouse Communication
Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

Service des ventes : Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : À parution, N° ISSN : 2116-3324
Commission paritaire : K90 839

Périodicité : Trimestriel
Prix de vente : 9 €

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

ÉDITO

Imagination, changements et perspectives !

Tels pourraient être les maîtres mots de tout ingénieur ou développeur qui se respecte. Des mots auxquels on peut adjoindre inventivité, curiosité, paresse, ténacité, logique, humilité, cohérence... mais les maîtres mots sont toujours plus efficaces par trois (les nains, les péchés et les mercenaires par 7, et les commandements par 10 (parce que dix, ça fait « officiel »)).

Cela tombe bien, car ce numéro va clairement faire travailler votre imagination, en particulier concernant les projets et perspectives pouvant s'ouvrir à vous sur la base des sujets présents dans les pages qui suivent : programmation parallèle, utilisation de bibliothèques de routines en ROM, développement de périphériques USB ou encore utilisation d'Android dans un domaine tout autre que celui des smartphones, tablettes, TV, wearable...

Les changements eux sont plutôt constants, car dès lors que nous envisageons des perspectives séduisantes et faisons usage de notre imagination, ce n'est qu'une question de temps avant que tout ne change : notre façon d'envisager une problématique, les outils à mettre en œuvre, les solutions à réutiliser, les systèmes à évaluer ou encore la structure des algorithmes et des codes à produire.

Notre imagination, et les perspectives qu'elle nous permet de développer, est un don propre aux vrais acteurs de la technologie (développeurs, ingénieurs, sysadmin, etc.). Un don qui se doit d'être cultivé (ce n'est pas négociable). Je préciserai toutefois le sens de mon propos, ce don d'imagination créatrice n'est pas le privilège unique de ceux qui œuvrent dans, pour, par et avec la technologie. Tout un chacun peut le posséder dans son domaine, mais c'est un trait typique et indispensable qui fait la différence entre une activité professionnelle et un métier. Lorsque le don est exacerbé, dans le domaine qui est le nôtre, la personne est généralement désignée par un qualificatif particulier : c'est un hacker.

J'espère que ce numéro saura titiller votre don et vous ouvrir des perspectives attrayantes, sinon obsédantes. Bonne lecture et... pardon pour vos nuits !

Denis Bodor

TIVA EK-TM4C1294XL LAUNCHPAD

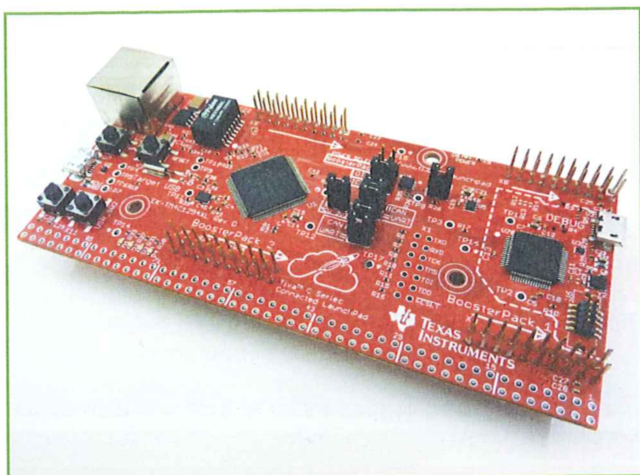
Denis Bodor

Cette carte serait presque passée inaperçue à la rédaction tant la promotion qui en est faite est axée vers la mode « tout est IoT » (IdO dans l'hexagone ou Web 3.0 pour ceux qui ne comprennent pas que l'évolution n'a pas de version stable). Ce n'est qu'en regardant de plus près qu'on se rend compte que nous avons là affaire à quelque chose de très sérieux et avec un rapport prix/(puissance+fonctionnalités) des plus intéressants.

1 Installation des éléments de compilation

1.1 Compilateur

Pour pouvoir produire des binaires à destination de la plateforme vous devez, bien sûr, disposer d'un compilateur adapté. Jusqu'à récemment le réflexe, pour un utilisateur GNU/Linux souhaitant développer en *bare metal* sur ARM, consistait à utiliser les éléments mis à disposition par *Piotr Esden-Tempski* (alias « esden ») sur GitHub (https://github.com/esden/summon-arm-toolchain). Construit autour d'un script shell imposant, la création d'esden automatisait le téléchargement, la configuration, la construction et l'installation d'une chaîne de compilation complète éventuellement complétée de certaines bibliothèques (comme la libstm32). Ce projet cependant est maintenant obsolète comme le précise le `README`. Il faut alors



se tourner vers https://launchpad.net/gcc-arm-embedded. Ici, vous trouverez une chaîne de compilation croisée complète ciblant les architectures ARM Cortex-M et Cortex-R, supportée par ARM et d'autres acteurs du marché.

Le compilateur ainsi que l'ensemble des outils de développement sont disponibles sous forme de sources naturellement, mais également de binaires à destination des plateformes Windows, Mac OS X et GNU/Linux.

L'installation relativement simple et propre consiste à télécharger les quelques 60 Mo du tarball contenant les binaires GNU/Linux et à désarchiver le fichier dans un emplacement quelconque. L'archive vous fournira un répertoire `gcc-arm-none-eabi-4_9-2014q4` contenant les sous-répertoires `arm-none-eabi`, `bin`, `lib` et `share`. Il vous suffira alors de spécifier le chemin `<mon_emplacement>/gcc-arm-none-eabi-4_9-2014q4/bin` dans votre variable d'environnement `PATH` et le tour est joué.

Comprenez bien que cette chaîne de compilation est destinée à produire des binaires *bare metal* devant fonctionner sur une plateforme **sans OS**. Le compilateur et tous les outils correspondants sont préfixés par `arm-none-eabi` selon la syntaxe `A-B-C-D` avec respectivement :

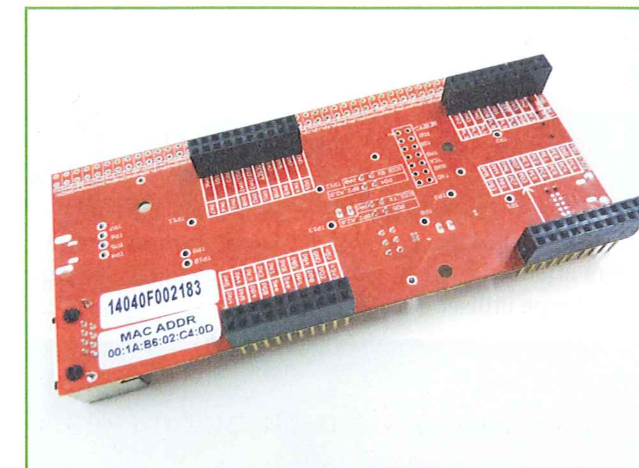
- `A` : l'architecture cible,
- `B` : le *vendor* ou *fournisseur* qui est optionnel. Il est par exemple absent ici, tout comme pour `arm-linux-gnueabi` (non *bare metal*),
- `C` : le système d'exploitation auquel sont destinés les binaires produits. Ceci est à « aucun » (`none`) dans le cas présent, il n'y a pas de système,
- `D` : définissant l'ABI ou *Application Binary Interface* qui sera utilisée. Rappelons ici qu'une ABI décrit une interface entre plusieurs composants modulaires d'un programme (comme un programme et une bibliothèque par exemple ou encore un programme et un système d'exploitation). L'ABI définit par exemple la façon de procéder aux passages de paramètres ou encore la manière de réaliser un appel système (si un OS est utilisé).

Les versions récentes de la chaîne de compilation, comme GCC 4.9.3 actuellement proposée, permettent plus facilement d'en apprendre davantage sur la plateforme supportée. Ainsi, en utilisant `arm-none-eabi-gcc --help=target`, vous obtiendrez des informations importantes concernant les architectures prises en charge (`-march`), le type de gestion de virgule flottante utilisable (`-mfpu`) ou encore les types de processeurs supportés (`-mcpu`). Pour la présente architecture, ce sera, par exemple `-mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=softfp` (`-mcpu` associé à une architecture générique est équivalent à l'utilisation de `-march` et `-mtune` avec le même argument), auquel s'ajoutera `-mthumb` pour l'utilisation du jeu d'instructions *Thumb2* (par opposition à `-marm`). Notez l'utilisation de `softfp` qui contrairement à ce qu'on pourrait imaginer n'implique en rien des calculs des flottants « logiciels » (*soft-float*) et concerne bien l'utilisation d'une FPU matérielle, mais en utilisant des syntaxes d'appel (*calling conventions*) identiques à celles du *soft-float* et donc non spécifiques au FPU présent (`hard`).

Le compilateur ainsi installé ne sera pas uniquement utile pour les microcontrôleurs de chez TI. Toutes les architectures ARM sont ciblées par cette chaîne de compilation et ces outils pourront être utilisés indifféremment avec des produits Atmel, Freescale, ST ou encore NXP. Les spécificités et caractéristiques propres à chaque marque et modèles de microcontrôleur ne relèvent pas de la chaîne de compilation, mais d'un environnement de développement généralement fourni par le constructeur ou mis à disposition par la communauté de développeurs (exemple, la libopencm3 (ex-libopenstm32) proposant une alternative à la libstm32 pour les microcontrôleurs STM32). Vous pouvez voir cela comme une sorte de BSP pour le monde des microcontrôleurs sur base ARM.

1.2 TivaWare, bibliothèques et exemples

Les microcontrôleurs de la gamme Tiva, tout comme la gamme Stellaris avant elle, présentent une fonctionnalité intéressante : TivaWare (anciennement StellarisWare). Afin de réduire l'empreinte mémoire de vos programmes et accélérer le développement, une bibliothèque de pilotes de périphériques est présente en ROM en compagnie du bootloader. Il ne s'agit, bien entendu pas de pilotes tel qu'on peut l'entendre avec un système d'exploitation, mais plutôt d'un ensemble de routines en ROM permettant de faciliter la configuration et/ou:w l'utilisation des périphériques embarqués tels que les GPIOs, le moteur cryptographique, les ADC, les ports SPI/i2c, les timers, etc.



L'accès à ces routines/pilotes se fait via l'utilisation de bibliothèques spécifiques représentant l'aspect logiciel de TivaWare et nécessitant donc une installation dans l'environnement de développement. On notera que ces éléments sont embarqués, en compagnie de la chaîne de compilation, dans l'IDE Energia offrant une équivalence à l'environnement de développement d'Arduino pour les hobbyistes. Ceci cependant ne nous intéressera pas ici étant donné la portée technique de l'article. En effet, Energia propose un environnement assez sommaire offrant certes certaines facilités, mais ne pouvant en aucun cas rivaliser avec un bon éditeur de code (Vim ou Emacs) et un lot de *Makefiles* savamment construits.

Texas Instrument met gracieusement à disposition un *Software Package* contenant les bibliothèques TivaWare ainsi qu'un grand nombre de codes de démonstration. Vous pourrez donc pointer votre navigateur web sur http://www.ti.com/tool/sw-ek-tm4c1294xl pour récupérer « *TivaWare for the Tiva C Series TM4C1294 Connected LaunchPad Evaluation Board Software* ». Si vous n'êtes pas coutumier du site web de TI et en particulier de la section de téléchargement, vous risquez d'être un peu surpris. En effet, la société est très à cheval sur les dispositions légales concernant l'exportation de technologies depuis les USA. On vous demandera donc non seulement de créer un compte, mais également de valider une quantité non négligeable d'informations prouvant (sur votre bonne foi) que vous n'êtes pas, de près ou de loin, un ennemi des États-Unis, que vous ne travaillez pas sur des projets militaires, ne comptez pas vous servir de ces informations pour créer des armes nucléaires ou biologiques, n'êtes pas sur la liste des personnes interdites, ne résidez pas en Iran ou en Corée du Nord ou encore ne comptez pas réexporter tout cela dans un pays « interdit » sans accord du gouvernement américain. Le site précise tout de même « *We apologize for any inconvenience* »...

Au final, après avoir juré sur la tête de vos trois chats que vous n'êtes pas le Docteur Denfer et ne travaillez pas sur un « giga laser » pour extorquer 1 million de dollars au

gouvernement US, vous serez à même de récupérer le fichier **SW-EK-TM4C1294XL-2.1.0.12573.exe**. Celui-ci est une archive auto-décompactable Zip :

```
% zipinfo -h SW-EK-TM4C1294XL-2.1.0.12573.exe
Archive:  ../SW-EK-TM4C1294XL-2.1.0.12573.exe
Zip file size: 78090980 bytes, number of entries: 4647
```

Il vous sera donc possible de facilement obtenir les fichiers attendus à l'aide d'un simple **unzip** quelle que soit la plateforme utilisée :

```
% mkdir tivaware
% cd tivaware
% unzip ../SW-EK-TM4C1294XL-2.1.0.12573.exe
Archive:  ../SW-EK-TM4C1294XL-2.1.0.12573.exe
  inflating: EULA.txt
  inflating: license.html
  inflating: makedefs
  inflating: Makefile
[...]
  creating: windows_drivers/win2k/
  inflating: windows_drivers/win2k/usb_dev_chidcdc_win2k.inf
  inflating: windows_drivers/win2k/usb_dev_cserial_win2k.inf
  inflating: windows_drivers/win2k/usb_dev_serial_win2k.inf
```

Si, à ce stade vous avez votre chaîne de compilation dans le **$PATH**, vous pouvez directement enchaîner sur un **make** :

```
% make
make[1]: entrant dans le répertoire
"/home/denis/EMB/TIVA/tivaware/driverlib"
  CC    adc.c
  CC    aes.c
  CC    can.c
  CC    comp.c
[...]
  CC    project.c
  CC    startup_gcc.c
  LD    gcc/project.axf
make[2]: quittant le répertoire
  "/home/denis/EMB/TIVA/tivaware/examples/project"
make[1]: quittant le répertoire
  "/home/denis/EMB/TIVA/tivaware/examples"
```

Le *software package* est conçu pour être compilé aussi bien avec IAR Embedded Workbench, TI Code Composer Studio, Keil uVision et bien entendu GCC (les *GNU tools* et Mentor Sourcery CodeBench). Si nous prenons l'exemple des codes fournis à titre de démonstration, nous pouvons nous rendre dans le sous-répertoire **examples/boards/ek-tm4c1294xl/blinky** contenant le code *helloWorld* de la classique led qui clignote.

Nous y trouvons un ensemble de fichiers destinés à tous les environnements de compilation supportés :

```
blinky.c  blinky_ccs.cmd  blinky.ewd  blinky.ewp
blinky.icf  blinky.ld  blinky.sct  blinky.uvopt
blinky.uvproj  ccs/  ewarm/  Makefile  readme.txt
rvmdk/  startup_ccs.c  startup_ewarm.c
startup_gcc.c  startup_rvmdk.S
```

Mais en réalité, ceux qui nous concernent pour GCC se limitent à :

- **Makefile** : le classique fichier interprété par **make** contenant quelques définitions et inclusions relativement faciles à modifier en cas de besoin (si l'on souhaite « sortir » un exemple de l'arborescence par exemple (variables **ROOT**, **VPATH** et **IPATH**)),
- **blinky.c** : le source de l'exemple, relativement sommaire et simple (cf plus loin),
- **blinky.ld** : le fichier script destiné à l'éditeur de liens désignant les sections du binaire et les adresses mémoire utilisées,
- **startup_gcc.c** : le code de démarrage incluant, entre autres choses, l'ISR pour le reset qui initialisera différents paramètres et lancera **main()** (le cousin de **crt0.s** en somme).

On remarquera que la section de commentaires en début de chaque fichier précise :

```
// Texas Instruments (TI) is supplying this
// software for use solely and exclusively
// on TI's microcontroller products. The software
// is owned by TI and/or its suppliers, and is
// protected under applicable copyright laws.
// You may not combine this software with "viral"
// open-source software in order to form a
// larger program.
```

Voilà qui est des plus perturbant, car même en dehors du jargon utilisé (*viral open source software*) une telle mention dans des fichiers contenant que quelques dizaines de lignes de code, et en particulier dans un simple *HelloWorld*, implique presque littéralement que vous n'avez pas le droit d'utiliser la plateforme avec un code sous licence GPL par exemple (insidieusement tout est basé sur un *HelloWorld* de 20 lignes, GNU Hello est là pour nous le rappeler après tout).

Le source de ce premier code est relativement simple comme on peut s'en douter. En voici la version francisée en espérant que TI ne voit pas cette publication comme « virale »...

```
// Fait clignoter la led D2 alias PN0

#include <stdint.h>
#include "inc/tm4c1294ncpdt.h"

int main(void) {
    volatile uint32_t ui32Loop;

    // Active le port N dans le registre RCGCGPIO
    SYSCTL_RCGCGPIO_R = SYSCTL_RCGCGPIO_R12;

    // Lecture bidon pour perdre quelques cycles
    ui32Loop = SYSCTL_RCGCGPIO_R;

    // Active PN0 en sortie
    GPIO_PORTN_DIR_R = 0x01;
    // fonctionnement numérique
    GPIO_PORTN_DEN_R = 0x01;
    while(1) {
        // Active la led
        GPIO_PORTN_DATA_R |= 0x01;

        // délai
        for(ui32Loop = 0; ui32Loop < 200000; ui32Loop++) {}

        // Désactive la led
        GPIO_PORTN_DATA_R &= ~(0x01);

        // délai
        for(ui32Loop = 0; ui32Loop < 200000; ui32Loop++) {}
    }
}
```

Plusieurs éléments sont remarquables même pour un code de cette taille et de cette simplicité. En premier lieu, nous avons le registre RCGCGPIO (**SYSCTL_RCGCGPIO_R**) permettant d'activer ou non les modules de gestion GPIO par port. Par défaut, ces modules sont inactifs et ne reçoivent pas de signal d'horloge. Il faut donc avant de s'en servir activer les modules correspondant aux ports qu'on souhaite utiliser. Ici, **SYSCTL_RCGCGPIO_R12** correspond au module du port N (voir **inc/tm4c1294ncpdt.h**) duquel dépendent 2 des 4 leds utilisateurs présentes sur la carte (respectivement PN1, PN2, PF4 et PF0).

Une lecture du registre s'en suit afin de laisser le temps au module de s'activer. Ceci semble être une pratique récurrente dans l'ensemble des codes pour Tiva-C ou Stellaris. Arrive ensuite la manipulation du registre de direction correspondant au port, **GPIO_PORTN_DIR_R**, mais entre également en jeu un autre registre, **GPIO_PORTN_DEN_R**. Celui-ci (GPIODEN) permet d'utiliser la sortie désignée en E/S numérique, mais doit être à 0 pour les lignes adoptant un fonctionnement analogique (si disponible).

Par défaut donc, et c'est important de le remarquer, les modules GPIO ne sont pas actifs, les registres de direction sont à 0 (entrée) et les GPIODEN sont également à 0. Il en va de même pour les registres que nous ne touchons pas ici comme GPIOAFSEL pour la fonction alternative, et les registres de configuration des résistances de rappel à VCC (GPIOPUR pour *Pull Up*) ou GND (GPIOPDR pour *Pull-Down*). En d'autres termes, en l'absence de configuration dans votre code, les GPIO sont inutilisables comme sur un AVR ou un MSP430.

1.3 Compilation et programmation

La structure même du *Software Package* et de ses Makefiles permet de travailler directement dans les répertoires des codes fournis en exemple. Ainsi, une modification de **blinky.c** suivie d'un **make** produira ou mettra à jour automatiquement le contenu du sous-répertoire **gcc** nommé d'après l'environnement de développement utilisé :

```
% ls gcc/
blinky.axf
blinky.bin
blinky.d
blinky.o
startup_gcc.d
startup_gcc.o
```

blinky.bin est bien entendu le binaire qui trouvera sa place en flash dans le microcontrôleur et **blinky.axf** est en réalité :

```
% file gcc/blinky.axf
gcc/blinky.axf: ELF 32-bit LSB  executable,
ARM, EABI5 version 1 (SYSV), statically linked,
not stripped

% arm-none-eabi-readelf -A gcc/blinky.axf
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "Cortex-M4"
  Tag_CPU_arch: v7E-M
  Tag_CPU_arch_profile: Microcontroller
  Tag_THUMB_ISA_use: Thumb-2
  Tag_FP_arch: VFPv4-D16
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align_needed: 8-byte
  Tag_ABI_align_preserved: 8-byte, except leaf SP
  Tag_ABI_enum_size: small
  Tag_ABI_HardFP_use: SP and DP
  Tag_ABI_optimization_goals: Aggressive Size
  Tag_CPU_unaligned_access: v6

% arm-none-eabi-size -A gcc/blinky.axf
gcc/blinky.axf  :
section          size        addr
.text             692           0
.bss              256   536870912
.comment          112           0
.ARM.attributes    55           0
Total            1115
```

L'utilisation de l'extension **.axf** semble provenir d'outils comme Keil MDK-ARM et/ou Mentor Sourcery CodeBench, mais il s'agit effectivement et tout simplement d'un fichier ELF tout à fait normal utilisé ensuite pour produire le **.bin** avec **arm-none-eabi-objcopy -O binary gcc/blinky.axf gcc/blinky.bin** (merci **make -n**).

Le fichier binaire obtenu, nous pouvons nous en servir pour programmer le microcontrôleur. À la connexion sur une machine GNU/Linux, la carte se présente comme un périphérique USB **1cbe:00fd** décrite (**lsusb**) comme *Luminary Micro Inc. In-Circuit Debug Interface* (**/var/lib/usbutils/usb.ids** ne semble pas à jour, Luminary Micro est le nom de la société acquise par TI pour développer la gamme de microcontrôleurs Stellaris maintenant renommée Tiva).

Ce périphérique USB propose 4 interfaces : deux pour la classe CDC-ACM, c'est le mode de communication associé à la partie *debug* de la carte, une de classe 255 (spécifique constructeur) pour l'interface de débogage ICDI (*In-Circuit Debug Interface*) et une de classe 254 (spécifique application) pour le DFU (*Device Firmware Upgrade*).

Tips : si vous souhaitez utiliser **usbview**, n'oubliez pas de monter **/sys/kernel/debug** via **sudo mount -t debugfs none_debugs /sys/kernel/debug** et d'utiliser, bien entendu, l'outil avec **sudo**.

Alors que la classe CDC-ACM est directement prise en charge sous Linux qui propose automatiquement un nouveau périphérique **/dev/ttyACM***, l'ICDI sera pris en charge par une application utilisateur nommée **lm4flash** développée par Fabio Utzig et disponible sur GitHub (https://github.com/utzig/lm4tools), mais également sous forme de paquet sous Debian GNU/Linux.

Bien entendu, qui dit « application utilisateur » dit **libusb** et donc gestion de permissions concernant l'accès au périphérique. Il faudra donc ajouter une règle udev permettant à un utilisateur ou un groupe d'utilisateurs de lire et écrire sur la bonne interface du périphérique USB. Dans notre cas, nous avons simplement ajouté une ligne à notre **/etc/udev/rules/Tilaunchpad.rules** prenant déjà en charge les cartes Launchpad Stellaris et MSP430 :

```
ATTRS{idVendor}=="1cbe",ATTRS{idProduct}=="00fd",
  GROUP="dialout",MODE="0660"
```

Le groupe **dialout** est l'un de ceux à qui appartient l'utilisateur courant et qui permet l'accès aux port séries (**ttyS***, **ttyUSB*** et **ttyACM***). Dans le cas d'une distribution Ubuntu ou dérivée, **plugdev** est un groupe plus approprié.

Pour programmer votre code en flash, il ne vous restera plus qu'à utiliser :

```
% lm4flash -v gcc/blinky.bin
Found ICDI device with serial: 0F002183
ICDI version: 12245
```

2 Clignoter c'est bien, avec les bonnes techniques c'est mieux !

Continuons à explorer naturellement la plateforme en passant à l'étape suivante pour **blinky.c**. Nous avons pour l'instant utilisé des opérations très basiques en accédant directement aux registres et en utilisant de simples boucles pour provoquer un délai entre deux états de la led visée. Mais si TI a choisi d'intégrer dans le microcontrôleur des routines sous la forme d'une ROM c'est précisément pour que nous n'ayons pas à procéder de la sorte. Nous pouvons donc nous tourner vers les bibliothèques à notre disposition : les *TivaWare Peripheral Driver Library*.

Les bibliothèques ou plus exactement cet ensemble de routines permettant à notre code d'utiliser celles en ROM ont été compilées lors du **make** initial à la racine du *Software Package* que nous avons désarchivé. Vous le retrouverez sous la forme d'une bibliothèque statique **driverlib/gcc/libdriver.a**.

Pour simplifier les choses, nous allons tout d'abord faire un peu de ménage en copiant le répertoire de **blinky** en **project1** après un **make clean** tout en le laissant dans **examples/boards/ek-tm4c1294xl/**. Nous faisons ensuite un brin de nettoyage pour ne garder que quelques fichiers, dont certains renommés pour l'occasion : **Makefile**, **project1.c**, **project1.ld** et **startup_gcc.c**.

Il faut ensuite se pencher sur le **Makefile** que nous allégerons et adapterons au projet/source :

```
# microcontrôleur
PART=TM4C1294NCPDT

# la racine des libs TivaWare.
ROOT=../../../..

# inclure des définitions pour make
include ${ROOT}/makedefs

# chemin vers les headers
IPATH=../../../..

# cible par défaut
all: ${COMPILER}
all: ${COMPILER}/project1.axf

# Nettoyage
clean:
	@rm -rf ${COMPILER} ${wildcard *~}

# création du rep cible pour les compilations
${COMPILER}:
	@mkdir -p ${COMPILER}

# les cibles objet
${COMPILER}/project1.axf: ${COMPILER}/project1.o
${COMPILER}/project1.axf: ${COMPILER}/startup_${COMPILER}.o

# la lib statique TivaWare
${COMPILER}/project1.axf: ${ROOT}/driverlib/${COMPILER}/libdriver.a

# Édition de liens
${COMPILER}/project1.axf: project1.ld
SCATTERgcc_project1=project1.ld
```

```
# point d'entrée
ENTRY_project1=ResetISR

# define pour le microcontrôleur utilisé
# **ET** la version de TivaWare
CFLAGSgcc=-DTARGET_IS_TM4C129_RA1

# Inclure les dépendances générées
ifneq (${MAKECMDGOALS},clean)
-include ${wildcard ${COMPILER}/*.d} __dummy__
endif
```

Les principales différences avec le **Makefile** de **blinky** concernent bien entendu le nom principal, mais aussi et surtout l'intégration de **libdriver.a** ainsi que le passage du paramètre **-DTARGET_IS_TM4C129_RA1**. Celui-ci est utilisé dans **driverlib/rom.h** pour définir la macro **ROM_GPIOPinTypeGPIOOutput** dont nous aurons besoin. Notez que l'utilisation de cette macro ainsi que des fonctions de la bibliothèque TivaWare nous dispense d'utiliser des éléments spécifiques à ce microcontrôleur précis. En réalité, nous ne devons plus inclure le **tm4c1294ncpdt.h** contenant la définition des registres (pour tout dire, il n'y a que dans **blinky** que ce *header* est directement utilisé).

Passons au code lui-même :

```
#include <stdint.h>
#include <stdbool.h>

#include "inc/hw_memmap.h"

#include "driverlib/gpio.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"

int main(void) {
  uint32_t g_ui32SysClock;

  // Fonctionnement sur PLL à 120 MHz.
  g_ui32SysClock = MAP_SysCtlClockFreqSet((
        SYSCTL_XTAL_25MHZ |     // fréquence de l'oscillateur
        SYSCTL_OSC_MAIN |       // oscillateur externe (quartz)
        SYSCTL_USE_PLL |        // sortie PLL comme sysclock
        SYSCTL_CFG_VCO_480),    // PLL VCO output à 480-MHz
        120000000);             // freq demandée

  // Activation du module GPIO port N
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);

  // Configuration de PN1 en sortie
  MAP_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_1);

  while(1) {
    // Active la led
    MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, GPIO_PIN_1);

    // délai
    MAP_SysCtlDelay(g_ui32SysClock /3);

    // Désactive la led
    MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0);

    // délai
    MAP_SysCtlDelay(g_ui32SysClock /3);
  }
}
```

Avant toutes choses, remarquez l'utilisation des fichiers *header* et en particulier ceux placés dans **driverlib/** nous fournissant l'accès aux macros et aux définitions nous permettant d'utiliser TivaWare. Notre **main()** n'est pas vraiment plus complexe que le précédent, mais se caractérise par une utilisation massive de fonctions préfixées par **MAP_** faisant référence aux routines en ROM... ou pas (cf plus loin).

Notre première opération consistera à configurer la source d'horloge utilisée pour le « système » avec ici l'utilisation de la PLL pour obtenir 120 Mhz à partir de 25 Mhz de l'oscillateur présent sur la carte et soudé à proximité du microcontrôleur. Notez que la fonction **MAP_SysCtlClockFreqSet()** nous retourne la fréquence effectivement configurée que nous utiliserons plus loin pour le délai de clignotement.

Il est important de relever ici **une** phrase absolument capitale du *TivaWare Peripheral Driver Library User's Guide*. Celle-ci se trouve page 488 et elle parle de la fonction **SysCtlClockGet()** permettant théoriquement d'obtenir la fréquence d'horloge actuellement configurée : « *This function can only be called on* **TM4C123** *devices. For TM4C129 devices, the return value from SysCtlClockFreqSet() indicates the system clock frequency* ». En clair, ne faites pas comme moi et évitez de perdre une bonne heure à comprendre pourquoi diable votre code vous retourne 9,6 Mhz alors que vous avez configuré initialement 120 Mhz... tout cela parce que vous n'avez pas vu une phrase de deux lignes dans un PDF de 700 pages. Évitez également de faire n'importe quoi avec la variable dans laquelle vous stockez la valeur retournée par **SysCtlClockFreqSet()** puisque c'est votre seule source d'information...

La configuration et l'utilisation du port GPIO se trouvent grandement simplifiées par l'utilisation de TivaWare.

Ainsi, nous n'avons qu'à utiliser les fonctions correspondantes aux routines mises à disposition.

2.1 Blizzard, snowflake, versions et ROM

Il est à présent impératif d'aborder un point qui est littéralement **LA** source de confusion lorsqu'on parle des plateformes Stellaris/Tiva : les versions et les révisions. Pour détailler l'étendue du problème, prenons un exemple avec la fonction **SysCtlClockFreqSet()**. Utilisée telle que, cette fonction se trouve définie dans **libdriver.a** et son code sera lié au vôtre pour former le fichier ELF et le contenu binaire à charger en flash.

En la préfixant, comme nous l'avons fait, de **MAP_** son destin sera différent et dépendant de ce qu'on trouve dans **driverlib/rom_map.h** :

```
#ifdef ROM_SysCtlClockFreqSet
#define MAP_SysCtlClockFreqSet \
        ROM_SysCtlClockFreqSet
#else
#define MAP_SysCtlClockFreqSet \
        SysCtlClockFreqSet
#endif
```

La question qui se pose est donc : d'où sort **ROM_SysCtlClockFreqSet** ? La réponse se trouve dans **driverlib/rom.h** :

```
#if defined(TARGET_IS_TM4C129_RA1)
#define ROM_SysCtlClockFreqSet  \
  ((uint32_t (*)(uint32_t ui32Config, \
        uint32_t ui32SysClock))ROM_SYSCTLTABLE[48])
#endif
```

Pour résumer, **MAP_SysCtlClockFreqSet()** sera soit **SysCtlClockFreqSet()** et donc une implémentation « logicielle » présente dans **libdriver.a**, soit **ROM_SysCtlClockFreqSet()** qui n'est qu'une adresse en ROM, ceci en fonction de **TARGET_IS_TM4C129_RA1** que nous passons au compilateur dans notre **Makefile**. La question suivante est donc, tout naturellement : qu'est-ce que **RA1** puisque **TARGET** et **TM4C129** sont relativement explicites ?

En cherchant un peu sur le web, on apprend rapidement que **TARGET_IS_TM4C129_RA1** correspond à TivaWare 2.1 (la version du *Software Package* que nous avons téléchargé) et que les dernières éditions de TivaWare 2.0 utilisaient **TARGET_IS_SNOWFLAKE_RA0**. Ajoutons à cela que d'autres plateformes sont désignées par **TARGET_IS_BLIZZARD_RA1** (**TARGET_IS_TM4C123_RA1**) ou encore **TARGET_IS_BLIZZARD_RB1** qui pourrait laisser penser à un TM4C129 mais non, c'est **TARGET_IS_TM4C123_RB1**, etc. TI semble apprécier les changements de désignation. L'abandon des « noms de code » fait suite au passage de TivaWare 2.0.1 à 2.1, ce qui pourrait être une bonne chose, mais finalement ne fait qu'ajouter de l'huile sur le feu initié par le *switch* Stellaris/Tiva...

Mais, lors de la prise en main d'une carte Launchpad ou d'un microcontrôleur Stellaris/Tiva, la question reste entière puisque la version du *Software Package* est une chose, mais celle des routines en ROM une autre. En effet, comme le choix de l'utilisation des routines en ROM est finalement l'affaire d'un simple argument passé au compilateur, il est facile de se tromper et de produire un code binaire qui ne fonctionnera pas intégralement.

À titre d'exemple, voici l'incidence que peut avoir ce choix, simplement sur la taille du binaire :

```
# pas d'argument -D passé au compilateur
% arm-none-eabi-size -B gcc/project1.axf
text  data  bss  dec  hex filename
3516     0  264 3780  ec4 gcc/project1.axf

# -DTARGET_IS_TM4C129_RA0 utilisé
% arm-none-eabi-size -B gcc/project1.axf
text  data  bss  dec  hex filename
2768     0  264 3032  bd8 gcc/project1.axf

# -DTARGET_IS_TM4C129_RA1 utilisé
% arm-none-eabi-size -B gcc/project1.axf
text  data  bss  dec  hex filename
1680     0  264 1944  798 gcc/project1.axf
```

Dans ce dernier cas, **libdriver.a** n'a même pas besoin d'être spécifié dans le **Makefile**, mais mieux vaut l'y laisser sachant que l'éditeur de liens fera le travail de lui-même.

Pour répondre à la question, c'est dans le microcontrôleur lui-même que se trouve la réponse et en particulier dans le registre **SYSCTL_DID0** (pour *Device IDentification 0*). Ainsi, **(char)('A'+((did0 & SYSCTL_DID0_MAJ_M)>>8))** nous donne le majeur et **(did0 & SYSCTL_DID0_MIN_M)** le mineur. Dans le cas du TM4C1294NCPDT équipant le Tiva EK-TM4C1294XL LaunchPad, ceci nous donne **A1**, et donc **TARGET_IS_TM4C129_RA1**. Nous avons donc en ROM TivaWare 2.1 et non 2.0.1 comme l'ensemble des exemples le laisse penser :

```
% cd tivaware/examples/boards/ek-tm4c1294xl
% find -type f -name "Makefile" -exec grep -H "DTARGET_IS_TM4C129" '{}' \;
./usb_host_keyboard/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./usb_dev_cserial/Makefile:CFLAGSgcc=-DUART_BUFFERED -DTARGET_IS_TM4C129_RA0
./interrupts/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./usb_host_msc/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./sleep_modes/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./udma_demo/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./usb_stick_demo/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./usb_dev_bulk/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DUART_BUFFERED
./usb_host_mouse/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./watchdog/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./bitband/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./enet_io/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./enet_uip/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DDEBUG_OUTPUT -DUIP_
OFFLOAD_ICMP_CHKSUM
./usb_dev_keyboard/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DUART_BUFFERED
./enet_lwip/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
```



```
./usb_stick_update/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./gpio_jtag/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./enet_weather/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DLWIP_1_4_1 -DUSE_HTTP_1_0 -DUART_BUFFERED
./hibernate/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DSNOWFLAKE -DUART_BUFFERED
./hello/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./qs_iot/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0 -DLWIP_1_4_1 -DUSE_HTTP_1_0 -DUART_BUFFERED
./timers/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./uart_echo/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
./mpu_fault/Makefile:CFLAGSgcc=-DTARGET_IS_TM4C129_RA0
```

Oui, ce sont bien les fichiers de **SW-EK-TM4C1294XL-2.1.0.12573.exe**, donc destinés à exploiter la version 2.1 et surtout à servir de démonstration technologique ! On ne peut que supposer qu'à force de renommer les gammes, les plateformes, les macros et les versions, Texas Instrument lui-même ne s'y retrouve pas beaucoup plus que nous. Notez que ceci n'a pas manqué de déchaîner des critiques enflammées de la part de certains développeurs, tout autant que les 80 pages de *Tiva C Series TM4C129x Silicon Errata* (ceci dit, mieux vaut 80 pages d'errata que des bugs et pas d'errata du tout).

3 Un dernier exemple pour la route

Les exemples de TI mettent également à disposition quelques fonctions intéressantes qui ne font pas partie de TivaWare. Ainsi, il devient très aisé de disposer d'une connexion série au travers du périphérique USB composite. Ceci correspond au périphérique UART0 du microcontrôleur Tiva-C et il nous suffit d'intégrer **utils/uartstdio.o** à nos cibles de compilation, tout en n'oubliant pas d'inclure **utils/uartstdio.h** à nos sources et de préciser le bon répertoire pour la variable **VPATH** dans le **Makefile**.

Ce faisant, nous pouvons alors très simplement configurer la connexion série sous la forme d'une fonction dédiée :

```
void initConsole() {
  // activation port A
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
  // config lignes RX/TX
  MAP_GPIOPinConfigure(GPIO_PA0_U0RX);
  MAP_GPIOPinConfigure(GPIO_PA1_U0TX);
  // activation UART0
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
  // config lignes PA0/PA1 en UART
  MAP_GPIOPinTypeUART(GPIO_PORTA_BASE,GPIO_PIN_0|GPIO_PIN_1);
  // config param. série
  UARTStdioConfig(0, 115200, g_ui32SysClock);

  // titi pause
  MAP_SysCtlDelay(g_ui32SysClock/80);

  // efface terminal
  UARTprintf("\033[2J\033[H");
  // coucou monde
  UARTprintf("Console ok.\n");
}
```

Nous pourrons ensuite nous en servir à loisir pour afficher, par exemple, les informations utiles contenues dans le registre **SYSCTL_DID0** :

```
initConsole();

int did0 = *(uint32_t*)SYSCTL_DID0;

UARTprintf("System clock: %u Mhz\n", g_ui32SysClock);
```

```
UARTprintf("TivaWare SYSCTL_DID0: 0x%08X: %c%x\n",
   did0,
   'A'+((did0 & SYSCTL_DID0_MAJ_M)>>8),
   (did0 & SYSCTL_DID0_MIN_M));
```

4 Conclusion temporaire

Nous avons joyeusement occupé une place non négligeable dans ce magazine et n'avons cependant pas été en mesure de faire plus qu'égratigner la surface du sujet. La carte EK-TM4C1294XL LaunchPad est un plat conséquent qu'il est difficile de digérer en une fois. La présence de la ROM TivaWare facilite effectivement les développements et l'exploration des ressources à notre disposition même s'il faut reconnaître que sa déclinaison logicielle fournit exactement les mêmes services. Il en ressort que ce que propose Texas Instrument n'est rien de plus qu'une excellente bibliothèque de gestion de périphériques avec, en prime, un gain d'espace en flash.

Bien entendu, comme nous l'avons vu, tout cela n'est pas exempt de problèmes ou de sources de confusion, mais une communauté active, accompagnant une importante masse de documentation, rendent littéralement cette plateforme très agréable à utiliser. Sans trop de difficulté, on arrive ainsi à vite faire évoluer le simpliste **blinky** dans une démarche pédagogique en ajoutant une fonctionnalité après l'autre : timers, PWM, UART, SPI, Ethernet... et ce en glanant de-ci de-là des bouts de codes provenant des exemples et du web. Enfin, n'oublions pas de signaler notre satisfaction à voir un *Software Package* construit et composé de manière à satisfaire non seulement les développeurs appréciant des IDE graphique, mais également ceux, plus « classiques » faisant usage des outils traditionnels que sont l'éditeur de code, **make** et une chaîne de compilation utilisée en ligne de commandes.

Nous reviendrons très certainement sur le sujet en couvrant avec cette carte l'utilisation de la pile TCP/IP légère lwIP. ■

PARRALLELLA BOARD : UN PAS VERS LE PARALLEL COMPUTING

Denis Bodor

Comme vous devez le savoir, les limites de vitesse, de densité et donc de puissance par cœur sont aujourd'hui quasiment atteintes. Durant des années, la conjecture empirique de Moore s'est révélée exacte et la densité des transistors a effectivement doublé tous les deux ans. Mais voilà, le mur est devant nous et il paraît plus que certain que la limite des 20 nm ne pourra pas être franchie (dame nature est contre). Aujourd'hui, les fréquences se stabilisent et la densité fait de même, mais c'est le nombre de processeurs ou de cœurs qui change.

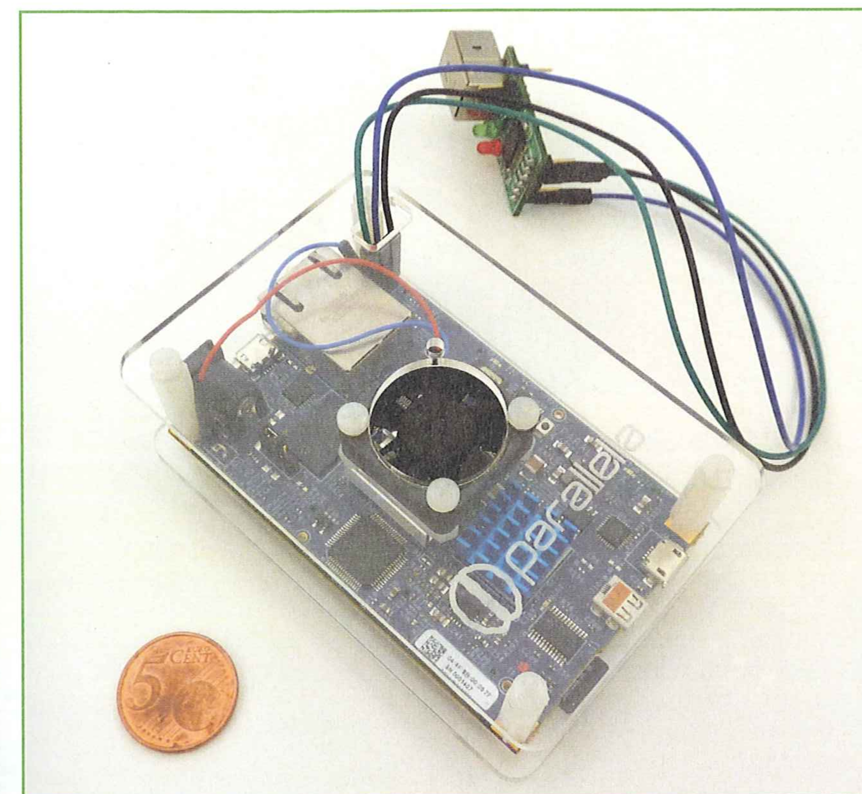
Il faut appréhender ce phénomène au sens large du terme. Certes, il existe des processeurs de 2, 4, 8, 10, 12 et même 16 cœurs comme l'AMD Opteron 6274 et il est toujours possible de multiplier ces composants avec des cartes mères 2 ou 4 processeurs, mais la multiplication des sources de puissance de calcul prend d'autres formes. Je parle, bien entendu des processeurs graphiques ou GPU qui depuis quelque temps gagnent énormément en puissance à chaque génération. Il est intéressant de noter les circonvolutions amusantes de l'histoire de l'informatique. Il y a fort longtemps, cette répartition était évidente, puis sont arrivés les concepts « on peut tout faire avec le CPU », puis les cartes 3D ont à nouveau changé la donne (souvenez-vous des cartes 3DFX). Aujourd'hui, les processeurs 3D sortent du cadre de leurs attributions originales et les GPU peuvent être utilisés pour bien d'autres usages que le simple rendu graphique. Leurs capacités mathématiques et leurs spécialisations en traitement de flux de données en font des monstres de calcul qui permettent de décharger le CPU qui ne conserve à sa charge que les tâches les plus généralistes. Là encore, on assiste à une multiplication des « unités » et le code est exécuté de manière parallèle. Ceci est à présent valable aussi bien pour les architectures de bureau, les serveurs, les machines de *gamers*, mais également les smartphones et les tablettes (le processeur Nvidia Tegra X1 est par exemple très représentatif de cette tendance).

C'est là le grand changement de ces dernières années, les processeurs n'étant plus en mesure d'augmenter la vitesse de traitement de l'information, la solution a été de miser sur le traitement parallèle (vraiment parallèle). Ceci n'a pas été sans difficulté en termes de développement logiciel, car pour écrire des programmes multi-threadés, il faut « penser parallèle » avant de « programmer parallèle ». C'est une philosophie et un domaine totalement nouveau pour beaucoup et qui n'est pas sans soulever un certain nombre de problèmes concernant la répartition des unités de calcul, l'accès concurrentiel aux données, la synchronisation, etc. Beaucoup de programmes connus, pendant longtemps, n'ont pas été adaptés à cette nouvelle architecture multicœurs ou multi-threads. C'est le cas du *raytracer* POV-Ray dont le développement officiel a mis énormément de temps à rattraper cette technologie (mais la version 3.7 disponible actuellement en version AGPL3 sait parfaitement utiliser plusieurs CPU/cœurs/threads).

1 Parallella

Ce type de problématique peut être approché en développant sur un simple PC standard aussi bien avec un CPU multicœur qu'avec un GPU. Mais il sera d'autant plus intéressant de le faire sur une plateforme dédiée à ce genre d'expérimentations intégrant un CPU bien sûr, mais également un coprocesseur multicœur spécialisé. Et c'est précisément la tâche que s'est attribuée la société Adapteva en créant l'accélérateur Epiphany. Celui-ci se décline en une version 16 et 64 cœurs et est intégré à une carte appelée Parallella (oui, deux fois deux « l »). Le projet initialement démarré sous la forme d'une campagne de financement participatif (*crowdfunding*) sur KickStarter a été lancé fin octobre 2012. Depuis cette date à laquelle l'objectif de financement a été atteint (presque $900000 sur les $750000 visés), le projet a subi de nombreux revers, déboires et retards. Ce n'est qu'en mai 2014 que la quasi-totalité des cartes ont finalement été envoyées et les *backers* livrés. La dernière génération de cartes est aujourd'hui en vente avec quelques différences notables par rapport au design original connaissant un important problème thermique nécessitant l'utilisation d'un système de refroidissement actif (ventilateur). Cette carte vendue sous le nom *Parallella-16 Desktop Computer* présente les caractéristiques suivantes :

- Soc Xilinx Zynq Z-7010 (Z-7020 sur la version KickStarter) incluant un ARM dual-core Cortex-A9 et intégrant un FPGA équivalant aux composants Artix série 7 de Xilinx,
- coprocesseur Epiphany III 16 cœurs,
- 1 Go DDR3 SDRAM,
- 128 Mb QSPI flash,
- Ethernet 10/100/1000,
- port Micro HDMI,
- emplacement Micro SD,
- contrôleur USB 2.0 hôte/OTG,
- 24 ou 48 GPIO difficiles à mettre en œuvre en raison de l'utilisation de connecteurs Samtec haute densité.



La carte Parallella d'Adapteva dans sa configuration de fonctionnement. Cet exemplaire de la première génération de cartes impose l'utilisation d'un refroidissement actif et donc d'un boîtier, de radiateurs et d'un ventilateur. La nouvelle génération permet d'utiliser un énorme radiateur couvrant le SoC, coprocesseur et environ 60% de surface de la carte.

On notera cependant que même si la carte a gagné en maturité après la campagne, le cahier des charges a été sensiblement revu à la baisse avec la version définitive. Cette dernière (modèle P160x-xxxx) utilise un SoC Zynq 7010 avec 28K cellules logiques par opposition à la version KickStarter (modèle A101040) que nous avons reçue, utilisant un 7020 avec 85K cellules logiques. La maturité de la carte, quant à elle, fait qu'il est à présent possible d'utiliser un refroidissement passif sous la forme d'un énorme radiateur (chose inapplicable avec la première génération en raison du placement gênant de certains condensateurs). Les *backers* comme moi ont donc été obligés de bricoler pour éviter la surchauffe, soit en adaptant un système de refroidissement fait maison, soit en optant pour un boîtier ventilé vendu via un tiers comme Ground Electronics à quelque 25€ (!).

Fait important, la Parallella est :

- Open Hardware : le design des cartes est disponible sous licence Creative Commons BY-SA 3.0, aussi bien pour les schémas que pour les circuits, mais aussi le code HDL trouvant sa place dans la partie FPGA et servant d'interface avec le coprocesseur Epiphany ;
- Open Source : tous les éléments logiciels sont disponibles sous licence GPL ou une autre licence plus permissive (type BSD), qu'il s'agisse du noyau, des éléments *userspace* du système en passant par le SDK propre au coprocesseur Epiphany,
- Open Documentation : toute la documentation est ouverte et sous licence Creative Commons BY-SA 3.0.

Enfin, précisons qu'il est souvent fait référence à la carte Parallella comme à un superordinateur et donc à des performances de calcul dignes d'un supercalculateur. C'est vrai... et ce n'est pas vrai. En termes de puissance de calcul pure, la Parallella n'est bien entendu en rien comparable avec des monstres comme le Tianhe-2 (Chine), Titan (USA),

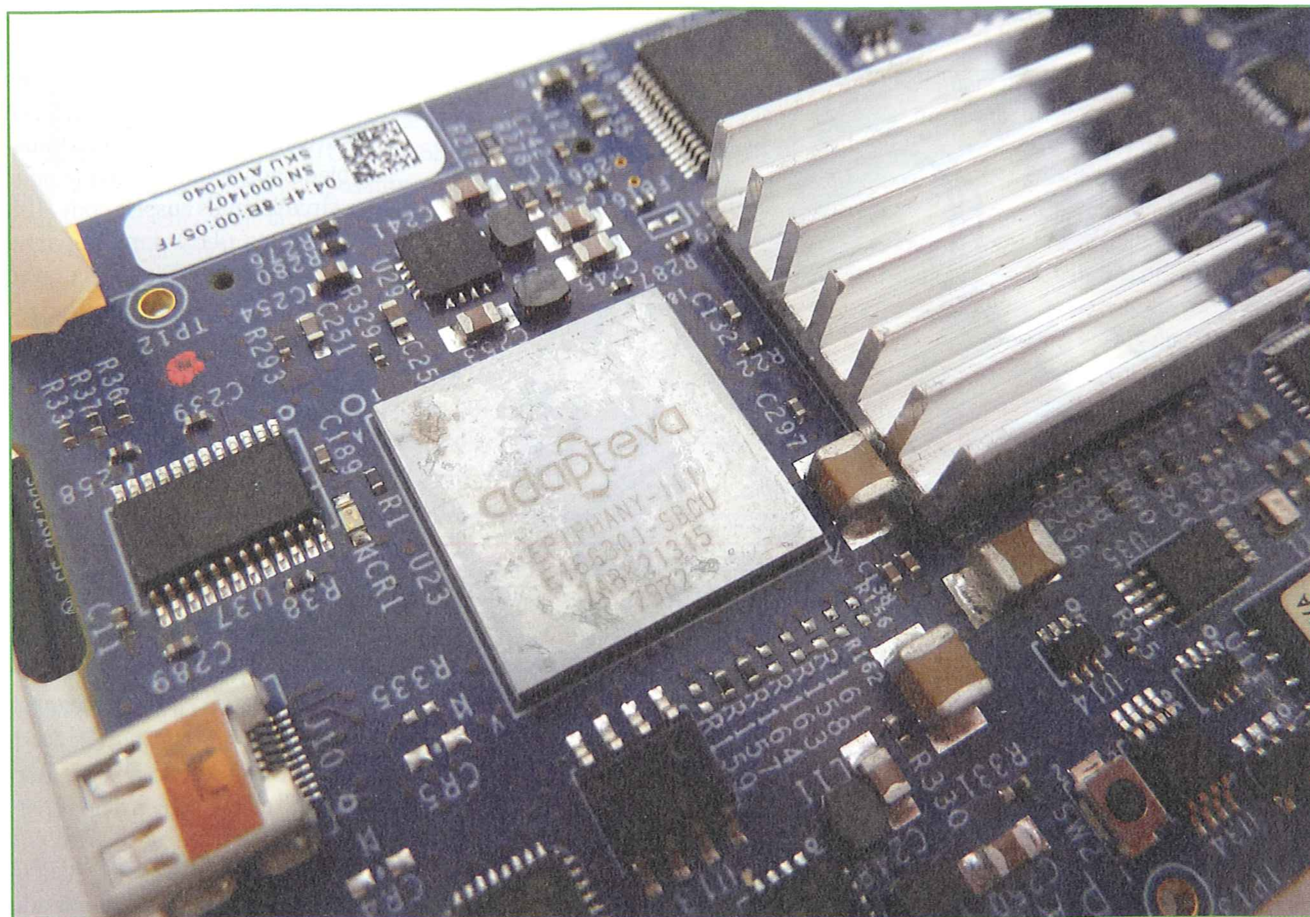
l'ordinateur K (Japon), Pangea (France) ou encore Piz Daint (Suisse). D'un point de vue architectural en revanche, la carte fonctionne exactement comme un superordinateur. En termes de rapport puissance/énergie, elle dépasse ces machines nécessitant des milliers de kilowatts. Il est plus exact de voir dans cette plateforme un support pour l'étude du développement multi-coprocesseur qu'une solution de calcul pure. Il s'agit donc d'une plateforme pédagogique, super-équipée, disposant d'une importante puissance de calcul parallèle. Une « sorte » de superordinateur modèle réduit en somme...

2 Installation du système et premiers pas

Comme de nombreuses plateformes aujourd'hui, la Parallella utilise une carte microSD contenant le système. Celui choisi pour le support officiel est Ubuntu GNU/Linux 14.04 (version Linaro). Ceci semble presque logique puisque la carte n'est en rien présentée comme un système embarqué, mais plutôt comme un nano-ordinateur spécialisé dans la découverte et la prise en main du parallélisme (une sorte de Raspberry Pi du parallélisme en somme). Toute la documentation est également axée dans ce sens et il est très rarement fait mention de compilation croisée au bénéfice de développements faits sur le système lui-même. Certes, un ARM Cortex A9 et 1 Go de DDR3 ne constituent pas une plateforme que l'on pourrait qualifier de « faible », mais c'est tout de même un peu dérangeant...

Pour récupérer les images et fichiers que nous allons utiliser, pointez simplement votre navigateur sur http://www.parallella.org/create-sdcard. Vous trouverez là une image pour la carte SD ainsi que différentes versions du noyau et du bitstream FPGA. L'ensemble se décline en version Z-7010 et Z-7020, à la fois en version supportant la sortie HDMI et en version *headless* (sans écran, utilisation via la console série ou SSH). Il n'y a aucune recommandation particulière en dehors du fait de choisir le bon modèle de carte et du fait que l'image du bitstream et du kernel doivent aller de paire.

Après téléchargement relativement long de l'image SD (le FTP est d'une lenteur affligeante), on commencera par décompresser le fichier `ubuntu-14.04-140611.img.gz` à l'aide d'un simple `gunzip`. Le fichier résultant dépasse allègrement les 7 Go et il faudra ensuite le copier sur la SD avec



Le coprocesseur 16 cœurs Adapteva Epiphany placé à côté du SoC Xilinx Zynq 7020 regroupant ARM Cortex A9 et FPGA (les traces sur l'Epiphany sont des résidus de pâte thermique suite au retrait du radiateur).

DÉCOUVREZ NOS NOUVELLES OFFRES D'ABONNEMENTS !

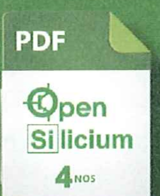
PRO OU PARTICULIER = CONNECTEZ-VOUS SUR :

www.ed-diamond.com

LES COUPLAGES PAR SUPPORT :

VERSION PAPIER
Retrouvez votre magazine favori en papier dans votre boîte à lettres !

VERSION PDF
Envie de lire votre magazine sur votre tablette ou votre ordinateur ?



ACCÈS À LA BASE DOCUMENTAIRE
Effectuez des recherches dans la majorité des articles parus, qui seront disponibles avec un décalage de 6 mois après leur parution en magazine.

Sélectionnez votre offre dans la grille au verso et renvoyez ce document complet à l'adresse ci-dessous !

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
E-mail :	

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

Vos remarques :

❑ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.
❑ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : boutique.ed-diamond.com/content/3-conditions-generales-de-ventes et reconnais que ces conditions de vente me sont opposables.

VOICI TOUTES LES OFFRES COUPLÉES AVEC OPEN SILICIUM !
POUR LE PARTICULIER ET LE PROFESSIONNEL ...

Prix TTC en Euros / France Métropolitaine

CHOISISSEZ VOTRE OFFRE !

Offre	ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + PDF (PDF 1 lecteur) Réf	PAPIER + PDF Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE Tarif TTC	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Réf	PAPIER + PDF + BASE DOCUMENTAIRE Tarif TTC
SUPPORT									
OS	4n° OS	OS1	30,-	OS12	44,-	OS13	53,-	OS123	64,-
LES COUPLAGES « EMBARQUÉ »									
D	4n° OS + 6n° HK*	D1	65,-	D12	98,-	D13	85,-*	D123	118,-*
E	4n° OS + 6n° HK* + 6n° MISC	E1	105,-	E12	158,-	E13	179,-*	E123	232,-*
E+	4n° OS + 6n° HK* + 6n° MISC + 2n° HS	E+1	119,-	E+12	179,-	E+13	193,-*	E+123	253,-*
F	4n° OS + 6n° HK* + 11n° GLMF	F1	125,-	F12	188,-	F13	229,-*	F123	292,-*
F+	4n° OS + 6n° HK* + 11n° GLMF + 6n° HS	F+1	183,-	F+12	275,-	F+13	287,-*	F+123	379,-*
G	4n° OS + 6n° HK* + 6n° LP	G1	100,-	G12	150,-	G13	164,-*	G123	214,-*
G+	4n° OS + 6n° HK* + 6n° LP + 3n° HS	G+1	129,-	G+12	194,-	G+13	193,-*	G+123	258,-*
LES COUPLAGES « GÉNÉRAUX »									
H	4n° OS + 6n° HK* + 6n° LP + 6n° MISC + 11n° GLMF	H1	200,-	H12	300,-	H13	402,-*	H123	499,-*
H+	4n° OS + 6n° MISC + 6n° HK* + 2n° HS + 6n° LP + 3n° HS + 11n° GLMF + 6n° HS	H+1	301,-	H+12	452,-	H+13	493,-*	H+123	639,-*

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | OS = Open Silicium | HC = Hackable

* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

N'hésitez pas à consulter les détails des offres ci-dessus sur : www.ed-diamond.com !

sudo dd bs=64k if=ubuntu-14.04-140611.img.gz of=/dev/periph_SD. On aura pris soin de repérer le chemin exact vers le périphérique bloc adéquat via dmesg | tail, fdisk ou tout simplement en retrouvant le périphérique dans /dev/disk/by-id/. Assurez-vous de spécifier le périphérique lui-même (/dev/sde par exemple) et non la partition FAT qu'on trouve généralement sur une carte vierge (/dev/sde1). L'opération peut être très longue étant donné la taille du fichier, soyez patient (vraiment patient).

```
% dd bs=64k if=ubuntu-14.04-140611.img of=/dev/sde
121280+0 enregistrements lus
121280+0 enregistrements écrits
7948206080 octets (7,9 GB) copiés, 504,492 s, 15,8 MB/s
```

Après cette étape, il sera peut-être nécessaire de débrancher/rebrancher la carte ou l'adaptateur USB afin que le système reconnaisse le contenu correctement. Vous devez alors trouver deux partitions, une FAT32 de 128 Mo marquée BOOT et une ext4 de ~7 Go marquée rootfs. Montez alors le système de fichiers FAT afin de procéder à la suite des opérations.

Nous avons retenu ici l'option *headless* pour le modèle Z-7020. Pour installer le noyau, il suffira de désarchiver le fichier kernel-headless-default.tgz au point de montage du système de fichiers :

```
% tar xfzv kernel-headless-default.tgz -C /mnt/SD/
uImage
devicetree.dtb
```

Sont installés une image U-Boot du noyau ainsi que le blob du *device tree*. Rappelons qu'il s'agit là d'une structure décrivant le matériel présent et permettant au noyau d'en faire usage. Cette fonctionnalité tend à se généraliser sur les systèmes modernes, car elle permet, entre autres choses, de ne plus avoir à multiplier les binaires du noyau pour différentes cartes et déclinaisons.

Il ne nous reste plus, ensuite, qu'à ajouter sur cette partition le fichier contenant le bitstream pour le FPGA. Après téléchargement, ceci revient à simplement copier le fichier avec le nom adéquat :

```
% cp parallella_e16_headless_gpiose_7020.bit.bin \
/mnt/SD/parallella.bit.bin
```

Démontez le système de fichiers et votre microSD est alors prête à être insérée dans la carte Parallella pour son premier démarrage. Comme nous avons choisi d'utiliser le support *headless*, nous devons faire usage d'un adaptateur USB/série. Le connecteur pour la console se trouve à côté de la prise Ethernet de la carte et est libellé GND/RX/TX. Assurez-vous d'utiliser un adaptateur avec des niveaux de tension 0/3,3V (bien que la carte semble supporter 5V, mieux vaut respecter les règles à la lettre).

3 Vous n'aimiez pas Upstart et Ubuntu ? Vous allez les haïr !

Alors que les premières versions du système pour la Parallella semblent être basées sur Debian GNU/Linux, l'image pour la microSD actuellement téléchargeable est une Ubuntu 14.04. Source de bon nombre de discussions tendues, il est un élément intégré dans cette distribution qui est problématique. Notez que les propos qui vont suivre sont teintés d'appréciations personnelles, mais sont assez communs auprès des utilisateurs de longue date. Ubuntu 14.04, alias *Trusty Tahr*, utilise Upstart en lieu et place du très fonctionnel et mature init SysV. Nombreux sont ceux qui estiment ce choix mauvais et préféreraient qu'Ubuntu suive la nouvelle norme consistant à faire usage de Systemd. En ce qui me concerne, Upstart et Systemd sont à la fois inappropriés pour un système embarqué ou un nano-ordinateur, mais aussi la probable raison future de mon abandon de Debian GNU/Linux au profit d'une distribution qui sera alors plus adaptée à ceux qui savent encore utiliser des scripts shell et se souviennent de la signification de l'acronyme KISS.

Je m'emporte, mais nous avons là l'exemple même des problèmes pouvant découler d'un système d'Init qui n'est pas suffisamment intelligible. Le démarrage de la carte Parallella équipée d'un adaptateur console USB/série n'aboutit pas. C'est initialement une erreur des développeurs ayant porté Ubuntu sur cette plateforme, mais la difficulté à trouver la source du problème, elle, découle bien de l'utilisation d'Upstart. Les systèmes de démarrage « concurrents », c'est parfait pour « grapiller » quelques secondes au démarrage, mais allez chercher la cause lorsqu'on constate que plusieurs services tentent d'utiliser la console série en même temps... à des vitesses différentes !

Car c'est bien là le problème qui se pose. Nous avons d'une part le bootloader et le noyau utilisant 115200 8N1 et de l'autre, un getty lancé via /etc/init/console.conf à 38400 bps, ainsi qu'un /etc/init/auto-serial-console.conf visant à auto-démarrer un getty reposant sur l'argument console= passé au noyau, et donc à 115200. Passez-moi l'expression, mais en jargon scientifique des systèmes embarqués distribués avec une mention « stable », on appelle cela « un bordel sans nom » !

Pour régler le problème, du moins partiellement, il vous faudra monter la seconde partition de la microSD (ext4) sur un PC GNU/Linux en utilisateur root pour aller éditer le fichier /etc/default/autogetty afin de modifier :

```
ENABLED=1
```

en

```
ENABLED=0
```

Ceci empêchera l'utilisation de **/bin/auto-serial-console**, mais ce n'est pas tout. Il faut ensuite se tourner vers **/etc/init/console.conf** pour changer **38400** en **115200** sur la ligne **exec /sbin/getty**. Et c'est là qu'on remarque également que ce service démarre avec les *runlevel* 2, 3, 4 et 5, mais uniquement si appelé dans un conteneur LXC. Il faut donc supprimer la chaîne « **and container CONTAINER=lxc** » du fichier.

Une fois ces modifications apportées, il suffit de démonter le système de fichiers, et de reloger la microSD dans la Parallella pour obtenir un démarrage complet se terminant sur un login. Le nom d'utilisateur et le mot de passe sont linaro/linaro. Cet utilisateur dispose de privilèges permettant d'utiliser **sudo** pour avoir les permissions super-utilisateur.

Le démarrage est cependant perturbé par l'injection de caractères sur la console et un message d'erreur concernant **plymouth-upstart-bridge**. En cherchant un peu, on se rend compte que ce service est simplement un relais passant les notifications de changement d'état d'Upstart reçues via D-Bus vers Plymouth. Et ce dernier se trouve être le frontend graphique masquant les messages du noyau à l'utilisateur (parce que le texte qui défile, ça fait peur aux gens). Or, bien entendu, sur un système *headless*, il n'y a aucune raison d'utiliser Plymouth.

S'en suit donc une suppression du paquet **plymouth** via **apt-get remove --purge** qui découle sur la suppression de quelques 95 paquets dépendants, dont tous les paquets Xorg, mais aussi **dmsetup**, **upstart**, **ppp** ou encore **parted**. Et je passe sur une dépendance non résolue concernant **logrotate** vers **cron**/**anacron**. Qu'un seul *rootfs* soit proposé à la fois pour l'installation standard (HDMI) et *headless* passe encore (de justesse), mais que ce système intègre des éléments qui n'ont clairement rien à faire sur un nano-ordinateur fonctionnant à partir d'une carte SD est intolérable. En tout, ce ne sont pas moins de 1170 paquets qui sont installés à la base, de quoi largement faire du ménage, en particulier pour une utilisation via la console série.

Et c'est là qu'arrive la goutte d'eau qui fait déborder le vase. Au détour de quelques nettoyages plus en profondeur et m'étonnant des insultes que profère le shell à mon endroit, je finis par utiliser :

```
linaro-nano:~> echo $SHELL
/bin/tcsh
```

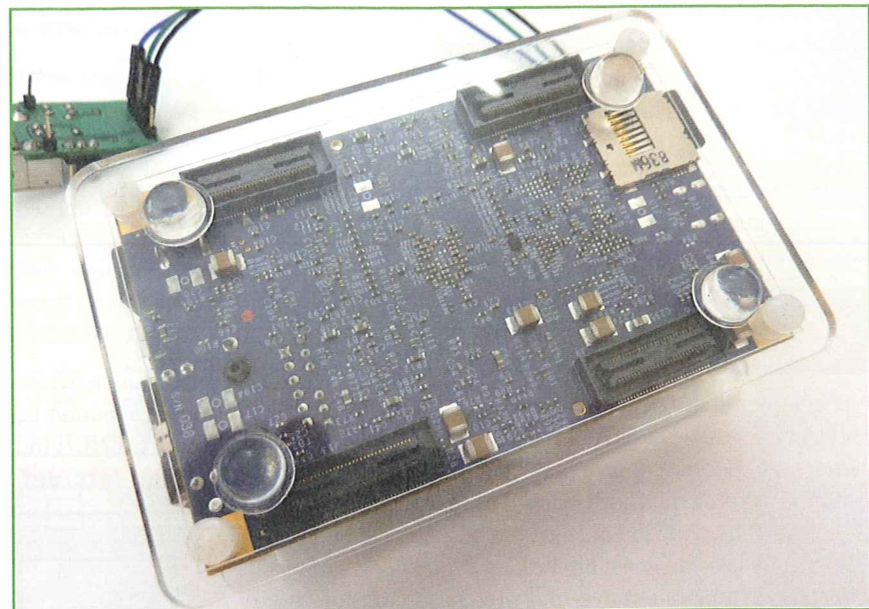
Est-ce une plaisanterie ?

```
linaro-nano:~> cat /etc/passwd | grep linaro
linaro:x:1000:1000:linaro,,,:/home/linaro:/bin/tcsh
```

Non, ce n'en est pas une, le shell **par défaut** de l'utilisateur **par défaut** est bel et bien un shell C. Notez que je n'ai rien contre, j'ai été longtemps utilisateur de Tcsh, mais à une époque où Bash ne proposait pas de fonctionnalités aussi avancées. Tcsh n'a pas évolué depuis trois ans, il est parfaitement incompréhensible qu'on puisse le proposer, l'imposer, par défaut (même Apple est passé à Bash), en particulier lorsqu'on propose une configuration *headless* ou une plateforme se voulant être une introduction aisée au monde du calcul parallèle. Intégrer Tcsh oui, le configurer par défaut non. C'en est trop, au diable Ubuntu, Upstart, Plymouth et Tcsh. Optons pour la méthode des hommes, des vrais avec de la barbe, pleins de poils et qui ne mangent pas de quiche !

4 Debian !

Notre première idée était de nous diriger vers un système léger dédié aux systèmes *headless*, mais ayant des similitudes avec le système officiellement proposé. En regard de ces critères, c'est naturellement Linaro Nano qui a en premier lieu retenu notre attention. Malheureusement, les points négatifs précédemment évoqués sont tout aussi présents avec cette distribution Linaro : basée sur Ubuntu, incluant Upstart, et



Les GPIOs se présentent sous la forme de connecteurs haute densité sous la carte, très peu hacker friendly...

dixit la documentation elinux.org, avec la nécessité de démarrer d'abord le système avec clavier/souris/écran pour ensuite basculer sur le noyau/bitstream en version *headless*. Bref, non merci.

On notera également au passage qu'une autre image « officielle » existe sur le FTP parallella.org, **ubuntu-14.04-headless-z7020-20141110.img.gz**. Cependant, excédé, je n'ai pas pris la peine de la tester, préférant m'en tenir à une solution définitivement plus « propre » : Debian.

Suivant les recommandations détaillées sur http://elinux.org/Parallella_Debian, la création d'une nouvelle carte microSD se résume au téléchargement de l'image **9600-debian7-parallella-190214.img** (~300 Mo) et à l'utilisation de **dd** comme précédemment, avec une microSD de 2 Go minimum. La distribution intègre :

- un système Debian de base comprenant un serveur OpenSSH, NTPdate, **sudo** et Python,

Remarque

Remarquez au passage que le noyau officiel est configuré avec **CONFIG_SYSFS_DEPRECATED=y**, chose clairement incompatible avec udev depuis la version 151 (la 175 qui est actuellement utilisée). Pour profiter pleinement du système, il est donc nécessaire, en principe, de cross-compiler un noyau avec :

```
% git clone https://github.com/parallella/parallella-linux

% cd parallella-linux

% scp <adresse_parallella>:/proc/config.gz

% zcat config.gz > .config

% make CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
ARCH=arm mrproper

% make CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
ARCH=arm oldconfig

% make -j 10 CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
ARCH=arm LOADADDR=0x8000 uImage

% scripts/dtc/dtc -I dts -O dtb -o \
arch/arm/boot/zynq-zed.dtb arch/arm/boot/dts/zynq-zed.dts
```

Notez qu'il existe une configuration par défaut pour la Parallella sous le nom **parallella_defconfig** utilisable en lieu et place du **oldconfig** et de la configuration extraite d'un noyau en fonction sur la plateforme. Mais plus important encore, vous devez disposer de la chaîne de compilation croisée adaptée (un **arm-linux-gnueabi-** standard, mais trop ancien ne fonctionnera pas).

Celle-ci est disponible sur le site de Xilinx, intégrée au *Xilinx Software Development Kit* avec tout un tas d'autres éléments volumineux qui ne sont pas nécessaires pour un développeur Unix digne de ce nom (je pense en particulier à Eclipse). Bien entendu, l'installation du SDK, avec ou sans Vivado Design Suite, nécessite un enregistrement sur le site de Xilinx et l'utilisation de l'installeur du constructeur (hey, Xilinx, vous connaissez GitHub ?).

En suivant la procédure d'installation, on se retrouve néanmoins avec une chaîne de compilation complète et adaptée au SoC Zynq de la Parallella. Dans l'installeur, vous devez désigner un répertoire d'installation (par défaut dans **/opt/Xilinx** qui n'est pas inscriptible par un utilisateur standard). C'est dans ce répertoire, après téléchargement et installation de ~700 Mo d'archives, que tout est installé et qui sert de racine pour l'ensemble des outils. Pour pouvoir utiliser la chaîne de compilation, vous devrez sourcer le script **Xilinx/SDK/2014.4/settings32.sh** avec **.** ou **source** (une version 64 bits de l'installeur est également disponible). Cette commande aura simplement pour effet d'ajouter des nouveaux chemins dans votre **$PATH**.

En invoquant le compilateur directement, on apprend finalement que c'est là le travail de *Mentor Graphics* :

```
% arm-xilinx-linux-gnueabi-gcc --version
arm-xilinx-linux-gnueabi-gcc (Sourcery CodeBench Lite
2014.05-23) 4.8.3 20140320 (prerelease)
Copyright (C) 2013 Free Software Foundation, Inc.
```

La version de la chaîne de compilation de Mentor Graphics est traditionnellement « en avance » sur GCC. La philosophie de cette société, qu'on l'apprécie ou pas, consiste à utiliser une chaîne GNU « améliorée » pour servir de base à ses produits. Ce n'est qu'ensuite que les modifications apportées sont « contribuées » *upstream* et donc intégrées dans la chaîne de compilation GNU. Il n'est donc pas absolument nécessaire d'utiliser la chaîne fournie par Xilinx, mais une version récente (4.9.x par exemple) des outils GNU sera indispensable.

L'ensemble de l'installation fait environ 4,8 Go, vous vous retrouvez avec une *n-ième* chaîne de compilation (si vous utilisez déjà comme moi baremetal + Linux GnuEABI + uClinux) et disposez d'un environnement complet dont vous ne comptez pas vous servir, mais ça fonctionne (sans permission **root** et sans saccager la gestion de paquets de votre distribution, c'est déjà ça).

Les éléments à copier sur la microSD post-compilation sont **arch/arm/boot/uImage** et **arch/arm/boot/zynq-zed.dtb** (renommé **devicetree.dtb**). Notez que les sources GitHub sont, bien entendu, les sources en cours de développement, ce qui implique des problèmes potentiels de stabilité.

- une chaîne de compilation *armhf* complète avec les AutoTools et Cmake,
- le SDK Epiphany,
- le SDK COPRTHR (prononcer *copper threads*).

Le démarrage avec une console série ne pose aucun problème et on arrive directement sur un shell **root**. Un utilisateur **epiphany** existe cependant, avec comme mot de passe **parallella**.

Dès ce premier démarrage, on s'empressera de se mettre à notre aise en installant un véritable éditeur complet (Vim en lieu et place de **vim-tiny**) et en procédant aux ajustements habituels avant de créer un nouvel utilisateur dédié à nos expérimentations. Il conviendra également d'éditer le fichier **/etc/inittab** de manière en commenter la dernière ligne (débutant par **T0:2345:respawn:/bin/login**) et de décommenter celle concernant le **getty** en console série (sud **ttyPS0**) :

```
T0:2345:respawn:/sbin/getty -L ttyPS0 115200 vt100
```

Ceci nous permettra de désactiver l'auto-login en **root**. Enfin, on prendra le temps de mettre à jour le bitstream pour le FPGA, le blob du *device tree* et l'image du noyau sur la partition FAT de la microSD.

Arrêtons là de nous battre avec le système installé, car arriver à un résultat parfais non seulement demande beaucoup de travail, mais en plus dépasse largement le cadre du présent article. On relèvera cependant qu'il est vraiment dommage que l'orientation initiale choisie, à savoir un système *desktop end-user*, ait laissé si peu de place pour une architecture minimaliste plus représentative du monde de l'embarqué et de ses usages.

5 Développement et SDK

Quel que soit le système installé, Ubuntu ou Debian, les images proposées fournissent généralement un compilateur natif ainsi que l'Epiphany ESDK constitué d'une chaîne de compilation dédiée au coprocesseur Epiphany et de bibliothèques permettant à la fois de créer du code exécuté par ce dernier et par le système hôte (ARM). L'ESDK est installé dans **/opt/adapteva** et un lien symbolique est présent, **esdk**, pointant vers le répertoire contenant effectivement les éléments (**esdk.5.13.09.10**). Notez qu'il existe une version plus récente du ESDK, mais son installation implique celle d'une nouvelle version du noyau ainsi que du bitstream pour le FPGA et donc un lot important de manipulations et implicitement quelques nouveaux tours dans la boucle essai/validation.

Pour pouvoir utiliser le SDK, il faut impérativement définir un certain nombre de variables d'environnement pour l'utilisateur courant. Ainsi, si vous avez comme moi créé un nouvel utilisateur, les lignes suivantes devront être ajoutés dans son **~/.bashrc** :

```
EPIPHANY_HOME=/opt/adapteva/esdk
. ${EPIPHANY_HOME}/setup.sh
```

Le script **setup.sh** initialise ou modifie plusieurs variables d'environnement :

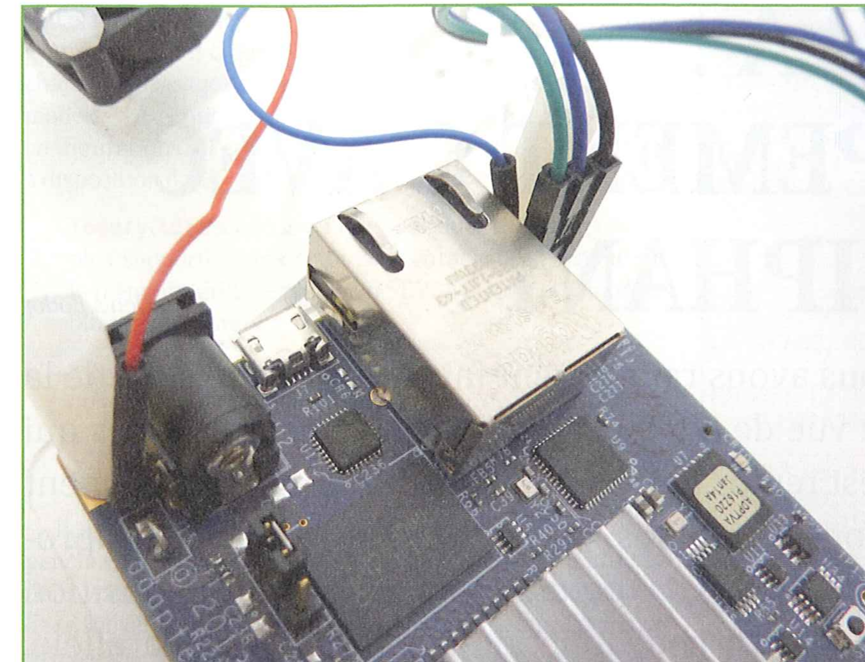
- **PATH** : le chemin de recherche pour les commandes afin de permettre l'utilisation du compilateur GCC dédié au coprocesseur Epiphany,
- **LD_LIBRARY_PATH** : le chemin de recherche pour l'éditeur de liens permettant de trouver les bibliothèques partagées du ESDK,
- **EPIPHANY_HOME** : la racine de l'arborescence du ESDK,
- **EPIPHANY_HDF** : le fichier de description de la plateforme spécifiant la version ainsi qu'une énumération de la configuration présente (nombre de coprocesseurs, types, registres, etc.).

Une fois tout cela correctement initialisé, nous n'avons plus qu'à regarder du côté de GitHub afin de récupérer les différents exemples à la fois officiellement proposés, mais aussi contribués par des utilisateurs de la plateforme. Si l'on dispose d'une configuration réseau fonctionnelle sur la Parallella, le plus rapide est simplement d'utiliser **git** :

```
% git clone https://github.com/adapteva/
epiphany-examples.git
% cd epiphany-examples
% git checkout 2014.05
HEAD is now at 7dcf01e... Removing -le-loader
  reference inside mk files (temp hack)
```

Notez qu'on se place sur une version du dépôt correspondant au tag « 2014.05 ». Les sources GitHub reflétant les développements en cours, la branche *master* actuelle correspond aux exemples reposant sur la version de développement (découlant de « 2015.1 ») où la communication système/Epiphany passe par un périphérique **/dev/epiphany** et non **mmap**. Cette évolution règle le problème de permissions obligeant l'exécution des binaires en tant que **root** puisqu'il est possible alors avec udev de spécifier des permissions plus précises. Cependant, l'utilisation des exemples en version plus récente implique l'utilisation d'un ESDK également plus récent et donc d'un noyau en version de développement, et donc une chaîne de compilation Xilinx, etc. Mieux vaut faire connaissance avec la plateforme dans sa version actuellement stable et prévoir ensuite éventuellement de faire migrer ses projets sur la prochaine version une fois stabilisée.

Nous pouvons maintenant nous pencher sur les exemples et en particulier l'incontournable *Hello World* se trouvant


L'alimentation du ventilateur se fait via les trous du support de fixation de la carte et ne sera fonctionnelle que dans le cas d'une alimentation par un « bloc secteur » et non en cas de connexion USB.

dans **apps/hello-world**. La compilation et l'exécution des exemples sont gérées par des scripts shell et non des **Makefile**. Ainsi, pour compiler et exécuter les exemples, il suffira d'utiliser les commandes :

```
% cd apps/hello-world
% ./build.sh
% ./run.sh
  0: Message from eCore 0x8ca ( 3, 2): "Hello World from core 0x8ca!"
  1: Message from eCore 0x84b ( 1, 3): "Hello World from core 0x84b!"
  2: Message from eCore 0x84b ( 1, 3): "Hello World from core 0x84b!"
  3: Message from eCore 0x888 ( 2, 0): "Hello World from core 0x888!"
[...]
17: Message from eCore 0x84b ( 1, 3): "Hello World from core 0x84b!"
18: Message from eCore 0x848 ( 1, 0): "Hello World from core 0x848!"
19: Message from eCore 0x8ca ( 3, 2): "Hello World from core 0x8ca!"
```

Le script **build.sh** n'a rien de particulier et ne fait qu'utiliser les variables d'environnement précédemment définies via **setup.sh**/**.bashrc**, pour lancer les différentes commandes de compilation et d'édition de liens (**gcc** local, **e-gcc**, **e-objcopy**, etc.). **run.sh** procède de même pour lancer l'un des binaires ELF (dans **./Debug**) via **sudo** et en spécifiant le chemin de recherche des bibliothèques partagées. Certains diraient que c'est un peu « bricolo », mais le fait est que cela fonctionne.

L'architecture générale des sources, et du développement, repose sur deux fichiers sources : un pour l'Epiphany et l'autre pour le processeur hôte (le SoC ARM). Le premier est un source C compilé avec **e-gcc** (fourni par l'ESDK) dont l'ELF résultant est transformé avec **e-objcopy** en fichier au format SREC (Motorola S-Record) : **e_hello_world.srec**. Ce fichier est « chargé » dans le coprocesseur Epiphany par le source C pour l'hôte qui en contrôle l'exécution. Il s'agit d'un source C parfaitement standard, compilé par le **gcc** natif ARM, mais utilisant les fonctions mises à disposition par l'API et les bibliothèques du ESDK.

Conclusion intermédiaire

Arrêtons là cette première partie orientée vers la découverte de la plateforme et du système afin de nous pencher plus sereinement sur l'utilisation du coprocesseur Epiphany et son SDK dans l'article suivant. Que dire pour résumer nos impressions générales si ce n'est que cette initiative mériterait une approche qualitative plus globale du « package » Parallella. Certes, Adapteva est une société spécialisée dans les processeurs et en particulier ceux dédiés au calculs multicœurs, et non dans le développement de systèmes embarqués complets et de BSP Linux. Il est donc tout à fait compréhensible de constater que certains choix qui ont été faits ne semblent pas pertinents ou adaptés à un public peu réceptif à une orientation trop « nano-ordinateur desktop ». Il n'en reste pas moins que ceci pourra en décourager plus d'un, car même si le domaine du calcul parallèle est très attirant, il faudra, selon ses propres préférences, tout d'abord franchir ce qui peut bel et bien être perçu comme un obstacle. Dans l'état, la Parallella permet d'appréhender facilement le domaine si on est un développeur « desktop », mais demandera un travail préliminaire important à un utilisateur avec davantage d'expérience Unix, ce qui est contre-intuitif.

Bien entendu, une fois le système complètement adapté à leurs besoins, les deux « types » d'utilisateurs se retrouvent à nouveau sur un pied d'égalité et pourront profiter de l'Epiphany et du SDK de la même manière : l'un avec un environnement équivalent à une Ubuntu PC et l'autre dans un contexte plus traditionnel de développement sur système embarqué. ■

PARALLELLA : DÉVELOPPEMENT C AVEC LE SDK EPIPHANY

Denis Bodor

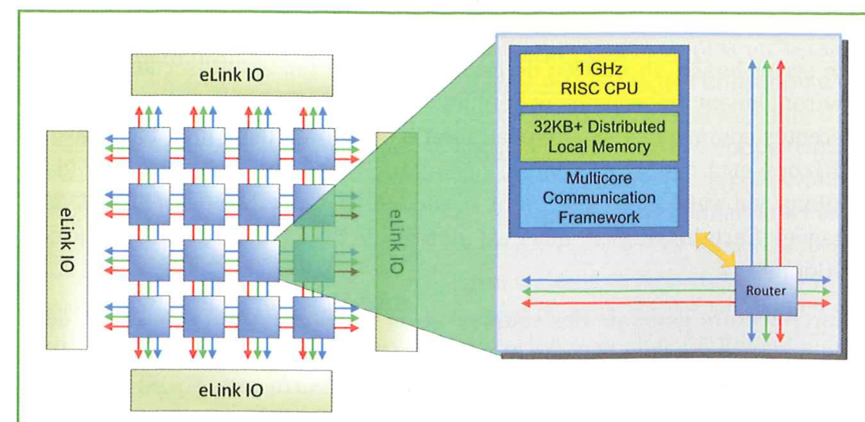
Dans l'article précédent, nous avons rapidement fait un tour d'horizon de la carte Parallella du point de vue de la plateforme et du ou des systèmes qui peuvent l'accompagner. Il est temps à présent d'aborder le point réellement important concernant ses fonctionnalités. Je parle, bien entendu, du coprocesseur Epiphany d'Adapteva, de ses 16 cœurs et du SDK mis à disposition pour s'initier à la programmation de code fonctionnant de manière parallèle.

Quel que soit le système GNU/Linux que vous avez installé sur la microSD de votre Parallella, ou même si vous n'avez pas de Parallella du tout, il est possible de développer pour le coprocesseur Epiphany. Bien entendu, dans le second cas, ce sera en aveugle, car bien que le SDK Epiphany puisse être installé sur une machine x86 pour la compilation croisée, vous ne serez pas en mesure de tester le code sans carte. Pour l'heure, il n'existe pas d'émulateur et à cette date aucun projet n'est en cours dans ce sens.

1 Architecture et principe de fonctionnement

Un code utilisant l'Epiphany se compose de deux éléments, un code exécuté sur l'hôte (l'ARM Cortex-A9) et un, chargé par le code hôte et devant s'exécuter sur un ou plusieurs cœurs Epiphany. Le code pour les cœurs, comme celui pour l'hôte, est écrit en C. Chaque source est compilée avec les outils propres à la cible. Le code hôte est à la charge de la chaîne de compilation locale pour ARM (**gcc**) et le code Epiphany est l'affaire de la chaîne de compilation livrée avec le SDK (**e-gcc**). Le code hôte utilise une fonction dédiée pour charger le binaire dans un ou plusieurs cœurs Epiphany sous la forme d'un fichier SREC produit avec le **e-objcopy** du SDK.

Un certain nombre d'exemples de codes est diffusé via le dépôt GitHub d'Adapteva (https://github.com/adapteva/epiphany-examples ou https://github.com/parallella/parallella-examples). C'est là une source d'inspiration conséquente qu'il s'agisse de petits projets (*HelloWorld*, **eprime**, **matmul-16**) ou d'applications complètes (*Mandelbrot*, FFT, traitements Kinect, John the Ripper, etc.). Un seul point important est à retenir concernant ces exemples : la branche *master* de ces dépôts est naturellement celle de développement et le code présent est donc adapté aux versions de développement du BSP Parallella, et donc à un noyau et un *bitstream* à jour. Pour que l'ensemble de ces codes soit compilable et utilisable avec les images systèmes « stables » actuelles, il est important de vous caler sur une version correspondante des dépôts. Le tag **2014.05** est généralement un bon choix, mais, bien entendu, cela dépendra du moment où vous lirez ceci et procéderez à vos expérimentations.



Architecture générale du processeur Epiphany (source : documentation Adapteva).

Le code hôte n'a pas pour seule tâche de charger le programme dans les cœurs Epiphany, il en gère l'exécution et se charge du passage des paramètres et des données. Plusieurs modèles d'exécution sont utilisables et réglés par le fichier de configuration de l'éditeur de liens. Trois scripts **ld** sont mis à disposition dans **/opt/adapteva/esdk/bsps/current/** :

- **legacy.ldf** est utilisé pour le code qui n'est actuellement plus supporté. Dans cette configuration, l'ensemble de la mémoire utilisée réside en SDRAM (code et données, bibliothèque standard et pile),
- **internal.ldf** est une configuration exactement inverse où les cœurs n'utilisent que la mémoire SRAM dédiée,
- **fast.ldf** est le modèle hybride avec le code, les données et la pile en SRAM, mais la bibliothèque standard en SDRAM.

Il n'est pas nécessaire, dans un premier temps, de vous pencher sur le contenu de ces scripts qu'il suffit d'utiliser dans votre processus de compilation, mais il est important de se souvenir de la structure globale. Chaque cœur Epiphany dispose de 32Ko de mémoire locale divisée en 4 banques de 8Ko et fournit un espace d'adresses total entre **0x0000** and **0x7FFF** pour chaque cœur d'un *Epiphany-III 16-core* (E16G301) tel qu'intégré à la Parallella. Notez également que cet espace est *plat* et qu'un cœur peut en principe accéder à la mémoire d'un autre en étendant l'espace d'adressage. Les bits « inutilisés » de l'adresse servent à identifier la ligne et la colonne de chaque cœur dans l'Epiphany. L'adresse ainsi obtenue est appelée « adresse globale », mais, là encore, vous n'aurez pas, dans un premier temps, à vous en soucier puisque les scripts de l'éditeur de liens se chargent de tout cela. Ce n'est qu'après la phase de prise en main qu'il pourra être intéressant de faire communiquer ainsi les cœurs entre eux pour obtenir un maximum de performances. Ici, nous nous contenterons d'utiliser les avantages de l'éditeur de liens ainsi que les fonctions d'abstraction (HAL) de plus haut niveau.

2 Les scripts shell c'est bien, un vrai Makefile c'est mieux

Si vous parcourez les différents exemples distribués par Adapteva, vous constaterez avec surprise que le processus de compilation repose sur des scripts shell. Certains pourraient argumenter que la différence entre scripts et *Makefiles* est, somme toute, relativement réduite, mais, que voulez-vous, lorsqu'on développe depuis un certain tant sous GNU/Linux, on acquière des habitudes (ou des obsessions). Ainsi, plutôt que d'utiliser de tels scripts qui ne semblent pas vraiment naturels, nous avons préféré composer un bon vieux **Makefile** :

```
HTARGET := hello_world
ETARGET := e_hello_world
WARN    := -Wall

ESDK=${EPIPHANY_HOME}
LIBSPATH=${ESDK}/tools/host/lib
LIBS=-le-hal
INCS=${ESDK}/tools/host/include
ELDF=${ESDK}/bsps/current/fast.ldf
ELIBS=-le-lib
EHDF=${EPIPHANY_HDF}

CC      := gcc
ECC     := e-gcc
EOBJCOP := e-objcopy

all: ${HTARGET} ${ETARGET}.srec

${HTARGET}: ${HTARGET}.c
	${CC} ${WARN} ${HTARGET}.c -I ${INCS} -L ${LIBSPATH} ${LIBS} -o $@

${ETARGET}.srec: ${ETARGET}
	${EOBJCOP} --srec-forceS3 --output-target srec ${ETARGET} \
	${ETARGET}.srec

${ETARGET}: ${ETARGET}.c
	${ECC} ${WARN} -T ${ELDF} ${ETARGET}.c ${ELIBS} -o $@

run: ${ETARGET}.srec ${HTARGET}
	sudo -E LD_LIBRARY_PATH=${LIBSPATH} \
	EPIPHANY_HDF=${EPIPHANY_HDF} ./${HTARGET}

clean:
	rm -rf *.o ${HTARGET}  ${ETARGET} ${ETARGET}.srec
```

Nous n'avons ici fait que reprendre les éléments déjà présents dans les scripts **build.sh** et **run.sh** accompagnant les exemples officiels. L'installation du SDK dans **/opt** et la modification de variables d'environnement comme **LD_LIBRARY_PATH** n'est pas quelque chose qui nous paraît très élégant, pas plus que la nécessité de lancer le binaire via **sudo**, mais pour l'heure, avec la version actuelle du système et du SDK, c'est le mieux qu'on puisse faire. D'après les informations tirées des versions de développement, ceci devrait bientôt changer (via l'utilisation de **/dev/epiphany** par exemple).

Ce **Makefile** définit les cibles :

- **all** : pour la compilation des code hôtes et pour les cœurs Epiphany,
- **run** : pour l'exécution, via **sudo**, du code hôte,
- **clean** : pour le classique nettoyage des sources.

3 Programmation pour les cœurs Epiphany

Il est temps de nous pencher sur le code lui-même. Nous commençons par celui qui typiquement est le plus concis, celui pour les cœurs Epiphany. Notre code de démonstration sera relativement simple et consistera à procéder à une multiplication de deux valeurs entières non signées passées en argument et à « retourner » le résultat. Ce code est identique

pour tous les cœurs Epiphany, mais ceux-ci recevront des paramètres différents, choisis aléatoirement.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <e_lib.h>

int main(void) {
   unsigned *a, *b, *res, *done;

   a = (unsigned *) 0x2000;
   b = (unsigned *) 0x4000;
   res = (unsigned *) 0x6000;
   done = (unsigned *) 0x7000;

   (*(res)) = (*(a)) * (*(b));
   (*(done)) = 1;

   __asm__ __volatile__("idle");

   return 0;
}
```

Afin de procéder aux échanges de données, nous définissons un ensemble de pointeurs vers des entiers non signés : **a** et **b** pointent les valeurs à multiplier entre elles, **res** pointe le résultat que nous allons produire et enfin **done** sert à désigner le drapeau marquant l'accomplissement de la tâche par le cœur.

Le code en lui-même est on en peut plus simple puisqu'il nous suffit d'initialiser les différents pointeurs avec l'adresse où se trouvent ou doivent se trouver les données à lire et écrire, puis à utiliser ces informations pour le calcul. Enfin, ceci fait, nous plaçons la valeur **1** dans l'emplacement désigné par **done** avant de mettre le cœur en veille. Lorsque l'exécution de ce code est terminée, il nous suffira, depuis le programme hôte, de lire le contenu à l'adresse **0x7000** de l'espace local pour savoir si le calcul est fait, puis lire la valeur à **0x6000**.

Ces adresses ne sont pas choisies par hasard. Chaque cœur Epiphany dispose de son espace de 32Ko divisé en 4 banques :

- la banque 0 à **0x0000** contient le code à exécuter et l'IVT (*Interrupt Vector Table*)
- la banque 1 à **0x2000**,
- la banque 2 à **0x4000**,
- la banque 3 à **0x6000**.

Comme le précise la documentation d'Adapteva (*Epiphany SDK Reference*), il est de bon ton d'utiliser des banques distinctes pour le code et les données. Nous reprenons ici le principe présent dans les exemples en utilisant plusieurs banques dans le seul but de présenter l'architecture. Stocker un simple **unsigned** par banque n'est absolument pas efficace. Pour un code plus avancé, l'utilisation d'une structure mappée sur **0x2000** est un choix plus pertinent.

Un autre point important est à prendre en compte lors qu'on parle de la mémoire, en particulier, lorsqu'on dispose uniquement de 32 Ko : la pile. Chaque cœur avec les scripts **fast.ldf** et **internal.ldf** dispose de sa pile en SRAM (ou mémoire interne) et plus précisément ici : **PROVIDE (__stack_start_ = ORIGIN(INTERNAL_RAM) + LENGTH(INTERNAL_RAM) - 0x10);** dixit les scripts de l'éditeur de liens. En clair, la pile se trouve à l'adresse la plus haute (moins **0x10**) et grossit vers le bas. Attention donc à la manière dont vous utilisez la banque 3 et en particulier avec un offset important. Dans notre exemple, l'utilisation de l'adresse **0x7000** n'est pas un problème étant donné la simplicité du code, mais si vous commencez à jouer avec la récursivité et beaucoup de données mieux vaut faire attention...

4 Programme hôte

Attaquons dans la foulée le code pour l'hôte. Celui-ci est chargé de préparer les données, initialiser le système et l'environnement, configurer le mode d'exécution, passer les données pour les cœurs, charger le code, déclencher le calcul, attendre la fin de l'exécution et collecter les résultats. Comme pour n'importe quel programme utilisant une API quelconque, on commence par planter le décor :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

#include <e-hal.h>

int main(int argc, char *argv[])
{
  unsigned int i,j;
  e_platform_t platform;
  e_epiphany_t dev;

  unsigned int a;
  unsigned int b;
  unsigned int res;
  unsigned int done;
  unsigned int alldone = 0;

  srand(time(NULL));
```

Rien de bien significatif ici si ce n'est l'utilisation de **e-hal.h** nous donnant accès aux fonctions de haut niveau de la couche d'abstraction matérielle. Les choses sérieuses commencent ensuite. Avant toutes choses nous devons initialiser notre environnement :

```
  if(e_init(NULL) == E_OK) {
    printf("(i) e_init done.\n");
  } else {
    fprintf(stderr, "Error initialising\n");
    return(EXIT_FAILURE);
  }
```

e_init() initialise les structures de données et établit la connexion avec le coprocesseur Epiphany. Cette fonction peut prendre en argument un pointeur vers une chaîne désignant un fichier de description matérielle fourni avec le SDK. Si ce pointeur est **NULL**, la chaîne est automatiquement obtenue depuis la variable d'environnement **EPIPHANY_HDF** (voir cible **run** du **Makefile**). Nous pouvons ensuite réinitialiser le « sous-système » :

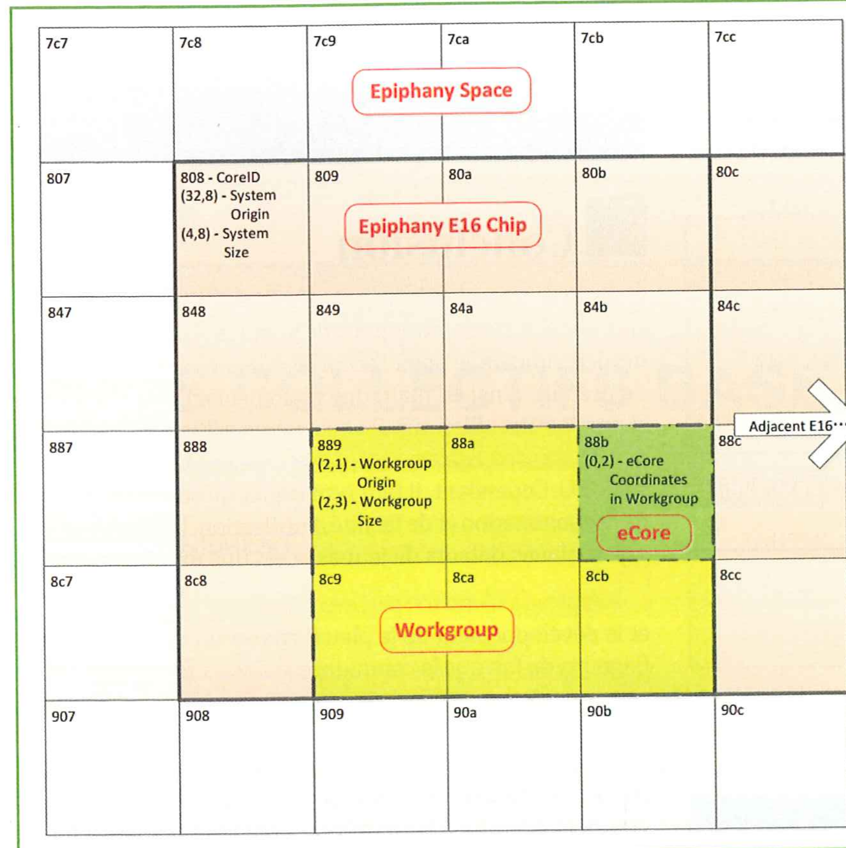
```
  if(e_reset_system() == E_OK) {
    printf("(i) e_reset_system done.\n");
  } else {
    fprintf(stderr, "Error system reset\n");
    return(EXIT_FAILURE);
  }
```

Ceci a pour effet de procéder à un reset complet à la fois du coprocesseur Epiphany et de l'interface en FPGA (*glue logic*). Il est temps ensuite de piocher quelques informations à retourner à l'utilisateur à propos du matériel en présence :

```
  if(e_get_platform_info(&platform) == E_OK) {
    printf("(i) e_get_platform_info done.\n");
  } else {
    fprintf(stderr, "Error getting plateform info\n");
    return(EXIT_FAILURE);
  }
  printf("========================\n");
  printf("== Using platform version : %s\n", platform.version);
  printf("== with %ux%u cores\n", platform.rows, platform.cols);
  printf("== %u chip(s) and %u external memory segment(s)\n",
     platform.num_chips, platform.num_emems);
  printf("========================\n");
```

Après cet interlude purement esthétique, nous pouvons accéder à l'Epiphany. Ici, la notion de groupe de travail (*workgroup*) est capitale. Il est en effet possible de faire travailler les cœurs Epiphany individuellement ou de concert, ou plus exactement d'orchestrer la gestion de manière groupée ou non. Pour notre exemple, nous voulons que l'ensemble des cœurs soit utilisé avec le même code et non accéder à chacun d'eux successivement comme c'est le cas dans l'exemple **hello-world** « officiel ». Nous formons donc un groupe de travail ayant pour origine le cœur aux coordonnées **0,0** et s'étendant sur l'ensemble de la matrice de cœurs disponibles :

```
  if(e_open(&dev, 0, 0, platform.rows, platform.cols) == E_OK) {
    printf("(i) e_open done.\n");
  }
```



Agencement de l'architecture des cœurs du coprocesseur Adapteva Epiphany. Le modèle E16 dispose de 6 cœurs accessibles individuellement (en vert), mais qui peuvent être assemblés sous forme de groupe de travail (en jaune). Chaque cœur dispose donc de coordonnées dans l'espace Epiphany, mais aussi au sein du groupe. (source : Epiphany SDK Reference – p73)

Nous obtenons en cas de succès un *handle* sous la forme d'une structure de type **e_epiphany_t** qui sera utilisée ensuite pour accéder au groupe. Nous pouvons alors parcourir tous les cœurs et initialiser les données pour chacun d'eux :

```
  for (i=0; i<platform.rows; i++) {
    for (j=0; j<platform.cols; j++) {
      a = (rand() % 1000)+1;
      b = (rand() % 1000)+1;
      res = 0;
      done = 0;
      e_write(&dev, i, j, 0x2000, &a,
sizeof(a));
      e_write(&dev, i, j, 0x4000, &b,
sizeof(b));
      e_write(&dev, i, j, 0x6000, &res,
sizeof(res));
      e_write(&dev, i, j, 0x7000, &done,
sizeof(done));
    }
  }
```

Là encore, ce code est purement démonstratif et sert principalement d'exemple pour l'utilisation de la fonction **e_write()** qui permet la **copie** de données depuis le programme hôte vers l'espace mémoire d'un cœur Epiphany. On lui passe respectivement en argument notre point d'accès au groupe de travail, les coordonnées ligne et colonne du cœur concerné, l'offset où écrire les données dans l'espace mémoire du cœur, un pointeur sur la donnée à copier et finalement sa taille. Notez qu'il s'agit d'une **copie** et non d'un passage de pointeur. Remarquons ici que nous utilisons la *Epiphany Hardware Abstraction Layer* (eHAL), mais qu'une fonction homonyme existe dans la *Epiphany Hardware Utility Library* (eLib) avec un ordre différent pour les arguments. L'eHAL met à disposition des fonctions pour communiquer avec l'Epiphany alors que l'eLib permet de configurer et interroger les ressources matérielles de l'Epiphany (**e_group_config** ou **e_emem_config**). Cette homonymie est, je trouve, un véritable piège à la fois dans le développement et dans la lecture des documentations...

Nos données sont en place, il ne nous reste plus qu'à charger le code dans les cœurs. Pour cela, du fait de l'utilisation d'un groupe de travail, nous n'avons qu'une simple fonction à invoquer :

```
if(e_load_group("e_hello_world.srec", &dev, 0, 0,
        platform.rows, platform.cols, E_TRUE) == E_OK) {
    printf("(i) core(s) loaded and started !\n");
} else {
    fprintf(stderr, "Error loading/starting eCore code\n");
    return(EXIT_FAILURE);
}
```

Cette fonction permet d'utiliser des sous-groupes en plus de spécifier le fichier SREC à charger. Ici, nos 16 cœurs seront à l'œuvre et notre sous-groupe est composé de l'ensemble des cœurs du groupe, et donc de l'Epiphany lui-même. Le dernier argument, est un `e_bool_t` qui, s'il est à `E_TRUE` indique que le code doit être immédiatement exécuté après chargement. Si l'argument est à `E_FALSE`, vous pourrez lancer l'exécution par la suite, pour tout le groupe, avec `e_start_group(&dev)`.

Nous partons alors dans l'attente des cœurs sous la forme d'une double-boucle visant à collecter les valeurs `done` pour des éléments du groupe de travail :

```
while(alldone != 0xffff) {
  for (i=0; i<platform.rows; i++){
    for (j=0; j<platform.cols;j++){
      e_read(&dev, i, j, 0x7000, &done, sizeof(done));
      if(done) {
        printf("(i) core %u,%u is done\n", i, j);
        alldone = alldone | (1 << (i*platform.cols+j));
      }
    }
  }
}
```

Une fois `alldone` à la bonne valeur, il est temps de relever les données obtenues et de les afficher :

```
for (i=0; i<platform.rows; i++){
  for (j=0; j<platform.cols;j++){
    e_read(&dev, i, j, 0x2000, &a, sizeof(a));
    e_read(&dev, i, j, 0x4000, &b, sizeof(b));
    e_read(&dev, i, j, 0x6000, &res, sizeof(res));
    printf("result from core %u,%u with args a=%u b=%u : %u\n",
      i, j, a, b, res);
  }
}
```

On n'oubliera pas, enfin, de terminer proprement le programme :

```
e_close(&dev);
e_finalize();
return(EXIT_SUCCESS);
}
```

La compilation et l'exécution de notre petit *HelloWorld* maison se feront simplement avec un simple `make && make run` et nous retournera une sortie ressemblant à :

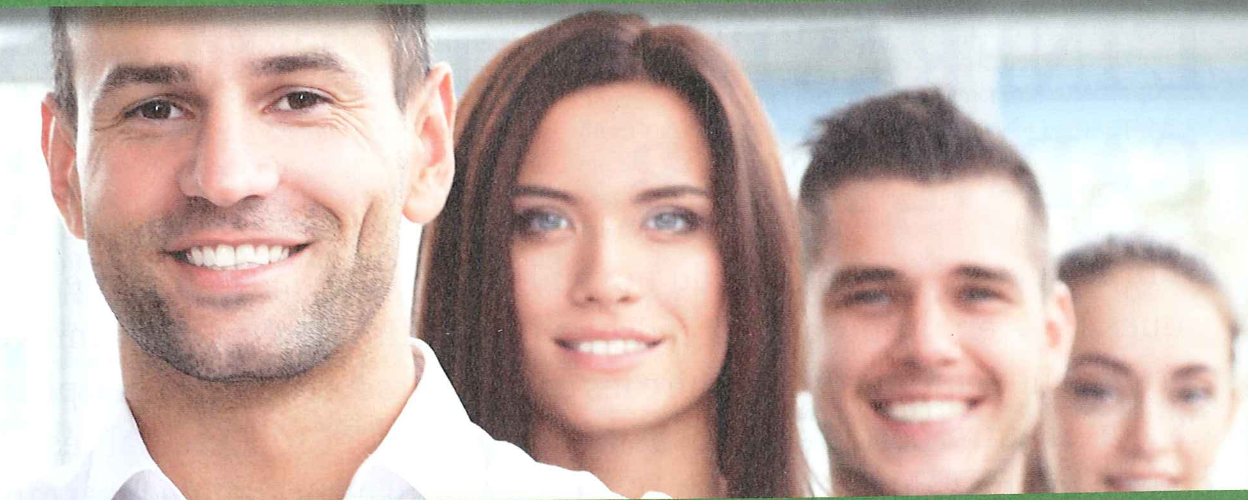
```
(i) e_init done.
(i) e_reset_system done.
(i) e_get_platform_info done.
========================
== Using platform version : PARALLELLA1601
== with 4x4 cores
== 1 chip(s) and 1 external memory segment(s)
========================
(i) e_open done.
(i) core(s) loaded and started !
(i) core 0,0 is done
(i) core 0,1 is done
(i) core 0,2 is done
(i) core 0,3 is done
(i) core 1,0 is done
(i) core 1,1 is done
(i) core 1,2 is done
(i) core 1,3 is done
(i) core 2,0 is done
(i) core 2,1 is done
(i) core 2,2 is done
(i) core 2,3 is done
(i) core 3,0 is done
(i) core 3,1 is done
(i) core 3,2 is done
(i) core 3,3 is done
result from core 0,0 with args a=11 b=477 : 5247
result from core 0,1 with args a=222 b=111 : 24642
result from core 0,2 with args a=431 b=462 : 199122
result from core 0,3 with args a=788 b=747 : 588636
result from core 1,0 with args a=202 b=138 : 27876
result from core 1,1 with args a=718 b=253 : 181654
result from core 1,2 with args a=870 b=103 : 89610
result from core 1,3 with args a=518 b=596 : 308728
result from core 2,0 with args a=207 b=67 : 13869
result from core 2,1 with args a=621 b=803 : 498663
result from core 2,2 with args a=466 b=166 : 77356
result from core 2,3 with args a=661 b=755 : 499055
result from core 3,0 with args a=286 b=173 : 49478
result from core 3,1 with args a=243 b=669 : 162567
result from core 3,2 with args a=377 b=87 : 32799
result from core 3,3 with args a=944 b=740 : 698560
```

Conclusion

Utiliser une plateforme *dual* ARM-Cortex A6 + FPGA + coprocesseur 16 cœurs pour de simples multiplications est un peu ridicule je vous l'accorde. Mais c'est un réel plaisir de prendre ainsi en main une plateforme d'une nature aussi particulière. Bien sûr, il vous est possible de découvrir le domaine d'une autre manière, avec votre carte graphique et son GPU. Cependant, il faut remarquer qu'en termes de prix, de consommation et de facilité d'utilisation, la Parallella malgré quelques défauts de jeunesse est une excellente option.

Adapteva a réussi son pari avec sa campagne de *crowfunding* et le développement de la plateforme et du SDK se poursuit. Gageons du fait que la communauté s'étant formée autour de cette carte va encore croître et améliorer à la qualité logicielle. Sans doute cet article participera à cette croissance, du moins je l'espère, car pour l'heure, bien que la carte bénéficie d'une bonne réputation, il lui manque le coup d'éclat qui la fera vraiment décoller. Peut-être ceci arrivera-t-il sous la forme d'une *killer app* que vous-même aurez développé... autour du traitement de signal, du repliement de protéines ou encore du calcul distribué. Si vous êtes en manque d'inspiration, jetez un œil à https://www.parallella.org/ideas. ■

PROFESSIONNELS !



DÉCOUVREZ NOS NOUVELLES OFFRES D'ABONNEMENTS ...

PDF COLLECTIFS

OFFRE	ABONNEMENT	PROFESSIONNELS 1 - 5 lecteurs Réf	Tarif TTC	6 - 10 lecteurs Réf	Tarif TTC	11 - 25 lecteurs Réf	Tarif TTC
PROOS2	4n° OS	PRO OS2/5	120,-	PRO OS2/10	240,-	PRO OS2/25	480,-

Prix TTC en Euros / France Métropolitaine

PROFESSIONNELS : N'HÉSITEZ PAS À NOUS CONTACTER POUR UN DEVIS PERSONNALISÉ PAR E-MAIL : abopro@ed-diamond.com OU PAR TÉLÉPHONE : 03 67 10 00 20

ACCÈS COLLECTIFS BASE DOCU

OFFRE	ABONNEMENT	PROFESSIONNELS 1 - 5 connexion(s) Réf	Tarif TTC	6 - 10 connexions Réf	Tarif TTC	11 - 25 connexions Réf	Tarif TTC
PROOS+3	OS	PRO OS+3/5	90,-	PRO OS+3/10	180,-	PRO OS+3/25	360,-
PROH+3	GLMF + HS + LP + HS + MISC + HS + OS	PRO H+3/5	447,-	PRO H+3/10	894,-	PRO H+3/25	1788,-

Prix TTC en Euros / France Métropolitaine

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France HS = Hors-Série LP = Linux Pratique OS = Open Silicium

...EN VOUS CONNECTANT À L'ESPACE DÉDIÉ AUX PROFESSIONNELS SUR : www.ed-diamond.com

L'INFRASTRUCTURE LINUX GADGET USB

Éric Lacombe

L'infrastructure Gadget USB du noyau Linux facilite la création de périphériques USB, en proposant un cadre et un certain nombre de primitives qui permettent d'une part d'abstraire les contrôleurs matériels de périphériques USB, et d'autre part d'en utiliser les ressources et fonctionnalités afin de créer n'importe quelle fonction USB désirée : qu'elle réponde aux standards (tels que « Mass Storage », « CDC Eth »...), qu'elle soit l'incarnation de vos besoins particuliers, ou bien encore qu'elle soit une combinaison des deux.

1 Quelques notions d'USB

1.1 Vue d'ensemble

L'USB (*Universal Serial Bus*) est une interface de communication entre un système hôte (comme un ordinateur) et un ou plusieurs périphériques. Ces périphériques (127 au maximum par contrôleur USB) sont connectés à cet hôte sous forme arborescente. Les feuilles sont des périphériques finaux, alors que les nœuds intermédiaires sont des hubs USB. Le système hôte se compose d'un contrôleur USB (responsable des aspects bas niveau du protocole) et d'un hub, appelé *root hub*, lequel est directement connecté à ce contrôleur.

Un périphérique USB est constitué d'une ou plusieurs fonctions (clavier, *mass storage*, souris, etc.). Chaque périphérique se voit attribuer une adresse unique par l'hôte, laquelle est utilisée au cours des échanges entre l'hôte et le périphérique. Ces échanges se font au travers de ce qui est appelé un *pipe*. Il s'agit d'un canal de communication entre un hôte et un tampon de mémoire adressable du périphérique (comme un registre), appelé *endpoint*. Un périphérique peut disposer de plusieurs endpoints, et chacun d'entre eux est associé à un pipe. Différents types de transfert de données peuvent être utilisés sur ces pipes (en fonction de leur nature) :

- *Control transfer* : employé pour l'envoi de commandes au périphérique (demande d'informations, configuration...) et n'est possible que sur un *control pipe*. Un périphérique USB dispose au minimum d'un control pipe, qui est actif au plus tôt de son initialisation.

- *Interrupt transfer* : utilisé pour l'envoi de petites quantités de données avec une latence minimale garantie. Un tel transfert n'est possible que sur un *data pipe*.

- *Bulk transfer* : consacré au transfert de larges blocs de données profitant de toute la bande passante de l'USB, mais sans garantie de vitesse et de latence. Toutefois, ce type de transfert garantit la transmission de données sans perte. Transfert possible uniquement sur un data pipe.

- *Isochronous transfer* : ce dernier type de transfert garantit la latence et la bande passante, au détriment de tout mécanisme de correction d'erreurs disponible pour les autres types de transfert. Un tel transfert n'est possible que sur un data pipe.

À noter que seul le *control pipe* est bidirectionnel (envois de requêtes de l'hôte et réponses du périphérique). Les *data pipes* sont unidirectionnels. Ainsi, pour un échange de données bidirectionnel, deux *pipes* doivent être configurés, mettant à contribution deux *endpoints* différents du périphérique. Un corollaire à cela, et que chaque *endpoint* d'un périphérique est exclusivement lisible ou inscriptible.

Chaque *endpoint* est accessible par l'hôte via l'émission de requêtes contenant l'adresse du périphérique (attribué par l'hôte) et le numéro du *endpoint* (attribué par le périphérique). L'accès se fera en lecture ou en écriture suivant le pipe, mais toujours à l'initiative de l'hôte. En effet, un périphérique USB est un périphérique esclave (sauf dans le cas des périphériques USB On-The-Go, qui peuvent jouer le rôle de contrôleur également). Ainsi, si l'hôte n'a pas émis de requête au périphérique, toute tentative d'émission de sa part sera ignorée.

1.2 Initialisation d'un périphérique USB

Lors du branchement d'un périphérique USB à l'hôte via le *root hub* par exemple, ce dernier notifie le contrôleur de la présence d'un périphérique (cette information est fournie en réponse aux requêtes de type `GET_PORT_STATUS` de l'hôte). Le contrôleur va alors procéder (sous la directive du driver USB) à une étape d'initialisation, appelée énumération du bus, afin d'adresser et d'identifier le périphérique. Tout d'abord l'hôte émet une demande de réinitialisation (requête `SET_PORT_FEATURE`) du périphérique au hub auquel il est connecté. Le périphérique est censé se placer dans son état par défaut (réinitialisation de ses registres...), et le hub lui fournit du courant (pas plus de 100 mA à ce stade). Le périphérique répond alors à l'adresse USB par défaut, qui est 0. L'hôte lui demande de lui envoyer son *device descriptor* (sur lequel nous revenons par la suite), afin de déterminer la taille maximale d'un paquet USB qu'il peut émettre sur son *control pipe*. Il assigne enfin une adresse unique au périphérique (avec la requête `SET_ADDRESS`). S'ensuit alors une étape de configuration du périphérique, dans laquelle l'hôte va d'abord récupérer différents descripteurs (via des commandes `SETUP`) caractérisant le périphérique, pour ensuite établir les valeurs de ses paramètres configurables.

Ces descripteurs font partie du standard USB, et permettent à un hôte de connaître les caractéristiques d'un périphérique USB en cours d'initialisation. Le *Device Descriptor* renseigne, entre autres, sur le standard USB auquel se conforme le périphérique, le *Vendor ID*, le *Product ID*, ainsi que sur la classe fonctionnelle du périphérique (*mass storage*, audio, *HID - Human Interface Device*, etc.). Ce descripteur demandé en premier par l'hôte, lui permet de charger un driver approprié (soit un driver spécifique associé au VID/PID, soit un driver générique suivant la classe du périphérique). Un périphérique dispose aussi d'au moins un *Configuration Descriptor*. Chaque descripteur de ce type fournit des informations spécifiques sur une configuration du périphérique. Différentes configurations peuvent être proposées. Ainsi l'hôte doit faire un choix sur la configuration qu'il souhaite utiliser sur le périphérique (au préalable, il les récupère via des commandes `GET_DESCRIPTOR` sur le *control pipe*). Grossièrement, une configuration donne les caractéristiques des *endpoints* du périphérique, ainsi que ses besoins en termes de courant électrique. Plus précisément, une configuration se compose d'un ensemble de *Endpoints Descriptors*, eux-même regroupés au sein d'une ou plusieurs interfaces, lesquelles sont aussi décrites au travers de descripteurs (*Interface Descriptor*). Une interface capture le concept de fonction USB, et une même configuration peut disposer de multiples interfaces utilisables simultanément (une webcam USB munie d'un micro par exemple). À noter que les *endpoints* sont associés de manière unique aux interfaces d'une même configuration. Chaque interface décrit d'ailleurs sa classe fonctionnelle USB au sein de son descripteur, ce qui aide un hôte à charger un driver pour cette fonction spécifique (ce cas se présente si la classe fonctionnelle n'est pas fournie dans le *device descriptor*). D'autres descripteurs fournissent des informations complémentaires, telles que des descriptions textuelles sur les fonctions du périphérique ou encore le comportement du périphérique suivant la vitesse configurée (*full-speed/high-speed*). Enfin, certains sont spécifiques aux différentes classes de périphérique USB.

1.3 Le protocole de communication

Finissons cette introduction en abordant, quelque peu, le protocole de communication USB. Le bus USB est découpé temporellement en *frames* de même durée : 1 ms (les périphériques *high-speed* redécoupent ces *frames* en 8 *microframes* de 125 µs). Chaque frame consiste en un paquet de type SOF (*Start Of Frame*), suivi d'une ou de plusieurs transactions. Chaque transaction est composée d'une série de paquets. Un paquet est précédé d'un motif de synchronisation (*Packet Sync*), et se termine par un motif de fin de paquet (EOP - *End Of Packet*). Une transaction est, quant à elle, composée au minimum, d'un paquet de type *Token* qui est toujours le premier dans la transaction. Elle peut également contenir un ou plusieurs paquets de type *Data*, et se terminer éventuellement par un paquet de type *Handshake*.

Les paquets Token proviennent toujours de l'hôte. Il en existe de quatre sortes :

- Token IN : Ces paquets sont utilisés pour demander à un périphérique d'émettre des données à l'hôte.

- Token OUT : Ces paquets précèdent toujours une émission de données de l'hôte vers le périphérique.

- Token SETUP : Ces paquets précèdent les commandes de l'hôte qui sont transmises sur le control pipe par défaut du périphérique (telles que : `GET_DESCRIPTOR`, `SET_ADDRESS`, etc.)

- Token SOF : Ces paquets servent à marquer le début d'une *frame*, et n'ont qu'un objectif de synchronisation.

Ces paquets sont composés d'un Paquet ID (IN, OUT, SETUP, SOF - sur 8 bits), de l'adresse du périphérique destinataire (sur 7 bits), du numéro de endpoint visé (sur 4 bits, avec au maximum 16 endpoints en lecture, et 16 en écriture), et se termine par un CRC sur 5 bits.

Les paquets de type Data servent aux transferts de données, ou à la transmission de commandes. On les retrouve ainsi à la suite des tokens IN, OUT et SETUP. Ils se composent d'un Packet ID (DATA0 et DATA1 qui sont utilisés en alternance lors d'un transfert), d'une payload de 0 à 1024 octets dépendant du type de transfert, et d'un CRC sur 16 bits.

Enfin, les paquets de type Handshake concluent chaque transaction. Ils permettent notamment d'acquitter (Handshake de type ACK) ou non (handshake de type NAK ou STALL),

la terminaison correcte d'une transaction. Ces paquets seront donc envoyés par le périphérique dans le cas de transferts de type OUT et SETUP, et par l'hôte pour les transactions de type IN.

On note ainsi trois types de transactions :

- Les transactions de lecture : Envoi d'un Token IN par l'hôte, puis envoi de Data packets par le périphérique, et enfin handshake par l'hôte.
- Les transactions d'écriture : Envoi d'un Token OUT par l'hôte, puis envoi de Data packets par l'hôte, et enfin handshake par le périphérique.
- Les transactions de contrôle : Elles permettent d'identifier, de configurer, et de contrôler les périphériques USB. Elles sont découpées en trois étapes (*stages*) :

1) *Setup Stage* : Cette étape est constituée par l'émission par l'hôte d'un Token SETUP, suivie d'un Data packet de 8 octets contenant la commande, appelée *USB request* (telle que `GET_DESCRIPTOR`). À noter que des arguments sont également fournis dans la requête et permettent de la préciser. On y trouve notamment la quantité de données à transmettre dans la deuxième étape, et éventuellement la cible de la commande. Dans notre cas, ses arguments contiennent notamment le type de descripteur que l'hôte souhaite récupérer du périphérique), ainsi que la taille du descripteur. Elle se termine par l'émission obligatoire d'un Hanshake de type ACK, par le périphérique.

2) *Data Stage* : cette étape est optionnelle, et dépend de la requête USB. Elle permet à l'hôte ou au périphérique de transférer des données. Par exemple, à la suite d'une requête `GET_DESCRIPTOR`, le périphérique transmet durant cette étape le descripteur USB demandé. Elle se compose ainsi d'un Token IN ou OUT (dans notre exemple, il s'agit d'un IN, pour récupérer le descripteur), de paquets Data, et d'un paquet Handshake. Les transactions IN sont qualifiées de *transactions de contrôle en lecture*, alors que les OUT, de *transactions de contrôle en écriture*.

3) *Status Stage* : Cette étape sert à renseigner sur le succès ou l'échec des étapes précédentes (ou encore que la commande est toujours en cours de traitement). Il s'agit toujours d'une notification de la part du périphérique à l'hôte. Un peu alambiqué, elle débute par un IN ou un OUT, suivant que la transaction de contrôle était de type écriture ou lecture. Elle se termine avec un paquet Data de taille nulle (c'est-à-dire contenant juste le Packet ID associé au Data et un CRC 16 bits) et/ou un paquet Handshake (émis par le receveur du précédent paquet) fournissant le statut à l'hôte. Dans le cas d'une transaction de contrôle en écriture, un token IN est transmis, et le statut est fourni via l'émission par le périphérique soit d'un paquet Data de taille nulle (signifiant que tout est bon), soit via l'émission directement d'un Handshake (STALL : pour signifier une erreur, NAK : pour notifier que l'action est toujours en cours). Dans le cas d'une transaction de contrôle en lecture, un token OUT est émis, suivi par un paquet Data de taille nulle, et le périphérique notifie son statut par un Handshake (ACK : pour signifier que tout est bon, STALL : pour l'erreur, et NAK : pour le non terminé). Ainsi, dans notre exemple de requête `GET_DESCRIPTOR` (transaction de contrôle en lecture), le périphérique va notifier à l'hôte s'il est prêt pour recevoir une nouvelle commande, ou s'il y a eu des erreurs lors de son exécution. Durant cette étape, l'hôte envoie donc un Token OUT, suivi d'un paquet Data de taille nulle, auquel le périphérique répond par un Hanshake qui permet de notifier à l'hôte que le périphérique a envoyé toutes les données, ou bien qu'une erreur a eu lieu.

2 Aperçu de l'infrastructure Gadget USB

L'infrastructure Gadget USB permet de faciliter la création de périphériques USB virtuels. Il faut entendre par virtuel, le fait que les fonctionnalités USB du périphérique sont entièrement logicielles. Cette infrastructure a besoin toutefois de se reposer sur un contrôleur matériel de périphériques USB, que l'on retrouve de facto dans l'ensemble des contrôleurs USB OTG (contrôleur ayant la capacité de jouer alternativement le rôle d'hôte USB ou de périphérique) de nos téléphones portables ou bien sur des plateformes ARM bien connues telles que la Raspberry Pi [*NDLR : le modèle A uniquement, le B se voyant équipé d'un hub soudé, il n'est pas raisonnablement possible de l'utiliser en USB périphérique*], la PandaBoard, etc. Ce contrôleur matériel fournit des ressources de communication, les end-points (registres mémoire du contrôleur), et implémente la partie bas niveau du protocole USB (gestion électrique, *bit stuffing*, CRC...). Dans le reste de cet article, nous vous proposons de découvrir les différentes couches logicielles établies au-dessus de ces contrôleurs et qui permettent la création de périphériques USB aux fonctionnalités limitées uniquement par votre imagination.

Avant d'entreprendre ce voyage, il est bon de fixer quelques notions et termes, qui faciliteront la lecture de l'article :

- Pilote de contrôleur USB esclave, que nous appelons dans le reste de l'article : **PDC (Peripheral Device Controller) driver**. Son principal objectif est de configurer le PDC (allocation mémoire nécessaire à son fonctionnement, enregistrement d'un gestionnaire d'interruption...).
- **Gadget Driver** : Il s'agit de la couche fondamentale de l'infrastructure Gadget. Elle fournit l'ensemble des abstractions nécessaires qui permettent de créer des périphériques USB virtuels, sans se préoccuper des spécificités matérielles.
- **Class Driver** ou **Function Driver** : Il s'agit des implémentations des fonctions USB proprement dites.
- **Composite Framework / Couche Composite** : Il s'agit d'une couche d'abstraction permettant de faire cohabiter plusieurs Function Drivers, afin de présenter à un hôte USB un périphérique composite remplissant plusieurs fonctions.

Un driver de PDC tel que `musb-hrdc` (`drivers/usb/musb/musb_core.c` ; utilisé pour piloter les PDC USB *Inventra* de *Mentor Graphics* que l'on retrouve dans la Pandaboard notamment) implémente l'interface `usb_gadget` afin d'abstraire le PDC pour faciliter l'implémentation de périphériques USB logiciels, et implémente l'interface `platform_driver` pour la liaison avec l'infrastructure des drivers Linux.

L'interface `usb_gadget` (`include/linux/usb/gadget.h`) représente un périphérique USB, que ce soit au travers des opérations qu'il peut effectuer (implémentées de façon agnostique par rapport au matériel dans la `struct usb_gadget_ops` incluse dans `usb_gadget`), ou des ressources matérielles dont il dispose (les *endpoints*, manipulables au travers de `struct usb_ep`). Ces structures et les primitives associées définissent une API qui abstrait le matériel sous-jacent. Un driver PDC implémente les différentes méthodes qui composent `usb_gadget_ops` et fournit ainsi une interface agnostique en ce qui concerne le fonctionnement interne du contrôleur matériel.

Du côté du `platform_driver`, la méthode `probe()` est appelée par le noyau Linux pour détecter un matériel donné et le configurer le cas échéant. Dans le cas de notre PDC, ces opérations sont effectuées au travers de `musb_probe()` (`drivers/usb/musb/musb_core.c`). Après avoir détecté le PDC, la fonction le configure (via `musb_init_controller()`), étape durant laquelle la `struct usb_gadget` est créée via `musb_gadget_setup()`. Et enfin, le driver de PDC et son interface gadget est ajoutée à la liste des drivers noyau de type UDC via `usb_add_gadget_udc()` (`drivers/usb/gadget/udc/udc-core.c`), qui est appelée à la fin de `musb_gadget_setup()`.

Pour créer un périphérique USB logiciel, il est alors nécessaire de « s'accrocher » à ce PDC. Pour cela, il faut d'abord définir un `gadget driver`, représenté par une structure `usb_gadget_driver` (`include/linux/usb/gadget.h`). Les différentes méthodes de cette interface sont appelées directement par le PDC aux moments opportuns dictés par le contrôleur USB hôte distant auquel se connecte le PDC. Ainsi, lors de la connexion du périphérique USB virtuel (s'exécutant sur une plateforme pourvue d'un PDC) à un hôte réel, la méthode `bind()` est appelée de façon à configurer le périphérique virtuel. Notons également la méthode `setup()` qui implémente le traitement des requêtes USB de type `SETUP` (récupération des différents descripteurs USB...).

`include/linux/usb/gadget.h` :

```
struct usb_gadget_driver {
  char    *function;
  enum usb_device_speed max_speed;
  int     (*bind)(struct usb_gadget *gadget,
          struct usb_gadget_driver *driver);
  void     (*unbind)(struct usb_gadget *);
  int     (*setup)(struct usb_gadget *,
          const struct usb_ctrlrequest *);
  void     (*disconnect)(struct usb_gadget *);
  void     (*suspend)(struct usb_gadget *);
  void     (*resume)(struct usb_gadget *);

  /* FIXME support safe rmmod */
  struct device_driverdriver;
};
```

Cette structure est notamment implémentée par l'infrastructure USB *Composite* (`drivers/usb/gadget/composite.c`) que nous détaillons par la suite, laquelle permet de créer des périphériques USB multifonctions. Elle est également implémentée par *GadgetFS* qui permet de faire la jonction entre des périphériques USB définis en espace utilisateur et l'infrastructure gadget dans le noyau.

Enfin, la fonction `usb_gadget_probe_driver` doit être appelée pour lier ce gadget driver à un driver de PDC et ainsi faire en sorte que le périphérique USB logiciel puisse réagir aux requêtes d'un contrôleur USB hôte. Son rôle est de trouver un PDC enregistré sur la plateforme, et d'y associer le `usb_gadget_driver` qui lui est passé en paramètre (via `udc_bind_to_driver(struct usb_udc *udc, struct usb_gadget_driver *driver)`).

Dans l'exemple de l'infrastructure USB Composite, cette fonction est appelée au sein de sa primitive `usb_composite_probe()` (`drivers/usb/gadget/composite.c`). Et cette dernière est appelée dans la procédure d'initialisation de tout module noyau implémentant un périphérique USB composite logiciel.

3 Pilote de contrôleur USB esclave (PDC USB)

3.1 Interaction avec le contrôleur matériel

Un PDC USB n'a pas de particularité liée à l'USB, car les spécificités USB sont gérées au niveau du matériel et dans les couches USB au-dessus du PDC. La fonction majeure du PDC driver est d'allouer, lorsque le PDC est détecté par la

plate-forme les ressources nécessaires au bon fonctionnement du périphérique (mémoire, lignes d'interruptions, etc.), et de libérer ces ressources lorsque ce périphérique est déconnecté. Pour l'implémentation d'un tel driver, la *platform* API est généralement utilisée. Elle se compose de deux structures principales : la **struct platform_device** et la **struct platform_driver**.

3.1.1 La struct platform_device et les Device Tree

La **struct platform_device** représente le PDC et le lie à l'arborescence des périphériques (via son champ **dev**) :

```
struct platform_device {
        const char      *name;
        int             id;
        bool            id_auto;
        struct device   dev;
        u32             num_resources;
        struct resource *resource;
[...]
};
```

Cette structure est utilisée afin de décrire des périphériques qui ne peuvent pas être découverts automatiquement par le noyau, comme cela est fait pour un périphérique PCI par exemple. Cette situation est typique des périphériques inclus dans les SoC que l'on retrouve dans de nombreux systèmes embarqués. Une façon de faire (plutôt obsolète aujourd'hui), est de décrire l'ensemble des périphériques « non découvrables » d'un SoC donné, via des **struct platform_device** (définis statiquement), auxquels on associe des **struct resource** (via le champ **resource**) décrivant précisément les ressources au sein du SoC (mémoire, ligne d'interruption...) qui appartiennent aux périphériques.

Par exemple, le PDC USB **tusb6010** d'un SoC OMAP particulier est décrit de la sorte (**arch/arm/mach-omap2/usb-tusb6010.c**) :

```
static struct platform_device tusb_device = {
        .name           = "musb-tusb",
        .id             = -1,
        .dev = {
                .dma_mask               = &tusb_dmamask,
                .coherent_dma_mask      = 0xffffffff,
        },
        .num_resources  = ARRAY_SIZE(tusb_resources),
        .resource       = tusb_resources,
};
```

Et l'ensemble des ressources physiques qui appartiennent au périphérique est inscrit dans le tableau **tusb_resource** :

```
static struct resource tusb_resources[] = {
        { /* Asynchronous access */
                .flags  = IORESOURCE_MEM,
        },
        { /* Synchronous access */
                .flags  = IORESOURCE_MEM,
        },
        { /* IRQ */
                .name   = "mc",
                .flags  = IORESOURCE_IRQ,
        },
};
```

Ce tableau de ressources est complété par la procédure d'initialisation **tusb6010_setup_interface()** qui est exécutée par Linux lors du démarrage de la plateforme OMAP. Ce tableau est complété avec des informations sur l'emplacement des zones mémoire, I/O, etc. au sein du SoC OMAP. Une fois complété, ce tableau pourrait ressembler à :

```
static struct resource tusb_resources[] = {
  {/* Asynchronous access */
   .start = 0x10000000,
   .end = 0x100009ff,
   .flags = IORESOURCE_MEM,
  },
[...]
  {/* IRQ */
   .start = 20,
   .end = 20,
   .flags = IORESOURCE_IRQ,
   .name= "mc",
  }
 };
```

Il sera alors possible pour un **platform_driver** (que nous décrivons par la suite) d'interagir avec ces périphériques et d'accéder à leurs ressources via des appels aux fonctions **platform_get_resource*()**, **platform_get_irq*()** de la *platform* API.

Toutefois, cette façon de faire est obsolète, et les *device tree* sont aujourd'hui le moyen classique pour décrire de tels périphériques « non découvrables » (à noter que les *device tree* ne se limitent pas à décrire seulement ce type de périphériques). Un *device tree* est un fichier de description textuelle d'une configuration matérielle spécifique d'un système (http://devicetree.org/Device_Tree_Usage), et se veut indépendant de l'OS. Ainsi pour la Pandaboard par exemple, un tel fichier définit l'ensemble de ses périphériques et des ressources associées (**arch/arm/boot/dts/omap4-panda-common.dtsi**, **arch/arm/boot/dts/omap443x.dtsi**). Linux au démarrage du SoC, va extraire le *device wtree* d'un blob et créer l'ensemble des abstractions logicielles (des **struct device_node** via **unflatten_device_tree()** dans **drivers/of/fdt.c**) qui permettra aux drivers d'y accéder. De plus, pour les périphériques « non découvrables » automatiquement, l'infrastructure de gestion des *device tree* de Linux s'occupe également de créer des **struct platform_device** (**drivers/of/platform.c**), de façon à rendre transparente (en grande partie) leur utilisation par les **platform_driver**.

3.1.2 La struct platform_driver

La **struct platform_driver** fournit un canevas générique à l'implémentation du driver. Un tel driver est lié à l'arborescence de tous les drivers via son champ **driver**.

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
        bool prevent_deferred_probe;
};
```

Dans le cas particulier du driver de PDC USB Inventra **musb-hdrc** (**drivers/usb/musb/musb_core.c**) que l'on retrouve sur les Pandaboard, cette structure est instanciée de la façon suivante (**drivers/usb/musb/musb_core.c**) :

```
static struct platform_driver musb_driver = {
        .driver = {
static struct platform_driver musb_driver = {
        .driver = {
                .name           = (char *)musb_driver_name,
                .bus            = &platform_bus_type,
                .owner          = THIS_MODULE,
                .pm             = MUSB_DEV_PM_OPS,
        },
        .probe          = musb_probe,
        .remove         = musb_remove,
        .shutdown       = musb_shutdown,
};
```

Une fois enregistrée via **platform_driver_register()** (après *preprocessing* de la macro **module_platform_driver()**), si un **platform_device** correspondant existe, la fonction **probe()** est appelée. Le code de **musb_probe()** récupère initialement les informations sur les ressources du PDC USB :

```
static int musb_probe(struct platform_device *pdev)
{
        struct device   *dev = &pdev->dev;
        int             irq = platform_get_irq_byname(pdev, "mc");
        struct resource *iomem;
        void __iomem    *base;

        iomem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if (!iomem || irq <= 0)
                return -ENODEV;

        base = devm_ioremap_resource(dev, iomem);
        if (IS_ERR(base))
                return musb_init_controller(dev, irq, base);
}
```

3.1.3 Digression sur la struct usb_phy

Un driver de PDC USB de type OTG (comme le PDC **musb-hdrc** définit dans **drivers/usb/musb/musb_core.c**), manipule également une **struct usb_phy** (**include/linux/usb/phy.h**). Cette structure sert à l'implémentation des drivers de contrôleur USB hôte. Dans le cas OMAP, elle sert à dialoguer avec la face « cachée » du PDC : le contrôleur hôte. Le passage d'un mode à l'autre se fait à l'initiative du contrôleur OTG qui est en mode hôte. Dans le cas où une telle opportunité est fournie à l'autre contrôleur OTG, alors en mode périphérique ; il peut, à sa discrétion, activer son mode hôte. D'où le besoin d'une interaction entre les deux parties du contrôleur OTG. Sur notre exemple de plateforme OMAP, on a donc un autre driver qui gère la partie contrôleur hôte du contrôleur OTG (**drivers/usb/phy/phy-omap-otg.c**), et qui s'appuie sur l'API **usb_phy**.

Un bref détour par la **struct usb_phy**, nous montre qu'elle se rattache à l'arborescence des périphériques via son champ **dev**. Elle définit une interface pour les méthodes nécessaires à la gestion du bus (**set_vbus**, **set_power**), ainsi que des méthodes de notification lorsque des périphériques se connectent (**notify_connect**, **notify_disconnect**)

```
struct usb_phy {
        struct device           *dev;
[...]
        struct usb_otg          *otg;
[...]
        int     (*set_vbus)(struct usb_phy *x, int on);
        int     (*set_power)(struct usb_phy *x, unsigned mA);
        int     (*set_suspend)(struct usb_phy *x, int suspend);
        int     (*set_wakeup)(struct usb_phy *x, bool enabled);

        int     (*notify_connect)(struct usb_phy *x, enum usb_device_
speed speed);
        int     (*notify_disconnect)(struct usb_phy *x, enum usb_device_
speed speed);
};
```

3.2 Interaction avec la Gadget API

Le PDC driver agit comme une abstraction du matériel sous-jacent. Il se conforme à la Gadget API :

- en enregistrant (via **usb_add_gadget_udc()**) une **struct usb_gadget** (regroupant les ressources comme les *endpoints* USB) et une **struct usb_gadget_ops** associée (définissant les opérations n'impliquant pas les *endpoints*) ;

- en définissant des **struct usb_ep** pour chacun des *endpoints* du PDC et la **struct usb_ep_ops** associée. À noter que ces *endpoints* sont ensuite utilisables après activation via **usb_ep_enable()** qui a pour effet de les rendre visibles dans l'infrastructure Gadget USB.

Ces différentes structures (définies dans **include/linux/usb/gadget.h**) augmentées de la **struct usb_request** (permettant de représenter les requêtes USB émises par le contrôleur hôte) forment les fondations de la Gadget API. Elles sont manipulées par les Gadget driver qui s'appuient sur des primitives dédiées. Nous décrivons dans la suite ces différentes structures.

3.2.1 Les structures usb_gadget et usb_gadget_ops

La structure **usb_gadget** représente le périphérique USB au sein de l'infrastructure USB Gadget. Elle conserve des informations sur les capacités et ressources du PDC. Elle contient aussi des méthodes abstraites pour effectuer des opérations typiques de l'USB, mais dont l'implémentation est propre au PDC. À noter que la majorité de ses champs sont en lecture seule pour un gadget driver.

```
struct usb_gadget {
  struct work_struct  work;
  const struct usb_gadget_ops  *ops;
  struct usb_ep     *ep0;
  struct list_head  ep_list; /* of usb_ep */
  enum usb_device_speed  speed;
  enum usb_device_speed  max_speed;
  enum usb_device_state  state;
struct usb_gadget {
  struct work_struct  work;
  const struct usb_gadget_ops  *ops;
  struct usb_ep     *ep0;
  struct list_head  ep_list; /* of usb_ep */
  enum usb_device_speed  speed;
  enum usb_device_speed  max_speed;
  enum usb_device_state  state;
```

Une liste chaînée de tous les *endpoints* du PDC est référencée par le champ **ep_list**. Le champ **ep0** pointe sur l'*endpoint* éponyme, il est utilisé pour traiter les requêtes USB de type **SETUP** (*cf.* section 1.3). À noter que ce *endpoint* n'est pas présent dans la liste précédente. Le champ **dev** fait le lien avec l'infrastructure des périphériques sous Linux. Le champ **state** est employé pour conserver l'état USB dans lequel se trouve le PDC : *attached, powered, configured, suspended*, ...(tel que défini par le chapitre 9 de la spécification USB 2.0). Les cinq drapeaux **is_otg**, **is_a_peripheral**, **b_hnp_enable**, **a_hnp_support**, **a_alt_hnp_support**, sont utilisés lorsque le PDC est compatible OTG. Ces drapeaux permettent de distinguer si le PDC agit en tant que périphérique (**is_a_peripheral** égal à 1) ou en tant qu'hôte. Ils renseignent également sur le support du protocole HNP (*Host Negociation Protocol*) par l'hôte auquel est connecté le PDC. Ce protocole est utilisé, entre périphériques OTG, pour passer de l'état périphérique à l'état hôte.

Enfin, le dernier champ **ops** pointe vers une structure **usb_gadget_ops** qui définit des méthodes abstraites qui peuvent être appelées sur le PDC.

```
struct usb_gadget_ops {
        int     (*get_frame)(struct usb_gadget *);
        int     (*wakeup)(struct usb_gadget *);
        int     (*set_selfpowered) (struct usb_gadget *, int is_selfpowered);
        int     (*vbus_session) (struct usb_gadget *, int is_active);
       int     (*vbus_draw) (struct usb_gadget *, unsigned mA);
        int     (*pullup) (struct usb_gadget *, int is_on);
        int     (*ioctl)(struct usb_gadget *, unsigned code, unsigned long param);
        void    (*get_config_params)(struct usb_dcd_config_params *);
        int     (*udc_start)(struct usb_gadget *, struct usb_gadget_driver *);
        int     (*udc_stop)(struct usb_gadget *, struct usb_gadget_driver *);
};
```

Une description de ces fonctions est donnée ci-dessous :

- **get_frame** : renvoie le numéro de frame USB (cf. section 1.3).
- **wakeup** : permet l'envoi de notifications, lors du réveil du PDC (ayant été préalablement mis en veille), à l'hôte auquel il est attaché. Le support de cette capacité doit être notifié à l'hôte au travers d'un descripteur USB de type **CONFIGURATION**, transmis lors de l'étape d'énumération.
- **set_selfpowered** : permet de configurer le PDC pour qu'il utilise une source de courant indépendante du bus USB auquel il est connecté (cette dernière dépendant de l'hôte).
- **vbus_session** : cette fonction est appelée par le noyau Linux, lorsqu'il détecte du courant sur la ligne VBUS du bus USB. Elle permet de notifier le PDC de cela.
- **vbus_draw** : cette fonction permet de renseigner le PDC sur la quantité de courant que l'hôte lui autorise d'utiliser.
- **pullup** : active la résistance de rappel sur la ligne D+ ou D- du bus USB (ces lignes servent à la transmission des données USB) permettant à l'hôte de déclencher l'énumération. Il est à noter qu'un périphérique notifie sa vitesse à l'hôte suivant cette résistance de rappel. Si cette dernière est connectée à la ligne D+, le périphérique démarre en full-speed (12 Mb/s), sinon il démarre en low-speed (1,5Mb/s). Le mode high-speed (480 Mb/s) est similaire au full-speed, sauf que le PDC doit communiquer cette information à l'hôte en agissant sur les niveaux des lignes D+ et D- selon un schéma spécifique.
- **ioctl** : permet l'implémentation de commandes additionnelles sur le PDC.
- **udc_start** : cette méthode est appelée par le noyau, lorsqu'il est sur le point d'enregistrer un gadget driver avec ce PDC. Cette méthode est appelée juste avant la méthode d'initialisation **bind()** du gadget driver.

- **udc_stop** : cette méthode est appelée lors du déchargement du gadget driver associé au PDC. C'est-à-dire juste après l'appel à sa méthode **unbind()**.

Ces méthodes doivent être implémentées, bien entendu, par le driver de PDC. Ainsi, dans notre exemple de PDC **musb-hdrc**, cette structure est instanciée dans **musb_gadget_operations** (**drivers/usb/musb/musb_gadget.c**). Si nous revenons sur l'initialisation de ce driver, voici les étapes principales aboutissant à l'enregistrement de cette structure dans le noyau :

1) **musb_probe()** : détecte, puis configure le PDC via la fonction **musb_init_controller()**.

2) **musb_init_controller()** : configure le PDC et alloue une **struct musb** qui héberge toutes les informations nécessaires au driver du PDC **musb-hdrc** et notamment la **struct usb_gadget**. La fonction **musb_gadget_setup()** est alors appelée avec cette structure en paramètre dans le cas où le PDC est en mode périphérique (sinon c'est la fonction **musb_host_setup()** qui est appelée).

3) **musb_gadget_setup()** : cette primitive complète la **struct usb_gadget** en lui associant notamment la **usb_gadget_ops**. Enfin, elle enregistre le PDC au sein de l'infrastructure gadget.

Ainsi, lors de l'enregistrement d'un gadget driver dans le noyau, **musb-hdrc** lui sera associé (cf. section 3.3).

3.2.2 Les structures usb_ep et usb_ep_ops

La structure **usb_ep** représente un *endpoint* USB. Le driver du PDC définit une telle structure (dans le cas **musb-hdrc**, elle est encapsulée dans une structure **musb_ep**) pour chacun de ses *endpoints*. Une liste chaînée les regroupe (à l'exception du *endpoint* 0) via leur champ **ep_list**. Cette liste est celle qui est accessible depuis la structure **usb_gadget**.

```
struct usb_ep {
        void                    *driver_data;

        const char              *name;
        const struct usb_ep_ops *ops;
        struct list_head        ep_list;
        unsigned                maxpacket:16;
        unsigned                maxpacket_limit:16;
        unsigned                max_streams:16;
        unsigned                mult:2;
        unsigned                maxburst:5;
        u8                      address;
        const struct usb_endpoint_descriptor    *desc;
        const struct usb_ss_ep_comp_descriptor  *comp_desc;
};
```

Dans cette structure, le champ **maxpacket** renseigne sur la taille maximale utilisée sur ce *endpoint*. Elle correspond aux informations transmises à l'hôte en fin d'étape d'énumération (lorsque le descripteur USB de type ENDPOINT est transmis). Enfin, le descripteur USB (cf. section 1) relatif au *endpoint* est associé à la structure **usb_ep** via les champs **desc**, et champs **comp_desc** (pour les descripteurs compatibles USB 3.0 en mode SuperSpeed). Ces descripteurs sont en fait configurés lors de l'initialisation d'un gadget driver, en fonction de ses besoins : type et sens des *endpoints* nécessaires à son fonctionnement, taille de paquet, etc. Remarquons enfin le champ **ops**, qui pointe vers une structure **usb_ep_ops** définissant les méthodes abstraites de manipulations des endpoints. Ces méthodes doivent être implémentées par le driver du PDC. (dans le cas du PDC **musb-hdrc**, elles sont instanciées dans **musb_ep_ops** définit dans **drivers/usb/musb/musb_gadget.c**).

```
struct usb_ep_ops {
        int (*enable) (struct usb_ep *ep,
                const struct usb_endpoint_descriptor *desc);
        int (*disable) (struct usb_ep *ep);
        struct usb_request *(*alloc_request) (struct usb_ep *ep,
gfp_t gfp_flags);
        void (*free_request) (struct usb_ep *ep, struct usb_request *req);

        int (*queue) (struct usb_ep *ep, struct usb_request *req,
gfp_t gfp_flags);
        int (*dequeue) (struct usb_ep *ep, struct usb_request *req);

        int (*set_halt) (struct usb_ep *ep, int value);
        int (*set_wedge) (struct usb_ep *ep);

        int (*fifo_status) (struct usb_ep *ep);
        void (*fifo_flush) (struct usb_ep *ep);
};
```

Une description de ces fonctions est donnée ci-dessous :

- **enable** : active et configure un endpoint via la fourniture d'un descripteur USB de type ENDPOINT. Le driver de PDC active aussi les interruptions pour ce *endpoint*. Cette méthode est accessible pour un gadget driver au travers de la fonction **usb_ep_enable()** de l'API Gadget (**include/linux/usb/gadget.h**).
- **disable** : désactive un *endpoint*, et libère toutes les transactions USB associées (matérialisées par des **usb_request**) qui n'ont pas encore été traitées. La méthode est accessible au travers de la **usb_ep_disable()** de l'API Gadget.
- **alloc_request** : alloue une **struct usb_request** associée à un endpoint donné. Ce type de structure sur lequel nous revenons en 3.2.3, permet de décrire des requêtes d'E/S auprès du PDC, pour transmettre ou rece-

voir des données via un *endpoint*. La méthode est appelée par la fonction `usb_ep_alloc_request()` de l'API Gadget.

- `free_request` : libère la mémoire associée à une requête USB. Appelée par la fonction `usb_ep_free_request()` de l'API Gadget.
- `queue` : permet à un gadget driver d'enregistrer une transaction USB (décrite par une `struct usb_request`) sur un *endpoint* donné du PDC. Ces transactions sont enregistrées dans une file propre aux endpoints. Une fois enregistré, le gadget driver est notifié de la terminaison grâce à la fonction *callback* associée à la requête. Cette méthode est appelée par la fonction `usb_ep_queue()` de l'API Gadget.
- `dequeue` : permet d'enlever une requête de la file du endpoint. Appelée par la fonction `usb_ep_free_request()` de l'API Gadget.
- `set_halt` : utilisée pour signifier à l'hôte USB, une erreur de transaction sur un endpoint donné.
- `set_wedge` : similaire à `set_halt`.
- `fifo_status` : permet de récupérer l'état de la FIFO matérielle du PDC (dans la cas où une FIFO existe) liée à un *endpoint*. Lors de la terminaison du traitement d'une `usb_request` en écriture ou lecture (OUT ou IN), des données peuvent encore se trouver dans une FIFO matérielle, notamment lors d'erreurs USB. Cette méthode peut être employée pour connaître l'état de cette FIFO (le nombre d'octets laissés de côté), et prendre les dispositions nécessaires pour revenir à une situation nominale. Appelée par la fonction `usb_ep_fifo_status()` de l'API Gadget.
- `fifo_flush` : permet de vider la FIFO matérielle associée à un *endpoint*. Appelée par la fonction `usb_ep_fifo_flush()` de l'API Gadget.

Notons enfin, les fonctions suivantes de la Gadget API qui facilite la configuration de *endpoints* pour un gadget driver (définie dans `drivers/usb/gadget/epautoconf.c`) :

- `usb_ep_autoconfig(struct usb_gadget *, struct usb_endpoint_descriptor *)` : Cette fonction permet de rechercher un *endpoint* correspondant aux besoins du gadget driver (spécifié au travers du descripteur fourni en paramètre), et de le retourner le cas échéant. La fonction itère sur la liste des endpoints du PDC (`usb_gadget->ep_list`), jusqu'à trouver un candidat acceptable, sinon elle renvoie un pointeur NULL.
- `usb_ep_autoconfig_reset(struct usb_gadget *)` : Cette fonction restaure l'état initial de tous les endpoints précédemment réservés par la fonction `usb_ep_autoconfig()`.

4 Gadget driver

Un gadget driver est l'interface commune utilisée par les *function drivers* (aussi appelés *class driver*) que nous voyons dans la suite (section 5). Un tel driver définit un certain nombre de fonctions de retour qui sont appelées (par les couches basses de l'infrastructure Gadget) lors de l'occurrence de différents types d'événements (requête de l'hôte, connexion du PDC à l'hôte, etc.). Ces fonctions sont regroupées dans une structure `usb_gadget_driver`.

4.1 La structure usb_gadget_driver

```
struct usb_gadget_driver {
      char                    *function;
      enum usb_device_speed   max_speed;
      int                     (*bind)(struct usb_gadget *gadget, struct usb_gadget_driver *driver);
      void                    (*unbind)(struct usb_gadget *);
      int                     (*setup)(struct usb_gadget *, const struct usb_ctrlrequest *);
      void                    (*disconnect)(struct usb_gadget *);
      void                    (*suspend)(struct usb_gadget *);
      void                    (*resume)(struct usb_gadget *);

      struct device_driver    driver;
};
```

Un gadget driver doit implémenter les fonctions de cette structure et l'enregistrer auprès du noyau (plus précisément auprès du driver de PDC) via l'appel à `usb_gadget_probe_driver()` avec en paramètre la `struct usb_gadget_driver`. Cet appel a lieu généralement dans la procédure d'initialisation du module noyau implémentant le gadget driver et appelle la fonction `bind()` du driver. De même, la fonction `usb_gadget_unregister_driver()` est appelée dans la fonction de déchargement du module noyau, elle s'occupe du nettoyage en appelant tout d'abord la fonction `disconnect()` si le PDC est connecté à l'hôte, et enfin la fonction `unbind()`.

Lorsqu'un gadget driver est associé à un driver de PDC, ce dernier est rendu visible par l'hôte à ses clients seulement si la méthode `bind()` s'est déroulée sans erreur, et si le driver a pu traiter correctement les requêtes USB de type `SETUP` lors de l'étape d'énumération de l'hôte (via sa méthode `setup()`). À noter que ces requêtes sont généralement traitées par les *function drivers* enregistrés auprès du gadget driver (cf. section 5).

Le driver de PDC gère un certain nombre de commandes USB standard : `GET_STATUS`, `SET_FEATURE`, et `CLEAR_FEATURE`. Pour les autres, la méthode `setup()` du gadget driver est appelée. Ainsi, elle doit répondre aux commandes : `GET_DESCRIPTOR` (pour retourner les descripteurs USB standards), `SET_CONFIGURATION` (choix d'une configuration par l'hôte parmi celles transmises par le gadget driver), `SET_INTERFACE` (choix d'une configuration alternative pour l'interface donnée en paramètre de la commande), `GET_CONFIGURATION` (numéro de la configuration active), `GET_INTERFACE` (numéro de l'interface active). À noter que lors de la réception des commandes `SET_CONFIGURATION` et `SET_INTERFACE`, le gadget driver doit prendre soin d'activer les *endpoints* nécessaires, ou bien de les désactiver (lorsque la configuration 0 est choisie). De plus, les descripteurs de *endpoints* transmis lors d'un `GET_DESCRIPTOR` doivent être cohérents avec les contraintes matérielles du PDC. Suivant les PDC, les *endpoints* ne sont pas tous configurables de n'importe quelle façon. Certains ne peuvent servir que pour des transferts IN, d'autres que pour du OUT, ou encore, certains ne supportent pas le mode de transfert USB *isochrone*, etc.

Enfin, les méthodes `suspend()` et `resume()` servent à implémenter des comportements adéquats lorsque l'hôte est mis en veille et lorsqu'il se réveille. Il n'est pas obligatoire d'implémenter ces méthodes.

Remarquons la `struct usb_ctrlrequest` (définie dans `include/uapi/linux/usb/ch9.h`) passée en paramètre de la méthode `setup()` et qui décrit précisément les requêtes USB émises par l'hôte tel que définies par la norme USB.

Concluons en résumant le cycle de vie d'un gadget driver :

1) Un driver de PDC est enregistré ;

2) Un gadget driver est enregistré, provoquant l'appel de sa méthode `bind()` ;

3) Le bus USB est alimenté, l'hôte procède alors à l'étape d'énumération ;

4) Le gadget driver renvoie les descripteurs USB demandés via sa méthode `setup()`, et s'ensuit le choix de la configuration par l'hôte, et le traitement correspondant côté gadget driver ;

5) Les fonctions USB du class driver sont utilisées par le client côté hôte, provoquant des transferts de données (IN et OUT) ;

6) Le PDC est déconnecté et le gadget driver déchargé via sa méthode `unbind()`.

4.2 La structure usb_request

Un *gadget driver* ou un *functional driver* (cf. section 5) utilise des `struct usb_request` pour dialoguer avec le driver du PDC, plus précisément pour lui demander d'effectuer des transactions USB en lecture ou écriture sur certains *endpoints*.

```
struct usb_request {
      void                    *buf;
      unsigned                length;
      dma_addr_t              dma;

      struct scatterlist      *sg;
      unsigned                num_sgs;
      unsigned                num_mapped_sgs;

      unsigned                stream_id:16;
      unsigned                no_interrupt:1;
      unsigned                zero:1;
      unsigned                short_not_ok:1;
      void                    (*complete)(struct usb_ep *ep, struct usb_request *req);
      void                    *context;
      struct list_head        list;

      int                     status;
      unsigned                actual;
};
```

Cette structure contient notamment les champs suivants :

- `buff` : tampon mémoire qui contient une donnée à envoyer ou pointe vers un espace mémoire pour accueillir une donnée de taille `length`.
- `length` : contient la taille de la donnée que l'on souhaite envoyer ou recevoir.
- `complete` : ce pointeur permet d'associer une fonction de retour à la requête USB. Une fois traitée par l'infrastructure Gadget, cette fonction est appelée.
- `status` : indique l'état du traitement de la requête. Le champ est positionné à 0 si tout s'est bien passé, sinon à un nombre négatif. Dans ce dernier cas, le retour de la fonction `complete()` est attendu pour poursuivre sur le traitement des requêtes sur l'*endpoint* concerné.
- `actual` : contient le nombre d'octets qui ont été transférés du *endpoint* vers `buff`, ou de `buff` vers le *endpoint*. Cette quantité peut être inférieure à `length` dans le cas d'une lecture du *endpoint* (transfert USB de type OUT). Lors d'une écriture d'*endpoint* (transfert de type IN), il est à noter que des données peuvent toujours se trouver dans une FIFO matérielle, même si `actual` est égal à `length`.

Notons enfin que cette structure est similaire à la `struct urb` que l'on retrouve dans la partie « hôte » de la stack USB. ■

L'INFRASTRUCTURE LINUX GADGET USB : LE FRAMEWORK COMPOSITE

Éric Lacombe

Après une introduction à l'infrastructure Gadget USB, penchons-nous à présent plus spécifiquement sur la couche Composite permettant de faire cohabiter plusieurs Function Drivers et donc de présenter à un hôte USB un périphérique remplissant plusieurs fonctions...

1 Le Framework Composite : gadget USB multifonctions

Le *framework* Composite définit un modèle de gadget driver et des abstractions facilitant la construction :

- de périphériques USB virtuels (que nous appelons **gadgets USB**) multifonctions (appelés aussi « *composite* »), dont les fonctions USB sont regroupées au sein d'une même configuration (une interface par fonction) ;
- de périphériques USB virtuels à plusieurs configurations, pouvant accueillir plusieurs fonctions, sans nécessairement en avoir plus d'une par configuration.

Un exemple de tel gadget composite pourrait être un périphérique ayant une seule configuration qui supporte les fonctions « network link » et « mass storage », lesquelles peuvent être sollicitées par l'hôte *en même temps*. Bien entendu, ces fonctions pourraient également être organisées dans des configurations différentes, mais leur utilisation simultanée en deviendrait impossible ; une seule configuration peut être utilisée à la fois sur un périphérique USB.

Nous présentons le *framework* Composite au travers de l'analyse de son code source (`include/linux/usb/composite.h, drivers/usb/gadget/composite.c` et `drivers/usb/gadget/functions.c`), et de l'exemple du gadget composite `zero` (`drivers/usb/gadget/zero.c`), qui est constitué de deux fonctions USB : `SourceSink` et `Loopback`, dont nous reparlons dans la suite. (`zero` a pour vocation de servir de modèle pour la création de gadgets composites).

1.1 Multiplexage de fonctions USB

1.1.1 La structure usb_composite_driver

Tout d'abord, ce composite *framework* défini un gadget driver au sein d'une structure de plus haut niveau qui lui permet de multiplexer différentes fonctions USB. Cette structure `usb_composite_driver` (`include/linux/usb/composite.h`) est définie ci-dessous :

```
struct usb_composite_driver {
        const char                              *name;
        const struct usb_device_descriptor      *dev;
        struct usb_gadget_strings               **strings;
        enum usb_device_speed                   max_speed;
        unsigned                   needs_serial:1;

        int                        (*bind)(struct usb_composite_dev *cdev);
        int                        (*unbind)(struct usb_composite_dev *);
        void                       (*disconnect)(struct usb_composite_dev *);
        void                       (*suspend)(struct usb_composite_dev *);
        void                       (*resume)(struct usb_composite_dev *);
        struct usb_gadget_driver                gadget_driver;
};
```

Les champs `dev` et `strings` permettent de référencer les descripteurs USB correspondants qui seront utilisés par le composite driver.

On remarque bien entendu la `struct usb_gadget_driver`, ainsi qu'un certain nombre de méthodes, proches de celles que l'on retrouve dans un `usb_gagdet_driver`, sauf que le paramètre qui leur est passé est un `struct usb_composite_dev`. Nous voyons par la suite comment le *composite driver* use d'indirections pour appeler ces fonctions lors de l'occurrence de commandes USB ; ces commandes étant transmises par l'infrastructure Gadget via l'appel des fonctions adéquates du gadget driver.

Remarquons aussi que la méthode `setup()` n'est pas disponible dans cette structure. Le composite driver s'occupe de combiner les différentes fonctions USB au sein d'une configuration USB. C'est donc lors de la construction d'un composite Gadget que ces descripteurs sont créés. Le composite driver peut ensuite se charger de communiquer ces descripteurs à l'hôte USB. On peut constater cela dans sa méthode `composite_setup()` (`drivers/usb/gadget/composite.c`), qui centralise la gestion des commandes USB `SETUP` et récupère les informations adéquates de la `struct usb_composite_dev`. Rappelons toutefois que la signature de la méthode `setup()` est :

```
int setup(struct usb_gadget *gadget, const struct usb_ctrlrequest *ctrl)
```

dans laquelle la structure précédente n'apparaît pas bien entendu. Elle est en fait tout simplement référencée dans le champ `dev` de la `struct usb_gadget`, pour ensuite être récupérée via la primitive `get_gadget_data(gadget)` (`include/linux/usb/gadget.h`). Ce champ `dev` est renseigné lors de l'exécution de la fonction `composite_bind()` (`gadget_driver->bind()`) qui alloue le `usb_composite_dev`. Elle est exécutée lors de l'enregistrement du driver composite.

Dans le cas du gadget zero, cette structure est remplie de la façon suivante :

```
static __refdata struct usb_composite_driver zero_driver = {
        .name           = "zero",
        .dev            = &device_desc,
        .strings        = dev_strings,
        .max_speed      = USB_SPEED_SUPER,
        .bind           = zero_bind,
        .unbind         = zero_unbind,
        .suspend        = zero_suspend,
        .resume         = zero_resume,
};
```

On remarque l'utilisation des objets `device_desc` et `dev_strings` référençant les descripteurs USB correspondants. Ces objets sont en fait complétés par la fonction `zero_bind()` que nous voyons par la suite.

Enfin, l'enregistrement d'un composite driver se déroule de la même manière qu'un gadget driver. Les fonctions `usb_composite_probe()` et `usb_composite_unregister()` encapsulent leur équivalente de l'API Gadget driver. Dans le cas du driver zero, la fonction d'initialisation du module noyau est :

```
static int __init init(void)
{
  return usb_composite_probe(&zero_driver);
}
module_init(init);
```

Cette fonction instancie une structure `usb_gadget_driver` du nom de `composite_driver_template`, et l'enregistre via la fonction dédiée `usb_gadget_probe_driver()`.

Ce `composite_driver_template` implémente donc les méthodes du gadget driver que nous avons vu en section 4 (`bind()`, `setup()`, ...). La plupart de ces méthodes ne font que récupérer la structure `usb_composite_driver` à partir de la `struct gadget_driver` qui leur est passée en paramètre, pour ensuite appeler la méthode qui est définie à ce niveau. Dans notre exemple du composite driver `zero`, la fonction `zero_bind()` est donc appelée à l'issue de l'enregistrement. Nous la détaillons par la suite afin de découvrir son rôle.

À noter que le nettoyage se fait de la façon suivante :

```
static void __exit cleanup(void)
{
  usb_composite_unregister(&zero_driver);
}
module_exit(cleanup);
```

1.1.2 La structure usb_composite_dev

Revenons à présent sur la structure `usb_composite_dev` qui représente un composite device et renferme toutes les informations y afférant. Cette structure est manipulée notamment lors de la construction d'un gadget composite (comme nous pouvons le voir dans l'exemple du gadget zero que nous détaillons dans la suite) :

```
struct usb_composite_dev {
        struct usb_gadget                *gadget;
        struct usb_request               *req;
        struct usb_request               *os_desc_req;

        struct usb_configuration         *config;

        u8                               qw_sign[OS_STRING_QW_SIGN_LEN];
        u8                               b_vendor_code;
        struct usb_configuration         *os_desc_config;
        unsigned int                     use_os_string:1;

        unsigned int                     suspended:1;
        struct usb_device_descriptor     desc;
        struct list_head                 configs;
        struct list_head                 gstrings;
        struct usb_composite_driver      *driver;
        u8                               next_string_id;
        char                             *def_manufacturer;

        unsigned                         deactivations;
        int                              delayed_status;
        spinlock_t                       lock;
        unsigned                         setup_pending:1;
        unsigned                         os_desc_pending:1;
};
```

Cette structure contient notamment les champs suivants :

- `gadget` : pointe vers la `struct usb_gadget` définie par le PDC auquel le gadget composite est associé ;
- `req` : utilisée pour les réponses aux requêtes de contrôle USB, le tampon de la `struct usb_request` est préalloué ;
- `config` : pointe vers la configuration active (cf. section 5.1.3) ;
- `desc` : contient un descripteur USB de type `DEVICE` qui décrit le gadget composite au sens USB ;

- `configs` : pointe vers la liste chaînée de l'ensemble des configurations associées à ce gadget composite ;

- `driver` : pointe vers le `usb_composite_driver` associé (cf. section 5.1.1) ;

- `setup_pending` : est positionné à 1, lorsque des requêtes SETUP sont enregistrées dans la file, mais n'ont pas encore été traitées par l'infrastructure Gadget.

Cette structure est allouée au tout début du cycle de vie du driver gadget composite. Lors de l'enregistrement de ce dernier (via `usb_composite_probe()`), la fonction `usb_composite_driver->gadget_driver->bind()` de l'infrastructure Composite (`composite_bind()`) est appelée. Elle alloue alors une structure `usb_composite_dev`, et la référence dans le champ `dev` du `usb_gadget` du PDC driver auquel le *composite driver* est associé.

1.1.3 La struct usb_configuration

Cette structure sert de représentation pour une configuration d'un gadget composite. Il s'agit d'une couche de multiplexage qui permet de regrouper différentes fonctions USB, lesquelles sont décrites au moyen de structures de type `usb_function` et `usb_function_driver`. Nous revenons sur ces structures en section 5.3.

```
struct usb_configuration {
        const char                      *label;
        struct usb_gadget_strings       **strings;
        const struct usb_descriptor_header **descriptors;

        /* configuration management: unbind/setup */
        void                    (*unbind)(struct usb_configuration *);
        int                     (*setup)(struct usb_configuration *,
                                        const struct usb_ctrlrequest *);

        /* fields in the config descriptor */
        u8                      bConfigurationValue;
        u8                      iConfiguration;
        u8                      bmAttributes;
        u16                     MaxPower;

        struct usb_composite_dev        *cdev;

        struct list_head        list;
        struct list_head        functions;
        u8                      next_interface_id;
        unsigned                superspeed:1;
        unsigned                highspeed:1;
        unsigned                fullspeed:1;
        struct usb_function     *interface[MAX_CONFIG_INTERFACES];
};
```

Cette structure contient notamment les champs suivants :

- `strings` : référence les descripteurs USB de type `STRING` ;

- `descriptors` : référence les descripteurs précédant ceux des fonctions (comme les descripteurs OTG) ;

- `setup` : permet de déléguer le contrôle des requêtes de l'hôte de type `SETUP`, lorsqu'elles ne sont pas traitées directement par la couche *composite*, ou lorsqu'elles sont envoyées directement à une interface (fonction USB) précise (rappelons, qu'une requête `SETUP` dispose d'arguments) ;

- `bConfigurationValue`, `iConfiguration`, `bmAttributes`, `MaxPower` : décrivent les champs de même noms que l'on retrouve dans les descripteurs USB de type `CONFIGURATION` (se référer à la spécification USB) ;

- `cdev` : contient la référence du gadget composite auquel est associée cette configuration. Ce champ est assigné lors de l'appel à la fonction `usb_add_config()` ;

- `list` : point d'ancrage pour l'ensemble des configurations d'un gadget composite (via `usb_composite_dev->configs`) ;

- `functions` : regroupe l'ensemble des fonctions USB associées à la configuration ;

- `interface` : permet de faire la liaison entre interface USB et `usb_function`.

Le cycle de vie d'une telle structure commence par son allocation, puis une initialisation de ces différents champs, pour enfin être associé à un gadget composite (représenté par une `struct usb_composite_dev`) via la fonction `usb_add_config(struct usb_composite_dev *, struct usb_configuration *, ...)`.

1.2 Initialisation d'un composite driver (exemple de zero_bind())

La fonction `zero_bind()` appelée lors de l'enregistrement du gadget `zero` commence par effectuer les opérations suivantes :

1) Elle alloue des identifiants de descripteurs USB de type `STRING` et les renseigne dans l'objet `device_desc` qui héberge le descripteur `DEVICE` du gadget zero (référencé par le pointeur `cdev`). Ces chaînes de caractères décrivent le `VendorID`, le `ProductID`, et le `SerialID` de ce descripteur.

2) Elle récupère les fonctions USB : `SourceSink` et `Loopback`, via `usb_get_function_instance()`. Cette fonction retourne une `struct usb_function_instance` qui permet d'accéder à la fonction USB souhaitée (dans notre cas `func_inst_ss` et `func_inst_lb`, décrivant respectivement les fonctions `SourceSink` et `Loopback`). À noter que la fonction s'occupe de charger le module noyau définissant la fonction s'il n'est pas présent en mémoire.

3) Afin d'utiliser ces fonctions, il est nécessaire d'en récupérer des copies qui peuvent être personnalisées en fonction du besoin. C'est ce que fait `zero_bind()`, en positionnant les options souhaitées de `func_inst_ss` et `func_inst_lb` avant de les copier (`ss_opts` et `lb_opts`).

4) À partir de la structure précédente, elle récupère une copie privée de la fonction USB via `usb_get_function()` (ce mécanisme permet d'employer, au besoin, la même fonction USB au travers de copies indépendantes dans différentes configurations USB). Cette fonction appelle la méthode `alloc_func()` depuis la structure précédente (via son champ `fd` qui pointe vers la `struct usb_function_driver` correspondante, cf. section 5.3). Elle retourne une `struct usb_function`. Cette dernière comprend des méthodes (appelées lors des requêtes de l'hôte USB) ainsi que les ressources associées à la fonction (*endpoints*, USB *string descriptors*...). Nous revenons sur cette structure en section 5.3.

5) Elle ajoute au gadget *composite* (`cdev`) les deux fonctions USB ainsi obtenues :

5.1) Elle configure une `usb_configuration` pour chacune des fonctions USB : `sourcesink_driver` et `loopback_driver` (bien que leur nom soit trompeur).

5.2) Elle ajoute alors ces deux `usb_configuration` au gadget composite *zero* via `usb_add_config_only()`. Ainsi, un contrôleur USB hôte, pourra décider de choisir l'une ou l'autre des configurations, permettant d'accéder à l'une ou l'autre des fonctions.

5.3) Elle ajoute les fonctions proprement dites aux configurations via `usb_add_function()`. Cette fonction appelle notamment la méthode `bind()` de la `struct usb_function` qui lui est passée en argument, laquelle se charge d'allouer des ressources telles que les identifiants d'interfaces et de chaînes de caractères, ainsi que les *endpoints*.

6) Elle réinitialise les *endpoints* du PDC via `usb_ep_autoconfig_reset()` (cf. section 3.2.2).

7) Elle récupère enfin (via `usb_composite_overwrite_options(cdev, coverwrite)`) des paramètres spécifiques à utiliser dans les descripteurs USB qui seront transmis à l'hôte lors de l'étape d'énumération. Ces paramètres correspondent notamment au `VendorID` et au `ProductID`. Ils sont personnalisables au chargement du module `zero.ko`. Remarquons que la macro `USB_GADGET_COMPOSITE_OPTIONS()` (définie dans `include/linux/usb/composite.h`) est utilisée dans `zero.c` afin de créer l'objet `coverwrite`, ainsi que les paramètres du module (via `module_param_named()`).

À noter que les différentes fonctions que nous venons de citer sont définies par le *framework* Composite (`drivers/usb/gadget/composite.c` et `drivers/usb/gadget/functions.c`).

```
static int __init zero_bind(struct usb_composite_dev *cdev)
{
  struct f_ss_opts *ss_opts;
  struct f_lb_opts *lb_opts;
  int    status;

  status = usb_string_ids_tab(cdev, strings_dev);
  if (status < 0)
    return status;
```

```
device_desc.iManufacturer = strings_dev[USB_GADGET_MANUFACTURER_IDX].id;
device_desc.iProduct = strings_dev[USB_GADGET_PRODUCT_IDX].id;
device_desc.iSerialNumber = strings_dev[USB_GADGET_SERIAL_IDX].id;

setup_timer(&autoresume_timer, zero_autoresume, (unsigned long) cdev);

func_inst_ss = usb_get_function_instance("SourceSink");
if (IS_ERR(func_inst_ss))
  return PTR_ERR(func_inst_ss);

ss_opts =  container_of(func_inst_ss, struct f_ss_opts, func_inst);
ss_opts->pattern = gzero_options.pattern;
ss_opts->isoc_interval = gzero_options.isoc_interval;
ss_opts->isoc_maxpacket = gzero_options.isoc_maxpacket;
ss_opts->isoc_mult = gzero_options.isoc_mult;
ss_opts->isoc_maxburst = gzero_options.isoc_maxburst;
ss_opts->bulk_buflen = gzero_options.bulk_buflen;

func_ss = usb_get_function(func_inst_ss);
if (IS_ERR(func_ss)) {
  status = PTR_ERR(func_ss);
  goto err_put_func_inst_ss;
}

func_inst_lb = usb_get_function_instance("Loopback");
if (IS_ERR(func_inst_lb)) {
  status = PTR_ERR(func_inst_lb);
  goto err_put_func_ss;
}

lb_opts = container_of(func_inst_lb, struct f_lb_opts, func_inst);
lb_opts->bulk_buflen = gzero_options.bulk_buflen;
lb_opts->qlen = gzero_options.qlen;

func_lb = usb_get_function(func_inst_lb);
if (IS_ERR(func_lb)) {
  status = PTR_ERR(func_lb);
  goto err_put_func_inst_lb;
}

sourcesink_driver.iConfiguration = strings_dev[USB_GZERO_SS_DESC].id;
loopback_driver.iConfiguration = strings_dev[USB_GZERO_LB_DESC].id;

/* support autoresume for remote wakeup testing */
sourcesink_driver.bmAttributes &= ~USB_CONFIG_ATT_WAKEUP;
loopback_driver.bmAttributes &= ~USB_CONFIG_ATT_WAKEUP;
sourcesink_driver.descriptors = NULL;
loopback_driver.descriptors = NULL;
if (autoresume) {
  sourcesink_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;
  loopback_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;
  autoresume_step_ms = autoresume * 1000;
}

/* support OTG systems */
if (gadget_is_otg(cdev->gadget)) {
  sourcesink_driver.descriptors = otg_desc;
  sourcesink_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;
  loopback_driver.descriptors = otg_desc;
  loopback_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;
}

if (loopdefault) {
  usb_add_config_only(cdev, &loopback_driver);
  usb_add_config_only(cdev, &sourcesink_driver);
} else {
  usb_add_config_only(cdev, &sourcesink_driver);
  usb_add_config_only(cdev, &loopback_driver);
}
status = usb_add_function(&sourcesink_driver, func_ss);
if (status)
  goto err_conf_flb;

usb_ep_autoconfig_reset(cdev->gadget);
```

```
    usb_composite_overwrite_options(cdev, &coverwrite);

    INFO(cdev, "%s, version: " DRIVER_VERSION "\n", longname);

    return 0;

[... code  de gestion des erreurs ...]

}
```

1.3 Les fonctions USB

Une dernière couche logicielle permet de définir les fonctions d'un gadget USB. Il s'agit de la couche *functional driver*, également appelée *class driver* de par le nom donné par l'*USB group* aux spécifications de ces fonctions : les *class specifications*. On retrouve par exemple : l'*Audio Device Class Specification*, le *Mass Storage Class Specification*, etc. (http://www.usb.org/developers/docs/devclass_docs/)

Cette couche définit notamment les structures `usb_function_driver` et `usb_function`, que nous abordons dans la suite.

1.3.1 La structure usb_function_driver

Cette structure (`include/linux/usb/composite.h`) contient notamment des méthodes de gestion d'une fonction USB pour l'allocation de structures de type `usb_function_instance` et `usb_function`.

```
struct usb_function_driver {
        const char *name;
        struct module *mod;
        struct list_head list;
        struct usb_function_instance *(*alloc_inst)(void);
        struct usb_function *(*alloc_func)(struct usb_function_instance *inst);
};
```

Cette structure est définie par chaque fonction USB. Elle est fournie lors de leur enregistrement auprès de l'infrastructure Composite. Dans le cas de la fonction « Loopback », son enregistrement se fait de la façon suivante (`drivers/usb/gadget/function/f_loopback.c`) :

```
DECLARE_USB_FUNCTION(Loopback, loopback_alloc_instance, loopback_alloc);

int __init lb_modinit(void)
{
 int ret;
 ret = usb_function_register(&Loopbackusb_func);
 if (ret)
   return ret;
 return ret;
}
```

La macro `DECLARE_USB_FUNCTION()` crée une `struct usb_function_driver` nommée `Loopbackusb_func`, et y associe les fonctions `loopback_alloc_instance()` et `loopback_alloc()`.

Le *framework* Composite définit les fonctions `usb_get_function_instance()` et `usb_get_function()`, que nous avons vues utilisées précédemment dans le code de `zero_bind()` (cf. section 5.2). Ces fonctions ne font qu'appeler les méthodes correspondantes enregistrées dans la `struct usb_function_driver`. Pour exemple, voici le code de `usb_get_function()` (`drivers/usb/gadget/functions.c`) :

```
struct usb_function *usb_get_function(struct usb_function_instance *fi)
{
        struct usb_function *f;

        f = fi->fd->alloc_func(fi);
        if (IS_ERR(f))
                return f;
        f->fi = fi;
        return f;
}
```

Le rôle de `usb_get_function_instance(const char *name)` est de récupérer une référence sur une fonction USB enregistrée. Elle prend en paramètre le nom de la fonction et retourne une `struct usb_function_instance`. Il s'agit de l'instance définie par le module noyau correspondant et s'il n'est pas présent en mémoire la fonction s'occupe de le charger. De cette « instance », peuvent ensuite être allouées au travers de l'appel à la primitive `usb_get_function()` (qui appelle `usb_function_driver->alloc_func()`), une ou plusieurs `struct usb_function` (grâce au *function driver* référencé par l'instance). Cette structure est l'entité qui représente complètement une fonction USB et son état au sein de l'infrastructure Composite, et ainsi permet d'avoir des fonctions USB indépendantes issues d'une même souche. À noter toutefois qu'une fonction USB peut choisir de ne pas gérer de multiples copies.

Enfin, notons que le déchargement du module noyau de la fonction provoque son désenregistrement de l'infrastructure Composite :

```
void __exit lb_modexit(void)
{
 usb_function_unregister(&Loopbackusb_func);
}
```

1.3.2 La structure usb_function

Une fonction USB est représentée par une `struct usb_function`. Elle est associée de façon unique à une configuration via `usb_add_function()`, vu précédemment (cf. section 5.1.3). Cet appel entraîne l'exécution de la méthode `bind()` de la fonction USB dont le rôle est d'allouer les ressources nécessaires à son fonctionnement (comme les *endpoints*).

Les *function drivers* peuvent choisir leur propre stratégie pour gérer les différentes copies de `usb_function`. La stratégie la plus simple et de la déclarer de façon statique, ce qui signifie que la fonction ne peut être activée qu'une fois. Pour plus de flexibilité, afin de rendre possible l'utilisation de la fonction dans des configurations différentes, il est nécessaire pour le *function driver* de gérer de multiples structures `usb_function`.

```
struct usb_function {
        const char                      *name;
        struct usb_gadget_strings       **strings;
        struct usb_descriptor_header    **fs_descriptors;
        struct usb_descriptor_header    **hs_descriptors;
        struct usb_descriptor_header    **ss_descriptors;

        struct usb_configuration        *config;

        struct usb_os_desc_table        *os_desc_table;
        unsigned                        os_desc_n;

        /* configuration management:  bind/unbind */
        int                     (*bind)(struct usb_configuration *,
                                        struct usb_function *);
        void                    (*unbind)(struct usb_configuration *,
                                        struct usb_function *);
        void                    (*free_func)(struct usb_function *f);
        struct module           *mod;

        /* runtime state management */
        int                     (*set_alt)(struct usb_function *,
                                        unsigned interface, unsigned alt);
        int                     (*get_alt)(struct usb_function *,
                                        unsigned interface);
        void                    (*disable)(struct usb_function *);
        int                     (*setup)(struct usb_function *,
                                        const struct usb_ctrlrequest *);
        void                    (*suspend)(struct usb_function *);
        void                    (*resume)(struct usb_function *);

        /* USB 3.0 additions */
        int                     (*get_status)(struct usb_function *);
        int                     (*func_suspend)(struct usb_function *,
                                                u8 suspend_opt);

        struct list_head                list;
        DECLARE_BITMAP(endpoints, 32);
        const struct usb_function_instance *fi;
};
```

Cette structure contient notamment les champs suivants :

- `fs_descripors`, `hs_descripors`, `ss_descripors` : pointent vers les descripteurs USB correspondant à la fonction USB pour les vitesses respectives : full-spped, high-speed et super-speed. Si la valeur du champ est nulle, la fonction n'est pas disponible pour la vitesse correspondante.
- `config` : pointe vers la `struct usb_configuration` dans laquelle elle est présente.
- `bind` : est appelée lors de l'enregistrement de la fonction via `usb_add_function()`.
- `unbind` : défait ce qui a été alloué par `bind()`.
- `mod` : pointe vers le module noyau définissant la fonction.
- `set_alt` : est appelée lors de la réception par le PDC de la commande `SET_INTERFACE`.
- `get_alt` : est appelée lors de la réception par le PDC de la commande `GET_INTERFACE`.
- `disable` : est appelée lors d'une déconnexion à l'hôte ou lorsque ce dernier reconfigure le gadget.
- `setup` : est appelée pour gérer les requêtes de contrôle spécifiques à la fonction (cf. chapitre 9.3 de la spécification USB).
- `suspend`, `resume` : utilisées respectivement lorsque l'hôte passe en veille et puis se réveille.
- `get_status` : est appelée lors de la réception par le PDC de la commande `GET_STATUS` à destination de l'interface.
- `list` : sert de point d'ancrage à la liste de toutes les fonctions USB d'une `usb_configuration` (via son champ `functions`).
- `fi` : pointe sur la `usb_function_instance` dont est issue cette fonction USB.

Le cycle de vie de cette structure commence par sa création lors de l'initialisation d'un gadget composite, c'est-à-dire dans sa fonction `bind()` (cf. l'exemple de `zero_bind()` en section 5.2), via l'appel aux primitives de l'infrastructure Composite : `usb_get_function_instance()` et `usb_get_function()`. Elle est ensuite associée à une structure `usb_configuration`, via la primitive `usb_add_function()`. Cette dernière a pour effet d'exécuter sa méthode `bind()`, laquelle s'occupe de compléter les descripteurs USB pertinents (`INTERFACE`, `STRING`, ...) et d'allouer les ressources nécessaires à son fonctionnement (*endpoints*...).

Conclusion

Après ce long parcours visant à dégrossir l'infrastructure USB Gadget, nous encourageons le lecteur à poursuivre le chemin au sein des multiples gadgets USB définis dans le noyau, comme par exemple :

- Le gadget *File-backed Storage* (`g_mass_storage.ko`) qui implémente la classe USB *mass storage*. *Elle* présente un PDC comme un périphérique contenant jusqu'à huit disques de stockage. Ces disques sont définis à partir de fichiers ou de *block devices* sur la plateforme du PDC. Ils sont ensuite passés en paramètre du module.
- Le gadget *Webcam* (`g_webcam.ko`) qui implémente la classe composite USB *Audio and Video*. Elle fournit une API en espace utilisateur permettant de traiter les requêtes de contrôle UVC (*USB Video Class*) et de diffuser les données vidéo.

Enfin, signalons l'existence de l'infrastructure *USB Gadget ConfigFS* qui permet de définir des gadgets USB composites depuis l'espace utilisateur, mêlant des fonctions USB arbitraires définies en espace utilisateur ou en espace noyau au sein de différentes configurations USB. L'interaction avec cette infrastructure se fit au travers du système de fichiers *ConfigFS* (accessible après l'exécution d'une commande `mount -t configfs none /sys/kernel/config`). À noter que cette infrastructure succède à *FunctionFS* qui fut créée pour adapter le module *GadgetFS* (maintenant obsolète) à l'infrastructure Composite. ■

EXTENSION DE L'API ANDROID

Pierre Ficheux - Directeur technique Open Wide Ingénierie

Dans le numéro 8 de la revue Open Silicium, nous avions présenté l'utilisation d'Android dans le cadre d'un environnement industriel. À titre d'exemple, nous avions mis en place un nouveau pilote noyau pour un capteur de température USB puis intégré sommairement ce pilote dans l'environnement Android. Dans cet article, nous allons reprendre le même exemple en réalisant une intégration plus poussée conforme au standard Android. En l'occurrence, il s'agit de réaliser une extension de l'API pour un « service » lié à ce nouveau matériel. Il est bien évident que la même logique pourrait être utilisée pour un contrôleur de bus industriel absent de l'environnement Android, comme par exemple CAN, SPI ou I2C. Les tests ont été réalisés sur Android JB 4.3 dans l'environnement de l'émulateur Android et sur une carte BeagleBone Black (BBB) équipée d'un écran tactile.

1 Introduction

Nous ne reviendrons pas sur les généralités concernant le système Android. Pour cela, nous renvoyons le lecteur à l'article du numéro 8 d'*Open Silicium* ou aux nombreuses publications parues dans *GNU/Linux Magazine*. De même, notre talentueux camarade Benjamin Zores a publié aux Éditions Diamond un excellent ouvrage consacré à cette architecture **[1]**. Nous allons donc nous focaliser sur les différentes étapes liées à l'ajout d'un nouveau support de périphérique matériel à l'environnement Android. Nous précisons qu'il ne s'agit pas d'un nouveau contrôleur matériel Wi-Fi, Bluetooth ou autre caméra, mais bien d'ajouter un élément totalement absent du système, en l'occurrence un capteur de température **[4]**. Les exemples de l'article sont disponibles sur GitHub **[33]**.

Il faut noter que ce travail n'est jamais réalisé par le développeur Android courant et qu'il n'a de sens que dans un environnement industriel. Ce point est d'ailleurs un des freins tendant à restreindre le domaine d'utilisation d'Android (il y en a bien d'autres...) puisqu'un tel ajout est complexe et nécessite une bonne connaissance de sujets très mal documentés par Google. Il existe peu d'exemples mis à part l'article d'Opersys (Karim Yaghmour) cité en **[2]** ou décrit dans son livre **[29]**, mais qui ne correspond pas à un périphérique réel. Le cas du capteur de température est à la fois simple et concret et nous (Open Wide Ingénierie **[30]**) avons également pu réaliser une extension de l'API afin d'intégrer des tâches temps réel Xenomai **[3]**.

Les tests décrits dans cet article ont été réalisés sur l'émulateur Android construit à la compilation, mais également sur la carte BeagleBone Black (BBB) **[34]** équipée d'un écran tactile **[35]** connecté au bus d'extension (CAPE) de la carte. Il existe déjà une adaptation d'Android pour la BBB utilisant une sortie HDMI **[32]**, mais nous avons adapté ce portage afin de pouvoir utiliser l'écran tactile, ce qui est plus conforme à la notion de « device » Android. Ces travaux font l'objet d'un autre article **[37]** dans ce même numéro du journal et nous donnerons donc peu de détails sur ce sujet.

2 Support matériel dans Android

L'architecture d'Android est largement plus complexe que celle de GNU/Linux et l'ajout d'un support matériel implique de nombreuses modifications. Cette architecture est représentée par la figure 1 ci-après extraite de l'article Opersys déjà cité **[2]**. Nous avons précisé en bleu les répertoires concernés dans les sources d'Android (AOSP **[7]** pour *Android Open Source Project*). Chaque ajout de matériel correspond aux étapes suivantes.

1. Un pilote noyau, soit **/dev/foo** sur le schéma. Ce pilote est normalement sous licence GPL.
2. Une entrée dans la HAL Android **[9]** (pour *Hardware Abstraction Layer*), soit **libfoo.so** que l'on peut considérer comme un « pilote en espace utilisateur ». La bibliothèque doit être conforme à l'interface définie par Android dans **/hardware/libhardware**. Le pilote noyau et la bibliothèque sont fournis par le fabricant du matériel ou dans AOSP s'il s'agit d'un matériel officiel ou émulé. L'utilisation d'un tel découpage permet au fabricant de ne pas fournir le support matériel sous licence GPL, la bibliothèque pouvant être sous licence propriétaire (binaire **.so**).
3. Un *service* Android **[24]** comme il en existe pour la téléphonie, le Wi-Fi, le Bluetooth, etc. Ce dernier est divisé en deux parties, soit la partie JNI (en C/C++ afin de s'interfacer avec la HAL) et la partie Java qui contient le code effectif du service.
4. Une définition dans les classes **android.*** fournies avec le SDK afin que les développeurs d'application puissent l'utiliser. Cela implique une modification du SDK utilisé dans l'environnement de développement Android Studio **[25]**.

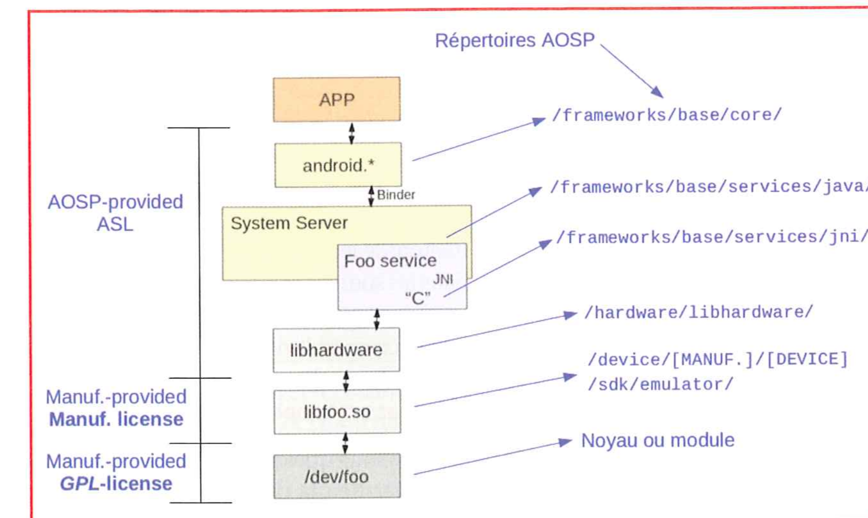


Figure 1 : Support matériel sous Android.

3 Ajout du nouveau service

Par abus de langage, l'ajout du service correspond à la mise en place des différents points décrits au point précédent. Après connexion à la cible, on peut obtenir la liste des services actifs en utilisant la commande suivante.

```
$ service list
Found 69 services:
0       phone: [com.android.internal.telephony.ITelephony]
1       iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2       simphonebook: [com.android.internal.telephony.IiccPhoneBook]
...
```

À l'issue de notre modification, nous aurons en plus la ligne indiquant la présence du service *temper* lié au capteur de température USB.

```
$ service list | grep temper
4       temper: [android.os.ITemperService]
```

3.1 Pilote noyau

Le pilote pour le capteur de température TEMPer **[4]** est développé pour un noyau Linux standard et n'a rien de spécifique à Android. Les détails du développement dépassent largement le sujet de cet article et nous renvoyons le lecteur aux références **[5]** et **[6]** de la bibliographie. Ce pilote fut d'ailleurs développé bien avant que l'auteur ne s'intéresse à Android !

Concrètement, la compilation du pilote **temper.c** produit un module **temper.ko** que l'on peut charger dans l'environnement Android à l'aide de la commande **insmod**.

```
# insmod temper.ko
```

Dans le cas précis de ce périphérique (USB), il est identifié comme appartenant à la classe HID (pour *Human Interface Device*) qui comprend les périphériques d'entrée de type clavier, souris, etc. Il est donc nécessaire de *détacher* le périphérique du pilote HID du noyau afin qu'il soit reconnu par notre pilote dédié. Le script **init_temper.sh** ci-dessous permet d'insérer correctement le pilote sous Android.

```
#!/system/bin/sh
log -t INIT_TEMPER init_temper

# Detach from USBHID
echo "1-1:1.0" > /sys/bus/usb/drivers/usbhid/unbind
echo "1-1:1.1" > /sys/bus/usb/drivers/usbhid/unbind

# Load driver + attach
insmod /data/temper.ko

# Android does not create /dev/temper0 but Linux
does (?)
ln -s /sys/class/usbmisc/temper0/device/temp /dev/
temper0
```

Le dernier point est assez étrange, car l'insertion du pilote USB ne créée pas automatiquement l'entrée **/dev/temper0** alors que c'est le cas pour un pilote simulé que nous avons utilisé pour mettre au point le système sans disposer du TEMPer. Le pilote réel fait partie de la classe de pilote *usbmisc* alors que le pilote simulé fait partie de la classe *misc*. À ce jour, nous n'avons

pas trouvé la raison de ce comportement qui n'a cependant pas d'influence sur la suite de l'article.

Une fois le pilote installé, la lecture du fichier spécial doit retourner la température codée sur un entier. L'obtention de la température s'effectue simplement par une suite d'appels système `open()`, `read()` puis `close()`. Nous verrons que ce point est important pour la mise en place globale du service.

```
# cat /dev/temper0
7834
```

L'automatisation du chargement au démarrage d'Android s'effectue en ajoutant les lignes suivantes à un fichier `/init*.rc` :

```
service init_temper /data/init_temper.sh
    class main
    oneshot
```

Notons que le mot-clé `service` ne correspond pas au démarrage global du service Android, sujet de cet article, mais d'une simple initialisation du pilote. Le service étudié est un service *Java* (et non un script ni un exécutable écrit en C/C++). Comme nous le verrons plus tard, le démarrage d'un service Java est réalisé par le *System Server* **[10][11]** correspondant au processus `system_server`.

Dans le cas de la carte BBB, le fichier `/init.am335xevm.rc` contient plusieurs entrées de ce type comme par exemple :

```
service pvr /system/bin/sgx/rc.pvr start
    class core
    oneshot
```

Ces lignes indiquent que le script est lancé une fois (et une seule) au démarrage du système. Nous rappelons que les fichiers `/init*.rc` sont démarrés par le programme `/init` d'Android, qui utilise une syntaxe différente – et plus riche – que le simple langage « shell » utilisé sur GNU/Linux. Sur la carte BeagleBone Black, nous avons :

```
$ ls -l /init*.rc
-rwxr-x--- root     root          2374 1970-01-01 00:00 init.am335xevm.rc
-rwxr-x--- root     root           787 1970-01-01 00:00 init.am335xevm.usb.rc
-rwxr-x--- root     root         19930 1970-01-01 00:00 init.rc
-rwxr-x--- root     root          1795 1970-01-01 00:00 init.trace.rc
-rwxr-x--- root     root          3915 1970-01-01 00:00 init.usb.rc
```

La partie système d'Android n'étant pas trop bien documentée, on devra se contenter de la description de la syntaxe disponible dans le fichier `system/core/init/readme.txt` des sources d'Android. Il faut cependant noter qu'il existe systématiquement un script lié à l'architecture matérielle de la cible, soit AM335x pour la BBB et Goldfish pour l'émulateur (donc un fichier `/init.goldfish.rc`).

Dans le cas de l'émulateur Android, il est nécessaire de simuler le capteur de température USB, donc d'utiliser un pilote qui n'a rien à voir avec l'interface USB. Une autre solution – plus simple – est de créer un fichier contenant une valeur de température factice.

```
# echo 8765 > /dev/temper0
# chmod 666 /dev/temper0
```

Cette solution est beaucoup plus simple dans le cas d'une courte expérimentation, car le noyau binaire fourni dans AOSP pour la cible ARM émulée (nommée *Goldfish*) n'intègre pas le support des modules dynamiques. Il n'est donc pas possible d'ajouter un pilote, sauf en recompilant le noyau à partir des sources comme décrit en **[19]**. L'utilisation de `chmod` est importante afin de permettre à l'application d'accéder au fichier, car les droits d'accès sous Android sont quelque peu différents de ceux sous GNU/Linux, l'application Java n'ayant pas les droits administrateur.

3.2 Module HAL

L'étape suivante correspond à l'extension de la HAL pour l'ajout du support TEMPer. Nous rappelons que tout comme le pilote noyau, ce module est fourni par le fabricant de matériel et varie donc en fonction de la cible utilisée. Les fichiers AOSP liés à la cible sont placés dans le répertoire `device/<constructeur>/<cible>`. Un mécanisme d'héritage basé sur des macros GNU-Make permet de traiter relativement simplement les caractéristiques logicielles et matérielles des cibles en « dérivant » les configurations, voir **[31]**. Dans le cas de la BBB, il est donc nécessaire d'ajouter un répertoire `device/ti/beagleboneblack/temperhw`. Ce répertoire contient le fichier `Android.mk` (fichier `Makefile` pour Android) et le fichier `temperhw_bbb.c` correspondant au code source de la bibliothèque. Nous invitons le lecteur à consulter le contenu du fichier `Android.mk` dans les sources disponibles avec l'article. Le cas de l'émulateur est plus particulier, car il est basé sur QEMU. Par principe, *aucune* plateforme cible ne dérive de l'émulateur et nous devons ajouter la fonctionnalité directement aux sources de QEMU dans `sdk/emulator/temperhw`.

Nous présentons ci-dessous quelques extraits du code de la bibliothèque, en premier lieu la définition des *méthodes* (fonctions). La méthode `open_temperhw()` correspond à l'ouverture du service. Dans le code qui suit, nous pouvons remarquer l'utilisation de la macro `ALOGI()` permettant d'afficher des traces d'information (code `I`) dans le service de trace Android accessible par la commande `logcat`.

```
static struct hw_module_methods_t temperhw_module_methods = {
   .open = open_temperhw
};

struct hw_module_t HAL_MODULE_INFO_SYM = {
   .tag = HARDWARE_MODULE_TAG,
   .version_major = 1,
   .version_minor = 0,
   .id = TEMPERHW_HARDWARE_MODULE_ID,
  .name = "Temper HW Module",
  .author = "Open Wide",
  .methods = &temperhw_module_methods,
};
```


Nous donnons ci-dessous le contenu de la fonction :

```
static int open_temperhw(const struct hw_module_t* module, char const* name,
struct hw_device_t** device)
{
  struct temperhw_device_t *dev = malloc(sizeof(struct temperhw_device_t));
  memset(dev, 0, sizeof(*dev));

  ALOGI("open_temperhw\n");
  dev->common.tag = HARDWARE_DEVICE_TAG;
  dev->common.version = 0;
  dev->common.module = (struct hw_module_t*)module;
  dev->common.close = (int (*)(struct hw_device_t *))close_temperhw;
  dev->read = temperhw__read;
  dev->test = temperhw__test;
  *device = (struct hw_device_t*) dev;
  return 0;
}
```

La fonction `temperhw__read()` effectue les mêmes actions que le test manuel avec la commande `cat` (ouverture/lecture/fermeture).

```
int temperhw__read(char* buffer, int length)
{
  int retval, fd;

  /*
   * Do something like "cat /dev/temper0" (open/read/close)
   */
  fd = open("/dev/temper0", O_RDONLY);

  if (fd < 0) {
    ALOGI("Failed to open device!\n");
    return -1;
  }

  ALOGI("Temper HW - read() for %d bytes called", length);
  retval = read(fd, buffer, length);
  ALOGI("Temper HW - read() = %s %d\n", buffer, retval);

  close(fd);

  return retval;
}
```

Le fichier `temperhw.h` définissant l'interface doit être ajouté au répertoire `hardware/libhardware/include/hardware` des sources d'Android. Ce répertoire contient les fichiers d'en-tête pour toutes les interfaces matérielles disponibles (y compris l'émulateur). Comme indiqué précédemment, cette partie est fournie par AOSP et n'est pas dépendante de la cible matérielle.

```
$ ls -l hardware/libhardware/include/hardware/
total 524
-rw-r--r-- 1 pierre pierre 16850 Aug  8  2013 audio.h
-rw-r--r-- 1 pierre pierre 51230 Aug  8  2013 audio_effect.h
-rw-r--r-- 1 pierre pierre 18577 Aug  8  2013 audio_policy.h
-rwxr-xr-x 1 pierre pierre 15486 Aug  8  2013 bluetooth.h
-rw-r--r-- 1 pierre pierre  2707 Aug  8  2013 bt_av.h
-rw-r--r-- 1 pierre pierre  1699 Aug  8  2013 bt_gatt.h
...
```

Voici la partie principale du fichier ajouté pour le nouveau service.

```
#define TEMPERHW_HARDWARE_MODULE_ID "temperhw"

struct temperhw_device_t {
  struct hw_device_t common;

  int (*read)(char* buffer, int length);
  int (*test)(int value);
};
```

3.3 Partie JNI

Nous rappelons que JNI **[8]** est une fonctionnalité standard du JDK permettant aux fonctions Java d'appeler des fonctions natives écrites en C/C+ (la réciproque étant également possible). Ce code est bien évidemment indépendant de la plateforme cible utilisée.

Les services Android utilisent JNI afin accéder à la couche HAL décrite précédemment. Le code à ajouter correspond au fichier `frameworks/base/services/jni/com_android_server_TemperService.cpp`. Bien entendu, il faut également ajouter une nouvelle entrée au fichier `Android.mk` du répertoire.

```
LOCAL_SRC_FILES:= \
    com_android_server_VibratorService.cpp \
    com_android_server_location_GpsLocationProvider.cpp \
    com_android_server_connectivity_Vpn.cpp \
    com_android_server_TemperService.cpp \
    ...
```

Dans `com_android_server_TemperService.cpp`, les fonctions natives sont enregistrées dans un tableau comme suit.

```
static JNINativeMethod method_table[] = {
  { "init_native", "()I", (void*)init_native },
  { "finalize_native", "(I)V", (void*)finalize_native },
  { "read_native", "(I[B)I", (void*)read_native },
  { "test_native", "(II)I", (void*)test_native},
};
```

Le deuxième champ (signature) permet de coder les paramètres de la fonction ainsi que le type retourné (`I` pour `int`, `V` pour `void`, `[B` pour `byte`, etc.). La fonction `read_native()` correspond finalement à l'appel depuis Java à `temperhw__read()` dans la HAL. On doit également enregistrer le nouveau service en définissant la fonction :

```
int register_android_server_TemperService(JNIEnv * env)
{
  return jniRegisterNativeMethods(env, "com/android/server/TemperService",
method_table, NELEM(method_table));
};
```

Cette fonction doit être déclarée, puis appelée dans le fichier `onload.cpp`.

```
namespace android {
...
int register_android_server_TemperService(JNIEnv* env);
...
};
extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    ...
    register_android_server_TemperService(env);
    ...
}
```

3.4 Partie Java

Le code Java correspond à la partie *fonctionnelle* du service, car n'oublions pas que c'est le langage de prédilection de l'environnement Android. Le nouveau service est défini dans le répertoire **frameworks/base/services/java/com/android/server**. Dans notre cas, il correspond au fichier **TemperService.java**. Le constructeur de la classe fait appel à la fonction **init_native()** citée précédemment afin d'allouer les ressources utilisées.

```
public class TemperService extends ITemperService.Stub {
  private static final String TAG = "TemperService";
  private Context mContext;
  private int mNativePointer;
  public TemperService(Context context){
    super();
    mContext = context;
    Log.i(TAG, "Temper Service started");
    mNativePointer = init_native();
    Log.i(TAG, "test() returns "+ test_native(mNativePointer, 20));
  }
...
} ;
```

La fonction **read()** obtient les données auprès de la couche HAL via **read_native()** et retourne une chaîne de caractères.

```
public String read(int maxLength) {
  int length;
  byte[] buffer = new byte[maxLength];
  length = read_native(mNativePointer, buffer);
  return new String(buffer, 0, length);
}
```

Comme nous l'avons introduit au début de l'article, le nouveau service (Java) est démarré par le *System Server* (soit **com.android.server.SystemServer**) lui-même démarré par **zygote** par la fonction **startSystemServer()** **[12]**. L'application ne réalise donc pas d'appel *direct* au service, mais passe par un *manager* que nous décrirons dans le paragraphe consacré au SDK. Il existe un manager pour chaque service (Power Manager, Package Manager, Activity Manager, etc.) et l'ajout du nouveau service conduira à la mise à disposition d'un *Temper Manager* dans le SDK.

Nous rappelons que la communication interne à Android utilise *Binder* **[14]**, une couche d'IPC spécifiquement développée par Google afin de remplacer les antiques IPC d'UNIX. Binder est basé sur la notion de *bus logiciel* par opposition avec une communication directe entre les processus. La version utilisée dans Android est dérivée d'OpenBinder **[15]**, développé pour BeOS. En effet, l'équipe de développement initiale de la société Android Inc. comptait plusieurs anciens développeurs de Be Inc. Binder n'est cependant pas directement compatible avec OpenBinder.

Il existe déjà des solutions similaires sous Linux comme *D-Bus*, mais Google a eu la clairvoyance de choisir une autre voie, D-Bus étant réputé pour sa complexité et ses performances pas toujours à la hauteur (pour ne pas dire catastrophiques). D-bus n'est d'ailleurs pas adapté aux besoins d'Android puisqu'il a une approche principalement événementielle. Le fonctionnement de Binder est basé sur un pilote noyau avec un fichier spécial **/dev/binder** accessible via la bibliothèque **libbinder** puis par des classes Java.

Pour revenir à **zygote**, nous rappelons que c'est le premier processus Java exécuté lors du démarrage d'Android, il est donc fils du processus **init** et les applications sont filles de **zygote**.

```
# ps | grep zygote
root      92    1     458472 35872 ffffffff 400af0e4 S zygote
 # ps | grep com.android
u0_a37    630   92    473416 27832 ffffffff 400afebc S com.android.systemui
u0_a40    668   92    469816 21632 ffffffff 400afebc S com.android.
inputmethod.l atin
radio     680   92    474592 22676 ffffffff 400afebc S com.android.phone
...
# ps | grep system_server
system    487   92    536596 33544 ffffffff 40055fc0 S system_server
```

Le processus **system_server** démarre les (nombreux) services existants (Power, Package, Activity, etc.) et le démarrage de notre nouveau service implique donc une modification du code source de la fonction **run()** dans **frameworks/base/services/java/com/android/server/SystemServer.java**.

```
try {
    Slog.i(TAG, "Temper Service");
    ServiceManager.addService(Context.TEMPER_SERVICE, new
TemperService(context));
} catch (Throwable e) {
   Slog.e(TAG, "Failure starting TemperService service", e);
}
```

3.5 Modification du SDK

Suite à sa mise en place, nous allons voir comment rendre le nouveau service utilisable dans une application. Cette dernière étape consiste donc à « exposer » ce nouveau service aux couches de développement d'Android (SDK), ce afin de pouvoir l'utiliser dans un programme Java. Avant de décrire les modifications réalisées pour l'ajout de notre service, il nous paraît judicieux de rappeler le cas d'un service existant largement utilisé, soit la gestion de l'énergie.

L'accès à un service passe par l'utilisation de la méthode **getSystemService()** dans la classe **Context** **[16]** permettant d'accéder aux informations globales de l'application. Si l'on met de côté les applications simplistes de type *Hello World*, la quasi-totalité des applications Android fait référence à cette classe par une ligne :

```
import android.content.Context;
```

L'extrait de code ci-dessous provient de l'application *Phone* fournie avec AOSP. L'exemple correspond au fichier **packages/apps/Phone/src/com/android/phone/BluetoothPhoneService.java**. L'application utilise le *Power Service* (gestion de l'énergie) et la fonction **getSystemService()** est donc appelée afin de créer une variable liée au type de *manager* pour le service choisi (**PowerManager** dans ce cas). L'application utilise ensuite un *wake lock* **[17]** afin de garder le système éveillé durant l'appel. Nous rappelons que le *wake lock* est une spécificité – controversée - d'Android (ou *Androidism)* permettant à l'application de prendre la main sur la politique de mise en veille du système.

```
/**
 * Bluetooth headset manager for the Phone app.
 * @hide
 */
public class BluetoothPhoneService extends Service {
    ...
    @Override
    public void onCreate() {
        super.onCreate();
        mCM = CallManager.getInstance();
        mAdapter = BluetoothAdapter.getDefaultAdapter();
        if (mAdapter == null) {
            if (VDBG) Log.d(TAG, "mAdapter null");
            return;
        }
        mPowerManager = (PowerManager) getSystemService(Context.
POWER_SERVICE);
        mStartCallWakeLock = mPowerManager.newWakeLock(PowerManager.
PARTIAL_WAKE_LOCK,
                                                      TAG +
":StartCall");
        ...
   }
}
```

Pour déclarer le *manager*, on devra :

- définir l'interface dans **frameworks/base/core/java/android/content/Context.java**

```
public static final String POWER_SERVICE = "power";
```

soit pour notre service :

```
public static final String TEMPER_SERVICE = "temper";
```

- définir l'implémentation dans **frameworks/base/core/java/android/app/ContextImpl.java**

```
     registerService(POWER_SERVICE, new ServiceFetcher() {
                public Object createService(ContextImpl ctx) {
                     IBinder b = ServiceManager.getService(POWER_SERVICE);
                         IPowerManager service = IPowerManager.Stub.
asInterface(b);
                          return new PowerManager(ctx.getOuterContext(),
service, ctx.mMainThread.getHandler());
                     }});
```

soit pour notre service :

```
   registerService(TEMPER_SERVICE, new ServiceFetcher() {
                public Object createService(ContextImpl ctx) {
                    IBinder b = ServiceManager.getService(TEMPER_SERVICE);
                    ITemperService service = ITemperService.Stub.
asInterface(b);
                     return new TemperManager(service);
             }});
```

- définir le code du *manager* dans **frameworks/base/core/java/android/os/PowerManager.java**

```
public final class PowerManager {
   private static final String TAG = "PowerManager";
   ...
}
```

soit pour notre service dans **frameworks/base/core/java/android/os/TemperManager.java**

```
   public class TemperManager{
     private static final String TAG = "TemperManager";
     private final ITemperService mService;
     public String read (int maxLength){
       try {
     return mService.read(maxLength);
       } catch (RemoteException e){
     Log.e(TAG, "RemoteExecption in read: ", e);
     return null;
       }
     }
     public TemperManager(ITemperService service){
       mService = service;
     }
   }
```

Nous remarquons bien évidemment que le code de notre service est beaucoup plus simple, car il définit uniquement une fonction de lecture **read()**.

Nous avons précédemment évoqué l'utilisation de Binder pour la communication interne sous Android. Les développeurs d'application n'utilisent pas directement Binder et l'on doit donc déclarer les fonctions du nouveau service un utilisant un langage dédié nommé *AIDL* **[13]** (pour *Android Interface Description Language*) dont la syntaxe est proche de celle du Java. Cela correspond à un fichier **.aidl** qui sera traité par l'utilitaire **aidl** afin de générer le code nécessaire. Dans la documentation Android, on évoque la notion de *mashalling* **[36]** à propos du code produit à partir d'AIDL. Ce concept correspond à la conversion d'objets – décrits dans le fichier AIDL – en format de données utilisable pour la communication (Binder).

Nous présentons ci-dessous un extrait du code correspondant au *Power Manager* dans **frameworks/base/core/java/android/os/IPowerManager.aidl**.

```
interface IPowerManager
{
   // WARNING: The first two methods must remain the first two methods
because their
   // transaction numbers must not change unless IPowerManager.cpp is
also updated.
interface IPowerManager
{
   // WARNING: The first two methods must remain the first two methods
because their
   // transaction numbers must not change unless IPowerManager.cpp is
also updated.


   void acquireWakeLock(IBinder lock, int flags, String tag, in
WorkSource ws);
   void releaseWakeLock(IBinder lock, int flags);
   ...
}
```

Dans le cas de notre service, le fichier `frameworks/base.core/java/android/os/ITemperService.aidl` contient simplement le code suivant puisque la lecture est la seule fonctionnalité définie par le service.

```
package android.os;

interface ITemperService{
  /**
  * {@hide}
  */
  String read(int maxLength);
}
```

4 Compilation d'AOSP

La procédure de compilation d'AOSP est largement documentée dans la documentation Google **[18]**. Nous y reviendrons également dans l'article concernant l'adaptation d'AOSP à la carte BBB.

La compilation est réalisée sur une Ubuntu 12.04.3 LTS (version officielle conseillée par Google), mais devrait également fonctionner sur la dernière version 14.04 ainsi que sur d'autres distributions GNU/Linux. L'utilisation de Jelly Bean 4.3 nous conduit au choix du JDK 1.6, version officielle fournie par SUN/Oracle.

```
$ type java
java est /usr/java/jdk1.6.0_41/bin/java
```

La branche AOSP utilisée est `android-4.3_r2.1`. Il est bien entendu nécessaire d'appliquer les « patches» fournis dans les compléments de l'article, ce afin de mettre en place les modifications décrites précédemment. Pour cela, nous allons tout d'abord rappeler quelques éléments importants concernant la gestion des sources d'AOSP.

Comme vous l'aurez constaté, les sources d'AOSP correspondent à plusieurs *dizaines* de Go (10 Go pour la version 2.3, plus de 16 Go pour la version 4.3, près 40 Go pour la version 4.4). Il existe plusieurs raisons à cela. En premier lieu, les « sources » ne contiennent pas uniquement des fichiers de texte, mais également les binaires des chaînes de compilation pour les différents systèmes d'exploitation utilisables (en l'occurrence GNU/Linux, Mac OS X... mais pas Windows !). Ces outils représentent près de 4 Go d'espace sur JB 4.3. Ensuite, et contrairement à la majorité des projets *open source*, Google fournit *l'intégralité* des composants externes utilisés pour Android (près de 2 Go sur JB 4.3). Enfin, AOSP contient les sources de la distribution à installer sur la cible, mais également tous les éléments du SDK à importer dans l'environnement de développement Android Studio.

AOSP utilise le système de gestion de version *Git* (co-écrit par Linus Torvalds), mais il n'est pas possible de gérer un tel espace dans un seul arbre Git. À titre de comparaison, le répertoire des sources d'un noyau Linux récent (3.x) représente environ 600 Mo, soit finalement pas grand-chose par rapport à Android. AOSP est donc découpé en *projets* qui ont chacun leur propre arborescence Git (donc un répertoire `.git`). Sur certains portages d'AOSP, le noyau Linux fait d'ailleurs partie de la (longue) liste de ces projets.

Comme on peut le voir dans la documentation Google précitée **[18]**, la gestion globale d'AOSP est basée sur l'utilitaire `repo` écrit par Google. Si l'on se place dans le répertoire des sources AOSP (mettons `work_43`), on peut obtenir facilement la liste et le nombre de dépôts utilisés dans JB 4.3.

```
$ cd work_43
$ repo list
abi/cpp : platform/abi/cpp
bionic : platform/bionic
bootable/bootloader/legacy : platform/bootable/bootloader/legacy
...
```

Le comptage du nombre de projets dans AOSP (plusieurs centaines) justifie le volume occupé par les sources.

```
$ repo list | wc -l
362
```

Cependant, l'outil `repo` est assez basique et son but est avant tout de faciliter la récupération des sources et gérer des modifications comme les nôtres n'est pas vraiment simple. Ce point n'est pas un problème pour Google vu qu'Android est tout sauf un projet collaboratif et que Google est tout sauf une *open source company* !

4.1 Production des patches

Nous allons ici traiter le cas des patches pour AOSP 4.3 dans le cas – plus simple – de l'émulateur. Après modification et ajout des fichiers décrits, nous pouvons obtenir l'état de l'arbre des sources par la commande suivante.

```
$ repo status
project frameworks/base/        (*** NO BRANCH ***)
 -m Android.mk
 -m core/java/android/app/ContextImpl.java
 -m core/java/android/content/Context.java
 -- core/java/android/os/ITemperService.aidl
 -- core/java/android/os/TemperManager.java
 -m services/java/com/android/server/SystemServer.java
 -- services/java/com/android/server/TemperService.java
 -m services/jni/Android.mk
 -- services/jni/com_android_server_TemperService.cpp
 -m services/jni/onload.cpp
project hardware/libhardware/        (*** NO BRANCH ***)
 -- include/hardware/temperhw.h
project sdk/        (*** NO BRANCH ***)
 -- emulator/temperhw/Android.mk
 -- emulator/temperhw/temperhw_qemu.c
```


Les modifications sont affichées pour chaque projet affecté (3 dans notre cas). Une ligne commençant par `-m` correspond à un fichier modifié et une ligne commençant par `--` à un fichier ajouté (donc absent de l'arbre Git courant). L'outil `repo` ne permet pas d'ajouter de fichiers à un projet (!) et l'on doit donc utiliser directement la commande `git`.

```
$ cd frameworks/base
$ git add <fichiers>
```

La commande `repo` dispose cependant d'une option `forall` permettant d'automatiser les tâches pour les 3 projets modifiés.

```
$ repo forall frameworks/base hardware/libhardware sdk -c 'git add .'
```

On peut alors vérifier le nouvel état de l'arbre, la présence du marqueur `A-` nous indique des fichiers ajoutés.

```
$ repo status
project frameworks/base/            (*** NO BRANCH ***)
 -m    Android.mk
 -m    core/java/android/app/ContextImpl.java
 -m    core/java/android/content/Context.java
 A-    core/java/android/os/ITemperService.aidl
 A-    core/java/android/os/TemperManager.java
 -m    services/java/com/android/server/SystemServer.java
 A-    services/java/com/android/server/TemperService.java
 -m    services/jni/Android.mk
 A-    services/jni/com_android_server_TemperService.cpp
 -m    services/jni/onload.cpp
project hardware/libhardware/          (*** NO BRANCH ***)
 A-    include/hardware/temperhw.h
project sdk/                           (*** NO BRANCH ***)
 A-    emulator/temperhw/Android.mk
 A-    emulator/temperhw/temperhw_qemu.c
```

On peut ensuite produire le patch global en utilisant la commande suivante.

```
$ repo diff
project frameworks/base/
diff --git a/Android.mk b/Android.mk
index 151621c..592bd05 100644
--- a/Android.mk
+++ b/Android.mk
@@ -150,6 +150,7 @@ LOCAL_SRC_FILES += \
  core/java/android/os/IUpdateLock.aidl \
  core/java/android/os/IUserManager.aidl \
  core/java/android/os/IVibratorService.aidl \
...
```

Là encore, le résultat contient la liste des patches à appliquer aux différents projets, donc aux différents dépôts Git. Curieusement, il n'existe pas d'option `patch` ni `apply` à la commande `repo` permettant d'appliquer ces modifications. La justification du leader de l'équipe Android de l'époque (Jean-Baptiste Queru alias JBQ) est sans appel **[20]** et va à l'encontre du fonctionnement de la plupart des communautés de développement open source. Cela prouve si c'était encore nécessaire que le projet AOSP n'est pas un projet communautaire. JBQ a d'ailleurs fini par quitter le projet AOSP puis plus récemment Google !

« The Android team doesn't accept patches, we only accept commits uploaded to Gerrit through repo upload. »

JBQ

La même discussion contient une réponse d'un utilisateur ayant réalisé un court script permettant de traiter le fichier des modifications. Nous l'avons légèrement adapté afin d'utiliser la commande `git apply` et non la commande `patch`.

```
#!/bin/sh
## Script to patch up diff reated by `repo diff`

if [ -z "$1" ] || [ ! -e "$1" ]; then
    echo "Usages: $0 <repo_diff_file>";
    exit 0;
fi

rm -fr _tmp_splits*
cat $1 | csplit -qf '' -b "_tmp_splits.%d.diff" - '/^project.*\/$/' '{*}'

working_dir=`pwd`

for proj_diff in `ls _tmp_splits.*.diff`
do
    chg_dir=`cat $proj_diff | grep '^project.*\/$' | cut -d " " -f 2`
    echo "FILE: $proj_diff $chg_dir"
    if [ -e $chg_dir ]; then
        ( cd $chg_dir; \
#           cat $working_dir/$proj_diff | grep -v '^project.*\/$' | patch
-Np1 --dry-run;);
            cat $working_dir/$proj_diff | grep -v '^project.*\/$' | git
apply --verbose;);
    else
        echo "$0: Project directory $chg_dir don't exists.";
    fi
done
```

Il suffit alors d'exécuter ce script dans le répertoire des sources Android en supposant que l'on a dirigé le contenu du patch vers un fichier `temper.patch`.

```
$ repo_apply.sh <path>/temper.patch
FILE: _tmp_splits.0.diff
fatal: unrecognized input
FILE: _tmp_splits.1.diff frameworks/base/
<stdin>:203: trailing whitespace.
<stdin>:291: trailing whitespace.
    return (jint)dev;
<stdin>:298: trailing whitespace.
    if (dev == NULL)
Checking patch Android.mk...
Checking patch core/java/android/app/ContextImpl.java...
Checking patch core/java/android/content/Context.java...
Checking patch core/java/android/os/ITemperService.aidl...
Checking patch core/java/android/os/TemperManager.java...
Checking patch services/java/com/android/server/SystemServer.java...
Checking patch services/java/com/android/server/TemperService.java...
Checking patch services/jni/Android.mk...
Checking patch services/jni/com_android_server_TemperService.cpp...
Checking patch services/jni/onload.cpp...
Applied patch Android.mk cleanly.
Applied patch core/java/android/app/ContextImpl.java cleanly.
Applied patch core/java/android/content/Context.java cleanly.
Applied patch core/java/android/os/ITemperService.aidl cleanly.
Applied patch core/java/android/os/TemperManager.java cleanly.
Applied patch services/java/com/android/server/SystemServer.java cleanly.
Applied patch services/java/com/android/server/TemperService.java cleanly.
Applied patch services/jni/Android.mk cleanly.
Applied patch services/jni/com_android_server_TemperService.cpp cleanly.
```

```
Applied patch services/jni/onload.cpp cleanly.
warning: 3 lines add whitespace errors.
FILE: _tmp_splits.2.diff hardware/libhardware/
Checking patch include/hardware/temperhw.h...
Applied patch include/hardware/temperhw.h cleanly.
FILE: _tmp_splits.3.diff sdk/
Checking patch emulator/temperhw/Android.mk...
Checking patch emulator/temperhw/temperhw_qemu.c....
Applied patch emulator/temperhw/Android.mk cleanly.
Applied patch emulator/temperhw/temperhw_qemu.c
cleanly.
```

4.2 Compilation

Cette phase correspond à la production de la nouvelle « ROM » (comme on dit sur les sites de téléphonistes !). Elle est conforme à la procédure décrite en **[18]** mis à part un léger détail.

Tout d'abord, nous chargeons les variables d'environnement comme décrit dans la documentation Google puis nous sélectionnons la cible par la commande **lunch**.

```
$ cd work_43
$ . build/envsetup.sh
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. aosp_arm-eng
     2. aosp_x86-eng
     3. aosp_mips-eng
     4. vbox_x86-eng
     5. aosp_deb-userdebug
     6. aosp_flo-userdebug
     7. full_grouper-userdebug
     8. full_tilapia-userdebug
     9. mini_armv7a_neon-userdebug
     10. mini_mips-userdebug
     11. mini_x86-userdebug
     12. full_mako-userdebug
     13. full_maguro-userdebug
     14. full_manta-userdebug
     15. full_toro-userdebug
     16. full_toroplus-userdebug
     17. full_arndale-userdebug
     18. beagleboneblack-eng
     19. full_panda-userdebug

Which would you like? [aosp_arm-eng] 1
```

La cible choisie correspond à la plateforme ARM émulée (Goldfish). Dans le cas de la carte BBB, la procédure est strictement la même en sélectionnant la cible numéro 18. Cette cible existera après la modification d'AOSP décrite dans l'article **[37]** cité au début.

Sachant que nous avons modifié l'API Android (dans le répertoire **frameworks/**), il est nécessaire de mettre à jour la définition de l'API par la commande :

```
$ make [-j N] update-api
```

On peut alors compiler AOSP par la commande :

```
$ make [-j N]
```

Il est fortement recommandé d'utiliser l'option **-j N**, la valeur **N** correspondant au nombre de cœurs plus un. Le temps de compilation est d'environ 70 minutes sur un PC i7 (option **-j 5**) disposant d'un disque SSD et de 8 Go de RAM !

5 Test du service ajouté

Lorsque la compilation est terminée, il suffit d'utiliser la commande **emulator** produite dans l'arborescence des sources AOSP. Dans le cas de la carte BBB, nous utiliserons la commande **fastboot** pour installer Android sur la carte comme décrit en **[32]** et **[37]**. Avant de réaliser une application utilisant les classes ajoutées, nous allons tout d'abord tester le fonctionnement du nouveau service à l'aide des outils internes à Android, en l'occurrence :

- La commande **adb** **[21]** (*Android Debug Bridge*) permettant – entre autres – la connexion à la cible ;
- La commande **logcat** **[22]** dédiée à l'exploitation des traces d'Android ;
- La commande **service** **[23]** permettant de tester les services Android.

L'exécution de la commande **emulator** conduit à l'affichage suivant.

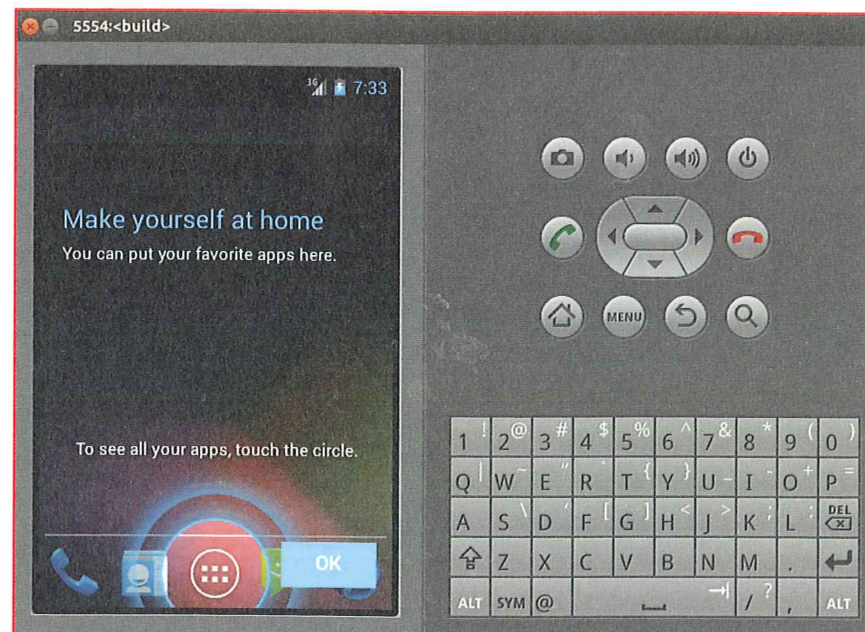


Figure 2 : Émulateur Android.

Cependant, la partie graphique du système ne nous intéresse pas pour l'instant et nous allons ouvrir une connexion vers la cible en utilisant la commande suivante.

```
$ adb shell
```


On peut alors vérifier la présence du nouveau service.

```
# service list | grep temper
7     temper: [android.os.ITemperService]
```

Nous rappelons que le but du service est d'obtenir la valeur de la température – soit le contenu du fichier **/dev/temper0** – via la HAL Android. Comme indiqué au début de l'article, nous créons un fichier factice contenant une valeur de température.

```
# echo 8888 > /dev/temper0
# chmod 666 /dev/temper0
```

La commande **logcat** utilisée sans paramètre affiche le flux des messages de trace système comme on pourrait le faire avec la commande **tail -f** sous GNU/Linux. Nous l'utiliserons ici pour afficher les traces du nouveau service.

```
W/ActivityManager(  492): Failed setting process group of 877 to 0
W/System.err(  492):    at android.os.Process.setProcessGroup(Native Method)
W/System.err(  492):    at com.android.server.am.ActivityManagerService.updateOomAdjLocked(ActivityManager
Service.java:13859)
W/System.err(  492):    at com.android.server.am.ActivityManagerService.updateOo
mAdjLocked(ActivityManagerService.java:13900)
W/System.err(  492):    at android.app.ActivityManagerNative.onTransact(Activity
ManagerNative.java:830)
W/System.err(  492): at com.android.server.am.ActivityManagerService.onTransact(ActivityManagerService.
java:1737)
W/System.err(  492):    at android.os.Binder.execTransact(Binder.java:388)
W/System.err(  492):    at dalvik.system.NativeStart.run(Native Method)
```

La commande **service** permet de tester un service en lui envoyant les « codes des méthodes » via Binder. On peut par exemple afficher (expand) puis masquer (collapse) la barre d'état par les commandes :

```
# service call statusbar 1
Result: Parcel(00000000    '....')
# service call statusbar 2
Result: Parcel(00000000    '....')
```

Le valeurs sont définies dans un fichier produit à la compilation d'Android, soit **out/target/common/obj/ JAVA_LIBRARIES/framework_intermediates/src/core/java/com/android/internal/ statusbar/IStattusBarService.java**.

```
static final int TRANSACTION_expand = (android.os.IBinder.FIRST_CALL_ TRANSACTION + 0);
static final int TRANSACTION_collapse = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
```

La valeur **FIRST_CALL_TRANSACTION** est par contre une valeur statique définie dans **frameworks/base/core/java/android/os/IBinder.java** ce qui explique les valeurs 1 et 2 pour le code de chaque méthode.

```
int FIRST_CALL_TRANSACTION = 0x00000001;
```

On retrouve le même principe pour le nouveau service dans le fichier **out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/os/ITemperService.java**.

```
static final int TRANSACTION_read = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
```

On peut donc utiliser la commande **service** dans la console ADB et observer le résultat en utilisant la commande **adb logcat** sur le PC de développement. Dans la console ADB, on a :

```
root@generic :/ # service call temper 1
Result: Parcel(00000000 00000000 00000000
'............')
```

Sur le PC de développement, on observe :

```
$ adb logcat | grep temper
I/temperhw_qemu(  275): Temper HW - read()
for 0 bytes called
I/temperhw_qemu(  275): Temper HW - read()
=  0
```

Si l'on supprime le fichier **/dev/temper0**, on obtient :

```
root@generic :/ # rm /dev/temper0
root@generic :/ # service call temper 1
Result: Parcel(Error: 0xffffffb6 "Not a data
message")
```

et sur le PC de développement :

```
I/temperhw_qemu(  275): Temper HW - read() =
�%RA�# 0
I/temperhw_qemu(  275): Failed to open device!
```

Cette petite expérience permet de confirmer que le cheminement des données depuis le pilote jusqu'au service en passant par la HAL fonctionne correctement. Nous allons donc pouvoir envisager la réalisation d'une véritable application.

6 Réalisation d'une application

La réalisation d'une application utilise désormais l'environnement Android Studio **[25]**. Ce dernier a avantageusement remplacé ADT qui était basé sur Eclipse. Android Studio est plus intuitif, plus rapide et permet également de développer pour Android Wear **[26]**, une version d'Android adaptée aux objets connectés (montres, lunettes, etc.). Ce nouvel outil est basé sur le projet libre Gradle **[27]** et le lecteur intéressé peut aller consulter la conférence en ligne du responsable des outils de développement Android **[28]** (Xavier Ducrohet, français donc bon accent pour nous...).

Android Studio est livré avec un SDK standard, mais vu que nous avons réalisé des modifications dans l'API, il est

nécessaire de construire un SDK spécial à partir de nos sources AOSP. La compilation du SDK s'effectue par :

```
$ cd work_43
$ lunch sdk-eng
$ make [-j N] sdk
```

À l'issue de la compilation, le SDK est disponible dans **out/host/linux-x86/sdk**. Après avoir démarré Android Studio, il suffit d'ouvrir le menu ***Files > Other settings > Project Structure*** puis de saisir le chemin d'accès au nouvel SDK dans le champ *Android SDK Location*.

Nous ne détaillerons pas ici les bases de la création d'une application Android sur Android Studio, car la procédure est très classique (***File*** puis ***New Project***, etc.). L'application est très simple graphiquement puisqu'elle comporte un *TextView* pour l'affichage de la température et un simple bouton pour la mise à jour de la valeur.

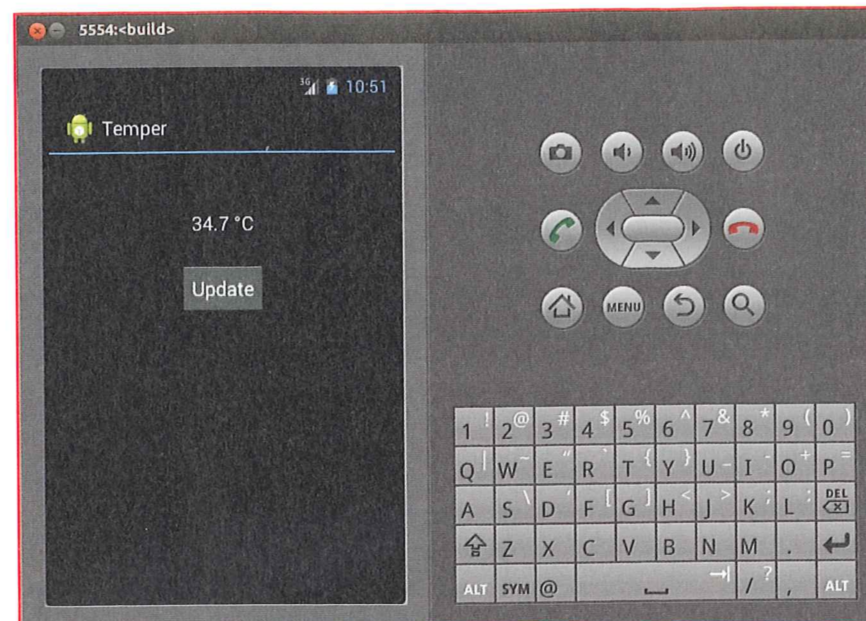


Figure 3 : Application de test.

Le code de l'application est très court, ce qui permet de le reproduire intégralement et de le commenter. Nous commençons par la déclaration du paquet et l'importation des classes du SDK Android, en particulier le nouveau *TemperManager*.

```
package com.temper.hellointernal;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.os.TemperManager;
import android.content.Context;
import android.view.View;
import android.widget.TextView;
```

Nous définissons ensuite l'activité principale. La fonction de lecture de température **update_temp()** utilise une variable de type **TemperManager** pour communiquer avec le service. Selon la documentation du capteur, on obtient la température en ° C en multipliant par 125, puis en divisant par 32000 la valeur obtenue via le pilote.

```
public class HelloTemperInternalActivity extends Activity {
    private static final String DTAG = "TemperActivity";
    private TextView tv;

    TemperManager mTemperManager;

    // Get temp fro JNI interface (C)
    void update_temp()
    {
  Float f = Ø.Øf;
  String s;
  tv = (TextView)this.findViewById(R.id.textView2);
  try {
      f = Float.parseFloat(mTemperManager.read(20));
```

```
      Log.d(DTAG, "Service returned: " + mTemperManager.read(2Ø));
      f = (f * 125) / 32000;
      s = String.format("%.1f °C", f);
      tv.setText(s);
  } catch (Exception e) {
      Log.d(DTAG, "Temper Activity FAILED to read service");
      e.printStackTrace();
  }
    }
```

À la création de l'activité, on alloue **mTemperManager** en utilisant la fonction **getSystemService()** déjà évoquée.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTemperManager = (TemperManager) this.getSystemService(Context.
TEMPER_SERVICE);
    update_temp();
}
```

L'action sur le bouton met à jour la température par un appel à la même fonction **update_temp()**.

```
public void sendMessage(View view){
   // Do something in response to button
   Log.d(DTAG, "onClick update value");
   update_temp();
}
```

Une fois l'application compilée et le paquet produit, on peut l'installer sur l'émulateur en utilisant :

```
$ adb install $HOME/AndroidStudioProjects/HelloTemperInternal/app/
build/outputs/apk/app-debug.apk.
```


Conclusion

L'architecture Android est complexe et la partie système est très peu documentée. Il existe peu d'ouvrages à part les deux articles cités en bibliographie **[1][29]**. Cet exemple pourra donc servir de base pour l'exploitation sous Android d'autres interfaces matérielles fréquemment utilisées dans l'industrie. ■

Remerciements

La version initiale de l'extension de l'API décrite a été réalisée par Michelle Le Grand (michelle.legrand@openwide.fr), stagiaire puis ingénieure de développement chez Open Wide Ingénierie. Merci à elle pour les conseils et informations avisés qui ont conduit à la réalisation de cet article.

Bibliographie

[1] Ouvrage de Benjamin Zores : http://boutique.ed-diamond.com/les-livres/773-android-4-fondements-internes-portage-et-adaptation-du-systeme-android.html

[2] Exemple d'extension d'API par Opersys : http://www.opersys.com/blog/extending-android-hal

[3] Extension d'API pour tâches Xenomai : http://www.linuxembedded.fr/2014/12/investigation-android-temps-reel

[4] Capteur de température TEMPer : http://www.dx.com/fr/p/temper-usb-thermometer-temperature-recorder-for-pc-laptop-81105

[5] Écriture d'un pilote USB pour Linux : http://www.opensourceforu.com/2011/10/usb-drivers-in-linux-1

[6] Ouvrage Linux Device Drivers 3 : http://lwn.net/Kernel/LDD3

[7] Android Open Source Project : http://source.android.com

[8] Interface JNI : http://docs.oracle.com/javase/7/docs/technotes/guides/jni

[9] La HAL Android : https://source.android.com/devices/sensors/hal-interface.html

[10] System Server : http://anatomyofandroid.com/?s=system+server

[11] System Server par K. Yaghmour : http://fr.slideshare.net/opersys/understanding-the-android-system-server

[12] Zygote et System Server : http://androidxref.com/4.3_r2.1/xref/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

[13] Langage AIDL : http://developer.android.com/guide/components/aidl.html

[14] Binder : http://elinux.org/Android_Binder

[15] OpenBinder : http://www.angryredplanet.com/~hackbod/openbinder/docs/html

[16] Classe Context : http://developer.android.com/reference/android/content/Context.html

[17] Mécanisme de Wake Lock : http://developer.android.com/reference/android/os/PowerManager.WakeLock.html

[18] Compilation d'AOSP : https://source.android.com/source/building.html

[19] Compilation du noyau AOSP : https://source.android.com/source/building-kernels.html

[20] Discussion à propos de repo diff : https://groups.google.com/forum/#!topic/repo-discuss/43juvD1qGIQ

[21] Android Debug Bridge (ADB) : http://developer.android.com/tools/help/adb.html

[22] L'outil logcat : http://developer.android.com/tools/help/logcat.html

[23] http://elinux.org/Android_Tools#service

[24] Service Android : http://developer.android.com/reference/android/app/Service.html

[25] Android Studio : http://developer.android.com/sdk/index.html

[26] Android Wear : http://www.android.com/wear/

[27] Projet Gradle : http://gradle.org

[28] Présentation Android Studio par Xavier Ducrohet : https://www.youtube.com/watch?v=LCJAgPkpmR0

[29] Ouvrage Embedded Android de Karim Yaghmour : http://shop.oreilly.com/product/0636920021094.do

[30] Open Wide Ingénierie : http://ingenierie.openwide.fr

[31] Adaptation d'AOSP : http://www.linuxembedded.fr/2014/06/customisation-daosp

[32] AOSP et Fastboot sur carte BBB : http://www.2net.co.uk/tutorial/android-4.3-beaglebone-fastboot

[33] Exemples de l'article : https://github.com/OpenSilicium

[34] Carte BeagleBone Black : http://beagleboard.org/BLACK

[35] Écran tactile pour carte BBB : http://www.4dsystems.com.au/product/4DCAPE_43

[36] Marshalling : http://en.wikipedia.org/wiki/Marshalling_%28computer_science%29

[37] Article « AOSP pour BeagleBone Black » dans Open Silicium n° 14

CROSS-COMPILATION SIMPLE SOUS DEBIAN GNU/LINUX

Denis Bodor

L'installation d'une chaîne de compilation croisée est souvent un problème dont on se passerait bien afin d'entrer le plus rapidement possible dans le vif du sujet qui nous occupe sur le moment. La diversité des solutions, des chaînes, des plateformes et le fait que certains constructeurs reposent sur leurs propres déclinaisons des outils GNU, tout comme certaines distributions ou initiatives comme Linaro, transforment vite une simple installation en chasse à la bonne version. Il n'y a malheureusement pas de solution miracle, mais celle qui va suivre a le mérite de parfaitement s'intégrer à une distribution Debian.

Nous connaissons tous le problème et la situation, il faut le reconnaître, s'est sensiblement améliorée ces derniers temps. Souvenez-vous, il n'y a pas si longtemps certaines chaînes de compilation binaires s'installaient « sauvagement » dans **/usr/local** ou **/usr** via un énorme **setup.sh** et nécessitaient donc des permissions **root** pour cette opération. Aujourd'hui, il est possible de tout simplement copier l'arborescence où bon nous semble et de modifier le **PATH** en conséquence, pour que les différents composants de la chaîne retrouvent leurs petits sans le moindre problème. C'est un mieux, mais ce n'est toujours pas parfait.

Malheureusement, ce qui est de plus en plus courant est de voir la chaîne de compilation GNU modifiée et intégrée dans des suites de développement comme celle de *Mentor Graphics* (ex *CodeSourcery*). Le plus souvent, toujours téléchargeables gratuitement, ces chaînes peuvent être considérées comme des « appeaux à développeurs » puisqu'une inscription est systématiquement nécessaire sur un site avant de pouvoir procéder au téléchargement. Ajoutons à cela que les éditions gratuites semblent maintenant laisser place à des éditions d'évaluation limitées (en fonctionnalités ou dans le temps). C'est le cas par exemple de *Sourcery CodeBench Lite Edition* qui étonnamment (sic) est encore disponible pour les architectures MIPS, AMD64, Nios II, LSI Axxia et Qualcomm Hexagon mais plus pour ARM et Intel. Seule une version *Sourcery CodeBench Professional Edition Evaluation* est disponible, via une demande en bonne et due forme, complétant la chaîne de compilation en logiciel libre par un IDE Eclipse (en logiciel libre) et un outil propriétaire d'analyse et de *profiling* (*Sourcery Analyzer*) incluant une API dédiée.

Il est clair que si votre objectif est de tout simplement cross-compiler un noyau Linux ou les composants d'un système GNU, un tel ensemble n'apporte pas grand-chose si ce n'est une consommation inutile d'espace disque et l'arrivée de messages promotionnels dans votre boîte mail. Au-delà de cet état de fait et des prix affichés pour ces environnements complets, allant de $399 (*Personal Edition*) à « *please contact* » (*Professional Edition*) en passant par $1000 (*Standard Edition*), on peut légitimement se demander si tout ceci est bien normal/moral. Après tout, même s'il y a un réel travail de développement de la part de ces éditeurs, cette manière de faire revient à *packager* des outils propriétaires dans un ensemble d'éléments open source et/ou en logiciels libres (Eclipse, GCC, Qemu, Libc, etc.) et ce dans une proportion discutable, pour ensuite vendre cela comme un tout « optimisé ». Personnellement, je trouve qu'il y a là quelque chose d'impalpablement malsain et divergeant. Bien entendu, le public visé n'est clairement pas le vieux routard Unix/Linux parfaitement à l'aise avec Emacs/Vim, GNU Make et des systèmes comme Yocto/OE ou encore Buildroot, et souhaitant rapidement disposer d'une chaîne de compilation croisée sans avoir à la construire lui-même. Ce type de solutions proches de ce qu'on connaît dans d'autres mondes (type Visual Studio ou XCode) plaît particulièrement aux développeurs souhaitant lancer un environnement de développement comme on lance une suite bureautique et voyant une série d'outils distincts, en ligne de commandes (en plus !), comme quelque chose d'un autre âge en oubliant joyeusement que le noyau et les autres composants d'un système GNU/Linux sont développés justement avec ces outils... (si c'est bon pour Linus, Greg, Alan, Ingo, Russell et David, c'est forcément bon pour nous).

1 Debian et Emdebian

Le but initial du projet Emdebian est de fournir une infrastructure et des outils permettant de faciliter l'utilisation d'une distribution Debian, au sens large du terme, sur un système embarqué. Parmi les éléments notables du projet on trouvait un environnement de construction croisée (*cross-build environment*), un ensemble de paquets standards, mais à faible empreinte disque (*grip*), et tout un lot de documentations dédiées, dont le « *Guide to Embedding Debian* ».

Je parle au passé, car depuis juillet 2014 les mises à jour des distributions Emdebian ont cessé et il a été annoncé qu'aucune autre version stable ne verrait le jour. La raison avancée expliquant cette décision est fort simple : la quantité de plateformes actives et populaires ne disposant pas d'un support de stockage externe (de type SD ou microSD) s'est drastiquement réduite ces dernières années. En effet, il n'y a plus de raison de maintenir des efforts de développement pour créer une sélection des paquets de la distribution optimisée en taille. Ce à quoi s'ajoute le fait que les développements *upstream* de certains paquets spécialement maintenus par Emdebian ont tout simplement également cessé, pour des raisons similaires. Emdebian était un projet qu'on pourrait qualifier de transitoire entre les premières générations de cartes pour l'embarqué et celles actuellement disponibles.

Emdebian présentait pour l'utilisateur Debian « classique » l'avantage de proposer des chaînes de compilation à destination de plusieurs plateformes, dont *armel* (sans FPU) et *armhf* (avec FPU), mais également *ia64*, *m68k*, *mips*, *mipsel*, *powerpc*, *s390* et *sparc*, et ce jusqu'à GCC 4.4.

Bien entendu, avec l'arrêt de la distribution Emdebian, la mise à disposition de ces compilateurs a également cessé, ce qui pose certains problèmes étant donné qu'un certain nombre de SoC cibles ne sont pas supportés par ces anciennes versions et donc qu'il n'est pas possible de compiler un noyau Linux, par exemple, à destination du Sitara d'une BeagleBone Black ou d'un Zynq Xilinx, pour ne citer que ces deux SoC présents pas ailleurs dans ce numéro...

Doit-on alors se tourner vers les chaînes Linaro ou Mentor Graphics ? Pas du tout ! La relève est déjà là et se présente sous la forme d'un mélange subtil entre Debian Multi-arch et outils Emdebian.

2 Emdebian + Multi-arch

Avec l'arrivée des plateformes 64 bits, le besoin s'est fait sentir de réunir sur une même machine des logiciels 32 et 64 bits tout en pouvant gérer les dépendances automatiquement comme sur n'importe quelle architecture précédente. Les développeurs de la distribution Debian sont allés un cran plus loin en mettant en place non seulement un tel système de gestion pour les architectures i386 et amd64, mais également armel, armhf ou encore ia64, mips ou arm64.

Quel intérêt me demanderez-vous d'installer les paquets pour une architecture armhf alors que nos PC x86 ne supportent que i386 et amd64 ? La réponse pourrait tenir dans une seule chaîne de caractères : **"lib*-dev"**. En effet, le système de gestion de dépendances Debian, porté vers un support multi-architectures (Multi-arch), est en mesure de gérer les dépendances de compilation. Installer une bibliothèque dans sa déclinaison armhf ne signifie pas forcément que l'on souhaite exécuter le code qu'elle contient, on peut tout simplement vouloir procéder à une édition de liens... Vous l'avez compris, l'idée est précisément de cross-compiler des éléments à destination d'une plateforme « étrangère » (*foreign*) et, pourquoi pas, construire des paquets à destination de celle-ci.

Mais en y pensant un peu, on se rend également compte que si l'on installe par exemple sur une plateforme i386 des éléments binaires tels des outils et des bibliothèques à destination d'armhf, il n'y a pas moyen de les exécuter (sauf via un émulateur) et donc que l'installation d'une chaîne de compilation ne nous permettra pas de produire des binaires armhf. Installer un GCC binaire armhf sur i386 est totalement inutile. Nous avons donc toujours besoin d'une chaîne de compilation fonctionnelle. Et c'est précisément là qu'intervient Emdebian ou plutôt le dépôt découlant de l'évolution du projet Emdebian : emdebian.org/tools.

En résumé, nous avons besoin d'un compilateur croisé provenant des outils Emdebian et des utilitaires, dépendances, en-têtes et bibliothèques fournis par Debian dans leurs déclinaisons Multi-arch pour l'architecture cible.

3 Installation !

Commençons par la chaîne de compilation croisée. Nous ajoutons un nouveau fichier, appelé **cross.list** par exemple, dans **/etc/apt/sources.list.d/** contenant le dépôt qui nous intéresse :

```
deb http://emdebian.org/tools/debian/ jessie main
```

Remarquez le chemin **tools/debian**, il ne s'agit pas de l'ancien dépôt Emdebian même s'il se trouve sur le même serveur. Notez au passage que si les anciens dépôts en question sont configurés sur votre système (sans le **/tools**), il est certainement de bon ton de les supprimer, ainsi que les paquets qui en proviennent et que vous avez pu installer par le passé (GCC *arm-linux-gnueabi* 4.3 par exemple).

Comme on ajoute un dépôt, un passage par la case « importation de la clé publique GnuPG » s'impose via un `curl http://emdebian.org/tools/debian/emdebian-toolchain-archive.key | sudo apt-key add -`.

Bien entendu, s'en suit un `sudo apt-get update` pour mettre tout cela proprement à jour. À présent, vous pouvez installer un compilateur adéquat pour votre plateforme, mais ce n'est pas fini. Les éléments « annexes » et en particulier les paquets de développement devront également être installés pour l'architecture que vous ciblez. Pour l'heure, et ça le restera, votre plateforme locale est :

```
$ dpkg --print-architecture
i386
```

Ce à quoi nous allons ajouter `armhf` avec :

```
$ sudo dpkg --add-architecture armhf
```

À présent, votre architecture est toujours `i386` (ou `amd64` plus probablement), mais votre système de gestion de paquets est en mesure d'en gérer une autre, une architecture « étrangère » :

```
$ dpkg --print-foreign-architectures
armhf
```

Bien entendu, vous pouvez réitérer avec d'autres architectures comme `armel` par exemple.

Comme vous avez ajouté une ou plusieurs architectures supportées par le système de gestion de dépendances, vous devez à nouveau mettre à jour la liste des paquets avec un `sudo apt-get update`. Notez que dans la liste d'URL utilisées qui défile à l'écran vous devez retrouver le ou les noms des architectures ajoutées.

Il est également probable qu'un avertissement s'affiche si vous avez spécifié d'autres dépôts ne proposant pas nécessairement des paquets binaires pour les architectures en question :

```
$ sudo apt-get update
[...]
Réception de : 123 http://ftp.fr.debian.org
    testing/main i386 2015-03-02-0247.47.pdiff
[1 098 B]
```

```
Réception de : 124 http://ftp.fr.debian.org
$ sudo apt-get update
[...]
Réception de : 123 http://ftp.fr.debian.org
    testing/main i386 2015-03-02-0247.47.pdiff [1 098 B]
Réception de : 124 http://ftp.fr.debian.org
    testing/main i386 2015-03-02-0247.47.pdiff [1 098 B]
218 ko réceptionnés en 1min 56s (1 872 o/s)
W: Impossible de récupérer http://dl.google.com/linux/chrome/deb/dists/stable/Release
  Impossible de trouver l'entrée " main/binary-armhf/Packages " attendue dans le
  fichier " Release " :  ligne non valable dans sources.list ou fichier corrompu
E: Le téléchargement de quelques fichiers d'index a échoué, ils ont été ignorés,
  ou les anciens ont été utilisés à la place.
```

Les dépôts installés automatiquement avec Google Chrome, Google Earth ou Google Talkplugin ont typiquement ce problème puisque les logiciels en question ne sont proposés que pour i386 et amd64 (et que Google n'a pas pris la peine de créer les répertoires adéquats et des fichiers `Packages*` et `Release` vides). Vous ne le saviez peut-être pas parce qu'on ne vous a pas demandé votre avis à l'installation, mais l'intégration des `.deb` de Google ajoute des sources APT dans votre dos.

La correction du problème ne peut être que temporaire et consiste à éditer le fichier présent dans `/etc/apt/sources.list.d/`, par exemple `google-chrome.list`, et à changer la ligne :

```
### THIS FILE IS AUTOMATICALLY CONFIGURED ###
# You may comment out this entry, but any other modifications may be lost.
deb  http://dl.google.com/linux/chrome/deb/ stable main
```

en ajoutant `[arch=amd64,i386]` juste après `deb` et avant l'URL. Ceci aura pour effet de préciser explicitement à APT que ce dépôt ne gère que les architectures i386 et amd64 et donc qu'il ne faut pas tenter de récupérer d'autres fichiers `Packages` pour les architectures `étrangères`. Cependant, comme le précise le commentaire dans le fichier, à la prochaine mise à jour d'un des paquets du dépôt (ou même simplement le lendemain), le fichier `google-chrome.list` sera écrasé. Il ne fait pas partie du contenu du paquet, mais est créé par le script `postinst`. Le même script qui installe automatiquement une clé publique pour le dépôt Google avec `apt-key` sans vous le dire ainsi que le script `/etc/cron.daily/google-chrome`, dont je vous laisse le plaisir de la lecture. Oui, il est possible de simplement supprimer `google-chrome.list`, mais on perd alors la capacité de mise à jour via `apt-get` (ce n'est peut-être pas *evil*, mais c'est clairement intrusif, sinon perfide ! Lisez le script et les commentaires qui s'y trouvent).

Une fois cette éventuelle correction apportée, la commande `apt-get update` n'affiche plus aucun avertissement (pour aujourd'hui). Tout est prêt, il ne vous reste plus qu'à installer l'élément déclencheur :

```
$ sudo apt-get install crossbuild-essential-armhf
[...]
Paramétrage de libc6-i686:i386 (2.19-15) ...
Paramétrage de libstdc++6:armhf (4.9.1-19) ...
Paramétrage de libasan1:armhf (4.9.1-19) ...
Paramétrage de libatomic1:armhf (4.9.1-19) ...
Paramétrage de libgomp1:armhf (4.9.1-19) ...
Paramétrage de libubsan0:armhf (4.9.1-19) ...
Paramétrage de libc-dev-bin (2.19-15) ...
Paramétrage de linux-libc-dev:i386 (3.16.7-ckt4-3) ...
Paramétrage de linux-libc-dev:armhf (3.16.7-ckt4-3) ...
Paramétrage de libc6-dev:armhf (2.19-15) ...
[...]
```

En utilisant ce paquet, vous procédez à l'installation du compilateur croisé pour cette architecture, mais également tous les paquets dépendants pour son fonctionnement. La chaîne de compilation provient effectivement de http://emdebian.org/tools/debian/, mais le reste est pioché dans le dépôt standard (ici http://ftp.fr.debian.org/debian/). Notez que ceux-ci sont installés pour chaque architecture, et qu'il ne faudra pas vous étonner de les retrouver plusieurs fois dans une sortie de `dpkg -l`. Exemple :

```
$ dpkg -l libc6-dev
||/ Nom              Version      Arch[...]
+++-============-===========-========[...]
ii  libc6-dev:armel  2.19-15      armel[...]
ii  libc6-dev:armhf  2.19-15      armhf[...]
ii  libc6-dev:i386   2.19-15       i386[...]
```

Votre environnement de compilation croisée est maintenant prêt à servir :

```
$ gcc --version
gcc (Debian 4.9.1-19) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
[...]

$ arm-linux-gnueabihf-gcc --version
arm-linux-gnueabihf-gcc (Debian 4.9.1-19) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
[...]
```

4 Quelques commandes utiles pour finir

Pourquoi s'arrêter en si bon chemin ? Nous venons d'installer une chaîne de compilation croisée fonctionnelle ainsi que des dépendances locales et des outils de cross-construction. Il peut être utile, en dehors de la compilation frénétique de nouveaux noyaux pour votre toute nouvelle carte, de produire rapidement des binaires empaquetés.

Ceci se fait en quelques commandes et sans le moindre souci. On commence par installer les dépendances de construction exactement comme avec une construction native :

```
$ sudo apt-get build-dep -aarmhf bc
```

Nous prenons ici l'exemple de GNU bc (dont les sources fournissent également la commande `dc`) et les dépendances concernent principalement des outils de construction (Bison) et des *headers* (libreadline, libstdc++, libtinfo5, etc.). Notez l'utilisation de l'option `-a` qui fait toute la différence, car permettant de spécifier pour quelle plateforme nous souhaitons résoudre ces dépendances.

Ensuite, comme pour un paquet classique, nous n'avons qu'à obtenir les sources :

```
$ sudo apt-get source bc
[...]
dpkg-source: info: mise en place de 06_read_dcrc.diff
dpkg-source: info: mise en place de 07_bc_man.diff
```

Les sources sont ainsi téléchargées, décompressées et patchées. Nous retrouvons le résultat adapté à Debian (présence de `debian/`) dans le répertoire `bc`. Il nous suffit alors de nous y placer et de lancer :

```
$ dpkg-buildpackage -uc -us -aarmhf
[...]
dpkg-deb : construction du paquet
  " bc " dans " ../bc_1.06.95-9_armhf.deb ".
dpkg-deb : construction du paquet
  " dc " dans " ../dc_1.06.95-9_armhf.deb ".
[...]
```

Les différentes options utilisées en dehors de `-a` qui précise, là aussi, la plateforme cible, n'ont rien de spécifique. Nous trouvons `-uc` pour ne pas signer le fichier `.changes` et `-us` pour ne pas signer le paquet source. Si vous souhaitez simplement produire un paquet binaire, utilisez classiquement l'option `-b`.

Au terme de la construction, on retrouve effectivement, dans le répertoire parent, les fichiers `bc_1.06.95-9_armhf.deb` et `dc_1.06.95-9_armhf.deb` qu'il nous suffira de copier sur le système Debian cible pour les installer avec `dpkg -i`. Ceci ouvre des perspectives très intéressantes, car à présent il n'y a rien qui vous empêche de packager vos outils ou versions modifiées d'outils existants, et donc de reposer sur une gestion de paquets propre, simple et claire.

Enfin, remarquez que `dpkg-buildpackage`, lors de l'utilisation de l'option `-a` fait appel à l'outil `dpkg-architecture`. Celui-ci pourra, par exemple, être utilisé directement pour lister les architectures utilisables (`-L`) ou encore les variables d'environnement utilisées pour une architecture donnée (`-l` et `-a`). De manière générale, la lecture des pages de manuel pour `dpkg-architecture`, `dpkg-buildpackage`, `dpkg-cross` et `dpkg-architecture` forme la suite logique de cet article (qui se voulait bref à l'origine).

Conclusion

Le sujet du présent article et l'existence de cette solution ont pour moi quelque chose de rassurant et de très réconfortant. C'est là la démonstration qu'il est possible avec la volonté technique adéquate de créer et maintenir un système cohérent reposant sur des bases saines et solides. Il n'est pas nécessaire de se tourner vers une philosophie adaptée d'autres systèmes comme un `setup.sh` hérité des `setup.exe` de Windows ou la copie pure et simple d'applicatifs façon Mac OS X. Les systèmes Unix tirent leur robustesse de choix historiques simples et évidents, et d'un ensemble d'outils matures et fonctionnels. Quitte à paraître radical, je dirai que, exactement comme on ne conduit pas de la même manière un camion et une voiture de sport, on ne développe pas de la même manière avec un système ou un autre. En ne respectant pas cela, vous arrivez certes à destination, peut-être même en étant persuadé d'avoir bien mené les choses et qu'il n'était pas nécessaire d'adhérer à la philosophie originelle du système. Mais après combien de dérapages, d'écarts et de détours ? Bien développer pour un système découle de choix techniques, mais aussi émotionnels. *Nisi credideritis, non intelligetis...* ■

ACME ARIETTA G25 : UN MODULE ARM9 COMPACT ET LOW COST

Denis Bodor

Le nombre de cartes et modules disponibles tentant d'adresser les besoins du plus grand nombre ne cesse d'augmenter. On assiste actuellement à une démultiplication des offres et des plateformes avec ce qui semble être, dans l'esprit des constructeurs, un unique mot d'ordre : intégrer un maximum de fonctionnalités de manière à transformer n'importe quel kit d'évaluation en ordinateur de bureau miniature. Un peu à contre-courant de cette tendance, ACME Systems propose à la vente depuis quelques mois une tout autre vision des choses...

Alors qu'on assiste à une véritable invasion de solutions à base de Raspberry Pi, de ses déclinaisons ou d'autres plateformes comme la BeagleBone Black (BBB), les Cubieboard ou encore la BananaPi, une certaine partie du marché semble presque perdre de vue un point qui nous paraît capital : une sortie vidéo de type HDMI n'est qu'un périphérique et une fonctionnalité comme une autre. À l'instar d'un bus CAN, d'une interface Ethernet ou Sata, l'intégration d'un composant graphique et d'une sortie numérique (HDMI) ou analogique (VGA ou composite) n'est en aucun cas un *must have*. Bien au contraire, ceci implique généralement un SoM intégrant davantage d'IP propriétaires non documentés et par voie de conséquence un support logiciel tout aussi obscur (firmware, pilote, etc.). Cette fonctionnalité n'a pourtant de sens que dans des cas bien spécifiques ou lorsqu'il est question de faire passer une plateforme aux ressources très limitées pour une solution *desktop* (nano-ordinateur) plus ou moins fonctionnelle.

Ainsi, à moins que vous ne soyez en train de concevoir un système d'affichage ou une *appliance* multimédia par exemple, la présence d'une sortie HDMI ou VGA n'a strictement aucun intérêt. Il est donc bien plus raisonnable de ne pas céder aux sirènes de la mode et s'en tenir à la bonne vieille méthode : on définit son cahier des charges et on détermine clairement les fonctionnalités indispensables. Le résultat est souvent très loin de la carte bling-bling à la mode, autant en termes de stabilité que de consommation ou encore de coût.

La carte ou plutôt le module qui nous intéresse ici dispose de caractéristiques qui peuvent paraître modestes au premier abord :

- SoM Atmel AT91SAM9G25 : construit autour d'un ARM926EJ-S à 400 Mhz, ceci fait pâle figure face à des monstres comme le Cortex A8, mais sera plus que raisonnablement suffisant pour bon nombre de projets, sans compter le fait que l'ARM9 est supporté de longue date à la fois par les OS embarqués courants et les chaînes de compilation les plus populaires.
- 128 Mo de mémoire DDR2 qui, là encore, sont loin des 512 Mo d'une BBB ou des 2 Go d'une Cubietruck (alias Cubieboard3). Une quantité aussi énorme que plusieurs Go de mémoire ne peut être synonyme que de deux choses : un projet très spécifique ou la présence d'un ou plusieurs développeurs qui n'auraient jamais dû quitter le monde PC.
- Un emplacement microSD et pas de flash intégrée. Sans doute le seul reproche qu'on puisse faire à ce module. En effet, l'absence de flash, même en eMMC, peut potentiellement poser des problèmes mécaniques pour certaines applications (vibration, chocs, etc.). On pourra éventuellement se rabattre sur une flash SPI externe ou envisager un autre type de stockage, mais l'intérêt de la plateforme sera alors limité.
- Pas de sortie vidéo ou d'Ethernet, mais uniquement un port console série et un emplacement dédié pour un module Wifi USB.
- Sans nul doute le point où la modestie est plus que souhaitable, le prix : 20 euros TTC.

Comme vous pouvez le voir, alors que la tendance est à la course à la puissance, le module Arietta G25 reste raisonnable dans ses caractéristiques. À titre de comparaison, la Cubieboard 8 annoncée en mai dernier, ainsi que le pcDuino8 encore en phase beta, se basent sur le SoC Allwinner A80, 8 cœurs (4 Cortex-A7 + 4 Cortex-A15), un GPU 64-cœurs (PowerVR G6230), interface USB 3.0 et un prix autour des 300 euros ! Mais est-il encore vraiment sensé de parler d'embarqué dans ce genre de situation ?

Au menu des caractéristiques intéressantes du module ACME, en dehors de son prix presque dérisoire, on trouve :

- une taille de 53×25 mm,
- une alimentation 5V via connecteur microUSB ou 3,3V via les broches dédiées,
- une ensemble de 2×20 connecteurs au pas de 2,54mm proposant 3 UART, 2 bus i2c, 1 bus SPI 50 Mhz avec 3 CS, 4 lignes PWM, 4 canaux ADC 10 bits, 3 ports USB (full/hi speed donc un en OTG) et une interface I2S/SSC,
- une plage de température de fonctionnement de -20/+70 ° C,
- une consommation ne dépassant pas les 800mW avec le module Wifi et de l'ordre de 400mW en moyenne pour un usage courant,
- et enfin un ensemble de GPIO (jusqu'à 29 selon la configuration/activation des périphériques du SoC).

Mais la fonctionnalité qui selon moi est plus importante encore est sans conteste un support quasi complet du SoC Atmel dans le noyau Linux 3.x et ce avec des fonctionnalités qu'on pourrait qualifier de modernes comme par exemple l'utilisation du *Device Tree* pour décrire le matériel (cf plus loin dans l'article).

Notons pour terminer cette brève introduction que le fabricant, l'italien ACME Systems, n'en est pas à son coup d'essai et certains lecteurs reconnaîtront quelques noms de ses cartes précédentes comme l'Aria G25, la FOX G20 ou encore la vieille FOX board LX832 ou LX416 à processeur Axis Etrax (ah, souvenirs).

1 Mise en œuvre rapide

Le module Arietta G25 vous arrivera normalement nu et sans microSD à moins bien entendu que vous n'ayez opté pour la solution d'ultra-facilité en commandant une microSD préparée. Il vous faudra donc préparer rapidement une carte de 2 Go ou plus avec les partitions suivantes :

- 1 : FAT16, 32 Mo, kernel : partition pour placer le bootloader et le noyau Linux,
- 2 : ext4, 800 Mo, rootfs : partition root pour le système,
- 3 : ext4, 800 Mo, data : partition de stockage montée en /media/data,
- 4 : SWAP, 128 Mo, swap : partition de swap typiquement utilisée pour l'hibernation (oui, swapper sur de l'embarqué, c'est mal).

Ceci fait avec votre outil préféré (`parted` par exemple), il ne vous restera plus qu'à initialiser les systèmes de fichiers, puis pointer votre navigateur sur http://www.acmesystems.it/download/microsd/Arietta-30aug2014/. Vous trouverez ici deux fichiers :

- `kernel.tar.bz2` (2 Mo) : archive contenant les fichiers `boot.bin` (binaire du bootloader at91bootstrap), `zImage` (image du noyau 3.16) et `acme-arietta.dtb` (le *Device Tree* au format binaire).
- `rootfs.tar.bz2` (134 Mo) : archive regroupant les données du rootfs à désarchiver sur la partition 2 via un petit `sudo tar -xvjpSf`.

Une fois ceci proprement copié sur la microSD qu'on placera dans l'emplacement dédié sur l'Arietta, il suffira de joyeusement la connecter en USB sur un PC. Contrairement au connecteur USB micro-B d'une Raspberry Pi par exemple, celui de l'Arietta est aussi par défaut utilisable et configuré en USB device (OTG) et le SoM se présentera alors au PC comme une nouvelle interface réseau. On en retrouvera ainsi la trace après quelques secondes dans les journaux du système (si le PC est bien entendu sous GNU/Linux) :

```
% dmesg | tail
usb 2-2.4.4: new high-speed USB
   device number 64 using ehci-pci
usb 2-2.4.4: New USB device found,
   idVendor=0525, idProduct=a4a2
usb 2-2.4.4: New USB device strings:
   Mfr=1, Product=2, SerialNumber=0
usb 2-2.4.4: Product: RNDIS/Ethernet Gadget
usb 2-2.4.4: Manufacturer:
   Linux 3.16.1+ with atmel_usba_udc
cdc_eem 2-2.4.4:1.0 usb0: register 'cdc_eem' at
   usb-0000:00:1d.7-2.4.4, CDC EEM Device,
   66:4c:95:af:72:d8
```

Si vous avez lu l'article sur l'USB Gadget dans le présent magazine, vous reconnaissez sans doute à quoi nous avons affaire. Pour vous faciliter les choses, vous

pouvez ajouter une règle udev côté PC sous la forme, par exemple, d'un fichier **/etc/udev/rules.d/99-arietta.rules** :

```
ACTION=="add",SUBSYSTEM=="net",ATTRS{idVendor}=="0525",
ATTRS{idProduct}=="a4a2" NAME="arietta%n"
```

Vous retrouverez alors l'interface réseau dans **/proc/net/dev**, et pour le reste du système, sous le nom **arietta0** (au lieu de **usb0**) qu'il vous suffira de configurer à l'aide d'un petit **sudo ifconfig arietta0 192.168.10.20**. Par défaut, le système sur la carte microSD est configuré de manière à utiliser l'adresse 192.168.10.10. Si cela entre en conflit avec votre adressage IP du LAN, vous pouvez monter la seconde partition de la microSD sur le PC et éditer le fichier **/etc/network/interfaces** afin de changer l'adresse de l'interface **usb0** qui y est décrite.

Un service SSH est lancé par défaut sur le système et vous pouvez donc vous y connecter via l'interface réseau CDC. Un compte utilisateur **acme** existe avec le mot de passe **acmesystems**, ainsi qu'un compte **root** utilisable via SSH avec le même mot de passe. L'accès SSH **root** c'est mal, mais **sudo**, comme sur une Debian PC, n'est pas installé par défaut. Le premier mouvement à opérer consistera donc à retirer l'accès SSH pour le super utilisateur, configurer côté PC un accès internet partagé pour cette connexion et installer/configurer **sudo**.

Mais le plus simple pour fournir une connectivité Internet à l'Arietta est d'utiliser une connexion Wifi. Le module complémentaire vendu par ACME Systems (7 euros + 7 autres pour l'antenne optionnelle selon le modèle) se soude directement sur l'Arietta, mais vous pouvez également opter pour une solution moins compacte sous la forme d'un connecteur USB et d'une clé Wifi compatible Linux.

Une fois le matériel correctement pris en charge (présence du firmware selon le type d'adaptateur Wifi), une interface **wlan0**, par exemple, sera disponible. Il vous suffira alors de configurer WPA supplicant déjà installé par défaut sous la forme du paquet **wpasupplicant**. Créez simplement un fichier de configuration propre à votre point d'accès Wifi local, par exemple **/etc/wpa_supplicant/wpa_supplicant_taf.conf** contenant :

```
network={
  ssid="le_SSID_du_AP"
  psk="phrase2passe"
}
```

Tournez-vous ensuite vers le fichier de configuration du réseau, **/etc/network/interface** pour ajouter une interface :

```
auto wlan0
iface wlan0 inet dhcp
  wpa-conf /etc/wpa_supplicant/wpa_supplicant_taf.conf
```

Une fois la configuration vérifiée et la connectivité en place au démarrage, vous pourrez éventuellement désactiver l'interface **usb0** provenant du module **g_ether** (*Gadget Ethernet over USB*) dans ce même fichier de configuration.

2 C'est bien marrant tout ça, mais c'est quand qu'on compile ?

Utiliser une image ou des archives préparées est bien pratique, mais cela ne présente pas vraiment d'intérêt technique. Le principal avantage de disposer d'une plateforme correctement supportée par le noyau *mainline* consiste justement à personnaliser cet élément et parfaitement l'adapter à ses besoins. Nous ne nous intéresserons pas ici à la partie *userland*. Le système proposé par défaut n'est autre que Emdebian Grip qui dispose d'une vaste collection d'outils et de bibliothèques au format binaire et empaqueté. Notons toutefois qu'il est parfaitement possible et relativement simple de reconstruire tout un rootfs via l'outil Multistrap dont la configuration sera la suivante :

```
[General]
arch=armel
directory=target-rootfs
cleanup=true
noauth=true
unpack=true
debootstrap=Emdebian Net Utils Python
aptsources=Emdebian

[Emdebian]
packages=apt
source=http://www.emdebian.org/grip
keyring=emdebian-archive-keyring
suite=wheezy-grip

[Net]
packages=netbase net-tools ethtool udev iproute iputils-ping ifupdown
isc-dhcp-client ssh
source=http://www.emdebian.org/grip

[Utils]
packages=locales adduser nano less wget vim rsyslog dialog
source=http://www.emdebian.org/grip

[Python]
packages=python python-serial
source=http://www.emdebian.org/grip
```

Remarquez le point important concernant l'architecture : nous sommes en *armel* et non en *armhf* ce qui implique un support *soft float* ARMv4 par opposition à l'utilisation d'un FPU physique ARMv7. Le SoC Atmel est un SAM9G utilisant une architecture ARMv5TEJ, il ne vous sera donc pas possible d'utiliser des binaires compilées en armhf.

Ce qui nous intéresse vraiment est de prendre la main sur le noyau ainsi que le bootloader et c'est justement là que le SoC de l'Arietta prend toute sa valeur, vous allez comprendre...

Nous commençons avant tout par nous assurer de disposer d'une chaîne de compilation. Si vous n'avez pas encore installé de compilateur **arm-linux-gnueabi** et que vous disposez d'une distribution Ubuntu ou Debian, il vous suffira d'ajouter

les dépôts **emdebian** à votre configuration puis installer les paquets adéquats (cf l'article sur l'installation d'une chaîne de compilation dans le présent magazine). Ceci fait, il ne vous restera plus qu'à récupérer les sources du noyau 3.16.1 sur kernel.org (https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.16.1.tar.xz) ainsi qu'un patch fourni par ACME à l'adresse http://www.acmesystems.it/www/compile_linux_3_16/acme.patch.

Et l'agréable surprise est précisément dans ce fichier, car :

```
% grep diff acme.patch
diff --git a/arch/arm/boot/dts/acme-acqua.dts
    b/arch/arm/boot/dts/acme-acqua.dts
diff --git a/arch/arm/configs/acme-acqua_defconfig
    b/arch/arm/configs/acme-acqua_defconfig
diff --git a/arch/arm/boot/dts/acme-aria.dts
    b/arch/arm/boot/dts/acme-aria.dts
diff --git a/arch/arm/boot/dts/at91-ariag25.dts
    b/arch/arm/boot/dts/at91-ariag25.dts
diff --git a/arch/arm/configs/acme-aria_defconfig
    b/arch/arm/configs/acme-aria_defconfig
diff --git a/arch/arm/boot/dts/acme-arietta.dts
    b/arch/arm/boot/dts/acme-arietta.dts
diff --git a/arch/arm/configs/acme-arietta_defconfig
    b/arch/arm/configs/acme-arietta_defconfig
diff --git a/arch/arm/boot/dts/at91-foxg20.dts
    b/arch/arm/boot/dts/at91-foxg20.dts
diff --git a/arch/arm/configs/at91-foxg20_defconfig
    b/arch/arm/configs/at91-foxg20_defconfig
diff --git a/arch/arm/boot/dts/acme_fox.dts
    b/arch/arm/boot/dts/acme_fox.dts
diff --git a/arch/arm/boot/dts/at91-foxg20.dts
    b/arch/arm/boot/dts/at91-foxg20.dts
diff --git a/arch/arm/configs/acme-fox_defconfig
    b/arch/arm/configs/acme-fox_defconfig
diff --git a/arch/arm/configs/at91-foxg20_defconfig
    b/arch/arm/configs/at91-foxg20_defconfig
```

Ce patch ne touche absolument pas au code, mais ne fait qu'ajouter les configurations par défaut (**defconfig**) et les DTS (*Device Tree Source*) pour différentes cartes ACME. Vous l'avez compris, qu'il s'agisse de la FOX G20, de l'Aria G25 ou encore de l'Arietta, c'est bel et bien un noyau *vanilla* qui est utilisé pour ce SoM. Ceci est une conséquence directe de l'implication d'Atmel dans le développement Linux qui, comme quelques autres constructeurs, a une participation réellement active et qui ne se résume pas à la diffusion de quelques documentations techniques au compte-goutte et de code vaguement open source faisant plus office d'interface pour un blob propriétaire que de vrais pilotes dignes du niveau de qualité du code du noyau...

Une fois le **linux-3.16.1.tar.xz** désarchivé, il vous suffira de vous placer dans le répertoire et :

```
% patch -p1 < ../acme.patch
patching file arch/arm/boot/dts/acme-acqua.dts
[...]
patching file arch/arm/configs/at91-foxg20_defconfig
```

```
% make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- acme-arietta_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
[...]
# configuration written to .config
#
```

À ce stade, vous pouvez explorer l'ensemble de la configuration en utilisant un simple **make ARCH=arm menuconfig**. Il ne vous sera cependant pas nécessaire de passer par cette étape afin de supporter les fonctionnalités directement intégrées au SoC, tout est configuré dans le **defconfig** proposé, mais l'activation des périphériques, et donc les utilisations alternatives des broches de l'Arietta, dépend des informations compilées dans le fichier **acme-arietta.dtb** placé en compagnie du bootloader et du noyau sur la première partition de la microSD.

Ce fichier, sur lequel nous allons revenir dans un instant, peut être généré avec :

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- acme-arietta.dtb
scripts/kconfig/conf --silentoldconfig Kconfig
  WRAP    arch/arm/include/generated/asm/auxvec.h
  WRAP    arch/arm/include/generated/asm/bitsperlong.h
[...]
  HOSTCC  scripts/conmakehash
  HOSTCC  scripts/sortextable
[...]
  DTC     arch/arm/boot/dts/acme-arietta.dtb
```

La première invocation de cette commande, tout comme c'est le cas avec un **menuconfig** produira le fichier escompté, mais également, en premier lieu, l'outil **scripts/dtc/dtc** permettant de produire le *Device Tree Blob* (**.dtb**) à partir du *Device Tree Source* (**.dts**), la description de la plateforme matérielle sous une forme interprétable par un être humain (ou compatible).

L'image binaire du noyau pourra ensuite être produite via un **make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage** qu'on assortira de préférence d'une option **-j** suivit du nombre de cœurs à votre disposition sur la machine afin de paralléliser les compilations. Classiquement, on enchaînera sur la compilation des modules avec **make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules**.

Enfin, après avoir monté la partition rootfs du système de l'Arietta, par exemple dans **/mnt/acmeroot**, vous pourrez très simplement installer les modules dans le système avec **make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- INSTALL_MOD_PATH=/mnt/acmeroot modules_install**. La variable **INSTALL_MOD_PATH** désigne le chemin vers la racine du système de destination et la cible **modules_install** s'occupe de l'installation.

Finaliser l'installation reviendra alors à monter la première partition de la microSD (FAT16) et y copier les fichiers **arch/arm/boot/dts/acme-arietta.dtb** et **arch/arm/boot/zImage**.

Pour allez jusqu'au bout des choses concernant la recompilation des éléments de démarrage, il est possible également de produire son propre binaire du bootloader. Le SAM9 démarre sur un code en ROM et selon la configuration du SoC tentera de passer le relais à un bootloader de second niveau, en flash NAND, en flash SPI, en EEPROM TWI ou sur SD. Dans le cas présent, il s'agit bien entendu de cette dernière solution avec un bootloader prenant la forme d'un fichier **boot.bin**.

Le bootloader en question est un fork de celui développé par Atmel : at91bootstrap. ACME et plus exactement Sergio Tanzilli, a donc décliné ce bootloader spécifiquement pour les cartes du fabricant (Acqua, Aria et Arietta). La compilation est relativement simple avec un téléchargement des sources depuis GitHub via un simple **git clone git://github.com/tanzilli/at91bootstrap.git**.

Après s'être placé dans le répertoire **at91bootstrap** il suffira, comme avec le noyau Linux, de choisir le bon **defconfig** :

```
% make arietta-128m_defconfig
```

ou

```
% make arietta-256m_defconfig
```

L'Arietta se déclinant en deux versions (128 ou 256 Mo), il est important de faire la distinction à ce niveau, le bootloader ayant, entre autres choses, la tâche d'initialiser le matériel et donc l'accès à la mémoire DDR2. Notez à ce propos que les explications données ici concernent principalement la version 128 Mo. Ainsi, dans le cas d'une compilation du noyau pour la version 256 Mo, une petite modification sera nécessaire dans le fichier **acme-arietta.dts** pour ne pas se retrouver avec seulement 128 Mo adressables.

3 Device Tree par l'exploration

Bien, à présent, nous avons fait le tour des opérations minimales qu'on se doit de réaliser avec une nouvelle plateforme. Il est temps de passer aux choses sérieuses et en particulier de faire connaissance avec une fonctionnalité intégrée depuis quelque temps aux noyaux récents : le *Device Tree*. Avant l'arrivée de cette fonctionnalité, voici comment se passaient les choses : le noyau contenait la description complète de tout le matériel. Le bootloader avait pour tâche d'initialiser le matériel et le préparer à l'exécution du noyau puis de charger l'image de ce dernier en lui passant une adresse où il pourra trouver des informations minimales (taille de la mémoire, argument de boot, etc.). Sur architecture ARM, deux registres étaient alors utilisés pour cela : **r1** contenant une valeur décrivant le type de machine et **r2** spécifiant l'adresse où trouver les informations passées (ATAGS).

Le problème qui se posa rapidement face à la multiplication des SoC et des cartes fut une prolifération des fichiers sources décrivant chaque plateforme (les *board files*). À chaque arrivée d'une nouvelle carte, même avec un SoC déjà pris en charge, un nouveau fichier faisait son apparition dans les sources. Si vous avez suivi les explications données dans le numéro 11 concernant le portage d'uClinux sur STM32F429, un bon exemple est le fichier **stm32_platform.c** reprenant la même structure que celle que l'on peut trouver dans tous les fichiers **board*.c** présents dans l'arborescence **arch/arm/** d'un noyau de cette version (2.6).

Cette comparaison de versions et de technologies est un bon point de départ pour comprendre tout l'intérêt des fichiers DTS/DTB. Ainsi, dans notre implémentation d'uClinux nous avions pris en charge les deux leds LD3 et LD4 de la plateforme sous la forme d'un fichier **arch/arm/mach-stm32/leds.c**. Nous avons commencé par définir les broches utilisées :

```
static struct gpio_led stm32f429idisco_led_pins[] = {
  {
    .name            = "LD3",
    .gpio            = 109,
    .default_trigger = "none",
  },
  {
    .name            = "LD4",
    .gpio            = 110,
    .default_trigger = "heartbeat",
  },
};
```

Puis nous avons créé une structure qui permet au support du périphérique d'obtenir ces informations :

```
static struct gpio_led_platform_data stm32f429idisco_led_data = {
  .num_leds    = ARRAY_SIZE(stm32f429idisco_led_pins),
  .leds        = stm32f429idisco_led_pins,
};
```

Et nous avons du déclarer notre périphérique :

```
static struct platform_device stm32f429idisco_leds = {
  .name              = "leds-gpio",
  .id                = -1,
  .dev.platform_data = &stm32f429idisco_led_data,
};

static struct platform_device *stm32f429idisco_devices[] __initdata = {
&stm32f429idisco_leds,
};
```

Pour enfin créer une fonction d'initialisation qui testait l'identité de la plateforme avant d'ajouter le périphérique. Fonction qui était alors appelée depuis **stm32_platform.c** si tant est que l'option dans la configuration était bien active (**CONFIG_LEDS_GPIO**).

Le même genre de choses avec une description du matériel sous forme de *Device Tree*, se résume à ceci :


```
leds {
  compatible = "gpio-leds";

  arietta_led {
    label = "arietta_led";
    gpios = <&pioB 8 GPIO_ACTIVE_HIGH>;
    linux,default-trigger = "heartbeat";
  };
};
```

Nous avons ici un *node* ou nœud **leds** décrivant un périphérique. Le premier mot-clé utilisé, **compatible**, permet au noyau de décider quel pilote attacher à ce périphérique. Ici **gpio-leds** est exactement équivalent au **.name="leds-gpio"** du code précédent. Vient ensuite un autre *node* (notez la structure arborescente), **arietta_led**, qui décrit le périphérique avec son label qu'on retrouve en **/sys/class/leds/arietta_led** par exemple, le *trigger* pas défaut, et surtout le port, la broche et la configuration utilisée (ici PB8 actif à l'état haut).

&pioB est une référence à un autre *node*. Il faut remonter tout en haut du fichier pour découvrir :

```
#include "at91sam9g25.dtsi"
```

Notre DTS peut, en effet, inclure des éléments qui sont communs entre cartes et en particulier ceux qui sont propres à un SoC ou une famille de SoC. Et c'est précisément ce que fait le fichier **at91sam9g25.dtsi** :

```
#include "at91sam9x5.dtsi"
#include "at91sam9x5_usart3.dtsi"
#include "at91sam9x5_macb0.dtsi"
```

Et c'est finalement dans **at91sam9x5.dtsi** qu'on retrouve notre *node* :

```
/ {
model = "Atmel AT91SAM9x5 family SoC";
compatible = "atmel,at91sam9x5";
[...]
ahb { /* Advanced High-performance Bus */
[...]
  apb { /* Advanced Peripheral Bus */
[...]
   pinctrl@fffff400 {
    pioB: gpio@fffff600 {
     compatible = "atmel,at91sam9x5-gpio", "atmel,at91rm9200-gpio";
     reg = <0xfffff600 0x200>;
     interrupts = <2 IRQ_TYPE_LEVEL_HIGH 1>;
     #gpio-cells = <2>;
     gpio-controller;
     #gpio-lines = <19>;
     interrupt-controller;
     #interrupt-cells = <2>;
     clocks = <&pioAB_clk>;
   };
```

Ça se complique un peu ici, mais on retrouve la même notion d'arborescence avec le bus AHB et APB typique des architectures ARM, puis un *node* **pinctrl** correspondant au support des ports communs aux plateformes AT94SAM9x5. Et nous avons notre port B, pris en charge par le pilote **at91sam9x5-gpio**.

Remarquez la syntaxe de **compatible** sous la forme d'une liste de chaînes de caractères en duo **"fabricant,modèle"**. La première paire correspond précisément au périphérique que le *node* représente, les suivantes désignent d'autres périphériques avec lesquels celui décrit est compatible.

Le système de *Device Tree* permet d'avoir une structure décrivant les périphériques que le système ne peut pas découvrir. Il ne concerne par exemple pas du tout les périphériques qui peuvent apparaître sur un bus USB. Ceci est parfaitement pris en charge via udev via un tout autre mécanisme. On ne parle ici que des périphériques qui forment le SoC (bus périphériques, CPU, GPIO) ou qui ne peuvent être détectés (périphériques sur un bus SPI ou i2c par exemple).

Le *Device Tree* embarque la liste des périphériques, mais également les données de configuration de ces périphériques. Les structures comme **gpio_led_platform_data** n'ont alors plus d'utilité dans le code et la configuration peut être détachée des sources.

De la même manière, la composition et la structure arborescente du *Device Tree* permettent de grandement faciliter le portage de Linux sous des nouvelles plateformes qui sont, d'une manière ou d'une autre, basées sur celles déjà existantes. Ainsi, la majorité du travail ne repose plus majoritairement sur la composition d'un ensemble de sources C difficiles à maintenir, mais plutôt sur un mécanisme de construction digne d'une boite de Lego. Les périphériques réellement nouveaux qui ne disposent pas encore de pilotes sont alors les seuls à nécessiter un véritable développement, qui sera bien entendu fait avec en tête un souci de compatibilité vis-à-vis du mécanisme de *Device Tree*. Une part importante de pilotes, du moins pour l'architecture ARM est déjà compatible *Device Tree*.

Pour en apprendre davantage sur cette technologie, consultez les liens précisés en fin d'article. Le système est relativement dense et prendra certainement du temps à être assimilé dans son ensemble. Cependant, il ne vous sera pas nécessaire de savoir intégralement composer un fichier DTS pour tirer avantage de cette « nouvelle » manière de voir la description matérielle d'une carte. ■

Liens utiles

- Le générateur DTB/DTS d'ACME pour l'Arietta : http://www.acmesystems.it/pinout_arietta
- Une documentation officielle sur le Device Tree http://devicetree.org/Device_Tree_Usage
- Un article LWN très intéressant sur la transition Platform devices vers device trees : http://lwn.net/Articles/448502/
- Une introduction au Device Tree sur SoC Zynq : http://xillybus.com/tutorials/device-tree-zynq-1

AOSP POUR BEAGLEBONE BLACK

Pierre Ficheux - Directeur technique Open Wide Ingénierie

Dans ce numéro 14, nous évoquons un exemple d'extension de l'API Android afin d'ajouter le support d'un matériel spécifique [4]. Cette extension est par défaut démontrée sur l'émulateur Android, mais peut également fonctionner sur une carte BeagleBone Black (BBB). Même si le support Android de la BBB n'est pas officiel et s'il existe plusieurs projets permettant d'utiliser cette carte dans un tel environnement. Les projets existants utilisent la sortie HDMI de la carte alors que nous désirons utiliser un écran tactile connecté au bus d'extension de la BBB. Dans cet article, nous allons voir comment modifier le support AOSP de la BBB (Jelly Bean 4.3) afin d'utiliser un écran tactile à bas coût conçu par la société australienne 4D Systems.

De par sa popularité grandissante, il existe plusieurs versions d'AOSP pour la carte BBB. Nous pouvons citer :

- le support « officiel » par Texas Instruments lié au processeur Sitara utilisé dans la BBB **[1]** ;
- le projet « Rowboat Android » **[2]** ;
- la version réalisée par la société anglaise 2net **[3]**.

La version de TI n'étant pas trop à jour à l'époque, nous avons longtemps utilisé le projet « Rowboat Android », mais ce dernier utilise une méthode de construction un peu éloignée du standard AOSP. Nous avons récemment découvert les travaux de la société 2net et cette version a l'avantage d'être totalement conforme au standard AOSP. Cependant, cette version utilise la sortie HDMI et l'accélération SGX **[5]** fournie également par TI. Notre but est de modifier le BSP afin d'utiliser Android sur une carte BBB équipée d'un écran tactile même si ce dernier ne supporte pas l'accélération graphique. Une telle configuration est tout à fait acceptable pour réaliser un démonstrateur, un industriel suisse utilise même cette configuration sous GNU/Linux et Qt pour une application réelle de contrôle de machine-outil.

Dans la suite de l'article, nous verrons tout d'abord comment mettre en place un noyau compatible avec l'écran choisi puis nous décrirons les modifications à apporter aux sources AOSP en vue de l'utilisation de l'écran choisi.

1 La carte BeagleBone Black

Nous allons tout d'abord rappeler quelques éléments concernant la BBB qui est souvent présentée comme la principale concurrente de la célèbre Raspberry Pi. La carte est issue de la communauté BeagleBoard.org **[11]** . Suite à la première carte BeagleBoard, une première version BeagleBone (white) a été développée puis une BeagleBone black beaucoup moins onéreuse.

La BBB est équipée d'un cœur Cortex-A8, d'un véritable contrôleur Ethernet et d'un bus d'extension permettant de connecter une carte fille nommée « CAPE » **[6]**. De plus, contrairement à la Raspberry Pi, la BBB utilise une véritable approche « open hardware » puisque tous les schémas sont disponibles **[7]**, mais également les composants permettant de la produire. La Raspberry Pi est quant à elle basée sur un cœur Broadcom totalement propriétaire et il est quasiment impossible de produire la carte de manière autonome. La version B+ n'est pas utilisable pour Android (cœur ARM11 peu puissant) et nous n'avons pas encore eu le loisir de tester la nouvelle Raspberry Pi 2 qui dispose d'un quadri-coeur Cortex-A7 et de 1GO de mémoire (la BBB n'ayant que 512 Mo).

Une vue générale de la BBB et de ses interfaces est disponible sur la figure 1 ci-après. Nous pouvons remarquer les deux connecteurs du bus d'extension

fournissant – entre autres – des interfaces GPIO, I2C, SPI et permettant la connexion des interfaces CAPE.

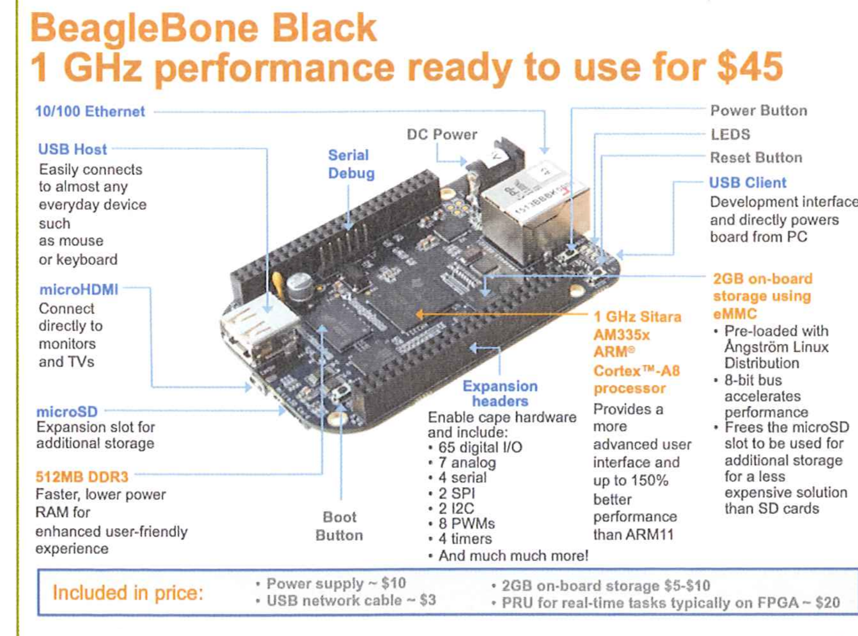


Figure 1 : Les interfaces de la carte BBB.

La figure suivante présente l'avant et l'arrière de l'écran 4DCAPE-43T (tactile) de taille 4,3 pouces et de résolution 480 sur 272 pixels **[8]**. Notons que cet écran peu coûteux (autour de 50 €) utilise une technologie « résistive » et non « capacitive » ce qui dégrade le confort d'utilisation de la partie tactile. Cela reste cependant très acceptable et l'on peut toujours utiliser un stylet pour améliorer les choses. Attention, car il existe une version non tactile (4DCAPE-43) bien moins intéressante, mais que l'on peut malheureusement commander involontairement si l'on se trompe de ligne sur le site (je confirme).

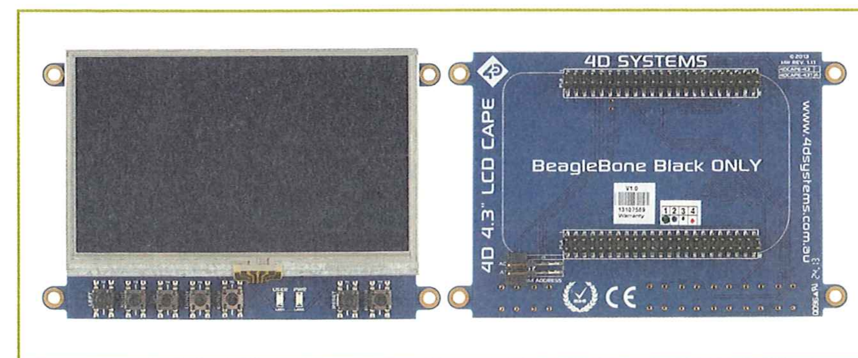


Figure 2 : L'écran tactile 4D 43T.

2 Choix du noyau Linux

Tout comme la Raspberry Pi, la carte BBB fait partie de la catégorie du matériel de « hobbyiste » ou dédié à l'enseignement. De ce fait, n'est pas forcément conseillé d'utiliser ce type de carte pour un projet comportant des contraintes environnementales fortes (température, vibrations, etc.). De plus, la carte n'est pas supportée dans le noyau Linux « mainline », ce qui complexifie quelque peu le choix du support logiciel.

Nous conseillons aux lecteurs d'utiliser le noyau « officiel » de la communauté disponible sur le dépôt GitHub du projet BeagleBoard **[10]**. Dans le cadre de cet article, nous avons choisi la version 3.8.13 qui – contrairement au noyau 3.2 de Rowboat Android – utilise la notion de « device tree » (DT) **[12]**. Le DT est une technique permettant de simplifier l'architecture du noyau Linux dans le cas du support de nombreuses cibles utilisant des architectures proches, mais des périphériques différents. Cette situation est typique de l'architecture ARM pour laquelle il n'existe pas la même compatibilité que pour x86. Le DT est utilisé depuis longtemps pour l'architecture PowerPC, mais son utilisation est plus récente dans le cas de l'ARM. La description matérielle de la carte utilise un fichier `.dts` basé sur un langage dédié. Lors de la compilation, on produit un fichier binaire `.dtb` que l'on doit utiliser avec le noyau statique `zImage`. Le bootloader doit cependant être compatible avec l'utilisation du DT et si ce n'est pas le cas, on devra ajouter le fichier `.dtb` à la fin du fichier `zImage` et activer l'option `CONFIG_ARM_APPENDED_DTB`.

```
$ cat <nom_fichier>.dtb >> zImage
```

Il est important de remarquer que le noyau 3.8.13 utilisé fonctionne à la fois pour GNU/Linux ET pour Android. Les mêmes fichiers binaires (`zImage` et `.dtb`) sont utilisables pour les deux cibles si l'on a pris la peine d'activer le support des pilotes Android fournis dans le répertoire `drivers/staging/android` des sources du noyau.

2.1 Compilation

Dans un premier temps, il convient d'effectuer une copie de l'arbre Git contenant le noyau Linux pour la BBB. Ensuite, nous pourrons nous positionner sur la branche correspondant au noyau utilisé.

```
$ git clone https://github.com/beagleboard/linux.git linux_git
$ cd linux_git
$ git checkout -b android_kernel 3.8.13-bone67
```

Le système Android a pour habitude d'utiliser très peu (ou pas) de modules dynamiques et les pilotes nécessaires sont donc compilés statiquement dans le fichier **zImage**. Nous pouvons compiler le noyau avec un compilateur disponible (Linaro, Sourcery Codebench Lite...). L'environnement AOSP (JB 4.3) fournit un compilateur croisé, mais il semble qu'il ait un problème de compatibilité avec le noyau 3.8.

Le script suivant permet d'initialiser l'environnement pour l'utilisation du compilateur croisé.

```
$ cat ~/bin/set_env_sourcery_2013.sh
#!/bin/sh

ARCH=arm
CROSS_COMPILE=arm-none-linux-gnueabi-
PATH=$PATH:$HOME/arm-2013.05/bin
export ARCH CROSS_COMPILE PATH

$ source ~/bin/set_env_sourcery_2013.sh
```

Une configuration utilisable est déjà fournie dans les sources du noyau par le fichier **arch/arm/configs/bb.org_defconfig** et l'on charge donc la configuration par la ligne :

```
$ make bb.org_defconfig
```

Nous pouvons ensuite invoquer l'utilitaire de configuration par **make menuconfig** afin d'activer les deux options manquantes **CONFIG_ARM_APPENDED_DTB** et **CONFIG_ANDROID_LOW_MEMORY_KILLER**. La nouvelle configuration dans **.config** est sauvée dans le fichier **arch/arm/configs/bb.org_android_fastboot_defconfig**.

On effectuera donc la configuration puis la compilation par les commandes :

```
$ make bb.org_android_fastboot_defconfig
$ make zImage dtbs
```

À l'issue de la compilation, nous obtenons les fichiers **arch/arm/boot/zImage** et **arch/arm/boot/dts/am335x-boneblack.dtb** que nous pourrons intégrer plus tard à l'environnement AOSP.

3 Utilisation de l'environnement AOSP

Comme nous l'avons précisé dans l'introduction, nous utilisons le portage AOSP (JB 4.3) réalisé par la société 2net. Cet environnement a l'avantage d'être très proche de l'environnement AOSP standard **[22]** en particulier grâce à l'utilisation du protocole Fastboot **[13]**. Ce protocole a été défini par Google et permet d'installer très simplement les différentes images Android sur la cible en s'affranchissant des spécificités du matériel (bootloader utilisé, type de mémoire flash, connectivité). Dans le cas de la BBB, la société 2net a réalisé une adaptation du bootloader U-Boot afin de pouvoir utiliser Fastboot **[19]**.

3.1 Mise en place de l'environnement

En premier lieu, on doit récupérer les sources officielles d'AOSP correspondant à la branche 4.3_r2.1. On installe l'outil **repo** puis les sources par la procédure suivante.

```
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/
bin/repo
$ chmod a+x ~/bin/repo
$ mkdir work_43
$ cd work_43
$ repo init -u https://android.googlesource.com/platform/manifest -b
android-4.3_r2.1
$ repo sync
```

Dans le source AOSP, la partie dépendante du matériel est localisée dans le répertoire **device**.

```
$ ls -l
total 36
drwxr-xr-x  7 pierre pierre 4096 août   8  2013 asus
drwxr-xr-x  4 pierre pierre 4096 août   8  2013 common
drwxr-xr-x 10 pierre pierre 4096 août   8  2013 generic
drwxr-xr-x  3 pierre pierre 4096 août   8  2013 google
drwxr-xr-x  4 pierre pierre 4096 août   8  2013 lge
drwxr-xr-x 10 pierre pierre 4096 août   8  2013 sample
drwxr-xr-x  7 pierre pierre 4096 août   8  2013 samsung
drwxr-xr-x  3 pierre pierre 4096 août   8  2013 samsung_slsi
drwxr-xr-x  4 pierre pierre 4096 janv. 10 18:25 ti
```

Par défaut, seule la carte Pandaboard est (ou était) supportée dans le répertoire **ti**.

```
$ ls -l ti/
total 4
drwxr-xr-x 9 pierre pierre 4096 août   8  2013 panda
```

On doit donc ajouter le répertoire **beagleboneblack** contenant les fichiers spécifiques à la carte BBB.

```
$ cd work_43/device/ti
$ git clone https://github.com/csimmonds/bbb-android-device-files.git
beagleboneblack
$ cd beagleboneblack
$ git fetch
$ git checkout jb4.3-fastboot
```

Nous rappelons que la sélection de la cible s'effectue par la séquence suivante à partir du répertoire des sources :

```
$ source build/envsetup.sh
$ lunch beagleboneblack-eng
```

Il suffit alors de construire la distribution Android par :

```
$ make
```

Le répertoire ajouté contient un certain nombre de fichiers dont le script **vendorsetup.sh** qui ajoute l'entrée **beagleboneblack-eng**.

```
$ cat device/ti/beagleboneblack/vendorsetup.sh
add_lunch_combo beagleboneblack-eng
```

Les autres fichiers constituent la configuration spécifique à la BBB. Ce point est très peu documenté par Google, mais l'on peut se référer aux ouvrages cités en **[15]** et **[16]** ainsi qu'à la conférence sur AOSP en **[17]**. La référence **[18]** par l'auteur de cet article décrit également la configuration d'AOSP pour la cible BBB.

```
$ ls
Android.mk          fstab.am335xevm         README.md
AndroidProducts.mk  gpio-keys.kl            sgx
audio_policy.conf   init.am335xevm.rc       ti-tsc.idc
beagleboneblack.mk  init.am335xevm.usb.rc   uEnv.txt
BoardConfig.mk      liblights               ueventd.am335xevm.rc
CleanSpec.mk        media_codecs.xml        vendorsetup.sh
device.mk           media_profiles.xml      vold.fstab
device-sgx.mk       mixer_paths.xml
egl.cfg             overlay
```

Les fichiers **.mk** sont au format GNU-Make et utilisent des macros du type :

```
$(call inherit-product, device/ti/beagleboneblack/device.mk)
```

Ces macros permettent d'« hériter » des paramètres d'un autre fichier **.mk**. En premier lieu, le fichier **AndroidProducts.mk** définit la (ou les) cible(s) définie(s) :

```
PRODUCT_MAKEFILES := $(LOCAL_DIR)/beagleboneblack.mk
```

Le fichier **beagleboneblack.mk** utilise des variables Android afin de définir un certain nombre de paramètres comme le nom de la cible, le constructeur, etc.

```
...
PRODUCT_NAME := beagleboneblack
PRODUCT_DEVICE := beagleboneblack
PRODUCT_BRAND := Android
PRODUCT_MODEL := BEAGLEBONEBLACK
PRODUCT_MANUFACTURER := Texas_Instruments_Inc
```

Le fichier **device.mk** permet de définir d'autres paramètres comme des fichiers ajoutés à l'image AOSP produite, dont le noyau Linux que nous avons compilé précédemment. Ce dernier doit être copié sur le répertoire sous le nom **kernel**.

```
ifeq ($(TARGET_PREBUILT_KERNEL),)
LOCAL_KERNEL := device/ti/beagleboneblack/kernel
else
LOCAL_KERNEL := $(TARGET_PREBUILT_KERNEL)
endif
```

La variable **PRODUCT_COPY_FILE** permet tout simplement de copier des fichiers vers l'image AOSP en utilisant le séparateur « deux points ».

```
PRODUCT_COPY_FILES := \
        $(LOCAL_KERNEL):kernel \
        device/ti/beagleboneblack/init.am335xevm.rc:root/init.am335xevm.rc \
        device/ti/beagleboneblack/init.am335xevm.usb.rc:root/init.am335xevm.usb.rc \
        ...
```

De même, la variable **PRODUCT_PROPERTY_OVERRIDE** permet de définir (ou modifier) une propriété Android.

```
PRODUCT_PROPERTY_OVERRIDES += \
        persist.sys.strictmode.visual=0 \
        persist.sys.strictmode.disable=1
```

La variable **DEVICE_PACKAGE_OVERLAYS** permet de remplacer une arborescence complète par une nouvelle fournie dans la définition de la carte, soit dans notre cas le répertoire **overlay**.

```
DEVICE_PACKAGE_OVERLAYS := \
  device/ti/beagleboneblack/overlay
```

3.2 Modifications apportées

Nous rappelons que notre but est de permettre l'utilisation d'un écran tactile sous Android en remplacement de la sortie HDMI. Cette dernière utilise l'accélération SGX **[5]**. Le fichier **device.mk** traite ce point en vérifiant l'existence du répertoire **sgx/system** qui doit contenir à terme les sources du pilote. Dans notre cas, il ne faut bien entendu pas installer ce pilote.

```
# Include SGX if they exist: they won't on the first build
ifeq ($(wildcard device/ti/beagleboneblack/sgx/system),)
    $(call inherit-product, device/ti/beagleboneblack/device-sgx.mk)
endif
```

En l'absence d'accélération graphique, le système utilise le frame-buffer du noyau Linux comme dans le cas de l'émulateur Android (QEMU). Le fichier **BoardConfig.mk** décrit la configuration matérielle de la cible et en particulier les options utilisées lors du démarrage du noyau. Dans notre cas, nous définissons le nom de la cible à la valeur **am335xevm** et l'option **qemu=1** ce qui force l'utilisation du frame-buffer.

```
# CAPE touchscreen config (qemu=1) => no hw acceleration
```

```
BOARD_KERNEL_CMDLINE := console=tty00,115200n8 androidboot.console=tty00
androidboot.hardware=am335xevm rootwait ro init=/init ip=off qemu=1
vt.global_cursor_default=0
```

Le nom de la machine est important dans le cas d'Android puisque cela permet au processus `init` de démarrer le script d'initialisation `init.am335xevm.rc`.

Le deuxième point concerne le noyau Linux. L'écran tactile ne fonctionne pas avec le noyau 3.2 utilisé par 2net et nous devons donc utiliser la version 3.8.13 décrite précédemment. Comme nous l'avons dit précédemment, il convient d'intégrer la partie « device tree » (soit le fichier `am335x-boneblack.dtb`) et une solution simple est d'ajouter ce fichier à la suite du noyau `zImage` afin de créer le fichier `kernel` utilisé par Android.

```
$ cat arch/arm/boot/zImage  arch/arm/boot/dts/am335x-boneblack.dtb >
<path>/work_43/device/ti/beaglebonblack/kernel
```

Le dernier point concerne les touches Android (Back, Home ...), car l'écran tactile utilisé est trop petit pour pouvoir utiliser les boutons virtuels. Nous allons donc associer les boutons physiques de l'écran LEFT, RIGHT, etc.) à des actions Android. L'action BACK est la plus importante, car elle permet de quitter l'application courante.

```
$ cat gpio_keys_13.kl
# Beaglebone LCD Cape  GPIO KEYPAD keylayout
key 105   DPAD_LEFT        VIRTUAL
key 106   DPAD_RIGHT       VIRTUAL
key 103   DPAD_UP          VIRTUAL
key 108   DPAD_DOWN        VIRTUAL
key 28    BACK             VIRTUAL
```

Les codes des touches ainsi que le nom du périphérique (écran tactile) sont obtenus grâce à la commande `getevent`, ce qui permet ensuite de définir le nom du fichier de « mapping », soit `gpio_keys_13.kl`.

```
# getevent
could not get driver version for /dev/input/mice, Not a typewriter
could not get driver version for /dev/input/mouse0, Not a typewriter
add device 1: /dev/input/event1
  name:     "ti-tsc"
could not get driver version for /dev/input/mouse1, Not a typewriter
add device 2: /dev/input/event3
  name:     "PS/2+USB Mouse"
add device 3: /dev/input/event2
  name:     "gpio_keys.13"
add device 4: /dev/input/event0
  name:     "tps65217_pwr_but"
/dev/input/event2: 0001 0069 00000001
/dev/input/event2: 0000 0000 00000000
/dev/input/event2: 0001 0069 00000000
/dev/input/event2: 0000 0000 00000000
```

4 Installation sur la cible

Comme nous l'avons dit au début de l'article, Android utilise le protocole Fastboot **[13]** pour l'installation sur la cible des images produites. Là encore, la société 2net fournit en **[19]** une documentation détaillée pour l'installation d'une version modifiée de U-Boot supportant Fastboot puis l'installation des images `.img` produites lors de la compilation d'AOSP dans le répertoire `out/target/product/beagleboneblack`. Après avoir formaté la mémoire flash (de type eMMC), on installe le firmware MLO puis l'image U-Boot depuis une carte Micro-SD.

```
$ fastboot oem format
$ fastboot flash spl MLO
$ fastboot flash bootloader u-boot.img
```

Après redémarrage sur la eMMC, on peut alors installer les images Android.

```
$ fastboot flash userdata
$ fastboot flash cache
$ fastboot flashall
```

Le démarrage de la carte à partir d'U-Boot donne les traces suivantes sur la console. On peut remarquer le formatage de la flash réalisé par la commande `fastboot`.

```
U-Boot 2013.01.01-gc0dd2af-dirty (Aug 30 2014 - 20:36:55)

I2C:   ready
DRAM:  512 MiB
WARNING: Caches not enabled
NAND:  No NAND device found!!!
0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
*** Warning - readenv() failed, using default environment

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, bulk combine, bulk split,
HB-ISO Rx, HB-ISO Tx, SoftConn)
musb-hdrc: MHDRC RTL version 2.0
musb-hdrc: setup fifo_mode 4
musb-hdrc: 28/31 max ep, 16384/16384 memory
USB Peripheral mode controller at 47401000 using PIO, IRQ 0
musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, bulk combine, bulk split,
HB-ISO Rx, HB-ISO Tx, SoftConn)
musb-hdrc: MHDRC RTL version 2.0
musb-hdrc: setup fifo_mode 4
musb-hdrc: 28/31 max ep, 16384/16384 memory
USB Host mode controller at 47401800 using PIO, IRQ 0
Net:   <ethaddr> not set. Validating first E-fuse MAC
cpsw
Hit any key to stop autoboot:  0
mmc_send_cmd : timeout: No status update
mmc1(part 0) is current device
mmc_send_cmd : timeout: No status update
Loading efi partition table:
```



```
     256     128K spl
     512     512K bootloader
    1536     128K misc
    2048       8M recovery
   18432       8M boot
   34816     256M system
  559104     256M cache
 1083392     256M userdata
 1607680    1047M media
Loaded eMMC partition table
SELECT MMC DEV: 1
mmc1(part 0) is current device
...
Cleanup before kernel ...

Starting kernel, theKernel ...

Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.8.13 (pierre@XPS-pf) (gcc version 4.7.3
(Sourcery CodeBench Lite 2013.05-24) ) #8 SMP Thu Jan 15 12:47:24 CET 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=50c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing
instruction cache
[    0.000000] Machine: Generic AM33XX (Flattened Device Tree), model: TI
AM335x BeagleBone
...
```

Finalement, on arrive à l'écran d'accueil Android ci-après.

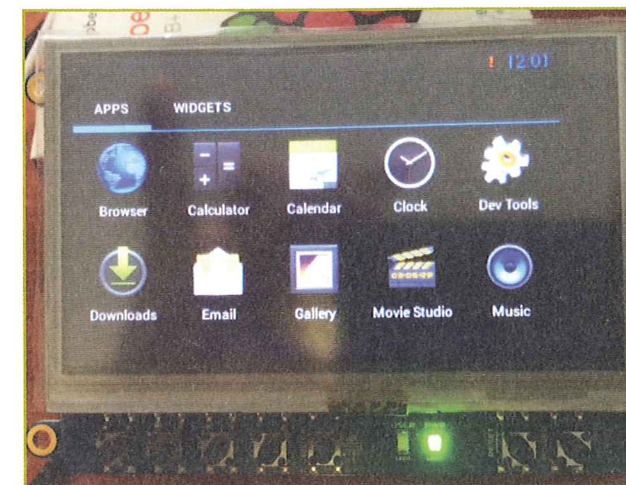


Figure 3 : Android sur l'écran tactile.

Conclusion

Même s'il traite le cas particulier (simple) de la carte BeagleBone Black, ce court article nous a présenté la marche à suivre pour l'adaptation d'AOSP à une nouvelle cible. Cette procédure a l'avantage d'être très proche de celle préconisée pour la compilation d'AOSP sur les cibles Google (NEXUS). L'utilisation d'Android dans le monde industriel montre que les BSP Android adaptés à certaines plateformes exotiques ne sont pas toujours aussi simples à utiliser ! ■

Bibliographie

[1] Android pour TI Sitara : http://www.ti.com/tool/ANDROIDSDK-SITARA

[2] Projet Rowboar Android : https://code.google.com/p/rowboat

[3] AOSP et Fastboot sur carte BBB : http://www.2net.co.uk/tutorial/android-4.3-beaglebone-fastboot

[4] Article « Extension de l'API Android » dans OS#14

[5] Pilotes SGX : http://processors.wiki.ti.com/index.php/Graphics_SDK_Quick_installation_and_user_guide

[6] Les extensions « CAPE » : http://beagleboard.org/cape

[7] Schémas BBB : http://beagleboard.org/hardware/design

[8] Écran 4D 43T de 4D Systems : http://www.4dsystems.com.au/product/4DCAPE_43

[9] Noyau Linux TI : http://processors.wiki.ti.com/index.php?title=Sitara_Linux_SDK_Kernel_Release_Notes&oldid=172302

[10] BeagleBoard sur GitHub : https://github.com/beagleboard

[11] Projet BeagleBoard.org : http://beagleboard.org

[12] Device Tree : http://elinux.org/Device_Tree

[13] Android Fastboot : http://elinux.org/Android_Fastboot

[14] AOSP et Fastboot sur carte BBB : http://www.2net.co.uk/tutorial/android-4.3-beaglebone-fastboot

[15] Ouvrage de Benjamin Zores : http://boutique.ed-diamond.com/les-livres/773-android-4-fondements-internes-portage-et-adaptation-du-systeme-android.html

[16] Ouvrage Embedded Android de Karim Yaghmour : http://shop.oreilly.com/product/0636920021094.do

[17] Conférences Karim Yaghmour à Linaro Connect : http://www.opersys.com/blog/ea-videos-lca-2013

[18] Adaptation d'AOSP à une cible BBB : http://www.linuxembedded.fr/2014/06/customisation-daosp

[19] Installation Fastboot sur BBB : http://www.2net.co.uk/tutorial/fastboot-beaglebone

[20] Carte BeagleBone (White) : http://beagleboard.org/bone

[21] Sources AOSP : http://source.android.com/source

[22] Exemples de l'article : https://github.com/OpenSilicium

ÉLECTRONIQUE ET DOMOTIQUE : PARTIE 5 :

PROGRAMMATION : PREMIERS SIGNES DE VIE

Nathael Pajani

Nous avons vu dans le quatrième article comment fabriquer le module GPIO-Démo (ou toute version modifiée), nous allons désormais nous atteler à lui donner vie. La programmation sera faite en C, bien que d'autres langages puissent être utilisés, à la seule condition qu'il existe un compilateur qui puisse générer du code binaire pour ARM à partir de ce langage...

1 Les documentations techniques

La première étape lorsqu'on veut écrire du code pour un système embarqué qui utilise un microcontrôleur, c'est d'en trouver la documentation technique.

L'accès aux documentations techniques est à mon avis l'un des critères principaux dans le choix des composants pour un projet, aussi bien pour les concepteurs que pour les utilisateurs.

Si vous avez réalisé la conception du circuit, vous aurez normalement déjà eu besoin de ces documentations, dans le cas contraire, c'est le moment de les récupérer. Pour le module GPIO-Démo, il y a plusieurs documentations techniques nécessaires pour utiliser la totalité des fonctionnalités du module. La première et la plus importante est celle du microcontrôleur **[7]**, mais vous aurez aussi besoin de celle du module, disponible sur le site de Techno-Innov **[4]**, et de celles de l'EEPROM I2C, et du capteur de température I2C.

Note : Pour ce qui est du bridge USB-UART, la documentation n'est pas nécessaire pour la programmation du microcontrôleur et du module, mais elle le devient si vous voulez modifier la façon dont le composant s'identifie lorsque vous le connectez, par exemple pour pouvoir lui donner un nom spécifique ou simplifier l'identification. Cela ne sera cependant pas le sujet de cet article.

Pour récupérer les documentations techniques, vous avez le choix entre le site du fabricant et celui du distributeur chez qui vous avez acheté les composants. Pour la majorité des composants, j'utilise le site du distributeur (possible uniquement s'il fournit les bonnes documentations), mais pour les microcontrôleurs je consulte le site du fabriquant, car il fournit aussi les « errata » (notes d'informations contenant les corrections sur la documentation) et le plus souvent des notes d'application qui indiquent comment utiliser le composant pour telle ou telle application (avec parfois des exemples de code).

2 Outils de programmation

La deuxième étape ne concerne toujours pas le code que vous voulez écrire.

En effet, pour écrire du code vous n'avez pas besoin de la « cible » (*target* en anglais) qui est le matériel sur lequel vous allez exécuter la version compilée du code, mais uniquement d'un poste de développement, couramment appelé hôte (*host* en anglais).

Cependant, la cible en question n'a que faire des fichiers source qui se trouvent sur votre poste de développement, et il vous faudra un certain nombre d'outils pour passer des fichiers source (quel que soit le langage) au code exécutable par votre carte électronique.

C'est aussi un point très important à mon sens dans le choix d'un système embarqué ou d'un microcontrôleur : quels sont les outils dont j'aurais besoin pour passer de mon code source à un système fonctionnant avec ?

Dans le cas des microcontrôleurs de la gamme LPC de NXP, il existe de nombreuses solutions pour passer de l'un à l'autre, mais ce qui est intéressant à mon sens c'est qu'il est possible de le faire avec un minimum de matériel et de logiciel : une liaison série TTL 3.3V, une chaîne de compilation croisée et un utilitaire pour « uploader » le code binaire.

2.1 Matériel : un port USB (et le PC qui va avec)

Pour ce qui est de la liaison série TTL 3.3V, il existe plein d'adaptateurs USB-UART fonctionnant en 3.3V, et pour le cas du module GPIO-Démo, cet adaptateur est intégré sur le module, un port USB suffit donc.

L'accès à cette liaison série se fera via un fichier spécial dont le nom devrait ressembler à `/dev/ttyUSB0`, le `USB0` pouvant différer selon votre système (`USB1`, `USB2`... ou même `ACM0` avec d'autres adaptateurs).

2.2 Compilation : GNU

Pour ce qui est de la chaîne de compilation croisée, les microcontrôleurs LPC de NXP utilisent des cœurs ARM Cortex-M, très bien supportés par `gcc`, bien connus de tous les lecteurs de ce magazine (du moins, je le pense), et parfaitement adaptés à la cross-compilation.

Dit comme ça, c'est simple ... mais en fait, pas tant que ça. Cette problématique pourrait faire l'objet d'un (petit ?) article, je ne l'inclurai donc pas ici, je vous donne juste quelques pistes et suppose que vous saurez trouver les informations sur le net. De mon côté, je dois les intégrer à la documentation technique du module GPIO-Démo, mais ce n'est pas encore fait à l'heure où j'écris ces lignes :(**[4]**.

Si vous n'avez pas déjà une chaîne de cross-compilation installée pour ARM, vous avez globalement trois solutions : EmDebian (celle que j'utilise), Launchpad, et Crosstools-ng.

Le projet EmDebian fournit des paquets debian pour différentes chaînes de cross-compilation (ARM parmi tant d'autres), qui sont en train d'être intégrées à Debian. Cela devrait simplifier les problèmes de dépôts et de dépendances dont souffraient les dépôts EmDebian, même si seule la version `arm-none-eabi` (qui nous suffit, nous n'avons pas besoin de libC) est actuellement intégrée dans SID sans problèmes de dépendances (j'ai bien dit « devrait »).

Le site de Launchpad fournit aussi des versions binaires de la chaîne de cross-compilation GCC ARM [*NDLR : ce projet est actuellement, semble-t-il, le plus avancé dans le domaine, en particulier en raison de la participation d'ARM*].

La solution CrossTools-ng : recompilation de sa propre chaîne de compilation croisée : la solution du dernier recours, si votre distribution ne fournit pas de solution packagée et que les versions binaires non signées ne vous conviennent pas.

2.3 Programmation : lpctools (ou autre)

Enfin, pour ce qui est de l'utilitaire permettant de charger (*uploader*) le code binaire sur le microcontrôleur (flasher le microcontrôleur), je n'avais pas trouvé d'outil libre permettant de réaliser cette étape au moment où j'ai étudié la possibilité d'utiliser les microcontrôleurs de NXP, mais le protocole série permettant de réaliser cette opération était documenté dans la documentation technique des microcontrôleurs, il ne devait donc pas y avoir de problème de ce côté-là.

La seule solution « gratuite » que j'avais trouvée à l'époque ne fonctionnait que sur un seul système (pas le mien), ne fournissait pas ses sources, et interdisait une utilisation commerciale sans payer une licence, or je voulais justement en faire une utilisation commerciale.

J'ai donc pris quelques heures de mon temps pour coder un utilitaire permettant de programmer les microcontrôleurs LPC, et placé le tout sous licence GPL v3 **[5]**. `lpctools` est désormais disponible sur le site de Techno-Innov, et est intégré depuis peu à la distribution Debian GNU/Linux (Sid et Jessie à l'heure de l'écriture de ces lignes).

J'ai entre-temps découvert d'autres projets permettant de programmer des microcontrôleurs LPC, mais je ne les ai pas testés (`nxpprog` - licence MIT, `mxli` - GPLv3, pyLPCTools - GPLv2) et constaté qu'un autre paquet Debian fournit les outils permettant de programmer les microcontrôleurs LPC de NXP : `lpc21isp` (qui dispose d'ailleurs d'une interface graphique : GLPC, qui elle n'est pas dans les dépôts). Voir lien **[6]** en fin d'article.

3 Makefile - particularités pour la compilation

Une dernière petite étape avant d'écrire notre code, bien que l'on se rapproche de la programmation, puisqu'il s'agit d'un élément essentiel de tout projet : la création du `Makefile`.

Dans le cas de la compilation pour une cible comme le module GPIO-Démo, le `Makefile` doit inclure quelques éléments supplémentaires que l'on ne retrouve habituellement pas dans un `Makefile` classique.

Le premier élément concerne la définition du compilateur à utiliser. Nous utiliserons la variable **CROSS_COMPILE** à laquelle nous affecterons une valeur par défaut correspondant au préfixe du compilateur. Si vous voulez utiliser un compilateur correctement installé sur le système, il suffit d'utiliser le préfixe, sinon, il faudra ajouter devant la totalité du chemin donnant accès au compilateur.

Par exemple, pour le compilateur EmDebian :

```
CROSS_COMPILE ?= arm-linux-gnueabi-
```

ou avec un chemin complet pour un autre compilateur :

```
CROSS_COMPILE ?= /usr/local/mon/compilateur/bin/arm-none-eabi-
```

Le **?=** permet de ne modifier la variable que si elle n'existe pas, notamment si elle n'est pas déjà présente dans la ligne de commandes.

Cette variable est ensuite utilisée pour modifier la variable **CC** utilisée par **make** pour compiler les fichiers de code source C. Si vous utilisez un autre langage, modifiez la variable correspondante.

```
CC = $(CROSS_COMPILE)gcc
```

Il est aussi possible d'en profiter pour spécifier une version du compilateur installé :

```
CC = $(CROSS_COMPILE)gcc-4.7
```

Nous allons en profiter pour définir la variable **LD** qui correspond à l'éditeur de liens (*linker* en anglais) :

```
LD = $(CROSS_COMPILE)ld
```

Il nous faut ensuite définir la cible pour laquelle nous voulons compiler. Dans notre cas, il s'agit d'un cœur ARM Cortex-M0, qui utilise le jeu d'instructions **thumb** (instructions sur 16bits permettant d'obtenir un code plus compact), nous allons donc utiliser les options **-mcpu** et **-mthumb** de gcc pour lui indiquer notre besoin :

```
CPU = cortex-m0
CFLAGS = -Wall -mthumb -mcpu=$(CPU)
```

En plus de cela, nous allons demander au compilateur de ne pas reconnaître les fonctions pour lesquelles il a une définition interne (*builtin*) tout simplement parce que la majorité de ces fonctions dépendent d'un environnement très différent de celui de notre microcontrôleur, soit par la présence d'une bibliothèque C, d'un système d'exploitation respectant le standard POSIX, ou d'une unité de calcul flottant. Rien de tout ceci n'est vrai dans notre cas, et il est donc préférable que le compilateur nous informe si nous tentons d'utiliser ces fonctions sans l'avoir explicitement demandé en utilisant le préfixe **__builtin__**.

Nous demanderons aussi au compilateur de bien placer les données et les fonctions dans leurs sections respectives. Malgré ce que dit la documentation de GCC, cela permet (au moins dans notre cas) de produire un binaire plus petit.

Cela se fait à l'aide des directives d'optimisation **-ffunction-sections** et **-fdata-sections**.

Nous ajouterons ces directives dans une variable dédiée :

```
FOPTS = -fno-builtin -ffunction-sections -fdata-sections
```

Nous obtenons donc les lignes suivantes à ajouter à votre **Makefile** :

```
CROSS_COMPILE ?= arm-linux-gnueabi-
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)ld
CPU = cortex-m0
FOPTS = -fno-builtin -ffunction-sections -fdata-sections
CFLAGS = -Wall -Wextra -mthumb -mcpu=$(CPU) $(FOPTS)
```

Notre **Makefile** ressemblerait alors à ceci :

```
NAME = mod_gpio
CROSS_COMPILE ?= arm-linux-gnueabi-
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)ld
CPU = cortex-m0
FOPTS = -fno-builtin -ffunction-sections -fdata-sections
CFLAGS = -Wall -Wextra -mthumb -mcpu=$(CPU) $(FOPTS)

.PHONY: all
all: $(NAME)

SRC = $(wildcard *.c)
OBJS = $(SRC:.c=.o)

$(NAME): $(OBJS)
	$(LD) $^ -o $@
```

Ce **Makefile** est encore incomplet et ne permet pas d'obtenir le binaire pour notre microcontrôleur, mais nous ajouterons les parties manquantes plus tard.

En attendant, ce **Makefile** nous permettra de compiler le code que nous allons écrire, passons donc aux parties intéressantes, et le reste du **Makefile** sera bien plus simple à comprendre.

4 Commençons simple

Pour ne pas trop compliquer les choses dès le début, nous allons tenter de faire clignoter la led bicolore présente sur le module GPIO-Démo.

Notre squelette de code ressemblera à s'y méprendre à ce que l'on pourrait avoir pour un programme plus commun :

```
/*
 * Notre exemple simple pour Open Silicium
 */
void system_init(void)
{
```

```
/*
 * Notre exemple simple pour Open Silicium
 */
void system_init(void)
{
	/* System init ? */
}

int main(void)
{
	/* Micro-controller init */
	system_init();

	while (1) {
		/* Change the led state */
		/* Wait some time */
	}
	return 0;
}
```

Si vous compilez ce petit morceau de code avec notre **Makefile**, vous obtiendrez un binaire qui globalement ne fait rien, mais a déjà une taille de 5.7 Ko. C'est gros pour ne rien faire, d'autant que notre Flash ne fait que 32 Ko.

Si on regarde plus loin, par exemple avec l'utilitaire **file**, on obtient quelques informations supplémentaires :

```
$ file mod_gpio
mod_gpio: ELF 32-bit LSB executable,
ARM, EABI5 version 1 (SYSV),
dynamically linked (uses shared libs),
[....], not stripped
```

Arghhh, notre binaire est un exécutable dynamique au format ELF, qui utilise des bibliothèques partagées !

Nous l'avons déjà évoqué, notre microcontrôleur n'a pas de libC ni de système d'exploitation, notre binaire ne peut donc pas être dynamique ! Il nous faut donc faire l'édition de liens en **static**, en apportant quelques modifications à notre **Makefile**. Nous allons introduire une variable **LDFLAGS**, que l'on ajoutera à la règle de compilation.

```
LDFLAGS = -static
$(NAME): $(OBJS)
	$(LD) $^ $(LDFLAGS) -o $@
```

Recompilons notre projet ... 608 Ko !!! Re-Argh !

Visiblement, quelqu'un ajoute du code dont nous n'avons pas besoin. L'option **static** demande à l'éditeur de liens de créer un binaire statique, mais résultat, il nous ajoute tout ce qui est utile de la libC pour un exécutable... pour un système Linux. D'ailleurs, **file** nous dit bien que notre exécutable est au format ELF.

Pour commencer, nous allons demander à l'éditeur de liens (**ld**) de ne pas inclure les éléments de démarrage dans notre binaire, car ils sont inutiles sur notre microcontrôleur. Pour cela, nous ajoutons l'option **-nostartfiles** à nos directives pour l'édition de liens.

À NE PAS MANQUER !

HORS-SÉRIE N°77

J'APPRENDS LA PROGRAMMATION ORIENTÉE OBJET EN 6 JOURS !

COMPATIBLE LINUX / MAC OS X / WINDOWS

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

www.ed-diamond.com

Et hop, une recompilation plus tard, notre binaire ne fait plus que 1 Ko :) Mais pour `file`, ce binaire est toujours au format ELF, et `gcc` nous informe qu'il n'a pas trouvé de symbole d'entrée `_start` et qu'il utilise une adresse par défaut (nous l'avons cherché, cela fait partie de ce que nous avons demandé à l'éditeur de liens en ajoutant l'option `-nostartfiles`).

En effet, le `main()` de nos programmes en C n'est que le point de départ de l'exécution de notre code, mais pas celui du programme. Il est précédé par un certain nombre de routines (fonctions) d'initialisation, dont le point d'entrée est la routine `_start`, qui appellera le main « classique ».

Mais notre but n'est pas de créer un exécutable pour un système Linux, nous devons créer un binaire pour notre microcontrôleur. Il est désormais temps de nous plonger dans la documentation du microcontrôleur pour comprendre comment il démarre, pour pouvoir adapter notre code au démarrage du microcontrôleur et fournir l'équivalent de cette routine `_start`.

5 Bootstrap

5.1 Table des vecteurs d'interruption

Pour avoir des informations sur le démarrage du microcontrôleur LPC1224 dans sa documentation technique **[7]** (`UM10441.pdf`), il y a deux chapitres intéressants.

Nous pouvons nous référer au chapitre 4 (*LPC122x System control*) qui inclut une section sur le reset du microcontrôleur (*4.6 : Reset*), dans laquelle on apprend qu'une fois que la condition de reset disparaît et que les procédures internes se sont exécutées, le processeur commence l'exécution à l'adresse contenue dans le vecteur « *Reset* » du « *boot block* ».

Nous avons ainsi quelques informations, mais qu'est-ce que le vecteur « *Reset* » et le « *boot block* » ? Cela nous mène au deuxième chapitre intéressant, le chapitre 25 sur le cœur ARM Cortex-M0. On y trouve une section sur le processeur (*25.3 : Processor*) avec une sous-section concernant les exceptions (*25.3.3 : Exception model*).

Dans les informations sur les exceptions (25.3.3.2), et plus particulièrement l'exception « *Reset* », il est indiqué qu'après un reset, l'exécution reprend à l'adresse indiquée dans le vecteur « *Reset* » dans la table des vecteurs d'interruption.

Bien, cela confirme ce qui se trouvait dans le chapitre 4. Mais cette fois, nous avons une sous-section « *25.3.3.4 Vector table* », qui décrit cette fameuse table des vecteurs d'exception, précisant qu'elle contient l'adresse de toutes les routines de gestion des exceptions (les fameux « vecteurs » d'exception, dont le vecteur « *Reset* »).

Remarquez au passage que les interruptions sont gérées de la même façon, l'adresse d'entrée des routines de gestion des interruptions se trouve dans cette même table.

Et en dessous de cette table, nous avons une autre information très intéressante : « *The vector table is fixed at address 0x00000000* ».

Si l'on se réfère à la table 65 de la section « *25.3.2 Memory model* », et à la figure 2 de la section « *2.3 Memory allocation* », on découvre que cette adresse est en fait le début de la mémoire flash du microcontrôleur, qui devra contenir notre code binaire.

Reste à créer cette table, et à la placer au bon endroit dans le code binaire compilé...

5.2 Les « handlers » et les « dummy handlers »

Nous allons commencer par la partie code. Ce code peut être placé dans le même fichier que celui où se trouve notre fonction main, ou dans un autre fichier, cela n'a pas d'importance. Si vous vous référez au code du module GPIO-Démo disponible dans les dépôts Git de Techno-Innov, vous trouverez ce code dans le fichier `core/bootstrap.c`.

```
/* Cortex M0 core interrupt handlers */
void Reset_Handler(void);
void NMI_Handler(void) __attribute__ ((weak, alias ("Dummy_Handler")));
void HardFault_Handler(void) __attribute__ ((weak, alias ("Dummy_Handler")));

void Dummy_Handler(void);

void *vector_table[] = {
        0,
        Reset_Handler,
        NMI_Handler,
        HardFault_Handler,
        0,
        /* [......] */
};

void Dummy_Handler(void) {
    while (1);
}

void Reset_Handler(void) {
        /* Our program entry ! */
}
```

Voici un petit morceau de code relativement court qui définit notre table de vecteurs ainsi que les routines de gestion correspondantes.

Avant d'aller plus loin, faisons un petit point sur certains éléments : les attributs utilisés pour la déclaration de deux de nos routines de gestion des exceptions : `__attribute__ ((weak, alias ("Dummy_Handler")))`.

Ces attributs permettent de définir un alias « faible » pour une fonction, ce qui veut dire que si le compilateur ne trouve pas de définition pour cette fonction, il pourra utiliser son alias, que nous avons défini plus bas. Au contraire, si dans un autre fichier (ou celui-ci) vous définissez l'une de ces fonctions, c'est votre définition qui sera utilisée.

Note : je n'ai pas mis la totalité de la table des vecteurs, cela n'a pas d'intérêt pour les explications, et notre programme fonctionnera sans, mais il faudrait la compléter pour un vrai programme, en faisant attention à l'ordre, à partir des informations de la figure 68 déjà évoquée, et de la table 4 du chapitre 3 concernant le gestionnaire d'interruptions (vous noterez la typo, il s'agit des numéros d'IRQ et non pas des numéros des vecteurs d'exception).

5.3 Appeler notre main()

Ce petit bout de code compile donc, puisque tout est défini, mais nous avons toujours l'avertissement concernant le symbole d'entrée `_start` qui n'a pas été trouvé. Nous ne sommes pas plus avancés.

Tout d'abord, nous avons déjà indiqué que le point d'entrée de notre code est la fonction `main()` (qui pourrait avoir n'importe quel nom), mais pour notre microcontrôleur la fonction d'entrée du programme est la routine de gestion de l'exception « *Reset* ».

La solution pour faire le lien est toute simple : il suffit d'appeler la fonction `main()` depuis la routine de gestion de l'exception « *Reset* ».

```
int main(void);

void Reset_Handler(void) {
        /* Our program entry ! */
        main();
}
```

5.4 Tisser des liens, c'est important

Cela ne résout cependant aucun de nos problèmes, nous allons donc devoir donner plus d'indications à l'éditeur de liens, en créant un script pour l'édition de liens (`lpc_link_lpc1224.ld`). Nous donnerons le nom de ce script à l'éditeur de liens en utilisant l'option `-Tlpc_link_lpc1224.ld`.

Ce script d'édition de liens (*linker script* en anglais) est en fait un fichier qui décrit la mémoire de notre microcontrôleur, ainsi que l'organisation que devra respecter notre binaire.

Nous allons donc commencer, toujours à partir des mêmes informations en provenance de la documentation technique, par définir quelle est la mémoire disponible, aussi bien la mémoire vive (SRAM) que la mémoire « morte » (FLASH).

```
MEMORY
{
        sram (rwx) : ORIGIN = 0x10000000, LENGTH = 4k
        flash (rx) : ORIGIN = 0x00000000, LENGTH = 32k
}
```


NE MANQUEZ PAS LE CINQUIÈME NUMÉRO !

L'ÉLECTRONIQUE PLUS QUE JAMAIS À LA PORTÉE DE TOUS !

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR : www.ed-diamond.com

Nous définissons ainsi l'adresse et la taille (et les droits d'accès) de la SRAM et de la flash, qui deviennent deux blocs de mémoire disponibles pour l'éditeur de liens, qui pourra donc y placer du code et des données.

Il nous faut maintenant expliquer à l'éditeur de liens comment organiser le code et les données dans ces blocs de mémoire, et tout particulièrement notre tableau de vecteurs d'interruptions.

Le code d'un programme est composé de sections, dans lesquelles le compilateur place chaque fonction et chaque variable en fonction de différents critères, que l'on peut contrôler soit à partir de directives de compilation, soit d'attributs ajoutés dans notre code C.

Nous allons donc modifier notre tableau de vecteurs d'interruptions pour lui ajouter un attribut **section** qui forcera l'éditeur de liens à le placer dans la section que nous avons définie.

```
void *vector_table[] __attribute__ ((section(".vectors"))) = {
```

Et en parallèle, nous allons définir dans notre script quelles sont les sections que nous voulons trouver dans notre binaire, et dans quel ordre. Commençons simple (si si, je vous assure), nous compliquerons plus tard :

```
SECTIONS {
        . = ORIGIN(flash);
        .text :
        {
                KEEP(*(.vectors))
                *(.text*)
                *(.rodata*)
        } >flash
        .data :
        {
                *(.data*)
                *(.bss*)
        } >sram
}
```

Pour commencer, nous définissons l'adresse courante (**.**) comme étant l'origine du bloc de mémoire **flash** (le bloc de mémoire défini précédemment et auquel nous avons donné ce nom, bien qu'un autre nom aurait pu faire l'affaire). Ceci est important pour que l'éditeur de liens puisse définir les adresses des fonctions pour l'exécution de notre code. À partir de ce point, nous définissons une section **text**, dans laquelle nous allons demander à l'éditeur de liens de mettre plusieurs éléments, à commencer par notre fameuse table de vecteurs, en lui interdisant d'en changer la position ! Après quoi, nous lui demandons de placer l'ensemble du code, que le compilateur a placé dans des sections **.text.***, puis les données en lecture seule (*Read-Only data* = **rodata**), et de placer le tout dans le bloc de mémoire **flash**.

À la suite, nous créons une section **data**, dans laquelle nous lui demandons de placer les données initialisées à des valeurs non nulles (**data**), puis les données initialement nulles (**bss** **[8]**), et de placer cette section en RAM.

Ne reste plus qu'une information à donner à l'éditeur de liens (pour l'instant) : le point d'entrée de notre programme. Ceci est relativement simple et explicite :

```
ENTRY(Reset_Handler)
```

Dans notre *Makefile*, la variable **LDFLAGS** devient :

```
LDFLAGS = -static -nostartfiles -Tlpc_link_lpc1224.ld
```

Une compilation plus tard, sans avertissements cette fois, nous avons un binaire... beaucoup trop gros : notre binaire fait maintenant plus de 66 Ko !

Soyons un peu curieux et allons donc voir ce qu'il contient, mais pas à la main, rassurez-vous, utilisons un utilitaire fait exprès pour cela : **objdump**. attention, il faut bien entendu utiliser la version ARM de **objdump**, donc avec le préfixe **CROSS_COMPILE**, soit dans mon cas **arm-linux-gnueabi-objdump**, avec l'option **--disassemble-all** (**-D**) :

```
$ arm-linux-gnueabi-objdump -D mod_gpio > dump
```

Et là, surprise, la version désassemblée de notre binaire est plus petite que la version binaire (le fichier **dump** fait 2.2 Ko).

Regardons tout de même son contenu. La première ligne nous informe que le format du fichier est **elf32-littlearm**... pas grand-chose à voir avec ce que nous voulions, un « simple » binaire (notre microcontrôleur n'a toujours pas appris à lire les exécutables au format ELF). Premier problème.

Quelqu'un est venu glisser une section **.note.gnu.build-id** avant notre section **.text**, que l'éditeur de liens aimerait placer en début de RAM (**10000000 <_sram_base>**), ce qui n'a pas d'intérêt pour nous.

S'en suit tout de même notre section **.text**, qui commence bien par notre table (enfin ! on nous écoute un peu !), suivie de nos quelques maigres fonctions. Notez les adresses des fonctions dans la table des vecteurs, qui sont toutes impaires. Ceci indique que notre processeur exécutera ces fonctions en mode **thumb** qui est bien le mode que nous avions demandé (**-mthumb**).

Et encore une fois, notre section **.text** n'est pas suivie de nos données, mais d'autres sections que nous n'avions pas demandé ! Certes, nous n'avons pas de variables, et l'éditeur de liens n'ayant rien à mettre dans notre section **.data**, il ne l'a pas créée, mais au final nous sommes relativement loin de ce que nous voulions.

Nous allons donc demander à l'éditeur de liens de générer un petit peu moins de choses en ajoutant l'option **-Wl,--build-id=none** à nos directives de compilation (variable **LDFLAGS** de notre **Makefile**), ce qui nous donne un binaire moitié plus petit, mais toujours trop gros.

5.5 Un binaire vraiment binaire

Mais nous ne pouvons pas en demander plus à l'éditeur de liens, dont le travail est de créer des exécutables au format ELF. Pour créer d'autres types de binaires, nous devrons utiliser l'utilitaire **objcopy** (toujours la version ARM, donc en fait **arm-linux-gnueabi-objcopy**). Le travail de cet utilitaire est de créer des binaires à partir des exécutables ELF, en ne conservant que les sections qui nous intéressent (du moins, c'est l'usage que nous en ferons).

Nous allons demander à cet utilitaire de nous générer une image binaire à partir du résultat de notre compilation en ajoutant dans notre **Makefile** une cible **$(NAME).bin** qui deviendra notre cible par défaut et dépendra de la génération du fichier au format ELF :

```
all: $(NAME).bin

$(NAME).bin: $(NAME)
	$(CROSS_COMPILE)objcopy -O binary $^ $@
```

Et notre objectif est enfin atteint, avec un binaire de ... 36 octets ! (rappelez-vous, je n'ai mis que 5 entrées dans ma table de vecteurs, soit 20 octets, suivis de quelques fonctions vides, tout va bien)

6 Un programme un peu plus utile

Nous allons désormais pouvoir nous occuper d'allumer nos leds, en quelque sorte le « *Hello world* » de l'électronique.

6.1 Entrées et sorties

Du point de vue de notre microcontrôleur, allumer ou éteindre une led revient à changer l'état de la sortie correspondante.

Les entrées/sorties du microcontrôleur peuvent avoir plusieurs fonctions, mais hormis quelques exceptions[1] elles sont par défaut configurées en entrées/sorties (GPIO (*General Purpose Input Output*), en anglais) avec la résistance de *pull-up* interne activée. Pour simplifier, nous laisserons donc cette étape de la configuration de côté pour l'instant, et reviendrons dessus lorsque nous attaquerons la programmation d'interfaces plus complexes.

6.2 Les registres

Pour changer l'état d'une de ces entrées/sorties, il faut commencer par la configurer en sortie, puis définir son état.

Avant d'aller plus loin, il est important de comprendre que le processeur de notre microcontrôleur ne voit que de la mémoire autour de lui. Il n'a pas accès directement aux signaux électriques. Pour accéder aux fonctions spéciales comme l'état électrique d'une sortie, le processeur doit donc modifier des données dans une zone mémoire spécifique, dédiée à cette sortie, que l'on appelle un registre.

Notre microcontrôleur dispose d'un grand nombre de registres qui ont tous une taille de 32 bits, même si souvent certains bits ne sont pas utilisés (ils sont alors « réservés »). Ces registres particuliers, qui nous donnent accès aux fonctions spéciales du microcontrôleur, comme la configuration et l'état des entrées sorties, sont accessibles chacun à une adresse fixe dans l'espace d'adressage du microcontrôleur.

Nous avons déjà eu besoin d'informations sur cet espace d'adressage lorsque nous avons cherché des informations sur le démarrage du microcontrôleur, il se trouve représenté sur la figure 2 dans la section « *2.3 Memory allocation* » du chapitre 2 (*LPC122x Memory map*).

L'adresse qui nous intéresse dépend du port sur lequel les leds sont connectées. La led bicolore étant connectée sur le port 1, ce sont les registres présents à l'adresse 0x50010000 que nous devrons utiliser. Cette adresse est aussi rappelée avec la description des registres dans le chapitre 8 (*LPC122x General Purpose I/O (GPIO)*).

À partir de cette adresse, se trouve une série de registres permettant de contrôler les entrées/sorties du port 1. La modification du contenu de la mémoire accessible ainsi modifiera donc le comportement des entrées/sorties correspondantes ou leur état lorsqu'elles sont configurées en sortie.

Très (trop) souvent, pour accéder à ces registres, les programmeurs utilisent des définitions selon le principe suivant :

```
#define LPC_AHB_BASE  (0x50000000UL)
#define LPC_GPIO_1_BASE  (LPC_AHB_BASE + 0x10000)
#define PORT1_MASK  (LPC_GPIO_1_BASE + 0x00)
#define PORT1_PIN   (LPC_GPIO_1_BASE + 0x04)
#define PORT1_OUT   (LPC_GPIO_1_BASE + 0x08)
/* [.....] */
```

Personnellement, je trouve cette façon de faire très inefficace, et peu lisible (voire complètement illisible), et je préfère l'utilisation de structures [*NDLR : parfois, on a aussi de « jolies » macros comme* **#define GPIO_PORTC_AHB_DATA_R (*((volatile uint32_t *)0x4005A3FC))**]. Cela me semble beaucoup plus lisible, plus adapté pour accéder à de la mémoire qui est structurée, et apporte aussi un très net avantage lorsqu'il existe plusieurs ensembles de registres utilisant la même organisation, par exemple quand il y a plusieurs liaisons séries sur le microcontrôleur, ou dans le cas qui nous intéresse pour l'instant, plusieurs ports d'entrées/sorties. La définition se passe alors ainsi :

```
#define LPC_AHB_BASE  (0x50000000UL)
#define LPC_GPIO_0_BASE  (LPC_AHB_BASE + 0x00000)
#define LPC_GPIO_1_BASE  (LPC_AHB_BASE + 0x10000)
```

[1] Les exceptions sont les GPIO 13, 25 et 26 du port 0 (respectivement Reset, SWDIO et SWDCLK), qui permettent le reset et le debug, ainsi que quatre GPIO sur lesquelles la fonction « 0 » est réservée : les GPIO 30 et 31 du port 0 et les GPIO 0 et 1 du port 1.

```
/* General Purpose Input/Output (GPIO) */
struct lpc_gpio
{
   volatile uint32_t mask;       /* 0x00 : Pin mask, affects data, out, set,
clear and invert */
   volatile uint32_t in;         /* 0x04 : Port data Register (R/-) */
   volatile uint32_t out;        /* 0x08 : Port output Register (R/W) */
   volatile uint32_t set;        /* 0x0C : Port output set Register (-/W) */
   volatile uint32_t clear;      /* 0x10 : Port output clear Register (-/W) */
   volatile uint32_t toggle;     /* 0x14 : Port output invert Register (-/W) */
   uint32_t reserved[2];
   volatile uint32_t data_dir;   /* 0x20 : Data direction Register (R/W) */
   /* [.....] */
};
#define LPC_GPIO_0      ((struct lpc_gpio *) LPC_GPIO_0_BASE)
#define LPC_GPIO_1      ((struct lpc_gpio *) LPC_GPIO_1_BASE)
```

À noter, l'utilisation du mot clé **volatile** qui interdira au compilateur d'optimiser les accès à ces registres en gardant des copies intermédiaires, forçant la lecture ou l'écriture immédiate [*NDLR : ou pire, les « sortir » de la RAM lors de la compilation optimisée parce qu'elles ne semblent jamais utilisées en écriture*].

Cette notation nous donne aussi l'information de la taille des données présentes à ces adresses, dans ce cas là, des valeurs sur 32 bits.

Pour l'utilisation dans le code, c'est ensuite très simple, il suffit de déclarer un pointeur vers cette structure, puis d'accéder au champ qui nous intéresse :

```
struct lpc_gpio* gpio1 = LPC_GPIO_1;
gpio1->out = 0;
```

6.3 Allumer les leds

Reste à déterminer quels sont les champs qui nous intéressent et quelles valeurs nous devons écrire dedans. Pour configurer les pins reliées à nos leds en sortie, nous devons modifier le registre **DIR** (appelé **data_dir** dans la structure), et mettre les bits 4 et 5 à **1** puisque chaque bit de ce registre permet de configurer une des pins du port correspondant, le bit 0 pour la pin 0, le bit 1 pour la pin 1, et ainsi de suite (attention, ici les numéros de pins ne sont pas les numéros des pattes du composant).

Nous pouvons donc modifier notre **main()** pour obtenir le code suivant qui réalise cette opération de façon lisible, et positionne la led verte à l'état « allumée », sans modifier l'état des autres sorties du port 1 :

```
/* The status LED is on GPIO Port 1, pin 4 (PIO1_4) and Port 1, pin
5 (PIO1_5) */
#define LED_RED     5
#define LED_GREEN   4
int main(void)
{
   struct lpc_gpio* gpio1 = LPC_GPIO_1;
   /* Micro-controller init */
   system_init();
   /* Configure the Status Led pins */
   gpio1->data_dir |= (1 << LED_GREEN) | (1 << LED_RED);
   /* Turn Green Led ON */
   gpio1->set = (1 << LED_GREEN);
   gpio1->clear = (1 << LED_RED);
   while (1) {
      /* Change the led state */
      /* Wait some time */
   }
}
```

Remarque **Décalages de bits**

La notation **<<** correspond à l'opérateur logique « décalage de bits vers la gauche » (**>>** pour le décalage à droite). Voir l'article Wikipédia sur le sujet, en anglais, la version française étant catastrophiquement incomplète **[9]**.

7 Mettre le code sur le microcontrôleur : lpctools

Je sais que vous êtes de plus en plus impatients et que vous voulez voir le résultat de tout ceci en vrai, nous allons donc passer à la mise en flash sur le microcontrôleur pour tester tout cela.

Je ne traiterais ici que le cas de la version 48 broches du microcontrôleur LPC1224 (package LQFP48). À vous d'adapter si vous utilisez un autre microcontrôleur, le principe étant très similaire pour tous les LPC de NXP. Pour les autres gammes de microcontrôleurs, référez-vous à leurs docs techniques et aux informations sur Internet.

7.1 Connexion

Si vous avez mis l'adaptateur USB-UART sur votre module comme proposé dans les articles précédents, l'étape de connexion au PC est simple, connectez votre module sur un port USB, directement ou via un câble en fonction du connecteur USB que vous avez choisi.

Sinon, il vous faudra un adaptateur USB-UART (souvent vendu comme adaptateur USB-série) fonctionnant en 3.3V, et relier les signaux Rx et Tx aux signaux TXD0 et RXD0 correspondants à l'UART 0 du microcontrôleur LPC1224 qui se trouve sur le port 0, pins 1 et 2 (broches 16 et 17 du composant). Il est alors préférable d'avoir rendu ces signaux accessibles sur un connecteur de votre choix, sans quoi il vous faudra souder des fils directement sur les pattes du microcontrôleur... possible, mais pas conseillé du tout).

7.2 Mode ISP

Il faut ensuite mettre le microcontrôleur en mode programmation (ISP : *In System Programming* en anglais). Le chapitre 20 (*LPC122x Flash ISP/IAP*) indique qu'il faut maintenir la pin 12 du port 0 (broche 27) à l'état bas (0V) pendant au moins 3ms après avoir relâché le signal « *Reset* » (pin 13 du port 0, broche 28). La suite du chapitre décrit le protocole de programmation, uniquement utile si vous voulez créer votre propre utilitaire pour programmer le microcontrôleur, ou si vous voulez réaliser des opérations très spécifiques.

Puisque nous avons placé des boutons poussoir reliant ces signaux à la masse, avec des résistances de *pull-up*, cette opération est très simple : il faut appuyer sur les deux boutons en même temps, puis relâcher le bouton reset avant de relâcher le bouton ISP. Lorsque tout s'est bien passé, la led bicolore doit avoir deux petits points lumineux à peine visibles dans l'obscurité (un rouge et un vert).

7.3 Dialogue avec le microcontrôleur

La suite se passe sur le PC. Le paquet **lpctools** contient deux binaires : **lpcisp** et **lpcprog**. Le premier donne accès aux opérations élémentaires du protocole de programmation, et ne nous sera pas utile. Le second permet de programmer le microcontrôleur en utilisant une unique commande, bien que d'autres commandes permettent d'effectuer quelques opérations intéressantes.

Nous allons d'ailleurs commencer par une de ces autres opérations : demander les identifiants de notre microcontrôleur avec la commande **id**. La syntaxe des commandes **lpcprog** est simple, il suffit de lui passer un nom de *device*, que l'on passe comme argument de l'option **-d**, une commande (argument de l'option **-c**), et si besoin le nom du fichier à utiliser. Dans l'exemple suivant, le module GPIO-Démo est identifié sur le poste de développement comme périphérique **ttyUSB0**, nous utiliserons donc le *device* **/dev/ttyUSB0**.

```
$ lpcprog -d /dev/ttyUSB0 -c id
Part ID 0x3640c02b found on line 26
Part ID is 0x3640c02b
UID: 0x2c0cf5f5 - 0x4b32430e - 0x02333834 - 0x4d7c501a
Boot code version is 6.1
$
```

La première ligne nous indique que **lpcprog** a reconnu le microcontrôleur et qu'il sera donc possible de le programmer (si ce n'est pas le cas, vous devrez compléter le fichier de description des microcontrôleurs). Ensuite, **lpcprog** nous donne les informations propres au microcontrôleur, à savoir son identifiant unique et la version du *boot code* qui se trouve en ROM sur le microcontrôleur.

Si vous avez un affichage similaire, tout va bien. Sinon, vérifiez la connexion et que le microcontrôleur est bien en mode ISP.

La programmation se fait ensuite très simplement avec la commande **flash**, en ajoutant le nom du fichier binaire à envoyer.

```
$ lpcprog -d /dev/ttyUSB0 -c flash mod_gpio.bin
Part ID 0x3640c02b found on line 26
Flash now all blank.
Checksum check OK
Flash size : 32768, trying to flash 1 blocks of 1024 bytes : 1024
Writing started, 1 blocks of 1024 bytes ...
$
```

lpcprog se charge d'effacer la flash, de générer la somme de contrôle de l'en-tête et de la placer au bon endroit dans le binaire, et d'envoyer le tout au microcontrôleur pour mise en flash.

Pour tester, il suffit d'appuyer sur le bouton Reset, et de constater qu'il ne se passe rien, ce qui n'est pas ce que nous attendions.

Il y a plusieurs problèmes.

7.4 Votre code s'il vous plaît !

Le premier vient de la génération de la somme de contrôle, ou plutôt de sa position dans notre binaire. Tous les LPC de NXP utilisent (à ce jour de ce que j'ai pu voir) le même mécanisme pour déterminer si la flash contient une image valide avant d'essayer d'exécuter son contenu : la vérification du *checksum* des 8 premiers vecteurs d'interruption, qui doit être nulle.

Pour que ceci soit possible, conformément à la documentation technique des microcontrôleurs, **lpcprog** calcule le complément à 2 des 7 premiers vecteurs, et le place dans le huitième.

Sauf que notre tableau ne fait pour l'instant que 5 « cases » et que la huitième case contenait en fait notre code exécutable, qui a été écrasé.

La première modification est donc de remplir au moins 8 cases de ce tableau, au pire avec des valeurs nulles, sinon avec des valeurs correspondant à ce qui est attendu dans la documentation : les adresses des routines de gestion, ou au moins celles de notre routine « de remplacement » (**Dummy_Handler()**) lorsque la documentation indique que l'emplacement est utilisé (table 363 et figure 68 pour les exceptions, et table 4 pour les interruptions).

```
void *vector_table[] __attribute__ ((section(".vectors"))) = {
  0, /* 0 */
  Reset_Handler,
  NMI_Handler,
  HardFault_Handler,
  0,
  /* Entry 7 (8th entry) must contain the 2's complement of the check-sum
     of table entries 0 through 6. This causes the checksum of the first 8
     table entries to be 0 */
 (void *)0xDEADBEEF, /* Actually, this is done using an external tool. */
  0,
  /* [.....] voir le code du module GPIO-Demo pour la suite */
};
```

Cependant, ceci n'est pas suffisant (vous pouvez désormais le tester très simplement, je vous laisse faire).

7.5 Un peu de place pour la pile

Le dernier point un petit peu particulier est encore sur cette fameuse table des vecteurs, et concerne la première entrée. Si vous regardez attentivement la figure 68, le vecteur « Reset » n'est que la deuxième entrée de la table. La première est la valeur initiale du pointeur de la pile (*Stack pointer*), qui correspond en fait à son sommet puisque cette pile est remplie en faisant décroître les adresses (cette information se trouve dans le chapitre 25, mais cette fois je vous laisse chercher un petit peu).

Nous avons initialisé cette valeur à 0, ce qui forcément pose un souci.

Pour obtenir la bonne valeur, nous pourrions coder en dur l'adresse du haut de notre SRAM, mais ce n'est pas propre. Pour que tout soit fait correctement et automatiquement, nous allons modifier le script de l'édition de liens, et demander à ld de remplir cette adresse à notre place, en fonction de l'adresse de la SRAM et de sa taille.

Nous allons donc créer des variables dans notre script, juste après la définition de la mémoire disponible, et chose très importante, ces variables seront des variables externes utilisables dans notre code C !

```
_sram_size = LENGTH(sram);
_sram_base = ORIGIN(sram);
_end_stack = (_sram_base + _sram_size);
```

Nous appelons le bas de la pile (la « fin » de la pile) `_end_stack`, et plaçons cette fin de pile tout en haut de la mémoire vive.

Et pour l'utilisation dans notre code C, tout simplement :

```
extern unsigned int _end_stack;
void *vector_table[] __attribute__ ((section(".vectors"))) = {
  &_end_stack, /* Initial SP value */ /* 0 */
  Reset_Handler,
...
```

Si vous compilez votre code avec ces modifications, vous obtenez enfin un binaire qui allume la led !

Conclusion

Nous avons enfin notre « *Hello World !* » version électronique, mais la route est encore longue.

Vous pourrez voir dans le prochain article que cette première étape n'était qu'une étape, et qu'il reste encore plusieurs étapes importantes avant de pouvoir vraiment commencer à écrire du code « fonctionnel », comme la gestion du *watchdog*, la configuration de l'horloge interne, l'initialisation de la mémoire, et l'écriture des drivers pour chaque bloc fonctionnel (liaison série, I2C, SPI, ADC...).

Rendez-vous donc pour le second article dédié à la programmation de notre microcontrôleur.

Merci pour votre lecture attentive !

Les fichiers créés (`Makefile`, `main.c`, et `lpc_link_lpc1224.ld`) peuvent être récupérés ici : http://techdata.techno-innov.fr/Modules/GPIO_Demo/Code_LM/Article_5/. ■

Liens - Bibliographie

[1] https://gcc.gnu.org/onlinedocs/gcc-4.7.4/gcc/Option-Summary.html

[2] https://sourceware.org/binutils/docs/binutils/objdump.html

[3] https://sourceware.org/binutils/docs/binutils/readelf.html

[4] Page du module et manuel utilisateur : http://www.techno-innov.fr/technique-module-gpio-demo/ - http://techdata.techno-innov.fr/Modules/GPIO_Demo/System_Reference_Manual_Module_GPIO_Demo_v03.pdf

[5] Dépôt GIT lpctools : http://git.techno-innov.fr/?p=lpctools

[6] Lien vers d'autres outils pour programmer les LPC :

- mxli : http://www.windscooting.com/softy/mxli.html
- lpc21isp : http://sourceforge.net/projects/lpc21isp/
- nxpprog : http://sourceforge.net/projects/nxpprog/
- GLPC (GUI pour lpc21isp) : http://sourceforge.net/projects/glpc/

[7] Manuel utilisateur du LPC1224 : http://www.nxp.com/documents/user_manual/UM10441.pdf

[8] http://fr.wikipedia.org/wiki/Segment_BSS

[9] http://en.wikipedia.org/wiki/Bitwise_operation