INTRODUCTION AU « DEVICE TREE » SUR ARM

Thomas PETAZZONI
CTO et ingénieur Linux embarqué, Free Electrons

Depuis plusieurs années, le support de l'architecture ARM dans le noyau Linux est passé progressivement au « Device Tree » pour la description du matériel. Cet article se propose de décrire les motivations derrière ce changement, ainsi que l'utilisation et le fonctionnement du « Device Tree ».

1 Principe de System-on-chip

La plupart des architectures de processeur autres que l'architecture Intel reposent sur le principe de System-on-chip, ou SoC. Ce principe consiste pour les fabricants de processeurs à fournir une unique puce incluant non seulement le processeur, mais également un large ensemble de périphériques (contrôleur graphique, contrôleurs réseau, contrôleur UART, SPI, I2C, CAN, accélérateurs spécialisés, etc.). Ainsi, les concepteurs de systèmes embarqués peuvent réaliser des cartes électroniques comportant un nombre réduit de composants, la majorité des périphériques nécessaires pour le fonctionnement du système étant directement inclus dans le SoC.

Dans le cas de l'architecture ARM, la société ARM Limited ne réalise principalement que la conception du processeur lui-même : ARM926, Cortex-A8 et Cortex-A53 en sont quelques exemples. Les fabricants de SoC choisissent donc un cœur de processeur proposé par ARM, et le combinent avec un ensemble de périphériques, dont le design aura été réalisé soit par leur soin, soit par achat d'IP (bloc matériel) auprès d'autres fournisseurs. Ainsi, ARM Limited peut également proposer d'autres blocs matériels que le processeur lui-même (cache L2, contrôleur d'interruption, contrôleur d'UART) et de nombreuses autres entreprises telles que Synopsys ou Designware proposent de tels blocs matériels aux fabricants de SoC.

Les fabricants de SoC vont choisir le processeur et les périphériques à intégrer dans leur SoC en fonction du marché ciblé par celui-ci : un SoC pour le marché de la téléphonie mobile n'aura ainsi pas du tout les mêmes périphériques qu'un SoC pour des applications industrielles ou pour le marché du stockage personnel (NAS).

Les fabricants de systèmes embarqués vont donc choisir sur le marché le SoC qui correspond le mieux à leurs besoins (en termes de fonctionnalités, de coût, de performance, de consommation

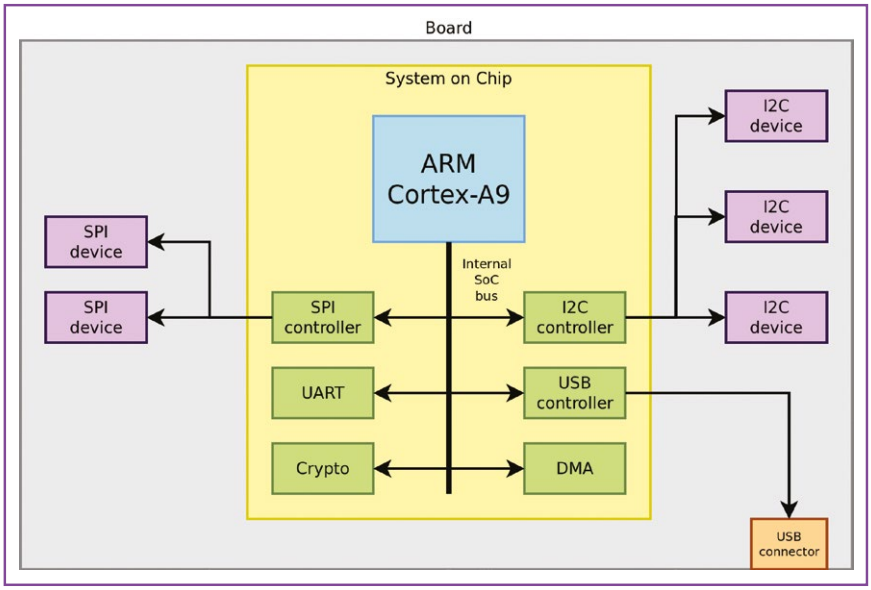

Figure 1 : Représentation schématique d'une carte embarquée à base de SoC ARM. Le SoC ARM (en jaune) se compose d'un processeur ARM (ici le Cortex-A9, en bleu) et d'un certain nombre de périphériques : contrôleur I2C, SPI, UART, USB, accélérateur crypto ou contrôleur DMA (en vert). Ce SoC ARM est monté sur une carte embarquée, qui elle-même intègre d'autres composants : des périphériques sur bus I2C et SPI (en violet) et un connecteur USB (en orange).

énergétique, etc.) et concevoir une carte électronique combinant le SoC choisi avec divers composants additionnels selon les besoins : codec audio, PHY Ethernet, LED, boutons, connecteurs divers, éventuellement d'autres processeurs ou microcontrôleurs, etc. On peut donc résumer le support d'une plateforme ARM en trois niveaux : le support du processeur, le support du SoC et enfin le support de la carte.

Au niveau du support par le noyau Linux, il ne suffit donc pas de supporter « l'architecture ARM », mais il faut également supporter l'ensemble des périphériques d'un SoC et d'une carte pour que celle-ci puisse être pleinement utilisée avec Linux. Ainsi, ce n'est pas parce que le noyau Linux fonctionne sur le processeur ARM I.MX6 de Freescale (qui utilise un processeur Cortex-A9) que le noyau Linux va fonctionner sur l'ensemble des processeurs ARM à base de Cortex-A9 : les périphériques sont souvent très différents d'un SoC à un autre.

De plus et c'est un point crucial pour cet article, les périphériques intégrés dans un SoC ARM et sur une carte ne sont généralement pas « découvrables » de manière dynamique : il n'existe pas de mécanisme d'énumération et d'identification des périphériques comme on peut le trouver sur bus USB ou PCI. Pour cette raison, le système d'exploitation qui tourne sur un SoC ARM doit connaître, à l'avance, la liste et les caractéristiques des périphériques qui sont intégrés : l'adresse des registres, le numéro d'interruption, le canal DMA, la façon de programmer le périphérique, etc.

2 Avant le Device Tree

2.1 Côté noyau Linux

L'essentiel du code pour supporter l'architecture ARM dans le noyau Linux se trouve dans le répertoire **arch/arm/**. Dans ce répertoire, on va tout d'abord trouver le support du processeur lui-même (gestion des interruptions, de la MMU, du cache) principalement dans **arch/arm/kernel** et **arch/arm/mm**. En complément de ce support du processeur, on trouve un grand nombre de répertoires appelés **mach-***, qui contiennent du code supportant des familles de SoC de différents vendeurs. Ainsi, **mach-at91** contient le code spécifique aux processeurs ARM d'Atmel, **mach-mvebu** le code spécifique à la famille EBU de processeurs ARM de Marvell, **mach-socfpga** le code spécifique au processeur SoCFPGA d'Altera.

Avant l'introduction du Device Tree, une quantité très importante de code C dans ces répertoires servait à décrire l'ensemble des périphériques de chaque SoC, puis l'ensemble des périphériques de chaque carte électronique embarquant un SoC. Comme indiqué dans la partie précédente, l'architecture ARM ne fournissant pas de mécanisme d'énumération dynamique, il est nécessaire pour le kernel de connaître à l'avance l'ensemble des périphériques du système et leurs caractéristiques.

Chaque carte électronique était identifiée par un « machine ID » passé par le chargeur de démarrage, et qui permettait au kernel de savoir sur quelle plateforme il était démarré. À partir de cet identifiant, le kernel pouvait ainsi déterminer l'ensemble des périphériques disponibles sur cette plateforme, et les enregistrer dans le noyau afin que les pilotes de ces périphériques puissent être mis en œuvre.

Pour illustrer ce principe, prenons l'exemple de la plateforme « HP Media Vault mv2120 » qui intègre un SoC Marvell Orion 5x, dont le support dans le noyau Linux n'a pas encore été converti au Device Tree (en date du noyau 4.3). Le point d'entrée de cette plateforme se situe dans **arch/arm/mach-orion5x/mv2120-setup.c** :

```
MACHINE_START(MV2120, "HP Media Vault mv2120")
        /* Maintainer: Martin Michlmayr <tbm@cyrius.com> */
        .init_machine   = mv2120_init,
 [...]
MACHINE_END
```

Le « machine ID » est **MV2120**, dont la définition dans **arch/arm/tools/mach-types** nous indique une valeur de **1693**, qui sera celle passée au noyau par le chargeur de démarrage. Si une telle valeur est effectivement passée par le chargeur de démarrage, alors la fonction **mv2120_init()** sera appelée au démarrage du noyau pour initialiser les périphériques de cette plateforme.

```
static void __init mv2120_init(void)
{
 [...]
        orion5x_ehci0_init();
        orion5x_ehci1_init();
        orion5x_eth_init(&mv2120_eth_data);
        orion5x_i2c_init();
        orion5x_sata_init(&mv2120_sata_data);
        orion5x_uart0_init();
        orion5x_xor_init();
 [...]
        platform_device_register(&mv2120_button_device);
 [...]
}
```

Le code commence par enregistrer des périphériques du SoC : contrôleurs USB (EHCI0 et EHCI1), contrôleur Ethernet, I2C, SATA, UART et accélérateur XOR. Puis il se termine en enregistrant un périphérique de la carte électronique (en dehors du SoC) : un bouton.

À titre d'exemple, regardons comment le contrôleur USB EHCI0 est enregistré. La fonction **orion5x_ehci0_init()** étant commune à toutes les plateformes à base de Marvell Orion5x, nous trouvons son implémentation dans **arch/arm/mach-orion5x/common.c** :

```
void __init orion5x_ehci0_init(void)
{
        orion_ehci_init(ORION5X_USB0_PHYS_BASE, IRQ_ORION5X_USB0_CTRL,
                        EHCI_PHY_ORION);
}
```

orion5x_ehci0_init() ne fait que passer l'adresse physique des registres du contrôleur USB, le numéro d'interruption et le type de PHY USB utilisé, à la fonction **orion_ehci_init()**. Cette dernière est commune à plusieurs SoC de Marvell, autres qu'Orion5x, et se trouve donc implémentée dans **arch/arm/plat-orion/common.c** :

```
static struct platform_device orion_ehci = {
        .name           = "orion-ehci",
        .id             = 0,
  [...]
};

void __init orion_ehci_init(unsigned long mapbase,
                            unsigned long irq,
                            enum orion_ehci_phy_ver phy_version)
{
  [...]
        fill_resources(&orion_ehci, orion_ehci_resources, mapbase, SZ_4K - 1,
                       irq);
        platform_device_register(&orion_ehci);
}
```

Cette fonction a pour principal objectif d'enregistrer un **platform_device** : c'est la structure qui, dans le noyau Linux, décrit un périphérique qui est directement attaché à la plateforme (par opposition à un périphérique sur bus USB, PCI, I2C ou SPI par exemple). Le fait d'enregistrer cette structure **platform_device** va déclencher dans le pilote de périphérique correspondant (identifié par le champ **name**, ici **orion-ehci**) l'appel de la méthode **->probe()** qui initialisera le périphérique et l'enregistrera auprès du noyau Linux, dans notre cas, en tant que contrôleur USB. Le pilote de périphérique correspondant est implémenté dans **drivers/usb/host/ehci-orion.c** :

```
static int ehci_orion_drv_probe(struct platform_device *pdev)
{
  [...]
}

static struct platform_driver ehci_orion_driver = {
        .probe          = ehci_orion_drv_probe,
  [...]
        .driver = {
                .name   = "orion-ehci",
        },
};
```

Le champ **.driver.name** de la structure **platform_driver** contenant **ehci-orion**, c'est bien ce pilote de périphérique qui sera utilisé pour tout **platform_device** portant ce nom.

Pour résumer, en l'absence d'un mécanisme comme le Device Tree, le noyau Linux doit décrire, sous la forme de code C, l'ensemble des périphériques de chaque SoC et de chaque carte électronique.

2.2 En pratique, côté bootloader

D'un point de vue pratique, en l'absence de Device Tree, le noyau Linux après compilation est composé d'une image binaire unique : sur ARM, **zImage** ou **uImage**. **zImage** est le format par défaut, tandis que **uImage** est une simple encapsulation d'une **zImage**, avec un entête de 64 octets spécifique au chargeur de démarrage U-Boot.

Du point de vue du chargeur de démarrage, le plus souvent U-Boot sur les plateformes ARM, le lancement d'un kernel consiste à :

1. Charger l'image du noyau en mémoire, que ce soit depuis un périphérique de stockage (MMC, NAND, etc.) ou depuis le réseau (via TFTP) ;
2. Lancer l'exécution du noyau par l'intermédiaire de la commande **bootz** (pour une image **zImage**) ou **bootm** (pour une image **uImage**), qui prend en paramètre l'adresse en RAM où l'image a été chargée à l'étape 1.

Par exemple :

```
U-Boot> tftp 0x200000 zImage
[...]
U-Boot> bootz 0x200000
```

Le chargeur de démarrage ayant été configuré et compilé spécialement pour une plateforme donnée, il sait quel « machine ID » il doit communiquer au noyau, et le lui communique par l'intermédiaire du registre **r1**. Le lecteur intéressé par le détail du protocole de boot pourra se reporter au fichier **Documentation/arm/Booting** dans les sources du noyau Linux.

3 Motivation pour le passage au Device Tree

L'utilisation de code C pour décrire l'ensemble des périphériques des SoCs et des cartes électroniques a conduit à une augmentation continue et significative du code présent dans **arch/arm/**. En complément du code servant strictement à décrire les périphériques, une quantité importante de code était présente dans **arch/arm/** pour gérer des problématiques telles que les *clocks*, le *pin-muxing*, les contrôleurs d'interruption, les *timers*, etc.

En mars 2011, face à la quantité de plus en plus importante de code dans **arch/arm/** pour gérer un nombre

croissant de SoCs et de plateformes, Linus Torvalds déclare, avec sa finesse légendaire : « *Gaah. Guys, this whole ARM thing is a f*cking pain in the ass.* ».

La communauté des développeurs ARM du noyau Linux se met donc à l'œuvre, et va au fil des années suivantes faire très significativement évoluer le support de cette architecture, à plusieurs niveaux :

- L'introduction du Device Tree afin de supprimer le code C servant simplement à décrire les périphériques. Cet aspect est l'objet du présent article.
- L'introduction de sous-systèmes génériques pour gérer les clocks, les timers, les contrôleurs d'interruptions, le pin-muxing, et de nombreuses autres problématiques des SoCs. Par exemple, les drivers de clocks sont maintenant regroupés dans **`drivers/clk/`**, avec une infrastructure générique qui permet de factoriser du code entre les différents drivers.
- La mise en œuvre du principe de kernel « multiplatform », c'est-à-dire la possibilité de compiler une unique image du noyau qui pourra démarrer sur toutes les plateformes ARM supportées par le noyau. Ce principe s'oppose à celui qui prévalait jusqu'à alors et qui consistait à devoir compiler, quasiment pour chaque plateforme, une image différente du noyau.

Le passage au Device Tree s'est donc inscrit dans ce grand mouvement de nettoyage et de refactoring du support de l'architecture ARM dans le noyau Linux.

4 Device Tree : utilisation

4.1 Principe fondamental

Le principe fondamental du Device Tree est de décrire le matériel d'un système et de passer cette description du matériel au système d'exploitation lors du démarrage d'une plateforme. Le principe fondamental et la syntaxe de base du Device Tree sont normalisés par un document intitulé « Power.org Standard for Embedded Power Architecture Platform Requirements » (ou en plus court : *ePAPR*). Comme le nom du document le laisse supposer, ce mécanisme provient à l'origine du monde PowerPC, dans lequel l'OpenFirmware passe au système d'exploitation une description du matériel sous la forme de ce Device Tree. ARM, et d'autres architectures supportées par le noyau Linux, ont simplement repris ce mécanisme disponible sur PowerPC, qui n'était donc en rien nouveau.

Avec le Device Tree, chaque plateforme matérielle se retrouve décrite par un fichier appelé *Device Tree Source* (DTS), qui dans le cas de l'architecture ARM sont aujourd'hui stockés dans le répertoire **`arch/arm/boot/dts/`**. Lors de la compilation du noyau Linux, ces fichiers DTS sont transformés dans un format binaire appelé *Device Tree Blob* (DTB) grâce à un compilateur spécifique appelé *Device Tree Compiler* (DTC). Cette transformation sert principalement à fournir un format plus efficace à parser qu'un format texte. Ainsi, lorsque l'on compile le noyau Linux pour une plateforme ARM, on trouve dans **`arch/arm/boot/dts/`** un ensemble de fichiers **`.dtb`** correspondants aux différentes plateformes utilisant des SoC ARM dont le support aura été activé dans notre configuration du noyau :

```
$ make ARCH=arm mvebu_v7_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf
$ ls arch/arm/boot/dts/
arch/arm/boot/dts/armada-370-db.dtb              arch/arm/boot/dts/armada-385-linksys-cobra.dtb
arch/arm/boot/dts/armada-370-dlink-dns327l.dtb   arch/arm/boot/dts/armada-388-db.dtb
arch/arm/boot/dts/armada-370-mirabox.dtb         arch/arm/boot/dts/armada-388-gp.dtb
arch/arm/boot/dts/armada-370-netgear-rn102.dtb   arch/arm/boot/dts/armada-388-rd.dtb
arch/arm/boot/dts/armada-370-netgear-rn104.dtb   arch/arm/boot/dts/armada-398-db.dtb
arch/arm/boot/dts/armada-370-rd.dtb              arch/arm/boot/dts/armada-xp-axpwifiap.dtb
arch/arm/boot/dts/armada-385-db-ap.dtb           arch/arm/boot/dts/armada-xp-lenovo-ix4-300d.dtb
arch/arm/boot/dts/armada-385-linksys-caiman.dtb  arch/arm/boot/dts/armada-xp-linksys-mamba.dtb
arch/arm/boot/dts/dove-sbc-a510.dtb              arch/arm/boot/dts/dove-d2plug.dtb
[...]
```

Le Device Tree est donc un concept indépendant du système d'exploitation : un Device Tree décrit une plateforme matérielle, et doit donc théoriquement être utilisable par Linux, par un bootloader, par FreeBSD ou n'importe quel autre système d'exploitation.

De plus, le Device Tree ne doit théoriquement que représenter le matériel, et non pas une configuration spécifique du matériel liée à un OS particulier : par exemple, le Device Tree peut être utile pour décrire si un périphérique supporte ou non une fonctionnalité donnée, mais pas pour décrire si l'utilisateur choisit d'utiliser cette fonctionnalité ou non, car il ne s'agit alors plus d'une description du matériel, mais bien d'un choix de configuration.

Enfin, le Device Tree n'a vocation qu'à décrire le matériel qui ne peut pas être découvert ou énuméré dynamiquement : ainsi les périphériques USB ou PCI ne sont typiquement pas décrits dans le Device Tree. Il convient donc dans cette discussion de ne pas confondre le contrôleur USB ou PCI du périphérique USB ou PCI : le contrôleur USB ou PCI fait le plus souvent partie du SoC, n'est donc pas découvrable dynamiquement et doit être décrit dans le Device Tree, alors que les périphériques qui seront connectés aux bus USB ou PCI offerts par ces contrôleurs n'auront pas besoin d'être décrits dans le Device Tree.

4.2 Démarrage avec U-Boot

Pour démarrer un noyau Linux utilisant le Device Tree pour décrire une plateforme matérielle, il faudra donc au niveau du chargeur de démarrage charger deux fichiers : l'image du noyau Linux et le Device Tree Blob. Notons au passage que l'introduction du mécanisme de noyau « multiplatform » a rendu obsolète le format d'image **uImage**, et c'est donc désormais le format **zImage** qui est préféré. En effet, le format **uImage** servait essentiellement à encoder dans un entête l'adresse physique fixe à laquelle le noyau devait absolument être chargé pour s'exécuter. Avec le passage au multiplatform, cette information n'est plus nécessaire puisque le noyau a été rendu exécutable indépendamment de son adresse physique de chargement.

La commande U-Boot **bootz**, que nous avions utilisée plus tôt avec un seul argument, prend en réalité trois arguments :

- l'adresse de l'image du noyau ;

- l'adresse (optionnelle) d'un initramfs ;

- l'adresse (optionnelle) d'un Device Tree Blob.

Par conséquent, une séquence typique de commandes pour charger et démarrer un noyau Linux reposant sur le Device Tree serait :

```
U-Boot> tftp 0x2100000 zImage
[...]
U-Boot> tftp 0x2000000 armada-385-linksys-cobra.dtb
[...]
U-Boot> bootz 0x2100000 -  0x2000000
```

Notez que le « **-** » en deuxième argument est nécessaire, pour indiquer explicitement à **bootz** que nous n'utilisons pas d'initramfs. Si ce paramètre est omis, bootz considérerait alors que **0x2000000** est l'adresse de l'initramfs, et que nous n'utilisons pas de Device Tree. La notion de « machine ID » n'est plus utilisée, et le chargeur de démarrage passe l'adresse du Device Tree Blob au noyau par l'intermédiaire du registre **r2**.

Notons que le chargeur de démarrage est autorisé à adapter le DTB avant de le passer au kernel, par exemple pour lui passer des informations comme la ligne de commandes du noyau ou l'adresse MAC des différentes interfaces réseau.

4.3 Mode de compatibilité

Ce mécanisme de démarrage par Device Tree requiert donc un support au niveau du chargeur de démarrage. Bien que toutes les versions récentes d'U-Boot proposent ce support, il n'en est pas de même pour des versions plus anciennes, desquelles il est parfois malheureusement difficile de s'affranchir sur certaines plateformes matérielles peu récentes. Le noyau Linux propose donc un mode de compatibilité qui permet de démarrer un noyau utilisant un DTB sans avoir besoin de prise en charge spécifique de la part du chargeur de démarrage.

Ce mode de compatibilité est activable par l'option **CONFIG_APPENDED_DTB** du noyau Linux, qui comme son nom l'indique, consiste simplement à concaténer le Device Tree Blob à l'image du noyau. À l'issue de la compilation du noyau, il faudra donc concaténer ces deux fichiers :

```
cat arch/arm/boot/zImage \
  arch/arm/boot/dts/armada-385-linksys-cobra.dtb \
  > zImage.armada-385-linksys-cobra
```

On pourra alors charger l'image **zImage.armada-385-linksys-cobra** dans U-Boot, et la démarrer avec bootz comme si nous n'utilisions pas de Device Tree (donc avec un seul argument passé à bootz, l'adresse en RAM de l'image).

Dans le cas où la version ou la configuration d'U-Boot ne permet pas de démarrer une **zImage**, mais seulement une **uImage**, il faudra alors convertir **zImage.armada-385-linksys-cobra** en **uImage** en utilisant l'outil **mkimage** fourni par U-Boot.

Pour les développeurs du noyau, il s'agit bien d'un mode de compatibilité permettant d'assurer la transition, mais *in fine*, le souhait est que les bootloaders disposent progressivement du support du Device Tree.

5 Syntaxe de base du Device Tree

Comme son nom l'indique, un Device Tree est un arbre, composé d'un ensemble de **nœuds** (**node**) imbriqués les uns avec les autres. Chacun de ces nœuds peut contenir un ensemble de **propriétés** (**properties**) et de sous-nœuds. Voici un extrait d'un Device Tree, provenant du fichier **tegra20-harmony.dts** :

```
        i2c@7000c000 {
                status = "okay";
                clock-frequency = <400000>;

                wm8903: wm8903@1a {
                        compatible = "wlf,wm8903";
                        reg = <0x1a>;
                        interrupt-parent = <&gpio>;
                        gpio-controller;
                        #gpio-cells = <2>;
                        [...]
                };
        };
```

Dans cet exemple, **i2c@7000c000** est un nœud, qui comporte deux propriétés (**status** et **clock-frequency**) ainsi qu'un sous-nœud, **wm8903@1a**, qui lui-même comporte de nombreuses propriétés. La chaîne **wm8903**, qui précède le **:**

est appelée un **label** et permet à une propriété de faire référence à ce nœud par l'intermédiaire d'un **phandle**.

Les propriétés peuvent être de différents types, avec notamment :

- des chaînes de caractère (cas de la propriété **status**) ;
- des entiers (cas de la propriété **clock-frequency**) ;
- des booléens (cas de la propriété **gpio-controller**) ;
- des pointeurs vers d'autres nœuds (cas de la propriété **interrupt-parent**), appelés *phandle*.

Le Device Tree étant un arbre, il commence à une racine, désignée par **/**. La syntaxe du Device Tree permet également d'inclure d'autres fichiers :

- Avec la directive **/include/**, interprétée par le compilateur DTC lui-même.
- Avec la directive **#include**, interprétée par le pré-processeur C, qui est utilisée par le noyau Linux pour préparer les Device Tree sources avant de les compiler avec DTC. Cette directive est désormais préférée, car elle permet en plus de remplacer des valeurs magiques dans le Device Tree par des **#define** traditionnels, définis dans des **.h**.

```
#include <dt-bindings/input/input.h>
#include "tegra20.dtsi"

/ {
  [...]
} ;
```

Le compilateur du Device Tree va vérifier la syntaxe du fichier, mais n'a pas de connaissance du nom des nœuds possibles, du nom et du type des propriétés : il ne fait qu'une pure vérification syntaxique. Ainsi, du point de vue du compilateur DTC, il est possible d'utiliser n'importe quel nom de nœud, de propriétés ou de valeurs.

6 Inspection du Device Tree

Il est parfois utile de pouvoir inspecter le Device Tree, soit un **.dtb** existant sur notre machine hôte, soit le Device Tree en cours d'utilisation sur une plateforme embarquée.

Un **.dtb** existant peut être « décompilé » en utilisant le compilateur DTC, il suffit d'invoquer :

```
dtc -I dtb -O dts arm/boot/dts/armada-xp-gp.dtb
```

La sortie sera quasiment équivalente au Device Tree Source utilisé pour générer ce DTB, à l'exception des **#define** qui auront été résolus, et des *phandles* qui ne seront plus visibles sous la forme de la notation symbolique **&label**, mais sous la forme d'un numéro qui identifie le nœud pointé par le *phandle*.

Sur la cible, il est possible d'accéder au Device Tree en cours d'utilisation et de naviguer dedans par l'intermédiaire d'une hiérarchie de répertoires et de fichiers dans **/sys**. Chaque nœud du Device Tree devient un répertoire, chaque propriété devient un fichier :

```
# ls -l /sys/firmware/devicetree/base/
-r--r--r--    1 root     root     4 Jan  1 00:00 #address-cells
-r--r--r--    1 root     root     4 Jan  1 00:00 #size-cells
drwxr-xr-x    2 root     root     0 Jan  1 00:00 chosen
drwxr-xr-x    3 root     root     0 Jan  1 00:00 clocks
-r--r--r--    1 root     root    34 Jan  1 00:00 compatible
[...]
-r--r--r--    1 root     root     1 Jan  1 00:00 name
drwxr-xr-x   10 root     root     0 Jan  1 00:00 soc
```

7 Organisation générale du Device Tree et notion de « Device Tree binding »

Bien que le compilateur DTC accepte n'importe quel Device Tree syntaxiquement correct, la norme ePAPR définit l'organisation générale d'un Device Tree, avec un certain nombre de nœuds et de propriétés de base. Parmi celles-ci :

- À la racine, une propriété **model** de type chaîne de caractères, qui contient le nom « human readable » de la plateforme matérielle supportée par ce Device Tree, et une propriété **compatible**, qui contient une liste de chaînes de caractères permettant au système d'identifier cette plateforme.
- À la racine, un nœud **memory**, qui contient une propriété **reg** indiquant l'adresse de base de la RAM et la quantité de RAM disponible sur la plateforme.
- À la racine, un nœud **chosen**, qui contient diverses propriétés spécifiques au système d'exploitation. En particulier sous Linux, la propriété **/chosen/bootargs** permet au chargeur de démarrage de passer au noyau sa ligne de commandes.

Exemple avec le Device Tree **usb_a9260.dts** :

```
/ {
        model = "Calao USB A9260";
        compatible = "calao,usb-a9260", "atmel,at91sam9260", "atmel,at91sam9";

        chosen {
                bootargs = "mem=64M console=ttyS0,115200 root=/dev/mtdblock5
rw rootfstype=ubifs";
        };

        memory {
                reg = <0x20000000 0x4000000>;
        };
  [...]
};
```

En dehors des nœuds normalisés par ePAPR, il est possible de créer autant de nœuds spécifiques que nécessaire pour décrire le matériel de la plateforme. Chacun de ces nœuds décrira le plus souvent un périphérique du système grâce à un ensemble de propriétés, certaines étant normalisées par ePAPR et d'autres pouvant être ajoutées si ePAPR ne fournit pas de propriété adaptée pour décrire un aspect du matériel. Ces nœuds seront organisés dans le Device Tree sous la forme d'une hiérarchie qui reproduira celle des bus du système : tout d'abord les bus internes au SoC puis dans ses sous-noeuds les bus externes (SPI, I2C, MMC, etc.).

La façon de décrire un périphérique dans le Device Tree sous la forme de nœuds et de propriétés constitue une spécification appelée **Device Tree binding**. Dans le noyau Linux, chaque Device Tree binding fait l'objet d'une documentation au format texte dans **`Documentation/devicetree/bindings/`**.

Parmi les propriétés les plus courantes et normalisées par le ePAPR, on trouve :

- **`compatible`**, qui contient une liste de chaînes de caractères identifiant le *programming model* du périphérique, c'est-à-dire la façon de faire fonctionner le périphérique (organisation et utilisation des registres). En pratique, dans le contexte de Linux, la chaîne compatible permet d'associer un périphérique décrit dans le Device Tree et le pilote de périphérique correspondant dans le kernel. D'une certaine façon, cette propriété remplace le champ **`name`** de la structure **`platform_device`**.
- **`reg`**, qui contient une liste de couples (adresse, taille) donnant la localisation des registres permettant de contrôler le périphérique.
- **`interrupts`**, qui contient une liste des interruptions utilisées par le périphérique.
- **`status`**, chaîne dont les valeurs sont soit **`okay`**, soit **`disabled`**. Si **`okay`** est utilisé, alors le périphérique est considéré comme actif, et donc le pilote de périphérique correspondant au périphérique verra sa méthode **`->probe()`** appelée. À l'inverse, si **`disabled`** est utilisé, le périphérique ne sera pas activé.

Ainsi, si l'on regarde le Device Tree binding pour le contrôleur USB EHCI Orion, que nous avions utilisé plus tôt pour illustrer la réprésentation non-Device Tree du matériel, nous trouvons dans **`Documentation/devicetree/bindings/usb/ehci-orion.txt`** :

```
* EHCI controller, Orion Marvell variants

Required properties:
- compatible: must be "marvell,orion-ehci"
- reg: physical base address of the controller and length of memory
mapped
  region.
- interrupts: The EHCI interrupt

Optional properties:
- clocks: reference to the clock
- phys: reference to the USB PHY
- phy-names: name of the USB PHY, should be "usb"
```

On trouve ce Device Tree binding utilisé dans **`arch/arm/boot/dts/orion5x.dtsi`**, qui décrit les périphériques présents dans le SoC Orion5x de Marvell :

```
ehci@: ehci@50000 {
        compatible = "marvell,orion-ehci";
        reg = <0x50000 0x1000>;
        interrupts = <17>;
        status = "disabled";
};
```

Avec le Device Tree, le développeur d'un nouveau pilote de périphérique doit donc réfléchir à la façon dont son périphérique va être décrit dans le Device Tree et rédiger la documentation du Device Tree binding correspondant à ce périphérique.

8 Interaction entre Device Tree et pilote de périphérique

Au démarrage, le noyau reçoit dans le registre **`r2`** l'adresse du DTB et va donc pouvoir le parcourir pour découvrir le matériel de la plateforme sur laquelle il s'exécute. En particulier, pour chaque nœud dans le Device Tree qui est un fils d'un nœud identifié comme un **`simple-bus`**, le noyau va instancier une structure **`platform_device`** représentant ce périphérique. De la même façon, si un nœud décrit un contrôleur I2C, le noyau va instancier pour chacun des nœuds fils une structure **`i2c_client`**. Il est en de même pour d'autres types de bus. Ce mécanisme permet donc de remplacer la description et l'enregistrement « à la main », en C, de structures **`platform_device`** avec **`platform_device_register()`**.

```
/ {
  ahb {
    compatible = "simple-bus";
    [...]
    apb {
      compatible = "simple-bus";
      [...]
      uart@: serial@f8004000 {
        compatible = "atmel,at91sam9260-usart";
        reg = <0xf8004000 0x100>;
        interrupts = <27 IRQ_TYPE_LEVEL_HIGH 5>;
        [...]
        status = "okay";
      };
      i2c@: i2c@f8014000 {
```

```
      compatible = "atmel,at91sam9x5-i2c";
      reg = <0xf8014000 0x4000>;
      interrupts = <32 IRQ_TYPE_LEVEL_HIGH 6>;
      [...]
      status = "okay";
      wm8904: codec@1a {
        compatible = "wlf,wm8904";
        reg = <0x1a>;
        [...]
      };
    };
    uart1: serial@fc004000 {
      compatible = "atmel,at91sam9260-usart";
      reg = <0xfc004000 0x100>;
      interrupts = <28 IRQ_TYPE_LEVEL_HIGH 5>;
      [...]
      status = "okay";
    };
  } ;
} ;
```

Ainsi, dans l'exemple ci-dessus :

- Les nœuds **/ahb/apb/serial@f8004000**, **/ahb/apb/i2c@f8014000** et **/ahb/apb/serial@fc004000**, étant des nœuds fils d'un nœud **simple-bus**, ils vont donner lieu à la création et à l'enregistrement de trois structures **platform_device**.
- Le nœud **/ahb/apb/i2c@f8014000/codec@1a** va quant à lui donner lieu à la création et à l'enregistrement d'une structure **i2c_client**. Il s'agit en effet d'un périphérique I2C, puisqu'il est décrit dans le Device Tree comme un fils d'un contrôleur I2C.

Il est intéressant de noter que les nœuds décrivant **uart0** et **uart1** utilisent la même chaîne **compatible** : il s'agit en effet du même bloc matériel, réutilisé deux fois dans le SoC pour fournir deux UARTs indépendantes. Les nœuds diffèrent cependant dans l'adresse de base des registres ainsi que dans le numéro d'interruption utilisé. Un seul **platform_driver** est utilisé et sera responsable de gérer deux **platform_device** : il y aura donc deux appels à la méthode **->probe()** du driver d'UART, un pour chaque UART.

La hiérarchie des nœuds avec un nœud **ahb** à la racine et un sous-nœud **apb** reflète directement l'organisation

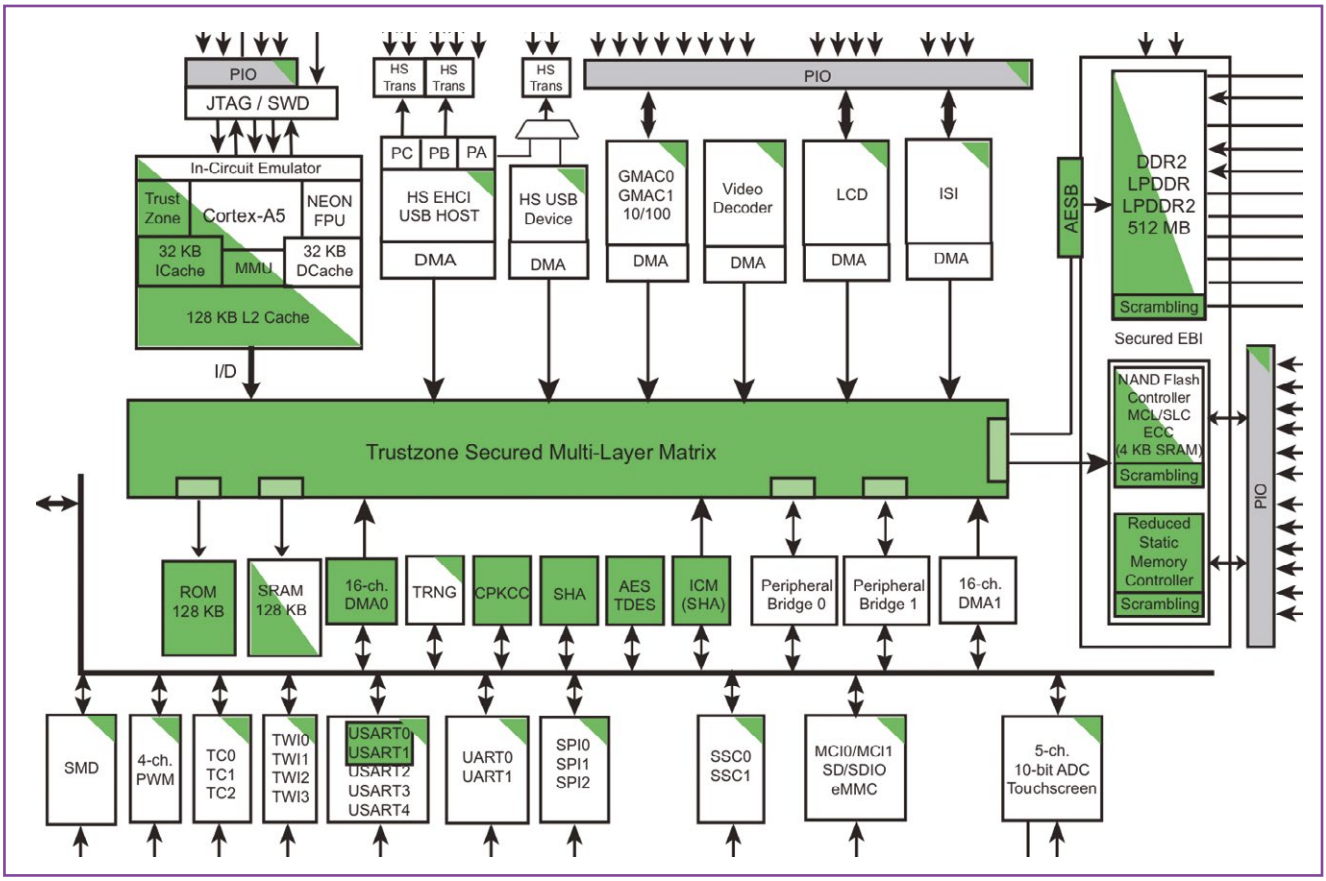

Figure 2 : Extrait du diagramme bloc du Atmel SAMA5D4, provenant de la datasheet de ce SoC ARM. Le bus AHB est la boîte « Trustzone Secured Multi-Layer Matrix » qui relie le processeur Cortex-A5 aux périphériques « haute performance » (contrôleurs RAM, LCD, réseau, USB, décodeur vidéo et entrée caméra). Le bus APB est la ligne noire grasse qui est reliée au AHB par l'intermédiaire des « Peripheral Bridge », et qui permet au processeur d'accéder aux périphériques « basse performance » (contrôleurs SPI, I2C, UART, MMC, PWM, ADC, etc.).

matérielle du SoC, le Atmel SAMA5D4 dans l'exemple que nous avons utilisé, dont un extrait du diagramme bloc est visible figure 2, page précédente.

Du côté du pilote de périphérique, ce n'est plus le champ **platform_driver.driver.name** qui permet de faire l'association entre le **platform_device** et le **platform_driver**, mais un nouveau champ **of_match_table**, qui pointe vers une table des chaînes compatibles qui sont supportées par ce pilote. Ainsi, dans **drivers/i2c/busses/i2c-at91.c**, on trouve :

```
static const struct of_device_id atmel_twi_dt_ids[] = {
  {
    .compatible = "atmel,at91rm9200-i2c",
    .data = &at91rm9200_config,
  }, {
    [...]
  }, {
    .compatible = "atmel,at91sam9x5-i2c",
    .data = &at91sam9x5_config,
  }, {
    .compatible = "atmel,sama5d2-i2c",
    .data = &sama5d2_config,
  }, {
    /* sentinel */
  }
};
MODULE_DEVICE_TABLE(of, atmel_twi_dt_ids);

static struct platform_driver at91_twi_driver = {
  .probe   = at91_twi_probe,
  .driver   = {
    .name= "at91_i2c",
    .of_match_table = of_match_ptr(atmel_twi_dt_ids),
  },
};
```

Notre nœud **/ahb/apb/i2c@f8014000** ayant comme propriété compatible la valeur **atmel,at91sam9x5-i2c**, et cette valeur étant listée dans **atmel_twi_dt_ids[]**, c'est bien ce driver qui sera utilisé pour gérer ce périphérique. La fonction **at91_twi_probe()** sera donc appelée pour initialiser ce contrôleur I2C.

Ce driver gère plusieurs variantes du même contrôleur I2C, qui fonctionnent de manière assez similaire, mais avec néanmoins quelques différences. C'est la raison pour laquelle le tableau **atmel_twi_dt_ids[]** contient plusieurs entrées : une pour chaque chaîne **compatible** supportée par ce driver.

Pour accéder aux informations standardisées du Device Tree, telles que l'adresse des registres, le numéro d'interruption ou une référence vers une clock, le pilote de périphérique n'a pas à se soucier de savoir si le périphérique est décrit par un Device Tree ou à l'ancienne en C : il utilise une API générique. Ainsi, si **pdev** est le pointeur vers la structure **platform_device**, la méthode **->probe()** pourra utiliser :

- **platform_get_resource(pdev, IORESOURCE_MEM, 0)** pour récupérer une **struct resource** qui donne l'adresse et la taille des registres, en provenance de la propriété **reg** du Device Tree ;
- **platform_get_irq(pdev, 0)** pour récupérer le numéro d'interruption, en provenance de la propriété **interrupts** du Device Tree ;
- **clk_get(&pdev->dev, NULL)** pour récupérer une référence vers la clock du périphérique, telle que décrit par la propriété **clocks** du Device Tree.

Cependant, certains Device Tree bindings font appel à des propriétés spécifiques. Dans ce cas, le kernel fournit un ensemble de fonctions **of_*()** qui permet de récupérer des informations du Device Tree. Par exemple :

```
of_property_read_u32(pdev->dev.of_node, "atmel,fifo-size", &dev->fifo_size)
```

permet de récupérer la valeur de la propriété **atmel,fifo-size**. **pdev->dev.of_node** est un pointeur vers une **struct device_node**, qui représente le nœud dans le Device Tree qui correspond au **platform_device pdev**.

9 Inclusion de Device Tree et surcharge

Comme nous l'indiquions plus tôt, la syntaxe du Device Tree permet en utilisant **/include/** ou **#include** d'inclure d'autres fichiers. Ce mécanisme d'inclusion permet d'utiliser des **#define**, mais surtout, il permet de factoriser dans des fichiers partagés les aspects communs de plusieurs plateformes.

Ainsi, toutes les cartes électroniques utilisant le même SoC ne redéfinissent pas chacune à partir de zéro tous les périphériques du SoC : cette description est factorisée dans un fichier commun. Par convention, de tels fichiers « communs » utilisent l'extension **.dtsi**, tandis que les Device Tree « finaux » utilisent l'extension **.dts**.

Si l'on prend l'exemple du Device Tree **at91-sama5d4_xplained.dts** qui décrit la carte Atmel Xplained SAMA5D4, il inclut le fichier **sama5d4.dtsi** qui lui décrit le SoC SAMA5D4 :

```
#include "sama5d4.dtsi"
```

Ce fichier **sama5d4.dtsi** lui-même inclut un fichier qui est à la racine de la chaîne d'inclusion de tous les Device Tree : **skeleton.dtsi**.

```
#include "skeleton.dtsi"
```

Là où ce mécanisme d'inclusion devient particulièrement puissant, c'est que l'arbre décrit dans un Device Tree qui inclut un autre Device Tree vient **surcharger** l'arbre du Device Tree inclus. Cela permet par exemple à un fichier Device Tree décrivant une carte électronique :

- d'ajouter aux périphériques du SoC les périphériques propres à la carte : périphériques sur I2C ou SPI, LEDs, boutons, etc.
- de surcharger des propriétés des périphériques du SoC pour préciser des aspects spécifiques à la carte. Notamment, le Device Tree d'un SoC utilise généralement **`status = "disabled"`** pour la majorité de ses périphériques, et laisse le soin au Device Tree décrivant la carte de surcharger cette propriété **`status`** à la valeur **`okay`** pour les périphériques qui sont effectivement utilisés.

Le fichier **`sama5d4.dtsi`** décrit par exemple :

```
/ {
  ahb {
    apb {
      spi0: spi@f8010000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "atmel,at91rm9200-spi";
        reg = <0xf8010000 0x100>;
        interrupts = <37 IRQ_TYPE_LEVEL_HIGH 3>;
        [...]
        clocks = <&spi0_clk>;
        clock-names = "spi_clk";
        status = "disabled";
      };
    };
  };
};
```

Et le fichier **`at91-sama5d4_xplained.dts`** décrit :

```
#include "sama5d4.dtsi"
/ {
  ahb {
    apb {
      spi0: spi@f8010000 {
        cs-gpios = <&pioC 3 0>, <0>, <0>, <0>;
        status = "okay";
        m25p8000 {
          compatible = "atmel,at25df321a";
          spi-max-frequency = <50000000>;
          reg = <0>;
        };
      };
    };
  };
};
```

Alors le Device Tree Blob généré par le compilateur à partir de **`at91-sama5d4_xplained.dts`** contiendra en réalité :

```
/ {
  ahb {
    apb {
      spi0: spi@f8010000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "atmel,at91rm9200-spi";
```

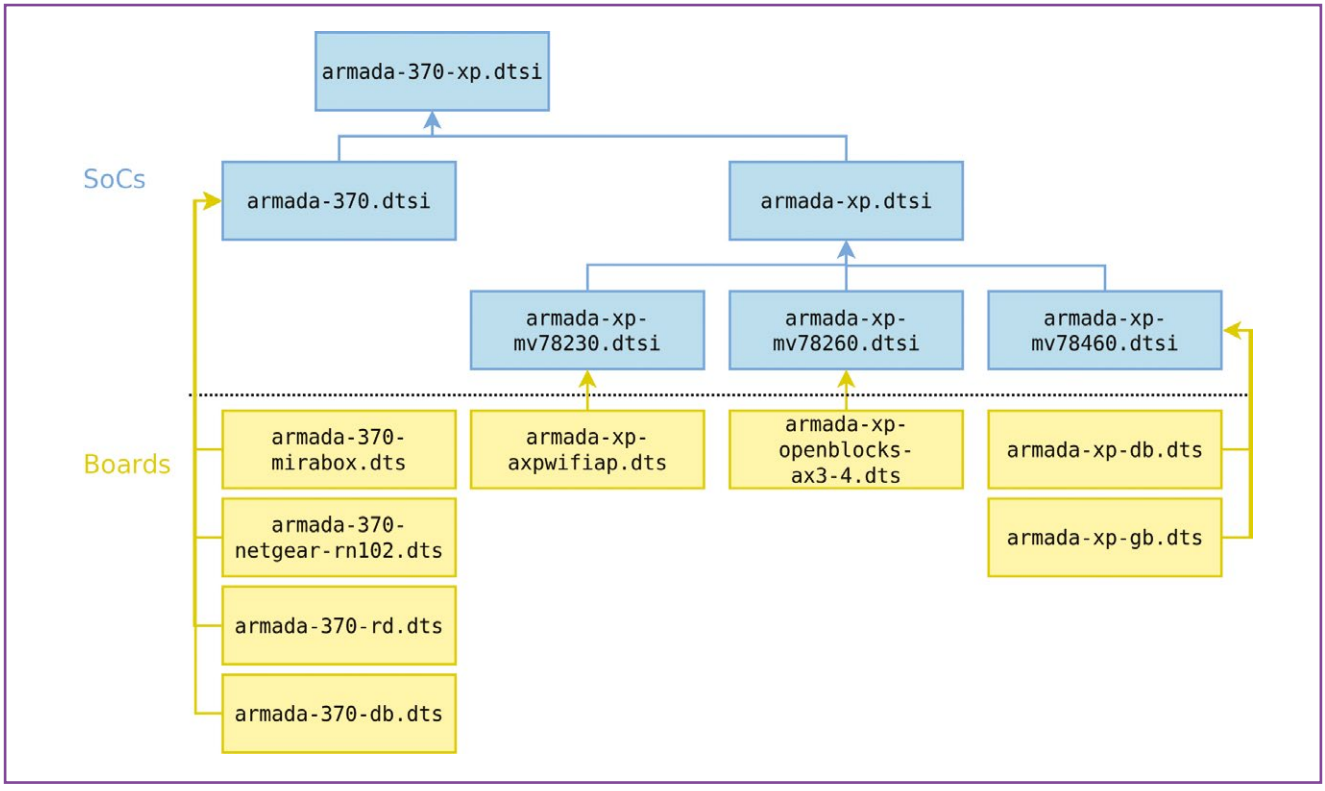

Figure 3 : Hiérarchie partielle des Device Tree utilisés pour décrire les plateformes Marvell EBU dans le noyau Linux. Le fichier armada-370-xp.dtsi contient la description commune aux SoCs Armada 370 et XP, puis les autres fichiers .dtsi précisent les aspects spécifiques à chaque SoC, et enfin les fichiers .dts décrivent les différentes cartes.

```
      reg = <0xf8010000 0x100>;
      interrupts = <37 IRQ_TYPE_LEVEL_HIGH 3>;
      [...]
      clocks = <&spi0_clk>;
      clock-names = "spi_clk";
      status = "okay";
      cs-gpios = <&pioC 3 0>, <0>, <0>, <0>;
      m25p80@0 {
        compatible = "atmel,at25df321a";
        spi-max-frequency = <50000000>;
        reg = <0>;
      };
    };
  };
};
```

Le fichier **sama5d4.dtsi** décrit le contrôleur SPI0, l'adresse de ses registres, son numéro d'interruption, sa clock, et marque le périphérique comme non utilisé (**status = "disabled"**). Le fichier **at91-sam5d4_xplained.dts** hérite de **sama5d4.dtsi** et y ajoute les aspects spécifiques à cette carte :

- activation du contrôleur SPI0 via **status = "okay"** ;
- indication des GPIOs à utiliser comme *chip select* pour sélectionner le périphérique SPI ;
- description du périphérique SPI connecté sur ce bus, ici une flash SPI Atmel AT25.

Comme illustré sur la figure 3, page précédente, d'autres plateformes utilisent plusieurs niveaux d'inclusion, notamment lorsqu'une famille de SoC comporte différents SoC avec de nombreux aspects communs, mais quelques spécificités. L'inclusion de Device Tree permet de décrire le matériel commun dans un seul fichier, réutilisé pour les différentes plateformes.

10 #address-cells, #size-cells

Certaines propriétés du Device Tree, définies par le ePAPR, ont un nom un peu étrange commençant par **#**, et leur rôle mérite quelques explications. Tout d'abord, contrairement à ce qu'on pourrait penser, ces propriétés ne sont pas des commentaires : les commentaires en Device Tree se font comme en C ou en C++ (avec **/* */** ou **//**) et non pas comme en shell ou Python (avec **#**).

Dans le jardon Device Tree, une *cell* est une valeur de 32 bits : par exemple **<32 12>** a deux cells, de valeur **32** et **12**. Les propriétés **#address-cells** et **#size-cells** servent donc à indiquer le nombre de cells qui serviront à décrire des registres dans la propriété **reg** dans les nœuds enfants du nœud où les propriétés **#address-cells** et **#size-cells** sont définies. Ainsi, si l'on reprend l'exemple précédent de la flash SPI sur la carte Atmel Xplained SAMA5D4 :

```
/ {
  ahb {
    apb {
      compatible = "simple-bus";
      #address-cells = <1>;
      #size-cells = <1>;
      spi0: spi@f8010000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "atmel,at91rm9200-spi";
        reg = <0xf8010000 0x100>;
        cs-gpios = <&pioC 3 0>, <0>, <0>, <0>;
        status = "okay";
        m25p80@0 {
          compatible = "atmel,at25df321a";
          spi-max-frequency = <50000000>;
          reg = <0>;
        };
      };
    };
  };
};
```

Le nœud **/ahb/apb** spécifie **#address-cells = <1>** et **#size-cells = <1>**, ce qui indique que la propriété **reg** des nœuds fils doit contenir deux cellules : une pour l'adresse, une pour la taille. Pour un bus dont les périphériques sont mappés en mémoire physique, cela semble logique : l'adresse physique des registres du périphérique, et la taille de la plage de registres. C'est la raison pour laquelle on trouve **reg = <0xf8010000 0x100>;** dans le nœud **spi@f8010000** : **0xf8010000** est l'adresse de base des registres, et **0x100** la taille des registres.

Le nœud **/ahb/apb/spi@f8010000** spécifie lui **#address-cells = <1>** et **#size-cells = <0>**. En effet, les nœuds fils de ce nœud sont des périphériques SPI : ils ne sont pas mappés en mémoire et donc la propriété **reg** ne sert plus à donner une adresse physique et une taille, mais à donner le numéro du chip-select utilisé pour ce périphérique SPI. C'est la raison pour laquelle on trouve **reg = <0>** dans le nœud **/ahb/apb/spi@f8010000/m25p80@0**.

Cette syntaxe **#<element>-cells** existe aussi pour d'autres éléments : **#gpio-cells**, **#interrupt-cells**, **#phy-cells**, **#clock-cells**, **#dma-cells**, avec à chaque fois le même principe.

11 Exemple de Device Tree : les interruptions

Pour conclure cet article, nous vous proposons d'étudier plus particulièrement la description des interruptions, avec deux exemples distincts. Le premier, le plus simple, basé sur le Device Tree de la plateforme Freescale I.MX28, et le

second, plus complexe, basé sur le Device Tree de la carte Nvidia Tegra 20 Harmony.

Pour la gestion des interruptions, plusieurs propriétés entrent en jeu :

- la propriété **interrupts**, qui permet à un périphérique utilisateur d'une interruption d'indiquer quel est le numéro d'interruption qu'il utilise ;
- la propriété **interrupt-parent**, qui permet à un périphérique d'indiquer à quel contrôleur d'interruption il est relié. Il s'agit donc d'une propriété de type phandle, qui pointe sur le nœud décrivant le contrôleur d'interruption. Si elle n'est pas définie au niveau du périphérique, le noyau va rechercher une telle propriété en remontant dans les nœuds parents. Le plus souvent, on trouve donc une propriété **interrupt-parent** au premier niveau de la hiérarchie des nœuds dans le Device Tree ;
- la propriété booléenne **interrupt-controller**, qui permet à un nœud de se déclarer comme étant un contrôleur d'interruption. Cela autorise entre autres ce nœud à être pointé par la propriété **interrupt-parent** ;
- la propriété **#interrupt-cells** qui permet à un nœud décrivant un contrôleur d'interruption de définir combien de cells sont nécessaires dans la propriété **interrupts** pour décrire une interruption de ce contrôleur.

Notre premier exemple de la plateforme Freescale I.MX28 vient du fichier **imx28.dtsi** :

```
/ {
 interrupt-parent = <&icoll>;
 apb@80000000 {
  apbh@80000000 {
   icoll: interrupt-controller@80000000 {
    compatible = "fsl,imx28-icoll", "fsl,icoll";
    interrupt-controller;
    #interrupt-cells = <1>;
    reg = <0x80000000 0x2000>;
   };
   duart: serial@80074000 {
    compatible = "arm,pl011", "arm,primecell";
    reg = <0x80074000 0x1000>;
    interrupts = <47>;
    status = "disabled";
   };
  };
 };
};
```

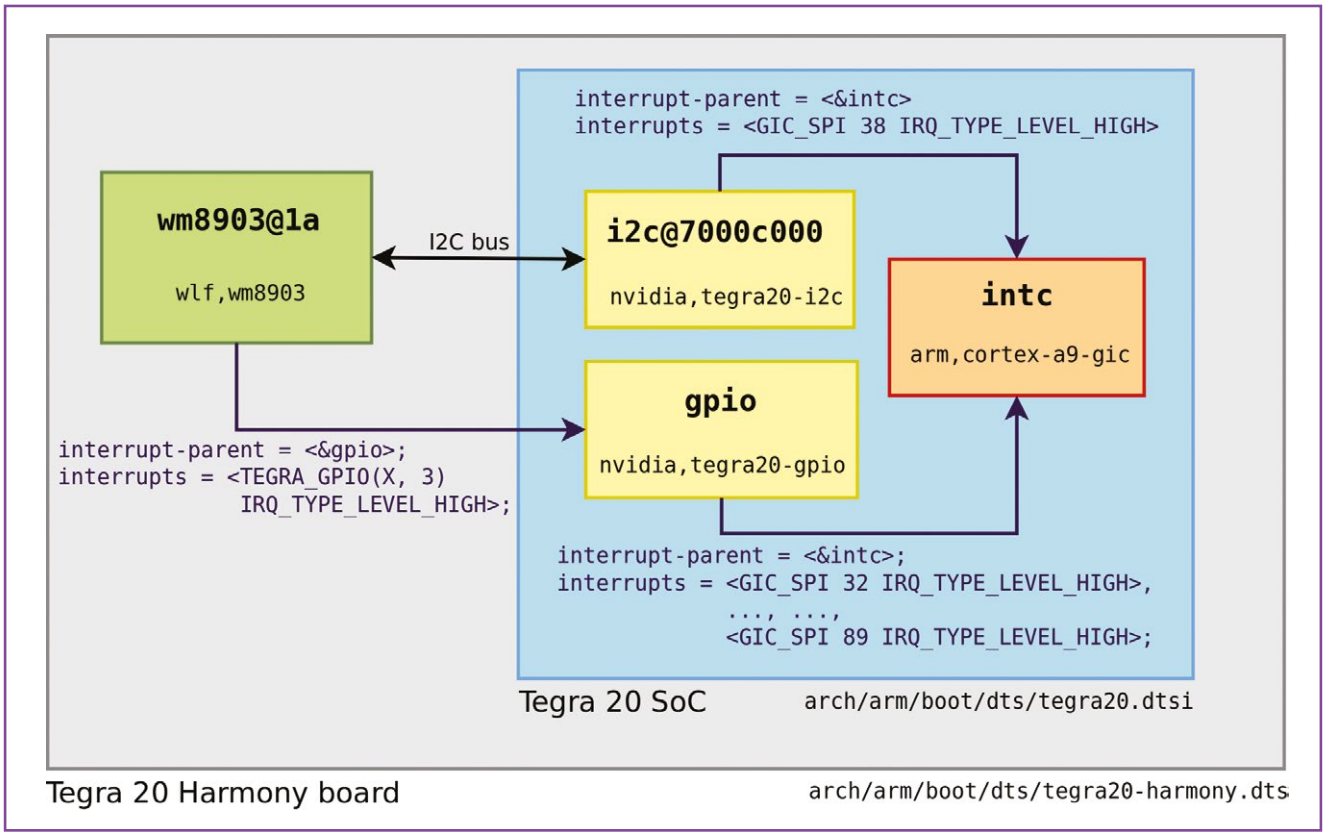

Figure 4 : Représentation schématique partielle du matériel de la carte Nvidia Tegra 20 Harmony, avec le contrôleur d'interruption, le contrôleur I2C et le contrôleur de GPIO du SoC, ainsi qu'un codec audio sur I2C sur la carte.

Dans cet exemple, on trouve un nœud **interrupt-controller@80000000** qui décrit le contrôleur d'interruption, et un nœud **serial@80074000** qui décrit un contrôleur d'UART. Comme prévu, le nœud décrivant le contrôleur d'interruption utilise la propriété **interrupt-controller**, et spécifie **<1>** comme valeur de **#interrupt-cells**. Ce contrôleur d'interruption est identifié comme étant le contrôleur d'interruption commun à la plupart des périphériques via la définition **interrupt-parent = <&icoll>**. Enfin, le contrôleur d'UART indique qu'il utilise l'interruption **<47>** de ce contrôleur d'interruption via la propriété **interrupts**, pour laquelle une seule cellule est nécessaire.

Notre exemple plus complexe de la plateforme Nvidia Tegra 20 Harmony est illustré par la figure 4, page précédente.

Dans cette figure, on retrouve dans la zone bleue le SoC Tegra 20 qui comporte (entre autres) un contrôleur d'interruption **intc**, un contrôleur de GPIO et un contrôleur I2C. Le contrôleur de GPIO et le contrôleur I2C ont chacun une interruption qui remonte vers le contrôleur d'interruption **intc** (ce dernier étant le contrôleur d'interruption principal, il est ensuite directement relié à la broche IRQ du Cortex-A9 afin de pouvoir interrompre le CPU). La zone grise autour du SoC représente la carte, qui pour notre illustration comporte un codec audio **wm8903**. Ce codec audio est connecté au processeur sur un bus I2C, qui permet de le contrôler. Mais en complément, ce codec audio est aussi relié au SoC par l'intermédiaire d'une GPIO qui est utilisée par le codec audio pour notifier au SoC une interruption.

D'un point de vue Device Tree, nous trouvons donc la représentation suivante du SoC Tegra 20 dans **tegra20.dtsi** :

```
/ {
  interrupt-parent = <&intc>;

  intc: interrupt-controller {
    compatible = "arm,cortex-a9-gic";
    reg = <0x50041000 0x1000 0x50040100 0x0100>;
    interrupt-controller;
    #interrupt-cells = <3>;
  };

  i2c@7000c000 {
    compatible = "nvidia,tegra20-i2c";
    reg = <0x7000c000 0x100>;
    interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
    [...]
  };

  gpio: gpio {
    compatible = "nvidia,tegra20-gpio";
    reg = <0x6000d000 0x1000>;
    interrupts = <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>, <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>,
           [...], <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
    #gpio-cells = <2>;
    gpio-controller;
    #interrupt-cells = <2>;
    interrupt-controller;
  };
};
```

On retrouve donc l'intc identifié comme le contrôleur d'interruption principal par **interrupt-parent = <&intc>**. Ce contrôleur d'interruption spécifie un **#interrupt-cells** de **<3>**, ce qui explique la présence de 3 cellules dans les propriétés **interrupts** de nœuds représentant le contrôleur I2C et le contrôleur de GPIO. Pour connaître la signification de ces trois cellules, il faut se référer au Device Tree binding du **"arm,cortex-a9-gic"** disponible dans **Documentation/devicetree/bindings/arm/gic.txt**. La première cellule spécifie le type d'interruption (**GIC_PPI** pour interruption per-CPU, **GIC_SPI** pour interruption globale), la seconde cellule donne le numéro d'interruption, et la troisième cellule des flags indiquant si l'interruption est sur front montant/descendant ou niveau haut/bas ainsi qu'éventuellement d'autres informations.

Le contrôleur de GPIO lui-même peut jouer le rôle de contrôleur d'interruption : lorsqu'une de ses GPIO change d'état, il peut remonter une interruption à l'**intc**. C'est la raison pour laquelle il dispose aussi de la propriété **interrupt-controller**. Cependant, il spécifie une **#interrupt-cells** de **<2>**.

Au niveau du Device Tree de la carte Tegra 20 Harmony, décrite dans **tegra20-harmony.dts**, on trouve :

```
i2c@7000c000 {
  status = "okay";
  clock-frequency = <400000>;

  wm8903: wm8903@1a {
    compatible = "wlf,wm8903";
    reg = <0x1a>;
    interrupt-parent = <&gpio>;
    interrupts = <TEGRA_GPIO(X, 3) IRQ_TYPE_
LEVEL_HIGH>;
  };
};
```

Comme prévu, notre codec audio est situé sur le bus I2C, donc décrit comme un sous-nœud du contrôleur I2C. Il est accessible à l'adresse 0x1a sur le bus I2C, comme indiqué par la propriété **reg**. Mais le plus intéressant est dans la description de l'interruption : comme nous l'indiquions plus haut, ce codec a une broche d'interruption, qui a été connectée à une GPIO du SoC Tegra 20. Le Device Tree spécifie donc que le contrôleur d'interruption pour ce périphérique est le contrôleur de GPIO en déclarant **interrupt-parent = <&gpio>**, et il déclare quelle GPIO est utilisée par la spécification

`interrupts = <TEGRA_GPIO(X, 3) IRQ_TYPE_LEVEL_HIGH>`. On reconnaît ici une spécification contenant deux cellules, comme requis par la propriété `#interrupt-cells` du contrôleur de GPIO.

Le lecteur attentif notera que le Device Tree pour cette plateforme a sensiblement changé dans le noyau depuis le commit 870c81a41f7074a99599be7f10ce5aab43c9f0c4 (intégré depuis 4.1). Nous avons choisi de décrire le Device Tree d'avant 4.1, qui était un peu plus simple pour illustrer le propos de cet article.

Comme on peut le constater, la syntaxe et l'organisation du Device Tree requièrent un peu d'habitude, mais une fois les principes de base compris, on s'aperçoit vite que le Device Tree permet de représenter de manière assez fidèle et succincte les périphériques matériels et leurs interconnexions.

12 Le Device Tree, une ABI stable ?

Comme nous l'indiquions plus tôt dans l'article, le Device Tree a en théorie pour vocation d'être une représentation du matériel indépendante du système d'exploitation. Afin de rendre cela effectivement possible, il est donc nécessaire de s'assurer qu'une fois un Device Tree binding est défini et utilisé pour décrire un matériel, ce binding continue d'être supporté éternellement par les systèmes d'exploitation qui veulent gérer ce matériel. Ainsi, un Device Tree créé aujourd'hui pour décrire une plateforme matérielle donnée, devrait pouvoir continuer à fonctionner avec toutes les versions futures du noyau Linux.

Les développeurs du noyau Linux ont, lors de l'introduction du Device Tree sur ARM, tenté de se plier à cette contrainte : dès lors qu'un binding Device Tree évolue, il est nécessaire que le driver correspondant continue à supporter l'ancienne version du binding Device Tree. Il a ainsi par exemple été question de séparer les Device Tree et leurs bindings du code du noyau Linux, et de les gérer dans un dépôt séparé, afin de faciliter leur réutilisation par d'autres projets.

En pratique, conserver une telle compatibilité ascendante n'est pas toujours évident. Pour certaines évolutions simples, comme l'ajout d'une nouvelle propriété, conserver une telle compatibilité ne demande qu'un effort minimal. Mais parfois, des évolutions plus radicales des bindings sont nécessaires, par exemple lorsque le matériel était insuffisamment compris ou documenté au moment du développement de son support initial. Conserver la compatibilité avec les bindings précédents devient alors beaucoup plus délicat.

Ce point fait toujours l'objet de débats dans la communauté du noyau Linux. Certaines plateformes ont choisi de déclarer explicitement leurs bindings Device Tree comme « non stables ». Votre auteur a d'ailleurs donné sur ce sujet une conférence intitulée « Device Tree as an ABI : a fairy tale ? » lors de l'Embedded Linux Conference 2015.

Conclusion

Au cours de cet article, nous avons tout d'abord pu comprendre le besoin d'une description statique du matériel pour des architectures comme les SoC ARM, étudier la façon dont ce matériel était décrit avant l'introduction du Device Tree, puis détaillé comment le Device Tree avait radicalement modifié la façon de décrire les périphériques matériels. Avec ces informations en main, le lecteur devrait maintenant mieux comprendre le rôle joué par le Device Tree, et commencer à pouvoir réaliser les premières modifications d'un Device Tree. ■

Références

- DeviceTree.org, http://www.devicetree.org/
- ePAPR, https://www.power.org/documentation/epapr-version-1-1/
- **`Documentation/devicetree/bindings/`** dans les sources du noyau
- Présentation « Device Tree for Dummies », Thomas Petazzoni, Embedded Linux Conference Europe 2013. Slides : https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf et vidéo : https://www.youtube.com/watch?v=m_NyYEBxfn8.
- Présentation « Device Tree as a stable ABI : a fairy tale », Thomas Petazzoni, Embedded Linux Conference 2015. Slides : http://free-electrons.com/pub/conferences/2015/elc/petazzoni-dt-as-stable-abi-fairy-tale/petazzoni-dt-as-stable-abi-fairy-tale.pdf, vidéo : https://www.youtube.com/watch?v=rPRqIS9q6CY.
- Présentation « Solving Device Tree Issues », Frank Rowand, Embedded Linux Conference Europe 2015. Slides : http://events.linuxfoundation.org/sites/events/files/slides/dt_debugging_1.pdf.

*À propos de l'auteur : Thomas Petazzoni est CTO et ingénieur Linux embarqué chez Free Electrons (*http://www.free-electrons.com*), société offrant des services de développement et de formation autour de Linux embarqué. Il contribue de manière importante au support des processeurs Marvell EBU dans le noyau Linux, et est un des contributeurs principaux du projet Buildroot.*