

Linux Embarqué

Séance 4 : Démarrage et Device Trees

Laurent Fiack

Bureau D212 – laurent.fiack@ensea.fr

Menu du jour

Démarrage

- Démarrage

Device Tree

- Introduction

- Pourquoi le device tree ?

- Principe de fonctionnement

- Syntaxe

- Écriture de driver

- Les interruptions

Démarrage

Pourquoi c'est compliqué ?

■ Dans FreeRTOS

- Un peu de code ASM avant le `main()` : Mémoires, horloges
- Initialisations basiques dans le `main()`
- Créations des tâches
- Démarrage de l'ordonnanceur

■ Dans Linux

- Exécution depuis la RAM
- Chargement de drivers (Contrôleurs mémoire, port série, réseau, affichage, clavier...)
- Système(s) de fichier
- Dans cet ordre
- Mais les drivers sont des fichiers...

Démarrage

- Composants logiciels impliqués dans le démarrage :
 - Bootloader
 - Kernel Image
 - Root file system (rootfs) – image `initrd` ou lien NFS
- Étapes du démarrage d'un PC
 - System Startup - PC-BIOS/BootMonitor
 - Stage1 bootloader - MBR
 - Stage2 bootloader - LILO, GRUB etc
 - Kernel - Linux
 - Init - The User Space
- Démarrage d'un système embarqué
 - Au lieu du BIOS, le processeur démarre à partir d'une adresse de la Flash
 - Le processeur exécute un seul et unique "boot strap firmware" ou "boot loader"
 - Le bootloader fournit des fonctionnalités utiles pour le développement et le debugging

Boot loader

- BIOS/UEFI (dépends du matériel)
 - Le CPU exécute le code du BIOS
 - Première étape : POST (Power On Self Test)
 - "run time services" : Énumération des périphériques et initialisation
 - À la fin du POST : code vidé de la mémoire. Certains services du BIOS restent en mémoire et sont disponibles pour l'OS
 - Recherche des périphériques bootables (disque durs), et sélectionne selon la configuration
 - Le Primary boot loader est chargé et prends le contrôle
- Primary boot loader
 - Initialisations optionnelles
 - Charge le secondary boot loader
- Secondary boot loader
 - Charge Linux et un RAM disk optionel dans la RAM
 - Chargement du initrd (temporaire sur PC, rootfs définitif sur système embarqué)
 - Le noyau est décompressé et initialisé

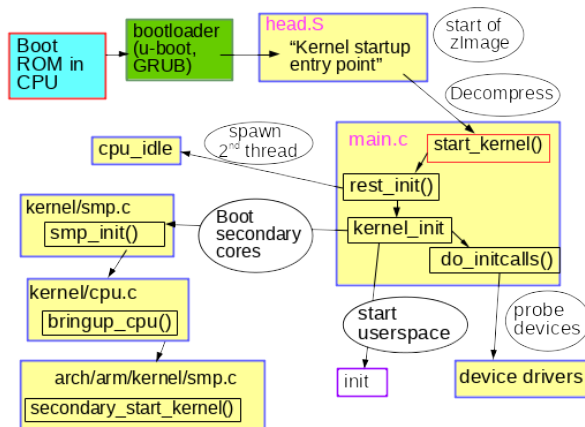
U-boot

- U-Boot (Universal Boot Loader) est l'équivalent du BIOS sur les cartes embarqués
- Si aucun système d'exploitation n'est requis pour l'application, U-Boot peut être utilisé comme base de développement
- U-Boot est open source (GPL), sans royalties et bénéficie d'une communauté de développeurs importante et extrêmement active
- Les tâches principales d'U-Boot
 - Initialisation du matériel et plus particulièrement du contrôleur mémoire
 - Passage des paramètres de démarrage au noyau Linux
 - Lancement du noyau Linux
- U-Boot permet aussi de :
 - Lire et écrire dans différentes zones mémoire
 - Charger des images binaires dans la RAM par câble série, Ethernet ou USB
 - Copier des images binaires de la RAM vers la FLASH
 - Configurer le FPGA

Le noyau

- Invocation du noyau
 - Vérifications du matériel, énumère les périphériques, monte le rootfs
 - Charge les modules nécessaires au démarrage du système
 - Démarrage du premier programme en user-space (init) et initialisations de haut-niveau
 - Similaire sur PC et cible embarquée
- Kernel Image
 - Image compressée (zlib)
 - Appelée zImage (<512 KB) ou bzImage (> 512 KB)
 - head.S à la tête du fichier pour décompresser l'image

Récapitulatif



Device Tree

Device Tree

C'est quoi ?

- Structure de données décrivant les périphériques d'une machine
- Utilisé par le noyau (Linux ou autre)
- Décrit le(s) CPU(s), la mémoire, les bus et les périphériques.
- Pas utilisé sur PC
 - Périphérique « découvrables » de manière dynamique
 - Mécanisme d'énumération et d'identification
 - PCI, USB...
 - Architectures très normalisées
- Provient des architectures SPARC et PowerPC
 - Autres bus sans énumération
 - Architectures moins normalisées
- Quasi-obligatoire pour Linux sur architecture ARM Cortex-Axx

*« Gaah. Guys, this whole ARM thing is a f*cking pain in the ass. »*

– Linus Torvalds

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 ».

http://jmfriedt.free.fr/devicetree_os19.pdf

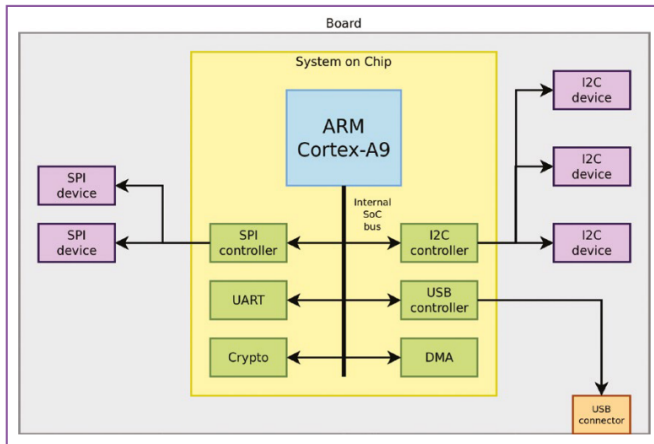
Spécificité des SoC

- Processeur + plein de périphériques
 - Bus : UART, SPI, I2C, CAN...
 - Accélérateurs : CRC, Crypto, GPU...
- Composants connectés aux bus
 - Codec audio, PHY ethernet, Capteurs, HMI...
- Trois niveaux : Processeur, SoC, Carte
- CPU ARM + Intégration d'IP → Grand diversité

Linux sur un SoC

- Pour porter le noyau Linux sur un SoC
 - Jeu d'instruction insuffisant
 - Porter l'ensemble des périphériques
- Périphériques pas découvrables dynamiquement
- Besoin de connaître l'ensemble des périphériques intégrés

Exemple de SoC



Sans Device Tree

- Code spécifique dans /arch/arm
 - Support CPU (IRQs, MMU, Cache..)
- mach-* contient le code spécifique aux SoC (eg mach-socfpga)
- Beaucoup de code C pour chaque SoC
- Encore plus pour les cartes
- Utilisation d'un *machine ID* pour trouver quels drivers charger

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
```

Exemple de fonction init()

mv2120_init()

```
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);  
    [...]  
}
```

Exemple avec un contrôleur USB

- Passage des adresses physiques, numéro d'IRQ et type de contrôleur

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);  
}
```

- Enregistrement d'un platform_device
 - Structure décrivant un périphérique attaché au SoC (\neq pas sur un bus)
- Déclenche l'appel de la fonction `->probe()`
 - It initialise le périphérique et l'enregistre auprès du noyau

Exemple avec un contrôleur USB

- Passage des adresses physiques, numéro d'IRQ et type de contrôleur

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);
}
```

Exemple avec un contrôleur USB

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",  
    },  
};
```

- En résumé :
 - L'ensemble des périphériques de chaque SoC doivent être décrits en C
 - Code très redondant
 - Taille de code importante
- Le Device Tree permet d'éviter ça

Côté bootloader

- Image binaire unique : zImage
- U-Boot responsable de :
 - Charger l'image du noyau dans la RAM
 - Lancer l'exécution du noyau
- Exemple :

Dans U-Boot

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

- U-Boot est compilé spécifiquement pour la plateforme
- Passe le *machine ID* à travers le registre R1

Principe

- Description du matériel du SoC passé au noyau lors du démarrage
- Description dans un fichier appelé *Device Tree Source* (DTS)
- Compilé en binaire dans un fichier *Device Tree Blob* (DTB)
 - Compilation réalisée par *Device Tree Compiler* (DTC)

Dans les sources 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-370-dlink-dns3271.dtb
arch/arm/boot/dts/armada-370-mirabox.dtb
arch/arm/boot/dts/armada-370-netgear-rn102.dtb
arch/arm/boot/dts/armada-385-linksys-caiman.dtb
arch/arm/boot/dts/armada-385-linksys-cobra.dtb
arch/arm/boot/dts/armada-388-db.dtb
arch/arm/boot/dts/armada-388-gp.dtb
arch/arm/boot/dts/armada-388-rd.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
[...]
$
```

Principe

- Indépendant de l'OS
- Peut être utilisé par Linux, un Unix, un bootloader...
- Uniquement utilisé pour des périphériques non découvrables ou énumérables
 - Périphériques USB ou PCI ne sont pas dans le device tree
 - Mais les contrôleurs y sont

Démarrage

- Un fichier en plus à charger par U-Boot
 - Il faut donc charger le noyau et le .dtb

Dans U-Boot

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

- bootz a 3 arguments
 - L'adresse de l'image du noyau
 - L'adresse (optionnelle) d'un initramfs
 - L'adresse (optionnelle) d'un .dtb

Compatibilité

- Sur des vieux systèmes avec un vieux U-Boot
- Il suffit de concaténer l'image du noyau avec le .dtb

Après la compilation

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

Syntaxe

- Structure en arbre
- Composée d'un ensemble de **nœuds** (nodes) imbriqués
- Chaque node peut contenir un ensemble de **propriétés** (properties)

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>;  
        [...]  
    };  
};
```

Syntaxe

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>;  
        [...]  
    };  
};
```

- `i2c@7000c000` est un node
- Deux propriétés `status` et `clock-frequency`
- `wm8903@1a` est un sous-node

Syntaxe

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>;  
        [...]  
    };  
};
```

- Chaîne de caractères (status)
- Entiers (clock-frequency)
- Booléens (gpio-controller)
- Pointeur vers d'autres nodes (interrupt-parent) = *phandle*

Syntaxe

- Commence par une racine /
- On peut include des fichiers */include/*
- Ou en utilisant le précompilateur C `#include`
- On peut aussi utiliser des `#define`
- dtc effectue une vérification syntaxique
 - Mais c'est tout !

Exemple d'include

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

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

Inspection d'un .dtb

Dé-compilation du Device tree

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

Dans /sys/

```
# 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
```

Nodes et properties de base

- La property `model` à la racine définit le nom de la plateforme
- `compatible` utilisable par le système
- Le node `memory` contient l'adresse et la quantité de RAM
- Le node `chosen` contient plusieurs propriétés
 - eg `bootargs` les paramètres passés au noyau

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 0x40000000>;  
    };  
    [...]  
};
```


Organisation générale

- Nodes décrivent des périphériques
- Organisés dans une hierarchie
- Même hierarchie que dans le système
 - Bus systèmes (AXI, AHB, APB...)
 - Bus externes (SPI, I2C, MMC...)
 - Périphériques externes

Interaction entre le DT et le driver

- Adresse du .dtb dans le registre R2
- Le noyau parcourt le .dtb pour découvrir le matériel
- Pour chaque node `compatible` = `"simple-bus"`
 - Le noyau instancie une structure `platform_device`
- Sinon, le node décrit un contrôleur (eg I2C)
 - Instancie les nodes fils en tant que `i2c_client`
 - Pareil pour les autres types de bus
- Permet d'éviter de décrire ces structures en C et de les enregistrer à la main.

Illustration

Création de platform_device

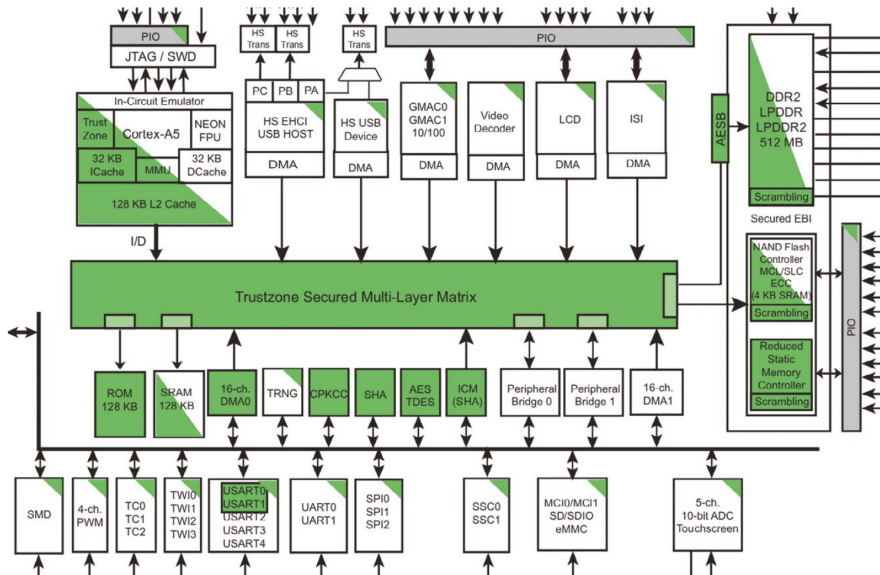
```

/ {
    ahb {
        compatible = "simple-bus";
        [...]
        apb {
            compatible = "simple-bus";
            [...]
            uart0: serial@f8004000 {
                compatible = "atmel,at91sam9260-usart";
                reg = <0xf8004000 0x100>;
                interrupts = <27 IRQ_TYPE_LEVEL_HIGH 5>;
                [...]
                status = "okay";
            };
            i2c0: i2c@f8014000 {
                compatible = "atmel,at91sam9x5-i2c";
                reg = <0xf8014000 0x4000>;
                interrupts = <32 IRQ_TYPE_LEVEL_HIGH 6>;
                [...]
            };
            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";
            };
        };
    };
};

```

Interaction entre le DT et le driver

- Les nodes `/ahb/apb/serial@f8004000`, `/ahb/apb/i2c@f8014000` et `/ahb/apb/serial@fc004000` sont sur un "simple-bus"
 - Il vont créer et enregistrer un `platform_device`
- Le node `/ahb/apb/i2c#f8014000/codec@1a` est le fils d'un contrôleur I2C
 - Il va instancier et enregistrer un `i2c_client`
- Les deux nodes `uart0` et `uart1` ont le même `compatible`
 - Même bloc matériel 2 fois dans le SoC
 - Adresses et interruptions différents
- Un seul `platform_driver` est utilisé
 - Gère deux `platform-device`
 - Il y a donc deux methodes `->probe()`
 - Un pour chaque UART
- La hierarchie dans le `.dts` représente la hierarchie des bus dans le SoC



drivers/i2c/busses/i2c-at91.c

```
static const struct of_device_id atmel_twi_dt_ids[] = {
    {
        .compatible = "atmel,at91rm9200-i2c",
        .data = &at91rm9200_config,
    }, {
        [...]
    }, {
        .compatible = "atmel,at91sam9x5-i2c",
        .data = &at91sam9x5_config,
    }
};
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),
    },
};
```

Découverte par le noyau

- Node `/ahb/apb/i2c@f8014000` a la property `compatible="atmel,at91sam9x5-i2c"`
 - Chaîne présente dans `atmel_twi_dt_ids[]`
 - La fonction `at91_twi_probe()` sera appelée pour initialiser le contrôleur
- Accès aux informations du .dtb (registres, irq, clocks...)
- `platform_get_resource(pdev, IORESOURCE_MEM, 0)`
 - Renvoie un pointeur sur une `struct resource`
 - Contient un champ `dev` qui contient les registres
- `platform_get_irq(pdev, 0)`
- `clk_get(&pdev->dev, NULL)`

#address-cells, #size-cells

- Certaines propriétés commencent par un #
 - `#address-cells`, `#size-cells`
 - `#gpio-cells`, `#interrupt-cells`
 - `#phy-cells`, `clock-cells`, `dma-cells`
- Une *cell* est une valeur de 32 bits
 - eg `<32 12>` est une liste de 2 cells
- Les propriétés `#address-cells`, `#size-cells` :
 - Indiquent le nombre de cells décrivant les registres de la propriété `reg` dans les nodes fils

sama5d4.dtsi

```

/ {
    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>;
                };
            };
        };
    };
};

```

Les interruptions

- Plusieurs propriétés :
- `interrupts` :
 - Indique le numéro de l'interruption
- `interrupt-parent` :
 - Indique à quel contrôleur d'interruption il est lié
 - Pointe sur le phandle d'un autre node
- `interrupt-controller` :
 - Booléen, indique si le node est un contrôleur d'interruptions
- `#interrupt-cells` :
 - Définit combien de cells sont nécessaires dans la propriété `interrupts` des nodes fils

Exercices

- On se base sur le fichier `soc_system.dts` et sur les sources du noyau utilisées en TP
- Explorer le fichier `soc_system.dts` et trouver le label `ledr`
- Quels sont les rôles des différentes propriétés ?
- Trouver dans les sources du noyau le fichier qui instancie ce module
 - On pourra se servir de la propriété `compatible`
- Dans ce fichier, expliquer comment on accède aux registres pour écrire et lire sur le gpio