Systèmes embarqués 5/7

J.-M Friedt, G. Goavec-Mérou

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

17 septembre 2017

Passage de paramètres du module au driver

Dans la plateforme :

```
static struct resource jmf_resources[] = {
        {.start  = 0x378,
         .end    = 0x379,
         .flags  = IORESOURCE_IO, // PAS IORESOURCE_MEM
         .name   = "io-memory1"
        },
        {.start  = 0x37a,
         .end    = 0x37b,
         .flags  = IORESOURCE_IO,
         .name   = "io-memory2"
        },
    };

static struct platform_device *pdev1;

static int __init gpio_simple_init(void)
{pdev1=platform_device_register_simple("jmf",0,jmf_resources,ARRAY_SIZE(jmf_resources));
...
```

Noter les deux derniers arguments de `platform_device_register_simple()` qui passent les ressources au driver.

Passage de paramètres du module au driver

Dans le driver :

```
static int gpio_simple_probe(struct platform_device *pdev)
{struct resource *r1,*r2;
 r1= platform_get_resource(pdev, IORESOURCE_IO, 0);
 r2= platform_get_resource(pdev, IORESOURCE_IO, 1);
 printk(KERN_ALERT "jmf %d %x %x\n",pdev->id,(int)r1->start,(int)r2->start);
...
```

qui se traduit par (dmesg)

```
[  331.353399] jmf 0 378 37a
```

et

```
# cat /proc/ioports
  0378-0379 : io-memory1
  037a-037b : io-memory2
```

⇒ plusieurs configurations possibles pour un même périphérique

Passage de paramètres du module au driver

⇒ un même driver reçoit diverses configurations pour un même périphérique placé à différentes adresses :

Dans le module :

```
static struct resource jmf_resources1[] = {
  {.start=0x378, .end=0x379, .flags=IORESOURCE_IO, .name="io-memory1"},
  {.start=0x37a, .end=0x37b, .flags=IORESOURCE_IO, .name="io-memory2"},
 };
static struct resource jmf_resources2[] = {
  {.start=0x37c, .end=0x37d, .flags=IORESOURCE_IO, .name="io-memory3"},
 };

static struct platform_device *pdev1,*pdev2;

static int __init gpio_simple_init(void)
{pdev1=platform_device_register_simple("jmf",0,jmf_resources1,ARRAY_SIZE(jmf_resources1));
 pdev2=platform_device_register_simple("jmf",1,jmf_resources2,ARRAY_SIZE(jmf_resources2));
...
```

et dans le driver

```
static int gpio_simple_probe(struct platform_device *pdev)
{struct resource *r1,*r2;
 printk("resources: %d",pdev->num_resources);
 r1= platform_get_resource(pdev, IORESOURCE_IO, 0);
 printk(KERN_ALERT "start1 %d %x\n",pdev->id,(int)r1->start);
 if (pdev->num_resources>1)
   {r2= platform_get_resource(pdev, IORESOURCE_IO, 1);
    printk(KERN_ALERT "start2 %d %x\n",pdev->id,(int)r2->start);
   } // $LINUX/Documentation/driver-model/platform.txt
...
```

dmesg :

```
[ 187.705199] resources : 2
[ 187.705200] start1 0 378
[ 187.705653] start2 0 37a
[ 187.706333] resources : 1
[ 187.706334] start1 1 37c
```

et on retrouve les entrées dans `/sys/bus/platform/devices/jmf*`

Problème de la description des périphériques

- pour les bus énumérés, un pilote est chargé lorsque le périphérique est détecté (USB, PCI, firewire)
- pour les autres bus (SPI, I^{2}C, ISA/PC104), description du matériel
- insérer la description dans le code source du noyau $\Rightarrow$ prolifération des sources (un pilote par plateforme) et redondance [1]

```
/* SPI */
static struct spi_board_info pcm037_spi_dev[] = {
        {
                .modalias       = "dac124s085",
                .max_speed_hz   = 400000,
                .bus_num        = 0,
                .chip_select    = 0,            /* Index in pcm037_spi1_cs[] */
                .mode           = SPI_CPHA,
        },
};

/* Platform Data for MXC CSPI */
static int pcm037_spi1_cs[] = {MXC_SPI_CS(1), IOMUX_TO_GPIO(MX31_PIN_KEY_COL7)};

static const struct spi_imx_master pcm037_spi1_pdata __initconst = {
        .chipselect = pcm037_spi1_cs,
        .num_chipselect = ARRAY_SIZE(pcm037_spi1_cs),
};

int __init pcm037_eet_init_devices(void)
{[...]
        /* SPI */
        spi_register_board_info(pcm037_spi_dev, ARRAY_SIZE(pcm037_spi_dev));
        imx31_add_spi_imx0(&pcm037_spi1_pdata);
```

1. linux-4.4.2/arch/arm/mach-imx/mach-pcm037_eet.c

Problème de la description des périphériques

- pour les bus énumérés, un pilote est chargé lorsque le périphérique est détecté (USB, PCI, firewire)
- pour les autres bus (SPI, I^2C, ISA/PC104), description du matériel
- insérer la description dans le code source du noyau ⇒ prolifération des sources (un pilote par plateforme) et redondance

```
From Linus Torvalds <>
Date Thu, 17 Mar 2011 19:50:36 -0700
Subject Re: [GIT PULL] omap changes for v2.6.39 merge window

On Thu, Mar 17, 2011 at 11:30 AM, Tony Lindgren <tony@atomide.com> wrote:
> Please pull omap changes for this merge window from:

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

You need to stop stepping on each others toes. There is no way that
your changes to those crazy clock-data files should constantly result
in those annoying conflicts, just because different people in
different ARM trees do some masturbatory renaming of some random
device. Seriously.
[...]
```

https://lkml.org/lkml/2011/3/17/492

Problème de la description des périphériques

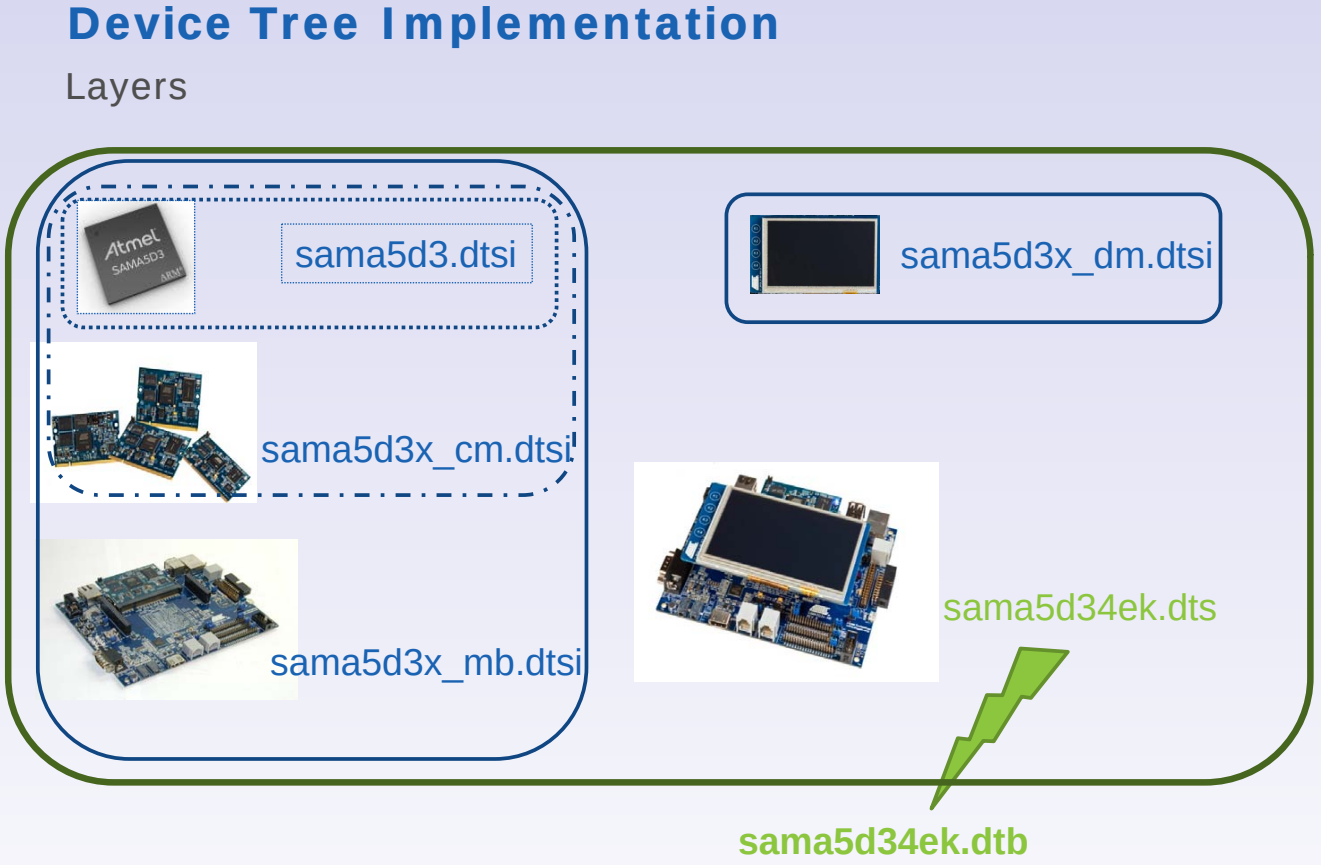
- pour les bus énumérés, un pilote est chargé lorsque le périphérique est détecté (USB, PCI)
- pour les autres bus (SPI, I^2C), description du matériel
- insérer la description dans le code source du noyau ⇒ prolifération des sources (un pilote par plateforme) et redondance
- **Solution** issue de l'architecture PowerPC & SPARC portée à ARM : le *devicetree*[2] pour décrire les périphériques et leurs dépendances

⇒ un fichier ASCII décrivant les périphériques et leurs dépendances, converti en fichier binaire pour exploitation par le noyau Linux lors de son démarrage puis son exécution (overlay)[3][4]

2. `https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt`
3. T. Petazzoni, *Introduction au "device tree" sur ARM*, Opensilicium Janvier-Février-Mars 2016, pp. 46-59, à `jmfriedt.free.fr/devicetree_os19.pdf`
4. T. Petazzoni, *Device Tree for dummies*, ELC 2014 `https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf`, conférence à `https://www.youtube.com/watch?v=uzBwHFjJ0vU`

Architecture du *devicetree*

Hiérarchie de périphériques : SoC=CPU+interfaces, sur une carte de développement [5]



Pas valable sur PC (architectures x86/amd64)

5. Atmel AN-8481, *Linux Device Tree Introduction* (2014), `http://atmel.force.com/support/servlet/fileField?id=0BEG0000000PDgm`

Description du matériel

Exemple de `linux-4.4.2/arch/arm/boot/dts/sun5i-a13-olinuxino-micro.dts`
1. dans le devicetree :

```
#include "sun5i-a13.dtsi"
#include <dt-bindings/pinctrl/sun4i-a10.h>
[...]

&spi2 {   // surchage la definition de SPI precedente
  pinctrl-0 = <&spi2_pins_a>, <&spi2_cs0_pins_a>;
  status = "okay";
  ad9834@0 {
    compatible = "ad9834_2";
    spi-max-frequency = <50000000>; // 5 MHz
    spi-cpol;
    reg = <0>;
    vcc-supply = <&reg_ad9834_vref>;
    freq0 = <136727>;
  };
};

reg_ad9834_vref: vref-reg@1 {
  compatible = "regulator-fixed";
  regulator-max-microvolt = <3300000>;
  [...]
  }
}
```

Lien avec le pilote

Exemple de `linux-4.4.2/arch/arm/boot/dts/sun5i-a13-olinuxino-micro.dts`

1. dans le devicetree :

```
#include "sun5i-a13.dtsi"
#include <dt-bindings/pinctrl/sun4i-a10.h>
[...]

&spi2 {   // surchage la definition de SPI precedente
  pinctrl-0 = <&spi2_pins_a>, <&spi2_cs0_pins_a>;
  status = "okay";
  ad9834@0 {
    compatible = "ad9834_2";
    spi-max-frequency = <50000000>; // 5 MHz
    spi-cpol;
    reg = <0>;
    vcc-supply = <&reg_ad9834_vref>;
    freq0 = <136727>;
  };
};
```

2. et dans le pilote :

```
static const struct spi_device_id ad9834_id[] = {
        {"ad9834_2", ID_AD9834}, {}
};
MODULE_DEVICE_TABLE(spi, ad9834_id);
```

Compilation du devicetree

1. Le devicetree "d'origine" est dans
   `redpitaya/board/redpitaya/zynq-red_pitaya.dts`
   ⇒ modifier ce fichier et compiler son noyau (nécessite les droits d'écriture sur le répertoire buildroot)
2. binaire→ ASCII d'un devicetree existant [6]

   ```
   dtc -I dtb -O dts fichier.dtb
   ```
3. ou `fdtdump zynq-red_pitaya.dtb`
4. ASCII → binaire :

   ```
   dtc -O dtb -o fichier.dtb fichier.dts
   ```

OF=OpenFirmware [7]

6. script `https://git.kernel.org/cgit/utils/dtc/dtc.git` ou `buildroot/output/host/usr/bin/dtc` : penser à prendre le `dtc` de buildroot et non celui de l'hôte
7. `https://www.openfirmware.info/Welcome_to_OpenBIOS`

Ajouter une entrée dans devicetree

```
# insmod composant.ko
```

ne fait rien tant qu'on n'a pas

```
# insmod dummy_platform.ko
```

qui insère le composant dans la plateforme et fait appel au pilote

```
[  838.108990] . Entering probe
[  838.111930] . Registering
[  838.114825] . Registered
```

⇒ ajout d'une entrée pour faire appel à notre pilote depuis le *devicetree*

1. obtenir le *devicetree* courant dans la partition de boot
   ```
   # mount /dev/mmcblk0p1 /mnt/
   # ls /mnt/
   ...  uImage  zynq-red_pitaya.dtb
   ```
2. binaire → ASCII
3. ajouter un nœud faisant appel au pilote (`compatible` = description de la plateforme auparavant)
   ```
   dummy_entry {compatible="composant1";};
   ```
4. ASCII → binaire
5. rebooter la Redpitaya pour tenir compte de la nouvelle configuration

Association pilote-devicetree

Vérifier que le nouveau nœud est inséré : arborescence du *devicetree* accessible dans `/sys/firmware/devicetree/base` :
⇒ présence de `dummy_entry`

```
$ cat compatible
composant1
```

dans le pilote, on ajoute les structures associant l'entrée dans *devicetree* et l'identifiant [8] :

```
static const struct of_device_id composant_id_of[] = { // pour devicetree
  { .compatible="composant1",
    .data=&composant_chip_info_tbl[ID_COMPOSANT12],},{} };
MODULE_DEVICE_TABLE(of, composant_id_of);          // pour devicetree

static struct platform_driver composant_driver = {
  .driver = { .name = "composant",
     .owner = THIS_MODULE,
     .of_match_table=of_match_ptr(composant_id_of), // pour devicetree
  },
  .probe = composant_probe,
  .remove = composant_remove,
  .id_table = composant_id,  // pour platform
};
module_platform_driver(composant_driver);
```

Cette fois, messages de `probe()` dès le chargement du pilote

Passage de paramètres

Une fois le pilote chargé, passage de paramètres depuis le *devicetree* pour décrire la configuration du périphérique

- dans le devicetree :

```
dummy_entry {compatible="composant1";toto32=<56>;};
```

- dans le pilote :

```
static int composant_probe(struct platform_device *pdev)
{u32 val32=42;u16 val16=42;
 struct device_node *np = pdev->dev.of_node;
 of_property_read_u32(np, "toto32", &val32);
 of_property_read_u16(np, "toto16", &val16);
 printk(KERN_ALERT ". Entering probe %u %u\n",val32,val16);
...
```

- vérifier que la variable est prise en compte

```
/sys/firmware/devicetree/base/dummy_entry
# ls
compatible  name        toto32

# hexdump toto32
0000000 0000 3800
0000004

# insmod composant_comm.ko
[  232.968359] . Entering probe 56 42
```

Overlay

- Ajout dynamique d'entrées dans le devicetree : les overlays [9] [10]
- Pas encore supporté dans le noyau linux officiel
- Documentation dans `SRC_LINUX/Documentation/devicetree/overlay-notes.txt`
- Utile lors de l'ajout dynamique de périphériques : IP FPGA et ressources associées décrites dans le *devicetree* [11]
- $\Rightarrow$ utile sur architectures combinant CPU + FPGA (Armadeus Systems, Xilinx Zynq, Intel/Altera SoC

9. M. Fischer, *FPGA Manager & devicetree overlays*, FOSDEM 2016, `https://archive.fosdem.org/2016/schedule/event/fpga_devicetree/`
10. P. Antoniou, *Transactional Device Tree & Overlays – Making Reconfigurable Hardware Work*, ELC 2015 `http://events.linuxfoundation.org/sites/events/files/slides/dynamic-dt-elce14.pdf` et conférence `https://www.youtube.com/watch?v=3Ag7ZBC_Nts`
11. `https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/tree/Documentation/devicetree/bindings/fpga/fpga-region.txt?id=refs/tags/next-20161207`

Overlay

L'ajout d'un overlay (fragments) dans le devicetree déclenche la méthode probe du module correspondant.

```
/dts-v1/;
/plugin/;
/ { compatible = "xlnx,zynq-7000";
    fragment@0
        {target-path = "/";
          __overlay__
          {  #address-cells = <1>;
             #size-cells = <1>;
             pilote_drv {compatible = "pilote_ctl";};
          };
        };
  };
```

qui se compile par

```
BUILDROOT/output/host/usr/bin/dtc -@ -I dts -o pl_ovl.dtbo pl_ovl.dts
```

où "@" indique qu'on compile un overlay
Pour ajouter l'overlay

```
mkdir /sys/kernel/config/device-tree/overlays/pilote
cat pl_ovl.dtbo > /sys/kernel/config/device-tree/overlays/pilote/dtbo
cd /sys/firmware/devicetree/base/pilote_drv/
cat compatible
```

Overlay

Le driver compatible avec cet overlay l'indique par

```
static const struct of_device_id composant_id_of[] = { // pour devicetree
        { .compatible="pilote_ctl", },
        {}
};
MODULE_DEVICE_TABLE(of, composant_id_of);           // pour devicetree

static struct platform_driver pilote_driver = {
        .driver = {.name = "pilote_de_jmf",.owner = THIS_MODULE,
                   .of_match_table=of_match_ptr(composant_id_of),
                   },
        .probe = gpio_simple_probe,
        .remove= gpio_simple_remove,
};

module_platform_driver(pilote_driver);
// fabrique le init() et exit() avec platform_register et platform_unregister

static int gpio_simple_remove(struct platform_device *pdev)
{printk("driver Au revoir\n");return 0;}

static int gpio_simple_probe(struct platform_device *pdev)
{printk("driver Bonjour\n");return 0;}
```

$\Rightarrow$ charger ce driver appelle immédiatement la méthode probe, sans avoir à charger explicitement une plateforme (le devicetree s'en est chargé)

Pour retirer l'overlay

```
rmdir /sys/kernel/config/device-tree/overlays/pilote/
```

Linux et les bitstreams FPGA

- `fpga_manager` (Linux) fournit une interface unifiée pour les SoC
- Vivado (Xilinx) fournit un bitstream en .bit. à convertir en .bin par le SDK Xilinx :

  ```
  $VIVADO_SDK/bin/bootgen -image fichier_bif.bif -arch zynq \
  -process_bitstream bin~
  ```

  avec le fichier de configuration .bif :

  ```
  all: {nom_bitstream.bit}
  ```

  → fichier .bit.bin
- le fichier `bit.bin` est copié dans `/lib/firmware` de la redpitaya.
- pour transférer le bitstream dans le FPGA

  ```
  echo "nom_bitstream.bit.bin" > /sys/class/fpga_manager/fpga0/firmware
  ```

Overlay associé à un bitstream

- par overlay : description cohérente du bitstream, du driver et des ressources

```
/dts-v1/;
/plugin/;
/ { compatible = "xlnx,zynq-7000";
    fragment@0 {
        target = <&fpga_full>;
        #address-cells = <1>;
        #size-cells = <1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;
            firmware-name = "top_redpitaya_axi_gpio_ctl.bin";
            gpio1: gpio@43C00000 {
                compatible = "gpio_ctl";
                reg = <0x43C00000 0x0001f>;
                gpio-controller;
                #gpio-cells = <1>;
                ngpio= <8>;
            };
        };
    };
};
```

Conclusion et mise en pratique

Exercice : modifier le devicetree et un pilote afin de charger automatiquement le pilote à l'insertion du module associé
Fonctions liées au *devicetree* déclarées dans

```
#include <linux/of.h>
#include <linux/of_device.h>
```

Compatibilité platform device + devicetree [12]

```
static int composant_probe(struct platform_device *pdev)
{struct composant_state *st;
 if (pdev->dev.of_node)
     st->chip_info=of_match_device(composant_id_of,&pdev->dev)->data; // dt
 else
     st->chip_info =                                              // plateforme
       &composant_chip_info_tbl[platform_get_device_id(pdev)->driver_data];
```

Adapter le pilote débloquant `read` sur un timer pour recevoir la période depuis une entrée dans le *devicetree*.

Consulter `LINUX/Documentation/devicetree`

12. `http://lxr.free-electrons.com/source/drivers/iio/adc/at91_adc.c` l.1142 : `static int at91_adc_probe(struct platform_device *pdev)`