Examen “électronique embarquée”

J.-M Friedt, G. Goavec-Merou, 8 janvier 2021

Nous avons décrit la distinction entre description d'une plateforme, définissant les ressources requis par un périphérique, et le pilote générique proposant les fonctions pour accéder à ces ressources. Nous avons vu dans la hiérarchie des niveau d'abstraction de Linux comment un pilote peut créer des points de communication uniformisés avec un type de périphérique donné au travers de pseudo-fichiers faisant le lien entre l'espace utilisateur et le noyau dans le répertoire `/sys`. Nous allons ici explorer les interfaces décrivant un générateur d'impulsions de largeur programmable – PWM pour *Pulse Width Modulation*. Nous proposons un squelette de pilote (*driver*) qu'il faudra instancier depuis une plateforme faisant appel à ce pilote, et compléter les fonctions manquant qui sont appelées implicitement (*callback*) lorsqu'un appel système est émis depuis l'espace utilisateur vers le noyau. Les développements génériques initiaux se feront soit sur PC, soit sur Redpitaya – on se rappellera que toute erreur de programmation au niveau du noyau Linux exécuté se traduit par un reboot et le temps perdu à relancer le système et son interface graphique, tandis que le développement sur Redpitaya impose l'environnement de *cross-compilation*. Nous laissons le choix de l'environnement pour se familiariser avec le code proposé, sachant que les dernières étapes (communication sur bus SPI) ne peuvent se faire que sur Redpitaya.

Pour rappel, l'ensemble des outils fournis par Buildroot pour la Redpitaya sont dans `/home/jmfriedt/buildroot-2019.05.1_redpit`. Une archive associée à ce travail est disponible dans `/home/jmfriedt/exam_M2elec_num2021.tar.gz`.

Nous proposons un squelette de pilote (*driver*) dans `ma_pwm1.c`) incomplet et qui ne compile donc pas.

1. compléter le pilote avec l'implémentation des fonctions répondant aux appels systèmes `set_polarity`, `disable`, `enable`, `config`, `free` et `request` selon leur prototype décrit dans les sources du noyau à `include/linux/pwm.h`, aussi disponible à [1]. Nous reproduisons les prototypes de ces fonctions ci-dessous : les arguments des fonctions ne seront pas utiles à cet étape du développement où nous allons simplement vérifier quelle fonction est appelée lors de quel appel système au travers des fichiers dans `/sys` :
   - `int request(struct pwm_chip *chip, struct pwm_device *pwm);`
   - `void free(struct pwm_chip *chip, struct pwm_device *pwm);`
   - `int config(struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns, int period_ns);`
   - `int enable(struct pwm_chip *chip, struct pwm_device *pwm);`
   - `void disable(struct pwm_chip *chip, struct pwm_device *pwm);`
   - `int polarity(struct pwm_chip *chip, struct pwm_device *pwm, enum pwm_polarity polar);`
2. valider le bon fonctionnement de votre implémentation du pilote de la PWM en chargeant le pilote que vous aurez complété de vos appels systèmes, et la description de la plateforme (aucune description de ressources matérielles n'est nécessaire dans cette implémentation, juste l'appel au pilote selon le mécanisme que nous avons étudié). Si le pilote est convenablement chargé, la méthode `probe` a été appelée (le vérifier) et un point de communication `pwmchip0` est créé dans `/sys/class/pwm` et nous pourrons obtenir le contrôle de la PWM numéro 0 par `echo "0" > /sys/class/pwm/pwmchip0/export`. Quelle(s) est(sont) le(s) fonction(s) appelées lors de ces opérations ?
3. modifier la période en écrivant dans `/sys/class/pwm/pwmchip0/pwm0/period` : quelle est la fonction appelée ? l'argument reçu de cette fonction est-il cohérent avec la valeur que nous avons fournie au pilote ?
4. nous avons vu que la méthode proposée par Linux pour homogénéiser la description du matériel est de le décrire dans un *devicetree*. Au lieu de proposer une plateforme faisant appel au pilote, proposer un *overlay* au *devicetree* qui fasse appel au pilote et lui passe un argument `prescaler` qui définisse le facteur de division de l'horloge qui est susceptible de cadencer la PWM. Dans ce cas, le nœud que nous ajoutons dans l'*overlay* est greffé à la racine de l'arbre, donc en fournissant `target-path="/";`

1. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/pwm.h`

5. De nombreuses PWM proposées sous forme de composant sont configurables par SPI [2] [3] [4], principalement pour la commande de l'intensité lumineuse des LEDs. Nous allons modifier le pilote afin de remplacer `platform_device` par un `spi_device`. Ce pilote sera appelé au moyen de l'*overlay* au *devicetree* de la Redpitaya qui décrit la configuration du bus SPI selon :

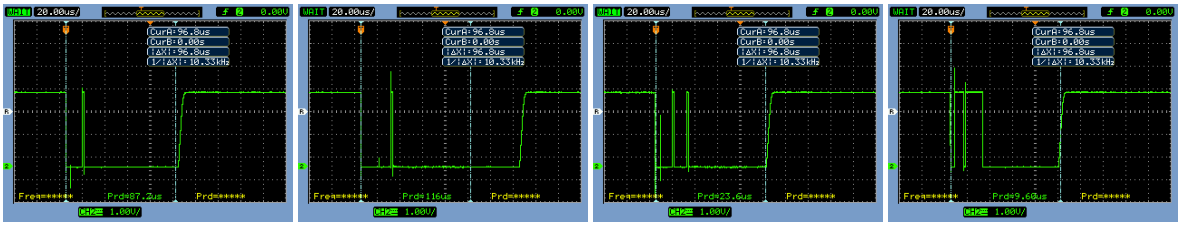
```
/dts-v1/;
/plugin/;
/ { compatible = "xlnx,zynq-7000";
   fragment@0 {
     target = <&spi1>;
     __overlay__ {
        #address-cells = <1>;
        #size-cells = <1>;
        status = "okay";
        num-cs = <1>;
        is-decoded-cs = <0>;
        pwm0: pwm0@0 {
           compatible = "ma_pwm";
           spi-max-frequency = <1000000>;
           reg = <0>;
        };
     };
  };
 };
```

   En effet, le `spi_device` est un nœud du bus SPI qui est nécessairement défini dans le *devicetree*. Nous constatons que contrairement au cas précédent, cet *overlay* n'est pas greffé à la racine mais surcharge le nœud `spi1` défini dans le devicetree de la RedPitaya : `target = <&spi1>;`

   La difficulté du passage au `spi_device` tient en la nécessité d'avoir accès à la configuration du bus SPI dans les diverses fonctions associées à la PWM. En effet, l'argument de chacune de ces fonctions est, par convention de l'API du noyau Linux, une structure décrivant la PWM et non le bus SPI sous-jacent. L'astuce proposée consiste à créer une structure de donnée dédiée à ce nouveau pilote dont la description de la PWM est un sous-ensemble. Ce sous-ensemble sera passé comme argument aux fonctions, et la puissance du C mise en œuvre dans la macro `container_of(nom, struct pwm_wb_pwm, chip);` consiste à être capable de trouver la structure mère contenant la description de la PWM. Comme cette structure mère de type nommé arbitrairement `pwm_wb_pwm` contient par ailleurs la description du bus SPI que nous avons déclarée dans le *devicetree*, nous pouvons faire appel à cette description dans l'appel à `spi_write()` sous la forme `spi_write(pc->spi,&variable, sizeof(variable));`

   Fort de ces connaissances, en vous inspirant de l'exemple 2 fourni dans l'archive, démontrez que vous être capables de transmettre sur le bus SPI1 la période configurée au travers de `/sys/class/pwm/pwmchip0/pwm0/period`

6. vérifier à l'oscilloscope, en transmettant une période au pilote, que vous êtes capables d'observer le mot correspondant sur la sortie MOSI du SPI [5] (on pourra utiliser pour ce faire la fonction `int spi_write(struct spi_device *spi, const void *buf, size_t len)` de Linux.



Captures d'écran d'oscilloscope de MOSI en émettant (de gauche à droite) 1, 256, 257 et 65531

2. `http://www.latticesemi.com/-/media/LatticeSemi/Documents/ReferenceDesigns/SZ/SPISlavetoPWMGeneration-Documentation.ashx?document_id=41302`
3. `http://www.chip-lead.com/wp-content/uploads/2017/12/VAS6685-data-sheet-v1.pdf`
4. `https://www.ti.com/lit/ds/symlink/tlc59711.pdf`
5. le brochage de la Redpitaya est décrit à `https://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html` avec E2 le connecteur le plus proche du bus SATA et la broche 1 dans le coin en bord de carte côté connecteur SATA (empreinte carrée)

Solutions

1. L'objectif de cette première question était de vérifier la capacité à identifier les prototypes des fonctions en cohérence avec les *file operations* (pointeurs de fonctions sur leur implémentation), de déclarer leur implémentation en se contentant de renvoyer un succès (0) et un message indiquant que la fonction a été appelée

```
1  #include <linux/clk.h>
2  #include <linux/err.h>
3  #include <linux/io.h>
4  #include <linux/module.h>
5  #include <linux/of.h>
6  #include <linux/platform_device.h>
7  #include <linux/pwm.h>
8
9  static void mapwm_free(struct pwm_chip *chip, struct pwm_device *pwm);
10
11 static int mapwm_request(struct pwm_chip *chip, struct pwm_device *pwm)
12 {
13    printk("%s\n", __func__);
14    return 0;
15 }
16
17 static int mapwm_config(struct pwm_chip *chip, struct pwm_device *pwm,
18                int duty_ns, int period_ns)
19 {
20    printk("%s␣%d\n", __func__, period_ns);
21    return 0;
22 }
23
24 static int mapwm_enable(struct pwm_chip *chip, struct pwm_device *pwm)
25 {
26    printk("%s\n", __func__);
27    return 0;
28 }
29
30 static void mapwm_disable(struct pwm_chip *chip, struct pwm_device *pwm)
31 {
32    printk("%s\n", __func__);
33 }
34
35 static int mapwm_set_polarity(struct pwm_chip *chip, struct pwm_device *pwm,
36             enum pwm_polarity polarity)
37 {
38    printk("%s\n", __func__);
39    return 0;
40 }
41
42 static const struct pwm_ops pwm_wb_pwm_ops = {
43    .request = mapwm_request,
44    .free    = mapwm_free,
45    .config  = mapwm_config,
46    .enable  = mapwm_enable,
47    .disable = mapwm_disable,
48    .set_polarity = mapwm_set_polarity,
49    .owner = THIS_MODULE,
50 };
51
52 static int mapwm_probe(struct platform_device *pdev)
53 {
54    int ret;
55    struct pwm_chip *chip;
```

```
56
57     chip = devm_kzalloc(&pdev->dev, sizeof(*chip), GFP_KERNEL);
58     chip->dev = &pdev->dev;
59     chip->ops = &pwm_wb_pwm_ops;
60     chip->base = -1;
61     chip->npwm = 1;
62     chip->of_pwm_n_cells = 3;
63     chip->of_xlate = of_pwm_xlate_with_flags;
64     platform_set_drvdata(pdev, chip);
65     ret = pwmchip_add(chip);
66     printk("ret␣:␣%d\n", ret);
67
68     return ret;
69
70 }
71
72 static void mapwm_free(struct pwm_chip *chip, struct pwm_device *pwm)
73 {
74     printk("%s␣%x\n", __func__);
75 }
76
77 static int mapwm_remove(struct platform_device *pdev)
78 {
79     struct pwm_chip *chip = platform_get_drvdata(pdev);
80          printk("Bye");
81     return pwmchip_remove(chip);
82 }
83
84 static const struct of_device_id pwm_wb_pwm_of_match[] = {
85     { .compatible = "ma_pwm", },
86     { /* sentinel */ }
87 };
88 MODULE_DEVICE_TABLE(of, pwm_wb_pwm_of_match);
89
90 static struct platform_driver pwm_wb_pwm_driver = {
91     .driver = {
92        .name = "ma_pwm",
93        .of_match_table = pwm_wb_pwm_of_match,
94     },
95     .probe = mapwm_probe,
96     .remove = mapwm_remove,
97 };
98 module_platform_driver(pwm_wb_pwm_driver);
99 MODULE_AUTHOR("Gwenhael␣Goavec-Merou␣<gwenhael.goavec-merou@trabucayre.com");
100 MODULE_DESCRIPTION("Dummy␣PWM␣driver");
101 MODULE_LICENSE("GPL");
```

qui est suivi du `Makefile` pour valider la compilation ou cross-compilation du pilote (*driver*)

```
1  obj-m += ma_pwm1.o device_alone.o ma_pwm3spi.o
2
3  pc:
4      make -C /usr/src/linux-headers-5.9.0-4-amd64 M=$(PWD) modules
5
6  red:
7      make ARCH=arm CROSS_COMPILE=arm-linux- -C \
8      /t/buildroot-2020.11.1_redpit/output/build/linux-xilinx-v2019.1/ M=$(PWD) modules
9
10 dtb:
11     dtc -I dts -O dtb -@ -o red_pwm-spi.dtb red_pwm-spi.dts
```

Ici `make pc` fabriquer le module pour le PC x86 et `make red` compile le module pour la Redpitaya.

2. un pilote (*driver*) doit être appelé par une description de la plateforme pour que sa méthode `probe()` soit effectivement appelée. Il nous faut donc définir une plateforme – ici très simple en l'absence de ressources à partager avec le pilote – pour appeler le pilote identifié par son `compatible`

```
#include <linux/module.h>
#include <linux/platform_device.h>

static struct platform_device *pdev1;

static int __init simple_init(void)
{printk(KERN_ALERT "Device init");
 pdev1 = platform_device_register_simple("ma_pwm", 0, NULL, 0);
 return 0;
}

static void __exit simple_exit(void)
{printk(KERN_ALERT "Device exit");
 platform_device_unregister(pdev1);
}

module_init(simple_init)
module_exit(simple_exit)
MODULE_DESCRIPTION("GPIO simple driver");
MODULE_LICENSE("GPL");
```

Cette définition de plateforme peut être incluse dans le même fichier que le pilote si nous éliminons `module_platform_driver(pwm_wb_pwm_driver);` pour éviter le doublon des méthodes init et exit. Insérer (`insmod`) la plateforme après le pilote charge effectivement ce dernier en mémoire.

```
$ sudo insmod ma_pwm1.ko
$ sudo insmod device_alone.ko
$ sudo dmesg | tail -2
[121375.755073] Device init
[121375.755132] ret : 0
```

qui confirme la création de `/sys/class/pwm/pwmchip0`. Lorsque nous créons la première PWM (d'indice 0) les fonctions suivantes sont appelées :

```
$ echo "0" > /sys/class/pwm/pwmchip0/export
[121375.755073] Device init
[121375.755132] ret : 0
[121499.297283] mapwm_request
```

On notera que cette opération peut se faire en tant qu'utilisateur avec la règle suivante dans `/etc/udev/rules.d/98-pwm.rules` :

```
SUBSYSTEM=="pwm", RUN+="/bin/chown -R etudiant.etudiant /sys/%p"
```

qui avait déjà été mis en place sur les ordinateurs de travaux pratiques.

3. Finalement, nous communiquons avec la PWM pour la configurer par

```
$ echo "42" > /sys/class/pwm/pwmchip0/pwm0/period
[121659.274319] mapwm_config 42
```

qui donne bien un résultat cohérent avec l'argument transmis à la fonction

4. Au lieu d'instancier explicitement la plateforme dans un module, nous passons par le *devicetree* de la Redpitaya et en particulier un overlay qui fait appel au pilote :

```
/dts-v1/;
/plugin/;
/ {
  compatible = "xlnx,zynq-7000";
```

```
    fragment@0 {
      target = <&spi1>;
        __overlay__ {
          #address-cells = <1>;
          #size-cells = <1>;
          status = "okay";
          num-cs = <1>;
          is-decoded-cs = <0>;
          pwm0: pwm0@0 {
            compatible = "ma_pwm";
            prescaler = <42>;
            reg = <0>;
          };
        };
    };
};
```

Le point important est que l'argument de `compatible` soit la même chaîne de caractères que celle identifiant le pilote. Le `Makefile` ci-dessus fournit la méthode de compilation du devicetree en prenant soin d'utiliser la version de `dtc` fournie par Buildroot et non par l'hôte.

5. Le *devicetree* pour la description sur SPI étant fournie, il suffisait juste d'ajouter dans l'exemple fourni

```
static int pwm_wb_pwm_config(struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns, int period_ns)
{
  struct pwm_wb_pwm *pc=to_pwm_wb_pwm(chip);       // a partir de chip on est on capable de
  spi_write(pc->spi,&period_ns,sizeof(period_ns)); // remonter 'a la structure contenant
                                                   // cet element, et une fois qu'on a la
                                                   // struct on peut trouver ses autres
  printk("Configuration␣%d␣%d\n",period_ns, duty_ns); // elements tels que la config de SPI
  return 0;
}
```

qui permet de bien comprendre comment nous récupérons la structure `pwm_chip`, analysons la structure contenant cette structure, et en extrayons la configuration du bus SPI.

6. Les captures d'écran d'oscilloscope sont fournies dans le sujet initial. On notera qu'il faut transmettre des valeurs *différentes* pour définir la période : en renvoyant la même valeur, le pilote se rend compte qu'il n'a pas besoin de redéfinir la valeur qui reste la même et aucun message n'est transmis sur le bus SPI.