Systèmes embarqués 4

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

21 octobre 2018

Interfaces de communication

- Le noyau accède au matériel et garantit l'intégrité des accès
- L'espace utilisateur (plusieurs processus) accède aux ressources au travers des requêtes au noyau
- Interface "classique" : `/dev` où "tout" est fichier [1]
- Interface uniforme en terme de **périphérique** représenté : `/sys/class` (PWM, IIO, ...) et `/sys/bus` (PCI, USB, SPI ...) : pas de nouvel IOCTL pour chaque périphérique de la même classe
- communication par chaînes ASCII
- contient à la fois les canaux de données (ex-read/write) et de configuration (ex-ioctl)

1. cependant, chaque ioctl etait unique à chaque périphérique, donc 2 DACs devaient être reprogrammés selon les ioctl spécifiques

Plateforme/pilote

- des pilotes sont chargés en mémoire et sont associés à certains périphériques (VID :PID en USB, nom en SPI)
- les pilotes sont appelés par une plateforme, potentiellement plusieurs fois, avec la configuration matérielle acquise lors du boot ou de l'insertion du périphérique,
- $\Rightarrow$ appels **évènementiels** aux pilotes
- `platform` est le périphérique générique : `/sys/devices/platform`

Plateforme

- créée en fonction de la description du périphérique : pour une plateforme générique (`/sys/bus/platform/devices/simple*`)

```
static struct platform_driver gpio_simple_driver = {
        .probe = gpio_simple_probe,
        .remove = gpio_simple_remove,
        .driver = { .name = "simple",
        },
};
```

alors que pour une interface SPI

```
static struct spi_driver ad5624r_driver = {
         .probe = ad5624r_probe,
         .remove = ad5624r_remove,
         .driver = { .name = "ad5624r",
                     .owner = THIS_MODULE,
                    },
         .id_table = ad5624r_id,
};
```

⇒ définition des fonctions appelées lors du chargement (`probe`) et déchargement (`remove`)

Plateforme

Enregistrement de la plateforme auprès du noyau :

```
static struct platform_device *pdev;

static int __init gpio_simple_init(void)
{ret =platform_driver_register(&gpio_simple_driver);
 pdev=platform_device_register_simple("simple",0,NULL,0);
}

static void __exit gpio_simple_exit(void)
{platform_device_unregister(pdev);
 platform_driver_unregister(&gpio_simple_driver);
}

module_init(gpio_simple_init)
module_exit(gpio_simple_exit)
```

fait appel aux méthodes probe (insmod) ou remove (rmmod)
⇒ création de /sys/bus/platform/drivers/simple* et /sys/bus/platform/devices/simple*

Communication avec le pilote

Exemple de définition des points d'entrée de chaque instance du pilote :

```
static DEVICE_ATTR(value1, 0440, show1, NULL); // show
static DEVICE_ATTR(value2, 0440, show2, NULL); // show
static DEVICE_ATTR(value3, 0220, NULL, read1); // store
```

se traduit par

```
# ls /sys/bus/platform/drivers/simple/simple.0/
driver driver_override modalias power subsystem uevent value1
value2 value3
```

dont les actions sont par exemple

```
int show1(struct device *dev, struct device_attribute \
  *attr, char *buf)
{ return sprintf(buf, "Hello World 1\n");}
```

Utilisation intensive des macros :
`linux-headers.../include/linux/device.h` indique

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
 struct device_attribute dev_attr_##_name = __ATTR(_name, →
     ↪_mode, _show, _store)
```

⇒ création de la variable `dev_attr_value` appelée par
`device_create_file(&pdev->dev, &dev_attr_value);` (init) et
`device_remove_file(&pdev->dev, &dev_attr_value);` (exit)

Plateforme v.s pilote

- Un module se charge par `init` et se décharge par `exit`
- un seul module peut être chargé à un instant donné

  `insmod: ERROR: could not insert module xxx.ko: File exists`
- un module ne peut pas recevoir de configuration au moment du chargement par le noyau
- un module peut charger une plateforme :

```
static struct platform_device *pd1,*pd2;

static int __init gpio_simple_init(void)
{pd1=platform_device_register_simple("jmf",0,NULL,0);
 pd2=platform_device_register_simple("jmf",1,NULL,0);
 return(0);
}
```

- la plateforme décrit le matériel

Plateforme v.s pilote

- les plateformes se voient passer un argument

```
static struct platform_driver jmf_driver = {
    .probe  = jmf_probe,
    .remove = jmf_rm,
    .driver = {.name   = "jmf",},
};

static int jmf_rm(struct platform_device *pdev)
{printk(KERN_ALERT "dr bye %d\n",pdev->id);return 0;}

static int jmf_probe(struct platform_device *pdev)
{printk(KERN_ALERT "dr lo %d\n",pdev->id);return 0;}
// $LINUX/Documentation/driver-model/platform.txt
//  * platform_device.id ... the device instance
// number, or else "-1" to indicate there's only one.

static int __init jmf_init(void)
{platform_driver_register(&jmf_driver);return 0;}
```

-
```
[ 1238.117164] platform init  # insmod pl
[ 1240.112258] driver init    # insmod dr
[ 1240.112283] jmf hello 0
[ 1240.112298] jmf hello 1
[ 1262.529881] jmf bye 0      # rrmod pl
[ 1262.529914] jmf bye 1
[ 1262.529922] platform exit  # rmmod dr
```

Séparation module/pilote

- un module charge plusieurs instances d'une plateforme (*device*)

```
static struct platform_device *p1, *p2;

static int __init gpio_simple_init(void)
{p1=platform_device_register_simple("jmf",0,NULL,0);
 p2=platform_device_register_simple("jmf",1,NULL,0);
...
```

- le *driver* ne contient que la définition du pilote : les méthodes init et exit ne contiennent que `platform_driver_register();` et `platform_driver_unregister();`
- $\Rightarrow$ définition **implicite** des méthodes `init` et `exit` du module par `module_platform_driver(gpio_simple_driver);` avec

```
static struct platform_driver gpio_simple_driver = {
        .probe          = gpio_simple_probe,
        .remove         = gpio_simple_remove,
        .driver = {.name   = "jmf",},
};

module_platform_driver(gpio_simple_driver);
```

Protection des accès par `mutex`

Si par exemple la valeur renvoyée par une lecture est celle transmise au cours d'une écriture : penser à protéger lecture et écriture par un *mutex*.

```
#include <linux/mutex.h>
struct mutex mymutex;

mutex_init(&mymutex);

mutex_lock(&mymutex);   // bloque
...
mutex_unlock(&mymutex); // debloque
```

Application

- Partant de l'exemple de module avec *timer* en `/dev`, implémenter une méthode `read` qui se débloque à intervalles de temps réguliers (utiliser un mutex)
- Proposer le programme C correspondant qui affiche un message à intervalle de temps déterminé par le *timer*
- Remplacer `/dev/` par une plateforme dans `/sys`

Pour chaque étape, effectuer la démonstration **sur PC** puis sur carte **Redpitaya**.