Embedded systems 4/7

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at `jmfriedt.free.fr`

September 6, 2020

Communication interfaces

- The kernel has access to hardware and enforces consistency
- Userspace (multiple processes) access hardware resources through system calls to the kernel
- "Classical" interface: `/dev` following the "Everything is a file" strategy [1]
- Uniform interface representing a given **peripheral class**: `/sys/class` (PWM, IIO, ...) and `/sys/bus` (PCI, USB, SPI ...): no new IOCTL call for each peripheral configuration within the same class
- communication through ASCII sentences
- includes both data channels (ex-read/write) and configurations (ex-ioctl)

[1] however, `ioctl` calls are unique to each peripheral, so that 2 DACs from different manufacturers might be programmed with dedicated `ioctl` calls

Device/driver

Character modules

- cannot dynamically change configuration (e.g. peripheral address)
- only one copy of a given module can be loaded

Need for a solution for dynamically attached hardware

- drivers are loaded and associated with a set of peripherals (VID:PID for USB, name for SPI)
- drivers are called by devices, possibly multiple times, with a hardware configuration documented during the boot process or when inserting the peripheral
- $\Rightarrow$ **event driven** call to drivers
- `platform` is a generic peripheral device: `/sys/devices/platform`

Driver

- created depending on the description of the peripheral: for a generic platform (/sys/bus/platform/drivers/*simple* [2])

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

while for an SPI interfaced peripheral

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

⇒ definitions of the functions called when the driver is loaded (probe) and unloaded (remove)

[2] plateformes/driver_alone.c

Platform device

Registering a platform with the kernel:

```
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)
```

call `probe` (`insmod`) or `remove` (`rmmod`) methods
⇒ `/sys/bus/platform/drivers/simple*` and
`/sys/bus/platform/devices/simple*` created "automatically"

Communication with the driver

Example of the definition of the entry points for each copy of the driver:

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

resulting in

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

whose actions are defined, for example, by

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

Extensive use of macros:
`linux-headers.../include/linux/device.h` hints at

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

⇒ creates the variable `dev_attr_value` called by
`device_create_file(&pdev->dev, &dev_attr_value);` (init) and
`device_remove_file(&pdev->dev, &dev_attr_value);` (exit)

Device v.s driver

- A module is loaded with `init` and unloaded with `exit`
- only a single module can be loaded at any given time

```
insmod: ERROR: could not insert module xxx.ko: File exists
```

- a module cannot be given a configuration when loaded by the kernel
- a module can load one or multiple device(s):

```
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);
}
```

- the platform device describes the hardware configuration
- the driver can be defined in a separate module

```
static struct platform_driver simple_driver = {
        .probe          = simple_probe,
        .remove         = simple_remove,
        .driver = {.name   = "jmf",},
};

static int __init simple_init(void)
{platform_driver_register(&simple_driver);return 0;}
```

Device v.s driver

- drivers are given arguments

```
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      # rmmod pl
[ 1262.529914] jmf bye 1
[ 1262.529922] platform exit  # rmmod dr
```

Device/driver separation

- one module loads multiple copies of devices

```
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);
...
```

- the driver module only defines the actions performed by each device: the init and exit functions only include `platform_driver_register();` et `platform_driver_unregister();`
- $\Rightarrow$ **implicit** definition [3] of the module `init` and `exit` when using `module_platform_driver();` with

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

module_platform_driver(gpio_simple_driver);
```

[3] plateformes/driver_alone2.c

Resource protection with `mutex`

If for example the value returned when reading is the one transmitted during writing, remember to protect the reading and writing functions with a *mutex*.

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

mutex_init(&mymutex);

mutex_lock(&mymutex);   // block
...
mutex_unlock(&mymutex); // unblock
```

Application

- Based on the module using a *timer* and communicating through `/dev`, implement a `read` function that periodically (once every second) unblocks a userspace reading action (using a mutex)
- Write the corresponding C function displaying a message every time its read function has been unblocked by the kernel timer
- Replace `/dev/` with a platform device communicating through `/sys`

At each step, demonstrate on **the PC** and then the **Redpitaya** board or emulator.