Writing our own device driver

In the previous chapter, we made use of the GPIO subsystem of the kernel to manage a LED, now we'll go further to manage the GPIOs from the kernel using a dedicated driver. Actually, what we are going to write is not properly a driver for a real device, but we can use it in order to show you how a complex kernel functionality can be abstracted as a file.

Let's suppose we need to count some pulses that arrive on our BeagleBone Black in a certain amount of time; in this case, we can use one GPIO for each pulse source. We can also consider that the maximum possible pulse frequency is really low (max 50 Hz).

> **TIP**
>
> Note that this situation is quite common, and it can be found in some counter devices. In fact, these devices simply count quantities (water or oil liters, energy power, and so on), and return the counting as frequency modulated pulses.

In this situation, we can use a really simple kernel code to implement a new devices class under the `sysfs` filesystem that we can use to abstract these measurements to the user space. Using our new driver, the user will see a new class named `pulse` and a new directory per device, where you can read the actual counting:

Here is a simple example of the final result:

```
root@BeagleBone:~# tree -l -L 2 /sys/class/pulse/
/sys/class/pulse/
|-- oil -> ../../devices/virtual/pulse/oil
|   |-- counter
|   |-- counter_and_reset
|   |-- gpio
|   |-- power
|   |-- set_to
|   |-- subsystem -> ../../../../class/pulse  [recursive, not fol
|   `-- uevent
`-- water -> ../../devices/virtual/pulse/water
    |-- counter
    |-- counter_and_reset
    |-- gpio
    |-- power
    |-- set_to
    |-- subsystem -> ../../../../class/pulse  [recursive, not fol
    `-- uevent

6 directories, 10 files
```

> **TIP**
>
> Note that the tool tree can be installed by using the following command:
>
> ```
> root@BeagleBone:~# aptitude install tree
> ```

In the preceding example, we have two pulse devices named `oil` and gpio, and set_to (the other files named power and subsystem are not of interest to us).

You can now use the counter file to read the counting data; while using the counter_and_reset file, you can do the same as with the counter file, but after reading the data, the counter is automatically reset to the 0 value. Using the set_to file, you can initialize the counter to a specific value different from 0, while the gpio file is simply an information file that displays the BeagleBone Black's GPIO pin number used to get the pulse signal.

Now, before we continue to describe the driver, let me explain the code.

We have four files and the first one is the Makefile, as shown in the following code:

```
KERNEL_DIR := .         # to be set from the command line
PWD := $(shell pwd)
CROSS_COMPILE = arm-linux-gnueabihf-

obj-m = pulse.o
obj-m += pulse-bbb.o

all: modules

modules clean:
    $(MAKE) -C $(KERNEL_DIR) ARCH=arm CROSS_COMPILE=$(CROSS_COMPI
        SUBDIRS=$(PWD) $@
```

TIP

The code is stored in the chapter_05/pulse/Makefile file in the book's example code repository.

As we can see, it's quite similar to the one presented in the *Device drivers* section of Chapter 3, *Compiling versus Cross-compiling*, the only difference is the obj-m variable; in fact, this time, it declares two object files: pulse.o and pulse-bbb.o.

The pulse-bbb.o file can obviously be obtained by compiling the pulse-bbb.c file that holds the definition of the two pulse devices named oil and water. In fact, as we can see in the following snippet, in the first line of the snippet such devices are defined using the following code:

```
/* Declare all the needed pulse devices */
static struct {
        int gpio;
        char *name;
} pulse_data[] = {
        {
            .gpio = 67,
            .name = "oil",
        },
        {
            .gpio = 68,
            .name = "water",
        },
};
```

TIP

The complete code is stored in the chapter_05/pulse/pulse-bbb.c file in the book's example code repository.

Here, we declare that the oil device is connected with the gpio67 (**P8.8**) while the water one is connected with gpio68 (**P8.10**).

In the first_pulse_init() function (which is executed when the module is first loaded into the kernel), we request the pulse devices by calling the pulse_device_register() function once per device. As an opposite action, during the module unloading (that is, when first_pulse_exit() is called), we release the allocated devices by calling the pulse_device_unregister() function once per device.

TIP

Note that when we write the kernel code, we must release whatever we ask the kernel to, otherwise we'll lose it till the next reboot.

These two functions are implemented in the `pulse.c` file, as shown in the following code snippet:

```
struct pulse_device *pulse_device_register(char *name, int gpio)
{
        struct pulse_device *pdev;
        int ret;

        /* First allocate a new pulse device */
        pdev = kmalloc(sizeof(struct pulse_device), GFP_KERNEL);
        if (unlikely(!pdev))
                return ERR_PTR(-ENOMEM);

        /* Create the device abd init the device's data */
        pdev->dev = device_create(pulse_class, NULL, gpio, pdev,
        if (unlikely(IS_ERR(pdev->dev))) {
                dev_err(pdev->dev, "unable to create device %s at
                ret = PTR_ERR(pdev->dev);
                goto error_device_create;
        }
        dev_set_drvdata(pdev->dev, pdev);
        pdev->dev->release = pulse_device_destruct;

        strncpy(pdev->name, name, PULSE_NAME_LEN);
        pdev->gpio = gpio;

        atomic_set(&pdev->counter, 0);
        pdev->old_status = -1;

        /* Then request GPIO */
        ret = gpio_request(gpio, pdev->name);
        if (ret) {
                dev_err(pdev->dev, "unable to request gpio %d\n",
                goto error_gpio_request;
        }
        gpio_direction_input(gpio);

        /* And as last task start the kernel timer */
        setup_timer(&pdev->ktimer, pulse_ktimer_event, (unsigned
        mod_timer(&pdev->ktimer, jiffies + KTIMER_FREQ);

        dev_info(pdev->dev, "pulse %s added on gpio%d\n",
                                pdev->name, pdev->gpio);

        return pdev;

error_gpio_request:
        device_destroy(pulse_class, 0);

error_device_create:
        kfree(pdev);

        return ERR_PTR(ret);
}
EXPORT_SYMBOL(pulse_device_register);

void pulse_device_unregister(struct pulse_device *pdev)
{
        /* Drop all allocated resources */

        del_timer_sync(&pdev->ktimer);
        gpio_free(pdev->gpio);

        device_destroy(pulse_class, pdev->gpio);

        dev_info(pdev->dev, "pulse %s removed\n", pdev->name);
}
EXPORT_SYMBOL(pulse_device_unregister);
```

You can see all the steps done to create the driver data structures in the `register` function and the respective inverse steps done in the `unregister` one. Note that both the functions are declared as **exported symbols** by the code:

```
EXPORT_SYMBOL(pulse_device_register);
EXPORT_SYMBOL(pulse_device_unregister);
```

This tells the compiler that these functions are special, and they cannot be used by other kernel modules unless the functions are exported.

> **TIP**
>
> This is another important concept that you should understand as-is, otherwise you can take a look the book *Linux Device Drivers, Third Edition* available at the bookshop and online at http://lwn.net/Kernel/LDD3/ (http://lwn.net/Kernel/LDD3/).

In the module initialization method (the `pulse_init()` function), we use the `class_create()` function to create our new pulse class, and as an opposite action, in the module exit (the `pulse_exit()` function), we will destroy it by calling `class_destroy()`.

You should now take a look at the `pulse_init()` function at line:

```
pulse_class->dev_groups = pulse_groups;
```

```
static struct attribute *pulse_attrs[] = {
        &dev_attr_gpio.attr,
        &dev_attr_counter.attr,
        &dev_attr_counter_and_reset.attr,
        &dev_attr_set_to.attr,
        NULL,
};
```

Each entry of the preceding structure is created by the `DEVICE_ATTR_XX()` function as follows:

```
static ssize_t gpio_show(struct device *dev,
       struct device_attribute *attr, char *buf)
{
    struct pulse_device *pdev = dev_get_drvdata(dev)

    return sprintf(buf, "%d\n", pdev->gpio);
}
static DEVICE_ATTR_RO(gpio);
```

This code specifies the attributes of the `dev_attr_gpio.attr` entry by declaring the file attribute `gpio` as read-only, and the `gpio_show()` function body is called each time from the user space when we do a `read()` system call on the file. In fact, as there are `read()` and `write()` system calls for files, similarly there are `show()` and `store()` functions for `sysfs` attributes.

As a dual example, the following code declares the attributes of the `dev_attr_set_to.attr` entry by declaring the `set_to` file attribute as write-only and the `set_to_store()` function body is called each time from the user space. We do a `write()` system call on the file:

```
static ssize_t set_to_store(struct device *dev,
                      struct device_attribute *attr,
                      const char *buf, size_t count)
{
        struct pulse_device *pdev = dev_get_drvdata(dev);
        int status, ret;

        ret = sscanf(buf, "%d", &status);
        if (ret != 1)
                return -EINVAL;

        atomic_set(&pdev->counter, status);

        return count;
}
static DEVICE_ATTR_WO(set_to);
```

TIP

Note that the `sprintf()` and `sscanf()` functions, which are quite common functions for C programmers, are not the ones implemented in the libc, but they are homonym functions written ad hoc for the kernel space to simplify the kernel code development by representing well-known functions to the developer.

You should also notice that for the `show()` and `store()` functions, we have the following:

- The attribute files are the ones that get/set the data from/to the user space by reading/writing the data into the buffer pointed by the `buf` pointer.
- All these functions work on the `dev` pointer that represents the device, which is currently accessed that is, if the user gets access to the `oil` device, the `dev` pointer will point to a data structure, representing such a device. Remember what we said about the object-oriented programming. This magic allows the developer to write a clean and compact code.

The core of our driver is stored in the `pulse_device_register()` and `pulse_ktimer_event()` functions. As we can see, in the former function, we first create a new device according to the data passed:

```
    /* Create the device abd init the device's data */
    pdev->dev = device_create(pulse_class, NULL, gpio, pdev, "%s",
    if (unlikely(IS_ERR(pdev->dev))) {
            dev_err(pdev->dev, "unable to create device %s at gpio
                                                  name, gpio);
            ret = PTR_ERR(pdev->dev);
            goto error_device_create;
    }
```

Then, we request the needed GPIO line to the kernel and set it as an input line:

```
        goto error_gpio_request;
}
gpio_direction_input(gpio);
```

TIP

For more information on the functions used, you can refer to the appropriate section of the kernel source tree documentation directory in the `Documentation/gpio/gpio.txt` file.

In the end, we schedule a kernel timer, which in turn will call the `pulse_ktimer_event()` function:

```
/* And as last task start the kernel timer */
setup_timer(&pdev->ktimer, pulse_ktimer_event, (unsigned long)
mod_timer(&pdev->ktimer, jiffies + KTIMER_FREQ);
```

A kernel timer is a kernel mechanism used to schedule an event at a later time; in our code, we schedule the next event from now (the current value of the `jiffies` variable) plus `KTIMER_FREQ` (defined in the file at the beginning), which means that we'll schedule the event at 10 ms (1/100 seconds) in the future. When the kernel timer expires, the `pulse_ktimer_event()` function is called, which is shown as follows:

```
static void pulse_ktimer_event(unsigned long ptr)
{
        struct pulse_device *pdev = (struct pulse_device *) ptr;

        /* Get the gpio status */
        int status = !!gpio_get_value(pdev->gpio);

        /* Check for the first event  */
        if (pdev->old_status == -1) {
                pdev->old_status = status;
                goto end;
        }

        /* Check for the state changing and, in case, increment t
        if (pdev->old_status != status) {
                pdev->old_status = status;
                atomic_inc(&pdev->counter);
        }

        end:
        /* Reschedule the kernel timer */
        mod_timer(&pdev->ktimer, jiffies + KTIMER_FREQ);
}
```

What we do in this function is quite clear: first, we get the GPIO status, then we check whether such a status has been changed, and in this case, we increment our counter. In the last step, we reschedule the kernel timer using the `mod_timer()` function (this is because the timer is already created, and we need to modify the expiring time only).

Note that this solution is far from the optimum; in fact, we should use interrupts instead of the kernel timers. However, again, this is out of the scope of this book, and so we decided to use the simplest solution to avoid the interrupts usage. Now to test the code, we should compile it, so let's use the following command:

```
$ make KERNEL_DIR=../linux-dev/KERNEL/
```

If everything works well, we should get the two kernel modules `pulse.ko` and `pulse-bbb.ko`, we defined in the `Makefile`.

TIP

Note that `KERNEL_DIR` points to the directory where the kernel sources are, so you should set it accordingly to your system configuration.

Note that we just compiled the code against the kernel code that we rebuilt from scratch in the *Compiling the kernel* section of Chapter 3, *Compiling versus Cross-compiling*, that is, the kernel version 3.13.x, which we stored in the microSD. So, to load these kernel modules into our BeagleBone Black, we must use the distribution in the microSD. To be sure that we are running the right kernel, we can use the following command that shows the current kernel release:

TIP

Of course, we can do the same on the distribution on the on-board eMMC, but in this case, we must compile the modules against the kernel version 3.8.x. To do so, we should first verify the kernel version on the eMMC on the BeagleBone Black:

```
root@BeagleBone:~# uname -r
3.8.13-bone47
```

Then, we should go to the directory where we downloaded the kernel sources into the host system and then retrieve the right kernel release using the following command:

```
$ git checkout -b 3.8.13-bone47 3.8.13-bone47
```

It may happen that we get an error like this:

```
error: Your local changes to the following files
    build_kernel.sh
    patches/defconfig
Please, commit your changes or stash them before
Aborting
```

In this case, we can fix it using the following command:

```
$ git reset --hard
```

Then, we retry the checkout command:

```
$ git checkout -b 3.8.13-bone47 3.8.13-bone47
Switched to a new branch '3.8.13-bone47'
```

Then, we have to rebuild the kernel:

```
$ ./build_kernel.sh
```

When finished, the kernel tree is ready to be used to compile against the new kernel modules.

So, let's copy the two files to the BeagleBone Black (for instance, using the `scp` command), and then first, load the `pulse.ko` module with the following command:

```
root@BeagleBone:~# insmod pulse.ko
```

If we take a look at the kernel messages with `dmesg`, we should see the following message:

```
Pulse driver support v. 0.50.0 - (C) 2014 Rodolfo Giometti
```

Great! The new device class is now defined in the kernel; in fact, if we take a look at the `/sys/class/` **sysfs** directory, we can see that the new class is up and running:

```
root@BeagleBone:~# ls -ld /sys/class/pulse/
drwxr-xr-x 2 root root 0 Jan  8 22:10 /sys/class/pulse/
```

Now we should add the two devices `oil` and `water` defined in the `pulse-bbb.ko` module, so let's load it into the kernel:

```
root@BeagleBone:~# insmod pulse-bbb.ko
```

Again, using the `dmesg` command, we should see two new kernel messages:

```
pulse oil: pulse oil added on gpio67
pulse water: pulse water added on gpio68
```

This is what we expected! Now in the **sysfs**, we now have the following output:

```
root@BeagleBone:~# ls -l /sys/class/pulse/
total 0
lrwxrwxrwx 1 root root 0 Jan  8 22:15 oil -> ../../devices/virtua
lrwxrwxrwx 1 root root 0 Jan  8 22:15 water -> ../../devices/virt
```

Well, this is quite simple, we can use the other two GPIOs as pulse generators and the script in the `chapter_05/pulse_gen.sh` file in the book's example code repository to actually generate the pulses.

We connect `gpio65` (**P8.18**) to `gpio67` (**P8.8**), and in another terminal window, we run the preceding script with the following command line:

```
root@BeagleBone:~# ./pulse_gen.sh 65 4
```

We generate a 4 Hz pulse signal from `gpio65` (**P8.18**) to `gpio67` (**P8.8**), where the `oil` counter is connected, so if we try to read its data, we get the following:

```
root@BeagleBone:~# cat /sys/class/pulse/oil/gpio
67
```

If we try to read the counter file, we can see that it increments at the speed of 4 pulses per second. However, the functioning may result more clear if we use the following commands that reset the counter first (using the `set_to` file), and then use the `counter_and_reset` file to restart the counting after each reading:

```
root@BeagleBone:~# echo 0 > /sys/class/pulse/oil/set_to ; while s
8
8
9
8
```

TIP

Note that we get 8 instead of 4 because the pulse driver counts both high-to-low and low-to-high transactions. Also, note that 9 is due the fact that there can be some delays in reading the counting data.

Now the user can use another GPIO in order to play with the `water` counter.

PREV
The modutils

NEXT
Summary