Articles » General Reading » Hardware & System » Device Drivers

Linux Platform Device Driver

Nagaraj Krishnamurthy, 27 Feb 2016 GPL3

4.17 (4 votes)

This tip will go into the details of platform devices and their corresponding drivers in Linux

- **Download source (TAR) - 10 KB**
- **Download source (ZIP) - 1.6 KB**

Introduction

A platform device is one which is hardwired on the board and hence not hot-pluggable. The driver for this device need not check for the presence of the device and can just go on and do what is required to enable the device to make it operational. If the device is not found, the driver is simply ignored.

Background

Platform device driver works a bit differently than normal hot-pluggable device driver. Platform device is either directly mounted on the board or part of the SoC and hence can't be physically removed. Compared to hot-pluggable devices, platform device might require some extra settings like gpios pin muxing, power on & reset using gpio control, initializing the clock source, etc., which are very specific to the board or SoC - hence the name platform device driver. This tip gives a brief overview of how to write a platform device driver for Linux.

Using the Code

The source code for this article is given for reference. The platform device code is usually part of the BSP/CSP. The device driver code is written as a separate module in the appropriate *linux-kernel-src/driver/* directory (char for serial device, net for networking device, etc). In the given example code, both the platform device and platform driver code are presented in a single file for easy viewing of the code.

Platform Data

This structure is specific to the platform device. As indicated in the previous section, since platform device is usually part of the BSP/CSP, this data structure apart from class specific device data, will contain platform specific data such as gpios used, irq number (if interrupt is used), clock frequency settings required by the platform device etc. It usually implements functions that can be called by the device driver for doing platform specific stuff. This way, the device driver can be made independent of the platform and hence can be used across multiple platforms. For example, the below example implements functions for power ON/OFF the device and for doing the device reset.

```
/* Data structure for the platform data of "my device" */
struct my_device_platform_data {
    int reset_gpio;
    int power_on_gpio;
    void (*power_on)(struct my_device_platform_data* ppdata);
    void (*power_off)(struct my_device_platform_data* ppdata);
    void (*reset)(struct my_device_platform_data* pdata);
};

/* "my device" platform data */
static struct my_device_platform_data my_device_pdata = {
    .reset_gpio         = 100,
    .power_on_gpio      = 101,
    .power_on           = my_device_power_on,
    .power_off          = my_device_power_off,
    .reset              = my_device_reset
};
```

Registering Platform Device

This is done by calling `platform_device_register()` function with the device instance as shown below. The name of the device is very important as this `string` is used by the OS for calling the `probe()` function when the corresponding driver is installed. Please note that this function must be called before registering Platform Driver.

```
static struct platform_device my_device = {
    .name                 = "my-platform-device",
    .id                   = PLATFORM_DEVID_NONE,
    .dev.platform_data    = &my_device_pdata
};

platform_device_register(&my_device);
```

Platform Driver

The platform driver implements a probe function that is called by the OS when this driver is inserted. Since the driver is supposed to be platform independent, it depends on the `platform_data` functions to set up the device and make it operational.

```
static struct platform_driver my_driver = {
    .probe      = my_driver_probe,
    .remove     = my_driver_remove,
    .driver     = {
        .name     = "my-platform-device",
        .owner    = THIS_MODULE,
        .pm       = &my_device_pm_ops,
    },
};
```

Registering Platform Driver

Here is the function for registering driver with the OS. Please note that we are using `platform_driver_probe()` instead of `platform_driver_register()`, since we know that this device is present for sure in the system. The difference between these two functions is that with `platform_driver_register()`, we are asking the OS to put this driver in the list of drivers it maintains for doing device to driver matching when the devices come in/out of the system. Since platform devices are either always present always absent in system (and not-hot-pluggable), we don't need to put our platform driver in the OS driver list. With `platform_driver_probe()`, we are asking the OS to check if a platform device is present with the matching name. If the device is present in the system, the corresponding `probe()` function is called. If not present, the driver is simply ignored.

```
ret = platform_driver_probe(&my_driver, my_driver_probe);
```

Platform Device and Driver binding

When the `platform_driver_probe()` function is called, the operating system scans through the list of available platform devices and checks if the `driver.name` matches with the device name. If yes, it calls the driver probe function with the platform data. The probe function can then initialize the clock settings, power-on, reset the device, allocated the driver data and register the corresponding class specific driver with the OS.

```
static int my_driver_probe(struct platform_device *pdev)
{
    struct my_device_platform_data *my_device_pdata;
    struct my_driver_data* driver_data;

    printk(KERN_ALERT " %s\n", __FUNCTION__);

    my_device_pdata = dev_get_platdata(&pdev->dev);

    /* Power on the device. */
    if (my_device_pdata->power_on) {
        my_device_pdata->power_on(my_device_pdata);
    }

    /* wait for some time before we do the reset */
    mdelay(5);

    /* Reset the device. */
    if (my_device_pdata->reset) {
        my_device_pdata->reset(my_device_pdata);
    }

    /* Create the driver data here */
    driver_data = kzalloc(sizeof(struct my_driver_data),
GFP_KERNEL);
    if(!driver_data)
        return -ENOMEM;
    /* Set this driver data in platform device structure */
    platform_set_drvdata(pdev, driver_data);

    /* Call the class specific register driver here */
    // tty_register_driver(ttyprintk_driver); // for UART driver
    // register_netdev(net_driver); // for Net driver
    // sdio_register_driver(sdio_river)

    return 0;
}
```

License

This article, along with any associated source code and files, is licensed under The GNU General Public License (GPLv3)

Share

About the Author

Nagaraj Krishnamurthy
India

No Biography provided

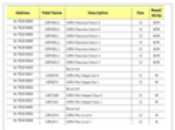
You may also be interested in...

Windows Device Drivers

The Path From COBOL to Mobile

A Simple Driver for Linux OS

Visual COBOL New Release: Small point. Big deal



Simple I/O device driver for RaspberryPi

SAPrefs - Netscape-like Preferences Dialog

Comments and Discussions

0 messages have been posted for this article Visit **http://www.codeproject.com/Tips/1080177/Linux-Platform-Device-Driver** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink | Advertise | Privacy | Terms of Use | Mobile
Web02 | 2.8.160615.1 | Last Updated 27 Feb 2016

Select Language ▼

Article Copyright 2016 by Nagaraj Krishnamurthy
Everything else Copyright © CodeProject, 1999-2016