American Journal of Embedded Systems and Applications
2022; 9(1): 1-5
http://www.sciencepublishinggroup.com/j/ajesa
doi: 10.11648/j.ajesa.20220901.11
ISSN: 2376-6069 (Print); ISSN: 2376-6085 (Online)

Design on Linux Platform Driver for Embedded Systems

Mei Rifei[1], Xiao Laisheng[2, *]

[1]Shenzhen Edge Medical Technology Co., Ltd. Shenzhen, China

[2]School of Mathematics and Computer, Guangdong Ocean University, Zhanjiang, China

Email address:

sun_of_may@qq.com (Mei Rifei), xiaolaisheng@163.com (Xiao Laisheng)

[*]Corresponding author

To cite this article:

Mei Rifei, Xiao Laisheng. Design on Linux Platform Driver for Embedded Systems. *American Journal of Embedded Systems and Applications.* Vol. 9, No. 1, 2022, pp. 1-5. doi: 10.11648/j.ajesa.20220901.11

Received: February 11, 2022; **Accepted**: March 4, 2022; **Published**: March 12, 2022

Abstract: In the development of embedded systems, driver design is one of its core technologies. In the driver design, Linux driver occupies an important position. For the design of Linux driver, platform model is an important driver design method which is introduced after Linux 2.6. This paper first introduces the driving principle and architecture of Linux platform model, and describes the device, driver and device registration and unloading of the platform model in detail. Then, the driver code of watchdog platform in Linux kernel is analyzed. Finally, taking the embedded development environment tiny4412 as an example, a driver design example of Linux platform is given. The platform driver architecture has the characteristics of reusing framework code, strong independence of device resources and drivers, simple code, unified kernel interface, easy maintenance and expansion. In the development of specific drivers, we only need to focus on keeping the underlying device operation function set corresponding one by one with the kernel interface provided by the driver structure, and ensuring device.name and driver.name consistent, and making the platform device registered in the kernel space before the platform driver, which can make the driver run well and stably, greatly reduce the work intensity and shorten the development time of new products. Compared with the traditional device driver mechanism, the Linux platform driver mechanism registers the resources of the device into the kernel which is managed by the kernel, and driver uses these resources by applying standard interface provided by platform_device, which improves the independence of driver and resource management, and has better portability and security. The developing test shows that the driver based on this architecture has good portability, maintainability and scalability.

Keywords: Linux Platform Driver, Embedded System, Ttiny4412

1. Introduction

Device driver is a collection of functions and data structures. Its purpose is to implement an interface for managing devices. The kernel uses this interface to request the driver to control the device. With the continuous progress of technology, the number of devices supported by the system is increasing, the topology of the system is becoming more and more complex, and the support requirements for plug and play are also higher and higher. Device drivers based on Linux [1] have a wide range of development applications, such as developing real-time applications [2], embedded SCADA and RFID systems [3], Linux Optimization Technique [4], non-contact infection screening systems [5], keyboard driver [6], GPIB instrument [7], PCI synchronous clock card [8], CAN driver [9], HI3210 driver [10], USB card reader [11], numerical control system [12], sensor equipment [13] etc.

The traditional device driver design is difficult to meet the needs of this situation, so a new driver management and registration mechanism based on platform is introduced into Linux 2.6 kernel. Platform is a kind of virtual bus, which is used to connect SOC integrated resources to CPU bus in embedded system. It uses the object-oriented idea to complete the abstraction from device driver to bus and core layer.

On Linux in the platform, bus, device and driver are called bus device driver model, that is, a framework is designed for the same kind of equipment. The core layer of the framework realizes some common functions of this kind of device, and the programmer does not need to implement it by himself. The separation of driver and device makes the writing of host controller driver and peripheral driver parallel, and they are no longer related to each other, realizing the idea of layering and separation, so as to improve the independence and portability of the driver. Bus binds the device and driver to the system every time it registers a device, it will automatically find the matching driver. Similarly, when registering a driver, it will automatically find the matching device. Therefore, the automation degree of device and driver matching is improved, so as to the efficiency of driver development is improved. Because of the advantages of Linux platform, most of the drivers in Linux 2.6 kernel are rewritten according to the platform mechanism.

This paper first introduces the driving principle and architecture of Linux platform model, and describes the device, driver and device registration and unloading of the platform model in detail. Then, the driver code of watchdog platform in Linux kernel is analyzed. Finally, taking the embedded development environment tiny4412 as an example, a driver design example of Linux platform is given.

2. Principle and Architecture of Linux Platform Driver

2.1. Linux Platform

After Linux version 2.6, the platform model was introduced, which has three concepts: hardware information (device), software algorithm (driver), and platform (platform). [14]

Hardware information (device): refers to which GPIO Interface, which interrupting number (IRQ), which physical memory and other hardware resources are occupied by the driver.

Driver: it refers to the software algorithm driven, such as filtering algorithm, Fourier transform and other classical algorithms. Its main function is control, operation and other functions.

Platform bus: it is a virtual bus, and there is no real wire. Its function is to match whether the name of device and driver is the same. If they are the same, the probe function in driver will be executed.

When designing drivers based on platform model, device and driver are often written in different source files and described with different structures. After being registered into the kernel, the platform bus will match the relationship between them according to whether their name members are the same. [15]

Linux Platform is as shown in Figure 1. [16-17]

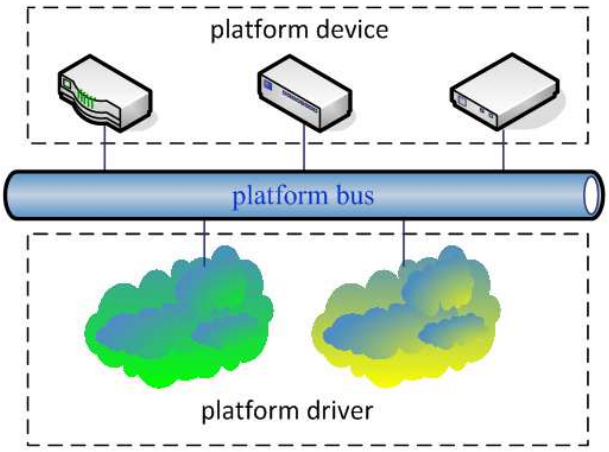

Figure 1. *Linux Platform.*

In Linux 2.6 kernel, the general process of developing underlying driver through platform mechanism is: (1) define platform_device; (2) register platform_device; (3) define platform_driver; (4) register platform_driver. The function entry point provided by the device driver is defined by the platform_driver which explains to the system. User develops his own device driver to realize the function interface in the structure of platform driver according to the function of the driver. [18-19]

2.2. Describing the Device of a Platform Device

In the Linux platform model, the following structure is commonly used to describe a device:

struct platform_device { // platform bus device
const char * name; // The name of the platform device. Use this as the basis for matching
int id; // When the device name conflicts with other devices, ID is used to distinguish them
struct device dev; // Built in device structure
u32 num_resources; // Number of resource structures
struct resource * resource; // Pointer to the resource structure array
const struct platform_device_id *id_entry; // ID used to match the device driver_Table
struct pdev_archdata archdata; //Additional parameters can be added
};

The resource structure is described as follows:

struct resource { // Resource structure
resource_size_t start; // The starting value of the resource
resource_size_t end; // End value of the resource
const char *name; // Resource name
unsigned long flags; // The identification of resources used to identify different resources
struct resource *parent, *sibling, *child; // Resource pointer, can form a linked list
};

2.3. Describing the Driver of a Platform Device

In the Linux platform model, the following structure is

commonly used to describe a driver:

struct platform_driver {

int (*probe)(struct platform_device *); // Probe function pointer. When the match is successful, the function pointed to by this pointer will be triggered;

int (*remove)(struct platform_device *); // Unload function. When either device or driver is deleted, the function pointed to by this pointer will be triggered;

void (*shutdown)(struct platform_device *); // Close the device function. When the device receives the shutdown command, the function pointed to by this pointer will be triggered;

int (*suspend)(struct platform_device *, pm_message_t state);// Sleep function: when the device receives the sleep command, the function pointed to by this pointer will be triggered;

int (*resume)(struct platform_device *); // Wake up function: when the device is awakened from sleep, the function pointed to by this pointer will be triggered;

struct device_driver driver; // Built in device_ Driver structure;

const struct platform_device_id *id_table; //List of devices supported by the device driver;

};

2.4. Register / Uninstall Device Functions

int platform_driver_register(struct platform_driver *); // Register driver with platform;

void platform_driver_unregister(struct platform_driver *); // Unload the driver from the platform;

int platform_device_register(struct platform_device *); // Register device with platform;

void platform_device_unregister(struct platform_device *); // Uninstall device from platform;

When we don't need the device driver, we can delete it from the bus to save memory space, because the memory resource of embedded system is very precious.

In short, the advantages of the platform model are as follows: when the driver needs to be transplanted to another platform, only the device part needs to be modified, and the driver part does not need to be modified. Because the register address and interrupt number of the driver part are obtained through device, the portability of the driver becomes very strong, and the product development cycle will be greatly reduced.

3. Bus Driver Code Analysis of Linux Kernel Watchdog Platform

Tiny4412's watchdog driver is designed based on the platform model. It is divided into two parts, one is device, the other is driver, of which is written in two source files.

It is worth noting that they use a large number of precompiled macro definitions here to facilitate menuconfig configuration and remove or add different functions.

3.1. The Section of Tiny4412 Watchdog Device

Source codes are displayed as following:

```
#ifdef CONFIG_S3C_DEV_WDT
static struct resource s3c_wdt_resource [] = {
[0] = DEFINE_RES_MEM(S3C_PA_WDT, SZ_1K),
[1] = DEFINE_RES_IRQ(IRQ_WDT),
};
struct platform_device s3c_device_wdt = {
.name= "s3c2410-wdt",
.id= -1,
.num_resources= ARRAY_SIZE(s3c_wdt_resource),
.resource= s3c_wdt_resource,
};
```

The codes are parsed as follows:

There are two kinds of hardware resources in the arrays 3c_wdt_resource:

Register address: S3C_PA_WDT is the base address of the register watchdog, its value is 0x10060000

Interrupt number: IRQ_WDT is Interrupt macro definition, its value is 75

Only some members of the platform_device structure are initialized in the device section:

The value of the name member is initialized to "S3C2410 WDT", and the platform bus will match based on it;

Id is set to - 1 here, which means you don't care about repetition;

Num_resources uses an array _size macro, which is used to calculate the number of array elements;

Resource points to s3c_wdt_resource, that is, the physical address of the watchdog register;

Then s3c_wdt_resource was put into the smdk4x12_devices array, and then the system sets the smdk4x12_ devices registered uniformly with device;

3.2. The Section of Tiny4412 Watchdog Driver

Source codes are displayed as following:

```
static struct platform_driver s3c2410wdt_driver = {
.probe= s3c2410wdt_probe,
.remove= __devexit_p(s3c2410wdt_remove),
.shutdown= s3c2410wdt_shutdown,
.suspend= s3c2410wdt_suspend,
.resume= s3c2410wdt_resume,
.driver= {
.owner= THIS_MODULE,
.name = "s3c2410-wdt",
.of_match_table= of_match_ptr(s3c2410_wdt_match),
},
};
```

The codes are parsed as follows:

The driver section is based on the value of driver.name as the basis for matching. After observing the previous devices, we know that their names are the same. Therefore, the matching is successful and S3C2410 WDT Probe function is executed, in which the initialization is completed to activate watchdog.

4. Design Example of Bus Driver on Linux Platform

In actual project, we can refer to the platform model for driver design, which can greatly shorten the time cycle of project migration. Taking the embedded development environment tiny4412 as an example in the project, a driver design example of Linux platform is given. There are two parts in Linux platform driver design: device section and driver section, which are shown as follows.

4.1. Device Section

Source codes are displayed as follows:

```
struct resource led_res [] = {
[0] = {
.start = 442,
.end = 442,
.flags = IORESOURCE_IRQ,
.name = "key1 irq"
},
[1] = {
.start = 0x110002e0, //starting adress of gpm4con
.end = 0x110002e7, //end adress of gpm4con,
.flags = IORESOURCE_MEM, // The type of resource is physical memory
.name = " GPM4CON ",
},
};
struct platform_device led_device = {
.name = "tiny4412_led", // matching with the name of tiny4412_ Led
.id = -1,
.resource = led_res,
.num_resources = ARRAY_SIZE(led_res),
.dev.release = leds_release, // Write an empty function to eliminate the warning when unloading the module
};
static int __init tiny4412_device_init(void)
{
int ret;
ret = platform_device_register(&led_device);
if(ret < 0){
printk("platform_device_register error\n");
return ret;
}
return 0;
}
static void __exit tiny4412_device_exit(void)
{
platform_device_unregister(&led_device);
}
module_init(tiny4412_device_init);
module_exit(tiny4412_device_exit);
MODULE_LICENSE("GPL");
```

The codes are parsed as follows:

module_init and module_exit macro determines the functions to execute when the user enters the command insmod/ rmmod, respectively.

platform_device_register/platform_device_unregister function is used to register/delete a device to/from the platform bus. You can see that when user use insmod to install the driver, it registers led_ device to the platform bus; when user use rmmod to unload the driver, it will remove the led_device from platform bus.

The matching name is "tiny4412_Led", the platform bus will match according to this name.

4.2. Driver Section

Source codes are displayed as follows:

```
struct platform_driver led_driver = {
.probe = led_probe,
.remove = led_remove,
.driver = {
.owner = THIS_MODULE,
.name = "tiny4412_led",
},
};
static int __init tiny4412_driver_init(void)
{
int ret;
ret = platform_driver_register(&led_driver);
if(ret < 0){
printk("platform_driver_register error\n");
return ret;
}
return 0;
}
static void __exit tiny4412_driver_exit(void)
{
platform_driver_unregister(&led_driver);
}
module_init(tiny4412_driver_init);
module_exit(tiny4412_driver_exit);
MODULE_LICENSE("GPL");
```

The codes are parsed as follows:

When user uses insmod/rmmod, tiny4412_driver_init/ tiny4412_ driver_ Exit function is executed respectively.

Tiny4412_driver_init/tiny4412_driver_exit register/delete the platform driver to/from the platform bus respectively.

In led_driver, the member driver.name is the basis for matching. You can see that it is the same as the name as device. Therefore, when both device and driver modules are registered into the kernel, the probe function in driver, namely led_probe function, will be executed. In the probe function, the initialization of related hardware is completed.

5. Conclusions

This paper introduces the driving principle and architecture of Linux platform model, and describes the device, driver and device registration and unloading of the platform bus model through examples. The platform driver architecture has the characteristics of reusing framework code, strong independence of device resources and drivers,

simple code, unified kernel interface, easy maintenance and expansion. In the development of specific drivers, we only need to focus on keeping the underlying device operation function set corresponding one by one with the kernel interface provided by the driver structure, and ensuring device.name and driver.name consistent, and making the platform device registered in the kernel space before the platform driver, which can make the driver run well and stably, greatly reduce the work intensity and shorten the development time of new products. Compared with the traditional device driver mechanism, the Linux platform driver mechanism registers the resources of the device into the kernel which is managed by the kernel, and driver uses these resources by applying standard interface provided by platform_device, which improves the independence of driver and resource management, and has better portability and security. The developing test shows that the driver based on this architecture has good portability, maintainability and scalability.

Acknowledgements

Fund projects: Science and technology projects of Guangdong Province (2014B040401014, 2016A040403115); Major scientific research and cultivation program of Guangdong Ocean University (Q18305).

References

[1] REN Yan-Yan, ZHAI Gao—Shou, ZHANG Jun-Hong, Automatic Updating and Auxiliary Tools of Linux Device Drivers Computer Systems & Applications, 2018, 27 (7), pp. 211-218.

[2] Marco Pagani, Alessandro Biondi, Mauro Marinoni, Lorenzo Molinari, Giuseppe Lipari, Giorgio Buttazzo, A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration, Future Generation Computer Systems 129 (2022) 125–140.

[3] Milorad Papi´c, Zlatko Bundalo, Dušanka Bundalo, Radovan Stojanovi´c, Živorad Kova˘cevi´c, Dražen Pašali´c, Branimir Cviji´c, Microcomputer based embedded SCADA and RFID systems implemented on LINUX platform, Microprocessors and Microsystems 63 (2018) 116–127.

[4] Jasleen Kaur and SRN Reddy, Implementation of Linux Optimization Technique for ARM Based System on Chip, Procedia Computer Science 171 (2020) 1780–1789.

[5] Cuong V. Nguyen, Truong Le Quang, Trung Nguyen Vu, Hoi Le Thi, Kinh Nguyen Van, Thanh Han Trong, Tuan Do Trong, Guanghao Sun, Koichiro Ishibashi, A non-contact infection screening system using medical radar and Linux-embedded FPGA: Implementation and preliminary validation, Informatics in Medicine Unlocked 16 (2019) 100225.

[6] Zhang Shilin, Development of serial port custom keyboard driver based on Embedded Linux, Information Communication, 2019, Sum. No 204, pp. 291-292.

[7] Zhao Xin, Guo Enquan, Li Xiaojie, Design and Implementation of GPIB Driver Optimal in LINUX System, Computer measurement and control, 2020, 28 (3), pp. 163-167.

[8] Chen Mengtong, Wei Feng, Yang Bingjian, Driver Design of PCI Synchronous Clock Card Under Linux Computer measurement and control, Vol. 26, 2018, 26 (1), pp. 145-148.

[9] SHI Xiao-Yan, ZHU Jian-Hong, Realization of Baud Rate Adaptive CAN Driver Under Embedded LinuxComputer Systems & Applications, 2018, 27 (1), pp. 231-234.

[10] ZHANG Tuo-zhi, KONG De-qi, ZHU En-liang, LI Xiao-dong, HI3210 Driver Software Design and Realize Based on Embeded Linux System, Aeronautical Computing Technique May 2019, Vol. 49, No. 3, pp. 99-102.

[11] Gao Jie, Research on USB Card Driver Based on Embedded Linux Microcontrollers & Embedded Systems 2018, 9, pp. 3-8.

[12] ZHAO Ming, Design and Implementation of NC System Based on Embedded Linux, Microcomputer Applications Vol. 35, No. 9, 2019, pp. 12-13, 25.

[13] Zhu Kun, Bai Pengfei, Li Hui, Wang Maochun, Zhou Guofu, Device Driver Design Based on Three-axis Acceleration Sensor in Linux Microcontrollers & Embedded Systems, 2020, 6, pp. 24-29.

[14] Zhou Derong, Xia ling, Research and application of Implementation mechanism of Linux platform driver architecture, Journal of Chifeng University (Natural Science Edition) Oct. 2010, Vol. 26, No. 10, pp. 28-30.

[15] ZHAO Jie, GONG Wei, Framebuffer Driver Based on Embedded Linux, Application of computer system, 2010, Vol. 19, 12, pp. 208-211.

[16] ZHAO Bo, GAO Zhenxiangzi, Xiang Boyang, YU Zhongde, Analysis and implementation Linux platform driver framework, Journal of Dalian Polytechnic University, Vol. 32, No. 1, Jan, 3013, pp. 71-74.

[17] Wang Xiaojun, Wang xin, Li Yuying, Design and application of platform driver based on Embedded Linux, Science and technology wind September 2018 pp. 1, 9.

[18] Huang Xiangping, Yu Shuibao, Xia Can, LCD driver module based on S3C6410 platform of embedded Linux Microcomputer and its application, 2013, Vol. 32, 12, pp. 9-12, 16.

[19] Guo Xiaomei, Embedded Linux Power Detect Driver Development in Portable Device, Microcontrollers & Embedded Systems, 2011, 5, pp. 78-81.