PCI device driver

原创 huangweiqing80 Posted at 2018-10-24 18:05:40 17842 Favorite 47

Category column: Linux driver

1. PCI device driver writing

The PCI bus is a very popular computer bus now, and it is very important to learn its driver design method. I believe that people who once wanted to learn PCI bus drivers have such an experience, that is, when they read books and materials explaining PCI bus drivers, they will be defeated by the complicated content inside, what is the configuration space and what is enumerated, I haven't really started to write the PCI driver, and I have already started to retreat here. In fact, as long as you work hard, although there are some things that you can't understand, they seem to be "not so important" for you to write PCI drivers. Because the Linux kernel already has perfect support for the PCI bus, what you need to do is a very small part.

The PCI bus under Linux will scan the devices (including devices and bridges) in the system one by one when the system is powered on. The bus number and interrupt number are all assigned to the device at this time. Not very clear, you can Skip it first, find a development board with a PCI bus, connect the PCI device, let the system restart and scan again, and then cooperate with the PCI bus driver framework given below, you will understand a lot of.

As we all know, the Linux 2.6 kernel introduced the concept of the bus driver model, so many bus-based device drivers are divided into bus drivers and device drivers. In fact, the PCI bus driver is similar to the platform bus in the 2.6 kernel, but the matching method of the platform bus is name matching, that is, the device name is consistent with the driver name. **The PCI bus matches the id_table; but there are more than one matching methods, the most common ones are the manufacturer number and the device number** . When you load the PCI driver, the driver will compare the manufacturer and device numbers of the existing devices in the system with those in the driver, and if they are consistent, it will register the PCI bus driver and proceed to the next step.

For the PCI bus power-on scanning process, it is recommended to read a blog, http://blog.csdn.net/linuxdrivers/article/details/5849698, he talked about it in detail.

The following is a PCI bus driver I wrote. Note that it is the driver for PCI device identification, and no specific function driver is implemented here. The driver of the PCI device is divided into two parts, one part is the bus, which is the part that recognizes the PCI device and calls the driver's probe function, and the other part is the specific function driver, such as the network card. There are many kinds of devices based on the PCI bus, but as far as the PCI bus driver is concerned, they are all similar. After the PCI bus driver is implemented, continue to make specific device drivers.

The program is as follows (it can run successfully in 2.6.31 to 3.1.4 kernels):

```
2
3  #include <linux/module.h>
4  #include <linux/errno.h>
5  #include <linux/sched.h>
6  #include <linux/pci.h>
7  #include <linux/init.h>
8  #include <linux/delay.h>
9  #include <asm/io.h>
10 #include <linux/ioport.h>
11 #include <linux/interrupt.h>
12 #include <linux/irq.h>
13
14 //设备相关
15 #define MY_VENDOR_ID 0x168c    //厂商号
16 #define MY_DEVICE_ID 0x002a     //设备号
17 #define MY_PCI_NAME "MYPCIE"  //自己起的设备名
18 static int debug = 1;
19 module_param(debug,int,S_IRUGO);
20 #define DBG(msg...) do{ \
twen    if(debug) \
twen       printk(msg); \
twen    }while(0)
twen
25 struct pcie_card
26 {
   //端口读写变量
   int io;
   long range,flags;
   void __iomem *ioaddr;
   int irq;
```

```
};
/* 设备中断服务*/
static irqreturn_t mypci_interrupt(int irq, void *dev_id)
{
   struct pcie_card *mypci = (struct pcie_card *)dev_id;
   printk("irq = %d,mypci_irq = %d\n",irq,mypci->irq);
   return IRQ_HANDLED;
}

/* 探测PCI设备*/
static int __init mypci_probe(struct pci_dev *dev, const struct pci_device_id *ent)
{
   int retval=0;//, intport, intmask;
   struct pcie_card *mypci;

   if ( pci_enable_device (dev) )
   {
    printk (KERN_ERR "IO Error.\n");
    return -EIO;
   }
   /*分配设备结构*/
   mypci = kmalloc(sizeof(struct pcie_card),GFP_KERNEL);
   if(!mypci)
   {
      printk("In %s,kmalloc err!",__func__);
      return -ENOMEM;
   }
   /*设定端口地址及其范围,指定中断IRQ*/
   mypci->irq = dev->irq;
   if(mypci->irq < 0)
   {
      printk("IRQ is %d, it's invalid!\n",mypci->irq);
      goto out_mypci;
   }
   mypci->io = pci_resource_start(dev, 0);
   mypci->range = pci_resource_end(dev, 0) - mypci->io;
   mypci->flags = pci_resource_flags(dev,0);
   DBG("PCI base addr 0 is io%s.\n",(mypci->flags & IORESOURCE_MEM)? "mem":"port");
   /*检查申请IO端口*/
   retval = check_region(mypci->io,mypci->range);
   if(retval)
   {
      printk(KERN_ERR "I/O %d is not free.\n",mypci->io);
      goto out_mypci;
   }
   //request_region(mypci->io,mypci->range, MY_PCI_NAME);
   retval = pci_request_regions(dev,MY_PCI_NAME);
   if(retval)
   {
      printk("PCI request regions err!\n");
      goto out_mypci;
   }
   mypci->ioaddr = ioremap(mypci->io,mypci->range);
   if(!mypci->ioaddr)
   {
      printk("ioremap err!\n");
      retval = -ENOMEM;
      goto out_regions;
   }
   //申请中断IRQ并设定中断服务子函数
   retval = request_irq(mypci->irq, mypci_interrupt, IRQF_SHARED, MY_PCI_NAME, mypci);
   if(retval)
   {
      printk (KERN_ERR "Can't get assigned IRQ %d.\n",mypci->irq);
      goto out_iounmap;
   }
   pci_set_drvdata(dev,mypci);
   DBG("Probe succeeds.PCIE ioport addr start at %X, mypci->ioaddr is 0x%p,interrupt No. %d.\n",mypci->io,mypci->ioa
   return 0;

   out_iounmap:
      iounmap(mypci->ioaddr);
```

```
98     out_regions:
99        pci_release_regions(dev);
100    out_mypci:
101       kfree(mypci);
102       return retval;
103 }
104
105 /* 移除PCI设备 */
106 static void __devexit mypci_remove(struct pci_dev *dev)
107 {
108    struct pcie_card *mypci = pci_get_drvdata(dev);
109    free_irq (mypci->irq, mypci);
110    iounmap(mypci->ioaddr);
111    //release_region(mypci->io,mypci->range);
112    pci_release_regions(dev);
113    kfree(mypci);
114    DBG("Device is removed successfully.\n");
115 }
116
117 /* 指明驱动程序适用的PCI设备ID */
118 static struct pci_device_id mypci_table[] __initdata =
119 {
120    {
121      MY_VENDOR_ID,  //厂商ID
122      MY_DEVICE_ID,  //设备ID
123      PCI_ANY_ID,   //子厂商ID
124      PCI_ANY_ID,   //子设备ID
125    },
126    {0, },
127 };
128 MODULE_DEVICE_TABLE(pci, mypci_table);
129
130 /* 设备模块信息 */
131 static struct pci_driver mypci_driver_ops =
132 {
133    name: MY_PCI_NAME,   //设备模块名称
134    id_table: mypci_table, //驱动设备表
135    probe: mypci_probe,   //查找并初始化设备
136    remove: mypci_remove  //卸载设备模块
137 };
138
139 static int __init mypci_init(void)
140 {
141    //注册硬件驱动程序
142    if ( pci_register_driver(&mypci_driver_ops) )
143    {
144       printk (KERN_ERR "Can't register driver!\n");
145       return -ENODEV;
146    }
147    return 0;
148 }
149
150 static void __exit mypci_exit(void)
151 {
152    pci_unregister_driver(&mypci_driver_ops);
153 }
154
155 module_init(mypci_init);
156 module_exit(mypci_exit);
    MODULE_LICENSE("GPL");
```

The above program is that I inserted a PCIE network card device into the development board. After the system restarts, the driver module is loaded, and a series of operations such as registering the driver will be performed.
The result after loading the module:

[root@board /]insmod ar9280.ko

Probe succeeds. PCIE ioport addr start at 98000000, mypci->ioaddr is 0xd4fa0000, interrupt No.17.

Seeing that the above Probe is successful, it means that the system has found my network card, and 98000000 is the physical starting address of the PCI bus of the system.

```
[root@board /] cat /proc/interrupts

           CPU0

 17:          0   UIC   Level     MYPCIE

 18:         24   UIC   Level     MAL TX EOB

 19:        225   UIC   Level     MAL RX EOB

 20:          0   UIC   Level     MAL SERR

 21:          0   UIC   Level     MAL TX DE

 22:          0   UIC   Level     MAL RX DE

 24:          0   UIC   Level     EMAC

 26:       1194   UIC   Level     serial

BAD:          0

[root@board /] cat /proc/iomem               //注意：查看iomem时出现了自己的设备占用的iomem，说明是IO内存

90000000-97ffffff : /plb/pciex@0a0000000

98000000-9fffffff : /plb/pciex@0c0000000

  98000000-980fffff : PCI Bus 0001:41

    98000000-9800ffff : 0001:41:00.0

      98000000-9800ffff : MYPCIE

ef600200-ef600207 : serial

ef600300-ef600307 : serial

fc000000-ffffffff : fc000000.nor_flash
```

It can be seen from the above results that the PCI bus driver has been loaded successfully. You can continue to do device driver content later.

2. Interrupts in PCI

Let's talk about PCI interrupts:

first look at the pin list of the pci device

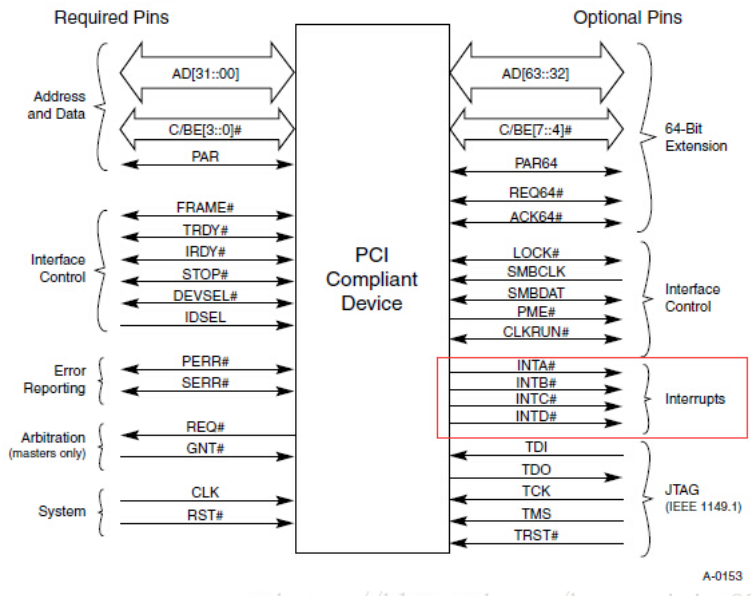

Figure 2-1: PCI Pin List

To digress, most of the signals inside are active low. It is said that it is because of low-level impedance and strong anti-interference ability.

As you can see, it has four interrupt pins, but they are placed on the right as optional.

In PCI, interrupts are level-triggered and low-level is effective. If it is not in the MSI mode, when the Device needs it, the Device driver will pull down the INTx line. Once this signal is pulled down, it will remain low. , until the Driver has no pending requests. If it is a single-function device, you only need to use INT A, and a multi-function device can use up all INT A, B, C, and D.

For multi-function devices, the logical device on the above can use any one of A, B, C, and D.

From the above, we can see that each PCI device contains four IO ports INTA# - INTD#, and the interrupt pins (INTA# - INTD#) of the device will be connected to the pins of the system interrupt controller (1RQO - IRQ15) up, so that when the INTA# - INTD# pins are pulled low, it is equivalent to pulling down the interrupt pin connected to the interrupt controller, thereby generating an interrupt.

You can also refer to PCI interrupts:
https://blog.csdn.net/shanghaiqianlun/article/details/7100825?utm_source=blogxgwz9