PCI device driver knowledge points

转载 wucongdonglai Posted at 2010-12-31 16:55:00 3785 Favorite 4

Category column: linux programming Article tags: structure module interface audio cam ide

Disclaimer: This article is not written by me, it is only modified and remarked

Please see the original post: http://hi.baidu.com/linux_kernel/blog/category/pci%C9%E8%B1%B8%C7%FD%B6%AF

That big guy is a big guy, there are a lot of good things on it!

1. Introduction to PCI

PCI is a peripheral bus specification. Let's first look at what a bus is: a bus is a path or channel for transmitting signals. Typically, a bus is an electrical connection connected to one or more conductors, and all devices connected to the bus receive all transmissions at the same time. The bus consists of an electrical interface and a programming interface. This article discusses device drivers under Linux , so focus on the programming interface.

PCI is the abbreviation of Peripheral Component Interconnect (peripheral device interconnection), which is a peripheral bus commonly used on desktops and larger computers. The PCI architecture is designed as a replacement for the ISA standard. It has three main goals: to obtain better performance when transferring data between the computer and peripherals; to be as platform-independent as possible; to simplify the work of adding and removing peripherals to the system .

2. PCI addressing

From now on, I want to illustrate the problem through some practical examples as much as possible, and reduce the description of theoretical problems, because relevant theoretical things can be found elsewhere.

Let's look at an example first. My computer is equipped with 1G of RAM, and the physical memory address space after 1G is the mapping of external device IO on the system memory address space. /proc/iomem describes the mapping of all device I/O in the system to the memory address space. Let's see how the first device whose address starts from 1G is described in /proc/iomem

: 40000000-400003ff : 0000:00:1f.1

This is a PCI device, and 40000000-400003ff is the memory address space it maps to. Occupies the 1024bytes of the memory address space, and 0000:00:1f.1 is the address of a PCI peripheral, separated by colons and commas into 4 parts:

The first 16 bits represent the domain;

The second 8 bits represent a bus number, 2^8-256, so each domain can have up to 256 buses;

The third 5 bits represent a device number, and each bus can mount up to 32 devices;

The last is 3 digits, indicating the function number. Each device can have up to 8 functions, that is, it can correspond to up to 8 logical devices, and each function uniquely corresponds to a pci_dev structure.

Note: Because the PCI specification allows a single system to have up to 256 buses, but for large systems, this is not enough, so the concept of a domain is introduced.

From this, we can conclude that the address of the above-mentioned PCI device is the No. 1 function on the No. 31 device on the No. 0 domain bus. So what exactly is the above-mentioned PCI device? Here is the output of the lspci command on my computer:

00:00.0 Host bridge: Intel Corporation 82845 845 (Brookdale) Chipset Host Bridge (rev 04)
00:01.0 PCI bridge: Intel Corporation 82845 845 (Brookdale) Chipset AGP Bridge(rev 04)
00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #1) (rev 02)
00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #2) (rev 02)
00: 1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev 42)
00:1f.0 ISA bridge: Intel Corporation 82801CAM ISA Bridge (LPC) (rev 02)

00:1f.1 IDE interface: Intel Corporation 82801CAM IDE U100 (rev 02)
00:1f.3 SMBus: Intel Corporation 82801CA/CAM SMBus Controller (rev 02)
00:1f.5 Multimedia audio controller: Intel Corporation 82801CA/CAM AC'97 Audio Controller (rev 02)
00:1f.6 Modem: Intel Corporation 82801CA/CAM AC'97 Modem Controller (rev 02)
01:00.0 VGA compatible controller : nVidia Corporation NV17 [GeForce4 420 Go](rev a3)
02:00.0 FireWire (IEEE 1394): VIA Technologies, Inc. IEEE 1394 Host Controller(rev 46)
02:01.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL- 8139/8139C/8139C+(rev 10)
02:04.0 CardBus bridge: O2 Micro, Inc. OZ6933 Cardbus Controller (rev 01)
02:04.1 CardBus bridge: O2 Micro, Inc. OZ6933 Cardbus Controller (rev 01)

lspci does not indicate the domain, but for a PC, there is generally only one domain, that is, domain 0. Through this output we can see that he is an IDE interface. As can be seen from the above output, there are 3 PCI buses (No. 0, No. 1, and No. 2) on my computer. On a single system, insertion of multiple buses is accomplished through a bridge, which is a special PCI peripheral used to connect the buses. Therefore, the overall layout of the PCI system is organized as a tree. We can use the above lspci output to draw the tree structure of the PCI system on my computer:
Bus No. 0 (main bridge)--00:01 (PCI bridge)

|--00:1d (USB number controller, which provides two logical devices, 0 and 1, that is, functions)
|--00:1e:0 (PCI bridge)

|--00:1f. Multi-function card (provides 5 logical devices, 0 ISA bridge, 1 IDE interface, 3SMBus, 5 Multimedia audio controller, 6 Modem, that is, functions)

No. 1 bus (main bridge) --01:00.0 VGA compatible controller

No. 2 bus (main bridge)--02:00.0 IEEE1394
|--02:01.0 8139 network card
|--02:04 CardBus bridge (provides two logical devices of bridge 0 and 1, that is, functions)
can be obtained from the above figure , There are 8 PCI devices on my computer, of which 4 are connected to No. 0 bus (main bridge), 1 is connected to No. 1 bus, and 3 are connected to No. 2 bus. 00:1f is a multi-function board with 5 functions. Each PCI device has its mapped memory address space and its I/O area, which is relatively easy to understand. In addition, PCI devices also have his setup registers. With the setup register, the PCI driver can access the device without probing. The layout of the setup registers is standardized, and the 4 bytes of the setup space contain a unique function ID, so a driver can identify a peripheral by querying it for its specific ID. Therefore, the main innovation of the PCI interface standard over the ISA is to set the address space.

During the system boot phase, the PCI hardware device remains inactive, but each PCI motherboard is equipped with firmware capable of handling PCI. The firmware provides access to the device setting address space by reading and writing registers in the PCI controller. The first 64 bytes of the setting address space are standardized. It provides information such as the manufacturer number, device number, version number, etc., and uniquely identifies a PCI device. At the same time, it also provides up to 6 I/O address areas, and each area can be a memory or an I/O address. These I/O address regions are the only way for a driver to find out exactly where a device is mapped into memory and I/O space. With these two points, the PCI driver has completed the function equivalent to detection. For details about the setting space of these 64 bytes, please refer to P306, Figure 12-2 of "Linux Device Drivers Third Edition", and no more details will be given.

Next, let's take a look at the details of the configuration space of the 8139too network card device.
In the 2.6 kernel system, you can see a lot of directories named after PCI devices under the directory /sys/bus/pci/devices/ , but it does not mean that all these devices exist in your system. We enter the 8139too directory, and there is a directory named after his device address 0000:02:01.0. In this directory, you can find a lot of information related to the network card device. Among them, resource records his 6 I/O address areas. 内容如下：

0x0000000000003400 0x00000000000034ff 0x0000000000000101
0x00000000e0000800 0x00000000e00008ff 0x0000000000000200
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000

由该文件能看出，8139too设备使用了两个I/O地址区域，第一个是他映射的I/O端口范围，第二个是他映射的内存地址 space. About these two values can be verified in /proc/iomem and /proc/ioport. In order to see the actual running effect,

we choose the 8139too network card as an example, and cut out the relevant code from the linux driver of the network card.

A PCI device driver must describe itself to the PCI core in the kernel. At the same time, he must also tell the PCI core which devices he can drive. Below, two related important data structures are introduced.

```
/* Used to define the list of different types of PCI devices supported by this driver*/
  struct pci_device_id {
      __u32 vendor;

      __u32 device; /* Specify the PCI vendor and device ID of the device. If the driver can handle any vendor or device ID, the value PCI_ANY_ID can be used*/
      __u32 subvendor;

      __u32 subdevice; /* Specify the PCI subsystem vendor and device ID of the device. If the driver can handle any subsystem vendor or device ID, you can use PCI_ANY_ID*/
      __u32 class;

      __u32 class_mask; /* Allows the driver to specify a PCI class (class) device, if it can handle any type, use PCI_ANY_ID */
      kernel_ulong_t driver_data; /* If necessary, it is used to save the PCI driver to distinguish different devices info */
  };

/* Used to describe the PCI driver to the PCI core*/
  struct pci_driver {
      struct list_head node;

      char *name; /* The name of all PCI drivers in the kernel must be unique, usually set to the same name as the driver module name */
      struct module *owner;
      const struct pci_device_id *id_table; //The device that the driver can manipulate list of ids.
      int (*probe)(struct pci_dev *dev, const struct pci_device_id *id); //Point to the detection function in the PCI driver for inserting new devices
      void (*remove)(struct pci_dev *dev); //remove Device
      int (*suspend)(struct pci_dev *dev, pm_message_t state); //point to the suspend function, the suspend state is passed as state, this function is optional
      int (*resume)(struct pci_dev *dev); //point to resume Function, always called after being suspended, this function can also be optional
      int (*enable_wake) (struct pci_dev *dev, pci_power_t state, int enable); //Enable wake event
      void (*shutdown) (struct pci_dev *dev );
      struct device_driver driver;
      struct pci_dynids dynids;
  };
```

pci_device_id uniquely identifies a PCI device. Several of his members represent respectively in turn: manufacturer number, device number, sub-manufacturer number, sub-device number, category, category mask (classes can be divided into base categories and subcategories), and private data. The driver of each PCI device has an array of pci_device_id, which is used to tell the PCI core which devices it can drive. The driver program of 8139too defines its pci_device_id array as follows:

```
static struct pci_device_id rtl8139_pci_tbl[];
```

This array is initialized as a group of network cards of 8139 series. When the PCI core gets this array, it will use each item in the array to follow the PCI setting space Compare the data read in to find the correct device for the driver. And pci_driver represents a pci driver. The member id_talbe is a pointer to the pci_device_id array. name is the name of the driver, and the probe completes the detection work, that is, compares the pci_device_id array with the data in the kernel. remove completes the removal of the driver. These
are .

The driver registers itself with the kernel through pci_module_init (we sometimes see the pci_register_driver function, in fact they are the same, you will see it in the kernel code, it is just a simple #define):

```
pci_module_init(&pci_driver);
```

After calling the function, if pci_device_id If the device identified in the array exists in the system, and the device does

not happen to have a driver, then the driver will be installed. Let's look at the pci device initialization code cut from the 8139too driver code:

pci_driver.h:

```
/* pci_driver.h
* helinqiang@hotmail.com
* 2006-3-5
*/
#ifndef PCI_DRIVER_H
#define PCI_DRIVER_H
#include //for struct pci_device_id
#include //for MODULE_DEVICE_TABLE
#include //for struct pci_driver
#define DRV_NAME "8139too"
#define DRV_VERSION "0.9
. RTL8139_DRIVER_NAME DRV_NAME " Fast Ethernet driver " DRV_VERSION
typedef enum{
    RTL8139 = 0,
    RTL8129,
}board_t;
static struct pci_device_id rtl8139_pci_tbl[] = {
    {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x10ec, 0x8138, PCI_ANY_ID , PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x4033, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1186, 0x1300, PCI_ANY_ID, PCI0,130x6,
    R1 , 0x1340, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x13d1, 0xab06, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1259, 0xa117, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1259, 0xa11e , PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x14ea, 0xab06, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x14ea, 0xab07, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8131db_3,
    {0x1PCI_ANY_ID, 0x1 , PCI_ANY_ID, 0, 0, RTL8139 },
    {0x1432, 0x9130, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x02ac, 0x1012, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139},
    {0x018a, 0x0106, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139},
    {0x126c, 0x1211, PCI_ANY_ID, PCI_ANY130, R71,
    R7 , 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
    {0x021b, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
#ifdef CONFIG_SH_SECUREEDGE5410
    /* Bogus 8139 silicon reports 8129 without external PROM :-( */
    {0x10ec , 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
#endif
#ifdef CONFIG_8139TOO_8129
    {0x10ec, 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8129 },
#endif
/* some invalid s0 crazy lend0 report
     * 0 . The other ids are valid and constant,
     * so we simply don't match on the main vendor id.
     */
    {PCI_ANY_ID, 0x8139, 0x10ec, 0x8139, 0, 0, RTL8139 },
    {PCI_ANY_ID, 0x8139, 0x1186, 0x1300, 0, 0, RTL8139 },
    {PCI_ANY_ID , 0x8139, 0x13d1, 0xab06, 0, 0, RTL8139 },
    {0,}
};
MODULE_DEVICE_TABLE(pci, rtl8139_pci_tbl);
static int __devinit rtl8139_init_one(struct pci_dev *pdev, const struct pci_device_id *id);
static void __devexit rtl8139_remove_one(struct pci_dev *pdev);
```

```
static struct pci_driver rtl8139_pci_driver = {
    .name = DRV_NAME,
    .id_table = rtl8139_pci_tbl,
    .probe = rtl8139_init_one,
    .remove = __devexit_p(rtl8139_remove_one),
};
#endif //PCI_DRIVER_H
```

pci_driver.c:

```
/* pci_driver.c
* helinqiang@hotmail.com
* 2006-3-5
*/
#include "pci_driver.h"
#include
MODULE_AUTHOR("Linqiang He, Hangzhou China");
MODULE_LICENSE( "Dual BSD/GPL");
static int __init rtl8139_init_module(void)
{
    /* when we're a module, we always print a version message,
     * even if no 8139 board is found.
     */
#ifdef MODULE
    printk (KERN_INFO RTL8139_DRIVER_NAME "\n");
#endif
    return pci_module_init(&rtl8139_pci_driver);
}
static void __exit rtl8139_cleanup_module (void)
{
    pci_unregister_driver(&rtl8139_pci_driver);
}
module_init(rtl8139_init_module);
module_exit(rtl8139_cleanup_module);
int __devinit rtl8139_init_one(struct pci_dev *pdev, const struct pci_device_id *id)
{
    // Various debugging codes can be inserted here, and there will be a special description below.
    return 0;
}
void __devexit rtl8139_remove_one (struct pci_dev *pdev)
{
}
```

After registering the driver successfully, rtl8139_init_one will be called. In this function, we can see the PCI setting address space and I/O by inserting some printout statements Something about the address area.

First, insert the following statement:

```
u16 vendor, device;
pci_read_config_word(pdev, 0, &vendor);
pci_read_config_word(pdev, 2, &device);
printk(KERN_INFO "%x, %x\n", vendor, device);
```

this code Read the first four digits of the setting address space of the network card device, which happens to be the manufacturer number and device number of the device. Here is the output:

```
10ec, 8139
```

10ec and 8139 are the manufacturer number and device number of my network card.

Then insert the following code:

```
u32 addr1,addr2,addr3, addr4,addr5,addr6;
pci_read_config_dword(pdev, 16, &addr1);
pci_read_config_dword(pdev, 20, &addr2);
pci_read_config_dword(pdev, 24, &addr3);
pci_read_config_dword(pdev, 28 , &addr4);
pci_read_config_dword(pdev, 32, &addr5);
```

```
        pci_read_config_dword(pdev, 36, &addr6);

printk(KERN_INFO "%x,%x,%x,%x,%x,%x\n",addr1, addr2 , addr3, addr4, addr5, addr6);
```

This code reads the initial address of the 6 I/O address areas of the network card device. The following is the output: 3401,e0000800,0,0,0,0 It can be seen that the device only uses the first two I/O address areas, which respectively identify its I/O port area and memory address space.

In addition, here, you can also directly print out the MAC address of the network card. No more details.

We can insert some different debugging codes in rtl8139_init_one to observe some actions of the device driver module in the kernel. 8139too The first 6 bytes of the device memory of the network card device store the 48-bit MAC address of the network card. We can get this MAC address by accessing the device memory. The following accesses device memory in four different ways by inserting code in rtl8139_init_one. The first is realized by accessing I/O memory, and the latter three are realized by accessing I/O ports.

The first type:

```
unsigned long mmio_start, addr1, addr2;
void __iomem *ioaddr;
mmio_start = pci_resource_start( pdev, 1);
ioaddr = pci_iomap(pdev, 1, 0);
addr1 = ioread32( ioaddr );
addr2 = ioread32( ioaddr + 4 );
printk(KERN_INFO "mmio start: %lX\n", mmio_start);
printk(KERN_INFO "ioaddr: %p\n", ioaddr);
printk(KERN_INFO "%02lX.%02lX.%02lX.%02lX. %02lX.%02lX\n",
(addr1) & 0xFF,
(addr1 >> 8) & 0xFF,
(addr1 >> 16 ) & 0xFF,
(addr1 >> 24 ) & 0xFF,

(addr2 >> 8) & 0xFF );
```

Running result:

```
  mmio start: E0000800
  ioaddr: f8aa6800 00.02.3F.AC.41.9D
  Type 2
:
unsigned long pio_start, pio_len, addr1, addr2;
void __iomem *ioaddr;
pio_start = pci_resource_start ( pdev, 0);
pio_len = pci_resource_len (pdev, 0);
ioaddr = ioport_map(pio_start, pio_len);
addr1 = ioread32( ioaddr );
addr2 = ioread32( ioaddr + 4 );
printk(KERN_INFO "pio start: %lX\ n", pio_start);
printk(KERN_INFO "ioaddr: %p\n", ioaddr);
printk(KERN_INFO "%02lX.%02lX.%02lX.%02lX.%02lX.%02lX\n",
(addr1) & 0xFF,
(addr1 >> 8) & 0xFF,
(addr1 >> 16 ) & 0xFF,
(addr1 >> 24 ) & 0xFF,
(addr2) & 0xFF,
(addr2 >> 8) & 0xFF );
```

Operation result:

```
   pio start: 3400
   ioaddr: 00013400
   00.02.3F.AC.41.9D The
third type:
unsigned long pio_start, addr1, addr2;
pio_start = pci_resource_start( pdev, 0 );
```

```
addr1 = inl( pio_start );
addr2 = inl( pio_start + 4 );
printk(KERN_INFO "port io start: %lX\n", pio_start);
printk(KERN_INFO "%02lX.%02lX.%02lX.%02lX.%02lX .%02lX\n",
(addr1) & 0xFF,
(addr1 >> 8) & 0xFF,
(addr1 >> 16) & 0xFF,
(addr1 >> 24) & 0xFF,
(addr2) & 0xFF,
(addr2 >> 8) & 0xFF );
Operation result:
  port io start: 3400
  00.02.3F.AC.41.9D The
fourth type:
unsigned long pio_start;
u8 addr1, addr2, addr3, addr4, addr5, addr6;
pio_start = pci_resource_start( pdev, 0 );
addr1 = inb( pio_start );
addr2 = inb( pio_start + 1 );
addr3 = inb( pio_start + 2 )
; ( pio_start + 3 );
addr5 = inb( pio_start + 4 );
addr6 = inb( pio_start + 5 );
printk(KERN_INFO "port io start: %lX\n", pio_start);
printk(KERN_INFO "%02X.%02X .%02X.%02X.%02X.%02X\n",
addr1, addr2, addr3, addr4, addr5, addr6 );
Running result:
  port io start: 3400
  00.02.3F.AC.41.9D
```