Linux driver learning--first understanding of PCI driver (1)

What is PCI?

PCI—Peripheral Component Interconnect, peripheral device interconnection bus. It is a widely used bus standard, which provides many new features superior to other bus standards (such as EISA), and has become the most widely used and most general bus standard in computer systems.
For some specific introductions to PCI, please refer to: PCI Specific Introduction

1. Several data structures related to PCI drivers

Drivers are always inseparable from data structures. In Linux, data structures are used to represent various devices or other things. Therefore, one of the keys for us to master the device driver is the understanding and application of various data structures.

1. pci_device_id:
Before introducing the structure, let us take a look at the address space of PCI: I/O space, storage space, configuration space.

The CPU can access all address spaces on PCI devices, among which I/O space and storage space are provided for device drivers, while the configuration space is used by the PCI initialization code in the Linux kernel. The kernel is responsible for all PCI devices at startup. Initialize and configure all PCI devices, including interrupt numbers and I/O base addresses, and list all found PCI devices in the file /proc/pci, as well as the parameters and attributes of these devices.

We don't need to understand what all the bits of the configuration register represent and what they mean. We only need to use three or five PCI registers to identify a device. Usually, we will choose the following three registers:
vendorID : identifies the hardware manufacturer and is a 16-bit register;

deviceID : Device ID, selected by the manufacturer, is also a 16-bit register. Generally paired with the manufacturer ID to generate a unique 32-bit hardware device identifier;

class : Each external device belongs to a certain class (class), which is also a 16-bit register. When a driver can support multiple similar devices, each with a different signature, but they all belong to the same class, then the class class can be used to identify their peripherals.

The following data structure is –pci_device_id.

```
struct pci_device_id {
   __u32 vendor, device;/* Vendor and device ID or PCI_ANY_ID*/
   __u32 subvendor, subdevice;/* Subsystem ID's or PCI_ANY_ID */
   __u32 class, class_mask;/* (class,subclass,prog-if) triplet */
   kernel_ulong_t driver_data;/* Data private to the driver */
   };
```

Now the problem comes again, as we said before, a driver can match one or more devices. So, what should we do at this time? You can think of arrays, right. Yes, but there is something to be aware of here

```
staticstruct pci_device_id example_pci_tbl [] __initdata ={
   {PCI_VENDOR_ID_EXAMPLE, PCI_DEVICE_ID_EXAMPLE, PCI_ANY_ID, PCI_ANY_ID,0,0, EXAMPLE},
   {0,}
   };
```

No matter how many devices you match here, remember that the last one is {0,}.

There are also two macros about initializing the structure, which can be used to simplify related operations.

PCI_DEVICE(vendor, device)

creates a struct pci_device_id that only matches a specific vendor and device ID. It sets the subvendor and subdevice of the structure to PCI_ANY_ID. PCI_ANY_ID is defined as follows:

```
#define PCI_ANY_ID (~0)
```

PCI_DEVICE_CLASS(device_class, device_class_mask)

creates a struct pci_device_id that matches a specific PCI class.

2.pci_driver:

According to the above, you have described the devices you want to match, but this is just a description, how does the kernel recognize them? Then use the following data structure – pci_driver.

```
struct pci_driver {
    struct list_head node;
    char*name;
    conststruct pci_device_id *id_table;/* must be non-NULL for probe to be called */
    int(*probe)(struct pci_dev *dev,conststruct pci_device_id *id);/* New device inserted */
    void(*remove)(struct pci_dev *dev);/* Device removed (NULL if not a hot-plug capable driver) */
    int(*suspend)(struct pci_dev *dev,pm_message_t state);/* Device suspended */
    int(*suspend_late)(struct pci_dev *dev,pm_message_t state);
    int(*resume_early)(struct pci_dev *dev);
    int(*resume)(struct pci_dev *dev);/* Device woken up */
    void(*shutdown)(struct pci_dev *dev);
    struct pci_error_handlers *err_handler;
    struct device_driver driver;
    struct pci_dynids dynids;
};
```

As can be seen from the above structure definition, its role is not only to identify the id_table structure of the device, but also includes the function probe( ) for detecting the device and the function remove( ) for uninstalling the device: this structure.

3. pci_dev:

Let us come to the last relevant data structure – pci_dev.

```
/*
* The pci_dev structure is used to describe PCI devices.
*/
struct pci_dev {
    struct list_head bus_list;/* node in per-bus list */
    struct pci_bus *bus;/* bus this device is on */
    struct pci_bus *subordinate;/* bus this device bridges to */

    void*sysdata;/* hook for sys-specific extension */
    struct proc_dir_entry *procent;/* device entry in /proc/bus/pci */
    struct pci_slot *slot;/* Physical slot this device is in */

    unsignedint devfn;/* encoded device & function index */
    unsignedshort vendor;
    unsignedshort device;
    unsignedshort subsystem_vendor;
    unsignedshort subsystem_device;
    unsignedintclass;/* 3 bytes: (base,sub,prog-if) */
    u8 revision;/* PCI revision, low byte of class word */
    u8 hdr_type;/* PCI header type (`multi' flag masked out) */
    u8 pcie_cap;/* PCI-E capability offset */
    u8 pcie_type;/* PCI-E device/port type */
    u8 rom_base_reg;/* which config register controls the ROM */
    u8 pin;/* which interrupt pin this device uses */
```

```
	struct pci_driver *driver;/* which driver has allocated this device */
	u64 dma_mask;/* Mask of the bits of bus address this
	device implements. Normally this is
	0xffffffff. You only need to change
	this if your device has broken DMA
	or supports 64-bit transfers. */

	struct device_dma_parameters dma_parms;

	pci_power_t current_state;/* Current operating state. In ACPI-speak,
	this is D0-D3, D0 being fully functional,
	and D3 being off. */
	int pm_cap;/* PM capability offset in the
	configuration space */
	unsignedint pme_support:5;/* Bitmask of states from which PME#
	can be generated */
	unsignedint pme_interrupt:1;
	unsignedint d1_support:1;/* Low power state D1 is supported */
	unsignedint d2_support:1;/* Low power state D2 is supported */
	unsignedint no_d1d2:1;/* Only allow D0 and D3 */
	unsignedint mmio_always_on:1;/* disallow turning off io/mem
	decoding during bar sizing */
	unsignedint wakeup_prepared:1;
	unsignedint d3_delay;/* D3->D0 transition time in ms */

	#ifdef CONFIG_PCIEASPM
	struct pcie_link_state *link_state;/* ASPM link state. */
	#endif

	pci_channel_state_t error_state;/* current connectivity state */
	struct device dev;/* Generic device interface */

	int cfg_size;/* Size of configuration space */

	/*
	* Instead of touching interrupt line and base address registers
	* directly, use the values stored here. They might be different!
	*/
	unsignedint irq;
	struct resource resource[DEVICE_COUNT_RESOURCE];/* I/O and memory regions + expansion ROMs */
	resource_size_t fw_addr[DEVICE_COUNT_RESOURCE];/* FW-assigned addr */

	/* These fields are used by common fixups */
	unsignedint transparent:1;/* Transparent PCI bridge */
	unsignedint multifunction:1;/* Part of multi-function device */
	/* keep track of device state */
	unsignedint is_added:1;
	unsignedint is_busmaster:1;/* device is busmaster */
	unsignedint no_msi:1;/* device may not use msi */
	unsignedint block_ucfg_access:1;/* userspace config space access is blocked */
	unsignedint broken_parity_status:1;/* Device generates false positive parity */
	unsignedint irq_reroute_variant:2;/* device needs IRQ rerouting variant */
	unsignedint msi_enabled:1;
	unsignedint msix_enabled:1;
	unsignedint ari_enabled:1;/* ARI forwarding */
	unsignedint is_managed:1;
	unsignedint is_pcie:1;/* Obsolete. Will be removed.
	Use pci_is_pcie() instead */
	unsignedint needs_freset:1;/* Dev requires fundamental reset */
	unsignedint state_saved:1;
	unsignedint is_physfn:1;
	unsignedint is_virtfn:1;
	unsignedint reset_fn:1;
	unsignedint is_hotplug_bridge:1;
	unsignedint __aer_firmware_first_valid:1;
	unsignedint __aer_firmware_first:1;
	pci_dev_flags_t dev_flags;
```

```
    atomic_t enable_cnt;/* pci_enable_device has been called */

    u32 saved_config_space[16];/* config space saved at suspend time */
    struct hlist_head saved_cap_space;
    struct bin_attribute *rom_attr;/* attribute descriptor for sysfs ROM entry */
    int rom_attr_enabled;/* has display of the rom attribute been enabled? */
    struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE];/* sysfs file for resources */
    struct bin_attribute *res_attr_wc[DEVICE_COUNT_RESOURCE];/* sysfs file for WC mapping of resources *
    #ifdef CONFIG_PCI_MSI
    struct list_head msi_list;
    #endif
    struct pci_vpd *vpd;
    #ifdef CONFIG_PCI_IOV
    union{
    struct pci_sriov *sriov;/* SR-IOV capability related */
    struct pci_dev *physfn;/* the PF this VF is associated with */
    };
    struct pci_ats *ats;/* Address Translation Service */
    #endif
};

```

It can be seen from the above definition that it describes in detail almost all hardware information of a PCI device, including vendor ID, device ID, various resources, etc.

2. Basic framework

Above, some basic information we will use are briefly introduced. Next, let's take a look at a basic framework of the PCI driver, how to organize these things into a program.

```

staticstruct pci_device_id example_pci_tbl [] __initdata ={
         {PCI_VENDOR_ID_EXAMPLE, PCI_DEVICE_ID_EXAMPLE, PCI_ANY_ID, PCI_ANY_ID,0,0, EXAMPLE},
         {0,}
        };
        /* 对特定PCI设备进行描述的数据结构 */
        struct example_pci {
         unsignedint magic;
         /* 使用链表保存所有同类的PCI设备 */
         struct example_pci *next;

         /* ... */
        }
        /* 中断处理模块 */
        staticvoid example_interrupt(int irq,void*dev_id,struct pt_regs *regs)
        {
         /* ... */
        }
        /* 设备文件操作接口 */
        staticstruct file_operations example_fops ={
         owner: THIS_MODULE,/* demo_fops所属的设备模块 */
         read: example_read,/* 读设备操作*/
         write: example_write,/* 写设备操作*/
         ioctl: example_ioctl,/* 控制设备操作*/
         open: example_open,/* 打开设备操作*/
         release: example_release /* 释放设备操作*/
         /* ... */
        };
        /* 设备模块信息 */
        staticstruct pci_driver example_pci_driver ={
         name: example_MODULE_NAME,/* 设备模块名称 */
         id_table: example_pci_tbl,/* 能够驱动的设备列表 */
         probe: example_probe,/* 查找并初始化设备 */
         remove: example_remove /* 卸载设备模块 */
         /* ... */
```

```
};
staticint __init example_init_module (void)
{
 /* ... */
}
staticvoid __exit example_cleanup_module (void)
{
 pci_unregister_driver(&demo_pci_driver);
}
/* 加载驱动程序模块入口 */
module_init( example_init_module);
/* 卸载驱动程序模块入口 */
module_exit( example_cleanup_module);
```

The above code gives the framework of a typical PCI device driver, which is a relatively fixed mode.