Linux *Kernel Hacking* Free Course 2003

UNIVERSITA' DEGLI STUDI DI ROMA TorVergata Università

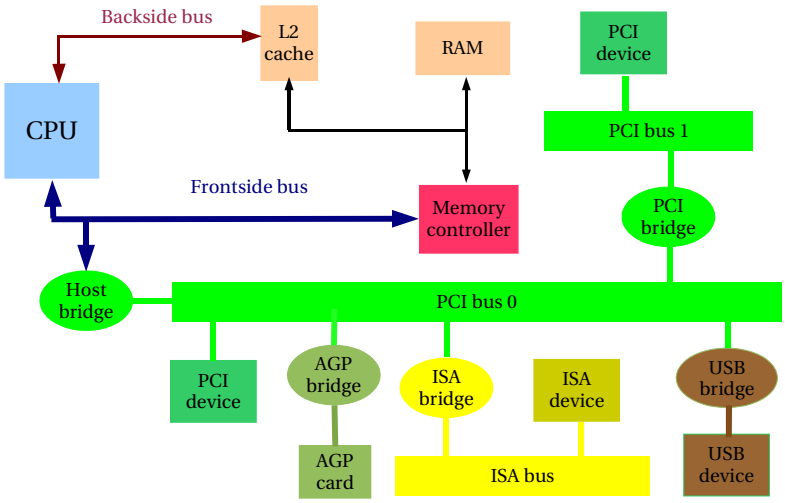
D. P. Bovet, M. Cesati

Writing a driver for a PCI device

Peripheral Component Interconnect

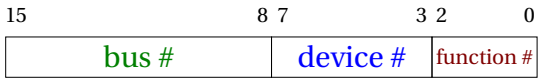
- PCI is a standard for I/O buses and device interfacing
- Introduced in 1991 by Intel and many others
- Used on architectures IA-32, Alpha, PowerPC, SPARC64, ...
- 32-bit internal bus (64 bit in PCI 2.1)
- Clock speed: 25/33MHz
  (66MHz in PCI 2.1, 133MHz in PCI-X)
- It allows transfers in *burst mode*
- It allows *bus mastering*
- PCI devices are *plug and play*
- PCI devices may be *hot-pluggable (i.e.: CardBus)*

PCI-based architecture



Buses, devices and functions (1)

Each PCI peripheral is identified by a 16-bit number:



- *bus number*: a PCI bus in the system (0-255)
- *device number*: a device (slot) on the bus (0-31)
- *function number*: a function of the device (0-7)

Buses, devices, and functions (2)

```
# /sbin/lspci
00:00.0 Host bridge: Intel Corp. 82845 845 (Brookdale) Chipset Host
Bridge (rev 04)
00:01.0 PCI bridge: Intel Corp. 82845 845 (Brookdale) Chipset AGP
Bridge (rev 04)
00:1e.0 PCI bridge: Intel Corp. 82801BA/CA PCI Bridge (rev 05)
00:1f.0 ISA bridge: Intel Corp. 82801BA ISA Bridge (LPC) (rev 05)
00:1f.1 IDE interface: Intel Corp. 82801BA IDE U100 (rev 05)
00:1f.2 USB Controller: Intel Corp. 82801BA/BAM USB (Hub #1) (rev 05)
00:1f.4 USB Controller: Intel Corp. 82801BA/BAM USB (Hub #2) (rev 05)
01:00.0 VGA compatible controller: ATI Technologies Inc Radeon QL
02:03.0 Multimedia audio controller: C-Media Electronics Inc CM8738
(rev 10)
02:04.0 USB Controller: NEC Corporation USB (rev 41)
02:04.1 USB Controller: NEC Corporation USB (rev 41)
02:04.2 USB Controller: NEC Corporation USB 2.0 (rev 02)
02:0a.0 Multimedia audio controller: Avance Logic Inc. ALS4000 Audio
Chipset
02:0b.0 Multimedia audio controller: Creative Labs SB Live! EMU10k1
(rev 07)
02:0b.1 Input device controller: Creative Labs SB Live! MIDI/Game Port
(rev 07)
02:0c.0 Ethernet controller: 3Com Corporation 3c905C-TX/TX-M [Tornado]
(rev 78)
```

Resources of a PCI device

- Local memory on the device (not system RAM!)
- Hardware registers (control, status, data registers) mapped onto either *I/O ports* or *local memory* of the device
- IRQ lines (4 lines per device)
- Configuration registers

Each PCI function may use up to 6 *regions*, each of which defines an interval of either the I/O address space or the local memory address space

I/O ports

A hardware device can be programmed by reading and writing values in specific I/O ports by means of the **`in`** and **`out`** CPU instructions

The address space of the I/O ports is global (for instance, if the addresses **`0x60`**-**`0x6f`** are assigned to the I/O ports of the keyboard, no other hardware device can use these I/O addresses)

In order to solve conflicts among peripherals, the BIOS or the kernel can assign any I/O address to each I/O port of a PCI device

Local memory

Some types of peripherals have their own local memory on the device

The address space of the local memory is global (for instance, the address **`0xe0000000`** identifies exactly one local memory location)

Generally speaking, the address space of the local memory is different from the address space of the system RAM

In order to solve conflicts among peripherals, the BIOS or the kernel can assign any range of addresses to the local memory of a PCI device

IRQ lines

Each PCI device has 4 different pins (lines) for interrupt requests (IRQs): **`#A`**, **`#B`**, **`#C`** and **`#D`**

Why 4 pins? A single PCI device may sport several peripherals (functions), and each peripheral may need its own IRQ line

All interrupt signals are routed indipendently to the Interrupt Controller, and then converted into interrupt signals sent to the CPU

The PCI devices may share the same IRQ vector (for instance, IRQ 9 can be used by a USB controller and a sound card at the same time)

Configuration registers

Each PCI peripheral (function) owns several configuration registers (256 bytes)

While bootstrapping, the firmware (BIOS) queries each PCI device and assigns to it some subranges of the global address spaces

When initializing, any PCI device driver reads the configuration registers in order to know what subranges of the global address spaces have been assigned to the peripheral

Plug and Play support (1)

Thanks to the configuration registers:

- no jumpers on the board of a PCI device
- no need to "probe" the peripheral in order to discover its resources

When initializing, however, a device driver must check whether the peripheral is actually installed in the system

How to do this safely?

Plug and Play support (2)

Every PCI peripheral has a unique PCI identifier stored in one of its configuration registers

When initializing, a device driver asks the identifiers of all PCI devices in the system and looks for the identifiers of the peripherals that it is able to drive

0200 : **10b7** : **9200**

class: Ethernet controller

vendor ID: 3Com Corporation

device ID: 3c905C-TX/TX-M [Tornado]

Plug and Play support (3)

In some cases, class, vendor ID, and device ID are not enough

Therefore, the PCI standard allows to specify also:

- A subsystem vendor ID identifier
- A subsystem device ID identifier

Supporto PCI in Linux 2.4 e 2.5

The Linux kernel allows to easily write drivers that are:

- modular
- with *Plug and Play* support
- with *hotplug* support

The kernel API for PCI device drivers is essentially the same in Linux 2.4 e Linux 2.5

Writing a PCI driver in 5+3 steps

#1: initialize a table of PCI identifiers

#2: initialize a PCI device driver descriptor

#3: initialize the PCI peripheral and the device driver

#4: release the device driver

#5: load and unload the module

#6: create an interface for the User Mode programs

#7: handle the interrupts

#8: handle the DMA

Step #1: the `pci_device_id` table (1)

- Allocate a `pci_device_id` table
- Initialize the `pci_device_id` table with the PCI identifiers associated with the supported peripheral(s)

It is possible to search for a specific peripheral, or a peripheral of a given vendor, or a peripheral in a given class, and so on

Step #1: the `pci_device_id` table (2)

```
#include <linux/init.h>
#include <linux/pci.h>
...
static struct pci_device_id
               foo_idtable[] __devinitdata = {
    { vendor: FOO_VENDOR_ID,
      device: FOO_DEVICE_ID,
      subvendor: PCI_ANY_ID,
      subdevice: PCI_ANY_ID,
      class: 0,
      class_mask: 0,
      driver_data: 0 },
    { 0, }
};
```

Step #1: the `pci_device_id` table (3)

The `class` and `class_mask` fields allow to search for a given class of hardware devices

The `PCI_ANY_ID` macro acts as a "wildcard": it is matched by any identifier

The `driver_data` field is left for use by the device driver

`__devinitdata` forces the kernel to release the data structure after device driver initialization, unless *hotplugging* is supported

Step #2: the `pci_driver` descriptor (1)

The `pci_driver` data structure is the descriptor of our device driver in the kernel's PCI layer, including:

- the name of the driver
- the address of the `pci_device_id` table
- the address of a function that should check the hardware device and initialize the driver
- the address of a function that should shut down the hardware device and de-initialize the driver
- the addresses of four functions that should implement the power-management features of the hardware device

Step #2: the `pci_driver` descriptor (2)

```
struct pci_driver foo_driver = {
    name: "foo",
    id_table: foo_idtable,
    probe: foo_probe,
    remove: __devexit_p(foo_remove),
    suspend: NULL, resume: NULL,
    save_state: NULL, enable_wake: NULL,
};
```

`__devexit_p(x)` expands to `x` if either the driver is included in a module or the kernel supports *hotplugging*, to `NULL` otherwise

Step #3: initialize the peripheral (1)

The `foo_probe()` function is invoked by the kernel's PCI layer once for every PCI peripheral whose identifiers match those specified in the `pci_device_id` table

Typically, this happens:

- while bootstrapping, if the driver is compiled statically
- when the module containing the device driver is loaded
- when the hardware device is hot-plugged to the system

The function must return 0 if the device driver is willing to drive the PCI peripheral, an error code otherwise

Step #3: initialize the peripheral (2)

```
int __devinit foo_probe(struct pci_dev *pcidev,
    const struct pci_device_id *id)
{
    ...
}
```

The `pci_dev` data structure is the descriptor of a PCI peripheral in the kernel's PCI layer

The `id` pointer refers to the entry of the `pci_device_id` data structure matching the PCI identifiers of the peripheral

`__devinit` forces the kernel to release the function code after the initialization, unless the kernel supports *hotplugging*

Step #3: initialize the peripheral (4)

The `foo_probe()` function should perform several operations:

a) initialize the driver's peripheral descriptor

b) allocate and initialize the device driver's data structures

c) store the driver's peripheral descriptor address into the `pci_dev` hardware device descriptor

d) enable the PCI peripheral

e) get the I/O address space

f) get the local memory address space

g) register the interrupt handler

Step #3a: the driver's peripheral descriptor

The *driver's peripheral descriptor* is a data structure defined by you that should include the information required to drive the peripheral

Typically it includes fields storing the assigned resources, the address of memory buffers, usage counters, and so on

```
struct foodescr_t {
  unsigned long ioadd;
  unsigned long iolen;
   unsigned long memadd;
  unsigned long memlen;
  void * memvadd;
  int irq;
  ...
} foo;
```

Step #3b: allocate the data structures

Many data structures required by the device driver are allocated (often inside the peripheral descriptor) and initialized now

Typical examples are:

- semaphores (to block waiting for a resource)
- spin locks (to protect against executions in other CPUs)
- lists (to collect objects and descriptors)
- wait queues (to collect processes waiting for an event)

It is also possible to allocate memory, although it would be better to defer this task until the PCI peripheral is effectively used

Step #3c: link the peripheral descriptor

In general, a device driver can handle several PCI peripheral of the same type, if present

In such cases, we have to allocate several peripheral descriptors

When the kernel's PCI layer invokes a method of our driver (for instance, remove), it provides a pointer to the `pci_dev` descriptor of the PCI peripheral

In order to get the address of the corresponding peripheral descriptor, `foo_probe()` stores it into the `pci_dev` data structure:

```
pci_set_drvdata(pcidev, &foo);
```

Step #3d: enable the PCI peripheral

Right after having been powered, any PCI peripheral is *disabled* : it reacts only to I/O operations on its configuration registers

The `foo_probe()` function must therefore *enable* the peripheral:

```
pci_enable_device(pcidev);
```

`pci_enable_device()` may also assign the resources (IRQ vector, I/O and local memory address spaces) to the device, so the peripheral must be enabled before any other operation

Step #3e: get the I/O addresses

The `foo_probe()` function asks the kernel's PCI layer about the I/O addresses to be used for accessing the I/O ports:

```
foo.ioadd=pci_resource_start(pcidev,FOO_IOREG);
foo.iolen=pci_resource_len(pcidev,FOO_IOREG);
```

Once obtained, the driver should reserve the subrange of I/O addresses so that it will be unavailable to the other device drivers:

```
request_region(foo.ioadd,foo.iolen,"foo");
```

Step #3f: get the memory addresses (1)

The `foo_probe()` function asks the kernel's PCI layer about the local memory addresses to be used for accessing the I/O ports or the local memory:

```
foo.memadd=pci_resource_start(pcidev,FOO_MEMREG);
foo.memlen=pci_resource_len(pcidev,FOO_MEMREG);
```

Once obtained, the driver should reserve the subrange of local memory addresses so that it will be unavailable to the other device drivers:

```
request_mem_region(foo.memadd,foo.memlen,"foo");
```

Step #3f: get the memory addresses (2)

In the PC architecture, PCI's local memory addresses are physical addresses, usually well above the address of the last cell of RAM (check with `/sbin/lspci -v`)

Before using such addresses, the `foo_probe()` function must remap them into the kernel virtual address space by using `ioremap()` or `ioremap_nochache()`:

```
foo.memvadd = ioremap(foo.memadd, foo.memlen);
```

Moreover, the device driver should access the local memory by using a few helper functions: `readb()`, `readw()`, `readl()`, `writeb()`, `writew()`, `writel()`, `memcpy_fromio()`, `memcpy_toio()`, ...

Step #3g: register the IRQ handler

If the PCI peripheral makes use of interrupts, the `foo_probe()` function must register the IRQ handler:

```
request_irq(pcidev->irq, foo_inthandl,
            SA_SHIRQ, "foo", &foo);
```

- The `irq` field of the `pci_dev` descriptor stores the IRQ vector
- `SA_SHIRQ` means that the interrupt line can be shared with other devices
- The address of the peripheral descriptor `&foo` will be passed to the interrupt handler whenever it is invoked

Step #4: release the device driver

The `foo_remove()` function is invoked by the kernel's PCI layer when the PCI device is unplugged or the module containing the device driver is unloaded

```
void __devexit foo_remove(struct pci_dev *dev)
{
 struct foodescr_t *foo_p=pci_get_drvdata(dev);
 free_irq(dev->irq, foo_p);
 iounmap(foo_p->memvadd);
 release_mem_region(foo_p->memadd,foo_p->memlen);
 release_region(foo_p->ioadd, foo_p->iolen);
}
```

`__devexit` means that the function will be compiled only if the kernel supports *hotplugging* or the driver is compiled as a module

Step #5: module loading and unloading

```
int __init foo_init(void)
{
  int rc = pci_module_init(&foo_driver);
  if (rc) return rc;
  ...
};

void __exit foo_cleanup(void)
{
  ...
  pci_unregister_driver(&foo_driver);
};

module_init(foo_init);
module_exit(foo_cleanup);
```