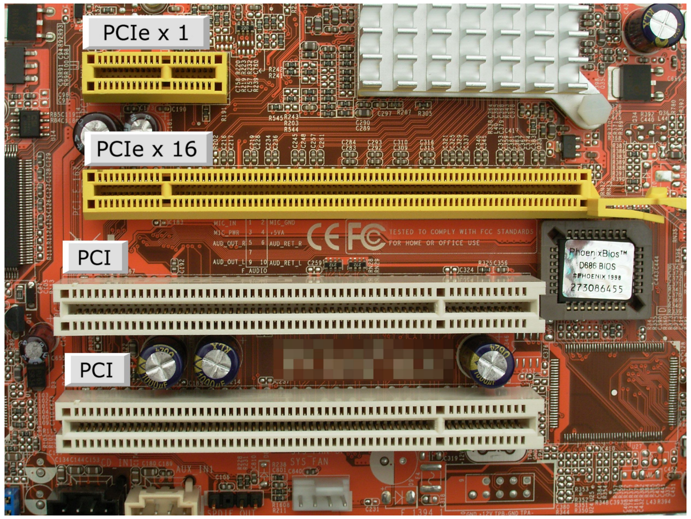
Module 5

PCI



Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI bus Family

- PCI (Peripheral Component Interconnect) was originally 32-bit parallel bus clocked at 33/66MHz and capable of 266 MB/second peak throughput.
- It was available in several form factors: cardbus, Mini PCI etc..
- PCI Extended (PCI-X) expands the bus width to 64-bit running at 133MHz and can achieve throughput of 1GB/sec
- PCIe (Express) is the 3rd generation of high performance IO buses.
    First generation buses include the ISA, EISA, VESA, Mirco Channel buses.
    Second generation were PCI, AGP PCI-X.
- PCIe has become IO Interconnect of choice for mobile, desktop, workstation and server. PCIe is:
  - Serial PCI bus and uses serial protocol for data transfer.
  - Supports 32 serial links, with ach link is capable of 250MB/sec tput in each transfer direction
  - Aggregate data rate of 8GB/sec and higher are possible.

- PCIe interface is backward compatible and employs the same load/store and familiar transactions such as memory IO and configuration address read/write as PCI.
- Existing driver software runs on PCIe system without any modification (backward compatible). Driver requiring using enhanced PCIe will required changes.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Express (PCI-e)Features

- PCIe is not a bus, it is a packet network of 32 bits elements.
- Hot pluggable, energy saving features.
- Each device has its own dedicated wires to the PCI controller.
- PCIe uses Transaction Layer Packets (TLPs) instead of bus transactions
- Support for Posted and non-posted transactions
- Data flow control, error detection and retransmit
- Design to look and feel like PCI
- Full backward compatibility and interoperability with legacy PCI (including INTx, emulate interrupt pins).
- A PCIe interconnect that connects two devices together is referred to as link.
  - A link consists of signal pairs: x1,x2,x4,x8,x16,x32. These signal pairs are called **Lanes**. Lanes work in sync to transmit 32-bit words faster.
  - More Lanes implemented in device means better tput

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Express (PCI-e)Features

- PCI and PCI/X buses are multi-drop parallel buses with devices share one bus. PCI express, on the other hand, is implemented as a serial, point-to-point interconnection for communication between two devices. One physical device per PCI bus.
- Multiple PCI-e devices can be connected switch which means one can practically connect a large number of devices together in a system.
- A point-to-point interconnect implies limited electrical load on the link allowing transmit/receive frequencies to scale to much higher numbers.
- Board cost is low because of few pins per device and less complexity.
- PCI-e is highly scalable and that is achieved by implementing scalable numbers of pins and signal lanes per interconnect based on performance requirements for that interconnect.
- Other useful features are: QoS, Hot Plug/Hot swap, Advanced error handling (RAS) and Advanced power management features that allows PCI-e to use for mobile applications,
- Configuration space available per function is extended to 4KB, allowing designers to define additional registers.

  Driver needs to use memory mapped (*MMCNFIG*) method. Also, kernel should be built with option: CONFIG_PCI_MMCONFIG=y to support extended configuration space.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

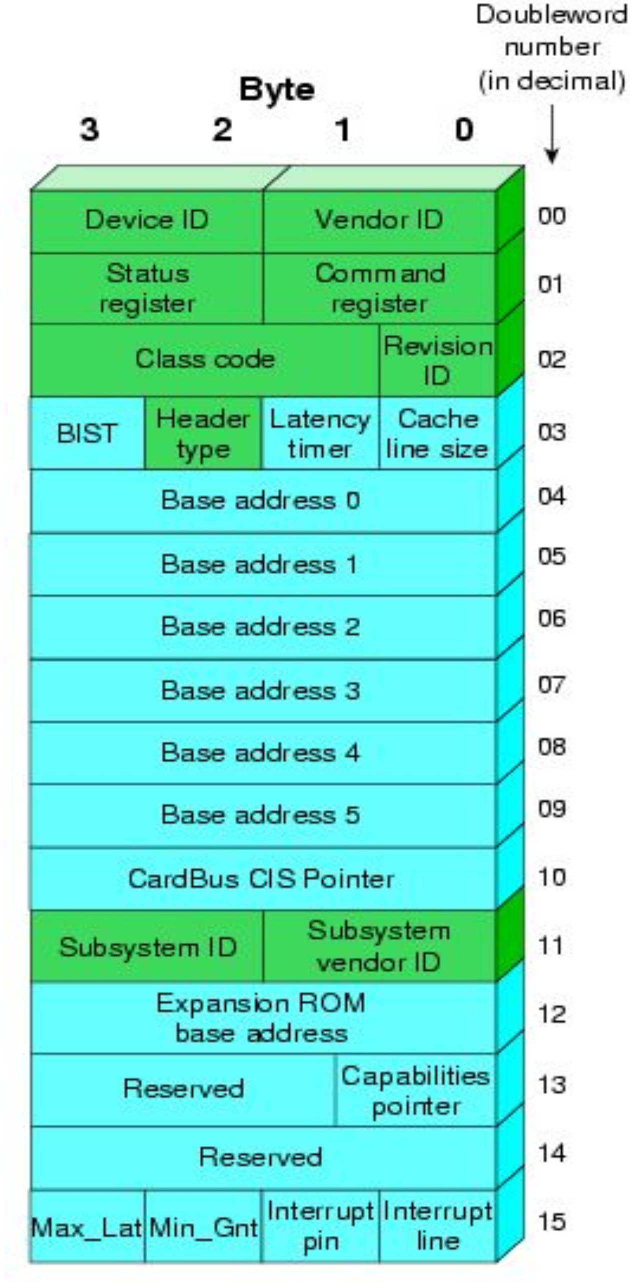
PCI Standards

- There are 4 components to the PCI subsystem: **Bus Number, Device Number, Function Number and Register Number**
- A Device is a physical thing on the PCI bus. It could be a video card, an Ethernet card, a Northbridge, or anything.
- A device resides on a bus and contains one or more functions. A device containing multiple functions is referred to as a multifunction device. Each function is stand alone, for example, one function could be a graphic controller while another might be a network interface.
- There are up to 256 available Buses on a PCI system, most commonly all the cards and chips will be located on Bus 0 and Bus 1.
- There is a software maximum of 32 devices that can exist on each bus. The physical hardware limit (10-12) is much lower than this due to electrical loading issues

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Standards

- All devices have at least 1 function (function #0). There are 8 possible functions per device, numbered 0-7. Any device that has more than 1 function is called a multi-function device. Functions start numbering at 0 and work up to 7.
- Every function of a device has 256 eight-bit registers, called **configuration registers (configuration space).** Registers 0-3F are defined by the PCI specification and provide a wealth of information about the particular function.
- Registers 40-FF are vendor defined and control the properties of the function itself. Without vendor specific documentation, these registers should probably be left untouched.



Byte 3	Byte 2	Byte 1	Byte 0	Doubleword number (in decimal)
Device ID		Vendor ID		00
Status register		Command register		01
Class code			Revision ID	02
BIST	Header type	Latency timer	Cache line size	03
Base address 0				04
Base address 1				05
Base address 2				06
Base address 3				07
Base address 4				08
Base address 5				09
CardBus CIS Pointer				10
Subsystem ID		Subsystem vendor ID		11
Expansion ROM base address				12
Reserved			Capabilities pointer	13
Reserved				14
Max_Lat	Min_Gnt	Interrupt pin	Interrupt line	15

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

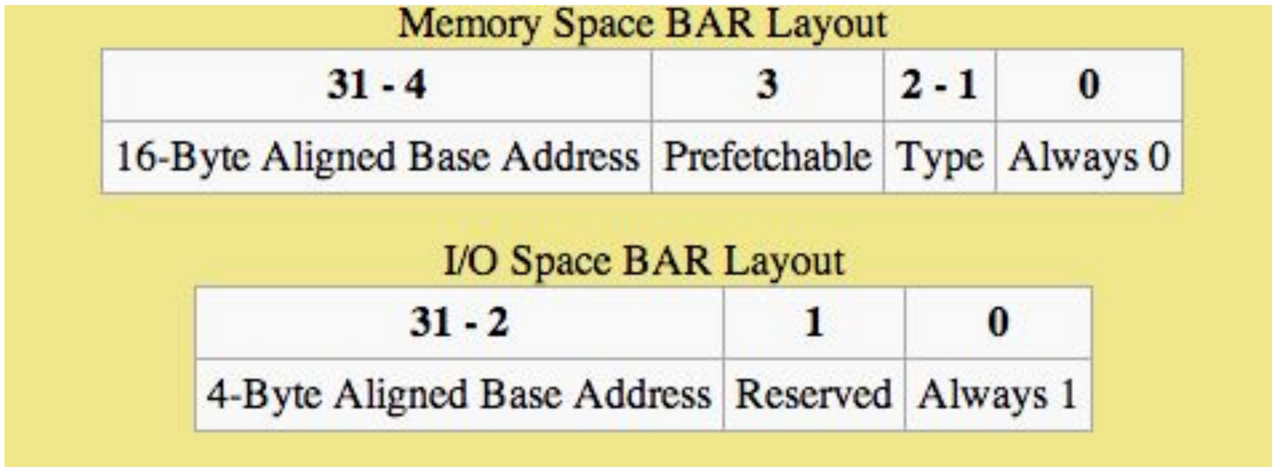
PCI Enumeration and Addressing

- Every PCI device is equipped with PCI-aware firmware. At system boot, the firmware and Linux Kernel performs configuration transactions with every PCI device in order to reserve an address space region for device in the computer physical memory
- In order to address a PCI device its registers or memory must be mapped into the I/O port address space or the memory-mapped address space of the system.
- During enumeration, BIOS or kernel programs the device *Base Address Registers* (commonly called BARs) to inform device of its address mapping by performing configuration transactions to the PCI controller.
- Device resources such as IO addresses, IRQ lines, base address are automatically assigned at boot by BIOS. Driver only needs to read these settings populated in the PCI configuration space
- PCI device configuration space is Little Endian, Make sure use kernel functions for proper conversion.
- PCI devices are addressed using bus, device, and function numbers and are identified via: venderIDs, deviceIDs and class codes.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Base Address Registers (BAR)

- Base Address Registers (BARs) on PCI device are used to hold memory addresses used by the device, or offsets for port addresses.
- BARs pointing to MMIO addresses are mapped in physical memory while PMIO addresses can reside at any memory address (even beyond physical memory)
- BAR can help find size of device memory and starting location



Memory Space BAR Layout

31 - 4	3	2 - 1	0
16-Byte Aligned Base Address	Prefetchable	Type	Always 0

I/O Space BAR Layout

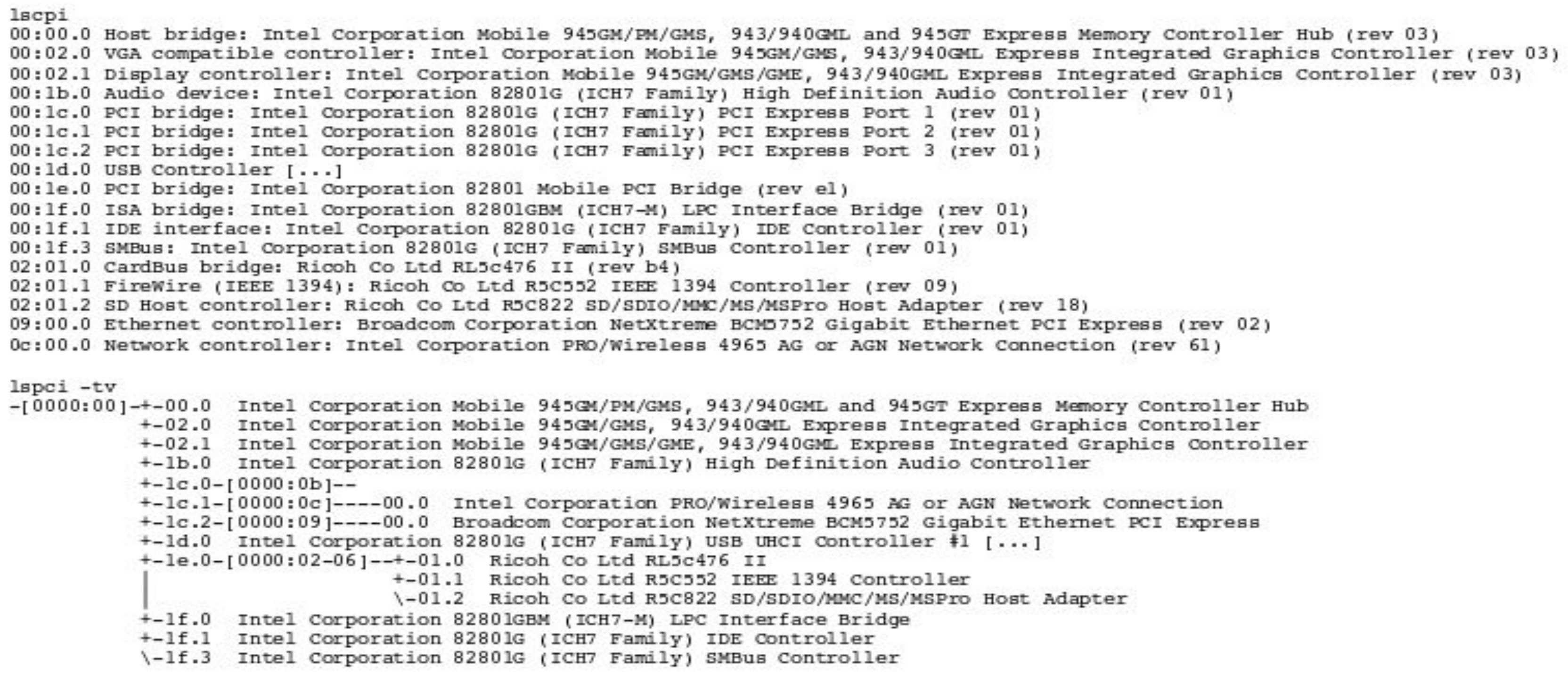
31 - 2	1	0
4-Byte Aligned Base Address	Reserved	Always 1

PCI Prefetchable Memory

- MMIO BAR indicates if device memory is Prefetchable or non-Prefetchable.
  - Prefetchable means a system can prefetch chunk of memory without having any negative impact on device's operations.
- PCI devices that map their control registers to a memory address range declare that range as non-prefetchable.
- If the MMIO region of PCI device is marked as prefetchable, the CPU can cache its contents and do all sorts of optimization with it. Video Memory may be the only one on PCI boards flagged as prefetchable.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

lspci can dump the PCI device list

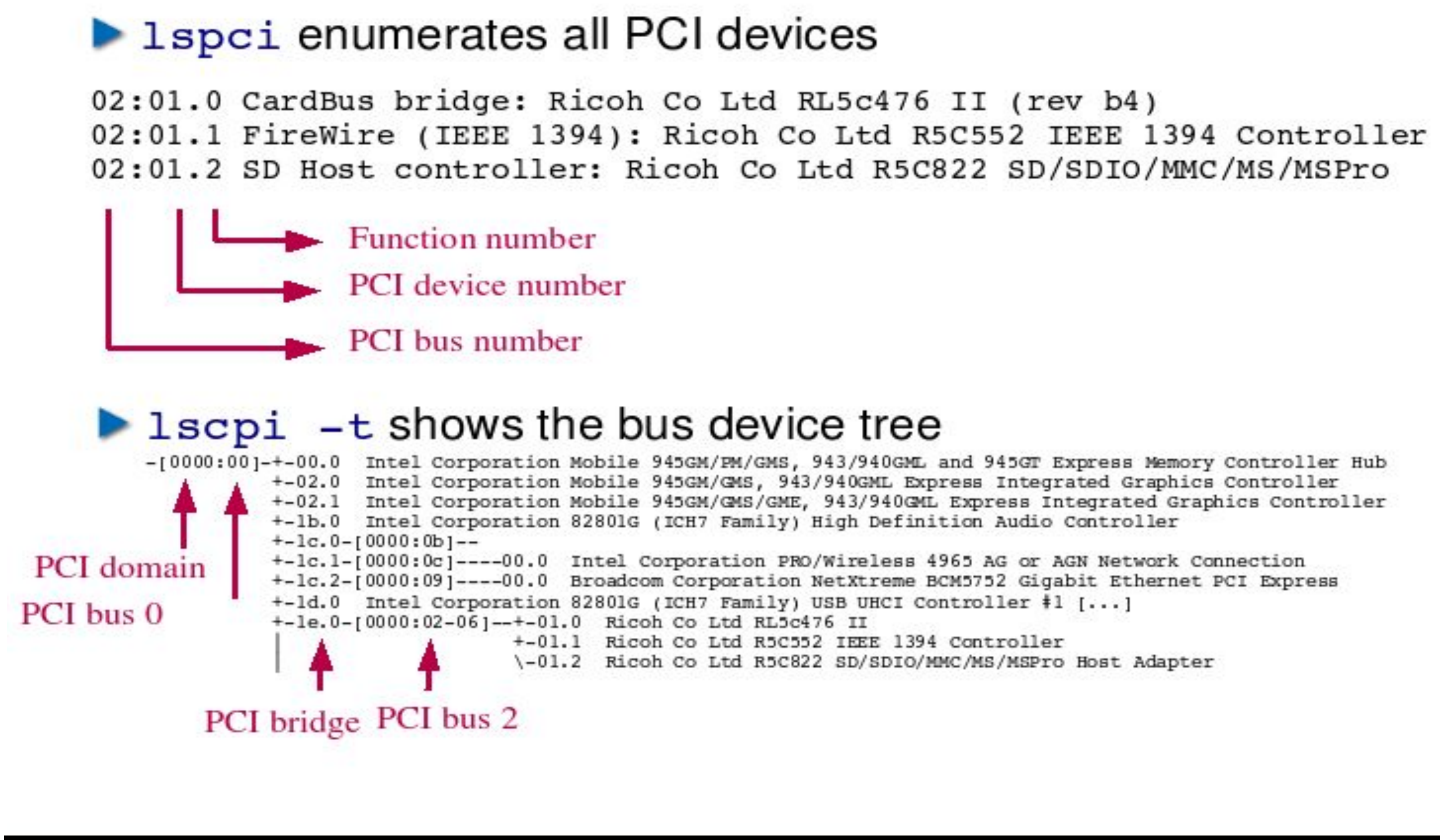


```
lspci
00:00.0 Host bridge: Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express Memory Controller Hub (rev 03)
00:02.0 VGA compatible controller: Intel Corporation Mobile 945GM/GMS, 943/940GML Express Integrated Graphics Controller (rev 03)
00:02.1 Display controller: Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express Integrated Graphics Controller (rev 03)
00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller (rev 01)
00:1c.0 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 1 (rev 01)
00:1c.1 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 2 (rev 01)
00:1c.2 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 3 (rev 01)
00:1d.0 USB Controller [...]
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev e1)
00:1f.0 ISA bridge: Intel Corporation 82801GBM (ICH7-M) LPC Interface Bridge (rev 01)
00:1f.1 IDE interface: Intel Corporation 82801G (ICH7 Family) IDE Controller (rev 01)
00:1f.3 SMBus: Intel Corporation 82801G (ICH7 Family) SMBus Controller (rev 01)
02:01.0 CardBus bridge: Ricoh Co Ltd RL5c476 II (rev b4)
02:01.1 FireWire (IEEE 1394): Ricoh Co Ltd R5C552 IEEE 1394 Controller (rev 09)
02:01.2 SD Host controller: Ricoh Co Ltd R5C822 SD/SDIO/MMC/MS/MSPro Host Adapter (rev 18)
09:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5752 Gigabit Ethernet PCI Express (rev 02)
0c:00.0 Network controller: Intel Corporation PRO/Wireless 4965 AG or AGN Network Connection (rev 61)

lspci -tv
-[0000:00]-+-00.0  Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express Memory Controller Hub
           +-02.0  Intel Corporation Mobile 945GM/GMS, 943/940GML Express Integrated Graphics Controller
           +-02.1  Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express Integrated Graphics Controller
           +-1b.0  Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller
           +-1c.0-[0000:0b]--
           +-1c.1-[0000:0c]----00.0  Intel Corporation PRO/Wireless 4965 AG or AGN Network Connection
           +-1c.2-[0000:09]----00.0  Broadcom Corporation NetXtreme BCM5752 Gigabit Ethernet PCI Express
           +-1d.0  Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #1 [...]
           +-1e.0-[0000:02-06]--+-01.0  Ricoh Co Ltd RL5c476 II
           |                    +-01.1  Ricoh Co Ltd R5C552 IEEE 1394 Controller
           |                    \-01.2  Ricoh Co Ltd R5C822 SD/SDIO/MMC/MS/MSPro Host Adapter
           +-1f.0  Intel Corporation 82801GBM (ICH7-M) LPC Interface Bridge
           +-1f.1  Intel Corporation 82801G (ICH7 Family) IDE Controller
           \-1f.3  Intel Corporation 82801G (ICH7 Family) SMBus Controller
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

lspci



- `lspci` enumerates all PCI devices

```
02:01.0 CardBus bridge: Ricoh Co Ltd RL5c476 II (rev b4)
02:01.1 FireWire (IEEE 1394): Ricoh Co Ltd R5C552 IEEE 1394 Controller
02:01.2 SD Host controller: Ricoh Co Ltd R5C822 SD/SDIO/MMC/MS/MSPro
```

Function number
PCI device number
PCI bus number

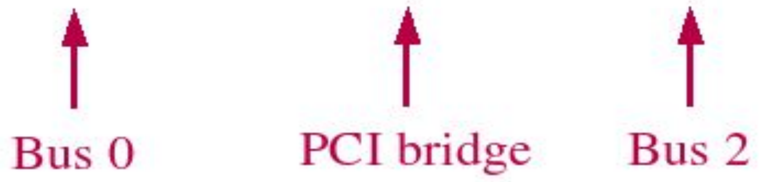
- `lscpi -t` shows the bus device tree

```
-[0000:00]-+-00.0  Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express Memory Controller Hub
           +-02.0  Intel Corporation Mobile 945GM/GMS, 943/940GML Express Integrated Graphics Controller
           +-02.1  Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express Integrated Graphics Controller
           +-1b.0  Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller
           +-1c.0-[0000:0b]--
           +-1c.1-[0000:0c]----00.0  Intel Corporation PRO/Wireless 4965 AG or AGN Network Connection
           +-1c.2-[0000:09]----00.0  Broadcom Corporation NetXtreme BCM5752 Gigabit Ethernet PCI Express
           +-1d.0  Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #1 [...]
           +-1e.0-[0000:02-06]--+-01.0  Ricoh Co Ltd RL5c476 II
           |                    +-01.1  Ricoh Co Ltd R5C552 IEEE 1394 Controller
                                \-01.2  Ricoh Co Ltd R5C822 SD/SDIO/MMC/MS/MSPro Host Adapter
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

lspci can dump the PCI device list

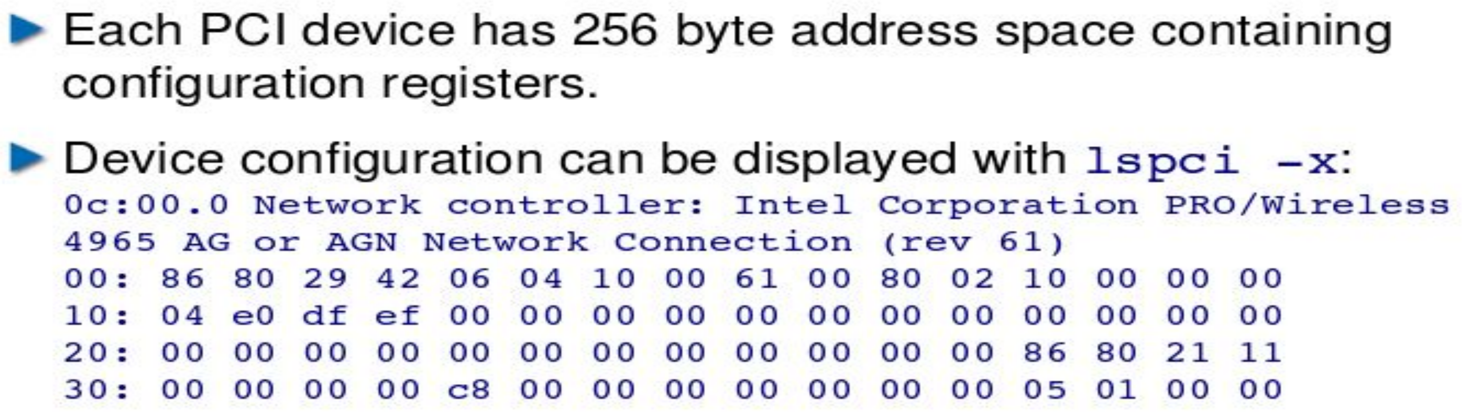
- This tree structure reflects the structure in `/sys`:
  `/sys/devices/pci0000:00/0000:00:1e.0/0000:02:01.2`



Copyright © 2010 The Regents of the
University of California. All Rights Reserved.

Display PCI configuration space

PCI devices contains three addressable regions: Configuration space, IO ports and memory regions. Access to the configuration space is vital for the driver because that is where IO memory addresses, assigned IRQ are found. PCI devices implements up to six IO address region, where each region consists of either memory or IO locations.



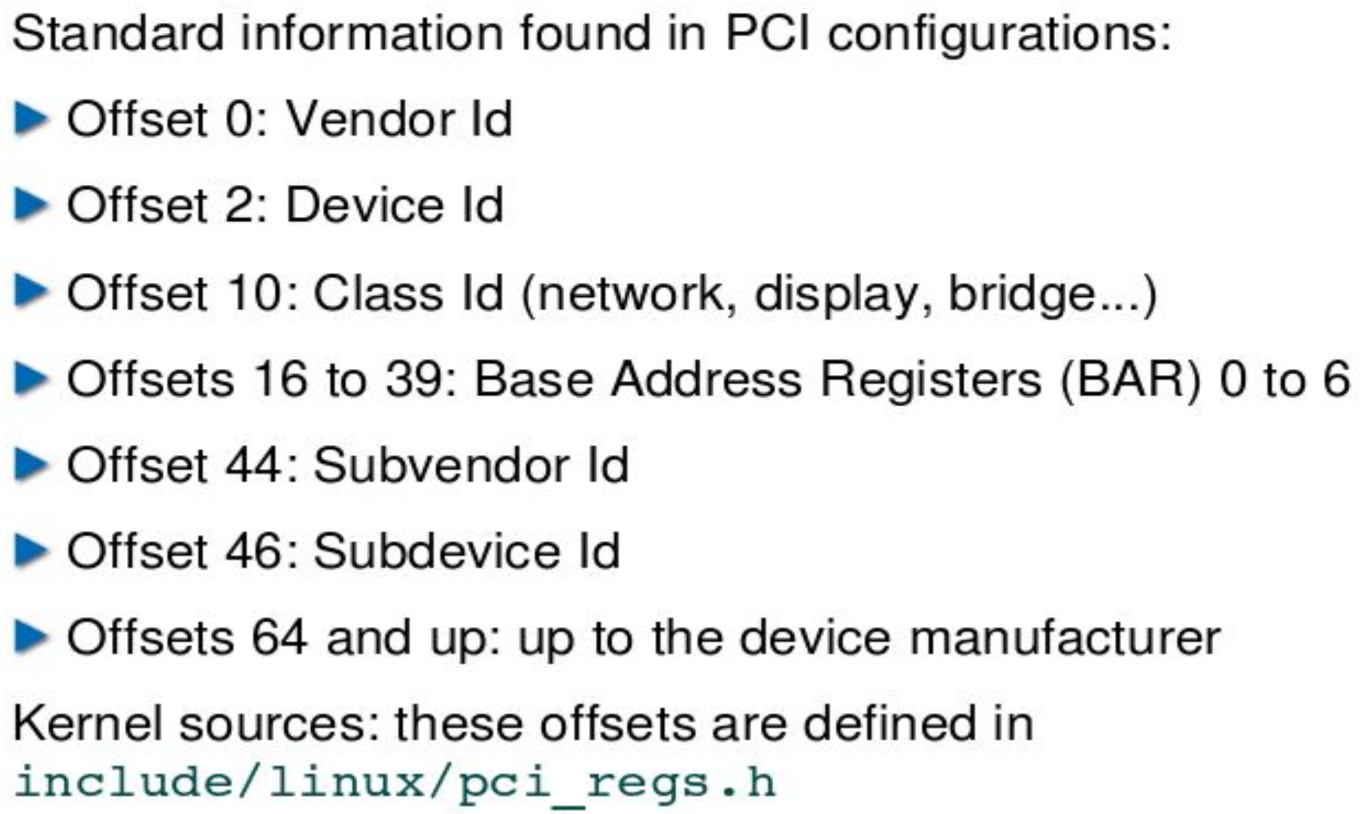
- Each PCI device has 256 byte address space containing configuration registers.
- Device configuration can be displayed with `lspci -x`:

```
0c:00.0 Network controller: Intel Corporation PRO/Wireless
4965 AG or AGN Network Connection (rev 61)
00: 86 80 29 42 06 04 10 00 61 00 80 02 10 00 00 00
10: 04 e0 df ef 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 86 80 21 11
30: 00 00 00 00 c8 00 00 00 00 00 00 00 05 01 00 00
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI configuration space

PCI configuration space is a non-volatile parameter storage area for PCI device function.



Standard information found in PCI configurations:

- Offset 0: Vendor Id
- Offset 2: Device Id
- Offset 10: Class Id (network, display, bridge...)
- Offsets 16 to 39: Base Address Registers (BAR) 0 to 6
- Offset 44: Subvendor Id
- Offset 46: Subdevice Id
- Offsets 64 and up: up to the device manufacturer

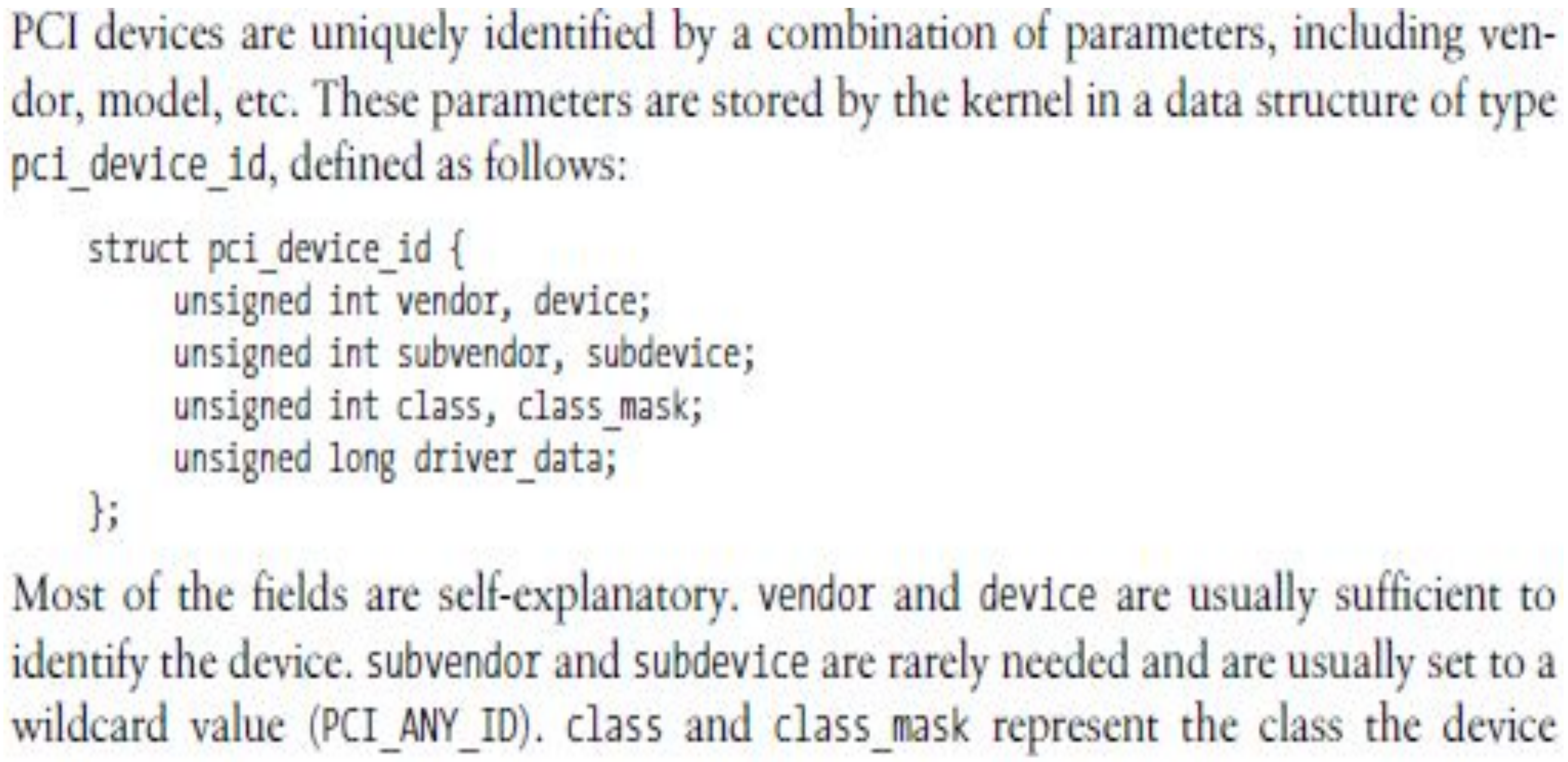
Kernel sources: these offsets are defined in
`include/linux/pci_regs.h`

First 16 bit register at offset 0 identifies the hardware manufacturer. For instance every Intel device is marked with the same vendor number, 0x8086. Device ID paired with Vendor ID is used to make a unique 32-bit identifier for a hardware device. Every PCI device belongs to a class. The class register is a 16-bit value whose top 8 bits identify the class or group and the last 8 bits specifies the device type. For example, ethernet belongs to network group

University of California. All Rights Reserved.

PCI DEVICE REGISTRATION

Each device driver registers with the kernel a vector **pci_device_id** instance that lists the IDs of the devices it can handle.



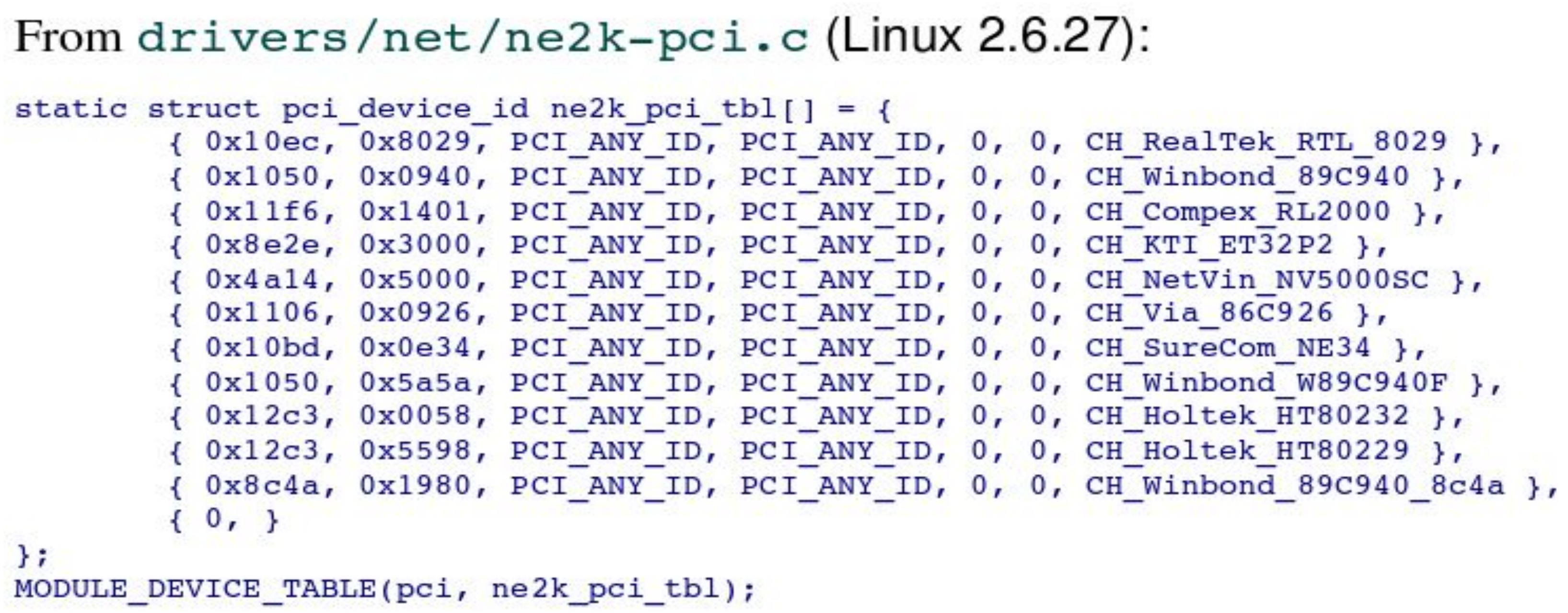
PCI devices are uniquely identified by a combination of parameters, including vendor, model, etc. These parameters are stored by the kernel in a data structure of type pci_device_id, defined as follows:

```
struct pci_device_id {
        unsigned int vendor, device;
        unsigned int subvendor, subdevice;
        unsigned int class, class_mask;
        unsigned long driver_data;
};
```

Most of the fields are self-explanatory. vendor and device are usually sufficient to identify the device. subvendor and subdevice are rarely needed and are usually set to a wildcard value (PCI_ANY_ID). class and class_mask represent the class the device

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI DEVICE REGISTRATION - example



From drivers/net/ne2k-pci.c (Linux 2.6.27):

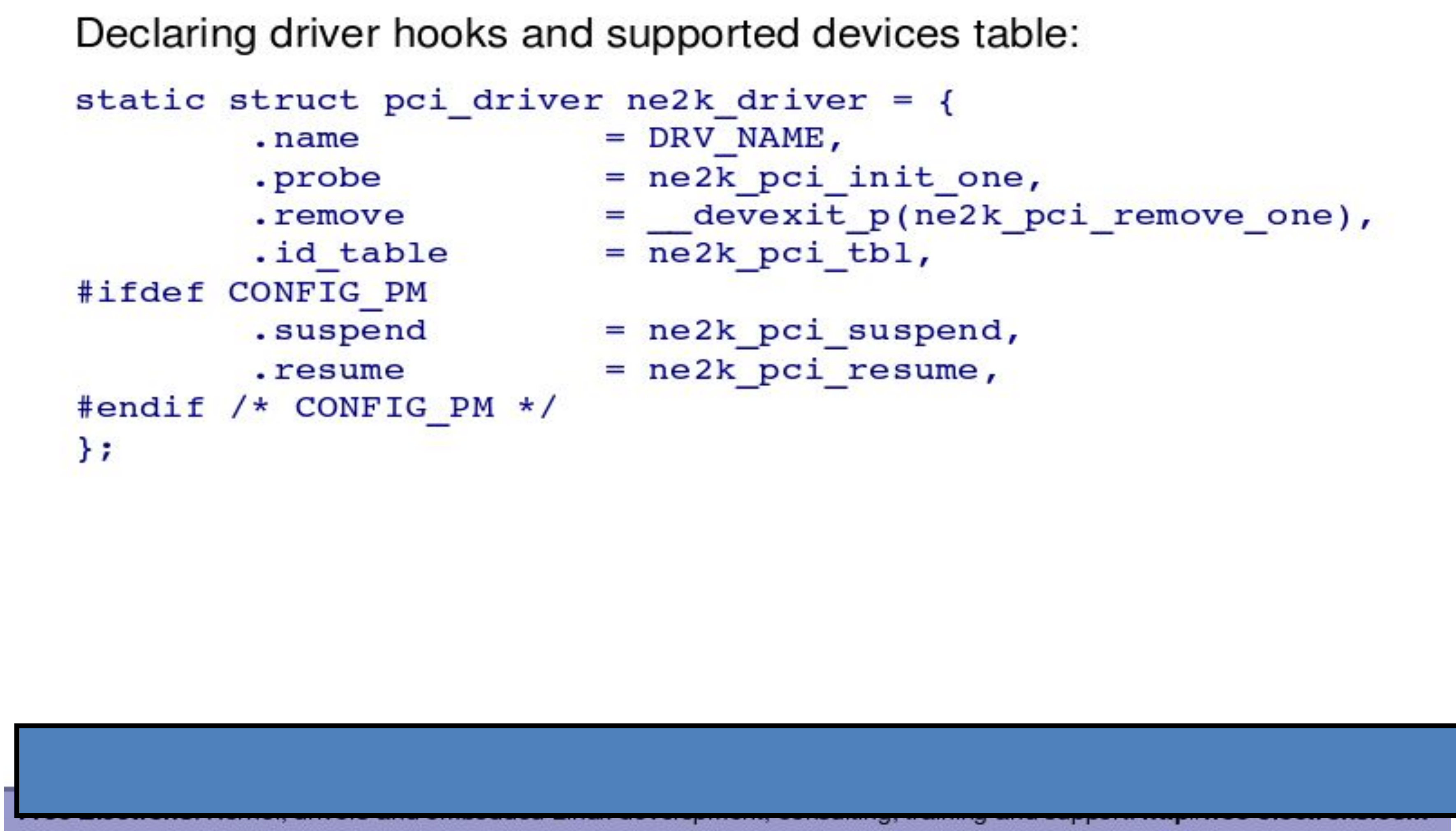
```
static struct pci_device_id ne2k_pci_tbl[] = {
        { 0x10ec, 0x8029, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_RealTek_RTL_8029 },
        { 0x1050, 0x0940, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_89C940 },
        { 0x11f6, 0x1401, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Compex_RL2000 },
        { 0x8e2e, 0x3000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_KTI_ET32P2 },
        { 0x4a14, 0x5000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_NetVin_NV5000SC },
        { 0x1106, 0x0926, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Via_86C926 },
        { 0x10bd, 0x0e34, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_SureCom_NE34 },
        { 0x1050, 0x5a5a, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_W89C940F },
        { 0x12c3, 0x0058, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80232 },
        { 0x12c3, 0x5598, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80229 },
        { 0x8c4a, 0x1980, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_89C940_8c4a },
        { 0, }
};
MODULE_DEVICE_TABLE(pci, ne2k_pci_tbl);
```

During compile time, driver specify the supported devices via id table. This information is captured from the module object. When the driver module is loaded, depmod utility deciphers the IDs present in the device table and add the entry in /lib/modules/<kernel>/module.alais.

Where v stands for Vendor ID, d for deviceID, sv subvendorID and * for wild card match. When the device is hotplug the kernel generates a uevent and sets the MODALIAS that announces the identity of the newly inserted device.

Udevd receives the uevent via netlink socket and invokes modprobe with the above MODALIAS that the kernel passed up to it. Modprobe finds the matching entry in /lib/module<kernel>module.alias and proceeds to insert the device. Device is now ready!!

PCI Driver Entry points or call back methods



Declaring driver hooks and supported devices table:

```
static struct pci_driver ne2k_driver = {
        .name                = DRV_NAME,
        .probe               = ne2k_pci_init_one,
        .remove              = __devexit_p(ne2k_pci_remove_one),
        .id_table            = ne2k_pci_tbl,
#ifdef CONFIG_PM
        .suspend             = ne2k_pci_suspend,
        .resume              = ne2k_pci_resume,
#endif /* CONFIG_PM */
};
```

Copyright © 2010 The Regents of the
University of California. All Rights Reserved.

PCI Device Registration

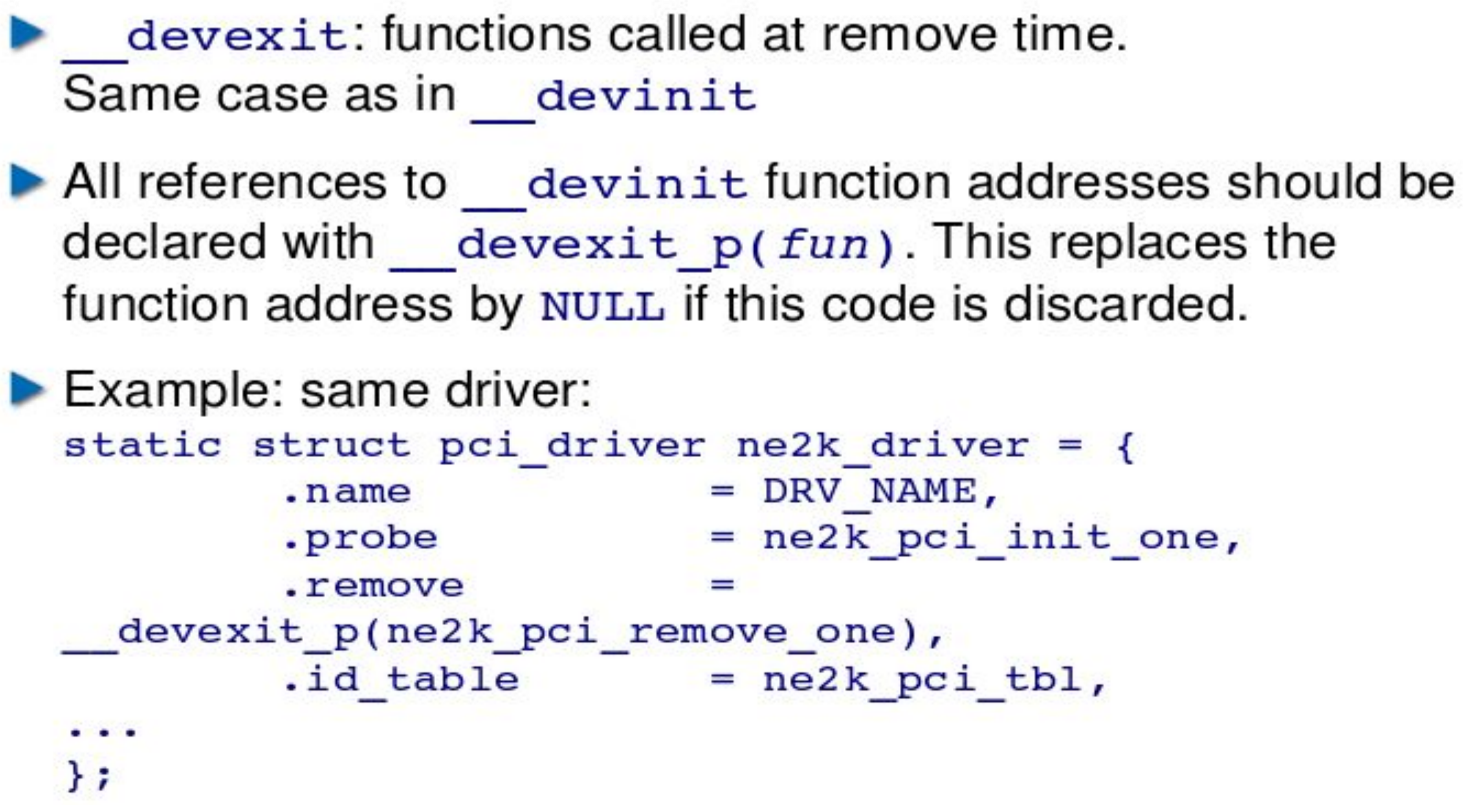
```
static int __init ne2k_pci_init(void)
{
        return pci_register_driver(&ne2k_driver);
}

static void __exit ne2k_pci_cleanup(void)
{
        pci_unregister_driver (&ne2k_driver);
}
```

- The hooks and supported devices are loaded at module loading time.
- The `probe()` hook gets called by the PCI generic code when a matching device is found.
- Very similar to USB device drivers!

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Driver entry points - Example



- `__devexit`: functions called at remove time. Same case as in `__devinit`
- All references to `__devinit` function addresses should be declared with `__devexit_p(fun)`. This replaces the function address by `NULL` if this code is discarded.
- Example: same driver:

```
static struct pci_driver ne2k_driver = {
        .name             = DRV_NAME,
        .probe            = ne2k_pci_init_one,
        .remove           =
__devexit_p(ne2k_pci_remove_one),
        .id_table         = ne2k_pci_tbl,
...
};
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Device Initialization

- Enable the device
- Request I/O port and I/O memory resources
- Set the DMA mask size (for both coherent and streaming DMA)
- Allocate and initialize shared control data (`pci_allocate_coherent()`)
- Initialize device registers (if needed)
- Register IRQ handler (`request_irq()`)
- Register to other subsystems (network, video, disk, etc.)
- Enable DMA/processing engines.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Enabling the device

Before touching any device registers, the driver should first execute `pci_enable_device()`. This will:

- Wake up the device if it was in suspended state
- Allocate I/O and memory regions of the device (if not already done by the BIOS)
- Assign an IRQ to the device (if not already done by the BIOS)

`pci_enable_device()` can fail. Check the return value!

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Enabling PCI Device

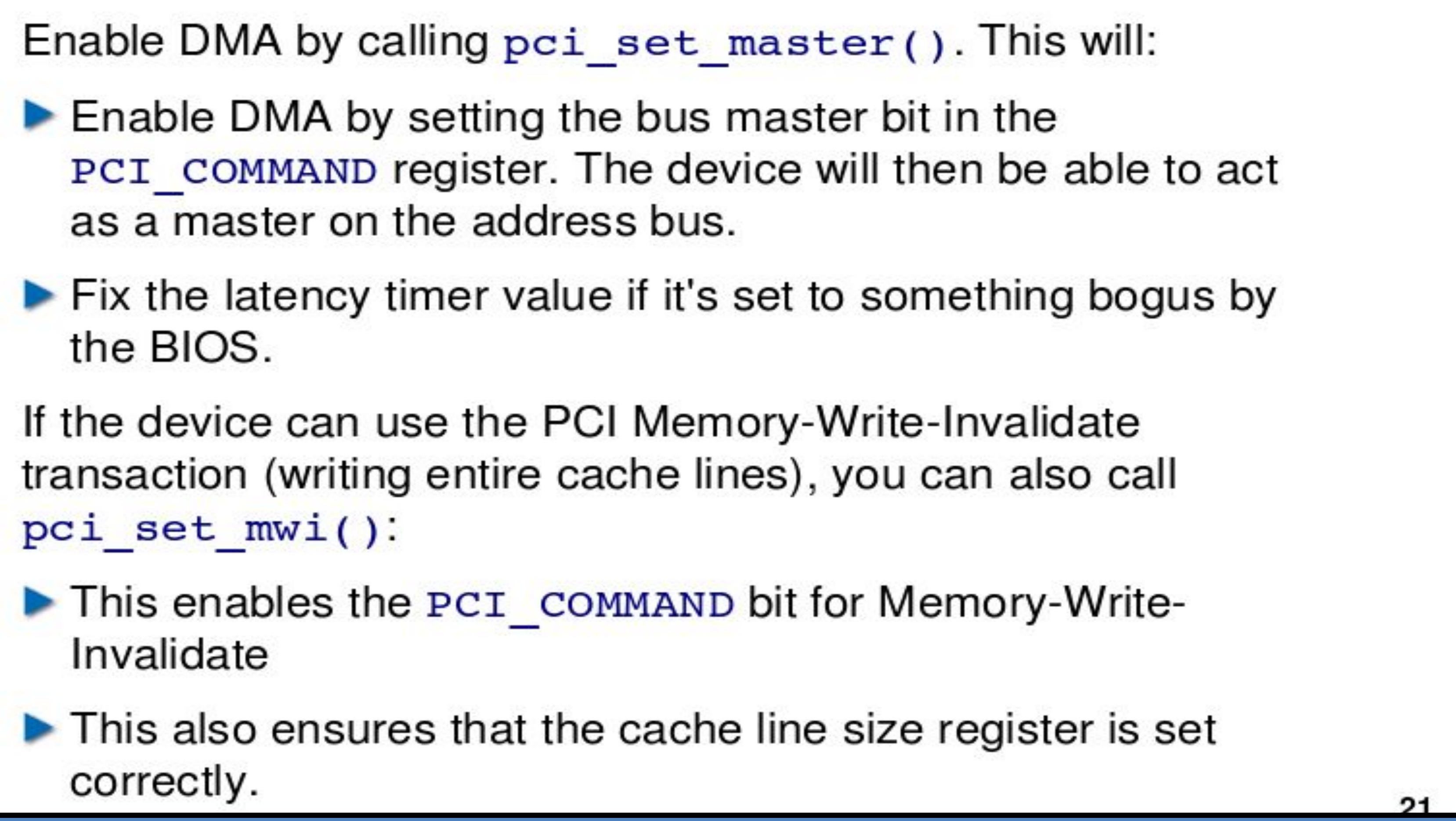
From `drivers/net/ne2k-pci.c` (Linux 2.6.27):

```
static int __devinit ne2k_pci_init_one
    (struct pci_dev *pdev,
     const struct pci_device_id *ent)
{
       ...
       i = pci_enable_device (pdev);
        if (i)
                return i;

        ...
}
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Enabling DMA for PCI Device



Enable DMA by calling `pci_set_master()`. This will:

- Enable DMA by setting the bus master bit in the `PCI_COMMAND` register. The device will then be able to act as a master on the address bus.
- Fix the latency timer value if it's set to something bogus by the BIOS.

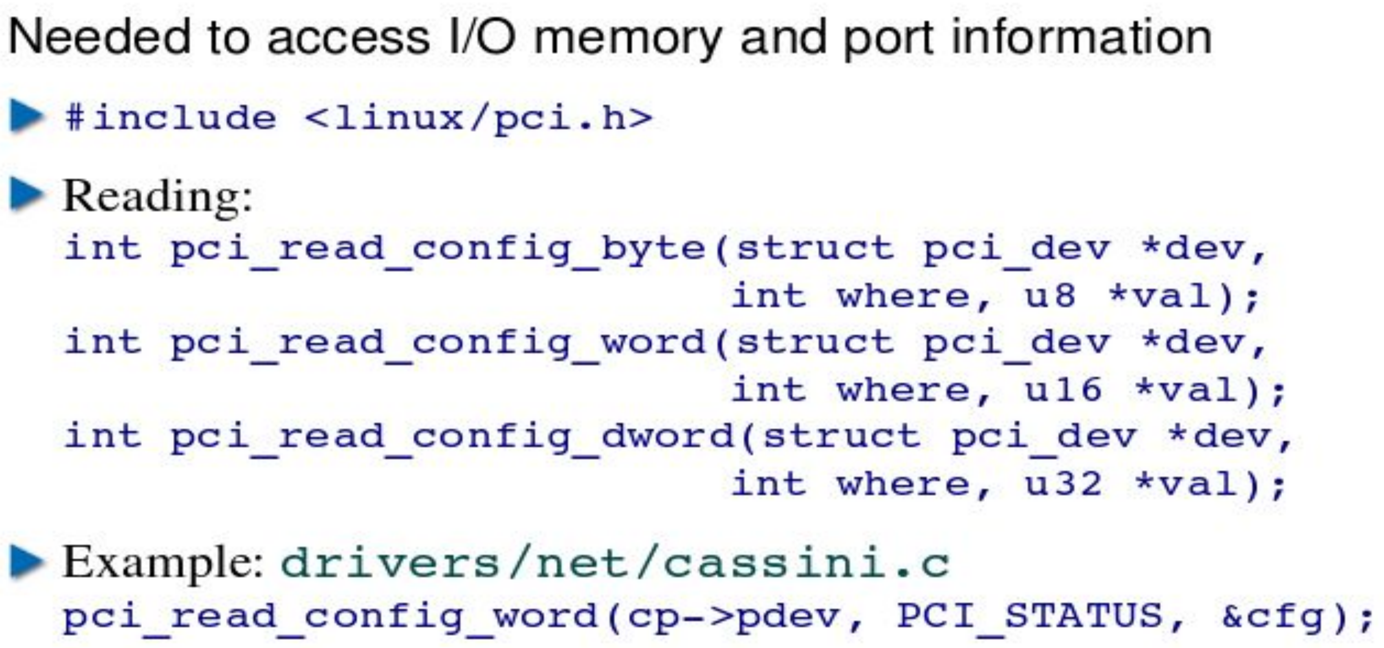
If the device can use the PCI Memory-Write-Invalidate transaction (writing entire cache lines), you can also call `pci_set_mwi()`:

- This enables the `PCI_COMMAND` bit for Memory-Write-Invalidate
- This also ensures that the cache line size register is set correctly.

21

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Accessing PCI configuration Space



Needed to access I/O memory and port information

- `#include <linux/pci.h>`
- Reading:

```
int pci_read_config_byte(struct pci_dev *dev,
                         int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev,
                         int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev,
                         int where, u32 *val);
```

- Example: `drivers/net/cassini.c`

```
pci_read_config_word(cp->pdev, PCI_STATUS, &cfg);
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Accessing PCI configuration space

- To read the IRQ number assigned to a card, offset 60 in pci configuration space.

```
pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &irq);
```

- To read the PCI status register, two bytes at offset six in the configuration space

```
pci_read_config_word(pdev, PCI_STATUS, &status);
```

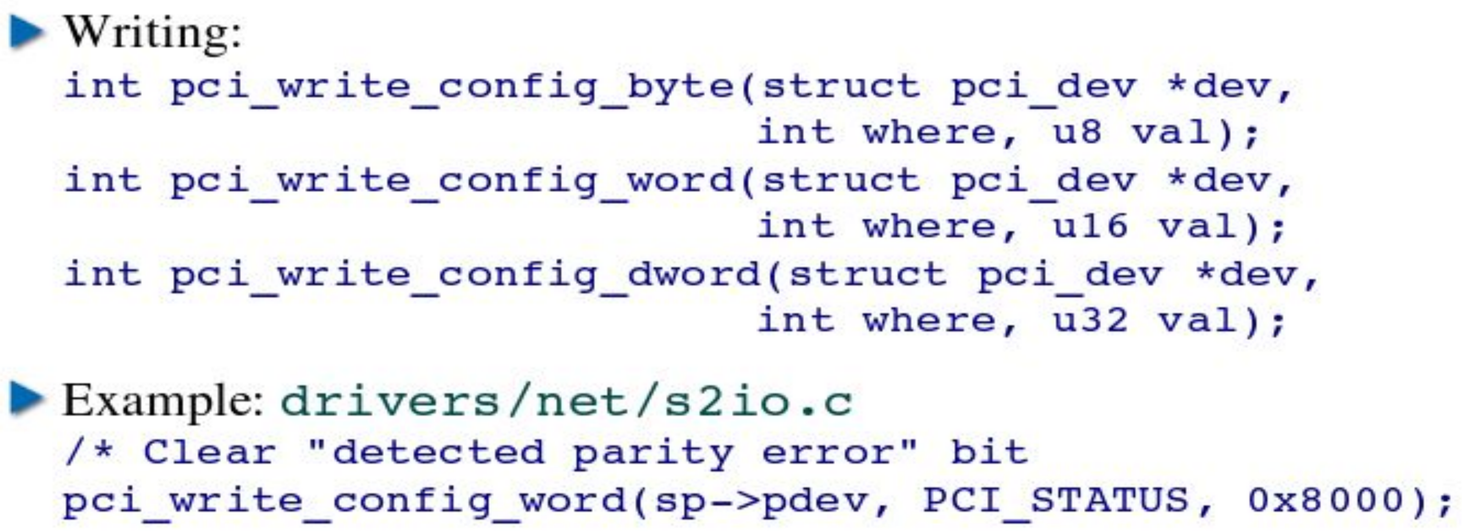
- Only first 64 bytes of the configuration space is standard. The device vendors defines desired semantics to the rest. Xircom card assigns four bytes at offset 64 for power management. To disable power management do this

```
 #define PCI_POWERMGMT 0x40
    pci_write_config_dword(pdev, PCI_POWERMGMT, 0x0000);
```

All configuration register offsets are defined in **include/linux/pci_regs.h**

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Accessing PCI configuration registers



- Writing:

```
int pci_write_config_byte(struct pci_dev *dev,
                          int where, u8 val);
int pci_write_config_word(struct pci_dev *dev,
                          int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev,
                          int where, u32 val);
```

- Example: `drivers/net/s2io.c`

```
/* Clear "detected parity error" bit
pci_write_config_word(sp->pdev, PCI_STATUS, 0x8000);
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

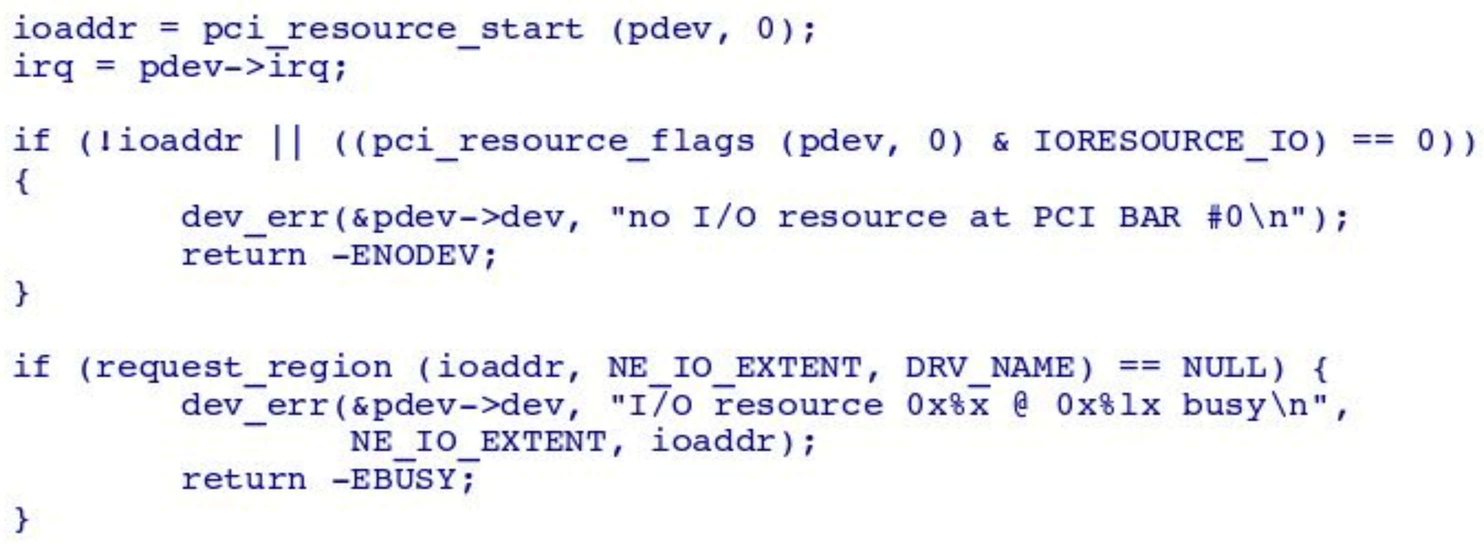
Reserving PCI PMIO or MMIO regions

- Each PCI device can have up to 6 I/O or memory regions, described in `BAR0` to `BAR5`.
- Access the base address of the I/O region:
```
#include <linux/pci.h>
long iobase = pci_resource_start (pdev, bar);
```
- Access the I/O region size:
```
long iosize = pci_resource_len (pdev, bar);
```
- Reserve the I/O region:
```
request_region(iobase, iosize, “my driver”);
```
  or simpler:
```
pci_request_region(pdev, bar, “my driver”);
```
  or even simpler (regions for all BARs):
```
pci_request_regions(pdev, “my driver”);
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Reserving PCI PMIO or MMIO regions

From `drivers/net/ne2k-pci.c` (Linux 2.6.27):



```
ioaddr = pci_resource_start (pdev, 0);
irq = pdev->irq;

if (!ioaddr || ((pci_resource_flags (pdev, 0) & IORESOURCE_IO) == 0))
{
        dev_err(&pdev->dev, "no I/O resource at PCI BAR #0\n");
        return -ENODEV;
}

if (request_region (ioaddr, NE_IO_EXTENT, DRV_NAME) == NULL) {
        dev_err(&pdev->dev, "I/O resource 0x%x @ 0x%lx busy\n",
                NE_IO_EXTENT, ioaddr);
        return -EBUSY;
}
```

Linux/include/linux/ioport.h

```
#define IORESOURCE_IO       0x00000100 /* PCI/ISA I/O ports */
#define IORESOURCE_MEM      0x00000200 /* PCI Memory IO */
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Accessing PCI MMIO region

- To operate on memory regions, follow the same steps:

Get the base address, length, and flags associated with the memory region:

unsigned long mmio_base = pci_resource_start (pdev, bar);

unsigned long mmio_length = pci_resource_length (pdev, bar);

unsigned long mmio_flags = pci_resource_flags (pdev, bar);

- Mark ownership of this region using:

reguest_mem_region(mmio_base, mmio_length, “my_driver”);

or called a wrapper function: pci_request_region()

- Use the ioremap to obtain kernel virtual addresses corresponding to the mapped region:

```
void __iomem *buffer;
buffer = ioremap(mmio_base, mmio_length);
```

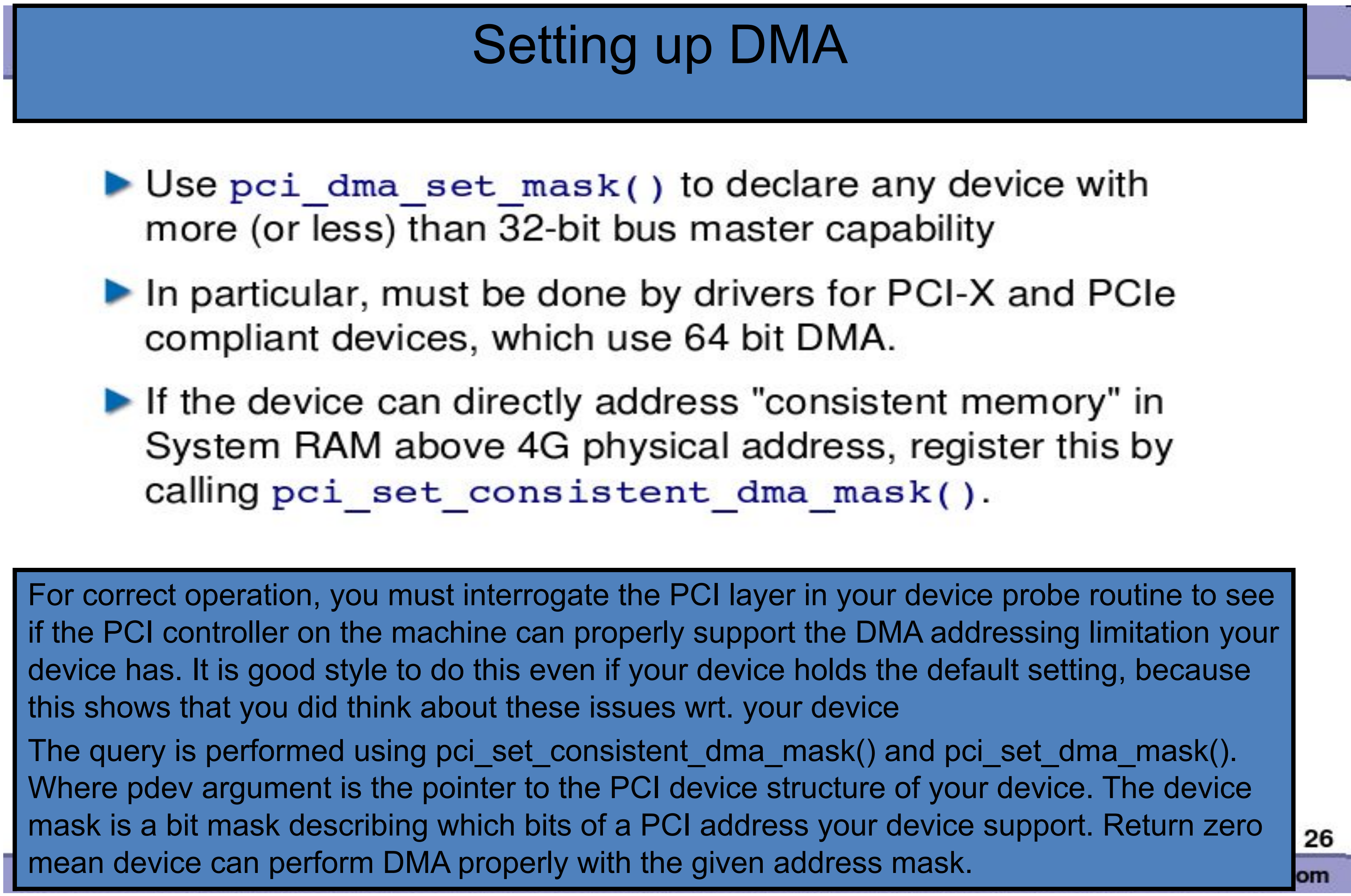
- You can also use:

buffer = pci_iomap(pdev, bar, mmio_length);

NOTE: ioremap() and ioremap_nocache() have same effect on x86.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Setting up DMA



- Use `pci_dma_set_mask()` to declare any device with more (or less) than 32-bit bus master capability
- In particular, must be done by drivers for PCI-X and PCIe compliant devices, which use 64 bit DMA.
- If the device can directly address "consistent memory" in System RAM above 4G physical address, register this by calling `pci_set_consistent_dma_mask()`.

26

om

For correct operation, you must interrogate the PCI layer in your device probe routine to see if the PCI controller on the machine can properly support the DMA addressing limitation your device has. It is good style to do this even if your device holds the default setting, because this shows that you did think about these issues wrt. your device

The query is performed using pci_set_consistent_dma_mask() and pci_set_dma_mask(). Where pdev argument is the pointer to the PCI device structure of your device. The device mask is a bit mask describing which bits of a PCI address your device support. Return zero mean device can perform DMA properly with the given address mask.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Setting up DMA

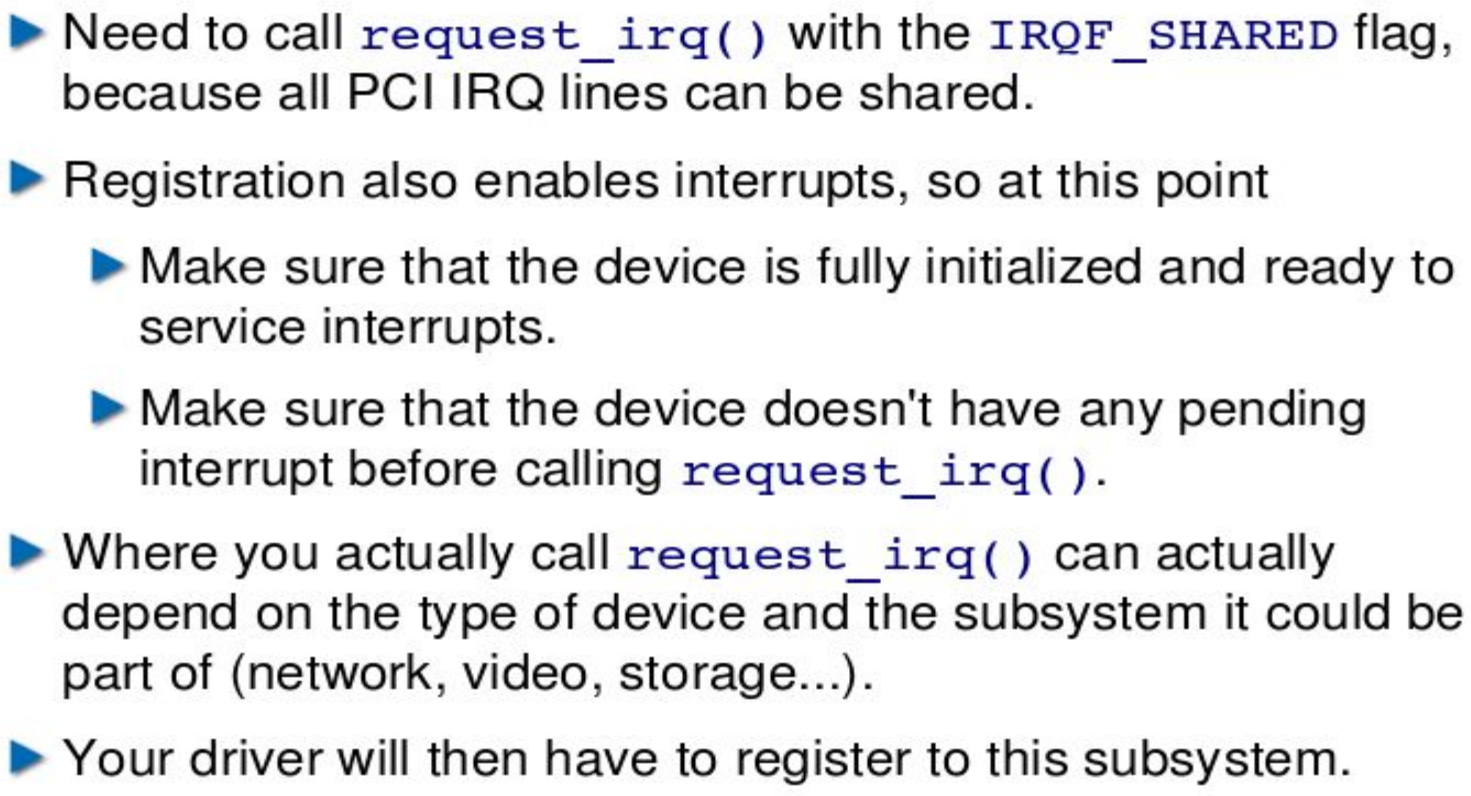
Example (`drivers/net/wireless/ipw2200.c` in Linux 2.6.27):

```
err = pci_set_dma_mask(pdev, DMA_32BIT_MASK);

if (!err)
    err = pci_set_consistent_dma_mask(pdev, DMA_32BIT_MASK);
if (err) {
    printk(KERN_WARNING DRV_NAME ": No suitable DMA available.\n");
    goto out_pci_disable_device;
}
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Interrupt Handling



- Need to call `request_irq()` with the `IRQF_SHARED` flag, because all PCI IRQ lines can be shared.
- Registration also enables interrupts, so at this point
  - Make sure that the device is fully initialized and ready to service interrupts.
  - Make sure that the device doesn't have any pending interrupt before calling `request_irq()`.
- Where you actually call `request_irq()` can actually depend on the type of device and the subsystem it could be part of (network, video, storage...).
- Your driver will then have to register to this subsystem.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

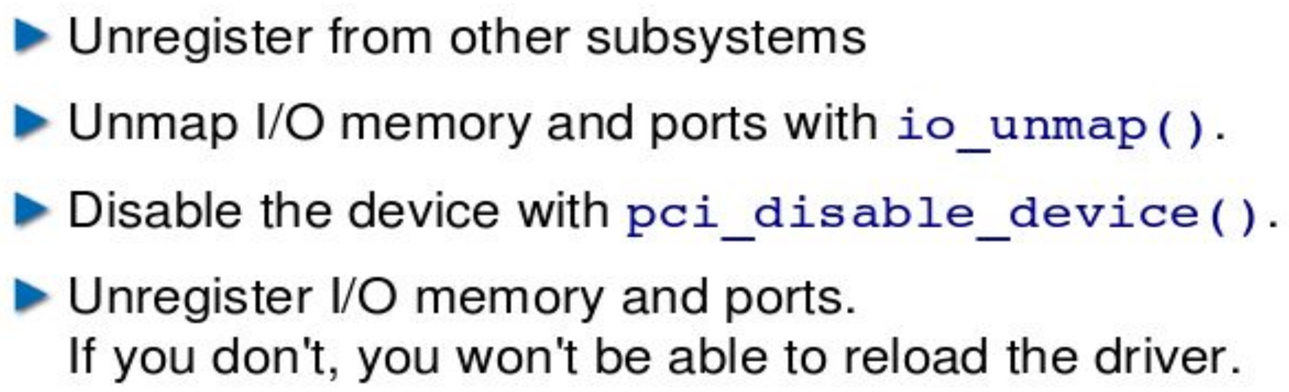
PCI Device Shutdown

In the `remove()` function, you typically have to undo what you did at device initialization (`probe()` function):

- Disable the generation of new interrupts.
  If you don't, the system will get spurious interrupts, and will eventually disable the IRQ line. Bad for other devices on this line!
- Release the IRQ
- Stop all DMA activity.
  Needed after IRQs are disabled (could start new DMAs)
- Release DMA buffers: streaming first and then consistent ones.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Device Shutdown



- Unregister from other subsystems
- Unmap I/O memory and ports with `io_unmap()`.
- Disable the device with `pci_disable_device()`.
- Unregister I/O memory and ports.
  If you don't, you won't be able to reload the driver.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Storing Data in your pci_driver

You can stuff device specific data in the pci_driver structure and retrieve it later

void pci_set_drvdata(struct pci_driver *, void *)

  Associates the given data with the given pci_driver structure

void * pci_get_drvdata(struct pci_driver *)

  Returns the data associated with the given pci_driver structure

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

pci iomap

- Instead of generic ioremap(), you can use pci specific function to map IO device memory into kernel virtual memory
- void * pci_iomap(struct pci_dev *pdev, int bar, unsigned int max)
- bar holds the BAR
  - Base Address Register
  - Where to start the mapping
- max is the amount to map (i.e. size)
- Once mapped, you can read and write to device memory:
- int ioread32(void *iomap)
  - Reads and returns the 32-bit word starting at the given address
- void iowrite32(u32 word, void *iomap)
  - Writes word to iomap
- Unmap with pci_iounmap(struct pci_dev *pdev, void *iomap)

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Review
PCI Driver

- Describe to the PCI layer the devices that this driver supports
```
static struct pci_device_id foo_id[] = {
 { 0x1111, 0x0201, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
 { 0 }
};
```
  - This driver is saying that it supports a device with a vendor ID of 0x1111, a device ID of 0x0201, with any subvendor and subdevice ID's. Array is zero-terminated
- Register PCI driver call back functions (probe, suspend, remove) and PCI ID table via pci_driver structure.
- The PCI layer will automatically calls your probe function for any matching device
- Your probe function can use pci_set_drvdata() to store data that you need during remove
- Driver can determine the interrupt number assigned to the device directly from pci device structure pdev (pdev->irq), passed as argument to pci probe.
- Register PCI device via:
```
    pci_register_driver(struct pci_driver *pdev);
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

Handle Endianness

Use these functions everywhere instead to be portable.

```
#define write_register(value, iomem, member) \
        iowrite32(cpu_to_le32(value),          \
            (iomem) + offsetof(struct lr3k_regs, member))


#define read_register(iomem, member) \
    le32_to_cpu(ioread32((iomem) + \
        offsetof(struct lr3k_regs, member)))
```

Copyright © 2010 The Regents of the University of California. All Rights Reserved.

PCI Device Driver
Need to know

When writing a device driver for PCI hardware, you need to do the following:

- Access device registers via PMIO or MMIO interface
  - Although the registers of devices on the PCI bus are typically mapped in both I/O space and memory space, it is strongly recommended that device drivers use the mappings in memory space (**MMIO**).
  - A device driver can locate the base location or starting address of a device's registers in memory space by obtaining the I/O handle from the appropriate **BAR** (Base Address Register) by accessing the PCI device configuration space.
  - Driver can then map the device IO memory using **ioremap()** and then read from or write to a device register using **ioread/iowrite[b|l|w]** and other pci portable functions.
- Performing DMA memory transfers
  - device driver perform a direct memory access (DMA) operation. It allows driver and device to transfer data between contiguous I/O bus addresses and physical memory addresses
  - DMA interfaces are designed to be CPU and bus hardware independent, thus makes the driver more portable across different CPU architectures and more than one CPU type within the same architecture, as well as across I/O buses.
- Registering device interrupt handlers
  - For devices attached to a PCI bus, you register the interrupt handlers within the driver's probe routine. Driver can always find the IRQ number assigned from the PCI device configuration space.

Copyright © 2010 The Regents of the University of California. All Rights Reserved.