


# Linux PCI drivers

VEDA Solutions
Inspired by open Source

# PCI bus family

The following buses belong to the PCI family:

- PCI
  32 bit bus, 33 or 66 MHz
- MiniPCI
  Smaller slot in laptops
- CardBus
  External card slot in laptops
- PIX Extended (PCI-X)
  Wider slot than PCI, 64 bit, but can accept a standard PCI card
- PCI Express (PCIe or PCI-E)
  Current generation of PCI. Serial instead of parallel.
- PCI Express Mini Card
  Replaces MiniPCI in recent laptops
- Express Card
  Replaces CardBus in recent laptops

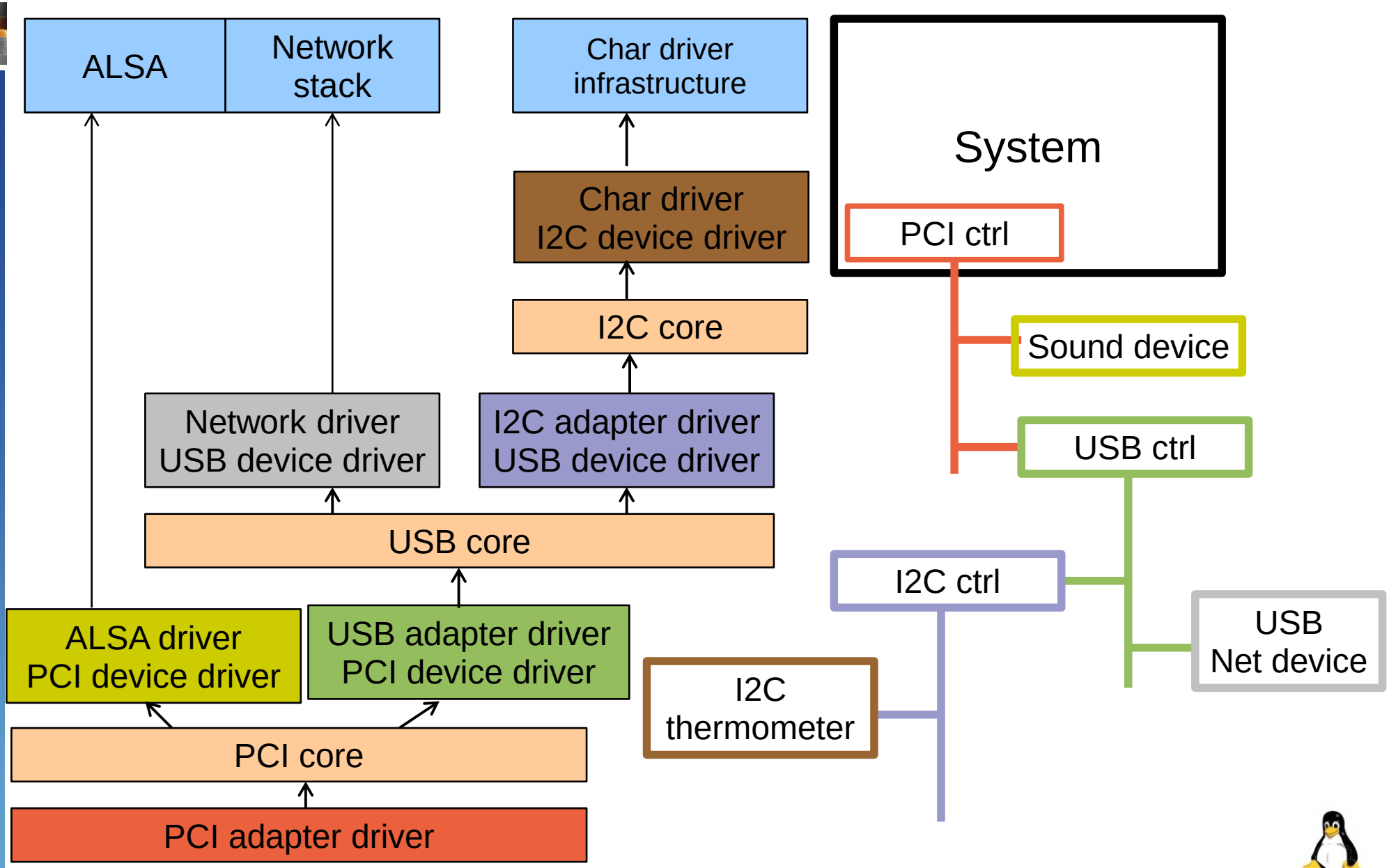
These technologies are compatible and can be handled by the same kernel drivers. The kernel doesn't need to know which exact slot and bus variant is used.

# PCI device types

Main types of devices found on the PCI bus

- Network cards (wired or wireless)
- SCSI adapters
- Bus controllers: USB, PCMCIA, I2C, FireWire, IDE
- Graphics and video cards
- Sound cards

# Driver stack



# PCI features

For device driver developers

- Device resources (I/O addresses, IRQ lines) automatically assigned at boot time, either by the BIOS or by Linux itself (if configured).
- The device driver just has to read the corresponding configurations somewhere in the system address space.
- Endianism: PCI device configuration information is Little Endian. Remember that in your drivers
(conversion taken care of by some kernel functions).

# Registering a driver (1)

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 */
};
```

# Registering a driver (2)

```
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!

# Marking driver hooks (1)

- __init: module init function.

Code discarded after driver initialization.

- __exit: module exit function.

Ignored for statically compiled drivers.

- __devinit: probe function and all initialization functions

Normal function if CONFIG_HOTPLUG is set. Identical to __init otherwise.

- __devinitconst: for the device id table

# Marking driver hooks (2)

- __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,
...
};
```

# Device initialization steps

- 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.

# 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!

# Enabling the device (2)

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.

# Accessing I/O registers and memory (1)

- 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”);
```

# Accessing I/O registers and memory (2)

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;
}
```

# Setting the DMA mask size (1)

- 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().

# Setting the DMA mask size (2)

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;
}
```

# Allocate consistent DMA buffers

Now that the DMA mask size has been allocated...

- You can allocate your cache consistent buffers if you plan to use such buffers.
- See our DMA presentation and Documentation/DMA-API.txt for details.

# Initialize device registers

If needed by the device

- Set some “capability” fields
- Do some vendor specific initialization or reset

Example: clear pending interrupts.

# Register interrupt handlers

- 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.

# PCI device shutdown (1)

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 to be done after IRQs are disabled (could start new DMAs)

- Release DMA buffers: streaming first and then consistent ones.

# 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.