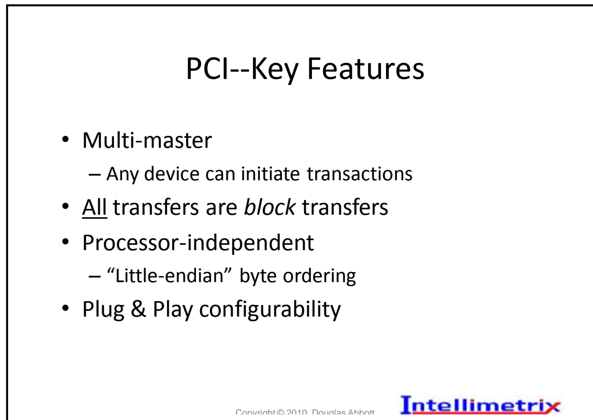
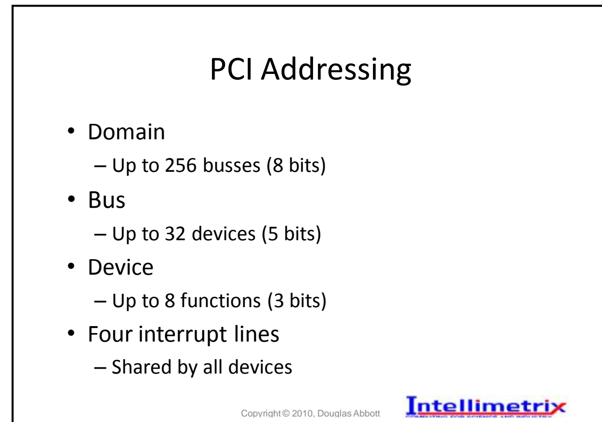
Slide 148



The PCI architecture was designed as a replacement for the original PC ISA bus standard, with three main goals in mind: to get better performance when transferring data between the computer and its peripherals, to be as platform independent as possible, and to simplify adding and removing peripherals to the system.
Features that support those goals include:

- Multi-master, meaning any device on the bus can be a bus master and initiate transactions. One consequence of this is that there is no need for the traditional notion of DMA.
- The transfer protocol is optimized around transferring blocks of data. A single transfer is just a block transfer with a length of one.
- Although PCI is officially processor-independent, it inevitably reflects its origins with Intel and its primary application in the PC architecture. Among other things it uses little-endian byte ordering.
- PCI implements Plug-and-Play configurability. This in fact is the main issue that affects device driver writers.

Slide 149

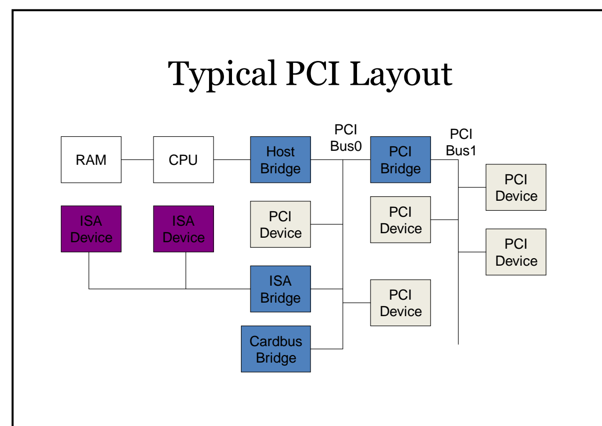


Every PCI device has a unique address, or ID, that consists of three components:

- The domain is a universe of up to 256 busses.
- A bus can support up to 32 devices.
- A device, in turn, can contain up to 8 functions.
- PCI incorporates four interrupt lines that are shared by all devices.

So, from a software standpoint, every PCI device can be uniquely identified by a 16-bit address.

Slide 150



This slide illustrates how a system can be organized into multiple busses connected together by bridges. A bridge is a special kind of PCI device whose function is to connect two busses together. The bus

Copyright © 2010 Douglas Abbott

closes to the processor is always designated Bus 0 and connects to the CPU through the host bridge. Many systems still support ISA peripherals. These connect to a PCI bus through an ISA to PCI bridge. Likewise, Cardbus peripherals connect through a Cardbus to PCI bridge.
Bridges are generally transparent to driver writers, but it's useful to understand how the system is structured.

Slide 151

/proc files and pciutils

- /proc/bus/pci/devices
- /proc/bus/pci/<bus_number>
- lspci

The kernel offers some tools to help you see what's on the PCI busses in your system. The file /proc/bus/pci/devices lists basic identifying information about the devices in the system, although, to be honest, the format isn't immediately clear. There are also one or more directories of the form /proc/bus/pci/<bus_number> containing files for the devices on each bus. These turn out to be binary and are the configuration space for the device.

As part of a package called pciutils, there's a program called lspci that takes information from various proc files and presents it in a reasonably useful form.

Slide 152

PCI Address Spaces

- Memory
  - Shared by all devices on bus
  - 32- or 64-bit addressing
- I/O
  - Shared by all devices on bus
  - 32-bit addressing
- Configuration
  - "Geographical" addressing, per PCI slot
  - 256 bytes per device function

The PCI bus defines three distinct address spaces with corresponding read and write commands. Memory space is shared by all devices on the bus and may have either a 32-bit or 64 bit address. I/O space is likewise shared by all devices on the bus, but is limited to a 32-bit address. The principal distinction between memory and I/O space is that memory read access has no side effects and thus it is pre-fetchable.

Configuration space is something altogether different. It implements "geographical" addressing where each physical PCI slot gets a unique configuration space address. Each device function gets 256 bytes of configuration space. Configuration address space is used only at bootup time to configure the community of PCI cards in a system.

Slide 153

Configuration Space

- Geographically addressed

0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xb 0xc 0xd 0xe 0xf

0x00	Vendor ID	Device ID	Cmd Reg	Status Reg	Rev ID	Class Code			
0x10	Base Address 0	Base Address 1	Base Address 2	Base Address 3					
0x20	Base Address 4	Base Address 5	Cardbus CIS Pointer	Sub Vendor ID	Sub Device ID				
0x30	Expansion ROM Base Address	Reserved	IRQ Line	IRQ Pin					

The first 64 bytes of a logical function's configuration space constitute a *header* whose format and content are defined in the specification. Fields such as Vendor ID and Device ID identify a device

Copyright © 2010 Douglas Abbott

and are used by the kernel to find a driver that can service the device.
A PCI function can have up to six independent address spaces in either memory or I/O space. At boot time, the PCI configuration software sets the Base Address registers to give each function unique addresses in the memory and I/O address spaces. Subsequently, the device driver interrogates these registers to determine where its device is located.

Slide 154

Device ID

- Vendor and subvendor
  - Vendor ID assigned by PCI SIG
- Device and subdevice
  - Assigned by vendor
- Class and class mask
  - Category of device
- linux/pci_ids.h

Let's take a closer look at this issue of device identification because it plays a major role in linking physical devices with the drivers that service them.

All devices have a vendor ID, which identifies the manufacturer of the device. It may optionally have a subsystem vendor ID as well. Vendor IDs are assigned and maintained by the PCI Special Interest Group and manufacturers must apply to have a unique number assigned to them.

All devices also have a device ID, which is assigned by the vendor. The 32-bit combination of vendor ID and device ID is often called a *signature*. This is what drivers usually rely on to identify their devices.

Devices also belong to a class, which is a way of identifying a device by its basic functionality.

The header file pci_ids.h defines symbols for all the classes and subclasses as well as a large number of vendors and their device IDs.

Slide 155

Device ID structure

```
struct pci_device_id {
        __u32 vendor, device;
        __u32 subvendor, subdevice;
        __u32 class, class_mask;
        kernel_ulong_t driver_data;
};
PCI_DEVICE (vendor, device)
PCI_DEVICE_CLASS (dev_class, dev_class_mask)

struct pci_device_id ids[] = {
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
        PCI_DEVICE_ID_INTEL_82801AA_3), },
        { 0, }    /* terminating entry */
};
MODULE_DEVICE_TABLE (pci, ids);
```

Every PCI driver has an array of struct pci_device_ids that identifies the devices that the driver supports. Typically a driver will support some set of devices from a particular vendor, or it may support a specific class of devices regardless of vendor. The kernel provides a couple of "helper" macros to simply the process of filling out this table. The driver_data field allows the driver to differentiate among the entries in the table.

The table is terminated by a NULL entry. The example shown here identifies a single Intel device followed by the NULL termination. The arguments to the PCI_DEVICE macro are symbols taken from the pci_ids.h header file.

The macro MODULE_DEVICE_TABLE makes ids externally visible. When you build a kernel and all of its driver modules, a program called depmod scans all of the driver modules looking for device tables created by MODULE_DEVICE_TABLE and extracts the information into a master file of supported PCI devices. This is how the kernel determines which drivers to load at boot time after determining which devices are present.

Copyright © 2010 Douglas Abbott

Slide 156

PCI Configuration

- Done at boot time by BIOS or boot loader
  - Uses Configuration Space to allocate unique Memory and I/O Space to each device
- Kernel determines which drivers to load based on found devices
  - Uses /lib/modules/<kernel_vers>/modules.pcimap to link devices with drivers

PCI configuration is done at boot time by either the BIOS or the boot loader. The configuration software allocates a unique chunk of memory and/or I/O space to each device.

When the kernel boots up, it scans the devices in the system and then consults the file generated by depmod to determine which drivers to load.

Slide 157

Registering a PCI Driver

```
struct pci_driver {
        const char *name;
        const struct pci_device_id *id_table;
        int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
        void (*remove) (struct pci_dev *dev);
        int (*suspend) (struct pci_dev *dev, u32 state);  // optional
        int (*resume) (struct pci_dev *dev);              //  optional
}

int pci_register_driver (struct pci_driver *drv);
```

Like all the drivers we've seen up 'til now, a PCI driver must be registered. The struct pci_driver includes a name that will be used by /proc/devices, a pci_device_id table, and a set of function pointers. The probe function is called by the kernel when it has a device that it thinks this driver might be able to handle. The driver returns 0 if it claims the device.

remove is called when a device is being removed from the system. This would only apply to hot-plug environments such as Compact PCI. suspend is called when a device is being suspended. Subsequently, resume will be called to resume device operation. These last two functions are optional. pci_register_driver returns either zero for success or a negative error code.

Slide 158

Accessing Configuration Space

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

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

- Symbolic values for offset in pci.h

Once a driver has claimed a device, it usually needs to read the configuration space to determine where its device is located and perhaps determine what interrupt line it's connected to. A set of functions is provided to read and write configuration space by bytes, words, or longs. The header file pci.h provides symbolic names for the offsets.

Slide 159

Where is Memory and I/O Space?

- Up to 6 Base Address Registers, 0 to 5

```
unsigned long pci_resource_start (struct pci_dev *dev, int bar);
unsigned long pci_resource_end (struct pci_dev *dev, int bar);
unsigned long pci_resource_flags (struct pci_dev *dev, int bar);
```

While it is possible to get the necessary address information by reading the configuration space directly, the functions shown here provide an easier way to get it.

pci_resource_start returns the beginning address of the region defined by the Base Address register specified in bar. pci_resource_end returns the last address associated with the specified bar region. Note that this is the last useable address, not the first address after the region.

Copyright © 2010 Douglas Abbott

pci_resource_flags returns the flags associated with the region. The most important of these specify whether the region is memory space or I/O space.

Copyright © 2010 Douglas Abbott