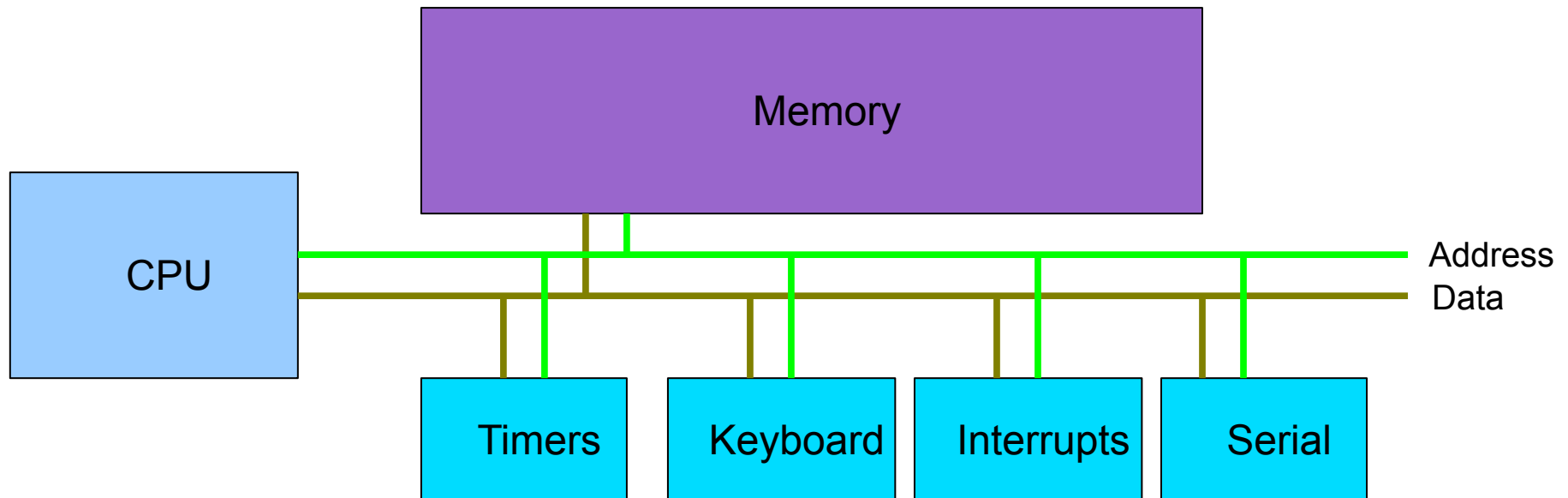# PCI Drivers

ECE 373

Talking to Hardware

- Fast memory

- Slow everything else

  – Clock, printer, UART, keyboard, mouse, interrupt controller, soundboard, disk controllers, joystick, ..

Memory Mapped I/O

- Memory spaces reserved for devices

- Read and write to memory locations tied to devices

- Easy to design and implement
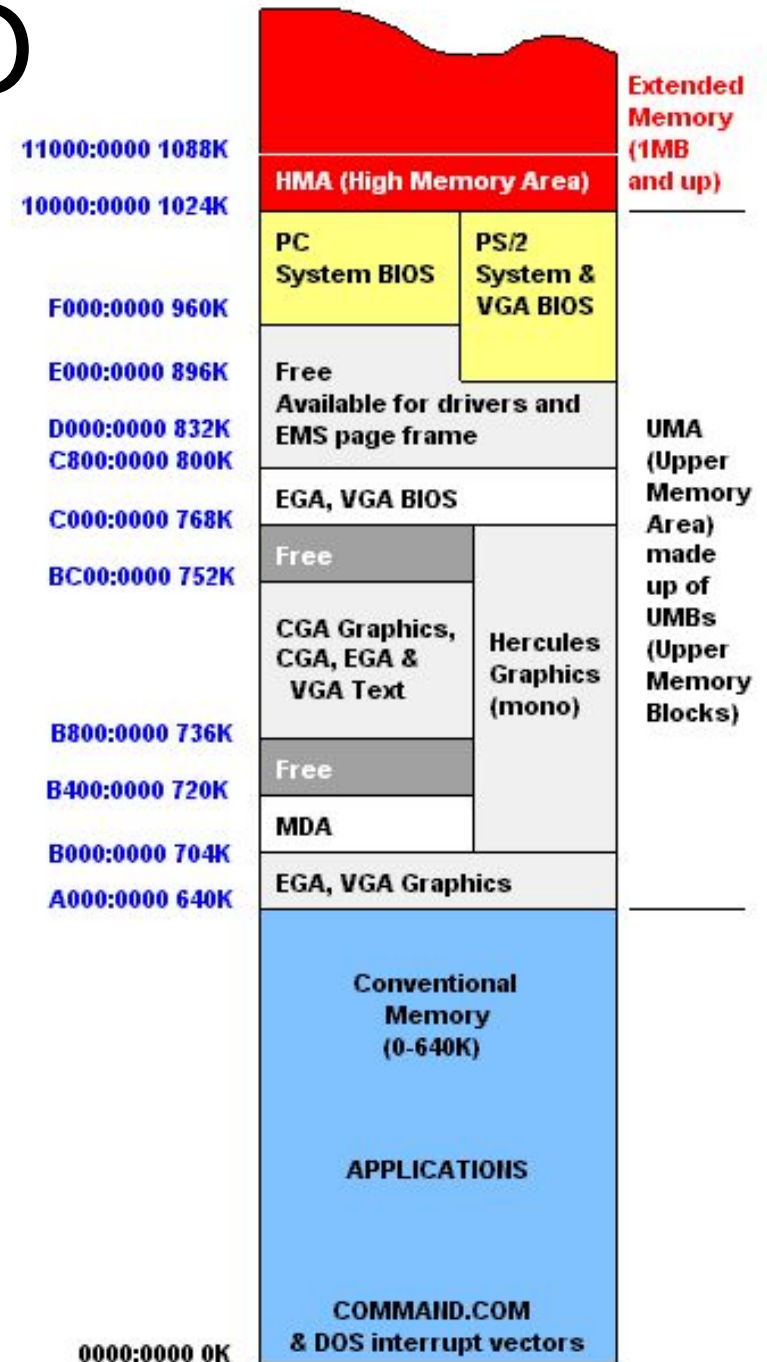
- Easy to program

```
#define VIDEO_BASE   0x800000
#define MAX_X        1024
#define MAX_Y        1024

void draw_pixel(int x, int y, pixel)
{
        *(VIDEO_BASE + (x * MAX_X) + y) = pixel;
}
```
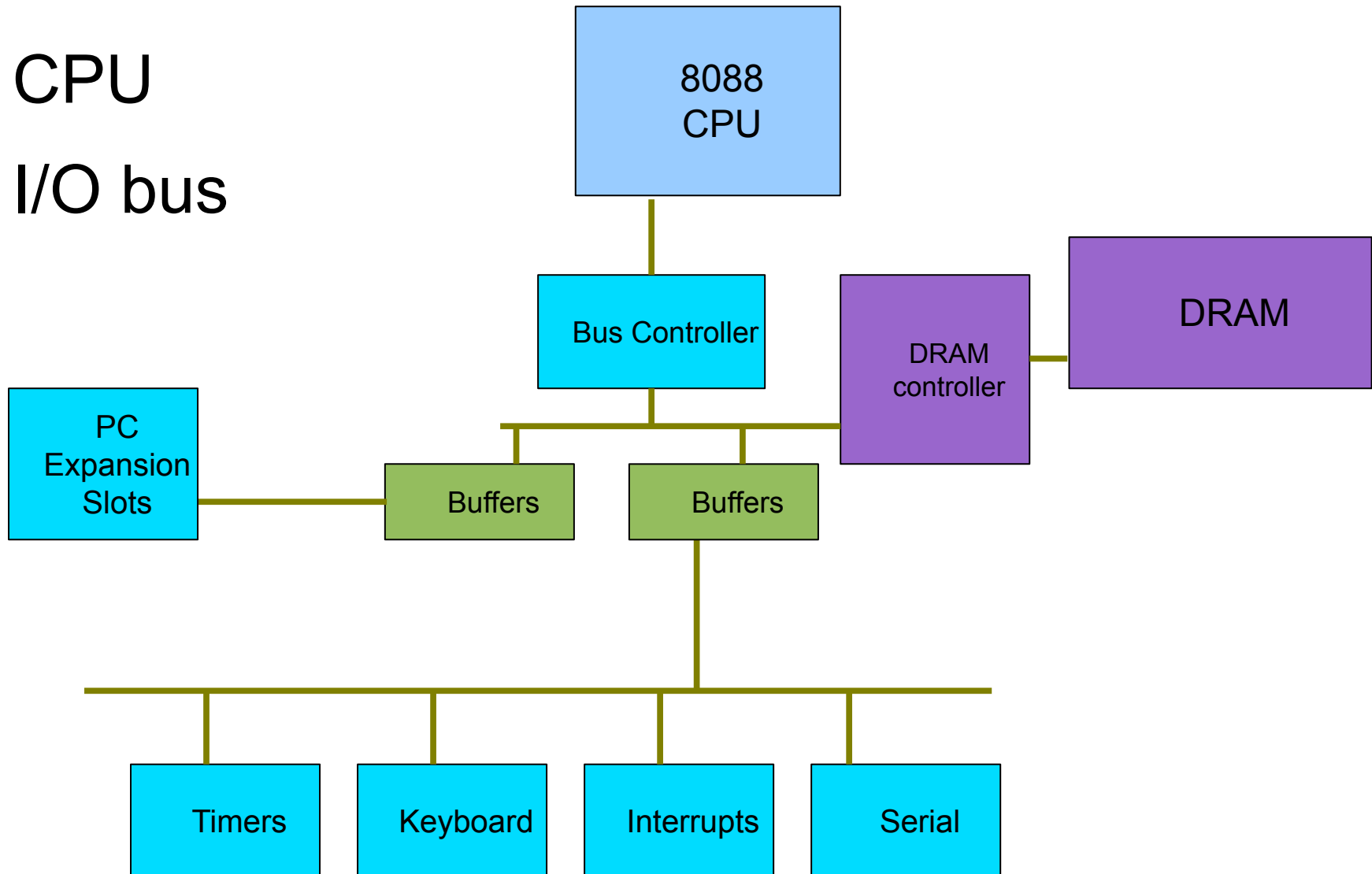
Memory Mapped I/O

- Example: IBM PC

- *Messy*!

  - Memory mapped graphics and other gee-gaws

  - PC architecture is *still* dealing with this mess

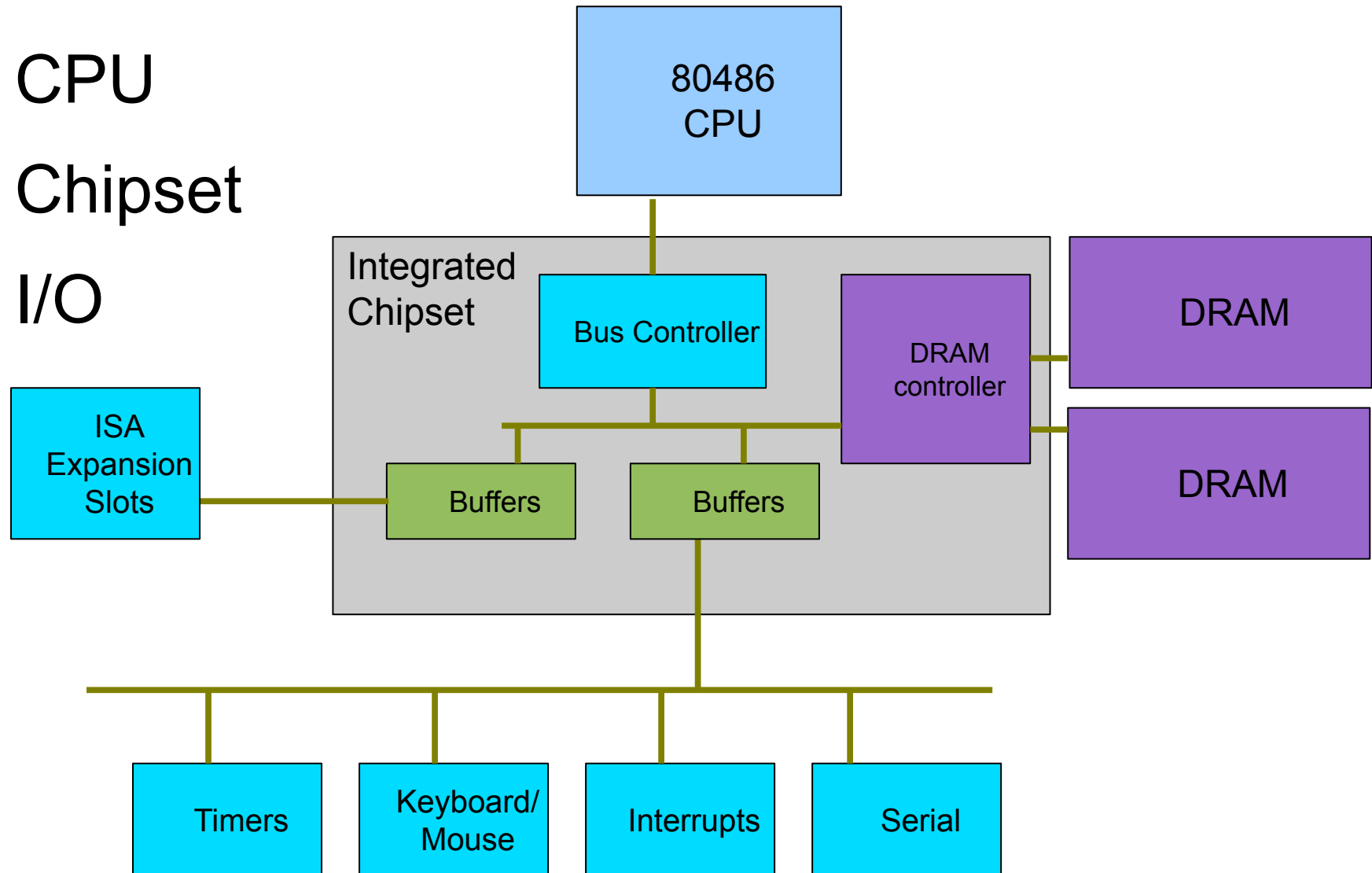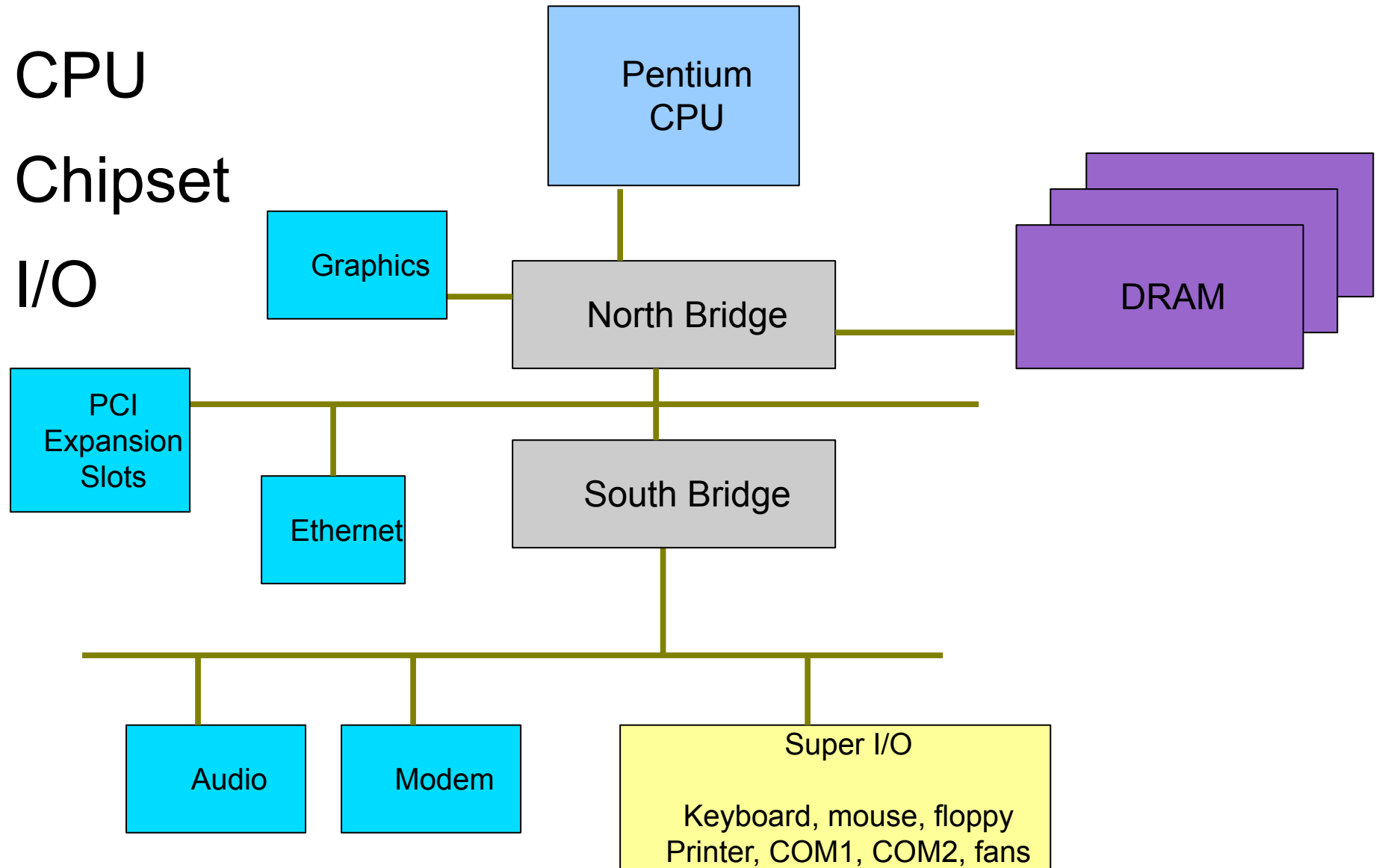 - Shiny new Alder Lake looks exactly like a 80386 when it powers on...

11000:0000 1088K

10000:0000 1024K — HMA (High Memory Area)

Extended Memory (1MB and up)

F000:0000 960K — PC System BIOS / PS/2 System & VGA BIOS

E000:0000 896K — Free Available for drivers and EMS page frame

D000:0000 832K
C800:0000 800K

UMA (Upper Memory Area) made up of UMBs (Upper Memory Blocks)

EGA, VGA BIOS

C000:0000 768K

Free

BC00:0000 752K

CGA Graphics, CGA, EGA & VGA Text / Hercules Graphics (mono)

B800:0000 736K — Free

B400:0000 720K — MDA

B000:0000 704K — EGA, VGA Graphics

A000:0000 640K

Conventional Memory (0-640K)

APPLICATIONS

COMMAND.COM & DOS interrupt vectors

0000:0000 0K

Early PCs

- CPU
- I/O bus

Later PCs

- CPU
- Chipset
- I/O

Later PCs

- CPU
- Chipset
- I/O

Pentium CPU

Graphics

North Bridge

DRAM

PCI Expansion Slots

Ethernet

South Bridge

Audio

Modem

Super I/O

Keyboard, mouse, floppy
Printer, COM1, COM2, fans

(Mostly) Current PCs

- CPU
- Chipset
- I/O

Graphics

Atom CPU

DRAM

PCIe Expansion Slots

Ethernet

ICH

Audio

Super I/O

Keyboard, mouse, floppy
Printer, COM1, COM2, fans
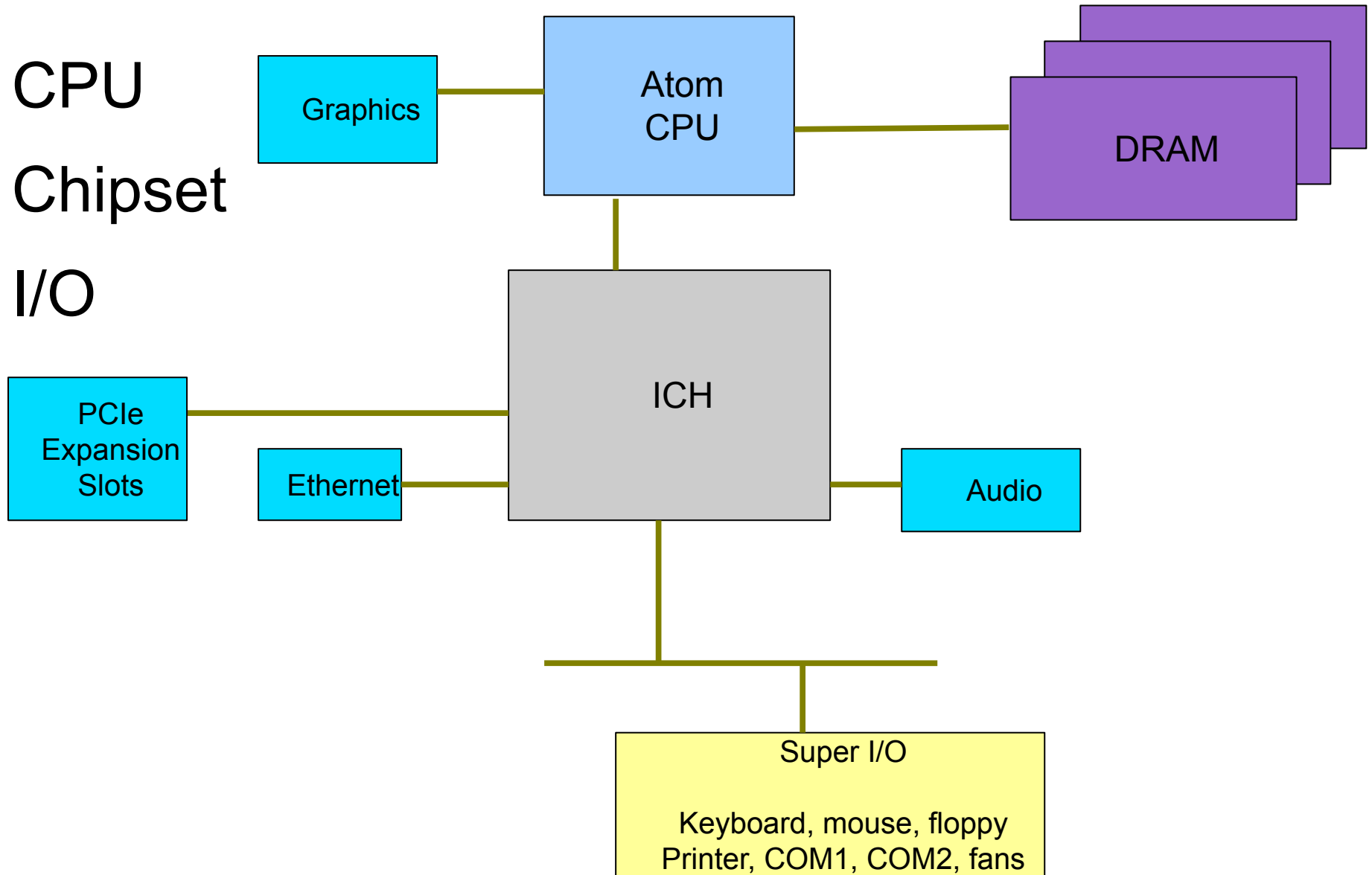
Current "PCs"

- SoC
- Chipset
- I/O

PCI at a glance

©2001 HowStuffWorks
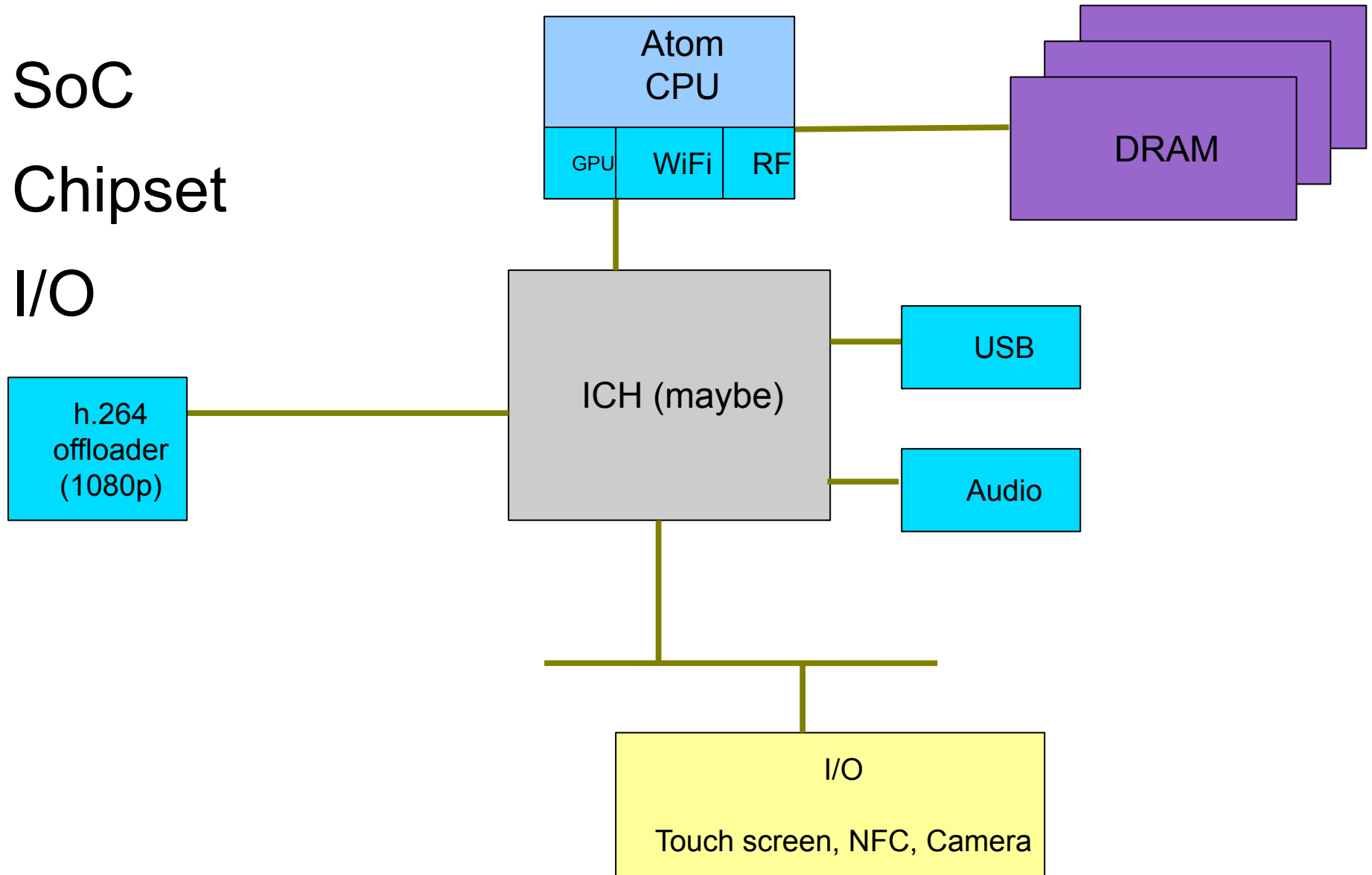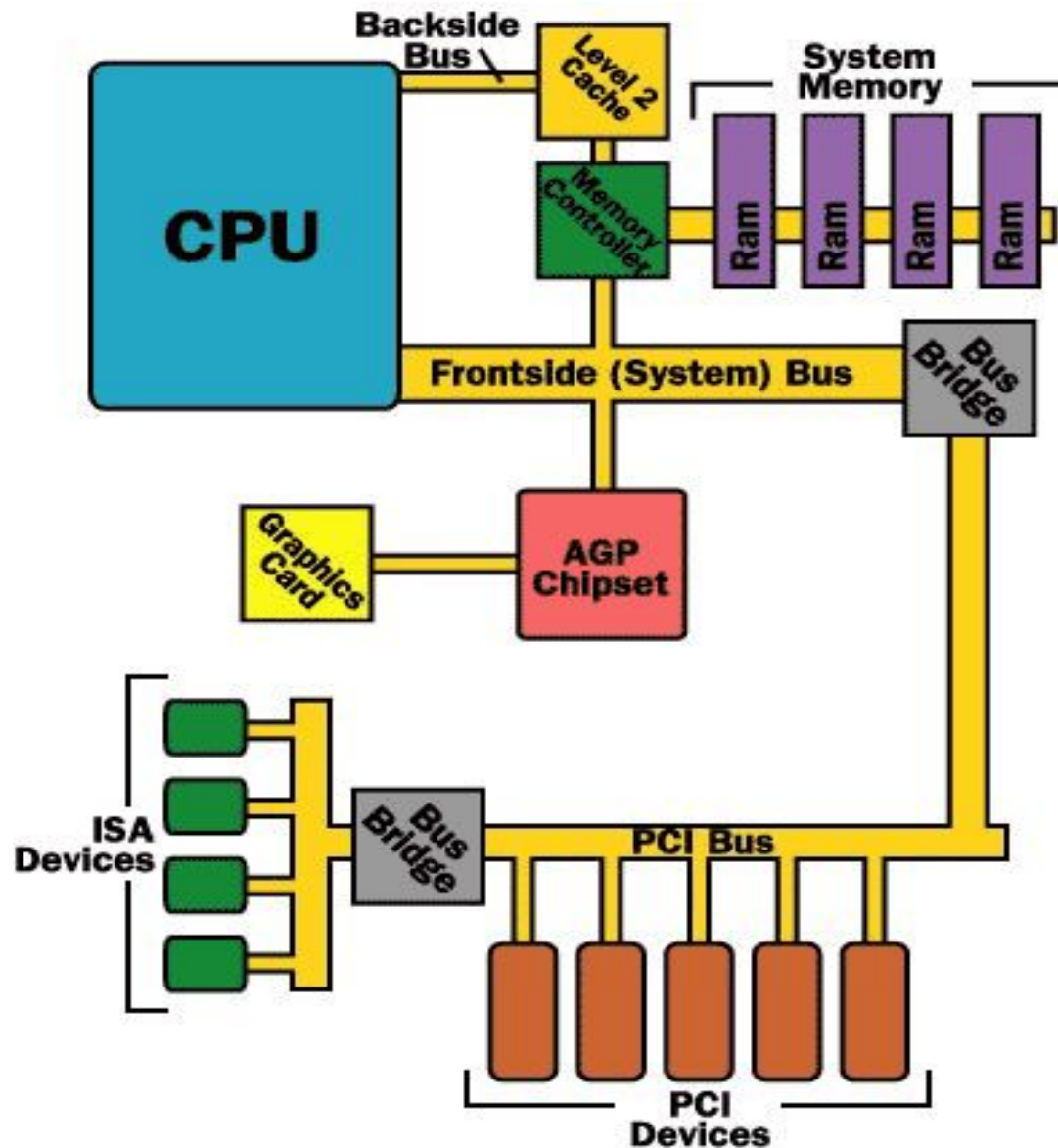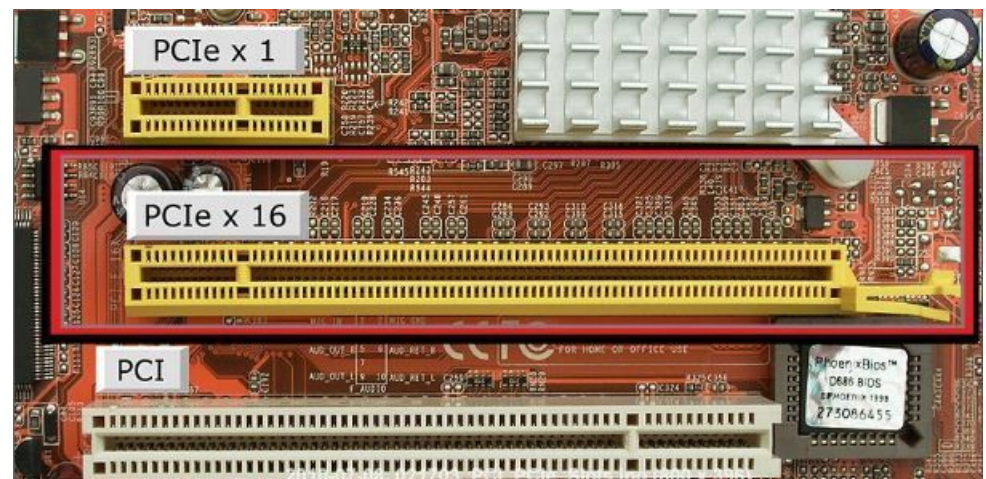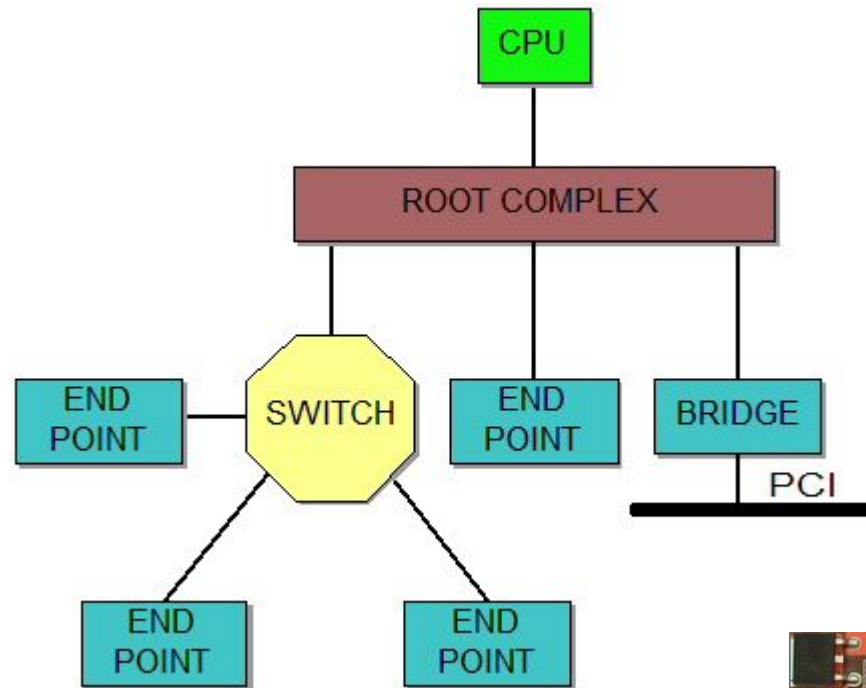
PCI Express (PCIe)

PCI devices

- Plug into PCI bus to get power and comm link

    - https://en.wikipedia.org/wiki/PCI_Express

- Well defined initial data interface

    - Data block starts with vendor & device ID

        - e.g. 0x8086 0x100f => Intel 82545EM
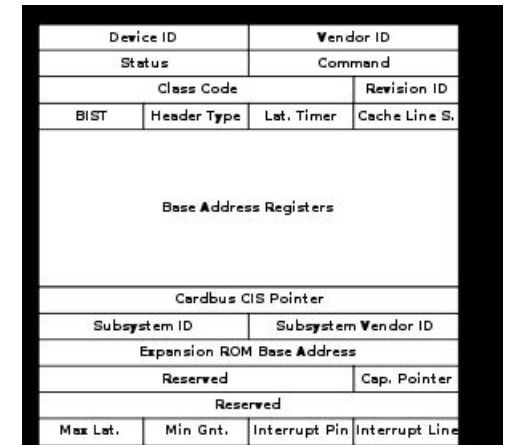
    - … and capabilities

        - MSIX, Power Management, IRQs, speed, memory

    - … and memory address for "registers"

        - https://en.wikipedia.org/wiki/PCI_configuration_space

- Internal register set maps into kernel memory

    - Data and config registers

    - Trigger (doorbell) registers

Device ID			Vendor ID	
Status			Command	
Class Code				Revision ID
BIST	Header Type	Lat. Timer		Cache Line S.
Base Address Registers				
Cardbus CIS Pointer				
Subsystem ID			Subsystem Vendor ID	
Expansion ROM Base Address				
Reserved				Cap. Pointer
Reserved				
Max Lat.	Min Gnt.		Interrupt Pin	Interrupt Line

PCI lifecycle

- The OS
  - scans PCI bus for devices
  - queries found device for config block
  - finds deviceID in PCI driver table
  - loads driver and calls `probe()`
  - calls `remove()` to unload device & driver

- The driver
  - inits device and maps it into memory
  - registers interrupt handlers
  - waits to service interrupts and OS requests
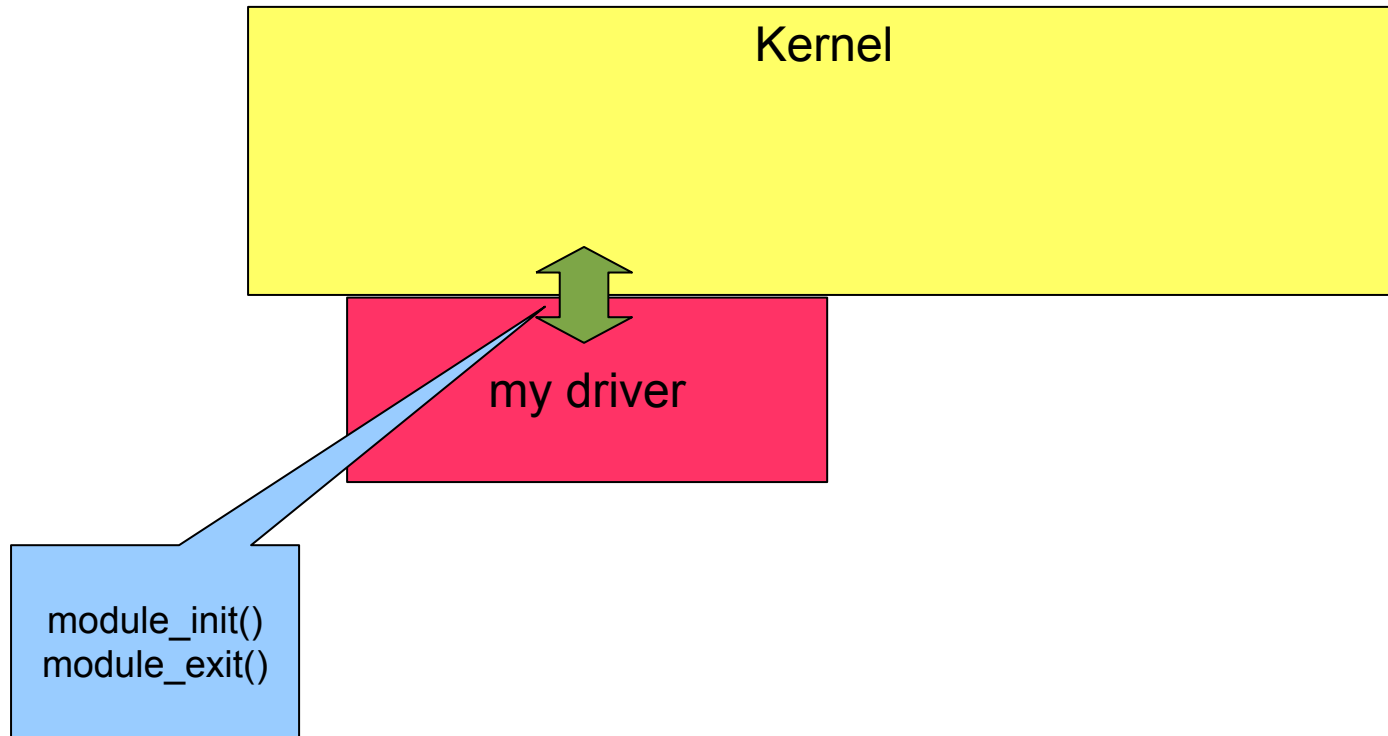 - stops device, releases memory
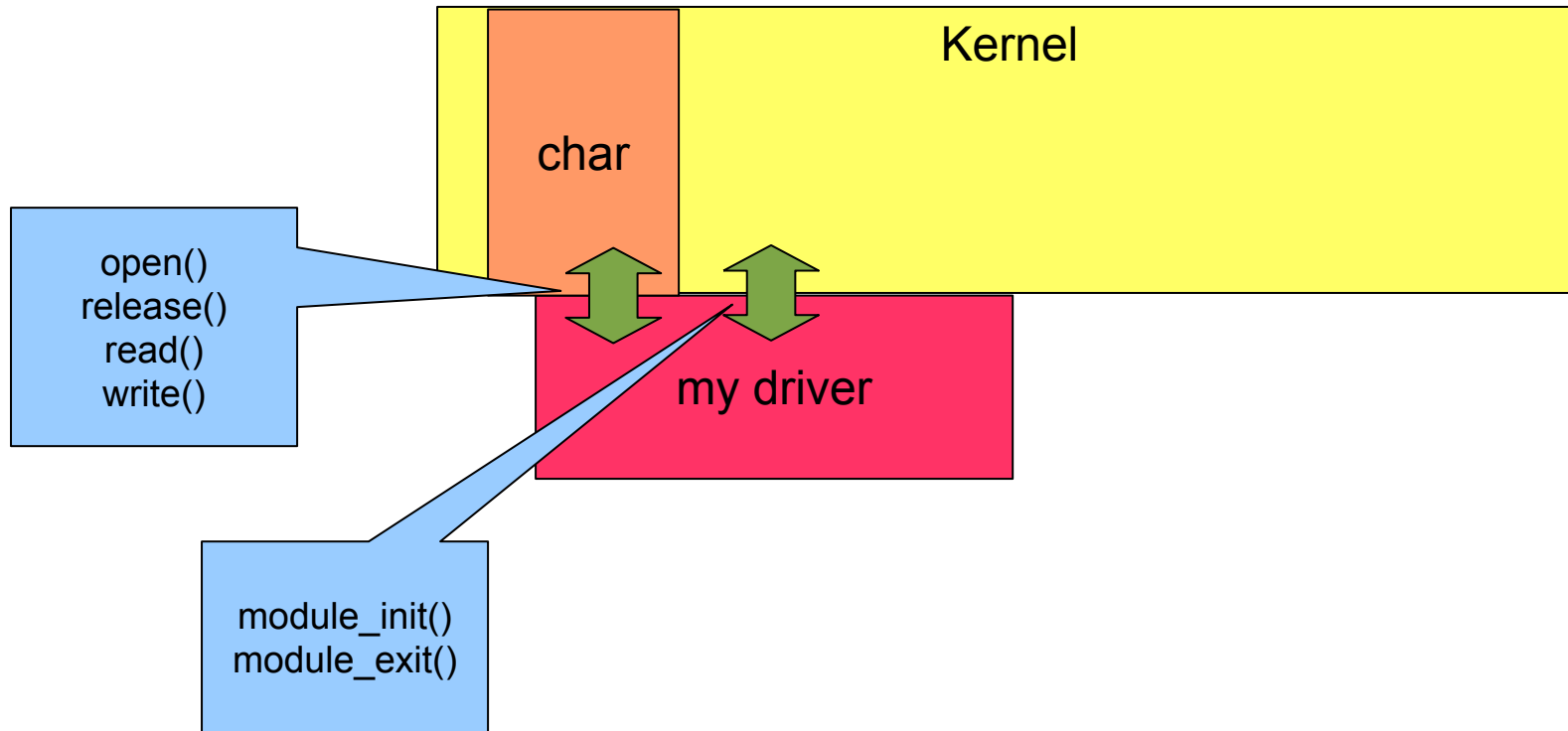
PCI access in Linux

- New structure for PCI devices:
  - `struct pci_driver`
- Needed to hook into PCI subsystem
- Init and exit the same, but have another step to device initialization: `probe()`
  - Various fields for probe: name, id_table, probe, remove, suspend, resume, etc.
- Still use same `init_module()` and `exit_module()` access

Driver callback hook-age
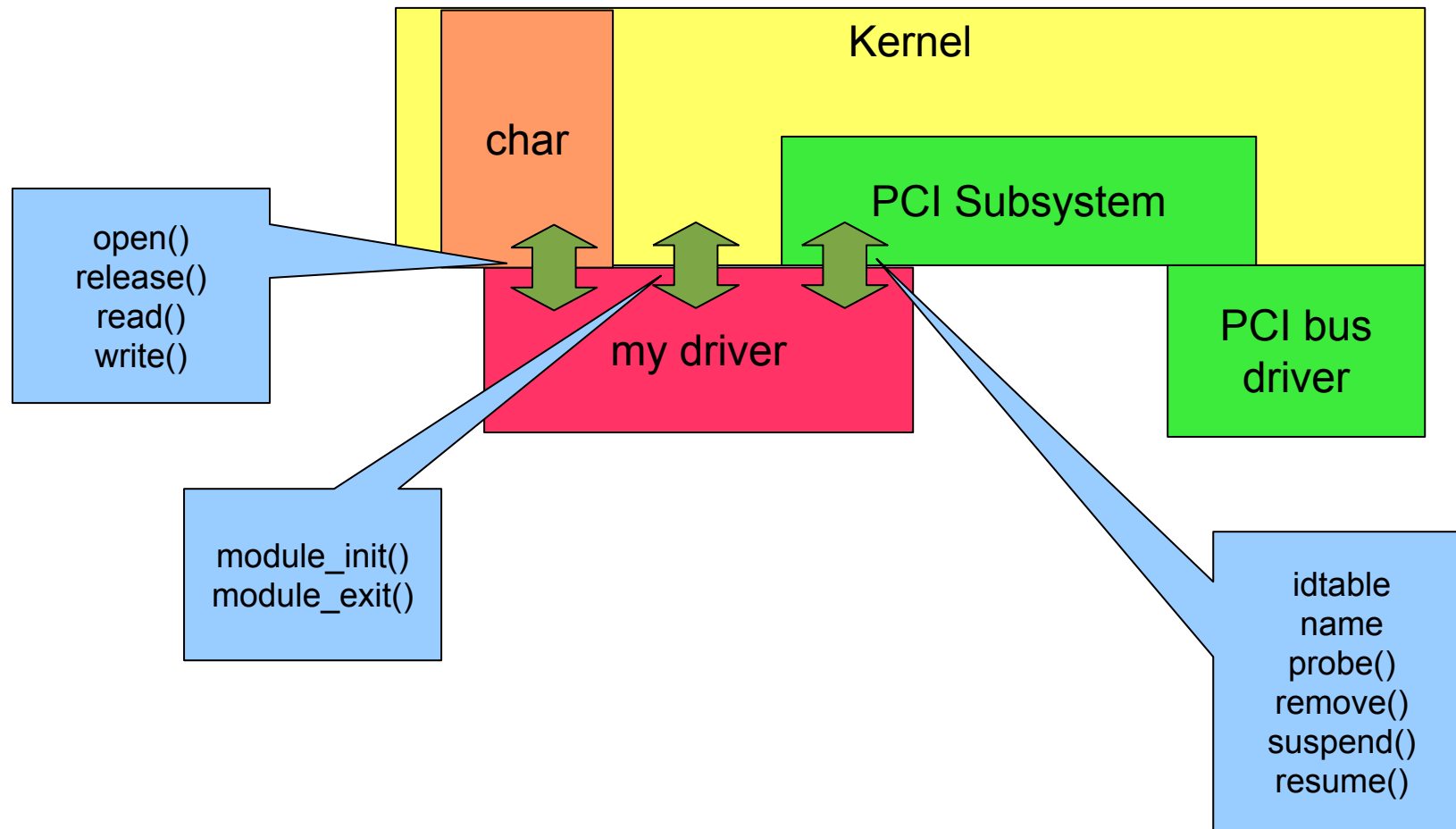
Driver callback hook-age

Driver callback hook-age

Kernel

char

PCI Subsystem

open()
release()
read()
write()

my driver

PCI bus driver

module_init()
module_exit()

idtable
name
probe()
remove()
suspend()
resume()

Example code

- Let's look at e1000e driver (PCI Express gigabit Ethernet driver)

- See how PCI driver hooks up to kernel

- https://elixir.bootlin.com/linux/latest/source/drivers/net/ethernet/intel/e1000e/netdev.c

The BAR

The BAR

- BAR is a Base Address Register

- Offset on devices to access device registers, resides in PCI "space"

- `# lspci -s <bus:device:fn> -vv`

Accessing PCI regions

- `pci_request_selected_regions()` - Maps I/O BAR into pci_device

- `ioremap()` - Returns the physical address of the requested BAR

- At this point, PCI reads and writes can be issued to device:

  - `writel()` and `readl()`, or `iowrite32()` and `ioread32()`

- We just mapped the BAR for access...

Driver to HW mappage

Memory Mapped
I/O port space

Kernel

char

PCI Subsystem

my driver

PCI bus driver

BAR 0

readl(), writel()
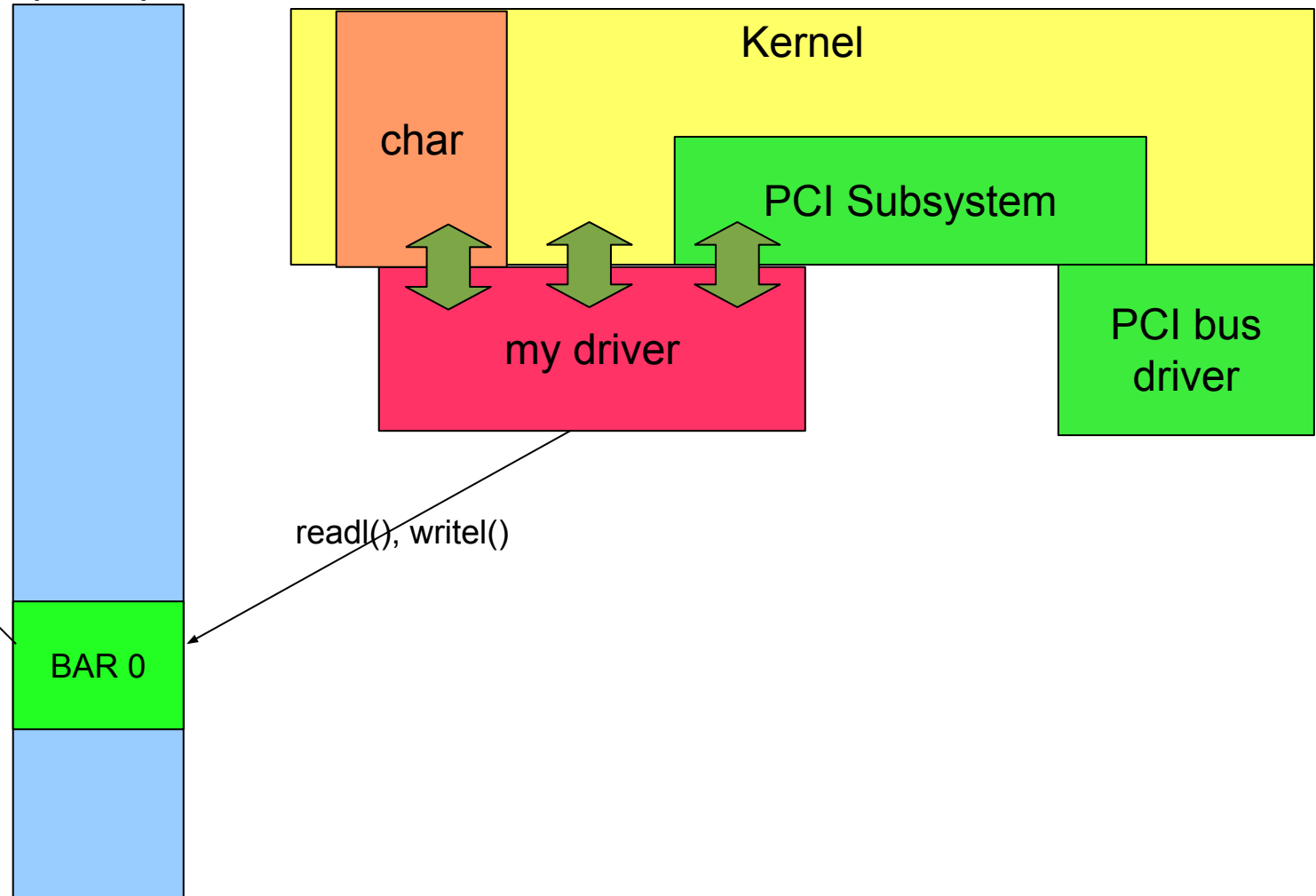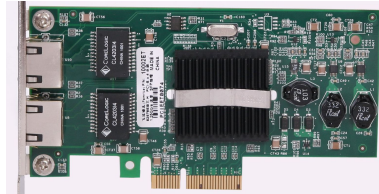
Tearing down the PCI device

- As the device exits, must unregister device from PCI layer

- `iounmap()` and `pci_release_selected_regions()`

- Must be called from the "remove" function, not "exit_module"!

- `exit_module()` must unregister the pci_device struct from driver subsystem

- Look at the source...

Taking over a claimed device

- Drivers can drive many different devices

- Sometimes they suck at what they do on certain families of hardware

    - nouveau vs. nv on certain GeForce chipsets...

- What to do?

- Bind/unbind a device from a driver:

    - `echo 0:0:19.0 > /sys/module/e1000e/drivers/pci:e1000e/bind`

    - `echo 0:0:19.0 > /sys/module/e1000e/drivers/pci:e1000e/unbind`

Datasheets

- What device addresses (registers) are there

- What values to use in the registers

- What order to read/write to get things done

- Usually available on vendor websites

- Example...

  - http://www.intel.com/content/dam/doc/datasheet/82583v-gbe-controller-datasheet.pdf

Wrap-up

- Know your memory types in Linux!

- PCI driver hookup, `pci_device` struct

- Probe entrypoint for PCI devices

- Map the BARs for access

- `writel()` and `readl()` for I/O access to device registers, or `iowrite32()` and `ioread32()`

- Unmap and unregister BARs

- Unregister PCI driver