Input/Output

Chapter 5: Section 5.1-5.3

Principles of I/O Hardware

- I/O can be roughly divided into two categories
  - Block devices - Stores information in a fixed size block each with its own address. Transfers are in units of blocks
  - Character devices - Delivers or accepts a stream of characters without regard to a block structure.

Principles of I/O Hardware

- » Classification scheme is not perfect.
  - » Clocks, not block addressable, don't generate or accept streams.
  - » Memory mapped screens
  - » Touch screens

Typical Device Data Rates

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Device Controllers

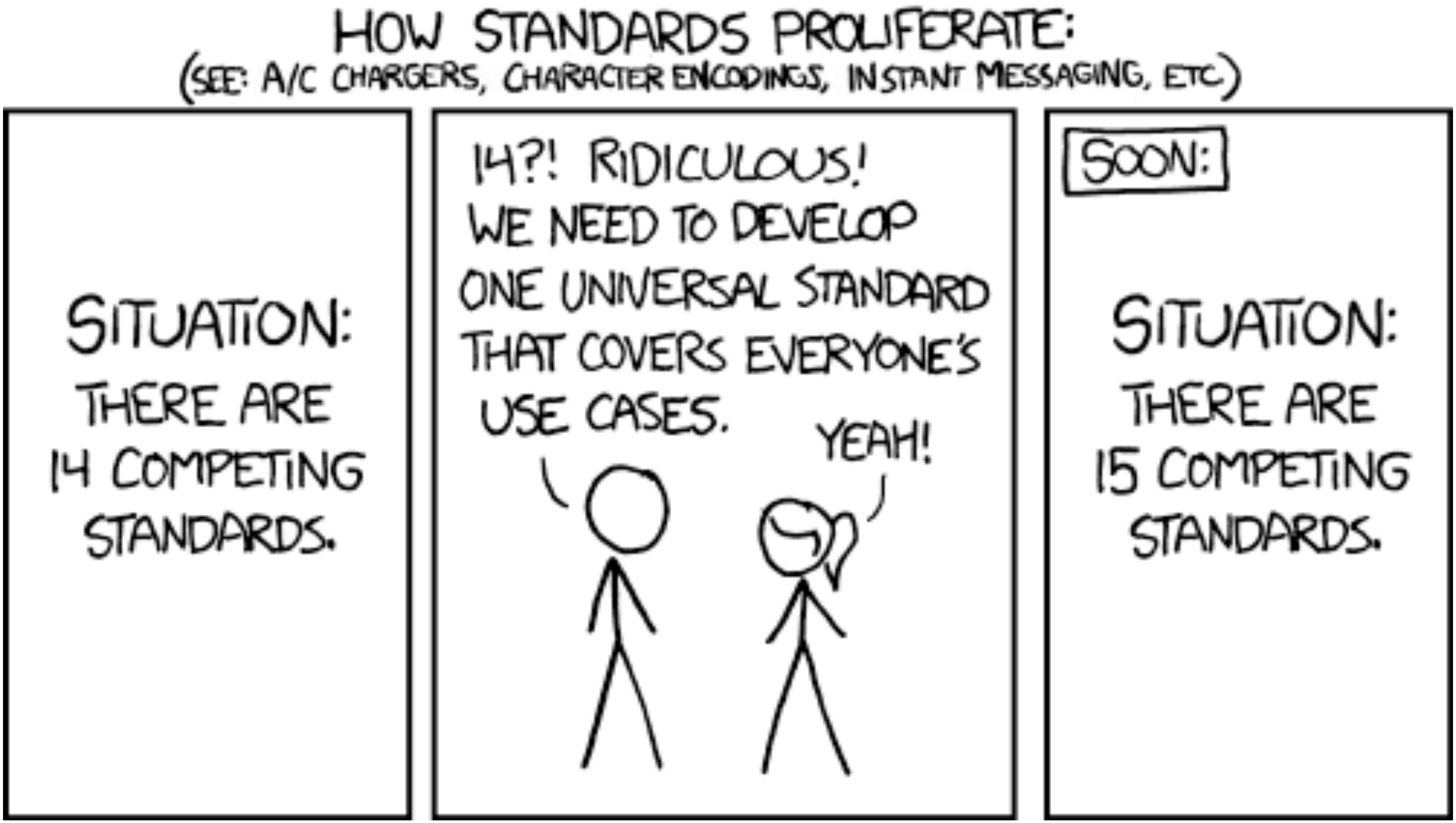
- I/O often consists of a mechanical component and an electronic component.
  - Generally split into two modular portions
  - Electric component is the device controller or device adapter.
  - Mechanical component is the device itself

Device Controllers

- If the interface between the controller and device is a standard interface companies can make controllers or devices that fit that interface.
  - ANSI, IEEE, ISO or de facto.

Standards

- de facto standard - One vendor comes up with a good idea and other vendors follow the lead. Or some organizations come together and agree on a standard.
  - QWERTY keyboard layout
  - Microsoft Word DOC
- de jure standard - Technically have the force of law behind them. Set by professional, national, or international organizations such as IEEE, ANSI and ISO.
  - PDF and HTML (started de facto eventually made de jury)
  - 802.11


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Device Controllers

- Interface is often very low level.
  - Disk formatted with 2,000,000 sectors.
  - What comes off the disk is a serial bit stream.
    - Preamble
    - 4096 bit sector
    - Checksum or Error Correcting Code

Device Controllers

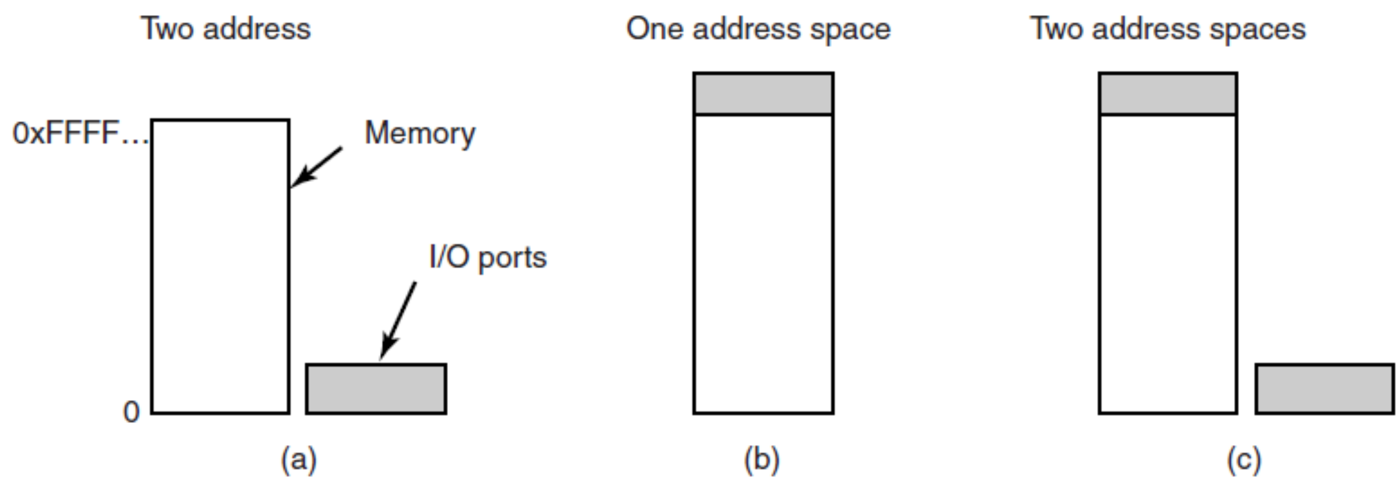
» Controller converts the serial bit stream into a block of bytes and performs any error correction.

» Once error free it can be copied to main memory.

Memory-Mapped I/O

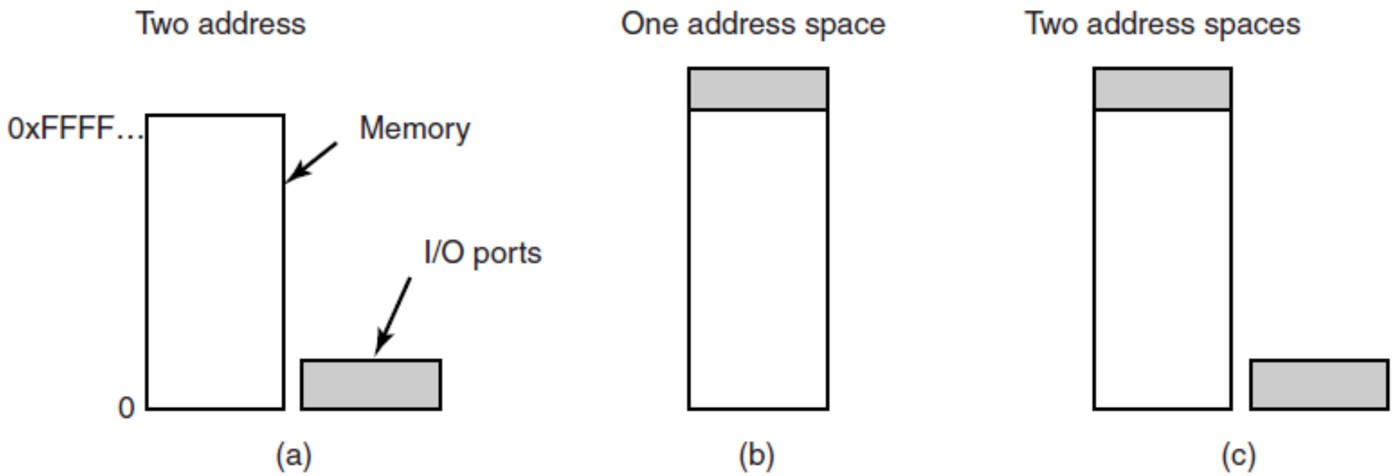
- Each controller has a couple registers used for communicating with the CPU.
  - Command the device to deliver data
  - Switch it on/off
  - Other actions
- Device also maintains data buffer.
  - Video RAM is effectively a data buffer

Device Controllers



- Two methods for CPU to communicate
  - Each control register is assigned an I/O port number (8 or 16 bit number)
  - Set of all ports is the I/O port space.
  - Ordinary user programs can not access
  - IN REG PORT or OUT PORT REG

Device Controllers

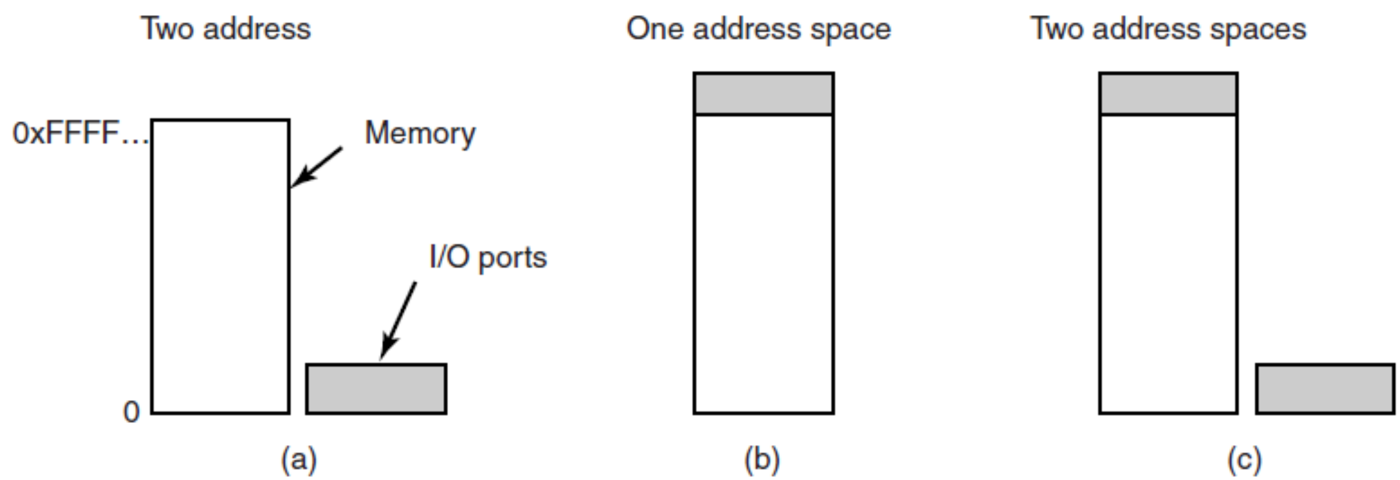


» Other method maps all control registers into memory space. (Memory Mapped I/O)

  » Each controller is assigned a unique memory address to which no memory is assigned.

  » Usually near the top of address space.

Device Controllers



» Hybrid scheme (x86) - Memory mapped I/O data buffers and separate I/O ports for control registers.

  » 640k to 1M-1 reserved for device data buffers. I/O ports 0 to 64K-1.

Device Controllers

- Different strengths and weaknesses
  - Memory mapped I/O
    - If special I/O instructions needed then need to access the control registers with assembly
    - Memory mapped allow devices drivers to be written in C.
    - No memory protection needed to keep user processes from performing I/O.
      - O/S just doesn't map I/O address range in process address space

Device Controllers

- Different strengths and weaknesses
  - Memory mapped I/O
    - Multiple devices can be given their own page of memory. Access can be given to users over some devices and not others
    - Different address spaces keep drivers from conflicting with each other.

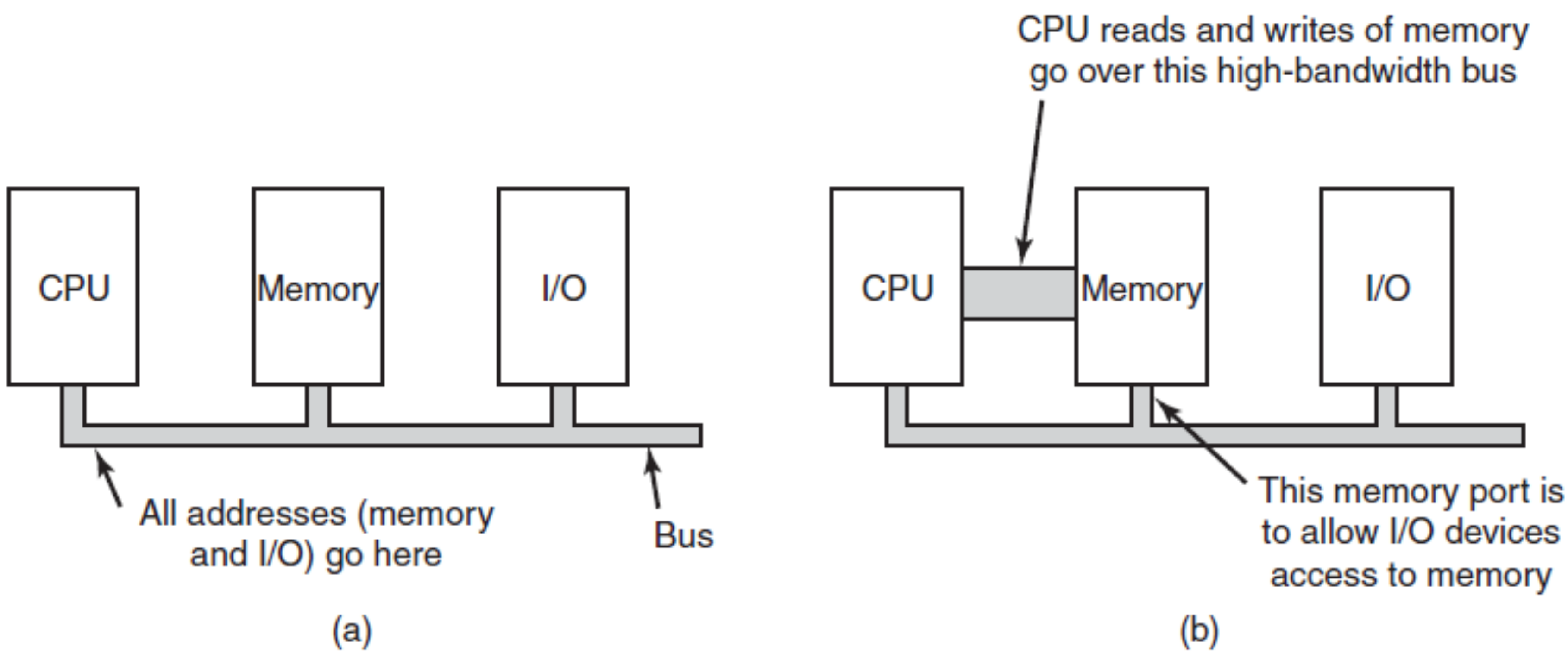
Device Controllers

- Different strengths and weaknesses
  - Memory mapped I/O weaknesses
    - Caching a control register would be disastrous.
    - Hardware has to disable caching on a per page basis.
    - If there is only one address space then all memory modules and all I/O devices must examine all memory references to see which to respond to.

Device Controllers

» Different strengths and weaknesses

   » Memory mapped I/O weaknesses

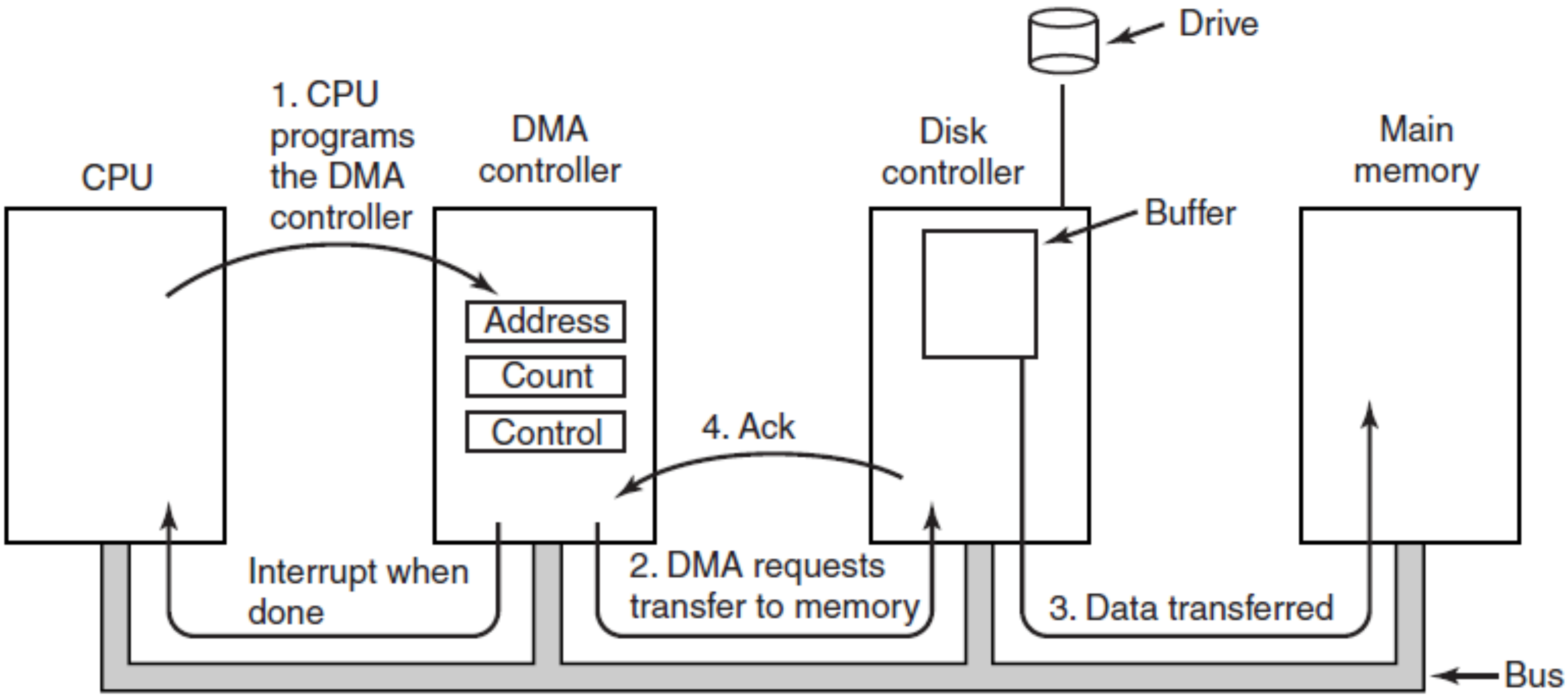


(a) A single-bus architecture.
(b) A dual-bus memory architecture.

Direct Memory Access

- CPU can request data from the controller one byte at a time, but wastes the CPUs time.
- Direct Memory Access (DMA)
  - Must have a DMA controller
  - DMA controller has access to the bus independent of the PCU.
  - CPU can read/write to the DMA controller control registers.
    - Address, Count, Control

Direct Memory Access



Direct Memory Access

- Normal disk read:
  - Disk controlled read the block form the driver bit by bit until the entire block is in the controller's internal buffer
  - Checksum computed
  - Controller causes interrupt.
  - O/S reads the disk block from the controller's buffer one byte or a word at a time.
    - Loop until done

Direct Memory Access

- DMA:
  - CPU programs the DMA controller (Step 1)
  - CPU commands the disk controller to read the dat from disk and verify the checksum
  - DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.
    - Disk controller doesn't know if it's the CPU or DMA controller issuing it.
    - Loop until done

Direct Memory Access

- DMA:
  - More sophisticated DMA can handle multiple transfers at a time.
    - Round robin the requests or priority requests
  - System busses can operate in two modes: word-at-a-time and block more.
  - Word-at-a-time mode allows the device controller to sneak in and steal an occasional bus cycle from the CPU
    - Cycle Stealing

Direct Memory Access

- DMA:
  - Block mode - the DMA controller tells the device to acquire the bus, issue the transfers then release the bus.
    - Burst mode
    - More efficient than cycle stealing
    - Blocks the CPU and other devices

Direct Memory Access

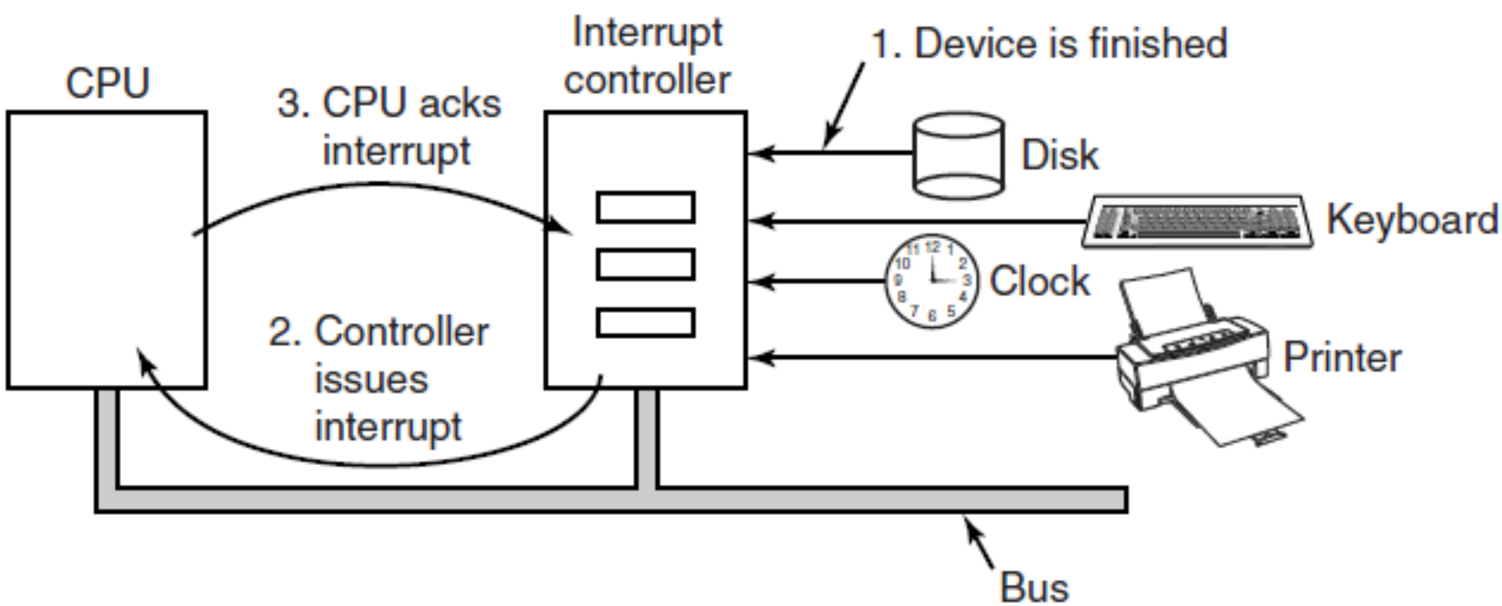
- DMA:
  - Fly-by mode - DMA tells the device controller to write directly to main memory
  - Alternate mode allows the device to send to the DMA controller which then issues a second bus request to write to a destination
    - Allows device to device DMA
- Some computers don't use DMA
  - Main CPU is often faster than DMA controller
  - Idling CPU waiting for slower DMA is pointless
  - Less hardware means cheaper cost

Interrupts Revisited

- Refresher:
  - When an I/O device is done with its work it issues an interrupt.
    - Asserts a signal on a bus line that it has been assigned.
    - Interrupt controlled chip on motherboard decides what to do
      - If no interrupt pending, the interrupt controller handles the interrupt immediately.
      - If other interrupts in progress it is ignored until the interrupt controller is free.
    - Interrupt handler puts a number on the address line specifying which device needs attention and asserts a signal to interrupt the CPU.

Interrupts Revisited

- Refresher:
  - The number on the address line is used as an index into the interrupt vector table that allows the CPU to fetch a new program counter.



Interrupts Revisited

- When executing the interrupt, what does the CPU save?
  - Program counter is bare minimum
  - All visible registers and most internal registers at the other end
- Where do you put the data?
  - Internal registers, but then can't acknowledge interrupt handled until all registers read back out
    - Takes time, leaves dead space
  - Stack
    - Can't be user stack. Stack pointer may not be legal
      - Might be on the end of a page. If you page fault where do you save the state to handle the fault?

Interrupts Revisited

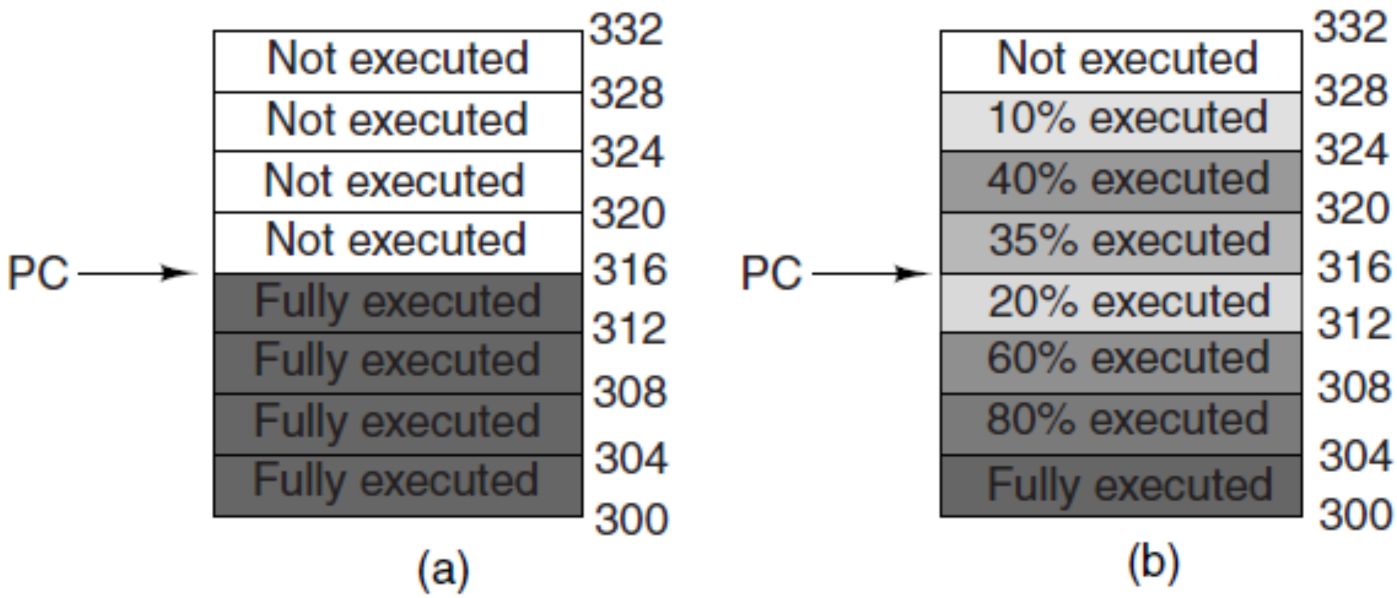
- Kernel stack?
  - May require switching into kernel mode
    - Change MMU context
      - Invalidate the cache and TLB

Interrupts Revisited

- Kernel stack?
  - May require switching into kernel mode
    - Change MMU context
      - Invalidate the cache and TLB

- Modern CPUs are heavily pipelined, often superscalar.
  - Can't assume if an interrupt occurs after an instruction that all instructions leading up to and including that instruction have been executed completely.
    - Many partially executed instructions.

Interrupts Revisited



(a) A precise interrupt. (b) An imprecise interrupt.

Interrupts Revisited

- Precise Interrupt - Interrupt that leaves the machine in a well-defined state.
- Four properties of a precise interrupt:
  1. The PC saved in a known place.
  2. All instructions before that pointed to by PC have fully executed.
  3. No instruction beyond that pointed to by PC has been executed.
  4. Execution state of instruction pointed to by PC is known.

Interrupts Revisited

- Imprecise Interrupt - Interrupt that does not meet the four requirements
  - Unpleasant for OS developer.
  - Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on.
    - Large amount of work to restart
    - Can make very fast superscalar CPUs unsuitable for real-time work due to slow interrupts
- x86 has very complex logic in the CPU to have precise interrupts

Principle of I/O software

- Key concepts:
  - Device independence - Write a program that can access any I/O without having to specify the device in advance
    - open should work on any device: drive, flash, cd
  - Uniform Naming - should not depend on device in any way.

Principle of I/O software

- Key concepts:
  - Error handling: errors should be handled as close to the hardware as possible.
    - If the controller discovers a read error it should correct it .
  - Synchronous / Asynchronous
    - Most physical I/O is asynchronous
    - Blocking programs easier to write

Principle of I/O software

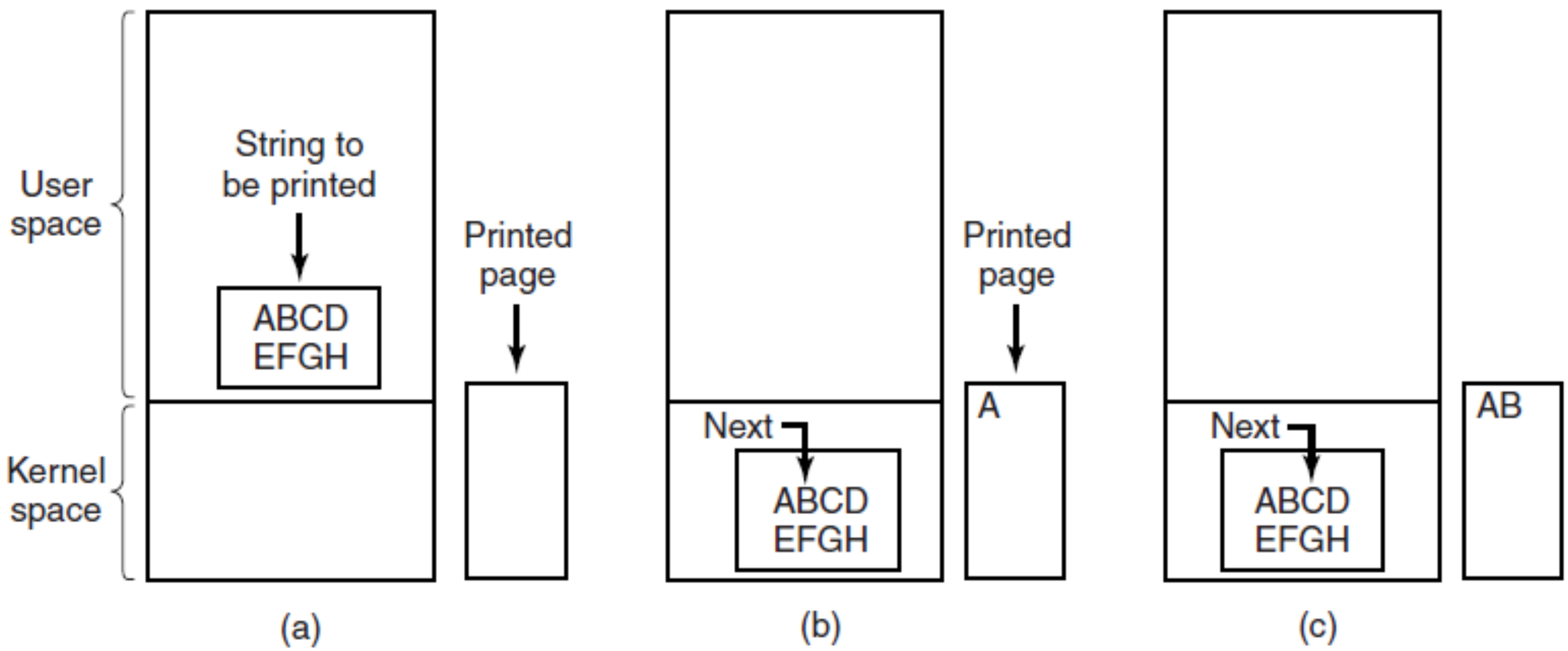
- Key concepts:
  - Buffering: Usually can't store data directly off the device into its final destination
    - Sometime must decouple rate at which the data is filled from the rate at which it is emptied.
  - Shareable / Dedicated devices
    - Drives can have multiple users sharing
    - Printers can only have one user at a time

Programmed I/O

- Three fundamentally different ways I/O can be performed.
  - Programmed I/O
  - Interrupt Driven I/O
  - I/O using DMA

- Simplest is programmed I/O, also known as letting the CPU do all the work.

Programmed I/O



Steps in printing a string.

Programmed I/O

- Essential aspect of programmed I/O:
  - The CPU continuously polls the device to see if it is ready to accept more data
    - Busy waiting / Polling

```
copy_from_user(buffer, p, count);              /* p is the kernel buffer */
for (i = 0; i < count; i++) {                  /* loop on every character */
        while (*printer_status_reg != READY) ;  /* loop until ready */
        *printer_data_register = p[i];          /* output one character */
}
return_to_user( );
```

Programmed I/O

- Essential aspect of programmed I/O:
  - Busy waiting is inefficient.
  - Ok for small embedded devices where the CPU has nothing else to do, but larger systems need a better solution

Programmed I/O

» Interrupt-driven I/O

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
      unblock_user( );
} else {
      *printer_data_register = p[i];
      count = count – 1;
       i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

Programmed I/O

- Problem with interrupt driven I/O:
  - Interrupt every character
- Use DMA for I/O

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

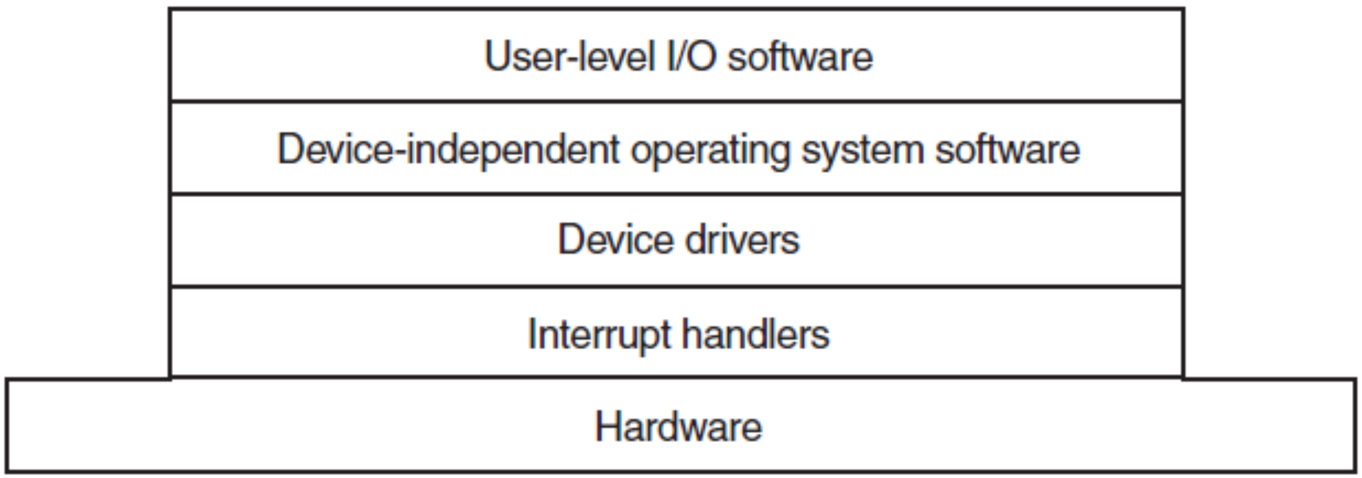
Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

Programmed I/O

- Big win for DMA is reducing the number of interrupts from one per character to one per buffer.
  - DMA controller is slower than CPU.
  - If DMA can't drive the device at full speed then CPU driven I/O or interrupt driven I/O may be better.

Programmed I/O

» I/O software typically organized in four layers.



Device Drivers

- Device registers and nature of commands can vary dramatically form device to device.
- Each I/O device attached to a computer needs some device specific code for controlling it, a device driver.
  - Usually written by the device manufacturer
- While wildly different many devices based on the same underlying technology
  - USB

Device Drivers

» USB devices are stacked.

1. Link Layer (Serial IO) - deals with differential line transitions and signaling, and decode the stream to binary data, very often in hardware
2. USB Packet Layer - deals with structure of USB data packets
3. USB Required Functionality - enumeration, buffers, endpoints
4. USB higher level APIs - Audio, HID, etc, that have their own restrictions and needs.

Device Drivers

- Generally device drivers must be part of the kernel
- Can write user space device drivers, e.g. MINIX
  - Most run in kernel space
- Need a well defined mode of what a driver does an how it interacts with the system so third party developers can write drivers.
  - Block devices
  - Character devices
- Standard interface all block drivers must support and a second standard interface for all character devices.

Device Drivers

- For many years the norm was to compile the drivers into the kernel in one single binary program.
- Advent of personal computers and myriad of devices changed.
  - Home users don't want to recompile their kernel.

Linux Device Drivers

```
# ls -l /dev/hda[1-3]
brw-rw----   1 root   disk   3, 1 Jul   5   2000 /dev/hda1
brw-rw----   1 root   disk   3, 2 Jul   5   2000 /dev/hda2
brw-rw----   1 root   disk   3, 3 Jul   5   2000 /dev/hda3
```

- Major and Minor Numbers
  - Major number tells you which driver is used to access the hardware.
    - Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver.
  - The minor number is used by the driver to distinguish between the various hardware it controls.

Linux Device Drivers

- Adding a driver to your system means registering it with the kernel.
  - Assigning it a major number during the module's initialization. You do this by using the `register_chrdev` function, defined by `linux/fs.h`.

Linux Device Drivers

- How do you get a major number without hijacking one that's already in use?
  - Look through Documentation/devices.txt and pick an unused one.
  - Ask the kernel to assign you a dynamic major number.

Linux Device Drivers

- How do you get a major number without hijacking one that's already in use?
  - Look through Documentation/devices.txt and pick an unused one.
  - Ask the kernel to assign you a dynamic major number.

Linux Device Drivers

- Each device is represented in the kernel by a file structure, which is defined in linux/fs.h.
  - The “file” is a kernel level structure and never appears in a user space program.
  - It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function.
  - Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode.
  - VFS

Linux Device Drivers

```
struct file_operations {
  struct module *owner;
   loff_t(*llseek) (struct file *, loff_t, int);
   ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
   ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
   ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
                        loff_t);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int,
                unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
   ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                    loff_t *);
   ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                     loff_t *);
   ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                       void __user *);
   ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                       loff_t *, int);
  unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                      unsigned long, unsigned long,
                                      unsigned long);
};
```

Linux Device Drivers

```
struct file_operations fops = {
        read: device_read,
        write: device_write,
        open: device_open,
        release: device_release
};
```

» Populate the file_operations struct with the functions your device will support.

Linux Device Drivers

```
/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

        if (Major < 0) {
          printk(KERN_ALERT "Registering char device failed with %d\n", Major);
          return Major;
        }

        printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
        printk(KERN_INFO "the driver, create a dev file with\n");
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
        printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
        printk(KERN_INFO "the device file.\n");
        printk(KERN_INFO "Remove the device file and module when done.\n");

        return SUCCESS;
}
```

» Create init_module function to register the device.

Linux Device Drivers

```
/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;

        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        sprintf(msg, "I already told you %d times Hello world!\n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}
```

» For each function you've registered in the file_operations structure define your function.

Linux Device Drivers

```
/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;

        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        sprintf(msg, "I already told you %d times Hello world!\n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}
```

» For each function you've registered in the file_operations structure define your function.

Linux Device Drivers

Copy from
kernel to user

```
/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

        /*
         * If we're at the end of the message,
         * return 0 signifying end of file
         */
        if (*msg_Ptr == 0)
                return 0;

        /*
         * Actually put the data into the buffer
         */
        while (length && *msg_Ptr) {

                /*
                 * The buffer is in the user data segment, not the kernel
                 * segment so "*" assignment won't work.  We have to use
                 * put_user which copies data from the kernel data segment to
                 * the user data segment.
                 */
                put_user(*(msg_Ptr++), buffer++);

                length--;
                bytes_read++;
        }

        /*
         * Most read functions return the number of bytes put into the buffer
         */
        return bytes_read;
}
```