I/O Management Intro

Chapter 5

THE UNIVERSITY OF
NEW SOUTH WALES

Learning Outcomes

- A high-level understanding of the properties of a variety of I/O devices.
- An understanding of methods of interacting with I/O devices.

I/O Devices

- There exists a large variety of I/O devices:
  - Many of them with different properties
  - They seem to require different interfaces to manipulate and manage them
    - We don't want a new interface for every device
    - Diverse, but similar interfaces leads to code duplication
- Challenge:
  - Uniform and efficient approach to I/O

Categories of I/O Devices (by usage)

- Human interface
  - Used to communicate with the user
    - Limited by human speed or perception
  - Printers, Video Display, Keyboard, Mouse
- Machine interface
  - Used to communicate with electronic equipment
    - Latency sensitive
  - Disk and tape drives, Sensors, Controllers, Actuators
- Communication
  - Used to communicate with remote devices
    - Latency or throughput sensitive
  - Ethernet, Modems, Wireless

I/O Device Handling

- Data rate
  - May be differences of several orders of magnitude between the data transfer rates
  - Example: Assume 1000 cycles/byte I/O
    - Keyboard needs 10 KHz processor to keep up
    - Gigabit Ethernet needs 100 GHz processor…..

Sample Data Rates

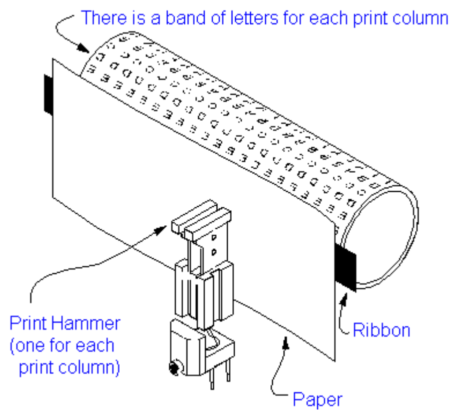
Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

USB 3.0 625 MB/s (5 Gb/s)
Thunderbolt 2.5GB/sec (20 Gb/s)
PCIe v3.0 x16 16GB/s

I/O Device Handling Considerations

- Complexity of control
- Unit of transfer
  - Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk
- Data representation
  - Encoding schemes
- Error conditions
  - Devices respond to errors differently
    - `lp0: printer on fire!`
  - Expected error rate also differs

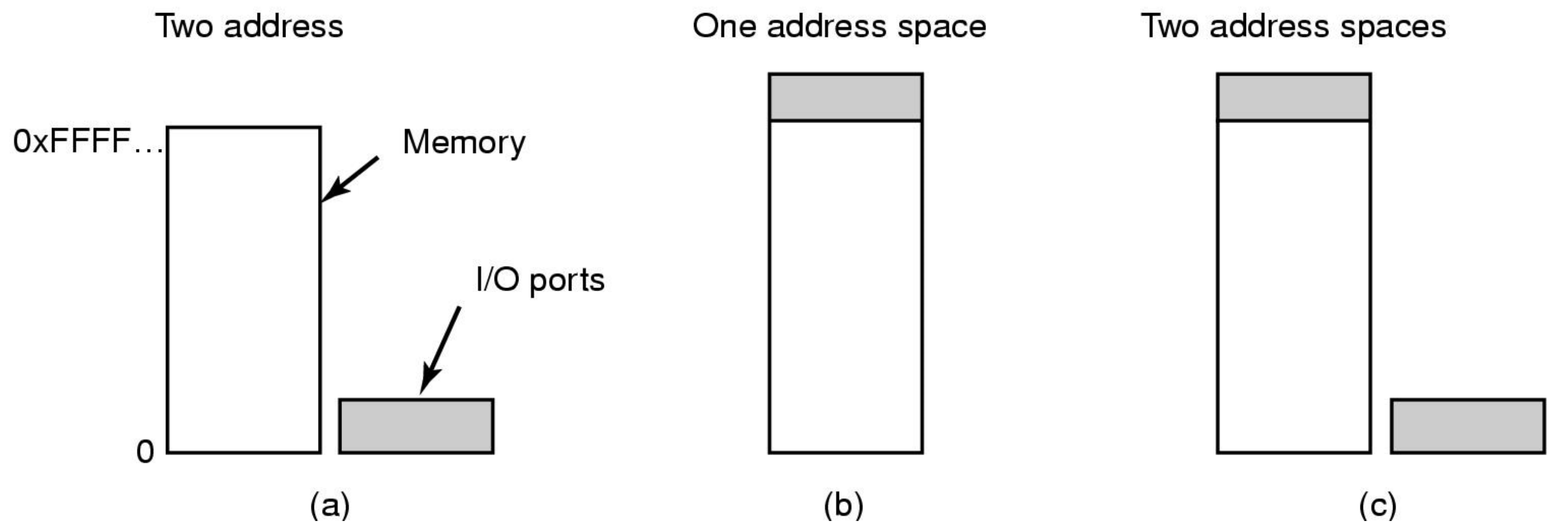
From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.


There is a band of letters for each print column
Print Hammer (one for each print column)
Ribbon
Paper

I/O Device Handling Considerations

- Layering
  - Need to be both general and specific, e.g.
  - Devices that are the same, but aren't the same
    - Hard-disk, USB disk, RAM disk
  - Interaction of layers
    - Swap partition and data on same disk
    - Two mice
  - Priority
    - Keyboard, disk, network

Accessing I/O Controllers



a) **Separate I/O and memory space**
   - I/O controller registers appear as I/O ports
   - Accessed with special I/O instructions

b) **Memory-mapped I/O**
   - Controller registers appear as memory
   - Use normal load/store instructions to access

c) **Hybrid**
   - x86 has both ports and memory mapped I/O

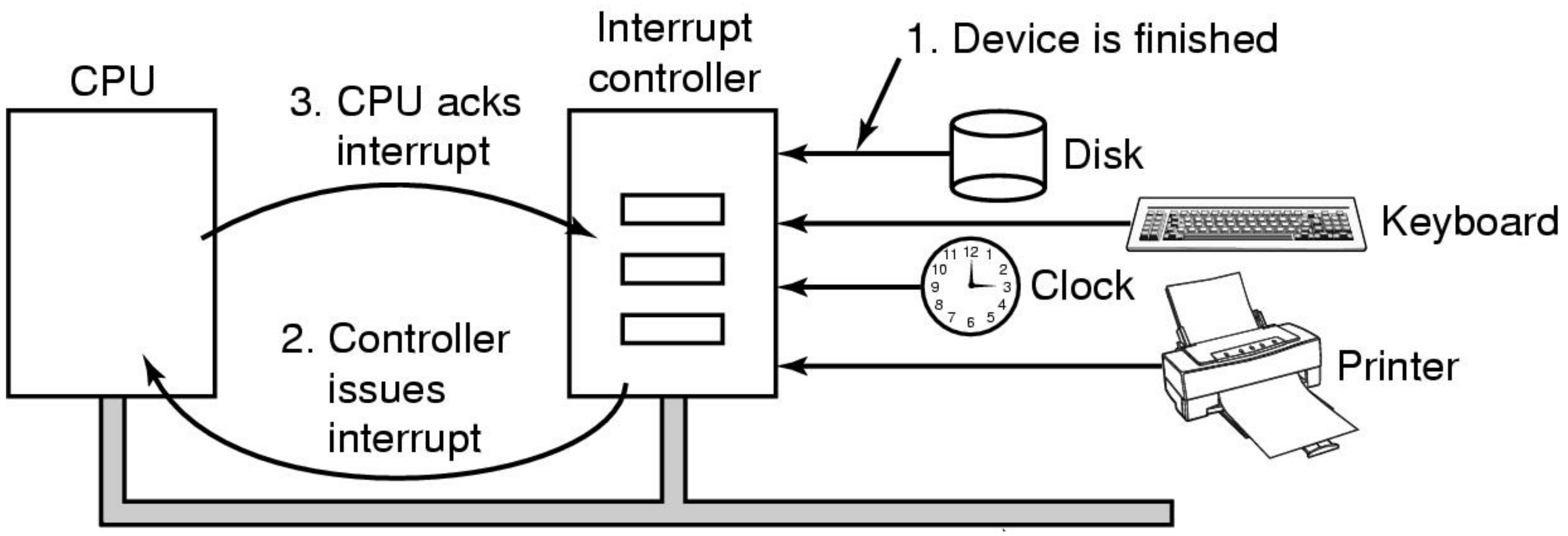
Port Access Example (x86)

```
static inline void outb(uint16_t port, uint8_t val)
{
    asm volatile ( "outb %0, %1" : : "a"(val), "Nd"(port) );
}
static inline uint8_t inb(uint16_t port)
{
    uint8_t ret;
    asm volatile ( "inb %1, %0" : "=a"(ret): "Nd"(port) );
    return ret;
}

uint16_t port = 0x378; /* assign port number */
input = inb(port);     /* read 8-bits from port */
outb(port, output);    /* write 8-bits to port */
```

Memory Mapped I/O

```
#define DEV_ADDR 0x12345678      /* address in memory */
volatile uint32_t *dev_reg;      /* type of register – 32-bit */

dev_reg = (uint32_t *) DEV_ADDR; /* init pointer to device reg */

input = *dev_reg;         /* read device register */
*dev_reg = output;        /* write device register */

while (*dev_reg == 0) {} /* spin waiting, need “volatile” */
```

Interrupts

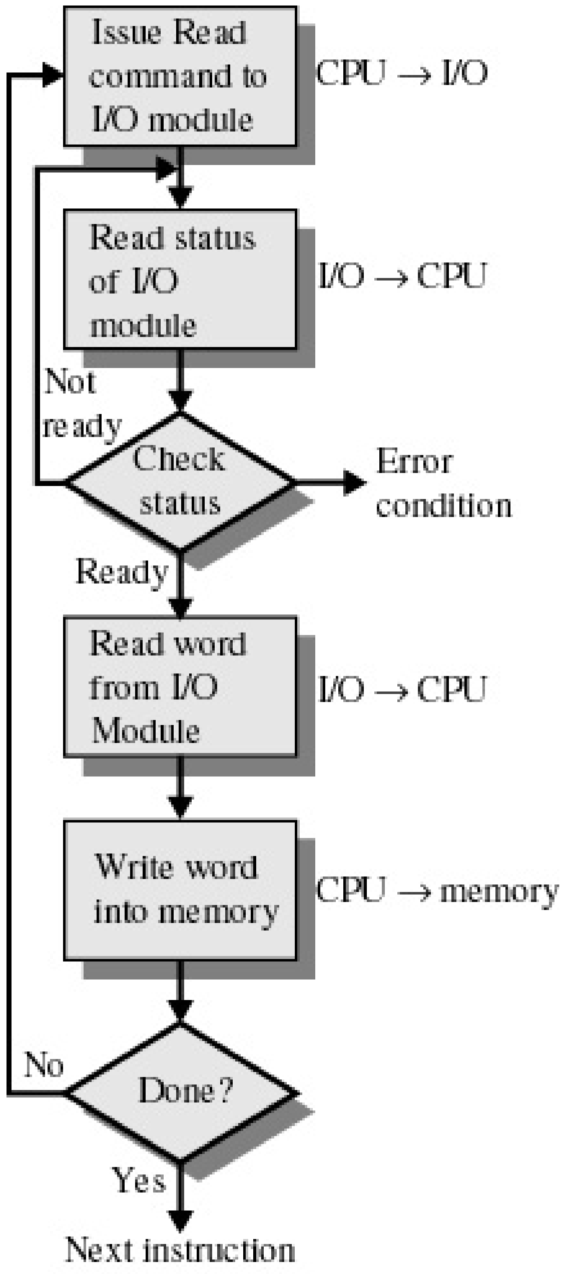


- Devices connected to an *Interrupt Controller* via lines on an I/O bus (e.g. PCI)
- Interrupt Controller signals interrupt to CPU and is eventually acknowledged.
- Exact details are architecture specific.

I/O Interaction

Programmed I/O

- Also called *polling*, or *busy waiting*
- I/O module (controller) performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
  - Wastes CPU cycles



(a) Programmed I/O

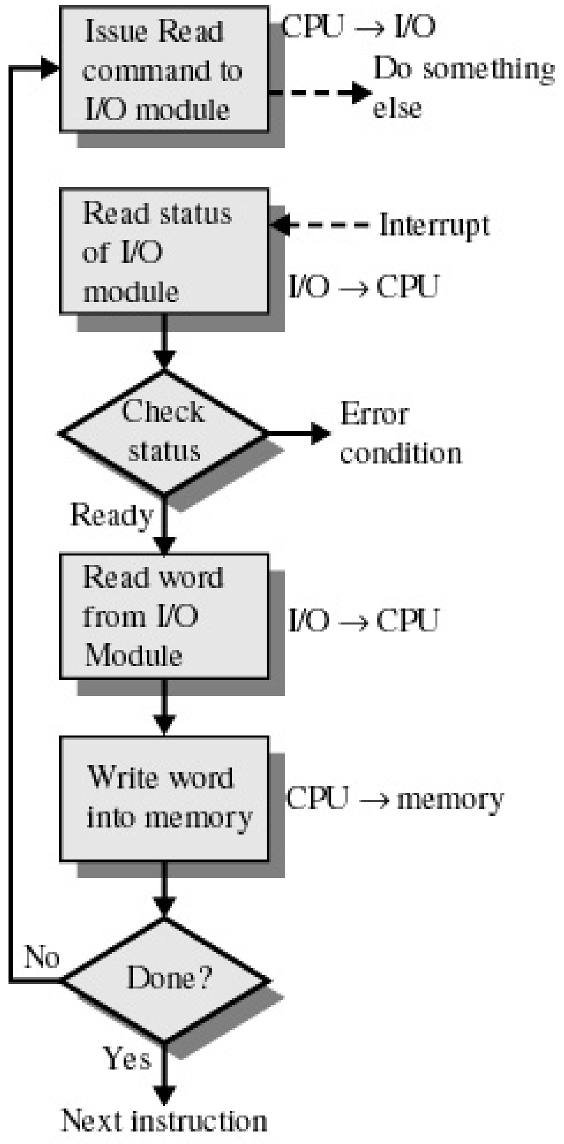
Sample Programmed I/O

```
volatile struct {
  uint8_t wstatus;
  uint8_t rstatus;
  char outreg;
  char inreg;
} * dev = 0x12345678;
```

```
void write_buf(char *buf, int n) {
  int i = 0;
  while (i < n) {
    while (dev->wstatus != READY) {}; /* busy wait */
    dev->outreg = buf[i];
    i++;
  }
}

void read_char(char *buf) {
  while (dev->rstatus != READY) {}; /* busy wait */
  *buf = dev->inreg;
}
```

Interrupt-Driven I/O

- Processor is interrupted when I/O module (controller) ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor



(b) Interrupt-driven I/O

THE UNIVERSITY OF NEW SOUTH WALES

Sample Interrupt Driven I/O

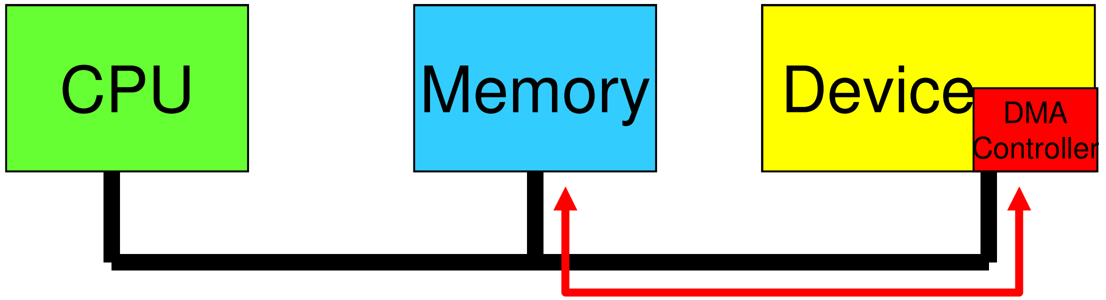
```
int cur_count;
int out_count;
char * outbuf;
semaphore_t *io_wait;
```

```
void write_buf(char *buf, int n) {
  cur_count = 0;
  out_count = n;
  outbuf = buf;

  dev->outreg = outbuf[cur_count];
  cur_count++;
  dev->wstatus = START;
  P(io_wait)
}

interrupt_handler ()
{
  if (cur_count < out_count) {
    dev->outreg = outbuf[cur_count];
    cur_count++;
    dev->wstatus = START;
  }
  else {
    V(io_wait);
  }
```

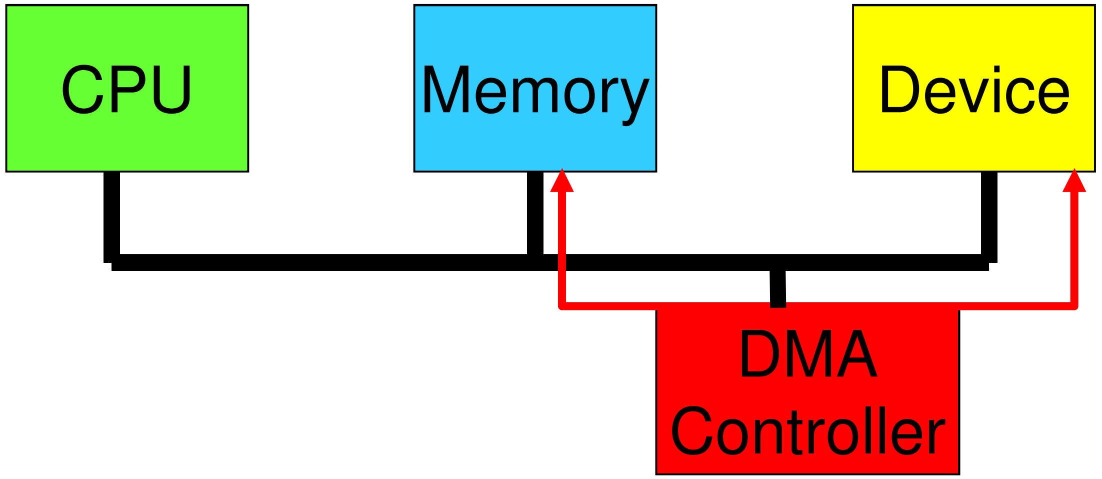
Direct Memory Access

- Transfers data directly between Memory and Device
- CPU not needed for copying

DMA Controller in Device

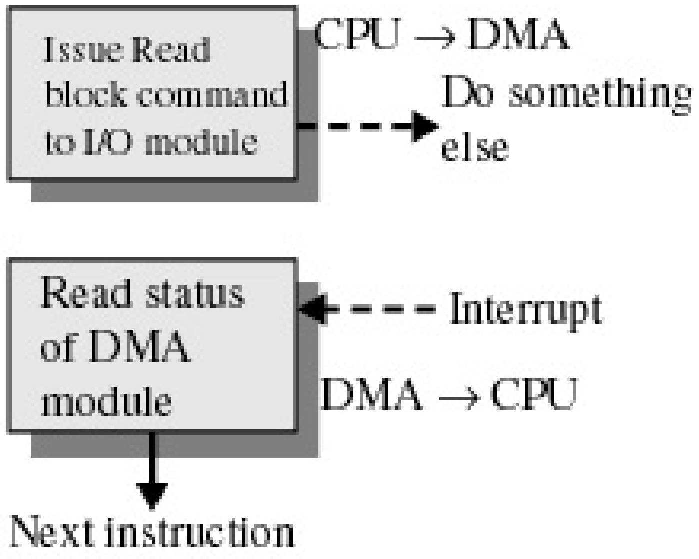


Separate DMA Controller



Direct Memory Access

- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer


Issue Read block command to I/O module
CPU → DMA
Do something else
Read status of DMA module
Interrupt
DMA → CPU
Next instruction

(c) Direct memory access

Sample Interrupt Driven DMA I/O

```
volatile struct {
  uint8_t wstatus;
  uint8_t rstatus;
  char outreg;
  char inreg;
  uint32_t dma_addr;
  uint32_t dma_size;
} * dev = 0x12345678;
semaphore_t *io_wait;
```

```
void write_buf(char *buf, int n) {

  dev->dma_addr = buf;
  dev->dma_size = n;
  dev->wstatus = START_DMA;
  P(io_wait)
}
interrupt_handler ()
{
  V(io_wait);
}
```

THE UNIVERSITY OF
NEW SOUTH WALES

DMA Considerations

- ✓ Reduces number of interrupts
  - Less (expensive) context switches or kernel entry-exits
- ✗ Requires contiguous regions (buffers)
  - Copying if not contiguous
  - Some hardware supports “Scatter-gather”
    - E.g. network controller add headers to start of data
- Synchronous
  - I/O occurs when we request it
  - Can allocate/create buffer in advance (or wait for memory)
- Asynchronous
  - I/O occurs without our direct request (e.g. incoming network packets)
  - Need to allocate free buffers in advance
    - how many?
    - too few, data is “dropped”; too many, impact application performance.