

INPUT AND OUTPUT

Richard Hayden (with thanks to Narancker Dulay)

rh@doc.ic.ac.uk

<http://www.doc.ic.ac.uk/~rh/teaching>

or

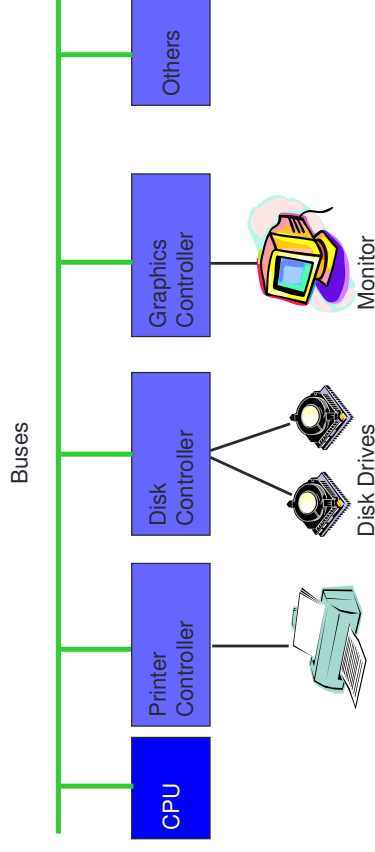
<https://www.doc.ic.ac.uk/~wl/teachlocal/arch1>

or

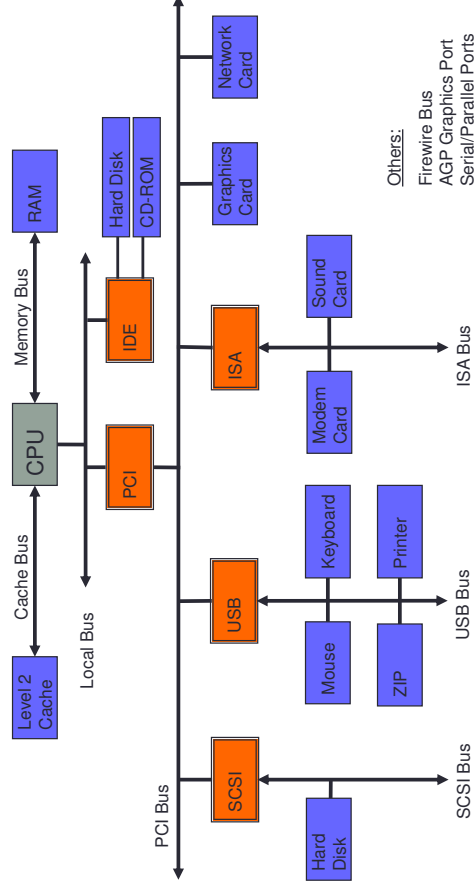
CATE

I/O controllers

- Otherwise known as I/O adapters or I/O modules. Provide the CPU with a programming interface

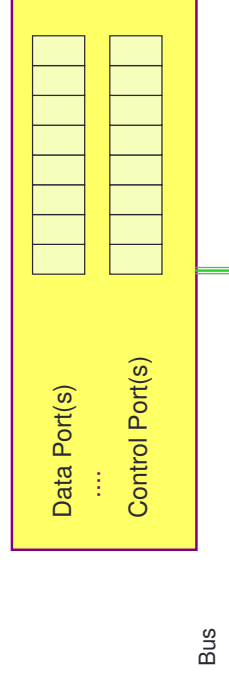


I/O structure



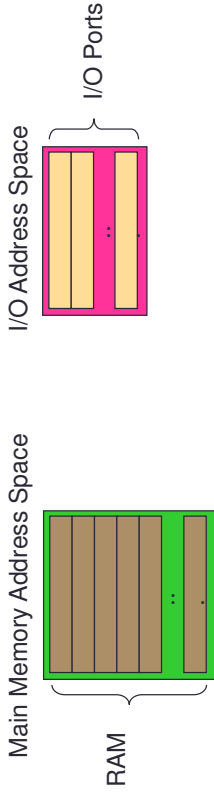
I/O ports

A Simple I/O Controller



- **Data ports** are used for passing data to/from the CPU and the I/O device
- **Control ports** are used to issue **I/O commands** (e.g. write char) and to **check device status**. Write to **specific bits** of control port to issue commands and read other bits in order to check status.

I/O addressing 1: separate address space

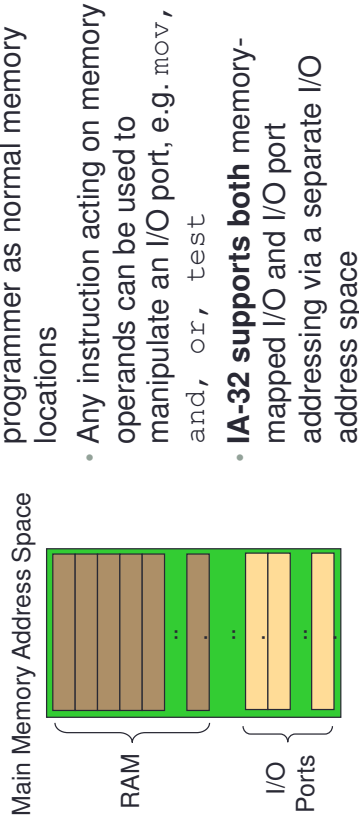


- I/O ports **have their own** (very small) address space. Architecture provides some method for accessing it, e.g. special I/O instructions:
 - in `ax, 20` ; copy 16 bits from I/O port 20 into ax
 - out `35, al` ; copy 8 bits from al to I/O port 35
- Control bus used to signal if a transfer is for I/O or main memory address space
- IA-32 provides **64K 8-bit I/O ports** numbered 0 to 65535

Four I/O schemes

- **Programmed I/O**: continually **poll a device's control port** until it's ready and then initiate transfer
- **Interrupt-driven I/O**: initiate transfer and then do something else. Device will "interrupt" the CPU when transfer is complete
- **DMA I/O**: initiate large data (block) transfer. Device will transfer block to/from memory and then interrupt the CPU after block is transferred
- **I/O processor**: delegate complex I/O processing tasks to a dedicated processor

I/O addressing 2: memory mapped I/O



- I/O ports appear to the programmer as normal memory locations
- Any instruction acting on memory operands can be used to manipulate an I/O port, e.g. `mov, and, or, test`
- **IA-32 supports both memory-mapped I/O and I/O port** addressing via a separate I/O address space

Programmed I/O

- Check control port before reading or writing to data port
- **Example**: writing a block of data to an I/O device

```

foreach byte in Block do
loop   Read Status Bit(s) of CONTROL Port
       if Status Bit(s) indicate ERROR then "Handle" Error
       exit when Status Bit(s) indicate DEVICE is "READY"
endloop
Copy byte from Block to DATA Port
Issue Write Request by writing to Command bits in CONTROL Port
endfor
    
```

- **Note**: for some output (input) devices the mere act of writing (reading) to (from) the data port initiates a new output (input) operation

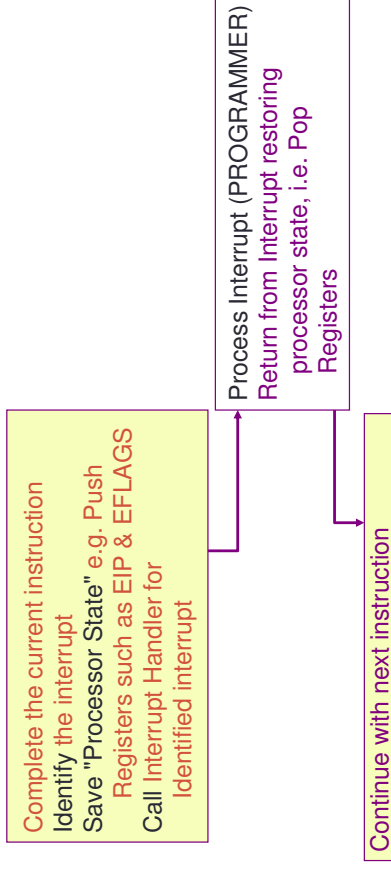
Programmed I/O trade-offs

- Simple to program
- Can guarantee response times
- Poor CPU utilisation (waste CPU by busy waiting)
- Multiple devices are awkward to handle

Interrupt processing

CPU

Interrupt Handler



Interrupt-driven I/O

- Initiate transfer and then do something else. **Device will interrupt CPU when transfer is complete.** On detecting an interrupt, control is transferred to device's interrupt-handling procedure (**interrupt handler**)
- CPU's fetch-execute cycle now becomes:

```
loop  
Fetch Instruction  
Decode Instruction  
Execute Instruction  
if INTERRUPT pending then  
    "CALL" Interrupt Handler for this particular Interrupt  
endif  
endloop
```

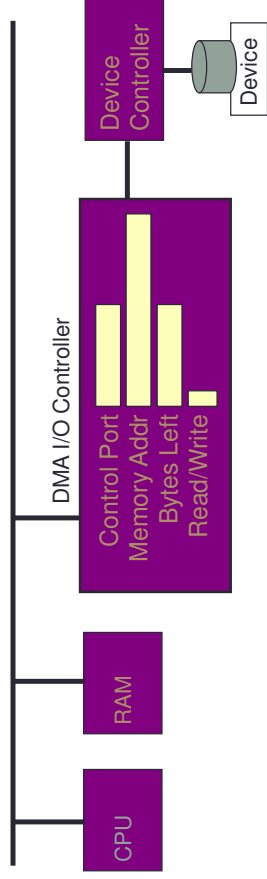
Interrupt-driven I/O trade-offs

- Big step forward when compared with programmed I/O
- Interrupt-processing time is relatively expensive – some overhead with saving and restoring CPU state etc.
- Bad for high-speed, high-data volume devices that might lose data if they are not serviced quickly enough
- Bad if many devices continually require attention

We need to be able to reduce the number of interrupts generated

DMA I/O for block transfers

- CPU writes **start address** of block, **number of bytes** of block and **direction of transfer** to DMA's I/O ports and issues start command
- DMA controller transfers block of data between the device and main memory **without direct CPU intervention**
- On completion, **DMA controller interrupts CPU**



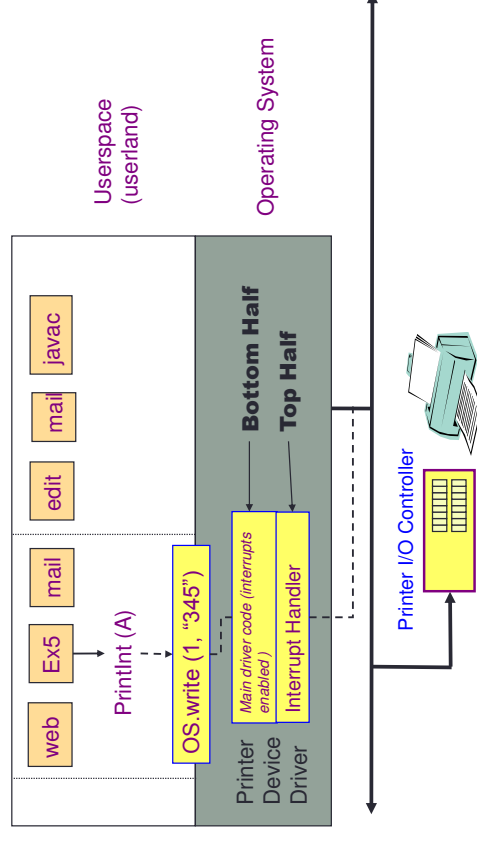
I/O processors

- Sometimes even **DMA I/O is inadequate**
- More powerful approach is to use one or more **dedicated I/O processors to relieve the CPU of I/O related tasks**
- I/O processors are **more capable than DMA controllers** and are often **full-featured CPUs** with their **own local memory** for buffering data and executing I/O related tasks

DMA speed-up

	Time to perform N-byte transfer
Interrupt I/O	(Time for 1 interrupt + Memory access time) * N
DMA I/O	Time for 1 interrupt + (Memory access time) * N

Device drivers



Example: printer device driver

- To write a string to the printer. String terminated by a zero byte.

Bottom-Half (interruptable/scheduled)

```
Copy 1st char from string to
Printer's Data Port

Issue Write Request by writing 1 to
Printer's Control Port W bit

OS.SUSPEND (suspend bottom-half
& run another process)
...
Return from Top-Half
```

Top-Half (Interrupt Handler)

```
IF not end-of-string {
  Copy next char from String to
  Data Port
  Issue Write Request by writing 1
  to Control Port's W bit
} ELSE {
  OS.RESUME bottom-half (when
  convenient)
}
Return from Interrupt
```

Our printer's I/O ports



STATUS (checked by reading the Control Port's W and E bits)

```
Bit W=0 Printer is idle
Bit W=1 Printer is busy
Bit E=0 No Error (All's well)
Bit E=1 An Error has occurred

COMMANDS (issued by writing to the Control Port's W bit)
Set W=1 Causes ASCII Character in Data Port to be printed. On
completion I/O controller will reset bit W to 0 and
generate an Interrupt (see later).
```

Questions

- How does the CPU know **which** interrupt handler to call?
- How is an interrupt handler **actually** called?
- What happens when we **return from a handler**?
- How is **data passed between the top and bottom halves** of the device driver?
- What happens if **another interrupt is signalled** while the interrupt handler is executing?

Bottom half of printer driver

```
controlport equ 80004444h ; address of control port
dataport equ 80004445h ; address of data port
string resb 8192 ; 8Kb string buffer
strptr resd ; ptr to chars in string buffer

bottomhalf: mov eax, strin ; get address of string
mov [strptr], eax ; save pointer to 1st char
mov al, [eax] ; get 1st char
test al, al
jz skip ; skip if char is zero-byte
mov [dataport], al ; else copy char to data port
or [controlport], 20H ; and issue write request
call OS.SUSPEND ; ask OS to suspend bottom-half
ret ; return from bottom-Half

skip:
```

Interrupt handler for printer driver

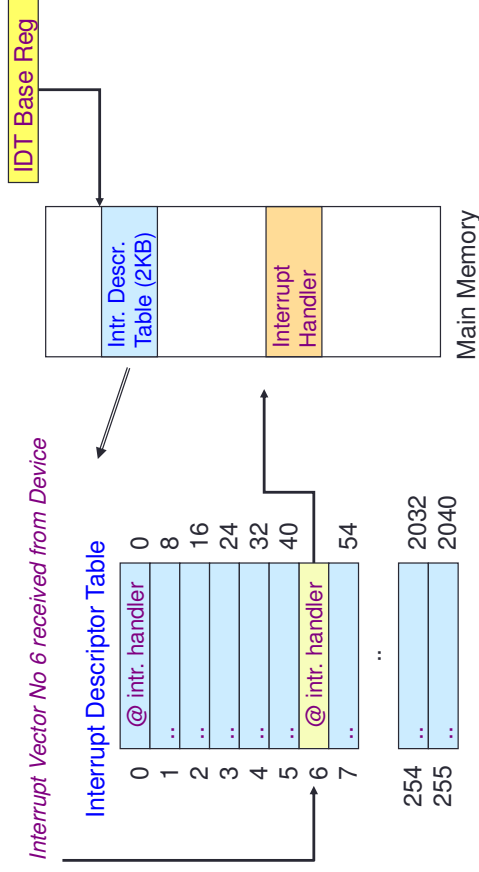
```

handler:  sti          ; re-enable interrupts
          push     eax          ; save registers used in handler
          inc     [strptr]    ; advance to next char
          mov     eax, [strptr] ; copy pointer to register
          mov     al, [eax]    ; and get char
          jz     endofstr     ; skip if end of string
          mov     [dataport], al ; copy char to data port
          or     [controlport], 20h ; issue write request
          jmp     exit        ;
endofstr: call    OS.RESUME   ; ask OS to resume bottom half
          pop     eax
          iret
    
```

Locating the interrupt handler

- When a device wishes to interrupt the CPU it sends a special **interrupt signal** to the CPU along with a **number identifying the interrupt**. This number is called the **interrupt vector number** and on the IA-32 is a number in the range 0 to 255
- Interrupt vector number used to index and **interrupt descriptor table (IDT)** with 256 entries. Each entry is a 64-bit “descriptor” including the handler’s start address (32 bits)
- Start address of the IDT is held in a special CPU register called the **IDT base register (IDTR)**

Locating the interrupt handler



Types of interrupt

- External I/O device-generated interrupts** (**asynchronous**): I/O device sends an **interrupt vector number to the CPU** via buses. Vector numbers 32-255 reserved for these.
 - CPU-generated interrupts (synchronous)**: e.g. attempt to execute an illegal operation, divide-by-zero. Vector numbers 0-18 are reserved for these.
 - Software-generated interrupts (synchronous)**: interrupts can be **generated with an instruction**, e.g. `int 80H` ; Call interrupt handler at vector 80H
- Such interrupts are commonly used to call O/S method (system call) since interrupts can cause the **privilege level to be switched between user and kernel mode**

Calling the interrupt handler

When an interrupt occurs, the IA-32 CPU will:

- Complete the currently executing instruction
- Push the EFLAGS register onto the stack (**why?**)
- Clear the interrupt flag bit in the EFLAGS register – disables further interrupts
- Push the return address
- Jump to the interrupt handler using the IDT



- To return, the handler executes `iret` which restores the state (EFLAGS and EIP) and jumps to the return address on the stack. **Preserving registers is the duty of the interrupt handler**

Device driver summary

- I/O devices are controlled by reading from and writing to I/O ports. With memory-mapped I/O this is simply reading and writing memory locations
- I/O devices signal completion of I/O request by sending an **interrupt vector number** to the CPU. This causes CPU to call the device driver's **interrupt handler**
- **Interrupt handler (top half)** services the interrupt – **checks for errors and copies data to/from memory** area it shares with **bottom half**
- **Bottom half** runs as a **schedulable thread** within the O/S and interacts with the device (via I/O ports), the top half (via shared memory) and the user-level process

Enabling and disabling interrupts

- IA-32 **automatically disables interrupts on entering handler and restores them when returning from the handler**. Prevents other devices from interrupting the handler.
- Interrupts can be **(re-)enabled more quickly** by setting the **interrupt enable flag (IF)** (bit 9 of EFLAGS) with the `sti` instruction
- Interrupts can be **disabled explicitly** by clearing the **interrupt enable flag (IF)** with the `ccli` instruction. This is usually done to **prevent corruption of shared data** – perhaps between the top and bottom halves of a driver

Interrupt vector numbers

- IA-32 supports up to 256 interrupts numbered from 0 to 255. This number is called the **interrupt vector number**

Examples:

Interrupt vector no.	Cause
0	Division by zero
6	Attempt to execute instruction with illegal opcode
12	Stack fault exception
17	General protection violation
32-255	User-defined – used for interrupts from I/O devices

Software interrupts (system calls)

- Programs can **generate an interrupt** with the `int` instruction, .e.g.:

```
int 80H ; Calls interrupt handler at vector number 80H
```

- Commonly used for **calling operating system functions (system calls)** where the address of the function need not be known and the **privilege level must be escalated in a controlled fashion**
- Also known as **traps**

Linux system calls

- Interrupt 80H is used for >150 Linux system calls
- **System call number** passed in register EAX, e.g.: 1 = exit program, 3 = read from standard input, 4 = write to standard output
- **Parameters 1,2,3,4,5** are passed in registers EBX, ECX, EDX, ESI, EDI. Result is passed back in EAX or via reference parameters
- **Example:** write string to standard output

```
mov eax, 4 ; Linux system call 4: Write( )
mov ebx, 1 ; Parameter 1: 1=File descriptor for std output
mov ecx, AddrOfString ; Parameter 2
mov edx, LengthOfString ; Parameter 3
int 80H ; Call Operating System
```