

I/O Structure

How data travel in and out of the machine.

youtube.com/ElhosseiniAcademy

Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

02

I/O Structure

A large portion of operating-system code is dedicated to **managing I/O** — because of its importance to reliability and performance, and because devices vary so widely.

What Is I/O?

CPU and memory alone are not enough — a computer must **interact with the outside world**.

DEFINITION

The overall process of **moving data in and out** of a computer system.

`in →` Reading a file from disk

`← out` Writing a file to disk

`↔` Browsing the web — both at once

IT HAPPENS THROUGH

- **Devices** — keyboard, mouse, monitor, printer
- **Programs** — reading and writing files
- **Networks** — sending and receiving data

Browser → sends request → Internet → Web Server
Web Server → sends response → Internet → Browser
OUT: “Please send me this webpage.”
IN: “Here is the webpage and its contents.”

In short: I/O is everything that goes into and out of a computer.

Analogy — The Restaurant

The Device Driver is **software** inside the operating system. It knows how to communicate with that controller.

CUSTOMERS ORDERING

I/O

The interaction itself —
requests going in, food coming out .

KITCHEN EQUIPMENT

Device controller

The Device Controller is the hardware that directly manages the physical device.

CHEF'S RECIPE BOOK

Device driver

The **instructions** that explain how to use the equipment correctly.

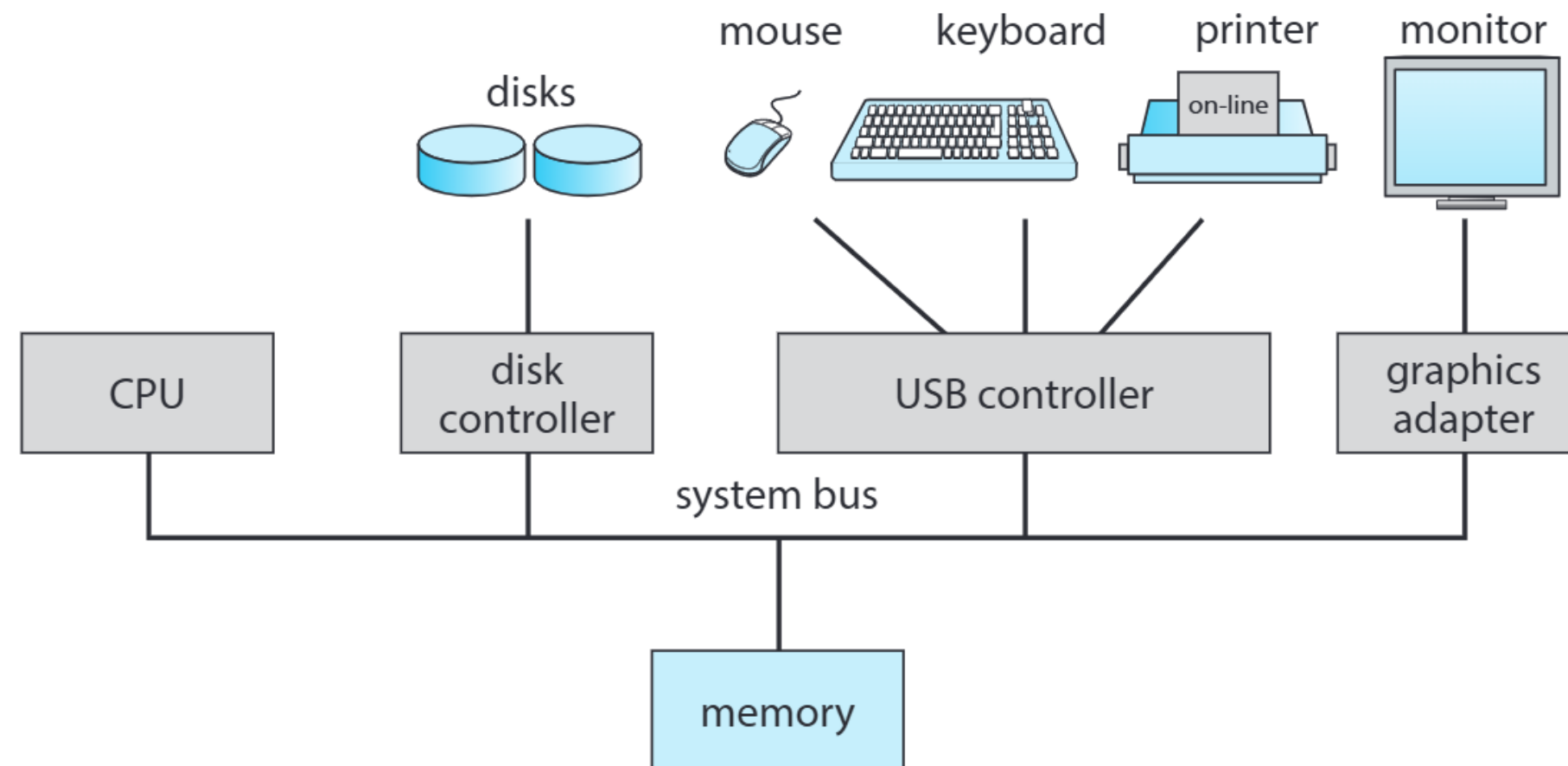
Disk **Controller** → controls disk
Keyboard **Controller** → handles keyboard events
USB **Controller** → manages USB devices

The controller usually has things like:
Control Register — what should the device do?
Status Register — is it ready/busy/done?
Data Register / Buffer — where data temporarily moves.

Device = oven Device **Controller** = oven control panel/electronics

In short: I/O = interaction | controller = hardware | driver = instructions.

Everything Shares One Bus



A general-purpose computer consists of **multiple devices**, all of which exchange data over a **common bus**.

System bus = one shared road
CPU, memory, disk controller, USB controller = cars trying to use the same road.

Every byte moved by I/O has to **cross this bus** — which is exactly where the cost shows up next.

Some high-end systems use a **switch** rather than a bus: multiple components then talk **concurrently** instead of competing for cycles on one shared path.

Switch = road network with several independent lanes/routes

If a device transfers a lot of data, for example a **fast SSD or network adapter**, the bus can become a **bottleneck** because all those bytes must cross the same path

Where Interrupt-Driven I/O Runs Out

WORKS WELL

Moving **small amounts of data** — a keystroke, a mouse move, a status change.

Key pressed → IRQ → CPU runs ISR → reads data → returns

One interrupt per byte is **affordable when bytes are rare**.

BREAKS DOWN

Bulk data movement such as NVS I/O — where one **interrupt per byte** becomes **crushing overhead**.

Interrupt → ISR → Return → Interrupt → ISR → Return → Interrupt → ...

The CPU spends its time **answering the door instead of working**.

- For bulk I/O, modern systems therefore avoid one interrupt per byte
- The device/controller usually **transfers a block or buffer of data**, often using DMA (Direct Memory Access), and then raises one interrupt when the block is complete.

DMA exists so the CPU does not have to personally move every byte.

Where Interrupt-Driven I/O Runs Out

DMA (Direct Memory Access) is hardware logic—a small controller/engine that can move data between an I/O device and main memory without the CPU copying every byte itself

- In older computers, DMA could be a separate DMA controller chip.
- In modern systems, the DMA engine is usually built into the device controller, chipset, or SoC.
- For example, a network card, SSD controller, GPU, or USB controller may contain its own DMA hardware.

Conceptually:
CPU tells DMA:
source = device
Buffer destination = RAM
Address size = 64 KB
direction = device → memory

Then the DMA hardware takes over:
Device → DMA engine → system interconnect/bus → RAM
Meanwhile, the CPU can continue doing other work.
When the transfer finishes:
DMA/controller → IRQ → CPU
So the CPU is involved mainly at the beginning and end, not for every byte.

Direct Memory Access (DMA)

01

The CPU sets up **buffers, pointers and counters** for the I/O device.

02

The device controller transfers an **entire block** between device and main memory.

03

No CPU intervention during the transfer — the controller does it alone.

04

One interrupt per block tells the driver the operation has completed.

While the controller works, the CPU is **free to accomplish other work** .

On **switch-based systems** , where components communicate concurrently rather than sharing a bus, DMA is **even more effective**.

One Block, Two Ways

Transferring a single **4 KB block** from disk into memory.

INTERRUPT PER BYTE

4,096

interrupts — each one a context save, a vector lookup, a handler, and a restore.

WITH DMA

1

interrupt — raised once, when the whole block has landed in memory.

In One Line Each

1 The CPU can load instructions **only from memory** — so programs must be loaded into RAM to run.

2 RAM is **volatile** , so the bootstrap program lives in **firmware** — ROM, EEPROM, or flash.

3 Memory is an **array of addressed bytes** , reached only by **load** and **store**.

4 **Secondary storage** exists because main memory is **too small and too volatile**.

5 The **hierarchy** trades speed against size and cost, with the fast, small levels **nearest the CPU** .

6 **DMA** moves whole blocks with **one interrupt** , leaving the CPU free to work.

Check Your Understanding

Q1

Why can the bootstrap program not be stored in RAM, and what property of EEPROM solves it?

Q2

A machine has 64-bit registers. How many bytes is one word, and why does that matter for load and store?

Q3

In Figure 1.6, which level is the first nonvolatile one — and what is directly above and below it?

Q4

Why does DMA reduce interrupts rather than eliminate them — and who receives the one that remains?