**Overview**

Performance is an important aspect.

Different devices need different speeds. Printer vs mouse

Device Driver: Software, written by manufacturers. Uniform interface between I/O and OS.

Ports, busses, device controllers:

CPU <-> Bus <-> Memory Bus <-> Memory/RAM

Memory address register.

Memory data register.

CPU puts address and data in relevant registers before writing to RAM.

**OLD ARCHITECTURE (modern vs old might be a exam Q) – Direct IO**

I/O Bus: Connects Device controller and CPU.

Each of these I/O devices have an I/O **port**.

These ports are linked to one of the registers on the device controller.

IN CPU, you have IN and OUT instructions (from CPU perspective)  
IN: Reading in from device controller  
OUT: Writing out from CPU.

egCPU wants to read in from port ID 3. This could be eg the Keyboard presses.

This is an extremely expensive methods as having many different busses were expensive. IN/OUT are privileged instructions (only assembly can use them… ie high level device drivers couldn’t use them… so had to build them in assembly)

**This is why we want to use memory map**

**MODERN ARCHITECTURE**

**Memory mapped I/O:** When computer starts up, queries all devices for what they are. In memory addresses, assigns a bunch of memory addresses for each one. Therefore IN/OUT not needed, can just use normal variables (eg INT). Can now use high level languages.

For security, whenever reading or writing occurs, CPU is active.

Eg of Memory maps for Monitor: (**IN EXAM NB – why are there 2 memory ranges)**  
Has 2 memory map ranges.   
1 for the current state.  
1 for the changed state.

**Memory Range:** Then the screen changes state from the current, to changed state (from those memory banks). It will keep swapping between the 2, allowing the effect of movement to be seen. (In memory bank, not RAM? – might be for a diff register)

**I/O range:** to give control info to VGA card (ie resolution) (**IN EXAM NB – why does the I/O port have a range)**

(**IRQ)**

**PORTS:** Used when describing the I/O ports. It is a connection point for a device. (**Direct I/O instructions,** for backwards compatibility. Eg Bootup code)

**I/O Hardware**

**Shared Direct Access:** One Bus that connects many devices

**Daisy chain:** Connects one device to another, in a link. A to B to C.

Integrated controller: One chip on motherboard that handles a bunch of devices. Eg handling sound, keyboard input ect. (There can be a few of these).

Devices normally have registers where device driver places commands, address and data to write/read data from after command execution (**NB).**

From **device controller** perspective, there are 4 registers: (in context of host)

Data-IN: Into CPU eg KeyInput  
Data-OUT: Out from CPU eg Print Page on Printer. (write character to printer buffer)  
Status register: eg Out of paper, busy printing ect  
Control register: Controls something to do with the device. Tells device x is going to happen now. Now printing will occur, be ready to receive data.

**Polling**

Devices + Controllers.

For each byte of I/O:

1. Go to controller, check if device connected to it if busy or not (**status register** - **busy bit**).
2. Host sets **read or write bit** in **control register**.
3. If writing, writes **copy of data** **into data out register**.
4. Host sets **command-ready** bit.
5. Controller sets **busy bit (status register**), **executes transfer**
6. Controller **clears** **busy** bit, **error** but, **command**-ready bit when done.

Disadvantage: (**BUSY-Wait cycle for I/O devices)**

For slow devices, CPU is just spinning, wasting CPU time.  
Only okay for fast devices. (As I/O super slow compared to CPU – eg keyboard typing speed vs CPU).

A solution would be CPU doing other tasks during the down time, but issue is during that time CPU can miss an instruction. **SOLUTION: Interrupts.**

**Interrupts**

Keyboard raises interrupt when there is a change in state, to inform CPU. So CPU only has to read the change, and can go back to what it was doing.

**Interrupt handler**: software that handles interrupts (small part of device driver)  
-can be **maskable**: ie can be ignored or handled later on.

**Interrupt vector:** A table. Has 2 main things. Interrupt number, pointer to interrupt handler.

1. Save state of process
2. Run interrupt handler
3. Restore previous process

A diagram of a computer process

Description automatically generated

Processer built to handle 0-255 interrupts. (**NBNBNB exam)**

**Direct memory access**

**Programmed I/O: This is the same thing as Busy-waiting, where CPU in loop. NBNB to know. We want to get rig of programmed I/O.**

**While interrupts** is a **good solution,** it still has a **problem.** For every byte loaded from IO, an interrupt occurs (wastes a eg byte).

**SOLUTION: Direct Memory Access (DMA)**

Handles memory transfer from devices to memory, bypassing CPU.

DMA is programmed with the:

1. source and destination address.
2. Read or write mode
3. Count of bytes
4. Writes location of command block to DMA controller
5. At end, interrupt occurs.

Therefore advantage is one interrupt occurs per block of bytes, instead of each byte.