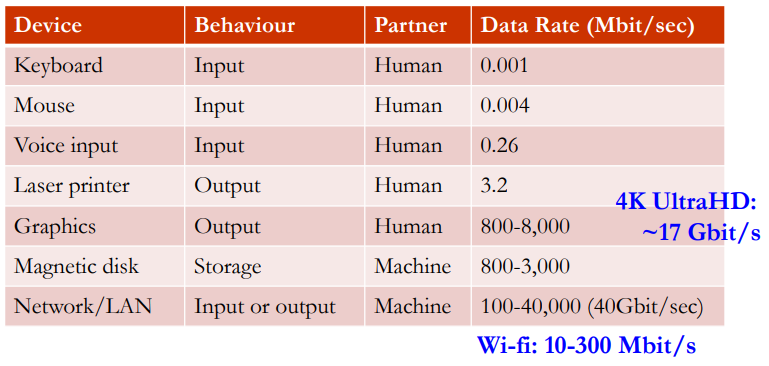
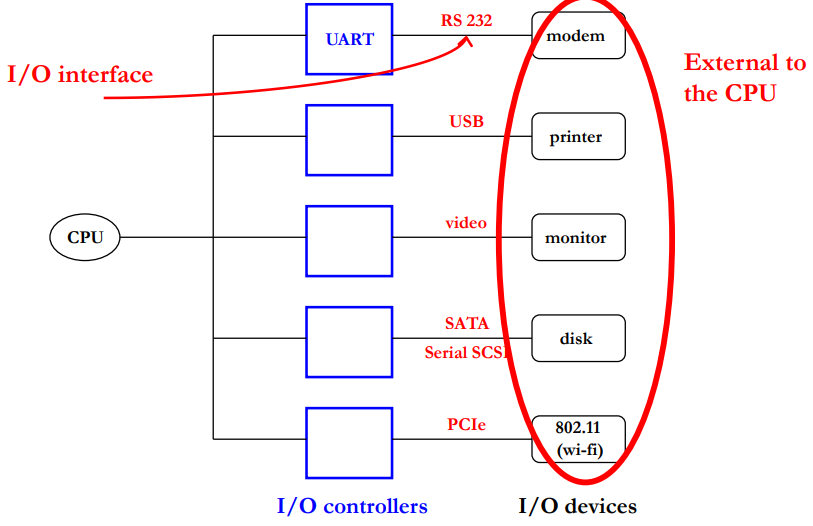
Computer Systems Lecture 21

Examples of I/O Devices



I/O Controllers & Devices

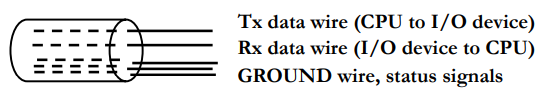


Example: RS232 Serial Interface

I/O controller: UART (stands for Universal Asynchronous Receiver Transmitter)

Used for modems and other serial devices

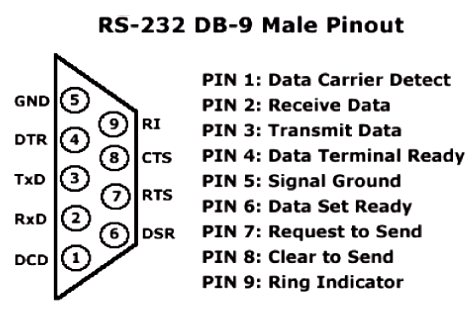
Physical Implementation: 2 data wires (one for each direction) + ground reference + status signals (this is very slow)



RS232 Modem: Bits and Wires

A modem is the old way of connecting to the internet, it worked through the phone lines.

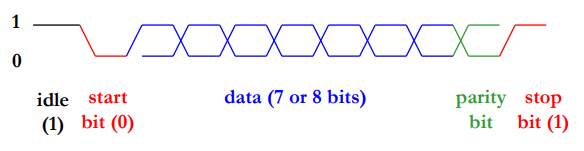


Example: RS232 Serial Interface

Encoding:

1 ASCII character = 10 or 11 bits (including signalling)

The Idle state is represented by a constant 1



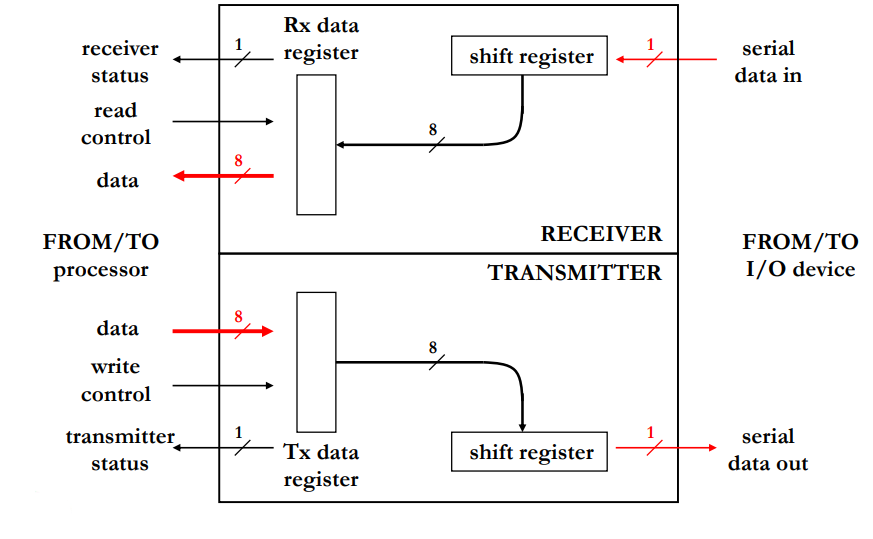
Parity: a 1-bit code for detection of transmission errors

- odd parity -> total number of 1’s (including parity bit) is odd

- even parity total number of 1’s is even

This doesn’t work if any even number of bits flip

UART Controller

Here both receiver status and transmitter status are exception lines. Receiver status tells the cpu that data has arrived, transmitter status tells the cpu that the transmitter is able to send data (will be 0 if its currently sending data)

Connecting CPU and I/O Controllers

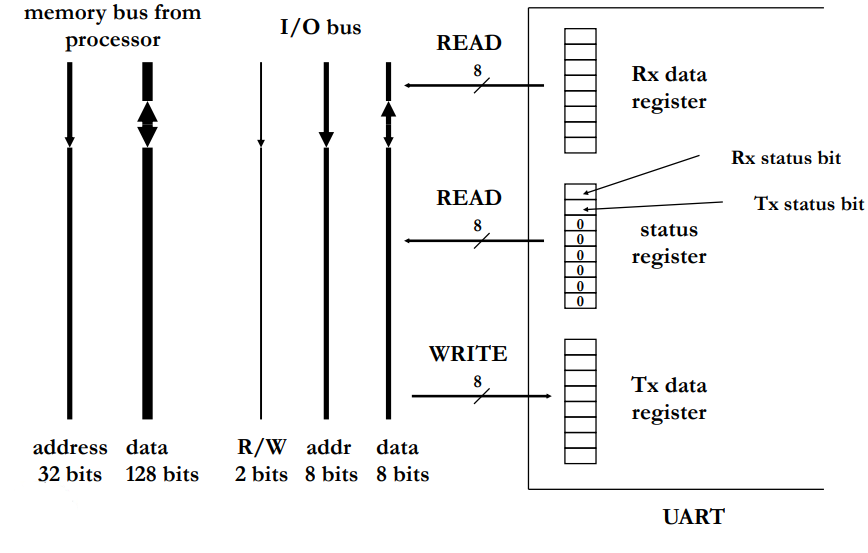
Option 1: connect the I/O tx and Rx registers directly into dedicated CPU I/O registers -> not flexible, not expandable

Option 2: keep I/O registers in a separate I/O controller and connect the CPU to the I/O controller through a special I/O bus -> requires a separate interface for I/O, potentially expensive I/O bus:

* Data lines (8 bits)
* Control lines (Read and Write signals)
* Address lines -> each I/O controller is assigned a range of addresses for its registers

Data is accessed through special I/O loads and stores

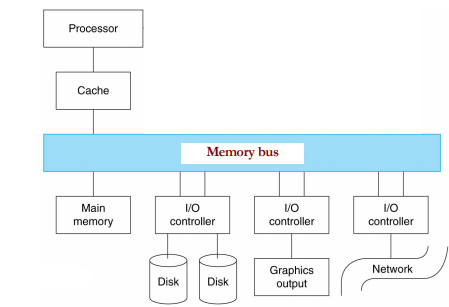
I/O via a Dedicated I/O Bus



On the left is the memory bus. In the middle is the I/O bus and on the right is an example of the I/O bus connecting to the UART.

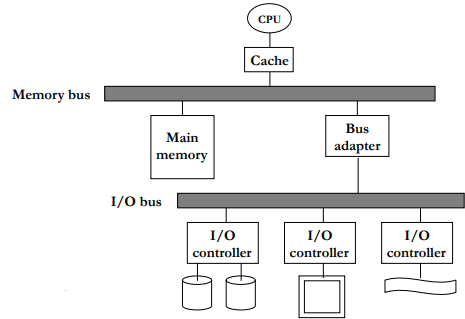
The I/O bus consists of 2 bits one for read one for write, 8 bits for the address and 8 bits for the data.

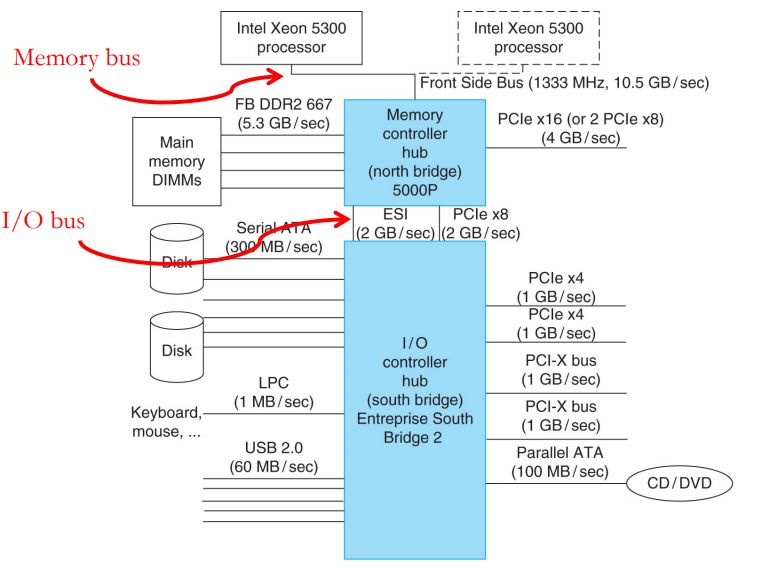
For UART, when the read line is set, the data register and the status register are sent on the data lines (which one is decided by the address bits) and when the write line is set the transmit data register is read from the data lines.

Option 3: keep I/O registers in I/O controller and connect to the CPU to the I/O controller through the memory bus this is called Memory-Mapped I/O:

* I/O controller registers (data and control) are mapped to a dedicated portion of memory, accessed via the memory bus
* Good news: accessed with regular load and store instructions
* Bad news: Takes bus bandwidth away from the CPU accessing the Memory as we’re adding lots of slow devices to a fast bus

Option 4: connect I/O controllers to I/O bus and the I/O bus to the memory bus through a bus adapter

* Off-load the I/O from the memory bus: multiple I/O devices appear as a single device to the memory bus
* Pros:
  + Better performance: slow and narrow I/O bus doesn’t clog up the fast memory bus
  + Higher flexibility: add/remove devices without impacting the high-performance processor-memory interface
  + Modularity and extensibility: memory bus and I/O bus technology can evolve separately

This is a conceptual view of an old chipset. The top boxs are the CPU. This then connects to the memory controller hub through a very fast connection (essentially the memory bus), this connects to the memory devices and possibly a graphics card through slower connections (5.3 and 4 GB/s connections). It also connects to the I/O controller hub through a again slower connections (2GB/s) and this connects to all of the slow devices through much slower connections (ranging from 300MB/s to 1GB/s).

Polling and Interrupt-Based I/O

How do we check I/O status (e.g. key clicked)?

Option 1: Polling

* User process calls OS at regular intervals to check status of I/O operation
* Wasteful for events that happen very infrequently (e.g. keyboard clicks)

Option 2: Interrupt

* I/O controller interrupts user process to signal an I?O event
* Heavy-0weight (break out of user code into kernel mode)

USB Example

USB devices don’t generate interrupts

* This keeps cost low

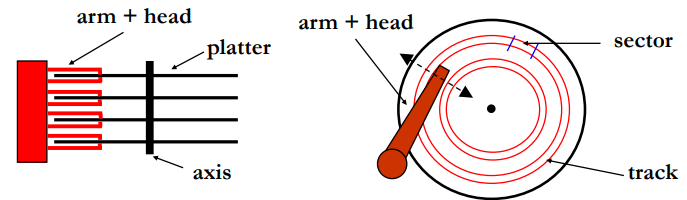
Computer can’t afford to keep polling the multitude of USB ports

* Would lose too much performance for nothing
* Not polling often enough not an option: will lose data

USB controllers do the polling (via a simple FSM) and generates an interrupt to inform the CPU as needed

* This functionality is in the SouthBridge (I/O controller)

Hard Disks



* 1-4 platters per drice, 2 surfaces per platter, 10-50k tracks per surface, 100-500 sectors per track, 512B-4KB per sector
* Spinning speed: 5400-15000 rpm (90-250 revs per sec)

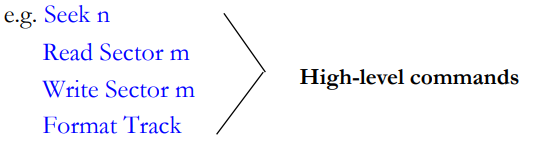
Disk Performance

The total time of a disk operation is divided into two parts:

* Access time: time to get the head into position to read/write data
  + Access time = seek time + rotational latency
  + Seek time: time to move head to appropriate track (< 10ms)
  + Rotational latency: time to wait for appropriate sector to arrive underneath the head (< 10ms) dependent on the spinning speed
* Transfer time: time to move data to/from disk
  + Transfer time = time to transfer 1 byte \* number of bytes of data
  + Dependent on both spinning speed and recording density
  + 75-125 MB/s (changing very slowly -limited by mechanics)

Disk Controllers

The disk controller is inside the disk unit and is responsible for all mechanical operation of the disk as well as the interface with the CPU, its I/O registers are used to exchange data and status words between the CPU and the disk controller, its command register tells the disk controller what to do next.



Using a Disk Controller

* Step 1: user program requests data from a file
* Step 2: OS file system determines sector(s) to be accessed
* Step 3: OS disk handler issues Seek command and CPU goes to work on some other process (multi-tasking)
* Step 4: I/O controller interrupts CPU to signal completion of seek
* Step 5: OS disk handler issues Read Sector command and CPU goes to work on some other process
* Step 6: I/O controller interrupts CPU to signal data ready
* Step 7: OS disk handler transfers sector data from disk controller to memory
  + This is a slow-running for loop that transfers data word-by-word via the I/O and memory busses
* Step 8: go to step 3 or 5 and repeat until all the data is transferred

Problem with the Interrupt Approach

CPU (through the OS) has to issue individual commands to read every sector from disk

The frequent interrupts are detrimental to the processor performance

* Switch to OS, then switch back to the application

The CPU moving the data from the disk controller via the slow I/O bus results in highly inefficient CPU utilization

The solution to this is Direct Memory Access (DMA)

Direct Memory Access (DMA)

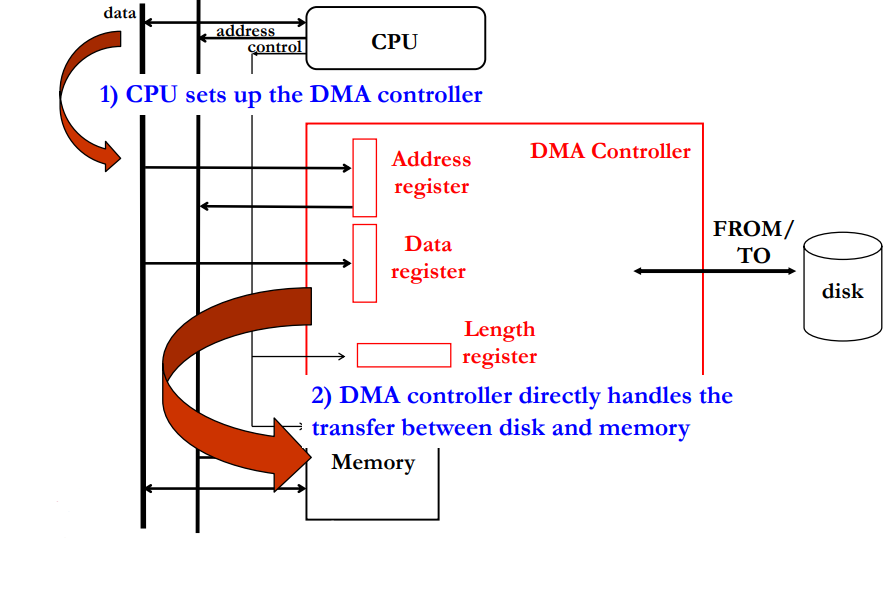
DMA controller: stateful device that sits on the memory bus and can independently transfer data between memory and disk

* Setup by the processor for each transfer using memory-mapped registers inside the DMA controller

DMA registers:

* Address register: position in memory of next data to be read/written
* Data register: temporary storage for data to be transferred
* Length register: number of bytes remaining to be transferred

DMA Organization

The CPU sets the address register and the length register to the appropriate values.

Once the data has been read, it sends an interrupt to the CPU saying that all of the requested data is now in memory

DMA Operation

Step 1,2: user program requests data, OS determines the location of the data on the disk

Step 3: the OS disk handler issues a Seek command and sets up the DMA registers (address, length); CPU goes to work on another process

Step 4,5: I/O controller interrupts CPU, OS disk handler issues Read Sector command

Step 6: I/O controller informs DMA controller that data is ready (no need to interrupt CPU)

Step 7: DMA controller transfers data into memory; length register is decremented until all data is moved (advanced DMA controllers can access multiple tracks with a single operation)

Step 8: DMA control interrupts the CPU to indicate the completion of the DMA operation

Bus Arbitration

DMA and CPU connect to memory bus -> access must be somehow arbitrated to avoid conflicts

* Note: I/O devices are slaves and cannot initiate bus transactions (can only respond)

Solution: additional logic (bus arbiter)

* Authorizes CPU or DMA controller to access memory at any given time

Two new wires are added to the memory bus:

* Bus Request -> asserted by CPU and/or DMA when it requires the bus
* Bus Grant ->asserted to indicate who (CPU or DMA) can use the bus
* In case of conflicting requests in the same cycle, CPU has priority

