I/O - input/output

system components: CPU, memory, and bus
-- now add I/O controllers and peripheral devices

CPU must perform all transfers to/from simple controller, e.g., CPU reads byte from buffer in memory and stores it in controller's data register then stores a write-to-device command in the controller's command register.
I/O - input/output

controller registers

- data register - holds data byte going to/from device
- status register - holds bits indicating if device is ready, error, etc.
- command register - bit for read, bit for write, etc. (may be combined with the status register)
I/O - input/output

access to controller registers either by:

• memory-mapped I/O

• isolated I/O - special instructions (e.g., IN, OUT, INS, OUTs) are required, use port numbers as addresses of the controller registers
I/O - input/output

- memory-mapped I/O
  - one address space is divided into two parts.
  - Some addresses refer to physical memory locations.
  - Other addresses actually reference peripherals.
  - registers respond to main memory addresses (typically high memory), so you can use normal load/store instructions to access
  - The I/O addresses are shared by many peripherals.
  - Some devices may need several I/O addresses
To send data to a device, the CPU writes to the appropriate I/O address. The address and data are then transmitted along the bus.

The device only responds when its address appears on the address bus.
isolated I/O

- supports separate address spaces for memory and I/O devices, with special instructions (e.g. IN, OUT, INS, OUTs) that access the I/O space.
- use port numbers as addresses of the controller registers
- Regular instructions like MOV reference RAM.
- The special instructions such as IN, Out access a separate I/O address space
## I/O - input/output

differences between memory-mapped I/O and memory-mapped I/O

<table>
<thead>
<tr>
<th>Isolated I/O</th>
<th>Memory Mapped I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolated I/O uses separate address space for memory and I/O devices.</td>
<td>Memory mapped I/O uses same address bus to address memory and I/O devices</td>
</tr>
<tr>
<td>Limited instructions can be used (e.g. IN, OUT, INS, OUTS).</td>
<td>Any instruction which references memory can be used to access I/O device (e.g. LDR, STR).</td>
</tr>
<tr>
<td>The addresses for Isolated I/O devices are called ports.</td>
<td>Memory mapped I/O devices are treated as memory locations on the memory map.</td>
</tr>
<tr>
<td>Efficient I/O operations due to use of separate bus</td>
<td>Inefficient I/O operations due to using single bus for data and addressing</td>
</tr>
<tr>
<td>Comparatively larger in size</td>
<td>Smaller in size</td>
</tr>
<tr>
<td>Uses complex internal logic</td>
<td>Common internal logic for memory and I/O devices</td>
</tr>
<tr>
<td>Slower operations</td>
<td>Faster operations</td>
</tr>
</tbody>
</table>
Comparison of Memory-Mapped and Isolated I/O

- Memory-mapped I/O with a single address space is nice because the same instructions that access memory can also access I/O devices.

- With isolated I/O, special instructions are used to access devices. This is less flexible for programming.
I/O - Transferring data between a device and Memory

• The second important question is how data is transferred between a device and memory.
  – Programmed I/O
  – Interrupt-driven I/O
  – DMA I/O
I/O - Transferring data between a device and Memory

- Under programmed I/O, it is all up to a user program or the operating system.
  - The CPU makes a request and then waits for the device to become ready (e.g., to move the disk head).
  - Buses are only 32-64 bits wide, so the last few steps are repeated for large transfers.
- A lot of CPU time is needed for this! If the device is slow the CPU might have to wait a long time
  - most devices are slow compared to modern CPUs.
  - The CPU is also involved as a middleman for the actual data transfer.
I/O - Transferring data

- Programmed I/O

CPU sends read request to device

Nor Ready

CPU waits for device

Ready

CPU reads word from device

CPU writes word to main memory

No

Done?

Yes
I/O - input/output

// write bytes from memory buffer to device

/***
pseudo-code

int count = N;
char *addr = memory_buffer;
char byte;

do{ byte = *addr;

    while( io_device_status != READY ) /* busy wait */ ;

    io_device_data = byte;
    io_device_command = WRITE;

    addr++;
    count--;
}

}while( count > 0 );

*/
I/O - input/output

//defines may differ for real device)
define(DEVICE_REGS_BASE, 0xffff0000)
define(DEVICE_READY, 0x80) // mask that defines ready bit in status reg.
define(WRITE_CMD, 0x1) // mask to set write command in command reg.
define(DATA, 0) // offset of data reg.
define(STATUS, 4) // offset of status reg
define(CMD, 8) // offset of command reg.

ldr addr_reg, =buffer
ldr count_reg, =buffer_size
ldr device_reg_ptr, =DEVICE_REGS_BASE // memory-mapped i/o
mov cmd_reg, WRITE_CMD

loop:
  ldrub byte_reg, [addr_reg] // get data byte from buffer

busy_wait:
  ldrub status_reg, [device_reg_ptr_r, STATUS]
  btst status_reg, DEVICE_READY // andcc to test ready bit
  beq busy_wait // branch if zero

ready:
  strb byte_reg, [device_reg_ptr, DATA] // send data byte
  strb cmd_reg, [device_reg_ptr, CMD] // send write command

  add addr_reg, addr_reg, #1
  sub count_reg, count_reg, #1
  cmp count_reg, #0
  bgt loop
I/O - input/output

consider 24 ppm printer, with 5,000 characters per page
  = 120,000 chars/min
  = 2000 chars/sec = 0.0005 sec/char = 0.5 ms/char

for a 3.6 GHz processor (= 3.6 B cycles/sec), the cycle
time is approx. 0.28 ns, thus 560,000 cycles in 0.5 ms,
and thus 560,000 cycles between characters

if the busy wait loop takes 100 cycles per iteration (of
which most will be required for the latency of the load
inst. accessing the device status register), the busy wait
loop requires 5,600 iterations between characters

1 success in 5,600 iterations => not an efficient use of
the CPU  (CPU spends 99.9+% of its time waiting)
/*
read bytes from device to memory buffer

pseudo-code

    int count = N;
    char *addr = memory_buffer;
    char byte;
    do{ io_device_command = READ;
        while( io_device_status != READY ) /* busy wait */ ;
        byte = io_device_data;
        *addr = byte;
        addr++; count--;
    }while( count > 0 );
*/
I/O - input/output

/** defines may differ for real device */

define(DEVICE_REGS_BASE, 0xffff0000) // mask that defines ready bit in status reg.
define(DEVICE_READY, 0x80) // mask to set read command in command reg.
define(READ_CMD, 0x2) // offset of data reg.
define(DATA, 0) // offset of status reg.
define(STATUS, 4) // offset of status reg.
define(CMD, 8) // offset of command reg.

ldr addr_reg, =buffer
ldr count_reg =buffer_size

ldr device_reg_ptr =device_regs_base // memory-mapped i/o
mov cmd_reg, READ_CMD

loop:
  strb cmd_reg, [device_reg_ptr, CMD] // send read command

busy_wait:
  ldrub status_reg, [device_reg_ptr, STATUS]
  btst status_reg, DEVICE_READY // andcc to test ready bit
  beq busy_wait // branch if zero

ready:
  ldrub byte_reg, [device_reg_ptr, DATA] // get data byte
  strb byte_reg, [addr_reg] // put in buffer

add addr_reg, addr_reg, #1
sub count_reg, count_reg, #1
cmp count_reg, #0
bgt loop
Interrupt-driven I/O

• An interrupt is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution.

• This hardware event is called a trigger. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer).

• When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag.
Interrupt-driven I/O

CPU can do something else while controller and device are busy; the controller grabs the CPU's attention when needed by causing what is essentially an unplanned procedure call.
Interrupt-driven I/O

we rely on an external interrupt from the controller to signal that the device is ready (i.e., that the previous I/O operation is complete);

this will cause the currently executing program to stop and the processor to enter the OS and start executing an interrupt service routine (ISR) -- sometimes called an interrupt handler (IH)

there are also internal interrupts (sometimes called exceptions) for divide by zero, unaligned memory accesses, memory protection errors, etc.
Interrupt-driven I/O

- moreover, to protect the OS, calls to the OS must be made by a special instruction that causes an interrupt-called SVC (supervisor call) on IBM mainframes, INT on x86.

- an interrupt must save a return address and information on the processor state to allow the interrupted program to be resumed later => save the program counter (PC) and processor state register (PSR)

- an interrupt switches execution mode to an OS-only mode by changing a mode bit (or bits) in the PSR
Interrupt-driven I/O

- there are typically interrupt control bits in the controller's command register, and interrupt enable bits (either a priority level or a bit mask) in the PSR - the processor typically disables interrupts (at least at that level and lower) whenever an ISR starts.

- the entry point address to the interrupt service routine (ISR) is typically provided by a table of such addresses in low memory; for I/O the entry is chosen according to the interrupt code placed on bus by controller.
Interrupt-driven I/O

The diagram illustrates an interrupt vector table (IVT) that maps to different interrupt service routines (ISRs). Each entry in the IVT corresponds to a specific type of interrupt:

- Type 0: addr of ISR for type 0
- Type 1: addr of ISR for type 1
- Type 2: addr of ISR for type 2
- Type 3: addr of ISR for type 3

The IVT contains pointers to the locations where the actual code implementing the interrupts is stored. The diagram also shows the flow of execution from an interrupt request to the return from interrupt, indicating the sequence of events when an interrupt occurs.

- ... (Indicating additional entries in the IVT)
- ... (Indicating more code for type 3 interrupt)
- return from interrupt
- ... (Indicating more code after returning from interrupt)

The IVT is a critical component in managing interrupts in a computer system, ensuring that each interrupt is handled by the appropriate routine.
Interrupt-driven I/O

- a special return from interrupt instruction at end of ISR switches back to previous processor state and restores saved PC
- the fetch-execute cycle is extended to check for interrupts after each instruction - the hardware response to an interrupt acts like procedure call if interrupt requested by device and if CPU has interrupts enabled
- note that the ISR is a software routine - and that instructions in the ISR are fetched, decoded, and executed, just like any other program
Interrupt-driven I/O

PC - program counter, contains address of next instruction

PSR - processor status register, contains:
  – processor execution mode (kernel/user)
  – interrupt enable/permission (can be single bit, mask, or priority code)
  – condition codes

IVT - interrupt vector table, contains entry-point addresses for ISRs

ISR - interrupt service routine
Interrupt-driven I/O

- fetch
- decode
- execute load
- execute add
- execute store
- ... return from interrupt (restore PC and PSR)

- if interrupt requested and interrupts are enabled
  - yes
    1) save PC and PSR
    2) switch execution mode to kernel (OS-only)
    3) disable/restrict further interrupts
    4) load new PC from IVT (interrupt vector table)
  - no
    +

...
Interrupt-driven I/O (nested interrupts)

- if the PC and PSR are saved on a stack (or in a set of registers), a high priority device can interrupt the execution of the ISR for a lower priority device.

  otherwise, the second interrupt is held pending until the first ISR finishes and executes its rti instruction; at that point, the rti briefly reestablishes user mode with interrupts enabled but immediately the highest-priority pending interrupt is accepted.
DMA I/O

DMA - direct memory access - extra registers and logic in the controller allow it to transfer a whole block of bytes without CPU involvement, interrupts the CPU after completion or after an error
- address register - address of buffer in main memory, controller increments
- count register - length of block, controller decrements
DMA controller can be bus master, so need to arbitrate for bus among DMA controllers and CPU extra logic in controller to implement loop:

```
while( count > 0 ) {
    transfer byte at address;
    address++;
    count--;
}
```

interrupt once at end of buffer
## Summary of I/O

<table>
<thead>
<tr>
<th>I/O methods</th>
<th>CPU involvement</th>
<th>#interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>programmed I/O</td>
<td>completely dedicated to the transfer</td>
<td>none</td>
</tr>
<tr>
<td>interrupt-driven I/O</td>
<td>transfers each byte</td>
<td>after each byte</td>
</tr>
<tr>
<td>DMA I/O</td>
<td>initially loads the address and count registers; gives transfer command</td>
<td>one, at end of each block</td>
</tr>
</tbody>
</table>
Effect of Offloading CPU

do {
    byte = *addr;
    while( status != READY ); <= interrupt-driven I/O complete block
        io_device_data = byte; relieves CPU of transfer is
        io_device_command = WRITE; busy-wait loop offloaded onto
        addr++;
        count--;
} while( count > 0 );

Further Offloading of I/O from the CPU

mainframe channel - provides for transfers of multiple blocks by traversing linked-list-like channel programs; interrupt only when end of channel program reached (or on error)

a channel has the equivalent of a program counter

each channel instruction is called a "channel command word" (CCW)

<table>
<thead>
<tr>
<th>r/w</th>
<th>address</th>
<th>count</th>
<th>next</th>
</tr>
</thead>
</table>

each CCW has address and count fields to support a block

additional fields indicate end of physical block, etc.
CCWs can also provide scatter-gather

scatter - read data from a single physical block on an I/O device and send different parts to multiple, non-contiguous I/O buffers in memory

gather - read data from multiple, non-contiguous I/O buffers in memory and write a single physical block to the device

I/O processors - offload I/O conversion, editing, etc