Simple Input and Output

Topics for this lecture:

1. I/O strategies: how basic input and output operations are managed.
2. The basics of interrupt handling.
3. The basics of connecting to the Internet, including the NIC (Network Interface Card).
Simple Input and Output

Input / Output and I/O Strategies

The Four Major Input / Output Strategies

Preliminary Definitions

A Silly Example to Illustrate

Basic Definitions

A Context for Advanced I/O Strategies
The Four Strategies

Here are the simple definitions of the four I/O strategies.

**Program Controlled I/O**
This is the simplest to implement. The executing program manages every aspect of I/O processing. I/O occurs only when the program calls for it. If the I/O device is not ready to perform its function, the CPU waits for it to be ready; this is "busy waiting".

The next two strategies are built upon program controlled I/O.

**Interrupt Driven I/O**
In this variant, the I/O device can raise a signal called an "interrupt" when it is ready to perform input or output. The CPU performs the I/O only when the device is ready for it.

In some cases, this interrupt can be viewed as an alarm, indicating an undesirable event.

**Direct Memory Access**
This variant elaborates on the two above. The I/O device interrupts and is sent a "word count" and starting address by the CPU. The transfer takes place as a block.

**I/O Channel**
This assigns I/O to a separate processor, which uses one of the above three strategies.
I/O Strategies: A Silly Example

I am giving a party to which a number of people are invited. I know exactly how many people will attend.

I know that the guests will not arrive before 6:00 PM.

All guests will enter through my front door. In addition to the regular door (which can be locked), it has a screen door and a doorbell.

I have ordered pizzas and beer, each to be delivered. All deliveries at the back door.

I must divide my time between baking cookies for the party and going to the door to let the visitors into the house.

I am a careless cook and often burn the cookies.

We now give the example, by I/O categories.
The Silly Example

Program Controlled
Here I go to the door at 6:00 PM and wait.

As each one arrives, I open the door and admit the guest.

I do not leave the door until the last guest has arrived; nothing gets done in the kitchen.

Interrupt Driven
Here I make use of the fact that the door has a doorbell. I continue working in the kitchen until I hear the doorbell.

When the doorbell rings, I put down my work, go to the door, and admit the guest.

Note 1: I do not “drop” the work, but bring it to a quick and orderly conclusion. If I am removing cookies from the oven, I place them in a safe place to cool before answering the door.

Note 2: If I am fighting a grease fire, I ignore the doorbell and first put out the fire. Only when it is safe do I attend to the door.

Note 3: With a guest at the front door and the beer truck at the back door, I have a difficult choice, but I must attend to each quickly.
The Silly Example (Part 2)

Direct Memory Access
I continue work in the kitchen until the first guest arrives and rings the doorbell.

At that point, I take a basket and place a some small gifts into it, one for each guest.

I go to the door, unlock it, admit the guest and give the first present.

I leave the main door open. I place the basket of gifts outside, with instructions that each guest take one gift and come into the house without ringing the doorbell.

There is a sign above the basket asking the guest taking the last gift to notify me, so that I can return to the front door and close it again.

In the Interrupt Driven analog, I had to go to the door once for each guest.
In the DMA analog, I had to go to the door only twice.

I/O Channel
Here, I hire a butler and tell him to manage the door any way he wants. He just has to get the guests into the party and keep them happy.
Another Simple Example

We first examine program controlled I/O. We give an example that appears to be correct, but which hides a real flaw. This flaw rarely appears in a high–level–language program.

We are using the primitive command “Input” to read from a dedicated input device. It is the ASCII codes for characters that are read, with a 0 used to indicate no more input.

```
Input
Skip if AC > 0
Jump Done
Loop: Store X[J] // Not really a low level instruction
    J = J + 1 // Again pseudo-code
    Input // Assume a dedicated input register.
    Skip if AC == 0
    Jump Loop // Go back and get another.
Done: Continue
```

**What’s wrong?** Simply put, what is the guarantee that the dedicated input device has a new character ready when the next *Input* is executed?
Program Controlled Input and the Busy Wait

Each input or output device must have at least three registers.

**Status Register**
- This allows the CPU to determine a number of status issues.
  - Is the device ready? Is its power on? Are there any device errors?
  - Does the device have new data to input? Is the device ready for output?

**Control Register**
- Enable the device to raise an interrupt when it is ready to move data.
  - Instruct a printer to follow every `<LF>` with a `<CR>`.
  - Move the read/write heads on a disk.

**Data Register**
- Whatever data is to be transferred.

Suppose our dedicated input device has three registers, including `Device_Status` which is greater than zero if and only if there is a character ready to be input.

**Busy:**
- `Input Device_Status`
- `Skip if AC > 0`
- `Jump Busy`
- `Input Device_Data`
When Is Program Controlled I/O Appropriate?

Basically put, it is appropriate only when the I/O action can proceed immediately. There are two standard cases in which this might be used successfully.

1. The device can respond immediately when polled for input. For example, consider an electronic sensor monitoring temperature or pressure. (However, we shall want these sensors to be able to raise interrupts**).

2. When a device has already raised an interrupt, indicating that it is ready to process data.

In a modern computer, the basic I/O instructions (IN and OUT for the IA–32) are considered privileged. They may be issued only by the Operating System.

User programs issue “traps” to the operating system to access these instructions. These are system calls in a standardized fashion that is easily interpreted by system programs.

** NOTE: This is what happened on Apollo–13. The temperature sensor on the service module oxygen tank was read constantly during pre–launch preparation. If it had been designed to signal when the temperature was too high, the disaster would have been avoided.
Diversion: Processes Management

We now must place the three advanced I/O strategies within the proper context in order to see why we even bother with them.

We use a early strategy, called “Time Sharing” to illustrate the process management associated with handling interrupts and direct memory access.

In the Time Sharing model, we have
1. A single computer with its CPU, memory, and sharable I/O resources,
2. A number of computer terminals attached to the CPU, and
3. A number of users, each of whom wants to use the computer.

In order to share this expensive computer more fairly, we establish two rules.

1. Each user process is allocated a “time slice”, during which it can be run. At the end of this time, it must give up the CPU, go to the “back of the line” and await its turn for another time slice.

2. When a process is blocked and waiting on completion of either input or output, it must give up the CPU and cannot run until the I/O has been completed.

With this convention, each user typically thinks he or she is the only one using the computer. Thus the computer is “time shared”.
The Classic Process Diagram

Here is the standard process state diagram associated with modern operating systems.

![Process Diagram]

When a process (think “user program”) executes an I/O trap instruction (remember that it cannot execute the I/O directly), the O/S suspends its operation & starts I/O on its behalf.

When the I/O is complete, the O/S marks the process as “ready to run”. It will be assigned to the CPU when it next becomes available.
### The Three “Actors” for Input

<table>
<thead>
<tr>
<th>User Program</th>
<th>Operating System</th>
<th>Input Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Block the process</td>
<td></td>
</tr>
<tr>
<td>(Is blocked)</td>
<td>Reset the input status register</td>
<td>Status = 0</td>
</tr>
<tr>
<td></td>
<td>Enable the device interrupt</td>
<td>Interrupt is enabled</td>
</tr>
<tr>
<td></td>
<td>Command the input</td>
<td>Input begins</td>
</tr>
<tr>
<td></td>
<td>Dispatch another process</td>
<td>Input is complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Raise interrupt</td>
</tr>
<tr>
<td></td>
<td>Acknowledge interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Input (place data into AC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Place data into buffer for process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mark the process as ready to run</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Copy from buffer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resume processing</td>
<td>Obviously, there is more to it than this.</td>
</tr>
</tbody>
</table>
Another Diversion: Volatile Data

I/O processing introduces challenges to the designer of an optimizing compiler. Consider the following code, based on the busy wait example discussed above.

```
Device_Status = 0
Call Other_Process      // Device_Status is local
If (1 == Device_Status) Then
{ Do something }
End If
```

Note that the value of `Device_Status` is not changed in the code.

It is reasonable to think that the body of the If statement cannot be executed. An optimizing compiler might eliminate this as "dead code".

However what appears to be a simple variable is not so simple.

The label `Device_Status` refers to a device register, the contents of which are changed by the device, not by the program.

In the C/C++ world, this label would refer to volatile data, which can be validly changed by something other than the program.
What is DMA?

Remember that Main Memory is accessed through two registers and some control signals

- **MAR** Memory Address Register
- **MBR** Memory Buffer Register (holds the data)
- **READ** and **WRITE** If one is true, the memory is either written or read. If both are true, only one action is performed.

The CPU normally issues these signals. This holds for both program controlled I/O and interrupt driven I/O. In DMA, the device controller issues these signals.

A DMA controller is one that can directly manipulate memory.

But suppose a fast disk wants to transfer a 512-byte block of data to memory. It would not be efficient to have 512 interrupts.

Scenario:
1. The disk raises an interrupt and the CPU responds.
2. The device driver sends a start address and byte count to the disk controller, which connects the disk to the bus.
3. The disk transfers its block of data directly to memory.
4. The disk again raises an interrupt when it is complete.
Structure of an Interrupt Handler

I/O devices and their controllers fall into three major classes.

1. Program Controlled
2. Interrupt Driven
3. Direct Memory Access

Of these classes, only the latter two can generate interrupts. Here we focus on how the CPU processes the interrupts associated with such devices.

This part focuses on what I call the “Interrupt Controller Hub”.

This hub processes interrupts from multiple devices and sends a single INT signal to the CPU when an interrupt is recognized.

The CPU sends a single ACK signal back to the hub, which sends it to the appropriate device.
Interrupt Priority and CPU Priority

We follow the design of the PDP–11, developed by the Digital Equipment Corporation (now defunct) in discussing an interrupt structure.

We begin with the idea of a CPU execution priority. This is specified by a 3–bit number in the program status register (PSR). Here is the structure that we shall use.

<table>
<thead>
<tr>
<th>Bits</th>
<th>15 – 8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>Other Uses</td>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
<td>I</td>
<td>CPU Priority</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We postulate a 16–bit program status register with the following bits:

- **N, Z, V, & C** Status of the previous arithmetic operation
  (Always included in the PSR, so we use these too)

- **I** Interrupts enabled.
  When I = 0, the CPU ignores any interrupt.

- **Priority** A 3–bit unsigned integer representing the CPU execution priority.
The I Bit and CPU Execution Priority

These four bits are used in processing interrupts.

Disabling Interrupts (I = 0)
This should be done very seldom. Only the Operating System can set the I bit.

There are certain times in processing an interrupt during which another interrupt cannot be processed. During these short times, the CPU sets I = 0.

CPU Priority
Normal programming practice allows for multiple interrupt priorities and nested interrupts. A high priority device, such as a disk, can take precedence over the processing of an interrupt for a low priority device, such as a keyboard.

To manage devices at various priorities, each interrupt is processed as follows.
1. A device interrupts with priority K.
2. The CPU sets I = 0 and saves various registers.
3. The CPU sets its priority to K (the same number), sets I = 1, and then begins execution of the interrupt handler.

Devices with higher priorities can now interrupt and have their interrupts handled.
More on CPU Priority

We follow the PDP–11 convention.

Priority = 0  All user programs execute at this level.

Priority = 1, 2, 3  Various Operating System Utilities operate at these levels. We generally ignore these levels.

Priority = 4, 5, 6, 7  Interrupt handlers operate at these levels.

The CPU will acknowledge and process an interrupt only if the priority of the interrupting device is higher than the CPU execution priority.

For this reason, almost all interrupt handlers are written to execute with a CPU priority exactly equal to the device priority.

The convention is that all hardware devices are assigned one of four interrupt levels: 4, 5, 6, and 7.

Priority 4  is the lowest hardware priority, reserved for the keyboard, etc.

Priority 7  is the highest hardware priority, reserved for disks, etc.
Vectored Interrupts

How does the CPU identify the device that asserted the interrupt and begin execution of its interrupt handler? The basics of interrupt handling:

1. The device sends its “vector”, which is an address of a data structure. This used to be an address in low memory, say addresses 0 – 1023.

2. The data structure at the specified address contains the following.
   a) The address of the interrupt handler associated with the device.
   b) The CPU execution priority for the interrupt handler.

The interrupting device basically identifies itself by specifying the program to handle its interrupt.

The interrupt handling sequence can be elaborated.

1. Clear the Interrupt Enabled bit (set I = 0) to block other interrupts.
2. Store the essential registers, so that the user program can be restarted later.
3. Load the PSR with the execution priority and load the PC with the address.
4. Set I = 1 to allow nested interrupts and start execution of the handler.
The IVT (Interrupt Vector Table)

The IVT (Interrupt Vector Table) is a data structure managed by the Operating System. The IVT associates the interrupt with the software to handle it.

In the original MS–DOS design, the base address of the IVT was 0, so that the 8–bit interrupt vector (multiplied by 4) was the address.

Newer MS–Windows designs call for the IVT to be located anywhere in memory.

Structures Stored in the IVT
Interrupt Lines and Assertion Levels

The structure of the interrupt lines on a typical computer is as follows:

We have four interrupt lines, one for each of the four priority levels. Each is paired with an acknowledge line for the same priority.

Interrupts are **asserted low**; that is, the signal goes to 0 when the device interrupts.

Acknowledgements are **asserted high**; that is, the signal goes to logic 1 to acknowledge.

Only the highest priority interrupt is acknowledged. If both Int 7 and Int 5 are asserted at the same time, only ACK 7 is asserted. Int 5 continues to be asserted.

When the interrupt at level 7 is handled, Int 5 can be acknowledged if there are no higher level interrupts. Supposing that, ACK 5 is asserted.
Mechanism for Asserting an Interrupt

Each interrupt line is attached to a “pull down” resistor.

When the device asserts an interrupt, it sets its Interrupt Flip–Flop. Thus $Q = 1$. This enables the tri–state, which becomes a closed switch with very low resistance.

With the tri–state enabled, all the voltage drop is across the resistor so that the voltage on the Interrupt Line becomes 0. The interrupt is asserted.

With the tri–state disabled, it becomes an open switch, a line with very high resistance. All the voltage drop is across the tri–state, so the Interrupt Line stays at voltage.
Multiple Devices on One Line

By design, Interrupts are active low in order to facilitate attaching multiple interrupting devices on a single interrupt line.

Here we see four devices attached to a single line.

If no device is interrupting, we have Int = Logic 1

If any one device is interrupting, we have Int = Logic 0. The interrupt is asserted.

If more than one device is interrupting, we still have Int = Logic 0. The devices cannot interfere with each other.
Daisy Chaining

In daisy chaining, the ACK is passed from device to device until it hits a device that has asserted an interrupt.

If the I/O device has not asserted an interrupt, it passes the ACK to the next “downstream” I/O device.

Priority rank is by physical proximity to the CPU.
Computer Networks: Overview and I/O Considerations

Overview of the Internet

Packets and Connectionless Networks

Three protocols: IP, UDP, and TCP

The NIC (Network Interface Card) as a DMA Device

Interaction of IP with the Operating System and Application Programs

Sources:

1. The Essentials of Computer Organization and Architecture

2. Computer Networks and Internets with Internet Applications

3. Internetworking with TCP/IP: Volume II (Design, Implementation, & Internals)
An Overview of the Internet

The global Internet is best seen as a mechanism that allows computers to communicate. The Internet is a collection of interconnected networks, each with its own protocol. It provides the illusion of a single network, but has considerable internal structure.
What Is Connected to the Internet?

We consider our computers to be connected to the Internet. Technically, it is the NIC (Network Interface Card) that is connected.

The Network Interface Card is an Input / Output device attached to the computer. It communicates with the computer using Direct Memory Access (DMA).

The physical network address, called MAC address (for Media Access Control) address, is a 48–bit address that identifies the NIC, not the computer.
Attaching a Computer to a Network

This is a typical attachment that uses the original (Thicknet) wiring.

A number of devices in a single room would be connected through a multiplexor to the cable (called an “Ethernet cable”, after its protocol) through a transceiver (called an “AUI” for “Attachment Unit Interface”)

The AUI was typically placed above a false ceiling, making it hard to locate and repair a malfunctioning unit.
Attaching a Computer to a Network (Part 2)

Current technology for attaching computers to a network uses a technology called **twisted pair wiring**. Names for this wiring include “10BaseT” and “100BaseT”.

The twisted pair cable attaches to the NIC through a **RJ–45 connector**.

The **hub** connects these computers to the larger network.

The mathematicians in the class will note that this is a star topology and not the expected ring topology, supposedly used for networks. It is treated as if it were a ring.
Typical Group Setup for Twisted–Pair Networks

Here is a typical group setup in which a number of computers are connected to the Internet through a hub.

The hub is often contained in a "network closet", which is a small locked room. This facilitates maintaining and securing the network assets.

The only part of the network above the ceiling is a collection of twisted–pair wires, which are usually quite reliable. In any case they are easy and cheap to replace.
Connection–Oriented vs. Datagram Networks

The Internet is a bit unusual in that it is not a connection–oriented network.

The U.S. telephone network is the best example of a connection–oriented network.

One dials a phone number, establishes a connection, and then keeps that connection open for the duration of the conversation. This might include a lot of time “on hold”.

Most networks break the messages and other data being passed into a number of packets, also called “datagrams”. Each packet can be routed independently from the source to the destination, leading to a great flexibility in the network.

At the physical level, a packet is embedded in an Ethernet frame as the frame payload. It has a 48–bit MAC address for the source node and destination node, a frame type, and a CRC for error detection. For IP version 4, the frame type is 0x0800.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Dest. Address</th>
<th>Source Address Type</th>
<th>Data In Frame</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
<td>6 2</td>
<td>46 - 1500</td>
<td>4</td>
</tr>
</tbody>
</table>

| Header | Payload |
The IP (Internet Protocol)

The current version is IP, version 4. The next version will be called IP, Version 6. It was originally called “IP – the Next Generation”. Guess why?

IP is responsible for communication between computers. It does not connect applications on those computers, as do the next few protocols.

An IP datagram will be placed within an Ethernet frame for transmission over the net.

![Diagram of IP header and data area within Ethernet frame]

Remember that the frame header has a flag set to 0x0800 for encapsulated IP.

The IP header contains a protocol flag that indicates the datagram type.

Typical values are

1  ICMP  (Internet Control Message Protocol)
6  TCP   (Transmission Control Protocol)
17 UDP  (User Datagram Protocol)
End–to–End Protocols

IP connects computers, but cannot determine the true end points of a connection, which must be two applications.

The two major end–to–end protocols are TCP and UDP.

The main difference between the two has to do with the reliability of the communication.

TCP is characterized by:

1. **Reliable transport.** TCP guarantees that the data sent across the connection will be delivered exactly as sent, with no data missing or out–of–order. This implies retransmission of lost packets and rearrangement of those that are delivered out of order.

2. **Connection Oriented** Although it uses datagrams, TCP establishes a connection between two applications that persists for the duration of the session.

UDP is characterized as a “best effort” protocol, with no guarantees of packet delivery.

Consider transmission of music over the Internet. In this case, efficiency is important and occasional dropped packets are only a minor nuisance.
More on the NIC

As noted above, the NIC is the unit that is actually attached to the network. For the attached computer, it acts as a DMA Input / Output device. Each NIC has a unique 48–bit MAC (Media Access Control) physical address. These are designed to be globally unique, and are administered by the IEEE.

In \textbf{standard mode}, a NIC scans frames being sent on the network but stores only those frames with its MAC address as a destination.

In promiscuous mode, it will process any frame on the network. This is used by devices, such as \textbf{network sniffers}, used to diagnose networks or spy on them.
The NIC Takes a Message

When the NIC has copied an entire frame from the network, the I/O sequence follows the standard DMA process.

1. The NIC asserts an interrupt to the CPU.
2. The CPU sends an ACK to the NIC.
3. The NIC places its vector on the I/O data lines.
4. The interrupt handler uses the vector to locate and start the interrupt handler appropriate for the NIC.
5. The interrupt handler sends the NIC a byte count (usually the Ethernet frame size) and a starting physical address in memory. It then commands the NIC to start data transfer and assert an interrupt when the input has been finished.
6. At the end of DMA, possibly due to an error, the NIC again interrupts the CPU. This interrupt is processed much as above.
7. The operating system then examines the frame to determine the type of service to be associated with the frame.
The O/S Takes a Message

In response to the “DMA done” interrupt issued by the NIC, the operating system schedules the appropriate utility program to examine the frame.

This utility extracts the frame type, and determines that it is 0x0800.

The frame type indicates that the payload is an IP version 4 datagram.

The O/S interrupt handler cannot call IP directly. It places the payload into a dedicated message queue and then uses a message passing primitive to signal IP.

Note that the term “IP” is used in two contexts
1. The protocol dictating how the computers will communicate, and
2. The actual software that implements that protocol.
IP Takes a Message

When the IP program processes the datagram, it extracts the service type from the IP header. The IP header has quite a few fields, shown in this figure.

![IP Header Diagram]

The version number will be either 4 or 6 for the next little while.

The protocol type is stored in the field called “TYPE”. It has one of several values:

1    ICMP
6    TCP
17   UDP
Protocol Ports

Each of UDP and TCP is an end–to–end protocol; that is, it connects two application programs (presumably of the same type).

The standard way of identifying an application is known as a protocol port. For example, the application to handle HTTP will associate itself with protocol port 80. Thus, a datagram will come to the computer with the message “I am a HTTP packet, please route me to whatever application processes HTTP”.

All protocol ports are defined to be 16–bit unsigned integers; 0 to 65,535 inclusive.

The first 1,024 ports (numbered 0 through 1,023) are defined as well–known protocol ports, defined by a standard called RFC 793.

Some of the well–known protocol port numbers are as follows.

<table>
<thead>
<tr>
<th>Port</th>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>FTP</td>
<td>File Transport Protocol Data</td>
</tr>
<tr>
<td>21</td>
<td>FTP</td>
<td>File Transport Protocol Control</td>
</tr>
<tr>
<td>23</td>
<td>Telnet</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>SMTP</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>53</td>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>80</td>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
</tbody>
</table>
TCP Takes A Message

Suppose that IP determines that the datagram contains a TCP packet.

TCP calls the packet a **segment**.

Here is the format of a TCP header.

![TCP Header Diagram]

TCP processes this header, extracting the destination port and determining the application to receive the data contained in the segment.

TCP then places the segment in the appropriate queue and uses an operating system primitive called a “semaphore” to signal the application that it has data.