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 MARIE instruction
          J = J + 1    // Again pseudocode
          Input
          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 (Input and Output for the MARIE) 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.

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.

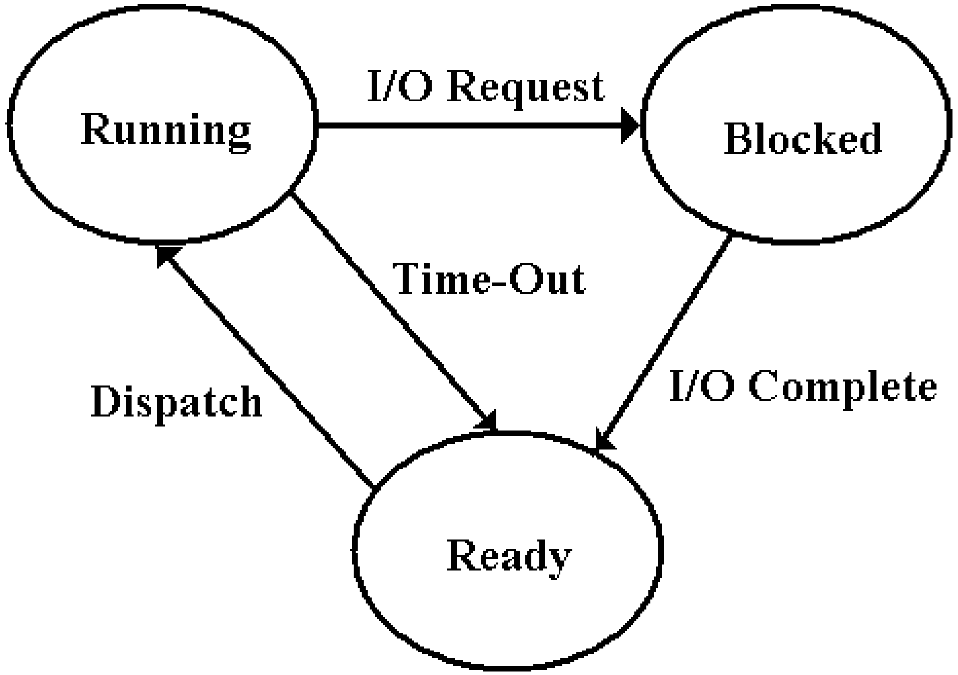
1. Each user process is allocated a **“time slice”**, during which it can be run.
   At the end of this time, it must give up the CPU, go to the “back of the line” and await its turn for another time slice.

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 Classis Process Diagram

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



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

User Program	**Operating System**	**Input Device**
Input		
(Is blocked)	Block the process	
	Reset the input status register	Status = 0
	Enable the device interrupt	Interrupt is enabled
	Command the input	Input begins
	Dispatch another process	
		Input is complete
		Raise interrupt
	Acknowledge interrupt	
	Input (place data into AC)	
	Place data into buffer for process	
	Mark the process as ready to run	
Copy from buffer		
Resume processing		

Obviously, there is more to it than this.

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.