DIRECT MEMORY ACCESS (DMA).

Is there another way of performing the I/O activity without wasting the processor's time? Computer systems also employ a similar technique. It is called *direct memory access* (DMA). In DMA, the processor gives the command such as "transfer 10 KB to I/O port 125" and the DMA performs the transfer without bothering the processor. Once the operation is complete, the processor is notified. This notification is done by using an interrupt mechanism. We use DMA to transfer bulk data, not for single word transfers. A special DMA controller is used to direct the DMA transfer operations.

DMA is implemented by using a DMA controller. The DMA controller acts as a slave to the processor and receives data transfer instructions from the processor. Figure 31 shows the difference between programmed I/O and DMA transfer. In programmed I/O, the system bus is used twice as shown in Figure 31a. The DMA transfer not only relieves the processor from the data transfer chore but also makes the transfer process more efficient by transferring data directly from the I/O device to memory.

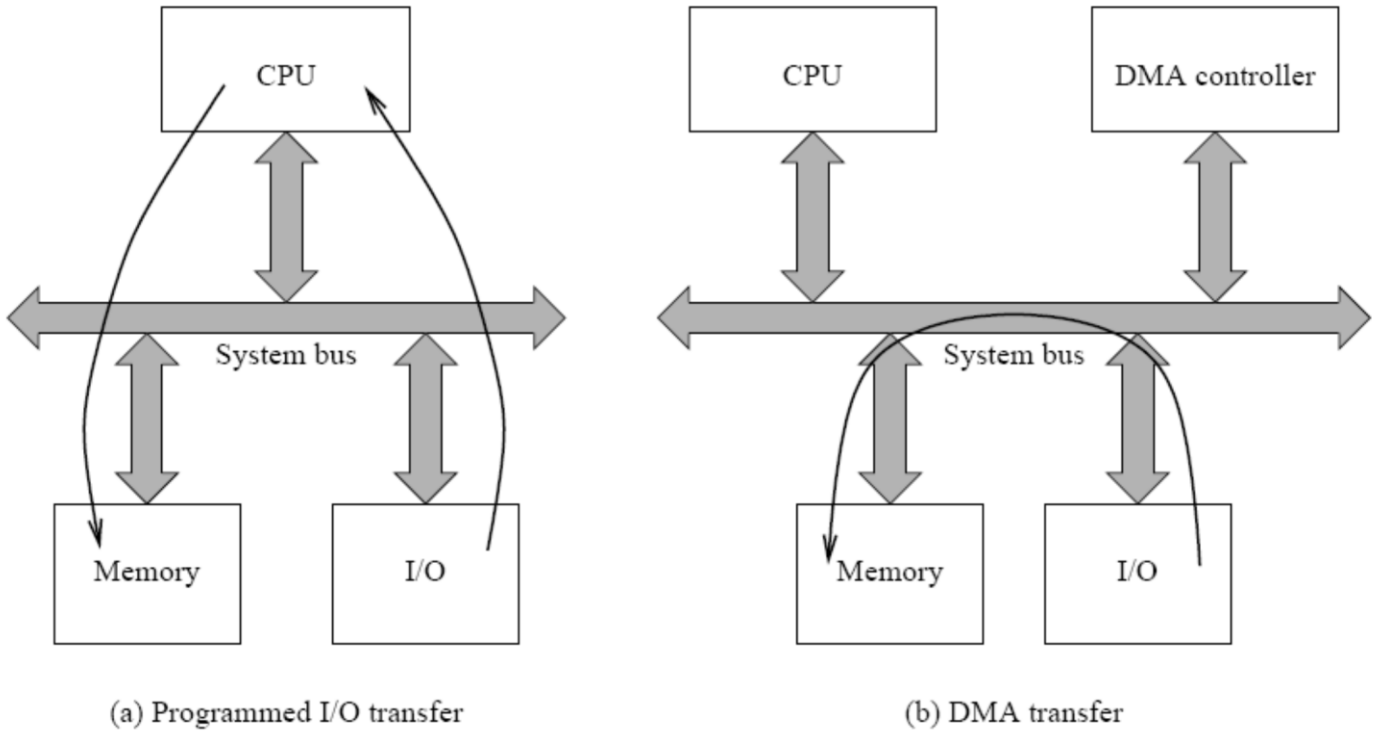

Figure 31 Data transfer from an I/O device to system memory: (a) in programmed I/O, data are read by the processor and then written to the memory; (b) in DMA transfer, the DMA controller generates the control signals to transfer data directly between the I/O device and memory.

INTERRUPT-DRIVEN I/O.

It is often necessary to have the normal flow of a program interrupted, for example, to react to abnormal events, such as power failure. An interrupt can also be used to acknowledge the completion of a particular course of action, such as a printer indicating to the computer that it has completed printing the character(s) in its input register and that it is ready to receive other character(s). The instruction sets of modern CPUs often include instruction(s) that mimic the actions of the hardware interrupts.

When the CPU is interrupted, it is required to discontinue its current activity, attend to the interrupting condition (serve the interrupt), and then resume its activity from wherever it stopped. Discontinuity of the processor's current activity requires finishing executing the current instruction, saving the processor, and transferring control (jump) to what is called the interrupt service routine (ISR). The service offered to an interrupt will depend on the source of the interrupt. For example, if the interrupt is due to power failure, then the action taken will be to save the values of all processor registers and pointers such that resumption of correct operation can be guaranteed upon power return. In the case of an I/O interrupt, serving an interrupt means to perform the required data transfer. Upon finishing serving an interrupt, the processor should restore the original status by popping the relevant values from the stack. Once the processor returns to the normal state, it can enable sources of interrupt again. A flowchart for interrupt driven I/O is shown in Figure 32.

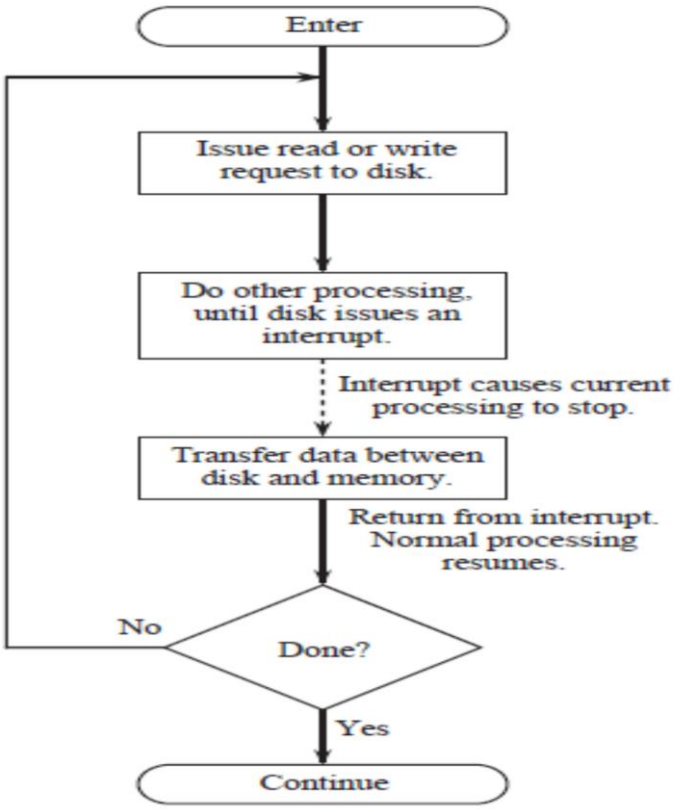

Figure 32 Interrupt driven I/O flowchart for a disk transfer.

Data Transmission.

We can interface I/O devices in one of two basic ways: using a parallel or serial interface. In parallel transmission, several bits are transmitted on parallel wires (***n* bits are transmitted on *n* wires in parallel**) as shown in Figure 33. On the other hand, in the serial transmission, only a single wire is used for data transmission (**one bit at a time**).

Parallel transmission is faster: we can send *n* bits at a time on an *n*-bit wide parallel interface. That also means it is expensive compared to the serial interface. A further problem with the parallel interface, particularly at high data transfer rates, is that skew (some bits arrive early and out of sync with the rest) on the parallel data lines may introduce error prone delivery of data. Because of these reasons, the parallel interface is usually limited to small distances. In contrast, the serial interface is cheaper and does not cause the data skew problems.

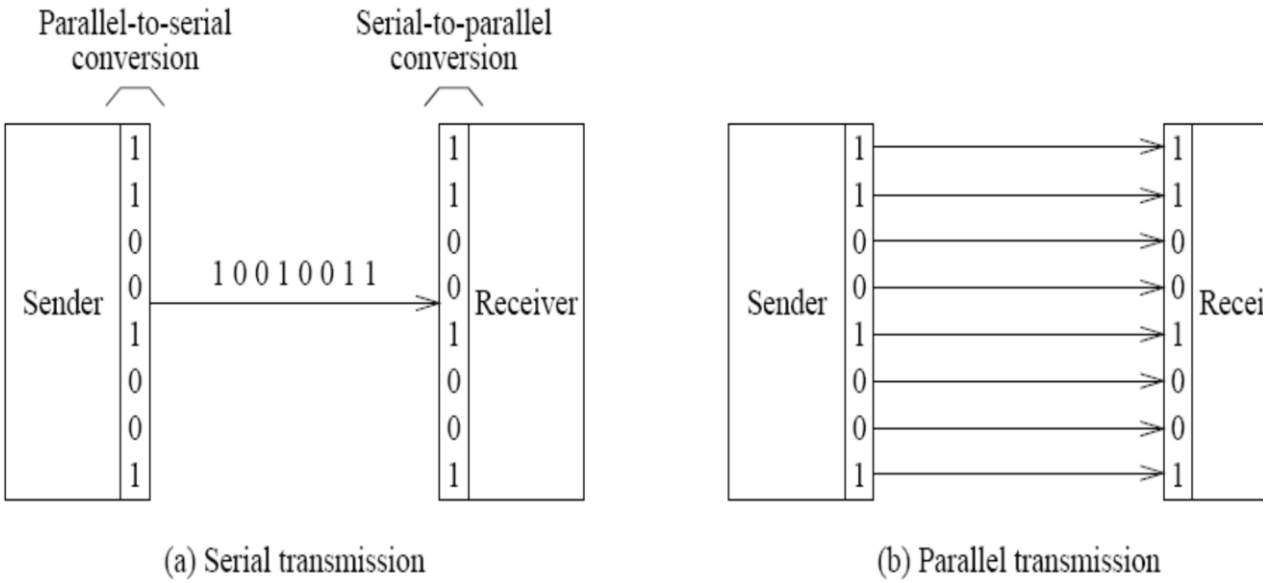

Figure 33 Two basic modes of data transmission.

To detect simple errors during data transmission, a single parity bit is added to the 7-bit data. Assume that we are using even parity encoding. That is, every 8-bit character code transmitted will contain an even number of 1 bits. Then, the receiver can count the number of 1s in each received byte and flag transmission error if the byte contains an odd number of 1 bits. Such a simple encoding scheme can detect single bit errors (in fact, it can detect an odd number of single bit errors). To encode, the parity bit is set or cleared depending on whether the remaining 7 bits contain an odd or even number of 1s, respectively. For example, if we are transmitting character A, whose 7-bit ASCII representation is 41H, we set the parity bit to 0 so that there is an even number of 1s.