Contents

Issues with I/O

An OS incorporates some I/O event recognition mechanism, I/O handling mechanisms which may be polling, or a programmed I/O data transfer, or an interrupt mechanism, or even may be a direct memory access (DMA) with cycle stealing.

The unit of data transfer may either be one character at a time or a block of characters.

It may require to set up a procedure or a protocol. This is particularly the case when a machine-to-machine or a process-to-process communication is required. Additionally, in these cases, we need to account for the kind of errors that may occur.

We also need procedure to recover when such an error occurs and to find ways to ensure the security and protection of information when it is in transit. Yet another important consideration in protection arises when systems have to share devices like printers.

Modes of data transfer

There are obviously various issues that arise from the need to support a wide range of devices. To meet these varied requirements, a few well understood modalities have evolved over time.

The basic idea is to select ***a mode of communication*** taking device characteristics into account or a need to synchronise with some event, or to just have a simple strategy to ensure a reliable assured I/O.

There are three main techniques for performing I/O:

1. Programmed I/O
2. Interrupt Driven I/O and
3. Direct Memory Access I/O

Polling can also be used for I/O so this is described here also.

1. Programmed I/O Data Mode

In the simplest method for performing I/O. In this mode of communication, the processor issues an I/O command, on behalf of a process, to an I/O module; that process then waits for the operation to be completed before proceeding. To that extent the I/O is assured to complete before anything else happens.

An I/O module is connected to a pair of I/O registers in the CPU via a bus.

> The I/O data register serves the same role in the real CPU as the input and output baskets served in the Little Man Computer.

Alternatively, one might view the I/O baskets as buffers, holding multiple inputs or outputs, with the I/O data register as the interface between the CPU and the buffer.

> The I/O operation is similar to that of the Little Man Computer. Input from the peripheral device is transferred from the I/O module or buffer for that peripheral device one word at a time to the I/O data register and from there to an accumulator register under program control, just as occurred in the Little Man Computer.
>
> Similarly, individual words of output data pass from an accumulator register to the I/O data register where they can be read by the appropriate I/O module, again under program control. Each instruction produces a single input or output.

Since each device must be recognised individually, address information must be sent with the I/O instruction. The address field of the I/O instruction can be used for this purpose. An I/O address register in the CPU holds the address for transfer to the bus. Each I/O module will have

an identification address that will allow it to identify I/O instructions addressed to it and to ignore other I/O not intended for it (In practice, it is most likely that there will be multiple devices connected to the CPU).

The I/O data and address registers work similarly to the memory address register (MAR) and memory data register (MDR). In fact, in some systems, they may even be connected to the same bus. The CPU places a control signal on the bus to indicate whether the transfer is I/O or memory

In this mode an I/O instruction is issued to an I/O device and the program executes in "busy-waiting" (idling) mode till the I/O is completed. During the *busy-wait* period the processor is continually interrogating to check if the device has completed IO.

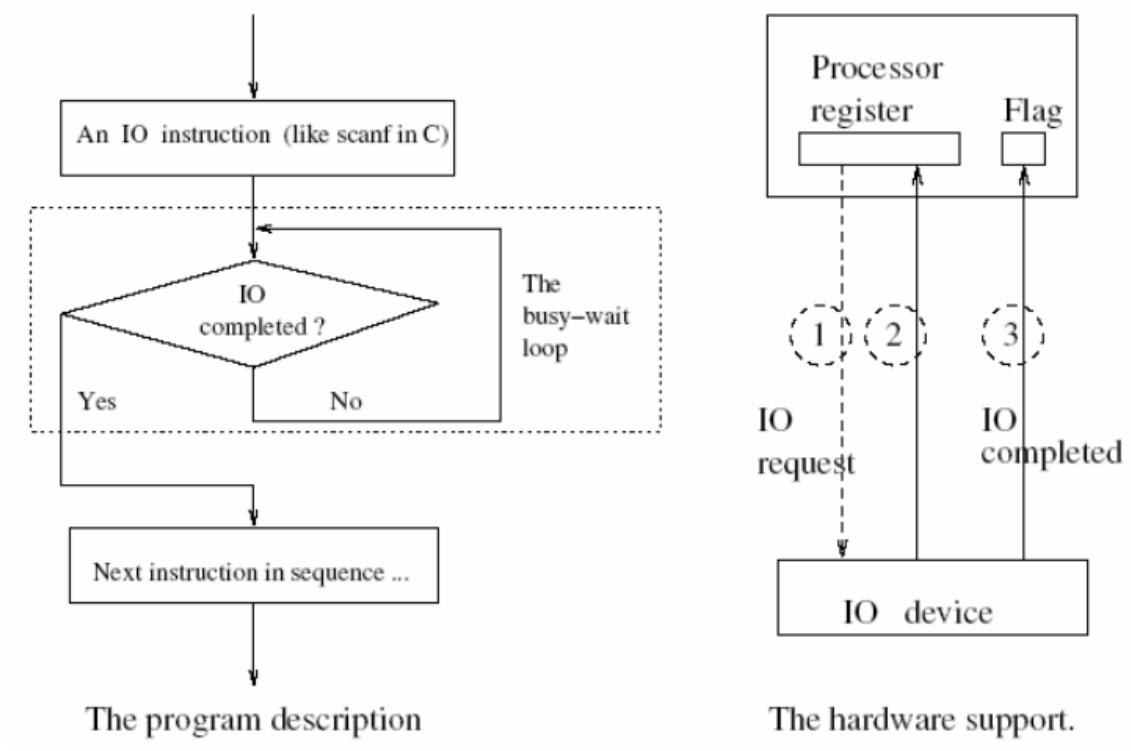

Figure 1: Programmed mode of IO

The steps for the hardware support are:

1) First the processor issues an I/O request,
2) followed by device putting a data in picks up from a register and
3) Finally the flag (which is being interrogated) is set.

The device either puts a data in (for input)/picks up from (output) the register. When the I/O is accomplished it signals the processor through the flag. During the *busy-wait* period the processor is busy checking the flag. However, the processor is idling from the point of view of doing anything useful.

Programmed I/O is obviously slow, since a full instruction fetch-execute cycle must be performed for each and every I/O data word to be transferred. Programmed I/O is used today primarily with keyboards, with occasional application to other simple character based data transfers, such as the transmission of commands through a network I/O module or modem. These operations are slow compared with the computer, with small quantities of data that can be handled one character at a time.

A simple way of moving the data between the peripheral device and main memory is to use the main processor to perform load or store operations for each byte or word of data to be moved. The processor must wait for the peripheral to be ready before transferring each byte or word, which can be done by polling a status register or by handling a "ready" interrupt from the device. This is all Programmed I/O.

If the device produces or consumes a small amount of data at low data rates, this may be the best way of managing the data transfer in and out of main memory;

Polling

With polling, the system cross-examines each device in turn to determine if it is ready to communicate. If it is ready, communication is initiated and subsequently the process continues again to interrogate in the same sequence. This is just like a round-robin strategy. Each I/O

device gets an opportunity to establish Communication in turn. No device has a particular advantage (like say a priority) over other devices.

Polling is quite commonly used by systems to interrogate ports on a network. Polling may also be scheduled to interrogate at some pre-assigned time intervals. It should be noted that most daemon software operate in polling mode. Essentially, they use a while true loop as shown:

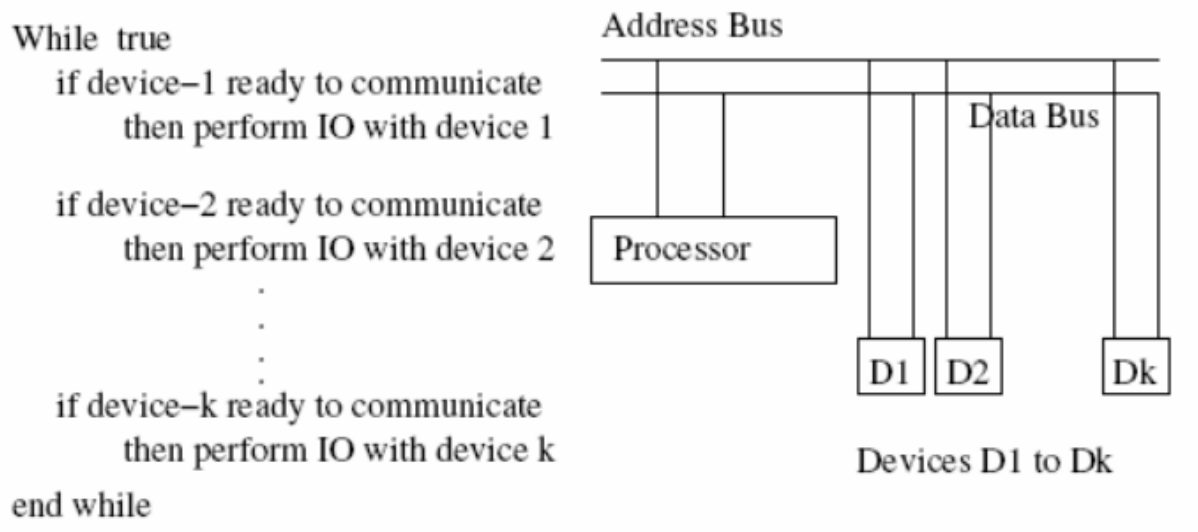

Figure 2: Polling Mode of IO

In hardware, this may typically translate to the following protocol:

1. Assign a distinct address to each device connected to a bus.
2. The bus controller scans through the addresses in sequence to find which device wishes to establish a communication.
3. Allow the device that is ready to communicate to leave its data on the register.
4. The I/O is accomplished.
5. Move to interrogate the next device address in sequence to check if it is ready to communicate.

Polling may also be used within an interrupt service mode to identify the device which may have raised an interrupt

2. Interrupt I/O Mode

There are many circumstances under which it is important to interrupt the normal flow of a program in the computer to react to special events. For example: an unexpected user command from the keyboard or other external input, an abnormal situation, such as a power failure, that requires immediate attention from the computer, an attempt to execute an illegal instruction, a request for service from a network controller, or the completion of an I/O task initiated by the program: all of these suggest that it is necessary to include some means to allow the computer to take special actions when required.

Interrupt capabilities are also used to make it possible to timeshare the CPU between several different programs or program segments at once

Modern computers provide interrupt capability by providing one or more special control lines to the central processor known as interrupt lines. For example, Intels APIC family of interrupt controllers support a programming interface for up to 255 physical hardware IRQ lines per APIC, labelled IRQ0, IRQ1, IRQ2, and so on (IRQ stands for Interrupt ReQuest.)

The messages sent to the computer on these lines are known as Interrupts. The presence of a message on an interrupt line will cause the computer to suspend the program being executed and jump to a special interrupt processing program

EXAMPLE

In a large, multiuser system there may be hundreds of keyboards being used with the computer at any given time. Since any of these keyboards could generate input to the computer at any time, it is necessary that the computer be aware of any key that is struck from any keyboard in use. This process must take place quickly, before another key is struck on the same keyboard, to prevent data loss from occurring when the second input is generated.

Theoretically, though impractically, it would be possible for the computer to perform this task by checking each keyboard for input in rotation, at frequent intervals. This technique is known as **polling**. The interval would have to be shorter than the time during which a fast typist could hit another key. Since there may be hundreds of keyboards in use, this technique may result in a polling rate of thousands of samples per second. Most of these samples will not result in new data; therefore, the computer time spent in polling is largely wasted.

This is a situation for which the concept of the interrupt is well suited. The goal is achieved more productively by allowing the keyboard to notify the CPU by using an interrupt when it has input. When a key is struck on any keyboard, it causes the interrupt line to be activated, so that the CPU knows that an I/O device connected to the interrupt line requires action. Interrupts satisfy the requirement for external input controls, and also provide the desirable feature of freeing the CPU from waiting for events to occur.

To begin with, a program may initiate I/O request and advance without suspending its operation. At the time when the device is actually ready to establish an IO, the device raises an interrupt to seek communication.

Immediately the program execution is suspended temporarily and current state of the process is stored.

The control is passed on to an interrupt service routine (ISR), which may be specific to the device to perform the desired input. Subsequently, the suspended process context is restored to resume the program from the point of its suspension.

Interrupt processing may happen in the following contexts:

a) Internal Interrupt

The source of interrupt may be a memory resident process or a function from within the processor, and such an interrupt as an internal interrupt. For example, a processor malfunction, an attempt to divide by zero (called a ***trap***) or execute an illegal or non-existent instruction code all result in an ***internal interrupt***.

Internal interrupt may be caused by a timer as well. This may be because either the allocated processor time slice to a process has elapsed or for some reason the process needs to be pre-empted. Note that an RTOS may pre-empt a running process by using an interrupt to ensure that the stipulated response time required is met. This would also be a case of internal interrupt.

b) External Interrupt

If the source of interrupt in not internal, i.e. it is other than a process or processor related event then it is an external interrupt. This may be caused by a device which is seeking attention of a processor. As indicated earlier, a program may seek an I/O and issue an I/O command but proceed. After a while, the device from which I/O was sought is ready to communicate. In that case the device may raise an interrupt. This would be a case of an external interrupt.

c) Software Interrupt:

Most OSs offer two modes of operation, the ***user mode*** and the ***system mode***. Whenever a user program makes a *system call*, be it for I/O or a special service, the operation must have a transition from user mode to system mode. An interrupt is raised to effect this transition from user mode to system mode of operation.

How an interrupt is serviced

Suppose the system is executing an instruction at ***i*** in program ***P*** when interrupt signal has been raised. Let us also assume that we have an ISR which is to be initiated to service the interrupt.

The following steps describe how a typical interrupt service may happen.

- Suspend the current program P after executing instruction i.
- Store the address of instruction at i + 1 in P as the return address.

  This is the point at which program P shall resume its execution following the interrupt service. The return address is essentially the incremented program counter value.

- Execute a branch unconditionally to transfer control to the interrupt service instructions. The immediately following instruction cycle initiates the interrupt service routine.
- Typically the last instruction in the service routine executes a branch indirect from the location RESTORE. This restores the program counter to take the next instruction at P=i+1. Thus the suspended program P obtains the control of the processor again

3. Direct Memory Access(DMA)

As an alternative to (1) Programmed I/O and (2) Interrupt drive I/O described above, (3) Direct Memory Access (DMA) is a mode of data transfer in which the DMA unit is capable of mimicking the processor and taking over control of the system bus just like a processor. It needs to do this to transfer data to and from memory over the system bus.

The I/O is performed in large data blocks and these large blocks are transferred directly between an I/O device and memory. For example, the disks communicate in data blocks of sizes such as 512 bytes or 1024 bytes.

DMA works in conjunction with interrupts.

When a process initiates a DMA transfer (a read/write of a block of data), its execution is briefly suspended (***using an interrupt***) to set up the DMA control.

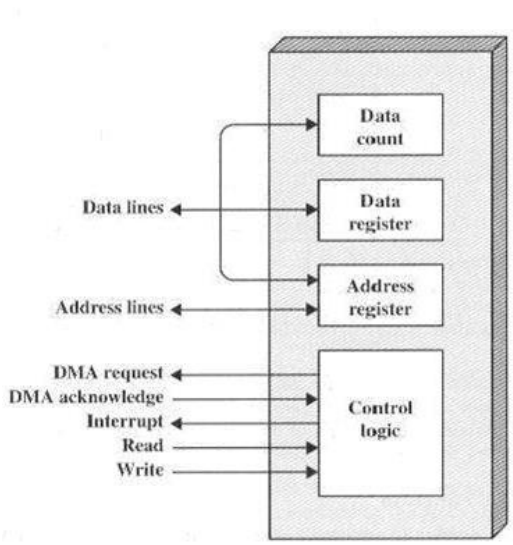

Figure 3: DMA Block Diagram

The DMA module, shown above, requires information on:

- Whether a read/write is requested (using the read or write control line between the processor and the DMA module)
- The address of the I/O device involved (communicated on the data lines)
- The starting location in memory to read from/write to (via the data lines and stored by the DMA module in its address register)

- The number of words to be read/written (again, via the data lines and stored by the DMA module in its data count register)

The device communicates with main memory stealing memory access cycles in competition with other devices and processor.

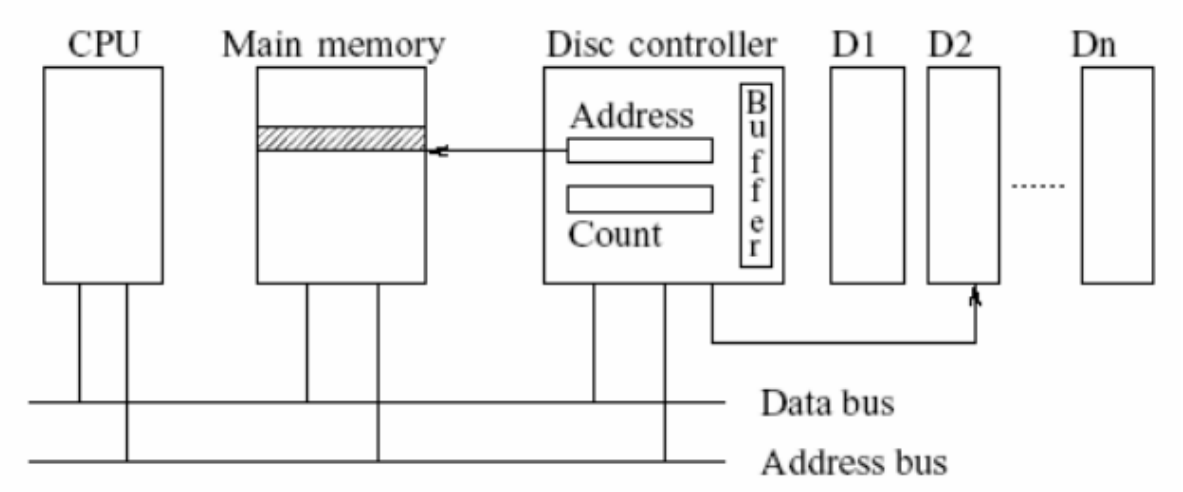

Figure 4: DMA Hardware Support

There is a disk controller to regulate communication from one or more disk drives. This controller essentially isolates individual devices from direct communication with the CPU or main memory.

The communication is regulated to: first happen between the device and the controller, and later between the controller and main memory or CPU if so needed.

Note that these devices communicate in blocks of bits or bytes as a data stream.

> Clearly, an un-buffered communication is infeasible via the data bus. The bus has its own timing control protocol. The bus cannot, and should not, be tied to device transfer bursts. The byte stream block needs to be stored in a ***buffer*** isolated from the communication to processor or main memory. This is precisely what the buffer in the disk controller accomplishes.

Once the controller buffer has the required data, then there starts competition for access to the bus between the CPU and main memory.

This transfer shall be completely independent of program control from the processor. So we can have a transfer of one block of data from the controller to main memory provided the controller has the address where data needs to be transferred and data count of the transfer required, without going through the processor.

This is the kind of information which initially needs to be set up in the controller address and count registers. Putting this information may be done under a program control as a part of DMA set up. The program that does it is usually the ***device controller*** (The device controller can then schedule the operations with much finer control).

We shall assume that we have a program P which needs to communicate with a device D and get a chunk of data D to be finally stored starting in address A.

This mechanism not only allows us to free the CPU to execute other commands, but it can also significantly improve the throughput of the memory transactions from the device to the memory and vice versa

As an alternative to Programmed I/O transfer is to set up a DMA transfer that gives the job of moving the data to a special-purpose device in the system. Once the processor has set up the transfer it can do something else while the transfer is in progress or wait to be notified when the transfer has finished.

The network-oriented traffic (between machines) may be handled in DMA mode. This is so because the network cards are often DMA enabled. Besides, the network traffic usually corresponds to getting information from a disk file at both the ends. Also, because network traffic is in bursts, i.e. there are short intervals of large data transfers. DMA is the most preferred mode of communication to support network traffic

A DMA descriptor is a data structure that contains all the information the hardware needs to execute its operations such as read or write. The descriptor is prepared by the OS in advance. Its location on the memory is known to the device. Once the hardware becomes available to execute the next DMA command, it reads the descriptor, executes the relevant command, advances to read the next descriptor, and so on until it reaches an empty descriptor.

Transferring Data from the Memory to the Device

Consider an example of a case in which the device reads data from the memory, as happens when a packet is sent to the Network Interface Card (NIC)

On driver registration, the driver allocates a set of descriptors, and, once there is data to transfer to the device:

- the driver chooses a descriptor belonging to the OS,
- updates it to point to the data buffer,
- writes in it the size of the buffer,
- marks the descriptor as belonging to the device, and
- interrupts the device to wake it up.
- The device reads the first descriptor, which belongs to it, and from that descriptor it reads the pointer and the size of the buffer to be read.

The device now knows the number of bytes to read and where to read from, and it start the transaction.

- After finishing the transaction, the device:
    - marks the descriptor as belonging to the OS,
    - advances to the next descriptor, and
    - interrupts the OS.

The OS detaches the buffer from the descriptor, leaving the descriptor free for the next DMA command

Most network cards are direct memory access (DMA) enabled to facilitate DMA mode of I/O with communication infrastructure.

Transferring Data from the Device to the Memory

When the device writes data to the memory, it interrupts the OS to announce that new data has arrived.

Then, the OS uses the interrupt handler to allocate a buffer and tells the hardware where to transfer its data.

After the device writes the data to the buffer and raises another interrupt, the OS wakes up a relevant process and passes the packet to it.

> A network card is a typical example of a device that transfers data asynchronously to the memory.

The OS:

- prepares the descriptor in advance,
- allocates a buffer,
- links it to the descriptor, and
- marks the descriptor belonging to the device.

If no packet arrives, the descriptor contains a link to an allocated buffer waiting for data to be written to it.

At the arrival of a packet from the network, the network card:

- reads the descriptor in order to identify the address of the buffer to write the data to,
- writes the data to the buffer, and
- raises an interrupt to the OS.
- Using the interrupt handler, the OS:
    - detaches the buffer from the descriptor,
    - allocates a new buffer,
    - links it to the descriptor, and
    - passes the packet written on the buffer to the network stack.

Now the descriptor, with a new allocated buffer, again belongs to the device and waits until a new packet arrives.

Summary of modes of data transfer

In all of the above modes of device communication, the OS makes it look as if we are doing a read or a write operation on a file.

We may have programmed I/O for synchronising information between processes or when speed is not critical. For instance, a process may be waiting for some critical input information required to advance the computation further. As an example of programmed IO, we may consider the PC architecture based on i386 CPU which has a notion of listening to an I/O port. Some architectures may even support polling a set of ports. The interrupt transfer is ideally suited for a small amount of critical information like a word, or a line i.e. no more than tens of bytes. DMA is great for network traffic.