MODULE 3

Input/ Output Organization: Accessing I/O Devices, Interrupts: Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Direct Memory Access

ACCESSING I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement as shown in figure 4.1. The bus enables all the devices connected to it to exchange information. It consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses.

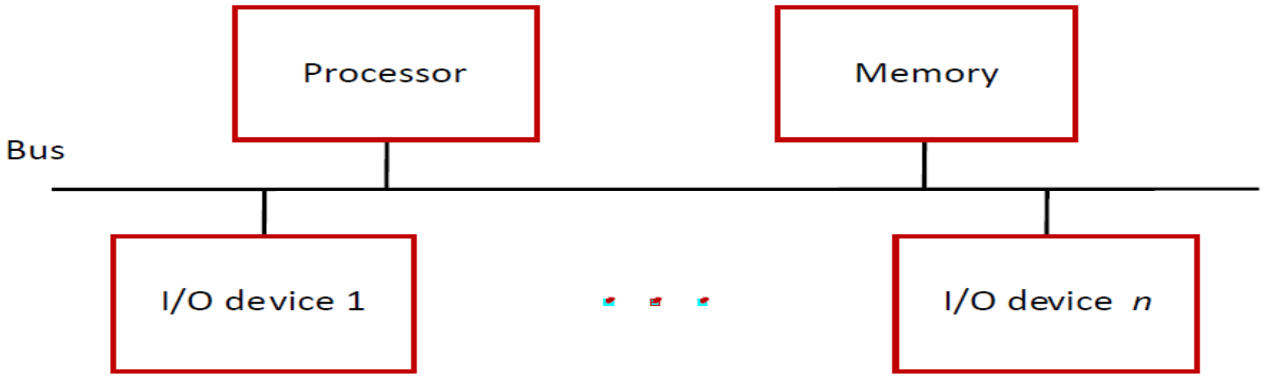


Figure 4.1 A single-bus structure.

To access an I/O device, the processor places the address on the address lines. The device recognizes the address, and responds to the control signals. The processor requests either a read or a write operation, and the requested data are transferred over the data lines.

There are 2 ways to deal with I/O-devices:

- Memory-mapped I/O &
- I/O-mapped I/O.

When I/O devices and the memory share the same address space, the arrangement is called *memory mapped I/O*. Any machine instruction that can access memory can be used to transfer data to or from an I/O device.

Example:

- Move DATAIN, R0; move the contents of location DATAIN to register R0.
- Move R0, DATAOUT; move the contents of register R0 to location DATAOUT.

When I/O devices and the memory have different address spaces, the arrangement is called *I/O mapped I/O*. Special In and Out instructions are used to perform I/O transfers. I/O devices may have to deal with fewer address lines. I/O address lines need not be physically separate from memory address lines.

Figure 4.2 illustrates the hardware required to connect an I/O device to the bus. I/O device is connected to the bus using an I/O interface circuit which has:

 Address decoder Control circuit Data and status registers.

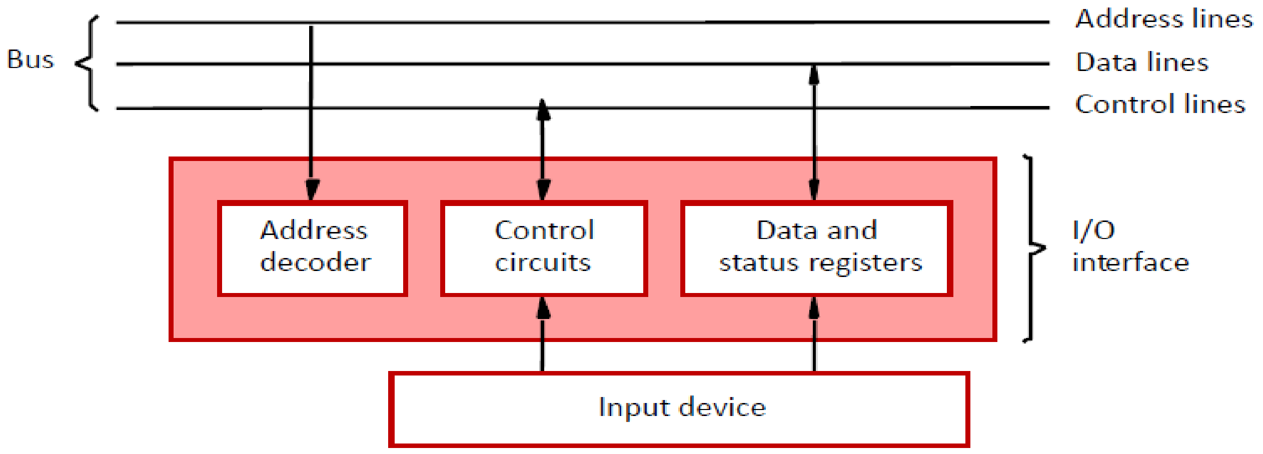


Figure 4.2 I/O interface for an input device.

Address decoder enables the device to recognize its address when this address appears on the address lines. Data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Data and status registers are connected to the data bus, and have unique addresses. I/O interface circuit coordinates I/O transfers.

The four registers shown in Figure 4.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. The KEN and DEN bits are in register CON TROL. Data from the keyboard are made available in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

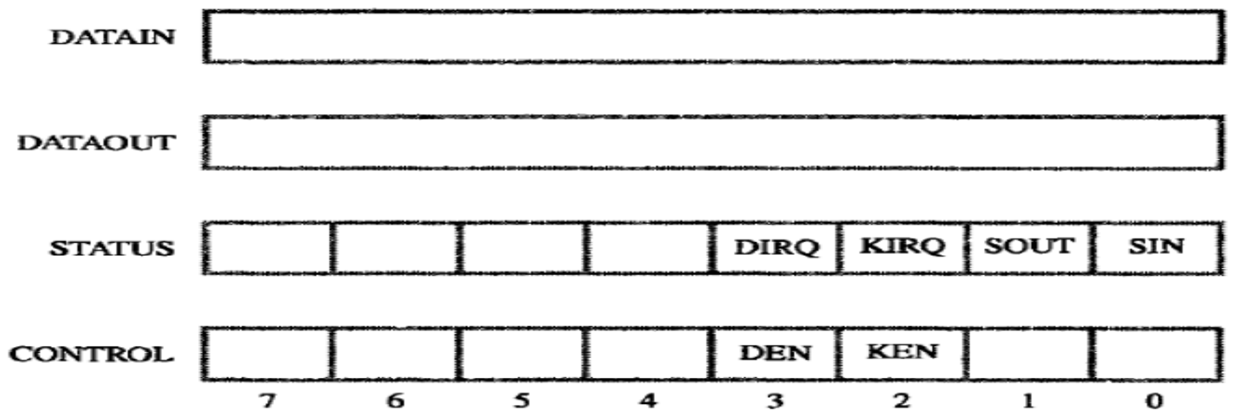


Figure 4.3 Registers in keyboard and display interfaces.

The program in figure 4.4, reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is echoed back to the display. Register R0 is used as a pointer to the memory buffer area.

```
        Move       #LINE,R0        Initialize memory pointer.
WAITK   TestBit    #0,STATUS       Test SIN.
        Branch=0   WAITK           Wait for character to be entered.
        Move       DATAIN,R1       Read character.
WAITD   TestBit    #1,STATUS       Test SOUT.
        Branch=0   WAITD           Wait for display to become ready.
        Move       R1,DATAOUT      Send character to display.
        Move       R1,(R0)+        Store charater and advance pointer.
        Compare    #$0D,R1         Check if Carriage Return.
        Branch≠0   WAITK           If not, get another character.
        Move       #$0A,DATAOUT    Otherwise, send Line Feed.
        Call       PROCESS         Call a subroutine to process the
                                     the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations. Each character is checked to see if it is the Carriage Return (CR) character, which has the ASCII code 0D (hex).

The mechanisms used for synchronizing data transfers between the processor and memory are:

- **Program controlled I/O:** Processor repeatedly monitors a status flag to achieve the necessary synchronization. Processor polls the I/O device. The main drawback of this is that the processor wastes time in checking status of device before actual data-transfer takes place.
- **Interrupts:** I/O-device initiates the action instead of the processor. I/O-device sends an INTR signal over bus whenever it is ready for a data-transfer operation. Like this, required synchronization is done between processor & I/O device.
- **Direct Memory Access:** Device-interface transfer data directly to/from the memory without continuous involvement by the processor. DMA is a technique used for high speed I/O-device.

INTERRUPTS

There are many situations where other tasks can be performed while waiting for an I/O device to become ready. A hardware signal called an *interrupt* will alert the processor when an I/O device becomes ready. Interrupt-signal is sent on the interrupt-request line. The processor can be performing its own task without the need to continuously check the I/O-device. The routine executed in response to an interrupt-request is called Interrupt Service Routine (ISR).

When an interrupt occurs, control must be transferred to the interrupt service routine. But before transferring control, the current contents of the PC (i+1), must be saved in a known location. This will enable the return from interrupt instruction to resume execution at i+1. Return address, or the contents of the PC are usually stored on the processor stack.

Example: Consider a task that requires some computations to be performed and the results to be printed on a line printer. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces a set of n lines of output, to be printed by the PRINT routine. First, the COMPUTE routine is executed to produce the first n lines of output. Then, the PRINT routine is executed to send the first line of text to the printer.

At this point, instead of waiting for the line to be printed; the PRINT routine may be temporarily suspended and execution of the COMPUTE routine continued. Whenever the printer becomes ready, it alerts the processor by sending an interrupt request signal. In response, the processor interrupts execution of the COMPUTE routine and transfers control to the PRINT routine. The PRINT routine sends the second line to the printer and is again suspended. Then the interrupted COMPUTE routine resumes execution at the point of interruption.

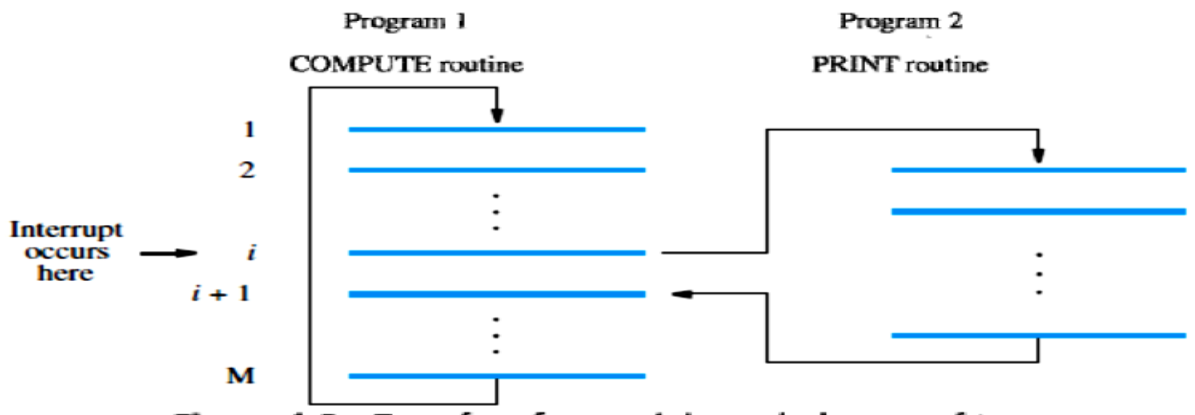


Figure 4.5 Transfer of control through the use of interrupts.

When a processor receives an interrupt-request, it must branch to the interrupt service routine. It must also inform the device that it has recognized the interrupt request. This can be accomplished in two ways:

- Some processors have an explicit interrupt-acknowledge signal for this purpose.
- In other cases, the data transfer that takes place between the device and the processor can be used to inform the device.

Saving and restoring information can be done automatically by the processor or explicitly by program instructions. Saving and restoring registers involves memory transfers:

- Increases the total execution time.
- Increases the delay between the time an interrupt request is received, & the start of execution of the interrupt service routine. This delay is called interrupt latency.

In order to reduce the interrupt latency, most processors save only the minimal amount of information:

- This minimal amount of information includes Program Counter and processor status registers.
- Any additional information that must be saved, must be saved explicitly by the program instructions at the beginning of the interrupt service routine.

INTERRUPT HARDWARE

An I/O device requests an interrupt by activating a bus line called interrupt-request. Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt-request line may be used to serve n devices as depicted in Figure 4.6. All devices are connected to the line via switches to ground.

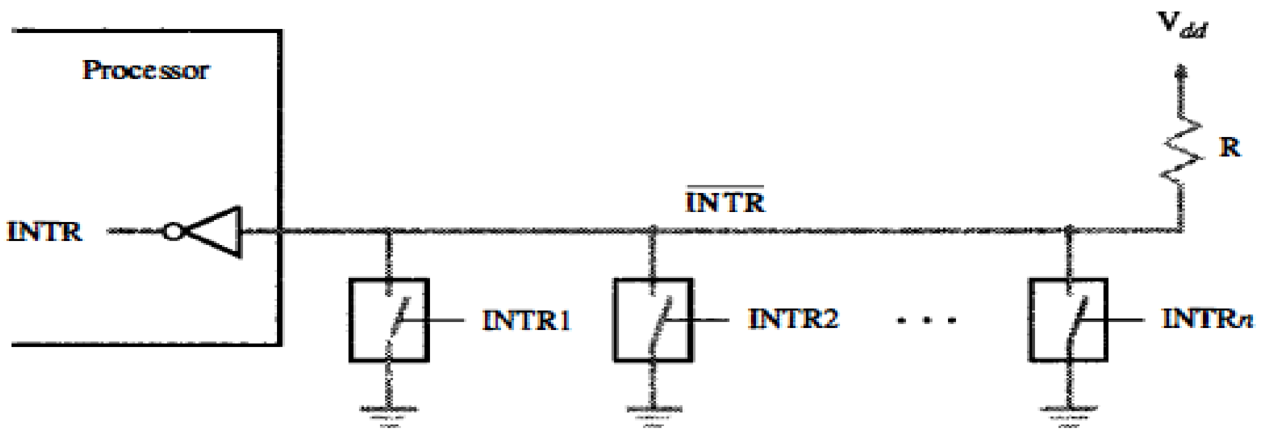


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

To request an interrupt, a device closes its associated switch. Thus, if all interrupt request signals INTR1to INTRn are inactive, i.e., if all switches are open, the voltage on the interrupt request line will be equal to V_{dd}. This is the *inactive state* of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt request signal, INTR, received by the processor to go to 1.

Since the closing of one or more switches will cause the line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices, i.e.,

$$\text{INTR} = \text{INTR1} + \text{INTR2} + \cdots\cdots + \text{INTRn}$$

In Figure 4.6, special gates known as open collector (for bipolar circuits) or open drain (for MOS circuits) are used to drive the INTR line. The output of an open collector or an open drain gate is equivalent to a switch to ground that is open when the gate's input is in the 0 state and closed when it is in the 1 state. Resistor R is called a pull up resistor because it pulls the line voltage up to the high voltage state when the switches are open.

ENABLING AND DISABLING INTERRUPTS

There are many situations in which the processor should ignore interrupt requests. Processors generally provide the ability to enable and disable such interruptions as desired. One simple way is to provide machine instructions such as ***Interrupt-enable*** and ***Interrupt-disable*** for this purpose.

Consider the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until acknowledgement. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop. There are 3 mechanisms to solve problem of infinite loop:

1. The first possibility is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the ISR has been completed. First instruction of an ISR can be Interrupt-disable. Last instruction of an ISR before the Return-from-interrupt instruction can be Interrupt-enable.

2. The second option is to have the processor automatically disable interrupts before starting the execution of the ISR. One bit in the PS (Program Status) register, called Interrupt-enable, indicates whether interrupts are enabled (if bit is 1, interrupt enabled). The processor clears the Interrupt-enable bit in its PS register, thus

disabling further interrupts. When a Return from interrupt instruction is executed, the contents of the PS are restored from the stack, setting the Interrupt enable bit back to 1. Hence, interrupts are again enabled.

3. In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be *edge-triggered*. In this case, the processor will receive only one request, regardless of how long the line is activated.

The sequence of events involved in handling an interrupt request from a single device are:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the ISR.
6. Interrupts are enabled and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

Consider the situation where a number of devices capable of initiating interrupts are connected to the processor. These devices are operationally independent. There is no definite order in which they will generate interrupts. Several devices may request interrupts at exactly the same time.

1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

When a request is received over the common interrupt request line, additional information is needed to identify the particular device that activated the line. Furthermore, if two devices have activated the line at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt service routine for the selected device has been completed, the second request can be serviced.

The information needed to determine whether a device is requesting an interrupt is available in its status register. The status register of each device has an IRQ bit which it sets to 1 when it requests an interrupt. For eg, bits KIRQ and DIRQ in Fig. 4.3 are the interrupt request bits for the keyboard and the display, respectively.

POLLING: Simplest way to identify interrupting-device is to have ISR poll all devices connected to bus. The first device encountered with its IRQ bit set is serviced. After servicing first device, next requests may be serviced.

Advantage: Simple & easy to implement.

Disadvantage: More time spent polling IRQ bits of all devices.

VECTORED INPUTS: A device requesting an interrupt identifies itself by sending a special-code to processor over bus. Then, the processor starts executing the ISR. The special-code indicates starting-address of ISR. The special-code length ranges from 4 to 8 bits. The location pointed to by the interrupting-device is used to store the staring address to ISR. The staring address to ISR is called the **interrupt vector**. The processor

→ loads interrupt-vector into PC &

→ executes appropriate ISR.

When processor is ready to receive interrupt-vector code, it activates INTA line. Then, I/O-device responds by sending its interrupt-vector code & turning off the INTR signal. The interrupt vector also includes a new value for the Processor Status Register.

INTERRUPT NESTING: I/O devices are organized in a priority structure. An interrupt request from a high-priority device is accepted while the processor is executing the interrupt service routine of a low priority device. A priority level is assigned to a processor that can be changed under program control. Processor's priority is encoded in a few bits of the processor status register. Priority can be changed by instructions that write into the processor status register. Usually, these are ***privileged*** instructions, or instructions that can be executed only in the supervisor mode.

A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in Fig.

4.7. Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

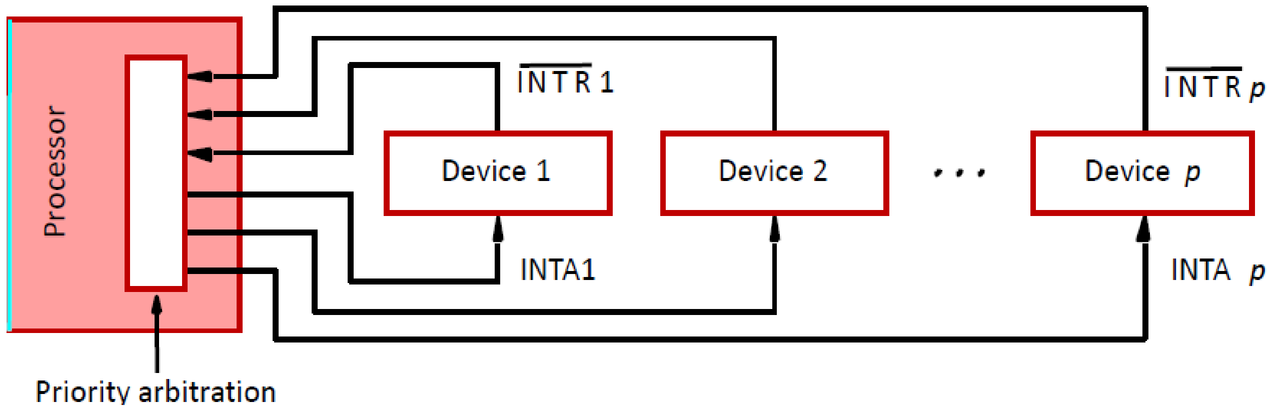
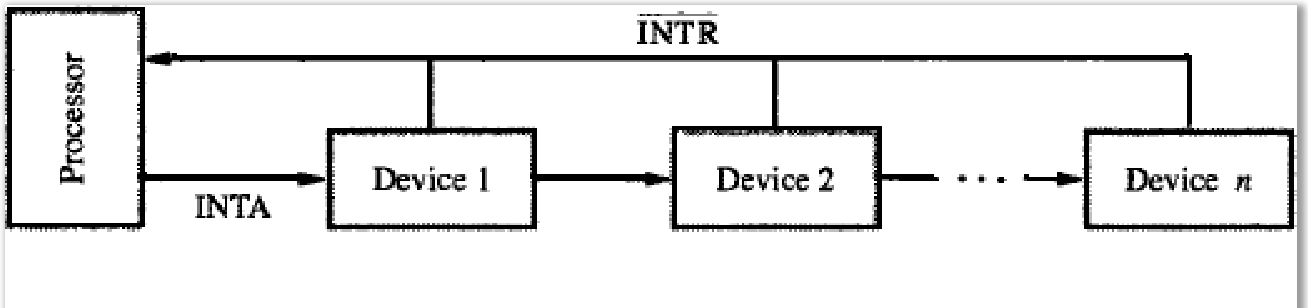


Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

Simultaneous Requests: Consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Using a priority scheme such as that of Figure 4.7, the solution is straightforward. The processor simply accepts the request having the highest priority. If several devices share one interrupt request line, as in Figure 4.6, some other mechanism is needed.

Polling scheme: If the processor uses a polling mechanism to poll the status registers of I/O devices to determine which device is requesting an interrupt. In this case the priority is determined by the order in which the devices are polled. The first device with status bit set to 1 is the device whose interrupt request is accepted.

Daisy chain scheme: Devices are connected to form a daisy chain. The interrupt request line INTR is common to all devices. Interrupt acknowledge line INTA is connected in a daisy chain fashion.

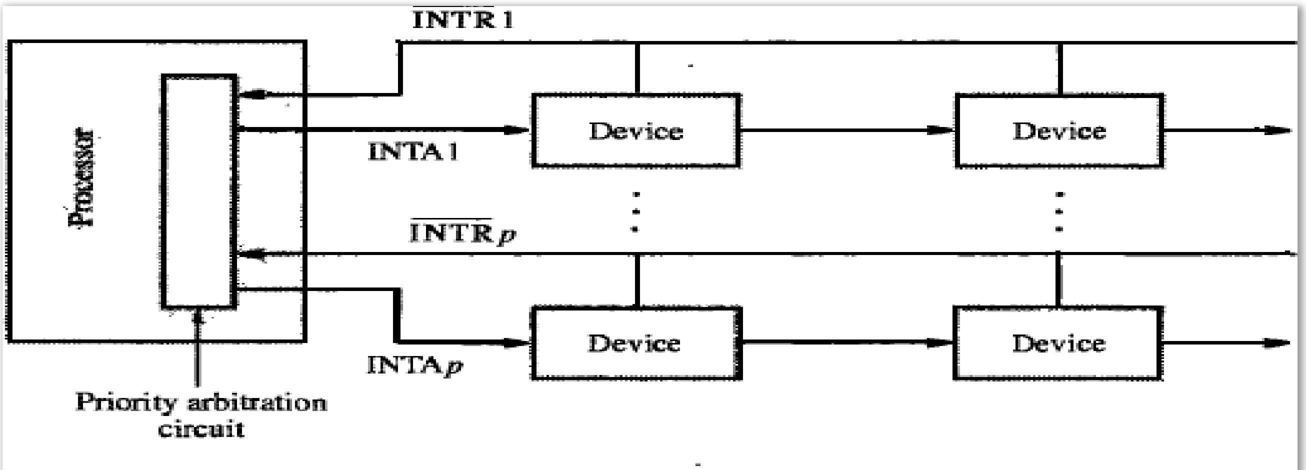


(a) Daisy chain

When devices raise an interrupt request, the interrupt request line INTR is activated. The processor responds by setting INTA line to 1. This signal is received by device 1; if device 1 does not need service, it passes the signal to device 2. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Device that is electrically closest to the processor has the highest priority.

When I/O devices were organized into a priority structure, each device had its own interrupt-request and interrupt-acknowledge line. When I/O devices were organized in a daisy chain fashion, the devices shared an interrupt-request line, and the interrupt-acknowledge propagated through the devices. A combination of priority structure and daisy chain scheme can also be used.

Devices are organized into groups. Each group is assigned a different priority level. All the devices within a single group share an interrupt-request line, and are connected to form a daisy chain.


$\overline{\text{INTR}}$ 1
Processor
INTA 1
Device
Device
$\overline{\text{INTR}}\,p$
INTA p
Device
Device
Priority arbitration circuit

(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

CONTROLLING DEVICE REQUESTS

Only those devices that are being used in a program should be allowed to generate interrupt requests. To control which devices are allowed to generate interrupt requests, the interface circuit of each I/O device has an interrupt-enable bit. If the interrupt-enable bit in the device interface is set to 1, then the device is allowed to generate an interrupt-request. Interrupt-enable bit in the device's interface circuit determines whether the device is allowed to generate an interrupt request. Interrupt-enable bit in the processor status register or the priority structure of the interrupts determines whether a given interrupt will be accepted.

For example, keyboard interrupt-enable, KEN, and display interrupt-enable, DEN, flags in register CONTROL. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.

To summarize, there are two independent mechanisms for controlling interrupt requests:

- At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request.
- At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

Consider a processor that uses the vectored interrupt scheme, where the starting address of the interrupt-service routine is stored at memory location INTVEC. Interrupts are enabled by setting to 1 an interrupt-enable bit, IE, in the processor status word, which we assume is bit 9. A keyboard and a display unit connected to this processor have the status, control, and data registers. Assume that at some point in a program called Main we wish to read an input line from the keyboard and store the characters in successive byte locations in the memory, starting at location LINE. To perform this operation using interrupts, we need to initialize the interrupt process, as follows:

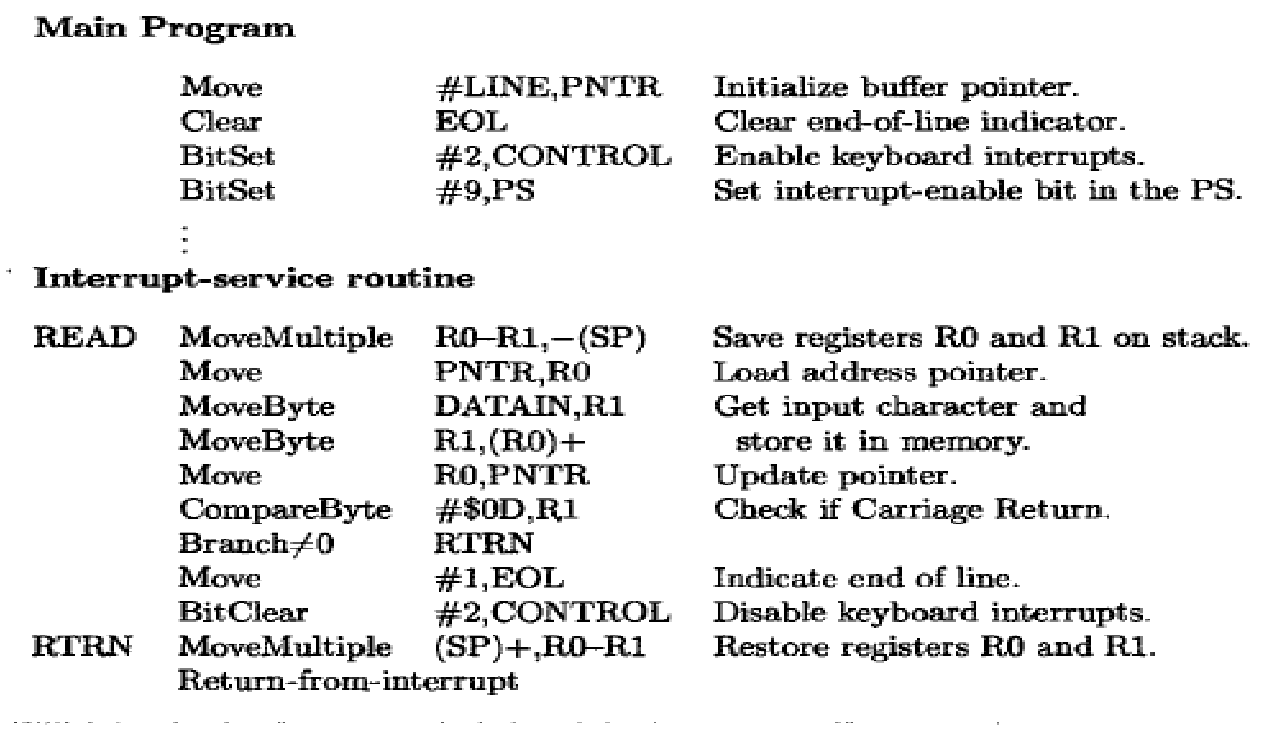
1. Load the starting address of the interrupt-service routine in location INTVEC.
2. Load the address LINE in a memory location PNTR. The interrupt-service routine will use this location as a pointer to store the input characters in the memory.
3. Enable keyboard interrupts by setting bit 2 in register CONTROL to 1.
4. Enable interrupts in the processor by setting to 1 the IE bit in the processor status register, PS.

Once this initialization is completed, typing a character on the keyboard will cause an interrupt request to be generated by the keyboard interface. The program being executed at that time will be interrupted and the interrupt-service routine will be executed.

This routine has to perform the following tasks:

1. Read the input character from the keyboard input data register. This will cause the interface circuit to remove its interrupt request.
2. Store the character in the memory location pointed to by PNTR, and increment PNTR.
3. When the end of the line is reached, disable keyboard interrupts and inform program Main.
4. Return from interrupt.

The instructions needed to perform these tasks are shown in Figure 4.9. When the end of the input line is detected, the interrupt-service routine clears the KEN bit in register CONTROL as no further input is expected. It also sets to 1 the variable EOL (End of Line).



```
Main Program

          Move           #LINE,PNTR     Initialize buffer pointer.
          Clear          EOL            Clear end-of-line indicator.
          BitSet         #2,CONTROL     Enable keyboard interrupts.
          BitSet         #9,PS          Set interrupt-enable bit in the PS.
            ⋮
Interrupt-service routine

READ      MoveMultiple   R0–R1,–(SP)    Save registers R0 and R1 on stack.
          Move           PNTR,R0        Load address pointer.
          MoveByte       DATAIN,R1      Get input character and
          MoveByte       R1,(R0)+         store it in memory.
          Move           R0,PNTR        Update pointer.
          CompareByte    #$0D,R1        Check if Carriage Return.
          Branch≠0       RTRN
          Move           #1,EOL         Indicate end of line.
          BitClear       #2,CONTROL     Disable keyboard interrupts.
RTRN      MoveMultiple   (SP)+,R0–R1    Restore registers R0 and R1.
          Return-from-interrupt
```

Figure 4.9 Using interrupts to read a line of characters from a keyboard via the registers in Figure 4.3.

DIRECT MEMORY ACCESS

A special control unit may be provided to transfer a block of data directly between an I/O device and the main memory, without continuous intervention by the processor. This approach is called *direct memory access*, or DMA.

DMA transfers are performed by *DMA controller*, which is a control circuit that is a part of the I/O device interface. DMA controller performs functions that would be normally carried out by the processor:

- For each word, it provides the memory address & all the control signals.
- To transfer a block of data, it increments the memory addresses and keeps track of the number of transfers.

DMA controller can transfer a block of data from an external device to the processor, without any intervention from the processor. However, the operation of the DMA controller must be under the control of a program executed by the processor. That is, the processor must initiate the DMA transfer.

To initiate the DMA transfer, the processor informs the DMA controller of:

- Starting address,
- Number of words in the block.
- Direction of transfer (I/O device to the memory, or memory to the I/O device).

Once the DMA controller completes the DMA transfer, it informs the processor by raising an interrupt signal. While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer.

For an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state, initiates the DMA operation, and starts the execution of another program. When the transfer is completed, the DMA controller informs the processor by sending an interrupt request. In response, the OS puts the suspended program in the Runnable state so that it can be selected by the scheduler to continue execution.

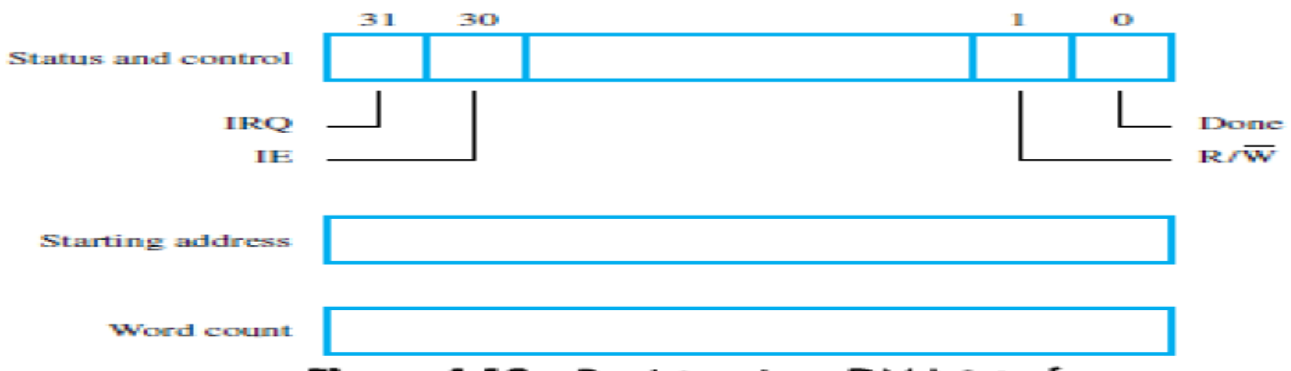


Figure 4.18 Registers in a DMA interface.

Figure 8.12 shows an example of the DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a read operation. Otherwise, it performs a write operation. When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt.

An example of a computer system is given in Figure 4.19, showing how DMA controllers may be used. DMA controller connects a high-speed network to the computer bus. Disk controller, which controls two disks also has DMA capability. It provides two DMA channels. It can perform two independent DMA operations, as if each disk has its own DMA controller. The registers to store the memory address, word count and status and control information are duplicated.

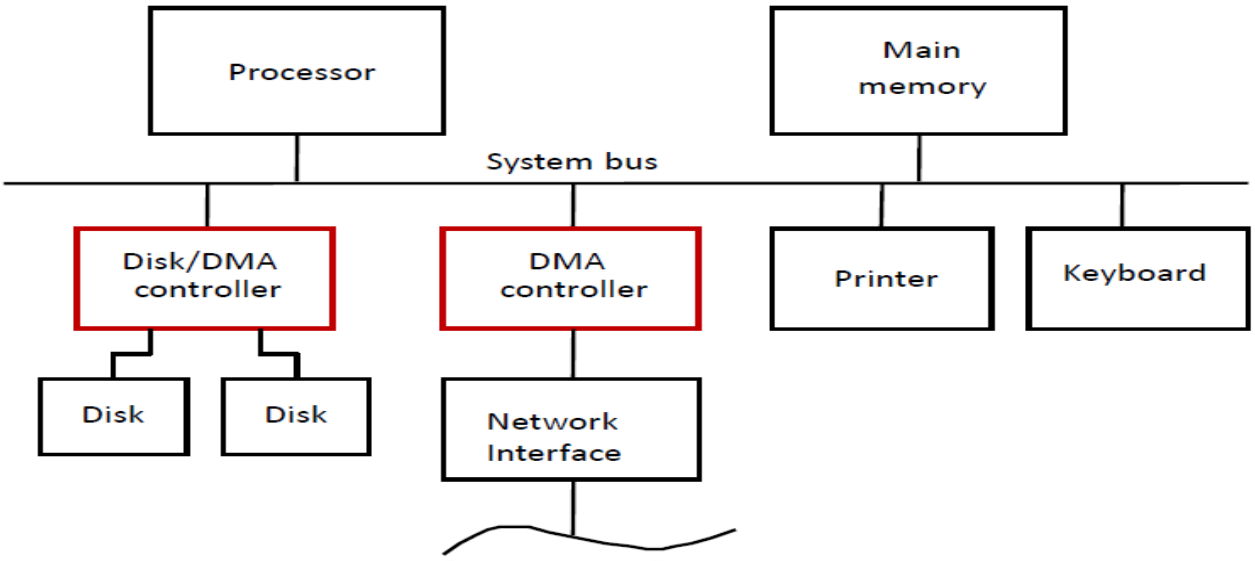


Figure 4.19 Use of DMA controllers in a computer system.

To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller. It also provides the disk controller with information to identify the data for future retrieval. The DMA controller proceeds independently to implement the specified operation. When the DMA transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. DMA devices are given higher priority than the processor to access the bus. Among different DMA devices, high priority is given to high-speed peripherals such as a disk or a graphics display device.

Processor originates most memory access cycles on the bus. DMA controller can be said to "*steal*" memory access cycles from the bus. This interweaving technique is called "*cycle stealing*". An alternate approach is to provide a DMA controller an exclusive capability to initiate transfers on the bus, and hence exclusive access to the main memory. This is known as the *block or burst mode.*

Most DMA controllers incorporate a data storage buffer. In the case of the network interface in Figure 4.19, for example, the DMA controller reads a block of data from the main memory and stores it into its input buffer. This transfer takes place using burst mode at a speed appropriate to the memory and the computer bus. Then, the data in the buffer are transmitted over the network at the speed of the network. A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.