UNIT - 3

INPUT/OUTPUT

Q.] The reader may wonder why one does not connect peripherals directly to the

system bus. The reasons are as follows:

■ There are a wide variety of peripherals with various methods of operation. It

would be impractical to incorporate the necessary logic within the processor to

control a range of devices.

■ The data transfer rate of peripherals is often much slower than that of the

memory or processor. Thus, it is impractical to use the high-speed system bus

to communicate directly with a peripheral.

■ On the other hand, the data transfer rate of some peripherals is faster than

that of the memory or processor. Again, the mismatch would lead to inefficiencies if not managed properly.

■ Peripherals often use different data formats and word lengths than the computer to which they are attached.

■ Thus, an I/O module is required. This module has two major functions (Figure 7.1):

■ Interface to the processor and memory via the system bus or central switch.

■ Interface to one or more peripheral devices by tailored data links.

***7.1]EXTERNAL DEVICES:***

I/O operations are accomplished through a wide assortment of external devices

that provide a means of exchanging data between the external environment and the

computer. An external device attaches to the computer by a link to an I/O module

(Figure 7.1). The link is used to exchange control, status, and data between the I/O

module and the external device. An external device connected to an I/O module is

often referred to as a peripheral device or, simply, a peripheral.

We can broadly classify external devices into three categories:

■ Human readable: Suitable for communicating with the computer user;

■ Machine readable: Suitable for communicating with equipment;

■ Communication: Suitable for communicating with remote devices.

Examples of human-readable devices are video display terminals (VDTs) and

printers. Examples of machine- readable devices are magnetic disk and tape systems, and sensors and actuators, such as are used in a robotics application. Note

that we are viewing disk and tape systems as I/O devices in this chapter, whereas

in Chapter 6 we viewed them as memory devices. From a functional point of view,

these devices are part of the memory hierarchy, and their use is appropriately discussed in Chapter 6. From a structural point of view, these devices are controlled by

I/O modules and are hence to be considered in this chapter.

Communication devices allow a computer to exchange data with a remote

device, which may be a human- readable device, such as a terminal, a machinereadable device, or even another computer.

In very general terms, the nature of an external device is indicated in Figure 7.2.

The interface to the I/O module is in the form of control, data, and status signals. Control signals determine the function that the device will perform, such as send data to

the I/O module (INPUT or READ), accept data from the I/O module (OUTPUT or

WRITE), report status, or perform some control function particular to the device (e.g.,

position a disk head). Data are in the form of a set of bits to be sent to or received from

the I/O module. Status signals indicate the state of the device. Examples are READY/

NOT-READY to show whether the device is ready for data transfer.

Control logic associated with the device controls the device’s operation in

response to direction from the I/O module. The transducer converts data from electrical to other forms of energy during output and from other forms to electrical

during input. Typically, a buffer is associated with the transducer to temporarily

hold data being transferred between the I/O module and the external environment.

A buffer size of 8 to 16 bits is common for serial devices, whereas block- oriented

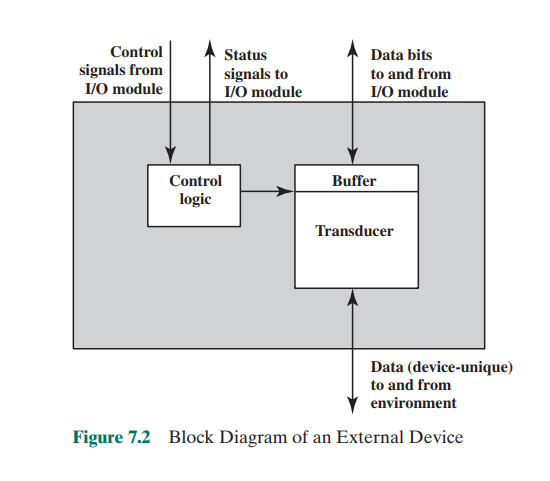
devices such as disk drive controllers may have much larger buffers.

**Keyboard/Monitor**

The most common means of computer/user interaction is a keyboard/monitor

arrangement. The user provides input through the keyboard, the input is then transmitted to the computer and may also be displayed on the monitor. In addition, the

monitor displays data provided by the computer.



The basic unit of exchange is the character. Associated with each character is a code, typically 7 or 8 bits in length. The most commonly used text code

is the International Reference Alphabet (IRA).1

Each character in this code is

represented by a unique 7-bit binary code; thus, 128 different characters can be

represented. Characters are of two types: printable and control. Printable characters are the alphabetic, numeric, and special characters that can be printed on

paper or displayed on a screen. Some of the control characters have to do with

controlling the printing or displaying of characters; an example is carriage return.

Other control characters are concerned with communications procedures. See

Appendix H for details.

For keyboard input, when the user depresses a key, this generates an electronic signal that is interpreted by the transducer in the keyboard and translated

into the bit pattern of the corresponding IRA code. This bit pattern is then transmitted to the I/O module in the computer. At the computer, the text can be stored

in the same IRA code. On output, IRA code characters are transmitted to an external device from the I/O module. The transducer at the device interprets this code

and sends the required electronic signals to the output device either to display the

indicated character or perform the requested control function.

**Disk Drive**

A disk drive contains electronics for exchanging data, control, and status signals with

an I/O module plus the electronics for controlling the disk read/write mechanism.

In a fixed-head disk, the transducer is capable of converting between the magnetic

patterns on the moving disk surface and bits in the device’s buffer (Figure 7.2). A

moving-head disk must also be able to cause the disk arm to move radially in and

out across the disk’s surface.

***7.2] I/O MODULES***

Module Function

The major functions or requirements for an I/O module fall into the following

categories:

■ Control and timing

■ Processor communication

■ Device communication

■ Data buffering

■ Error detection

During any period of time, the processor may communicate with one or more

external devices in unpredictable patterns, depending on the program’s need for

I/O. The internal resources, such as main memory and the system bus, must be shared

among a number of activities, including data I/O. Thus, the I/O function includes a

control and timing requirement, to coordinate the flow of traffic between internal

resources and external devices. For example, the control of the transfer of data from

an external device to the processor might involve the following sequence of steps:

1. The processor interrogates the I/O module to check the status of the attached

device.

2. The I/O module returns the device status.

3. If the device is operational and ready to transmit, the processor requests the

transfer of data, by means of a command to the I/O module.

4. The I/O module obtains a unit of data (e.g., 8 or 16 bits) from the external

device.

5. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

The preceding simplified scenario also illustrates that the I/O module must

communicate with the processor and with the external device. Processor communication involves the following:

■ Command decoding: The I/O module accepts commands from the processor,

typically sent as signals on the control bus. For example, an I/O module for a

disk drive might accept the following commands: READ SECTOR, WRITE

SECTOR, SEEK track number, and SCAN record ID. The latter two commands each include a parameter that is sent on the data bus.

■ Data: Data are exchanged between the processor and the I/O module over the

data bus.

■ Status reporting: Because peripherals are so slow, it is important to know the

status of the I/O module. For example, if an I/O module is asked to send data

to the processor (read), it may not be ready to do so because it is still working

on the previous I/O command. This fact can be reported with a status signal.

Common status signals are BUSY and READY. There may also be signals to

report various error conditions.

■ Address recognition: Just as each word of memory has an address, so does

each I/O device. Thus, an I/O module must recognize one unique address for

each peripheral it controls.

On the other side, the I/O module must be able to perform device communication.

This communication involves commands, status information, and data (Figure 7.2).

An essential task of an I/O module is data buffering. The need for this function is apparent from Figure 2.1. Whereas the transfer rate into and out of main

memory or the processor is quite high, the rate is orders of magnitude lower for

many peripheral devices and covers a wide range. Data coming from main memory

are sent to an I/O module in a rapid burst. The data are buffered in the I/O module

and then sent to the peripheral device at its data rate. In the opposite direction, data

are buffered so as not to tie up the memory in a slow transfer operation. Thus, the

I/O module must be able to operate at both device and memory speeds. Similarly, if

the I/O device operates at a rate higher than the memory access rate, then the I/O

module performs the needed buffering operation.

Finally, an I/O module is often responsible for error detection and for subsequently reporting errors to the processor. One class of errors includes mechanical

and electrical malfunctions reported by the device (e.g., paper jam, bad disk track).

Another class consists of unintentional changes to the bit pattern as it is transmitted from device to I/O module. Some form of error- detecting code is often used

to detect transmission errors. A simple example is the use of a parity bit on each

character of data. For example, the IRA character code occupies 7 bits of a byte.

The eighth bit is set so that the total number of 1s in the byte is even (even parity)

or odd (odd parity). When a byte is received, the I/O module checks the parity to

determine whether an error has occurred

**I/O Module Structure**

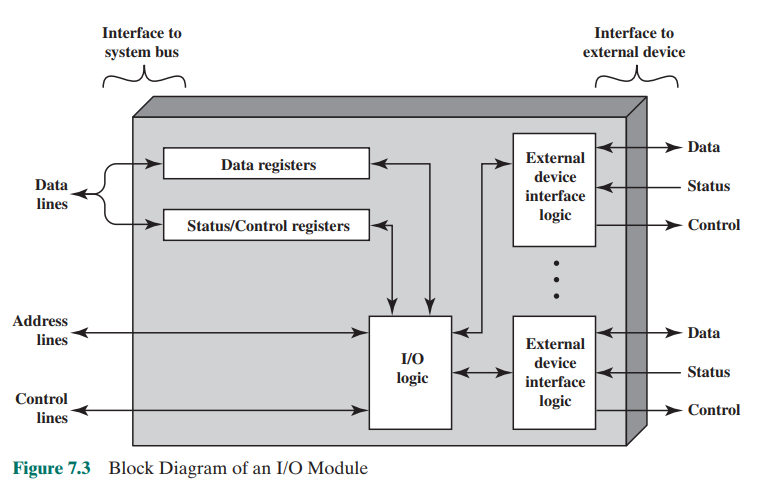
I/O modules vary considerably in complexity and the number of external devices

that they control. We will attempt only a very general description here. (One

specific device, the Intel 8255A, is described in Section 7.4.) Figure 7.3 provides a

general block diagram of an I/O module. The module connects to the rest of the

computer through a set of signal lines (e.g., system bus lines). Data transferred to

and from the module are buffered in one or more data registers. There may also

be one or more status registers that provide current status information. A status

register may also function as a control register, to accept detailed control information from the processor. The logic within the module interacts with the processor

via a set of control lines. The processor uses the control lines to issue commands

to the I/O module. Some of the control lines may be used by the I/O module (e.g.,

for arbitration and status signals). The module must also be able to recognize and

generate addresses associated with the devices it controls. Each I/O module has

a unique address or, if it controls more than one external device, a unique set of

addresses. Finally, the I/O module contains logic specific to the interface with each

device that it controls.

An I/O module functions to allow the processor to view a wide range of devices

in a simple-minded way. There is a spectrum of capabilities that may be provided.

The I/O module may hide the details of timing, formats, and the electromechanics

of an external device so that the processor can function in terms of simple read and

write commands, and possibly open and close file commands. In its simplest form,

the I/O module may still leave much of the work of controlling a device (e.g., rewind

a tape) visible to the processor.

An I/O module that takes on most of the detailed processing burden, presenting a high-level interface to the processor, is usually referred to as an I/O channel or

I/O processor. An I/O module that is quite primitive and requires detailed control

is usually referred to as an I/O controller or device controller. I/O controllers are

commonly seen on microcomputers, whereas I/O channels are used on mainframes.

In what follows, we will use the generic term I/O module when no confusion

results and will use more specific terms where necessary.

***7.3]PROGRAMMED I/O***

Three techniques are possible for I/O operations. With programmed I/O, data are

exchanged between the processor and the I/O module. The processor executes a

program that gives it direct control of the I/O operation, including sensing device

status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is

complete. If the processor is faster than the I/O module, this is waste of processor

time. With interrupt-driven I/O, the processor issues an I/O command, continues

to execute other instructions, and is interrupted by the I/O module when the latter

has completed its work. With both programmed and interrupt I/O, the processor is

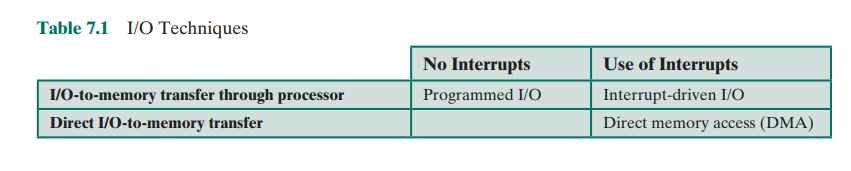
responsible for extracting data from main memory for output and storing data in

main memory for input. The alternative is known as direct memory access (DMA).

In this mode, the I/O module and main memory exchange data directly, without

processor involvement

Table 7.1 indicates the relationship among these three techniques. In this section, we explore programmed I/O. Interrupt I/O and DMA are explored in the following two sections, respectively.



**Overview of Programmed I/O**

When the processor is executing a program and encounters an instruction relating to

I/O, it executes that instruction by issuing a command to the appropriate I/O module.

With programmed I/O, the I/O module will perform the requested action and then

set the appropriate bits in the I/O status register (Figure 7.3). The I/O module takes

no further action to alert the processor. In particular, it does not interrupt the processor. Thus, it is the responsibility of the processor to periodically check the status

of the I/O module until it finds that the operation is complete.

To explain the programmed I/O technique, we view it first from the point of

view of the I/O commands issued by the processor to the I/O module, and then from

the point of view of the I/O instructions executed by the processor.

**I/O Commands**

To execute an I/O-related instruction, the processor issues an address, specifying

the particular I/O module and external device, and an I/O command. There are four

types of I/O commands that an I/O module may receive when it is addressed by a

processor:

■ Control: Used to activate a peripheral and tell it what to do. For example, a

magnetic-tape unit may be instructed to rewind or to move forward one record.

These commands are tailored to the particular type of peripheral device.

■ Test: Used to test various status conditions associated with an I/O module and

its peripherals. The processor will want to know that the peripheral of interest is powered on and available for use. It will also want to know if the most

recent I/O operation is completed and if any errors occurred.

■ Read: Causes the I/O module to obtain an item of data from the peripheral

and place it in an internal buffer (depicted as a data register in Figure 7.3). The

processor can then obtain the data item by requesting that the I/O module

place it on the data bus.

■ Write: Causes the I/O module to take an item of data (byte or word) from the

data bus and subsequently transmit that data item to the peripheral.

Figure 7.4a gives an example of the use of programmed I/O to read in a block of

data from a peripheral device (e.g., a record from tape) into memory. Data are read

in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must

remain in a status-checking cycle until it determines that the word is available in the

I/O module’s data register. This flowchart highlights the main disadvantage of this

technique: it is a time-consuming process that keeps the processor busy needlessly.

**I/O Instructions**

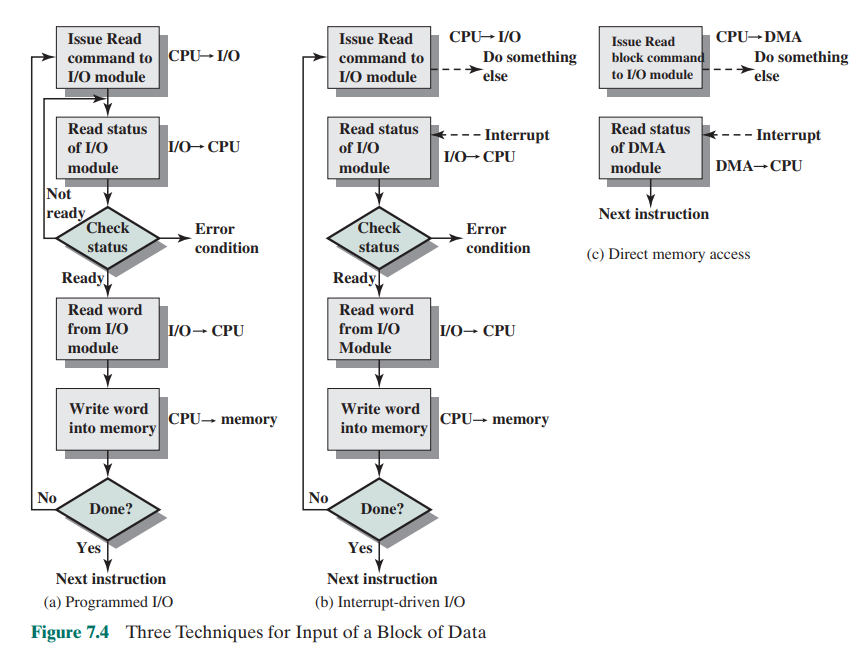
With programmed I/O, there is a close correspondence between the I/O-related

instructions that the processor fetches from memory and the I/O commands that the

processor issues to an I/O module to execute the instructions. That is, the instructions

are easily mapped into I/O commands, and there is often a simple one-to-one relationship. The form of the instruction depends on the way in which external devices

are addressed.



Typically, there will be many I/O devices connected through I/O modules to

the system. Each device is given a unique identifier or address. When the processor

issues an I/O command, the command contains the address of the desired device.

Thus, each I/O module must interpret the address lines to determine if the command is for itself.

When the processor, main memory, and I/O share a common bus, two modes

of addressing are possible: memory mapped and isolated. With memory- mapped

I/O, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations and

uses the same machine instructions to access both memory and I/O devices. So, for

example, with 10 address lines, a combined total of 210 = 1024 memory locations

and I/O addresses can be supported, in any combination.

With memory-mapped I/O, a single read line and a single write line are needed

on the bus. Alternatively, the bus may be equipped with memory read and write plus

input and output command lines. The command line specifies whether the address

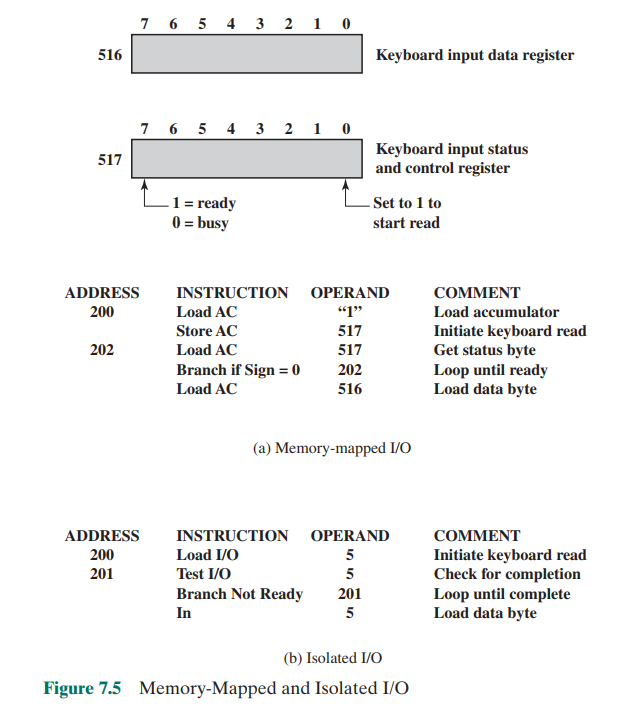
refers to a memory location or an I/O device. The full range of addresses may be

available for both. Again, with 10 address lines, the system may now support both

1024 memory locations and 1024 I/O addresses. Because the address space for I/O is

isolated from that for memory, this is referred to as isolated I/O.

Figure 7.5 contrasts these two programmed I/O techniques. Figure 7.5a shows how the interface for a simple input device such as a terminal keyboard might appear to a programmer using memory-mapped I/O. Assume a 10-bit address, with a 512-bit memory (locations 0–511) and up to 512 I/O addresses (locations 512–1023). Two addresses are dedicated to keyboard input from a particular terminal. Address 516 refers to the data register and address 517 refers to the status register, which also functions as a control register for receiving processor commands. The program shown will read 1 byte of data from the keyboard into an accumulator register in the processor. Note that the processor loops until the data byte is available. With isolated I/O (Figure 7.5b), the I/O ports are accessible only by special I/O commands, which activate the I/O command lines on the bus. For most types of processors, there is a relatively large set of different instructions for referencing memory. If isolated I/O is used, there are only a few I/O instructions. Thus, an advantage of memory-mapped I/O is that this large repertoire of instructions can be used, allowing more efficient programming. A disadvantage is that valuable memory address space is used up. Both memory-mapped and isolated I/O are in common use.



***7.4]INTERRUPT-DRIVEN I/O***

The problem with programmed I/O is that the processor has to wait a long time

for the I/O module of concern to be ready for either reception or transmission of

data. The processor, while waiting, must repeatedly interrogate the status of the I/O

module. As a result, the level of the performance of the entire system is severely

degraded.

An alternative is for the processor to issue an I/O command to a module and

then go on to do some other useful work. The I/O module will then interrupt the

processor to request service when it is ready to exchange data with the processor.

The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module.

For input, the I/O module receives a READ command from the processor. The I/O

module then proceeds to read data in from an associated peripheral. Once the data

are in the module’s data register, the module signals an interrupt to the processor

over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is

then ready for another I/O operation.

From the processor’s point of view, the action for input is as follows. The processor issues a READ command. It then goes off and does something else (e.g., the

processor may be working on several different programs at the same time). At the

end of each instruction cycle, the processor checks for interrupts (Figure 3.9). When

the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the

interrupt. In this case, the processor reads the word of data from the I/O module

and stores it in memory. It then restores the context of the program it was working

on (or some other program) and resumes execution.

Figure 7.4b shows the use of interrupt I/O for reading in a block of data.

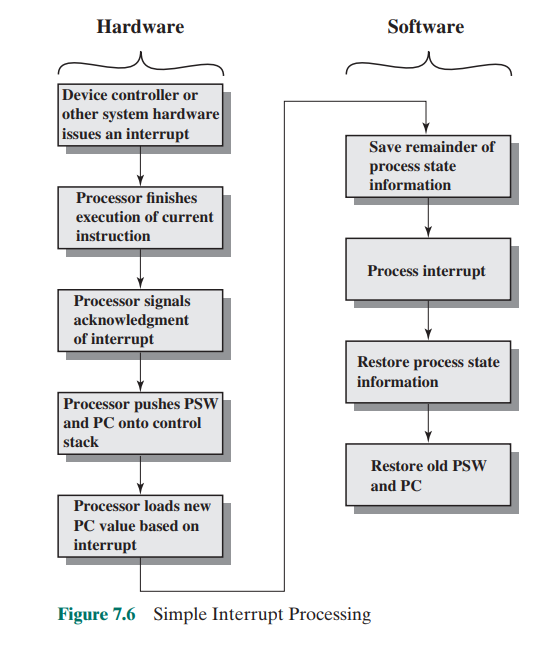
Compare this with Figure 7.4a. Interrupt I/O is more efficient than programmed

I/O because it eliminates needless waiting. However, interrupt I/O still consumes

a lot of processor time, because every word of data that goes from memory to I/O

module or from I/O module to memory must pass through the processor.

**Interrupt Processing**

Let us consider the role of the processor in interrupt-driven I/O in more detail. The

occurrence of an interrupt triggers a number of events, both in the processor hardware and in software. Figure 7.6 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.

2. The processor finishes execution of the current instruction before responding

to the interrupt, as indicated in Figure 3.9.

3. The processor tests for an interrupt, determines that there is one, and sends an

acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.

4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the

status of the processor, which is contained in a register called the program

status word (PSW); and (b) the location of the next instruction to be executed,

which is contained in the program counter. These can be pushed onto the system control stack.2

5. The processor now loads the program counter with the entry location of the

interrupt-handling program that will respond to this interrupt. Depending on

the computer architecture and operating system design, there may be a single

program; one program for each type of interrupt; or one program for each

device and each type of interrupt. If there is more than one interrupt-handling

routine, the processor must determine which one to invoke. This information

may have been included in the original interrupt signal, or the processor may

have to issue a request to the device that issued the interrupt to get a response

that contains the needed information

Once the program counter has been loaded, the processor proceeds to the

next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that

control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the “state” of the executing program. In

particular, the contents of the processor registers need to be saved, because

these registers may be used by the interrupt handler. So, all of these values,

plus any other state information, need to be saved. Typically, the interrupt

handler will begin by saving the contents of all registers on the stack. Figure 7.7a shows a simple example. In this case, a user program is interrupted

after the instruction at location N. The contents of all of the registers plus the

address of the next instruction (N + 1) are pushed onto the stack. The stack

pointer is updated to point to the new top of stack, and the program counter is

updated to point to the beginning of the interrupt service routine.

7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that

caused an interrupt. It may also involve sending additional commands or

acknowledgments to the I/O device.

8. When interrupt processing is complete, the saved register values are retrieved

from the stack and restored to the registers (e.g., see Figure 7.7b).

9. The final act is to restore the PSW and program counter values from the stack.

As a result, the next instruction to be executed will be from the previously

interrupted program.

Note that it is important to save all the state information about the interrupted

program for later resumption. This is because the interrupt is not a routine called

from the program. Rather, the interrupt can occur at any time and therefore at any

point in the execution of a user program. Its occurrence is unpredictable. Indeed, as

we will see in the next chapter, the two programs may not have anything in common

and may belong to two different users.

**Design Issues**

Two design issues arise in implementing interrupt I/O. First, because there will

almost invariably be multiple I/O modules, how does the processor determine which

device issued the interrupt? And second, if multiple interrupts have occurred, how

does the processor decide which one to process?

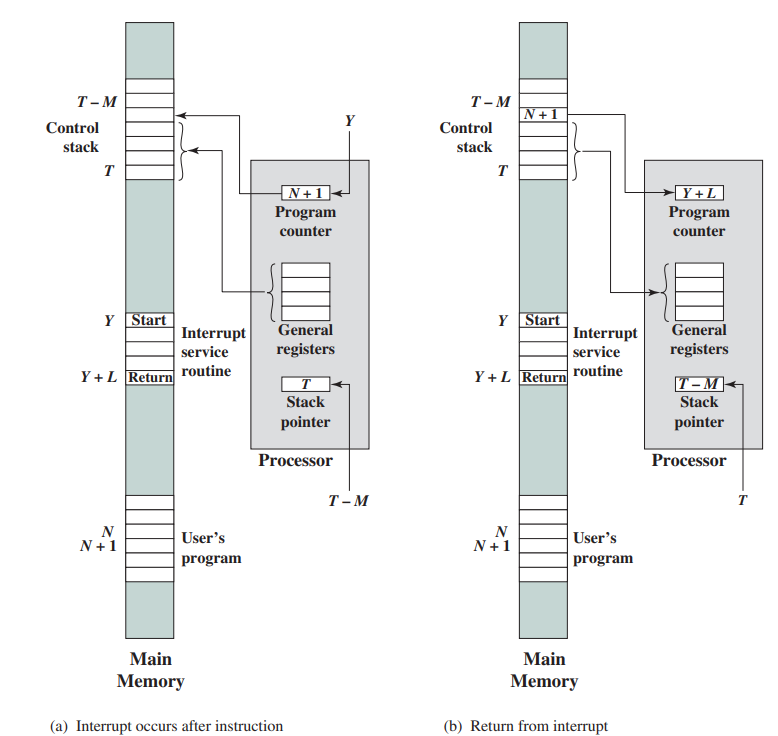
Let us consider device identification first. Four general categories of techniques are in common use:

■ Multiple interrupt lines

■ Software poll

■ Daisy chain (hardware poll, vectored)

■ Bus arbitration (vectored)



The most straightforward approach to the problem is to provide multiple interrupt lines between the processor and the I/O modules. However, it is impractical to

dedicate more than a few bus lines or processor pins to interrupt lines. Consequently,

even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus, one of the other three techniques must be used on each line.

One alternative is the software poll. When the processor detects an interrupt,

it branches to an interrupt-service routine that polls each I/O module to determine

which module caused the interrupt. The poll could be in the form of a separate command line (e.g., TESTI/O). In this case, the processor raises TESTI/O and places the

address of a particular I/O module on the address lines. The I/O module responds

positively if it set the interrupt. Alternatively, each I/O module could contain an

addressable status register. The processor then reads the status register of each I/O

module to identify the interrupting module. Once the correct module is identified,

the processor branches to a device-service routine specific to that device.

The disadvantage of the software poll is that it is time consuming. A more

efficient technique is to use a daisy chain, which provides, in effect, a hardware poll.

An example of a daisy-chain configuration is shown in Figure 3.26. For interrupts,

all I/O modules share a common interrupt request line. The interrupt acknowledge

line is daisy chained through the modules. When the processor senses an interrupt,

it sends out an interrupt acknowledge. This signal propagates through a series of

I/O modules until it gets to a requesting module. The requesting module typically

responds by placing a word on the data lines. This word is referred to as a vector and

is either the address of the I/O module or some other unique identifier. In either

case, the processor uses the vector as a pointer to the appropriate device- service

routine. This avoids the need to execute a general interrupt- service routine first.

This technique is called a vectored interrupt.

There is another technique that makes use of vectored interrupts, and that is

bus arbitration. With bus arbitration, an I/O module must first gain control of the

bus before it can raise the interrupt request line. Thus, only one module can raise the

line at a time. When the processor detects the interrupt, it responds on the interrupt

acknowledge line. The requesting module then places its vector on the data lines.

The aforementioned techniques serve to identify the requesting I/O module.

They also provide a way of assigning priorities when more than one device is requesting interrupt service. With multiple lines, the processor just picks the interrupt line

with the highest priority. With software polling, the order in which modules are

polled determines their priority. Similarly, the order of modules on a daisy chain

determines their priority. Finally, bus arbitration can employ a priority scheme, as

discussed in Section 3.4.

We now turn to two examples of interrupt structures.

**Intel 82C59A Interrupt Controller**

The Intel 80386 provides a single Interrupt Request (INTR) and a single Interrupt

Acknowledge (INTA) line. To allow the 80386 to handle a variety of devices and

priority structures, it is usually configured with an external interrupt arbiter, the

82C59A. External devices are connected to the 82C59A, which in turn connects to

the 80386.

Figure 7.8 shows the use of the 82C59A to connect multiple I/O modules for

the 80386. A single 82C59A can handle up to eight modules. If control for more

than eight modules is required, a cascade arrangement can be used to handle up to

64 modules.

The 82C59A’s sole responsibility is the management of interrupts. It accepts

interrupt requests from attached modules, determines which interrupt has the

highest priority, and then signals the processor by raising the INTR line. The processor acknowledges via the INTA line. This prompts the 82C59A to place the

appropriate vector information on the data bus. The processor can then proceed

to process the interrupt and to communicate directly with the I/O module to read

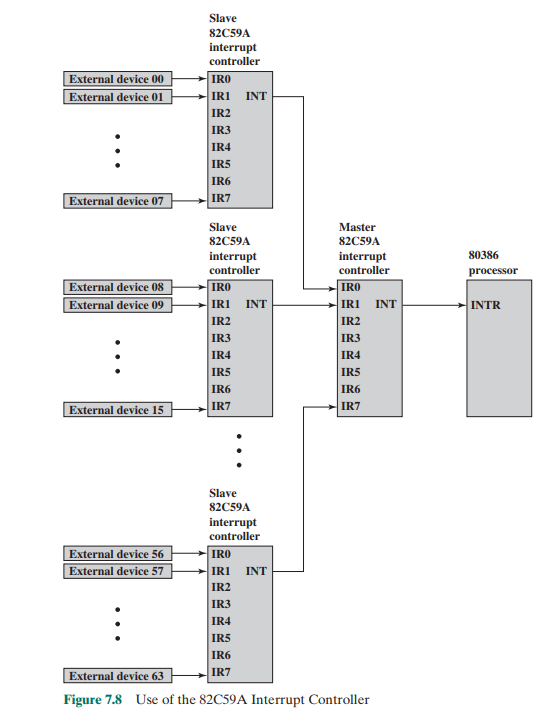
or write data.

The 82C59A is programmable. The 80386 determines the priority scheme to be

used by setting a control word in the 82C59A. The following interrupt modes are possible:

■ Fully nested: The interrupt requests are ordered in priority from 0 (IR0)

through 7 (IR7).



■ Rotating: In some applications a number of interrupting devices are of equal

priority. In this mode a device, after being serviced, receives the lowest priority in the group.

■ Special mask: This allows the processor to inhibit interrupts from certain

devices.

**The Intel 8255A Programmable Peripheral Interface**

As an example of an I/O module used for programmed I/O and interrupt-driven

I/O, we consider the Intel 8255A Programmable Peripheral Interface. The 8255A is

a single-chip, general-purpose I/O module originally designed for use with the Intel

80386 processor. It has since been cloned by other manufacturers and is a widely

used peripheral controller chip. Its uses include as a controller for simple I/O devices

for microprocessors and in embedded systems, including microcontroller systems.

architecture and operation Figure 7.9 shows a general block diagram plus

the pin assignment for the 40-pin package in which it is housed. As shown on the pin

layout, the 8255A includes the following lines:

■ D0–D7: These are the data I/O lines for the device. All information read from

and written to the 8255A occurs via these eight data lines.

■ CS (Chip Select Input): If this line is a logical 0, the microprocessor can read

and write to the 8255A.

■ RD (Read Input): If this line is a logical 0 and the CS input is a logical 0, the

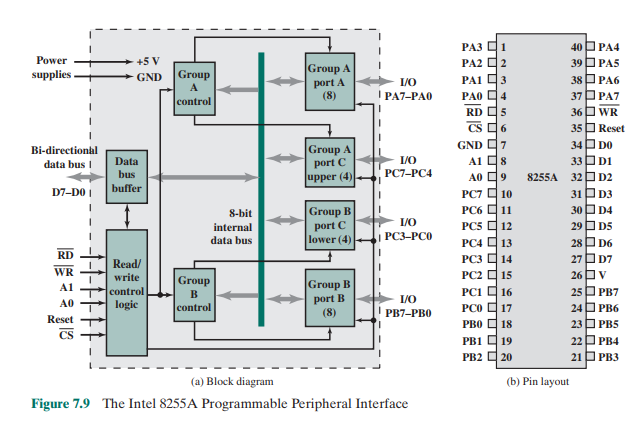
8255A data outputs are enabled onto the system data bus.

■ WR (Write Input): If this input line is a logical 0 and the CS input is a logical

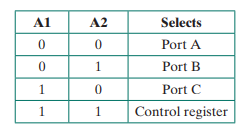
0, data are written to the 8255A from the system data bus.

■ RESET: The 8255A is placed into its reset state if this input line is a logical 1.

All peripheral ports are set to the input mode.



■ PA0–PA7, PB0–PB7, PC0–PC7: These signal lines are used as 8-bit I/O ports. They can be connected to peripheral devices. ■ A0, A1: The logical combination of these two input lines determine which internal register of the 8255A data are written to or read from. The right side of the block diagram of Figure 7.9a is the external interface of the 8255A. The 24 I/O lines are divided into three 8-bit groups (A, B, C). Each group can function as an 8-bit I/O port, thus providing connection for three peripheral devices. In addition, group C is subdivided into 4-bit groups (CA and CB), which may be used in conjunction with the A and B I/O ports. Configured in this manner, group C lines carry control and status signals. The left side of the block diagram is the internal interface to the microprocessor system bus. It includes an 8-bit bidirectional data bus (D0 through D7), used to transfer data between the microprocessor and the I/O ports and to transfer control information. The processor controls the 8255A by means of an 8-bit control register in the processor. The processor can set the value of the control register to specify a variety of operating modes and configurations. From the processor point of view, there is a control port, and the control register bits are set in the processor and then sent to the control port over lines D0–D7. The two address lines specify one of the three I/O ports or the control register, as follows:



Thus, when the processor sets both A1 and A2 to 1, the 8255A interprets the

8-bit value on the data bus as a control word. When the processor transfers an 8-bit

control word with line D7 set to 1 (Figure 7.10a), the control word is used to configure the operating mode of the 24 I/O lines. The three modes are:

■ Mode 0: This is the basic I/O mode. The three groups of eight external lines

function as three 8-bit I/O ports. Each port can be designated as input or output. Data may only be sent to a port if the port is defined as output, and data

may only be read from a port if the port is set to input.

■ Mode 1: In this mode, ports A and B can be configured as either input or

output, and lines from port C serve as control lines for A and B. The control

signals serve two principal purposes: “handshaking” and interrupt request.

Handshaking is a simple timing mechanism. One control line is used by the

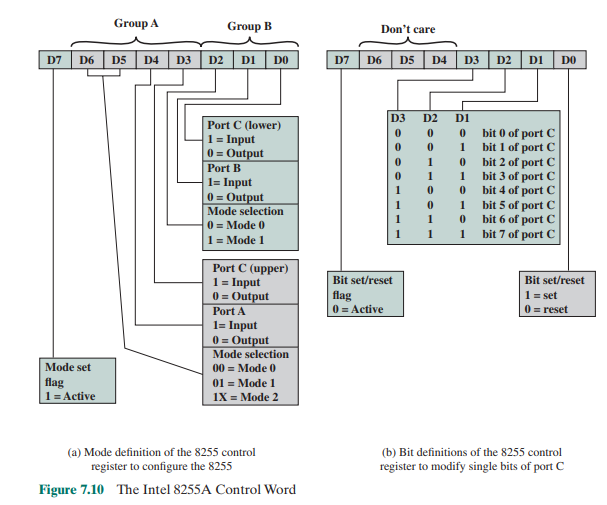
sender as a DATA READY line, to indicate when the data are present on the

I/O data lines. Another line is used by the receiver as an ACKNOWLEDGE,

indicating that the data have been read and the data lines may be cleared.

Another line may be designated as an INTERRUPT REQUEST line and tied

back to the system bus.



■ Mode 2: This is a bidirectional mode. In this mode, port A can be configured as either the input or output lines for bidirectional traffic on port B, with the port B lines providing the opposite direction. Again, port C lines are used for control signaling.

When the processor sets D7 to 0 (Figure 7.10b), the control word is used to program the bit values of port C individually. This feature is rarely used.

**keyboard/display example**

Because the 8255A is programmable via the

control register, it can be used to control a variety of simple peripheral devices.

Figure 7.11 illustrates its use to control a keyboard/display terminal. The keyboard

provides 8 bits of input. Two of these bits, SHIFT and CONTROL, have special

meaning to the keyboard-handling program executing in the processor. However,

this interpretation is transparent to the 8255A, which simply accepts the 8 bits of

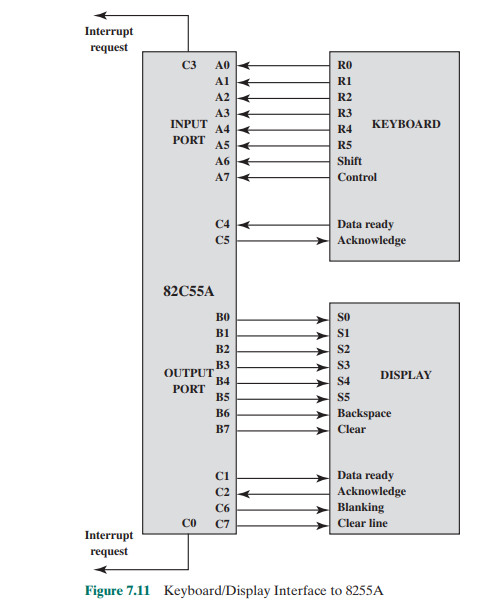
data and presents them on the system data bus. Two handshaking control lines are

provided for use with the keyboard.

The display is also linked by an 8-bit data port. Again, two of the bits have

special meanings that are transparent to the 8255A. In addition to two handshaking

lines, two lines provide additional control functions.



***7.5]DIRECT MEMORY ACCESS***

Drawbacks of Programmed and Interrupt-Driven I/O

Interrupt-driven I/O, though more efficient than simple programmed I/O, still

requires the active intervention of the processor to transfer data between memory

and an I/O module, and any data transfer must traverse a path through the processor. Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test

and service a device.

2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer (e.g., Figure 7.5).

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else. Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer rate. When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

**DMA Function**

DMA involves an additional module on the system bus. The DMA module

(Figure 7.12) is capable of mimicking the processor and, indeed, of taking over control of the system from the processor. It needs to do this to transfer data to and from

memory over the system bus. For this purpose, the DMA module must use the bus

only when the processor does not need it, or it must force the processor to suspend

operation temporarily. The latter technique is more common and is referred to as

cycle stealing, because the DMA module in effect steals a bus cycle.

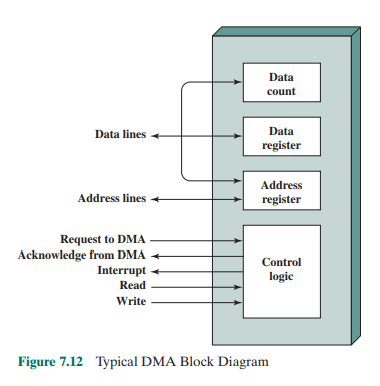
When the processor wishes to read or write a block of data, it issues a command

to the DMA module, by sending to the DMA module the following information:

■ Whether a read or 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 or write to, communicated on

the data lines and stored by the DMA module in its address register.

■ The number of words to be read or written, again communicated via the data

lines and stored in the data count register.

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data,

one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to

the processor. Thus, the processor is involved only at the beginning and end of the

transfer (Figure 7.4c).

Figure 7.13 shows where in the instruction cycle the processor may be suspended. In each case, the processor is suspended just before it needs to use the bus.

The DMA module then transfers one word and returns control to the processor.

Note that this is not an interrupt; the processor does not save a context and do

something else. Rather, the processor pauses for one bus cycle. The overall effect

is to cause the processor to execute more slowly. Nevertheless, for a multiple-word

I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

The DMA mechanism can be configured in a variety of ways. Some possibilities are shown in Figure 7.14. In the first example, all modules share the same system

bus. The DMA module, acting as a surrogate processor, uses programmed I/O to

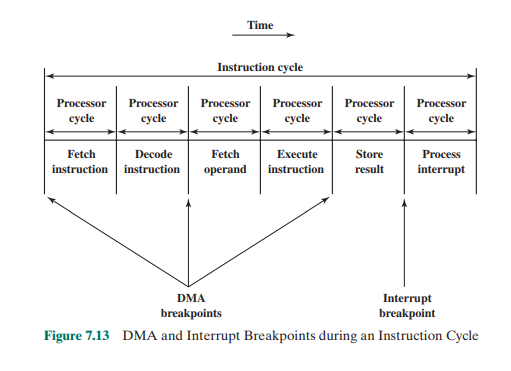
exchange data between memory and an I/O module through the DMA module. This

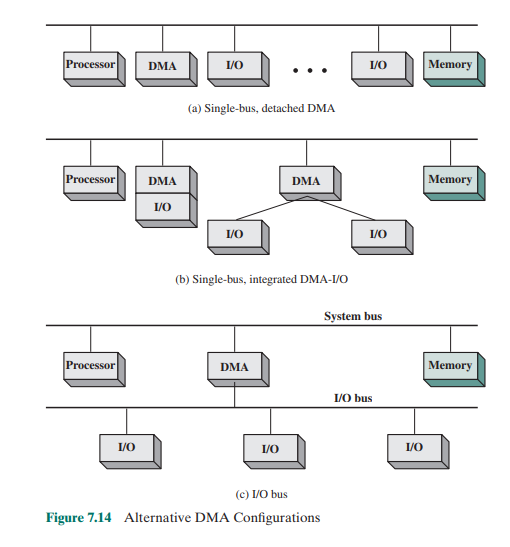
configuration, while it may be inexpensive, is clearly inefficient. As with processorcontrolled programmed I/O, each transfer of a word consumes two bus cycles.

The number of required bus cycles can be cut substantially by integrating the

DMA and I/O functions. As Figure 7.14b indicates, this means that there is a path

between the DMA module and one or more I/O modules that does not include





the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (Figure 7.14c). This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In both of these cases (Figures 7.14b and c), the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.

**Intel 8237A DMA Controller**

The Intel 8237A DMA controller interfaces to the 80 \* 86 family of processors and

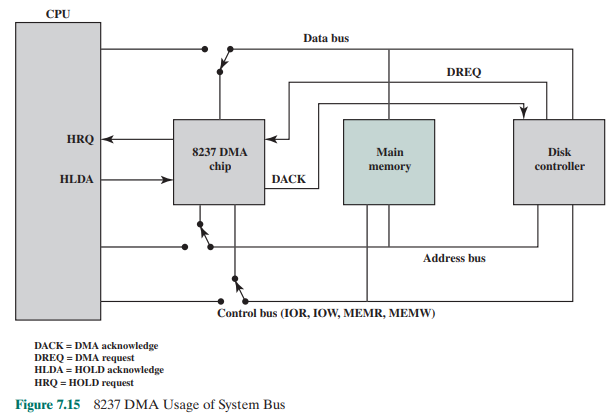
to DRAM memory to provide a DMA capability. Figure 7.15 indicates the location

of the DMA module. When the DMA module needs to use the system buses (data,

address, and control) to transfer data, it sends a signal called HOLD to the processor.

The processor responds with the HLDA (hold acknowledge) signal, indicating that

the DMA module can use the buses.



For example, if the DMA module is to transfer a block of data from memory to disk, it will do the following:

1. The peripheral device (such as the disk controller) will request the service of

DMA by pulling DREQ (DMA request) high.

2. The DMA will put a high on its HRQ (hold request), signaling the CPU

through its HOLD pin that it needs to use the buses.

3. The CPU will finish the present bus cycle (not necessarily the present instruction) and respond to the DMA request by putting high on its HDLA (hold

acknowledge), thus telling the 8237 DMA that it can go ahead and use the

buses to perform its task. HOLD must remain active high as long as DMA is

performing its task.

4. DMA will activate DACK (DMA acknowledge), which tells the peripheral

device that it will start to transfer the data.

5. DMA starts to transfer the data from memory to peripheral by putting the

address of the first byte of the block on the address bus and activating MEMR,

thereby reading the byte from memory into the data bus; it then activates IOW

to write it to the peripheral. Then DMA decrements the counter and increments the address pointer and repeats this process until the count reaches zero

and the task is finished.

6. After the DMA has finished its job it will deactivate HRQ, signaling the CPU

that it can regain control over its buses.

While the DMA is using the buses to transfer data, the processor is idle. Similarly, when the processor is using the bus, the DMA is idle. The 8237 DMA is known

as a fly-by DMA controller. This means that the data being moved from one location

to another does not pass through the DMA chip and is not stored in the DMA chip.

Therefore, the DMA can only transfer data between an I/O port and a memory address,

and not between two I/O ports or two memory locations. However, as explained subsequently, the DMA chip can perform a memory-to-memory transfer via a register.

The 8237 contains four DMA channels that can be programmed independently, and any one of the channels may be active at any moment. These channels are numbered 0, 1, 2, and 3.

The 8237 has a set of five control/command registers to program and control

DMA operation over one of its channels (Table 7.2):

■ Command: The processor loads this register to control the operation of

the DMA. D0 enables a memory- to- memory transfer, in which channel 0 is

used to transfer a byte into an 8237 temporary register and channel 1 is used

to transfer the byte from the register to memory. When memory- to- memory

is enabled, D1 can be used to disable increment/decrement on channel 0

so that a fixed value can be written into a block of memory. D2 enables or

disables DMA.

■ Status: The processor reads this register to determine DMA status. Bits

D0–D3 are used to indicate if channels 0–3 have reached their TC (terminal

count). Bits D4–D7 are used by the processor to determine if any channel has

a DMA request pending.

■ Mode: The processor sets this register to determine the mode of operation of

the DMA. Bits D0 and D1 are used to select a channel. The other bits select

various operation modes for the selected channel. Bits D2 and D3 determine

if the transfer is from an I/O device to memory (write) or from memory to

I/O (read), or a verify operation. If D4 is set, then the memory address register and the count register are reloaded with their original values at the end of

a DMA data transfer. Bits D6 and D7 determine the way in which the 8237 is

used. In single mode, a single byte of data is transferred. Block and demand

modes are used for a block transfer, with the demand mode allowing for

premature ending of the transfer. Cascade mode allows multiple 8237s to be

cascaded to expand the number of channels to more than 4.

■ Single Mask: The processor sets this register. Bits D0 and D1 select the channel. Bit D2 clears or sets the mask bit for that channel. It is through this register that the DREQ input of a specific channel can be masked (disabled) or

unmasked (enabled). While the command register can be used to disable the

whole DMA chip, the single mask register allows the programmer to disable

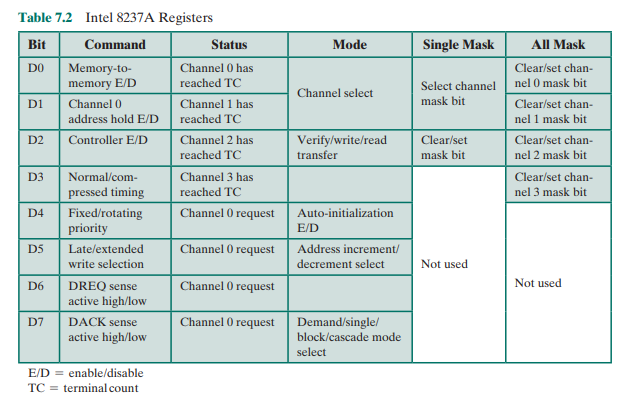
or enable a specific channel.

■ All Mask: This register is similar to the single mask register except that all four

channels can be masked or unmasked with one write operation.

In addition, the 8237A has eight data registers: one memory address register

and one count register for each channel. The processor sets these registers to indicate the location of size of main memory to be affected by the transfers.



***7.7]I/O CHANNELS AND PROCESSORS***

**The Evolution of the I/O Function**

As computer systems have evolved, there has been a pattern of increasing complexity and sophistication of individual components. Nowhere is this more evident than

in the I/O function. We have already seen part of that evolution. The evolutionary

steps can be summarized as follows:

1. The CPU directly controls a peripheral device. This is seen in simple

microprocessor-controlled devices.

2. A controller or I/O module is added. The CPU uses programmed I/O without

interrupts. With this step, the CPU becomes somewhat divorced from the specific details of external device interfaces

3. The same configuration as in step 2 is used, but now interrupts are employed.

The CPU need not spend time waiting for an I/O operation to be performed,

thus increasing efficiency.

4. The I/O module is given direct access to memory via DMA. It can now move

a block of data to or from memory without involving the CPU, except at the

beginning and end of the transfer.

5. The I/O module is enhanced to become a processor in its own right, with a

specialized instruction set tailored for I/O. The CPU directs the I/O processor

to execute an I/O program in memory. The I/O processor fetches and executes

these instructions without CPU intervention. This allows the CPU to specify a

sequence of I/O activities and to be interrupted only when the entire sequence

has been performed.

6. The I/O module has a local memory of its own and is, in fact, a computer in its

own right. With this architecture, a large set of I/O devices can be controlled,

with minimal CPU involvement. A common use for such an architecture has

been to control communication with interactive terminals. The I/O processor

takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of

I/O-related tasks, improving performance. With the last two steps (5–6), a major

change occurs with the introduction of the concept of an I/O module capable of

executing a program. For step 5, the I/O module is often referred to as an I/O

channel. For step 6, the term I/O processor is often used. However, both terms are

on occasion applied to both situations. In what follows, we will use the term I/O

channel.

**Characteristics of I/O Channels**

The I/O channel represents an extension of the DMA concept. An I/O channel

has the ability to execute I/O instructions, which gives it complete control over

I/O operations. In a computer system with such devices, the CPU does not execute

I/O instructions. Such instructions are stored in main memory to be executed by a

special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O

transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage,

priority, and actions to be taken for certain error conditions. The I/O channel follows

these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure 7.18. A

selector channel controls multiple high- speed devices and, at any one time, is

dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device, or a small set of

devices, is handled by a controller, or I/O module, that is much like the I/O modules we have been discussing. Thus, the I/O channel serves in place of the CPU

in controlling these I/O controllers. A multiplexor channel can handle I/O with

multiple devices at the same time. For low- speed devices, a byte multiplexor

accepts or transmits characters as fast as possible to multiple devices. For example,

the resultant character stream from three devices with different rates and individual streams A1A2A3A4 c , B1B2B3B4 c , and C1C2C3C4 c might be

A1B1C1A2C2A3B2C3A4, and so on. For high- speed devices, a block multiplexor

interleaves blocks of data from several devices.

