Module

3

Embedded Systems I/O

Lesson

15

Interrupts

Instructional Objectives

After going through this lesson the student would learn

- Interrupts
- Interrupt Service Subroutines
- Polling
    - Priority Resolving
    - Daisy Chain Interrupts
- Interrupt Structure in 8051 Microcontroller
- Programmable Interrupt Controller

Pre-Requisite

Digital Electronics, Microprocessors

15 Introduction

Real Time Embedded System design requires that I/O devices receive servicing in an efficient manner so that large amounts of the total system tasks can be assumed by the processor with little or no effect on throughput. The most common method of servicing such devices is the **polled** approach. This is where the processor must test each device in sequence and in effect "ask" each one if it needs servicing. It is easy to see that a large portion of the main program is looping through this continuous polling cycle and that such a method would have a serious, detrimental effect on system throughput, thus, limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices. A more desirable method would be one that would allow the microprocessor to be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete, however, the processor would resume exactly where it left off. This can be effectively handled by **interrupts.**

A signal informing a program or a device connected to the processor that an event has occurred. When a processor receives an interrupt signal, it takes a specified action depending on the priority and importance of the entity generating the signal. Interrupt signals can cause a program to suspend itself temporarily to service the interrupt by branching into another program called Interrupt Service Subroutines (ISS) for the specified device which has caused the interrupt.

Types of Interrupts

Interrupts can be broadly classified as

- Hardware Interrupts
  These are interrupts caused by the connected devices.
- Software Interrupts
  These are interrupts deliberately introduced by software instructions to generate user defined exceptions
- Trap

These are interrupts used by the processor alone to detect any exception such as divide by zero

Depending on the service the interrupts also can be classified as

- Fixed interrupt
    - Address of the ISR built into microprocessor, cannot be changed
    - Either ISR stored at address or a jump to actual ISR stored if not enough bytes available
- Vectored interrupt
    - Peripheral must provide the address of the ISR
    - Common when microprocessor has multiple peripherals connected by a system bus
- Compromise between fixed and vectored interrupts
    - One interrupt pin
    - Table in memory holding ISR addresses (maybe 256 words)
    - Peripheral doesn't provide ISR address, but rather index into table
        - Fewer bits are sent by the peripheral
        - Can move ISR location without changing peripheral

Maskable vs. Non-maskable interrupts

- Maskable: programmer can set bit that causes processor to ignore interrupt
    - This is important when the processor is executing a time-critical code
- Non-maskable: a separate interrupt pin that can't be masked
    - Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory

Example: Interrupt Driven Data Transfer (Fixed Interrupt)

Fig.15.1(a) shows the block diagram of a system where it is required to read data from a input port P1, modify (*according to some given algorithm*) and send to port P2. The input port generates data at a very slow pace. There are two ways to transfer data

(a) The processor waits till the input is ready with the data and performs a read operation from P1 followed by a write operation to P2. This is called **Programmed Data Transfer** (b) The other option is when the input/output device is slow then the device whenever is ready interrupts the microprocessor through an *Int* pin as shown in Fig.15.1. The processor which may be otherwise busy in executing another program (main program here) after receiving the interrupts calls an Interrupt Service Subroutine (ISR) to accomplish the required data transfer. This is known as **Interrupt Driven Data Transfer.**

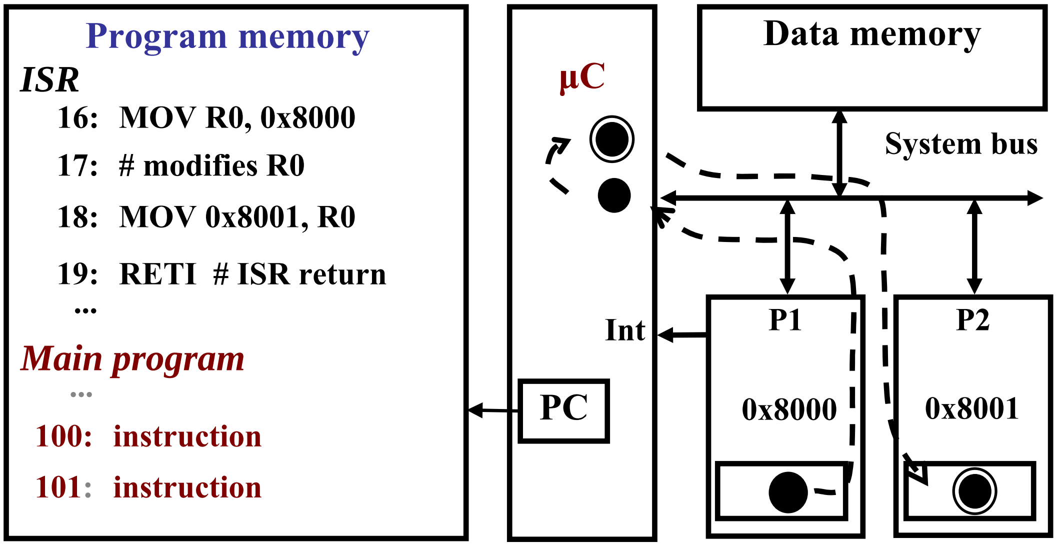


Fig: 15.1(a) The Interrupt Driven Data Transfer

PC-Program counter, P1-Port 1 P2-Port 2, μC-Microcontroller

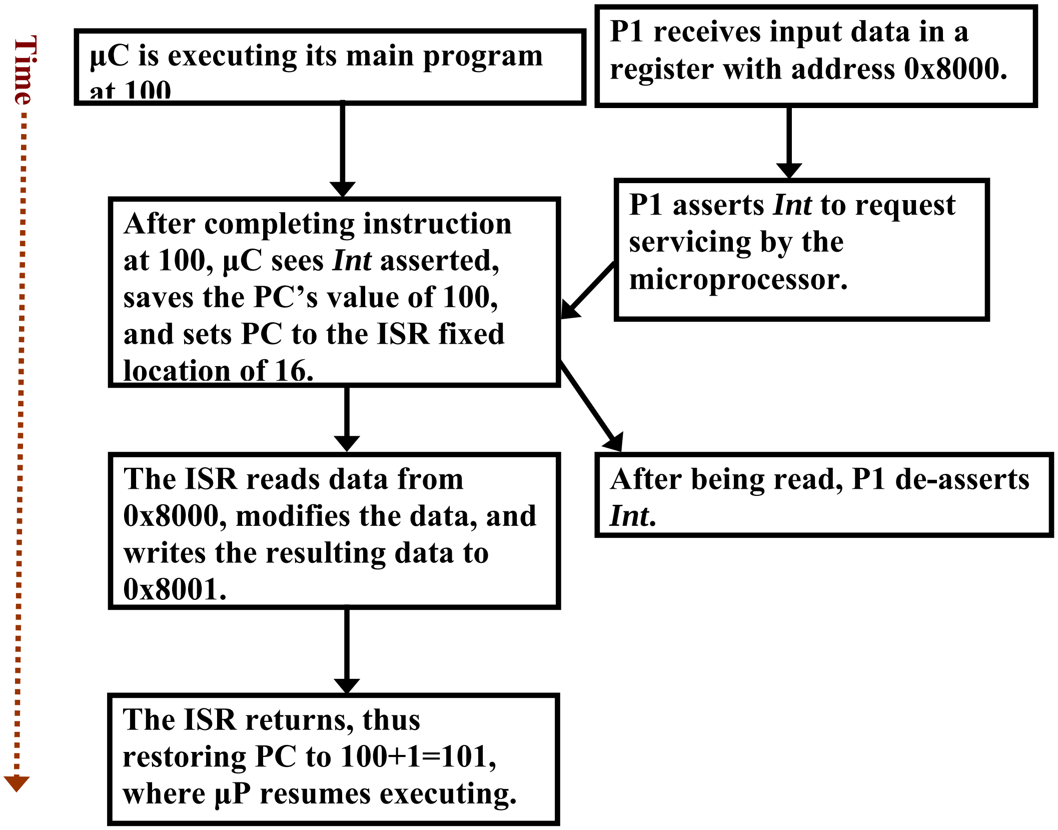

Time
μC is executing its main program at 100
P1 receives input data in a register with address 0x8000.
P1 asserts Int to request servicing by the microprocessor.
After completing instruction at 100, μC sees Int asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.
The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
After being read, P1 de-asserts Int.
The ISR returns, thus restoring PC to 100+1=101, where μP resumes executing.

Fig. 15.1(b) Flow chart for Interrupt Service

Fig.15.1(b) describes the sequence of action taking place after the Port P1 is ready with the data.

Example: Interrupt Driven Data Transfer (Vectored Interrupt)

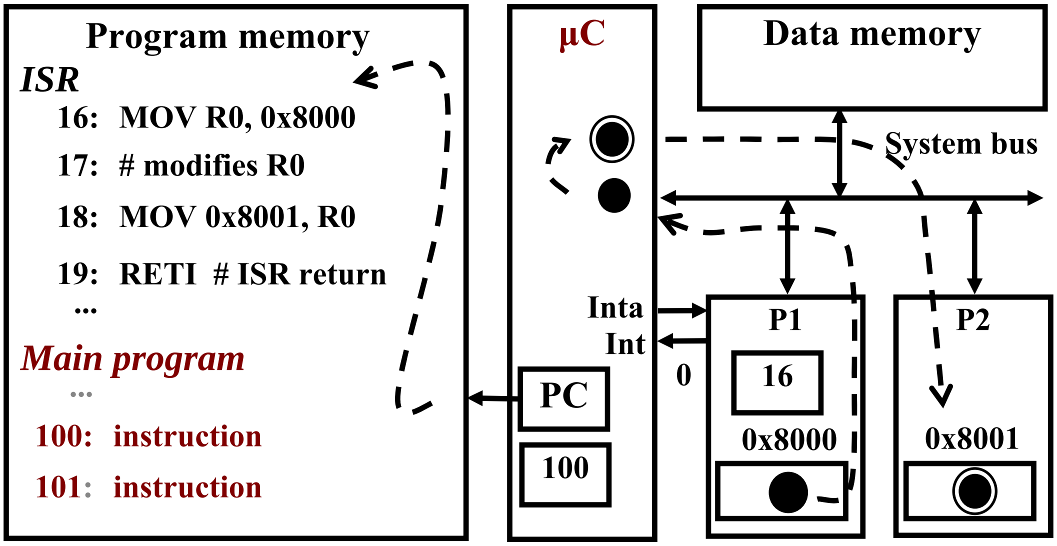
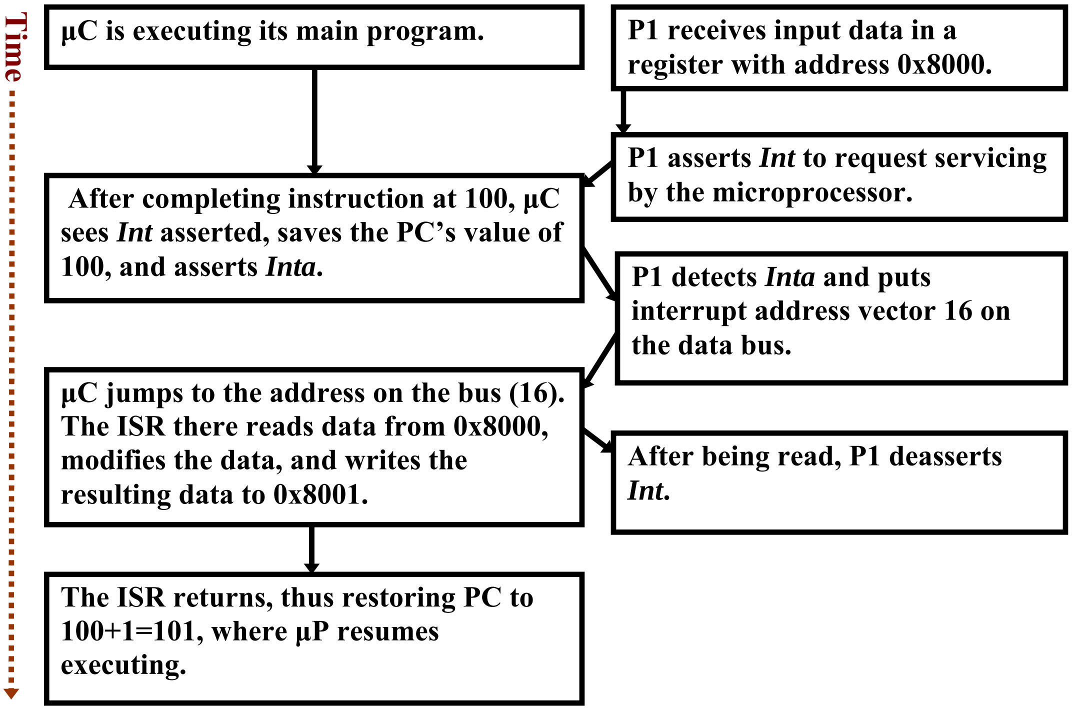


Fig. 15.2(a)



µC is executing its main program.

P1 receives input data in a register with address 0x8000.

P1 asserts *Int* to request servicing by the microprocessor.

After completing instruction at 100, µC sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*.

P1 detects *Inta* and puts interrupt address vector 16 on the data bus.

µC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

After being read, P1 deasserts *Int*.

The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Fig. 15.2(b) Vectored Interrupt Service

Interrupts in a Typical Microcontroller (say 8051)

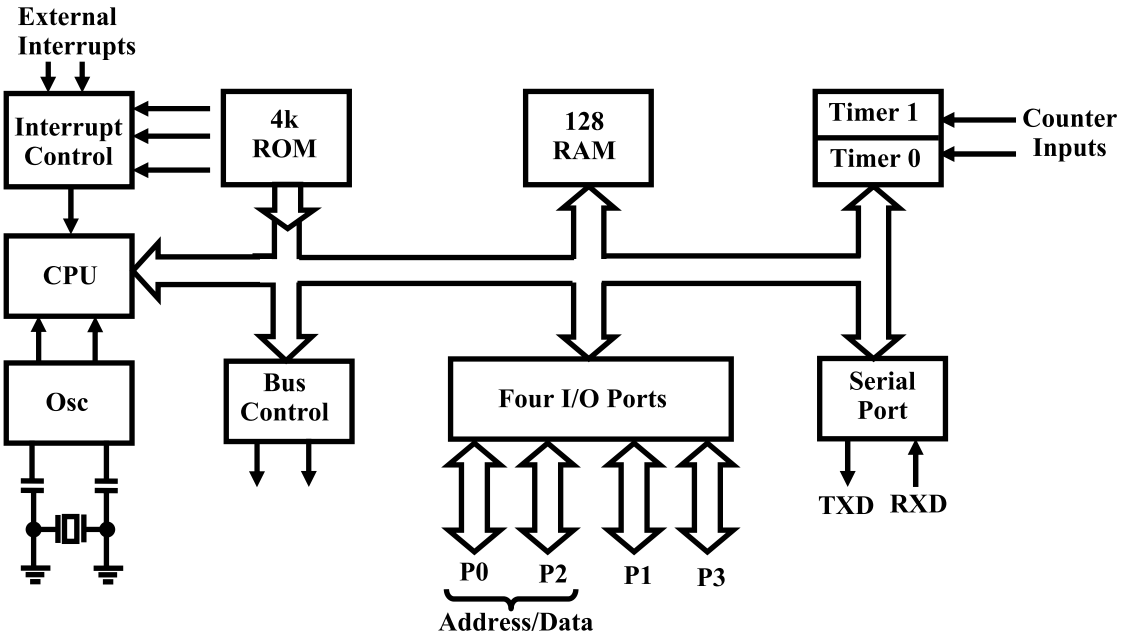


Fig. 15.3 The 8051 Architecture

The 8051 has 5 interrupt sources: 2 external interrupts, 2 timer interrupts, and the serial port interrupt.

These interrupts occur because of

1. timers overflowing
2. receiving character via the serial port
3. transmitting character via the serial port
4. Two "external events"

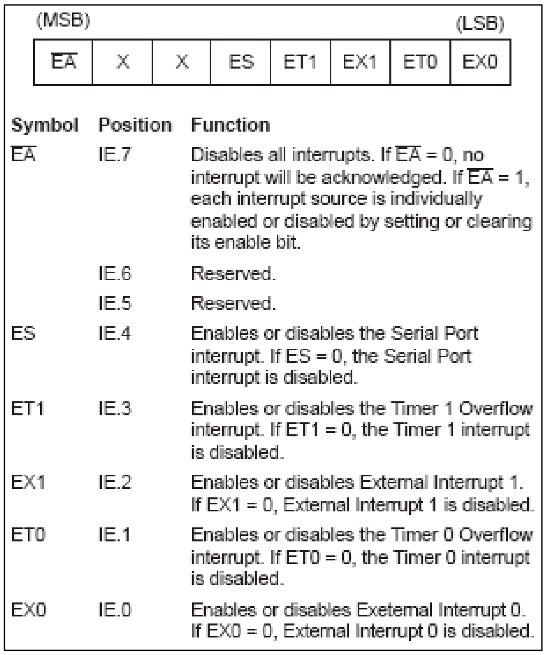
Interrupt Enables

Each interrupt source can be individually enabled or disabled by setting or clearing a bit in a Special Function Register (SFR) named IE (Interrupt Enable). This register also contains a global disable bit, which can be cleared to disable all interrupts at once.

Interrupt Priorities

Each interrupt source can also be individually programmed to one of two priority levels by setting or clearing a bit in the SFR named IP (Interrupt Priority). A low-priority interrupt can be interrupted by a high-priority interrupt, but not by another low-priority interrupt. A high-priority interrupt can't be interrupted by any other interrupt source. If two interrupt requests of different priority levels are received simultaneously, the request of higher priority is serviced. If interrupt requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence. In operation, all the interrupt flags are latched into the interrupt control system during State 5 of every machine cycle. The samples are polled during the following machine cycle. If the flag for an enabled interrupt is found to be set (1), the interrupt system generates a CALL to the appropriate location in Program Memory, unless some other condition blocks the interrupt. Several conditions can block an interrupt, among them that an interrupt of equal or higher priority level is already in progress. The hardware-generated CALL causes the contents of the Program Counter to be pushed into the stack, and reloads the PC with the beginning address of the service routine.

Interrupt Enable(IE) Register

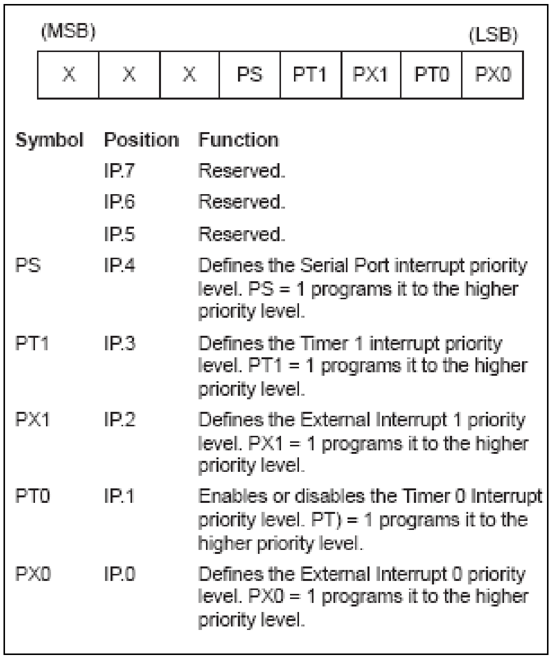


(MSB) (LSB)

$\overline{EA}$	X	X	ES	ET1	EX1	ET0	EX0

Symbol	Position	Function
$\overline{EA}$	IE.7	Disables all interrupts. If $\overline{EA}$ = 0, no interrupt will be acknowledged. If $\overline{EA}$ = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
	IE.6	Reserved.
	IE.5	Reserved.
ES	IE.4	Enables or disables the Serial Port interrupt. If ES = 0, the Serial Port interrupt is disabled.
ET1	IE.3	Enables or disables the Timer 1 Overflow interrupt. If ET1 = 0, the Timer 1 interrupt is disabled.
EX1	IE.2	Enables or disables External Interrupt 1. If EX1 = 0, External Interrupt 1 is disabled.
ET0	IE.1	Enables or disables the Timer 0 Overflow interrupt. If ET0 = 0, the Timer 0 interrupt is disabled.
EX0	IE.0	Enables or disables Exetemal Interrupt 0. If EX0 = 0, External Interrupt 0 is disabled.

terrupt Priority (IP) Regist



(MSB) (LSB)

X	X	X	PS	PT1	PX1	PT0	PX0

Symbol	Position	Function
	IP.7	Reserved.
	IP.6	Reserved.
	IP.5	Reserved.
PS	IP.4	Defines the Serial Port interrupt priority level. PS = 1 programs it to the higher priority level.
PT1	IP.3	Defines the Timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level. PX1 = 1 programs it to the higher priority level.
PT0	IP.1	Enables or disables the Timer 0 Interrupt priority level. PT) = 1 programs it to the higher priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level. PX0 = 1 programs it to the higher priority level.

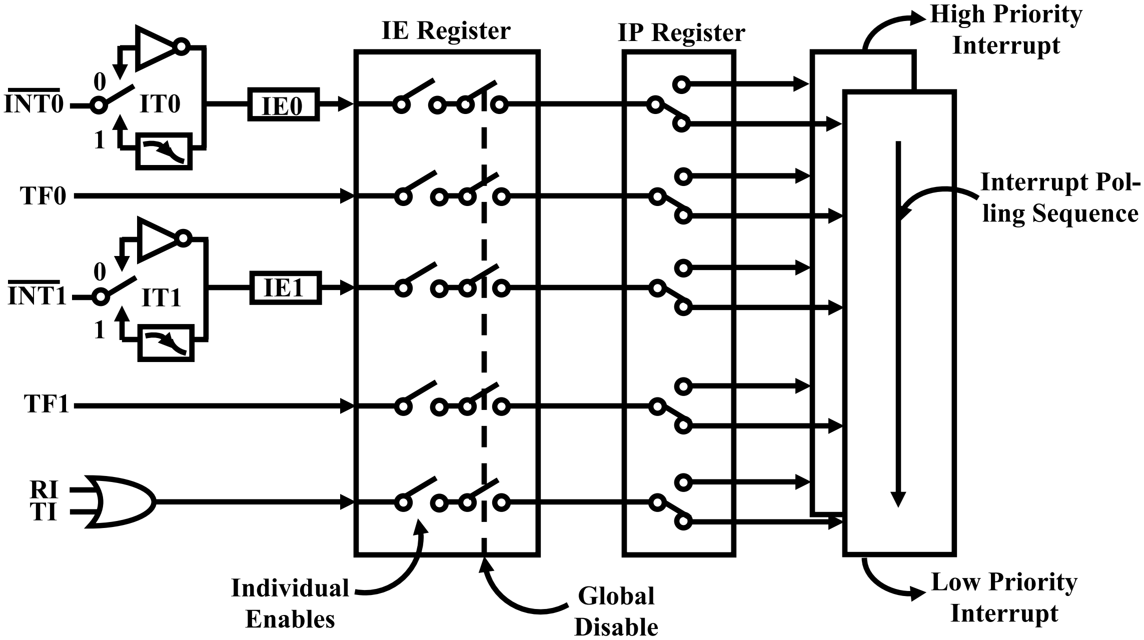


Fig. 15.4 8051 Interrupt Control System

$\overline{\text{INT0}}$: External Interrupt 0

$\overline{\text{INT0}}$: External Interrupt 1

TF0: Timer 0 Interrupt
TF1: Timer 1 Interrupt
RI,TI: Serial Port Receive/Transmit Interrupt

The service routine for each interrupt begins at a fixed location (fixed address interrupts). Only the Program Counter (PC) is automatically pushed onto the stack, not the Processor Status Word (which includes the contents of the accumulator and flag register) or any other register. Having only the PC automatically saved allows the programmer to decide how much time should be spent saving other registers. This enhances the interrupt response time, albeit at the expense of increasing the programmer's burden of responsibility. As a result, many interrupt functions that are typical in control applications toggling a port pin for example, or reloading a timer, or unloading a serial buffer can often be completed in less time than it takes other architectures to complete.

Interrupt Number	Interrupt Vector Address	Description
0	0003h	EXTERNAL 0
1	000Bh	TIMER/COUNTER 0
2	0013h	EXTERNAL 1
3	001Bh	TIMER/COUNTER 1
4	0023h	SERIAL PORT

Simultaneously occurring interrupts are serviced in the following order:

1. External 0 Interrupt
2. Timer 0 Interrupt
3. External 1 Interrupt
4. Timer 1 Interrupt
5. Serial Interrupt

The Bus Arbitration

When there are more than one device need interrupt service then they have to be connected in specific manner. The processor responds to each one of them. This is called **Arbitration**. The method can be divided into following

- Priority Arbiter
- Daisy Chain Arbiter

Priority Arbiter

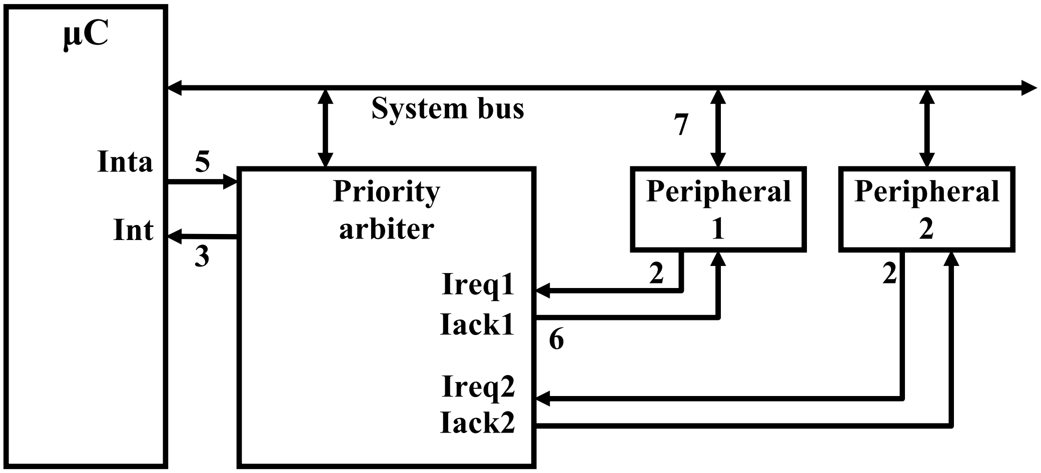


Fig. 15.5 The Priority Arbitration

Let us assume that the Priority of the devices are Device1 > Device 2 …

1. The Processor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Processor stops executing its program and stores its state.
5. Processor asserts *Inta*.
6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus
8. Processor jumps to the address of ISR read from data bus, ISR executes and returns(and completes handshake with arbiter).

Thus in case of simultaneous interrupts the device with the highest priority will be served.

Daisy Chain Interrupts

In this case the peripherals needing interrupt service are connected in a chain as shown in Fig.15.6. The requests are chained and hence any device interrupting shall be transmitted to the CPU in a chain.

Let us assume that the Priority of the devices are Device1 > Device 2 …

1. The Processor is executing its program.
2. Any Peripheral needs servicing asserts *Req out*. This *Req out* goes to the *Req in* of the subsequent device in the chain
3. Thus the peripheral nearest to the μC asserts *Int*.
4. The processor stops executing its program and stores its state.
5. Processor asserts *Inta* the nearest device.
6. The *Inta* passes through the chain till it finds a flag which is set by the device which has generated the interrupt.
7. The interrupting device sends the Interrupt Address Vector to the processor for its interrupt service subroutine.

8. The processor jumps to the address of ISR read from data bus, ISR executes and returns.
9. The flag is reset.

The processor now check for the next device which has interrupted simultaneously.

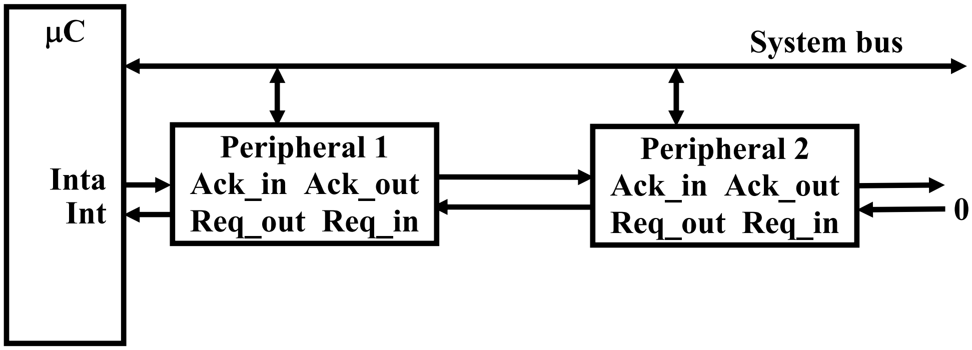


Fig. 15.6 The Daisy Chain Arbitration

In this case The device nearest to the processor has the highest priority

The service to the subsequent stages is interrupted if the chain is broken at one place.

Handling a number of Interrupts by Intel 8259 Programmable Interrupt Controller

The Programmable Interrupt Controller (PlC) functions as an overall manager in an Interrupt-Driven system. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

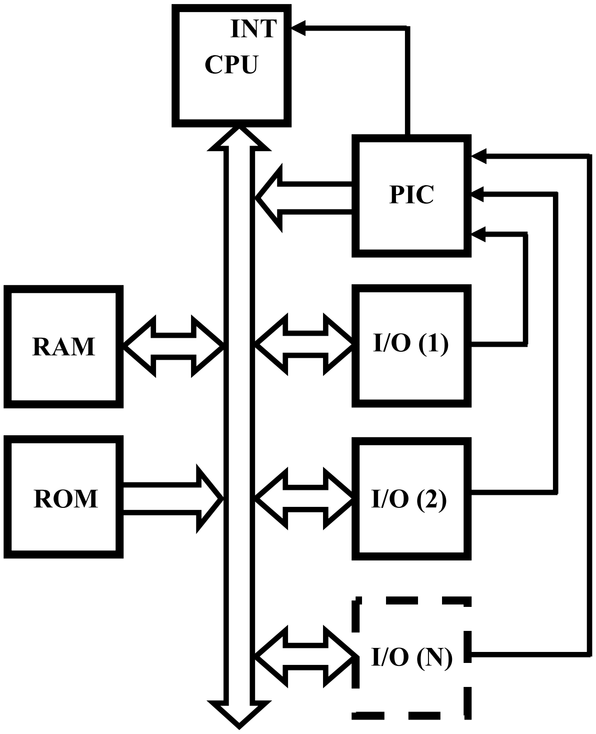


Fig. 15.7 Handling a number of interrupts

Each peripheral device or structure usually has a special program or "routine" that is associated with its specific functional or operational requirements; this is referred to as a "service routine". The PlC, after issuing an interrupt to the CPU, must somehow input information into the CPU that can point (vector) the Program Counter to the service routine associated with the requesting device.

The PIC manages eight levels of requests and has built-in features for expandability to other PIC (up to 64 levels). It is programmed by system software as an I/O peripheral. The priority modes can be changed or reconfigured dynamically at any time during main program operation.

Interrupt Request Register (IRR) and In-Service Register (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (lRR) and the In- Service Register (lSR). The IRR is used to indicate all the interrupt levels which are requesting service, and the ISR is used to store all the interrupt levels which are currently being serviced.

Priority Resolver

This logic block determines the priorities of the bits set in the lRR. The highest priority is selected and strobed into the corresponding bit of the lSR during the INTA sequence.

Interrupt Mask Register (IMR)

The lMR stores the bits which disable the interrupt lines to be masked. The IMR operates on the output of the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

Data Bus Buffer

This 3-state, bidirectional 8-bit buffer is used to interface the PIC to the System Data Bus. Control words and status information are transferred through the Data Bus Buffer.

Read/Write Control Logic

The function of this block is to accept output commands from the CPU. It contains the Initialization Command Word (lCW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the PIC to be transferred onto the Data Bus. This function block stores and compares the IDs of all PICs used in the system. The associated three I/O pins (CAS0- 2) are outputs when the 8259 is used as a master and are inputs when the 8259 is used as a slave. As a master, the 8259 sends the ID of the interrupting slave device onto the CAS0 - 2 lines. The slave, thus selected will send its preprogrammed subroutine address onto the Data Bus during the next one or two consecutive INTA pulses.

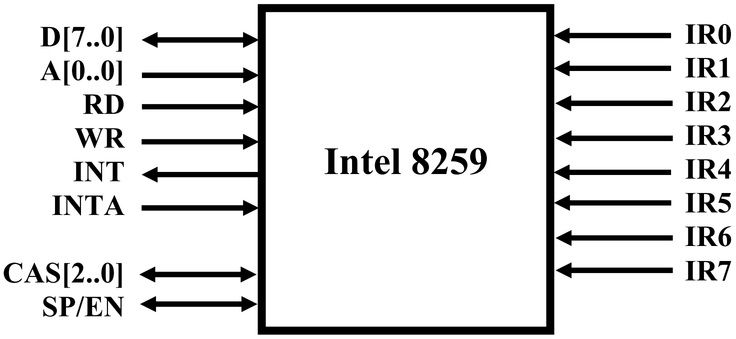


Fig. 15.8 The 8259 Interrupt Controller

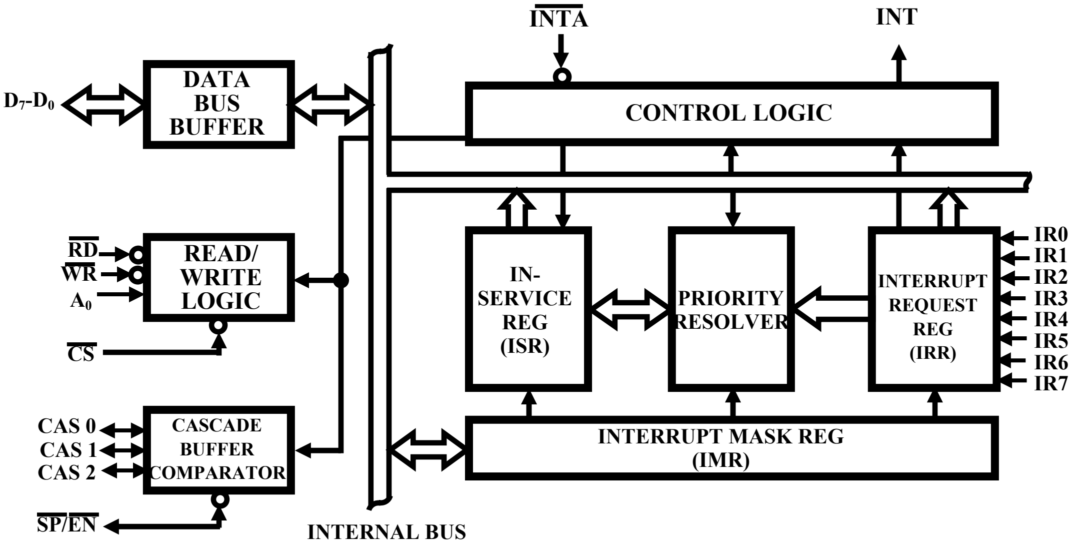


Fig. 15.9 The Functional Block Diagram

Table of Signals of the PIC

Signal	Description
D[7..0]	These wires are connected to the system bus and are used by the microprocessor to write or read the internal registers of the 8259.
A[0..0]	This pin acts in conjunction with WR/RD signals. It is used by the 8259 to decipher various command words the microprocessor writes and status the microprocessor wishes to read.
WR	When this write signal is asserted, the 8259 accepts the command on the data line, i.e., the microprocessor writes to the 8259 by placing a command on the data lines and asserting this signal.
RD	When this read signal is asserted, the 8259 provides on the data lines its status, i.e., the microprocessor reads the status of the 8259 by asserting this signal and reading the data lines.
INT	This signal is asserted whenever a valid interrupt request is received by the 8259, i.e., it is used to interrupt the microprocessor.

INTA	This signal, is used to enable 8259 interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the microprocessor.
IR 0,1,2,3,4,5,6,7	An interrupt request is executed by a peripheral device when one of these signals is asserted.
CAS[2..0]	These are cascade signals to enable multiple 8259 chips to be chained together.
SP/EN	This function is used in conjunction with the CAS signals for cascading purposes.

Fig.15.10 shows the daisy chain connection of a number of PICs. The extreme right PIC interrupts the processor. In this figure the processor can entertain up to 24 different interrupt requests. The SP/EN signal has been connected to Vcc for the master and grounded for the slaves.

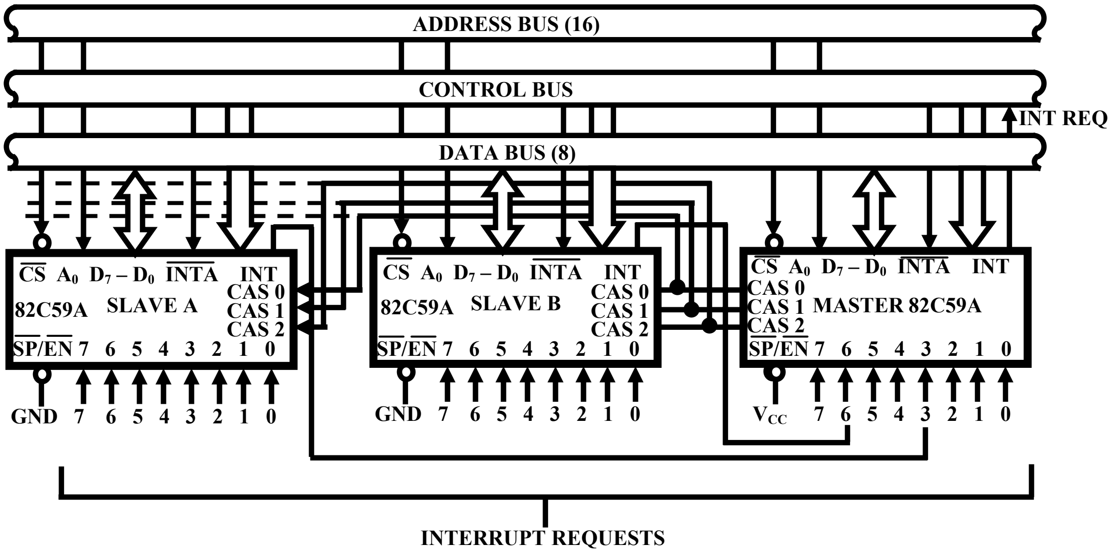


Fig. 15.10 Nested Connection of Interrupts

Software Interrupts

These are initiated by the program by specific instructions. On encountering such instructions the CPU executes an Interrupt service subroutine.

Conclusion

In this chapter you have learnt about the Interrupts and the Programmable Interrupt Controller. Different methods of interrupt services such as Priority arbitration and Daisy Chain arbitration have been discussed. In real time systems the interrupts are used for specific cases and the time of execution of these Interrupt Service Subroutines are almost fixed. Too many interrupts are not encouraged in real time as it may severely disrupt the services. Please look at problem no.1 in the exercise.

Most of the embedded processors are equipped with an interrupt structure. Rarely there is a need to use a PIC. Some of the entry level microcontrollers do not have an inbuilt exception handler called *trap*. The *trap* is also an interrupt which is used to handle some extreme processor conditions such as divide by 0, overflow etc.

Question Answers

Q1. A computer system has three devices whose characteristics are summarized in the following table:

Device	**Service Time**	**Interrupt Frequency**	**Allowable Latency**
D1	150µs	1/(800µs)	50µs
D2	50µs	1/(1000µs)	50µs
D3	100µs	1/(800µs)	100µs

Service time indicates how long it takes to run the interrupt handler for each device. The maximum time allowed to elapse between an interrupt request and the start of the interrupt handler is indicated by allowable latency. If a program P takes 100 seconds to execute when interrupts are disabled, how long will P take to run when interrupts are enabled?

Ans:

The CPU time taken to service the interrupts must be found out. Let us consider Device 1. It takes 400 µs to execute and occurs at a frequency of 1/(800µs) (1250 times a second). Consider a time quantum of 1 unit.

The Device 1 shall take (150+50)/800= 1/4 unit
The Device 2 shall take (50+50)/1000=1/10 unit
The Device 3 shall take (100+100)/800=1/4 unit

In one unit of real time the cpu time taken by all these devices is (1/4+1/10+1/4) = 0.6 units

The cpu idle time 0.4 units which can be used by the Program P. For 100 seconds of CPU time the Real Time required will be 100/0.4= 250 seconds

Q.2 What is TRAP?

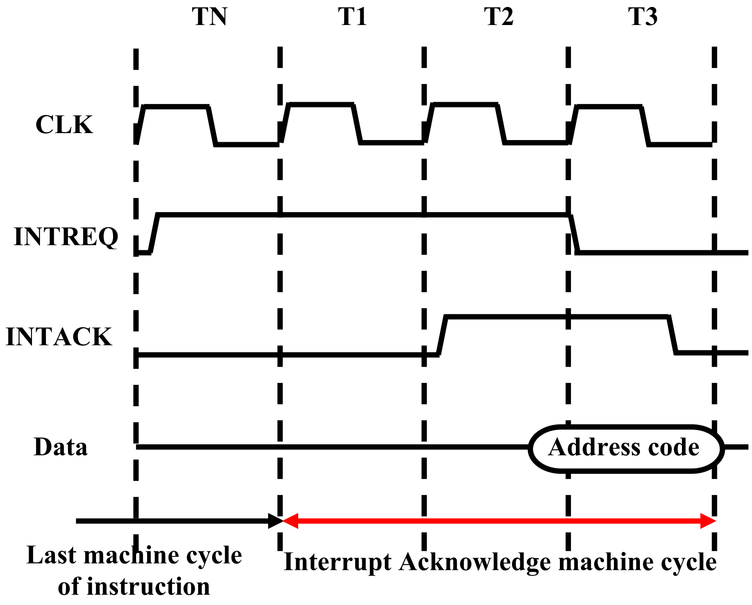
Ans:

The term trap denotes a programmer initiated and expected transfer of control to a special handler routine. In many respects, a trap is nothing more than a specialized subroutine call. Many texts refer to traps as software interrupts. Traps are usually unconditional; that is, when you execute an *Interrupt* instruction, control always transfers to the procedure associated with the trap. Since traps execute via an explicit instruction, it is easy to determine exactly which instructions in a program will invoke a trap handling routine.

Q.3. Discuss about the Interrupt Acknowledge Machine Cycle.

Ans:

For vectored interrupts the processor expects the address from the external device. Once it receives the interrupt it starts an Interrupt acknowledge cycle as shown in the figure. In the figure TN is the last clock state of the previous instruction immediately after which the processor checks the status of the Intr pin which has already become high by the external device. Therefore the processor starts an INTA cycle in which it brings the interrupt vector through the data lines. If the data lines arte 8-bits and the address required is 16 bits there will be two I/O read. If the interrupt vector is a number which will be vectored to a look up table then only 8-bits are required and hence one I/O read will be there.


TN
T1
T2
T3
CLK
INTREQ
INTACK
Data
Address code
Last machine cycle of instruction
Interrupt Acknowledge machine cycle