

UNIT – 4: 8051 INTERRUPT (PROGRAMMING & SERIAL COMMUNICATION WITH 8051):

Definition of an interrupt, types of interrupts, Timers and Counter programming with interrupts in assembly. 8051 Serial Communication: Data communication, Basics of Serial Data Communication, 8051 Serial Communication.

Interrupt Programming:

Interrupts vs. Polling Method:

A single microcontroller can serve several devices. There are two ways to do that: interrupts and polling.

In the interrupt method, whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device. The program which is associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler.

In polling method, the microcontroller continuously monitors the status of a given device; when the status conditions met, it performs the service. After that, it moves on to monitor the next device until each one is serviced.

In polling method there is no efficient use of microcontroller. Though in interrupt method not all devices can be serviced at the same time, each device can get the attention of microcontroller based on priority assigned to it. In polling method priority cannot be assigned since it checks all devices in a round-robin fashion. In interrupt method, microcontroller can also ignore (mask) a device request for service which is not possible in polling method. In polling method, microcontroller wastes its time by polling devices that do not need service. To save the time, interrupt method is employed.

For example, in timer programming, microcontroller waits till the TF flag is set to 1. In interrupt method, microcontroller will perform some useful task while the timer is running. It does not wait till the TF is set to 1. Once the TF flag is set to 1, timer generates interrupt.

Polling Vs Interrupts	
Polling	Interrupt
In Polling, The microcontroller continuously monitors the status of all devices in Round-robin manner. When the conditions are for a given device are met, it performs the service. After that, it moves on to monitor the next device until every one is serviced	In Interrupts, Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device
The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service	Interrupts serve only those devices which need service and hence are efficient
It is not possible to assign priority since it checks all devices in a round-robin fashion	Each device can get the attention of the microcontroller based on the assigned priority
Ignoring a device request is not possible	microcontroller can also ignore (mask) a device request for service

Interrupt service routine:

For every interrupt, there must be an interrupt service routine (ISR) or interrupt handler. When an interrupt is invoked, the microcontroller runs ISR. For every interrupt; there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

Interrupt Vector Table			
Interrupt	ISR ROM Address (Hex)	Pin	Flag Clearing
Reset	0000	9	Auto
External Hardware interrupt 0 (INT0)	0003	P3.2	Auto
Timer 0 Interrupt (TF0)	000B		Auto
External Hardware interrupt 1 (INT1)	0013	P3.3	Auto
Timer 1 Interrupt (TF1)	001B		Auto
Serial COM Interrupt (RI and TI)	0023		Programmer clears it.

Steps in executing an interrupt:

Upon activation of an interrupt, the microcontroller goes through the following steps:

1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack.
2. It also saves the current status of all the interrupts internally (i.e.: not on the stack)
3. It jumps to a fixed location in memory, called the interrupt vector table that holds the address of the ISR.
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC.

Types of Interrupts:

There are two types of interrupts: a.
External interrupts
b. Internal interrupts

8051 has three external interrupts and three internal interrupts. They are:

- i. Reset:** When the reset pin is activated, the 8051 jumps to address location 0000h.
- ii. Timer 0 and Timer 1 interrupt:** Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations 000BH and 001BH in the interrupt vector table belongs to timer 0 and timer 1 respectively.
- iii. INT0 and INT1:** Two interrupts are set aside for external hardware interrupts. Pin numbers 12 (P3.2) and 13 (P3.3) of Port 3 are INT0 and INT1 respectively. They are also referred as EX1 and EX2. Memory locations 0003H is assigned to INT0 and 0013H is assigned to INT1 in the interrupt vector table.
- iv. Serial Communication:** It has single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

From interrupt vector table, it is evident that limited number of bytes is set aside for each interrupt. For Reset interrupt only 3 bytes of location is allocated. For example, a total of 8 bytes from location 0003H to 000AH is set aside for INT0, 8 bytes from location 000BH to 0012H for Timer 0, 8 bytes from location 0013H to 001AH for INT1, 8 bytes from location 001BH to 0022H for timer 1. If the service routine for a given interrupt is short enough to fit in the memory space allocated to it, it is placed in the vector table; otherwise an LJMP instruction is placed in the vector table to point to the address of the ISR and rest of the bytes allocated to that interrupt are unused.

```
ORG 0          ; wake-up ROM reset location
LJMP MAIN ; by-pass int. vector table
; ---- the wake-up program
ORG          30H
MAIN:
....
END
```

Enabling and disabling an interrupt:

Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them. There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts.

Interrupt Enable (IE) register:



It is a bit addressable register.

EA: If 0, disables all interrupts, no interrupt is acknowledged. If 1, each interrupt source is individually enabled or disabled by setting or clearing individual bit.

--: Not implemented, reserved for future use.

ET2: Enables or disables timer 2 overflows or capture interrupt (8052) only.

ES: Enables or disables the serial port interrupt.

ET1: Enables or disables timer 1 overflow interrupt.

EX1: Enables or disables external interrupt 1 (INT1).

ET0: Enables or disables timer 0 overflow interrupt.

EX0: Enables or disables external interrupt 0 (INT0).

Show the instructions to (a) enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 (EX1), and (b) disable (mask) the timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

Solution:

(a) MOV IE, #10010110B; enable serial, timer 0, EX1 (b) CLR IE.1; mask (disable) timer 0 interrupt only (c) CLR IE.7; disable all interrupts.

Another way to perform the same manipulation is

SETB IE.7; EA=1, global enable

SETB IE.4; enable serial interrupt

SETB IE.1; enable Timer 0 interrupt

SETB IE.2; enable EX1

Programming Timer Interrupts:

Roll-over timer flag and interrupt:

The timer flag (TF) is raised when the timer rolls over. In polling TF, we have to wait until the TF is raised. The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and cannot do anything else. Using interrupts solves this problem and, avoids tying down the controller. If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other things until it is notified that the timer has rolled over.



Write a program that continuously get 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 μ s period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use timer 0 in mode 2 (auto reload). $TH0 = 100/1.085 \text{ us} = 92$

--upon wake-up go to main, avoid using memory allocated to Interrupt Vector Table ORG

0000H

LJMP MAIN ; by-pass interrupt vector table

--ISR for timer 0 to generate square wave

ORG 000BH ; Timer 0 interrupt vector table

CPL P2.1 ; toggle P2.1 pin

RETI

--The main program for initialization

ORG 0030H ; after vector table space

MAIN: MOV TMOD, #02H ; Timer 0, mode 2

MOV P0, #0FFH ; make P0 an input port

MOV TH0, #-92 ; TH0=A4H for -92

MOV IE, #82H ; IE=10000010 (bin) enable
Timer 0

SETB TR0 ; Start Timer 0

BACK: MOV A, P0 ; get data from P0

MOV P1, A ; issue it to P1

SJMP BACK ; keep doing it loop unless interrupted by

END TF0

Notice the following points in the above program:

1. Memory space allocated to the interrupt vector table should be avoided. Place all the initialization codes in memory starting at 30H. The LJMP instruction is the first instruction that the 8051 executes when it is powered up. LJMP redirects the controller away from the interrupt vector table.
2. The ISR for timer 0 is located starting at memory location 000BH since it is small enough to fit the address space allocated to this interrupt.
3. Enable timer 0 interrupt.
4. While the P0 data is brought in and issued to P1 continuously, whenever timer 0 is rolled over, the TF0 flag is raised and the microcontroller gets out of the BACK loop and goes to 000BH to execute the ISR associated with timer 0.
5. In the ISR for timer 0, notice that there is no need for a CLR TF0 instruction before the RETI instruction. This is because the 8051 clears the TF flag internally upon jumping to the interrupt vector table.

Write an assembly language program to create a square wave that has a high portion of 1085 μ s and a low portion of 15 μ s. Assume XTAL=11.0592MHz. Use timer 1.

Solution:

Since 1085 μ s is 1000×1.085 we need to use mode 1 of timer 1.

--upon wake-up go to main, avoid using memory allocated to Interrupt Vector Table ORG

```
0000H
    LJMP MAIN                ; by-pass int. vector table
;--ISR for timer 1 to generate square wave
    ORG 001BH                ; Timer 1 int. vector table
    LJMP ISR_T1              ; jump to ISR
;--The main program for initialization
    ORG 0030H                ; after vector table space
MAIN: MOV TMOD, #10H         ; Timer 1, mode 1
    MOV P0, #0FFH            ; make P0 an input port
    MOV TL1, #018H           ; TL1=18 low byte of -1000
    MOV TH1, #0FCH           ; TH1=FC high byte of -1000
    MOV IE, #88H             ; 10001000 enable Timer 1 int
    SETB TR1                 ; Start Timer 1
    BACK: MOV A, P0           ; get data from P0
    MOV P1, A                ; issue it to P1
    SJMP BACK                ; keep doing it
; Timer 1 ISR. Must be reloaded, not auto-reload
ISR_T1: CLR TR1              ; stop Timer 1
    MOV R2, #4               ; 2MC
    CLR P2.1                 ; P2.1=0, start of low portion
    HERE: DJNZ R2, HERE       ; 4x2 machine cycle 8MC
    MOV TL1, #18H            ; load T1 low byte value 2MC
    MOV TH1, #0FCH           ; load T1 high byte value 2MC
    SETB TR1                 ; starts timer1 1MC
    SETB P2.1                ; P2.1=1, back to high 1MC
    RETI                     ; return to main
    END
```

In the above program the low portion of the pulse is created by the 14 machine cycles (MC) where each MC = 1.085 μ s and $14 \times 1.085 \mu$ s = 15.19 μ s.

Programming the serial communication interrupt:

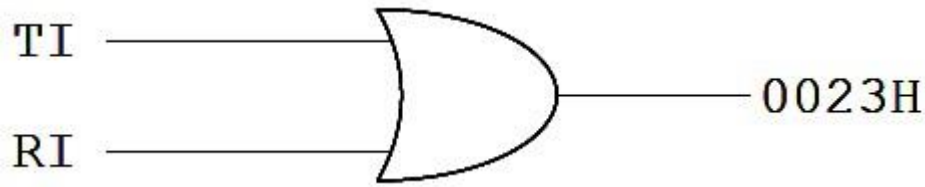
In the earlier topic, when data is transmitted or received serially, TI or RI flag was set which was monitored using JNB instruction. This method is known as polling method. Instead of this if interrupt is enabled due to serial communication, TI or RI flag need not be monitored. Once these flags are set, microcontroller will automatically generate interrupt.

RI and TI flags and interrupts:

TI is raised when the last bit of the framed data, the stop bit is transferred; indicating that the SBUF register is ready to transfer the next byte. RI is raised when the entire frame of data, including the stop bit is received. That

is when the SBUF register has a byte RI is raised to indicate that the received byte needs to be picked up before it is lost (overflow) by new incoming serial data. All the above concepts apply equally when using either polling or an interrupt. The only difference is in how the serial communication needs are served.

In the 8051 only one interrupt is set aside for serial communication. This interrupt is used to both send and receive data. If the interrupt bit in the IE register (IE.4) is enabled when RI or TI is raised, the 8051 gets interrupted and jumps to memory address location 0023H to execute the ISR. In that ISR we must examine the TI and RI flags to see which one caused the interrupt and respond accordingly.



Serial interrupt is invoked by TI or RI flags

Write a program in which the 8051 reads data from P1 and writes it to P2 continuously while giving a copy of it to the serial COM port to be transferred serially. Assume that XTAL=11.0592. Set the baud rate at 9600.

Solution:

```

      ORG 0000H
      LJMP MAIN
      ORG 23H
      LJMP SERIAL          ; jump to serial int ISR
      ORG 30H
MAIN:  MOV P1, #0FFH        ; make P1 an input port
      MOV TMOD, #20H        ; timer 1, auto reload

      MOV TH1, #0FDH        ; 9600 baud rate
      MOV SCON, #50H        ; 8-bit, 1 stop, ren enabled
      MOV IE, #10010000B    ; enable serial int.

      SETB TR1              ; start timer 1

BACK:  MOV A, P1            ; read data from port 1
      MOV SBUF, A           ; give a copy to SBUF

      MOV P2, A             ; send it to P2
      SJMP BACK            ; stay in loop indefinitely

; -----SERIAL PORT ISR
      ORG 100H
SERIAL: JB TI, TRANS        ; jump if TI is high
      MOV A, SBUF           ; otherwise due to receive
      CLR RI               ; clear RI since CPU doesn't
  
```

```

TRANS:    RETI                ; return from ISR
          CLR TI              ; clear TI since CPU doesn't RETI
          ; return from ISR
          END

```

The moment a byte is written into SBUF it is framed and transferred serially. As a result, when the last bit (stop bit) is transferred the TI is raised, and that causes the serial interrupt to be invoked since the corresponding bit in the IE register is high. In the serial ISR, we check for both TI and RI since both could have invoked interrupt.

Use of serial COM in the 8051:

In majority applications, the serial interrupt is used mainly for receiving data and is never used for sending data serially. This is like receiving a telephone call, where we need a ring to be notified. If we need to make a phone call there are other ways to remind ourselves and so no need for ringing. In receiving call, we must respond immediately. Similarly we use the serial interrupt to receive incoming data so that it is not lost.

Write a program in which the 8051 gets data from P1 and sends it to P2 continuously while incoming data from the serial port is sent to P0. Assume that XTAL=11.0592. Set the baud rate at 9600.

Solution:

```

          ORG 0000H
          LJMP MAIN
          ORG 23H
          LJMP SERIAL          ; jump to serial int ISR
          ORG 30H
MAIN:     MOV P1, #0FFH        ; make P1 an input port
          MOV TMOD, #20H      ; timer 1, auto reload
          MOV TH1, #0FDH      ; 9600 baud rate
          MOV SCON, #50H      ; 8-bit, 1 stop, ren enabled
          MOV IE, #10010000B  ; enable serial int.
          SETB TR1            ; start timer 1
BACK:     MOV A, P1            ; read data from port 1
          MOV P2, A           ; send it to P2
          SJMP BACK           ; stay in loop indefinitely
; ----- SERIAL PORT ISR
          ORG 100H
SERIAL:   JB TI, TRANS         ; jump if TI is high
          MOV A, SBUF          ; otherwise due to receive
          MOV P0, A            ; send incoming data to P0
          CLR RI               ; clear RI since CPU doesn't
          RETI                 ; return from ISR
TRANS:    CLR TI              ; clear TI since CPU doesn't
          RETI                 ; return from ISR
          END

```


Write a program using interrupts to do the following:

- (a) Receive data serially and sent it to P0,**
(b) Have P1 port read and transmitted serially, and a copy given to P2, (c) Make timer 0 generate a square wave of 5kHz frequency on P0.1. Assume that XTAL=11,0592. Set the baud rate at 4800.

Solution:

```
ORG 0
LJMP MAIN
ORG 000BH                ; ISR for timer 0

CPL P0.1                 ; toggle P0.1

RETI                     ; return from ISR
ORG 23H
LJMP SERIAL              ; jump to serial interrupt ISR
ORG 30H
MAIN: MOV P1, #0FFH       ; make P1 an input port
      MOV TMOD, #22H      ; timer 1, mode 2 (auto reload)

      MOV TH1, #0F6H      ; 4800 baud rate

      MOV SCON, #50H      ; 8-bit, 1 stop, ren enabled

      MOV TH0, #-92       ; for 5kHz wave

      MOV IE, #10010010B ; enable serial int.

      SETB TR1            ; start timer 1

      SETB TR0            ; start timer 0

BACK: MOV A, P1           ; read data from port 1
      MOV SBUF, A         ; give a copy to SBUF

      MOV P2, A           ; send it to P2

      SJMP BACK           ; stay in loop indefinitely

; ----- SERIAL PORT ISR
ORG 100H
SERIAL: JB TI, TRANS      ; jump if TI is high
        MOV A, SBUF       ; otherwise due to receive
        MOV P0, A         ; send serial data to P0
        CLR RI            ; clear RI since CPU doesn't
        RETI              ; return from ISR
TRANS:  CLR TI            ; clear TI since CPU doesn't
```

```
RETI                ; return from ISR
END
```

Clearing RI and TI before the RETI instruction:

In the above program RI and TI is cleared in the ISR before RETI instruction. This is necessary since there is only one interrupt for both receive and transmit, and the 8051 does not know who generated it. Hence programmer has to clear the flag in the ISR. Whereas if the interrupt is due to timers or external hardware interrupt, 8051 will clear the flag. The TCON register holds the four of the interrupt flags and SCON register has the RI and TI flags.

Interrupt Flag Bits

Interrupt	Flag	SFR Register Bit
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial Port	T1	SCON.1

Interrupt priority:

When the 8051 is powered up, the priorities are assigned according to the following table.

Interrupt Priority Upon Reset

Highest To Lowest Priority	
External Interrupt 0	(INT0)
Timer Interrupt 0	(TF0)
External Interrupt 1	(INT1)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)

If INT0 and INT1 are activated at the same time, INT0 is responded first because it has higher priority over INT1.

Setting interrupt priority with the IP register:

We can alter the sequence of interrupt priority by assigning a higher priority to any one of the interrupts. This is done by programming a register called IP (interrupt priority).

Interrupt Priority Register (Bit-addressable)

D7		D0					
--	--	PT2	PS	PT1	PX1	PT0	PX0
--	IP.7	Reserved					
--	IP.6	Reserved					
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)					
PS	IP.4	Serial port interrupt priority bit					
PT1	IP.3	Timer 1 interrupt priority bit					
PX1	IP.2	External interrupt 1 priority bit					
PT0	IP.1	Timer 0 interrupt priority bit					
PX0	IP.0	External interrupt 0 priority bit					

Priority bit=1 assigns high priority

Priority bit=0 assigns low priority

Discuss what happens if interrupts INT0, TF0, and INT1 are activated at the same time. Assume priority levels were set by the power-up reset and the external hardware interrupts are edge triggered.

Solution:

If these three interrupts are activated at the same time, they are latched and kept internally. Then the 8051 checks all five interrupts according to the sequence listed in Table 11-3. If any is activated, it services it in sequence. Therefore, when the above three interrupts are activated, IE0 (external interrupt 0) is serviced first, then timer 0 (TF0), and finally IE1 (external interrupt 1).

When two or more interrupt bits in the IP register are set to high, while these interrupts have a higher priority than others, they are serviced according to the normal priority sequence.

Interrupt inside an interrupt:

When 8051 is executing an ISR belonging to an interrupt and another interrupt is activated, then a high priority interrupt can interrupt a low priority interrupt. This is an interrupt inside an interrupt. Low priority interrupt can be interrupted by a higher priority interrupt, but not by another low priority interrupt. Although all the interrupts are latched and kept internally, no low priority interrupt can get the immediate attention of the CPU until 8051 has finished servicing the high priority interrupts.

Triggering the interrupt by software:

To test an ISR by way of simulation, set the interrupts high and thereby cause the 8051 to jump to the interrupt vector table. For example, if the IE bit for timer 1 is set, an instruction such as SETB TF1 will interrupt the 8051

in whatever it is doing and force it to jump to the interrupt vector table. That is we need not wait for timer 1 to roll over to to have an interrupt. We can cause an interrupt with an instruction that raises the interrupt flag.