

AVR INTERRUPTS

Hoda Roodaki

hroodaki@kntu.ac.ir

Interrupts vs. polling

- A single microcontroller can serve several devices.
- There are two methods by which devices receive service from the microcontroller:
 - Interrupts
 - In the interrupt method, whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal.
 - Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device.
 - The program associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler.
 - Polling.
 - In polling, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service.
 - After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller.

Interrupts vs. polling

- The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it.
- The polling method cannot assign priority because it checks all devices in a round-robin fashion.
- More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This also is not possible with the polling method.
- The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service.
- So interrupts are used to avoid tying down the microcontroller.

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 the interrupt service routine.
- Generally, in most microprocessors, 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.

Sources of interrupts in the AVR

- There are many sources of interrupts in the AVR:
 - There are at least two interrupts set aside for each of the timers, one for overflow and another for compare match.
 - Three interrupts are set aside for external hardware interrupts. Pins PD2 (PORTD.2), PD3 (PORTD.3), and PB2 (PORTB.2) are for the external hardware interrupts INT0, INT1, and INT2, respectively.
 - Serial communication's USART has three interrupts, one for receive and two interrupts for transmit.
 - The ADC (analog-to-digital converter).

Interrupt Vector Table for the ATmega32 AVR

Table 10-1: Interrupt Vector Table for the ATmega32 AVR

Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt request 0	0002
External Interrupt request 1	0004
External Interrupt request 2	0006
Time/Counter2 Compare Match	0008
Time/Counter2 Overflow	000A
Time/Counter1 Capture Event	000C
Time/Counter1 Compare Match A	000E
Time/Counter1 Compare Match B	0010
Time/Counter1 Overflow	0012
Time/Counter0 Compare Match	0014
Time/Counter0 Overflow	0016
SPI Transfer complete	0018
USART, Receive complete	001A
USART, Data Register Empty	001C
USART, Transmit Complete	001E
ADC Conversion complete	0020
EEPROM ready	0022
Analog Comparator	0024
Two-wire Serial Interface (I2C)	0026
Store Program Memory Ready	0028

Interrupt Vector Table for the ATmega32 AVR

- A limited number of bytes is set aside for interrupts.
 - For example, a total of 2 words (4 bytes), from locations 0016 to 0018, are set aside for Timer0 overflow interrupt.
 - Normally, the service routine for an interrupt is too long to fit into the memory space allocated.
 - For that reason, a JMP instruction is placed in the vector table to point to the address of the ISR.

Steps in executing an interrupt

- Upon activation of an interrupt, the microcontroller goes through the following steps:
 - It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.
 - It jumps to a fixed location in memory called the *interrupt vector table*. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
 - The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
 - 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 bytes of the stack into the PC. Then it starts to execute from that address.

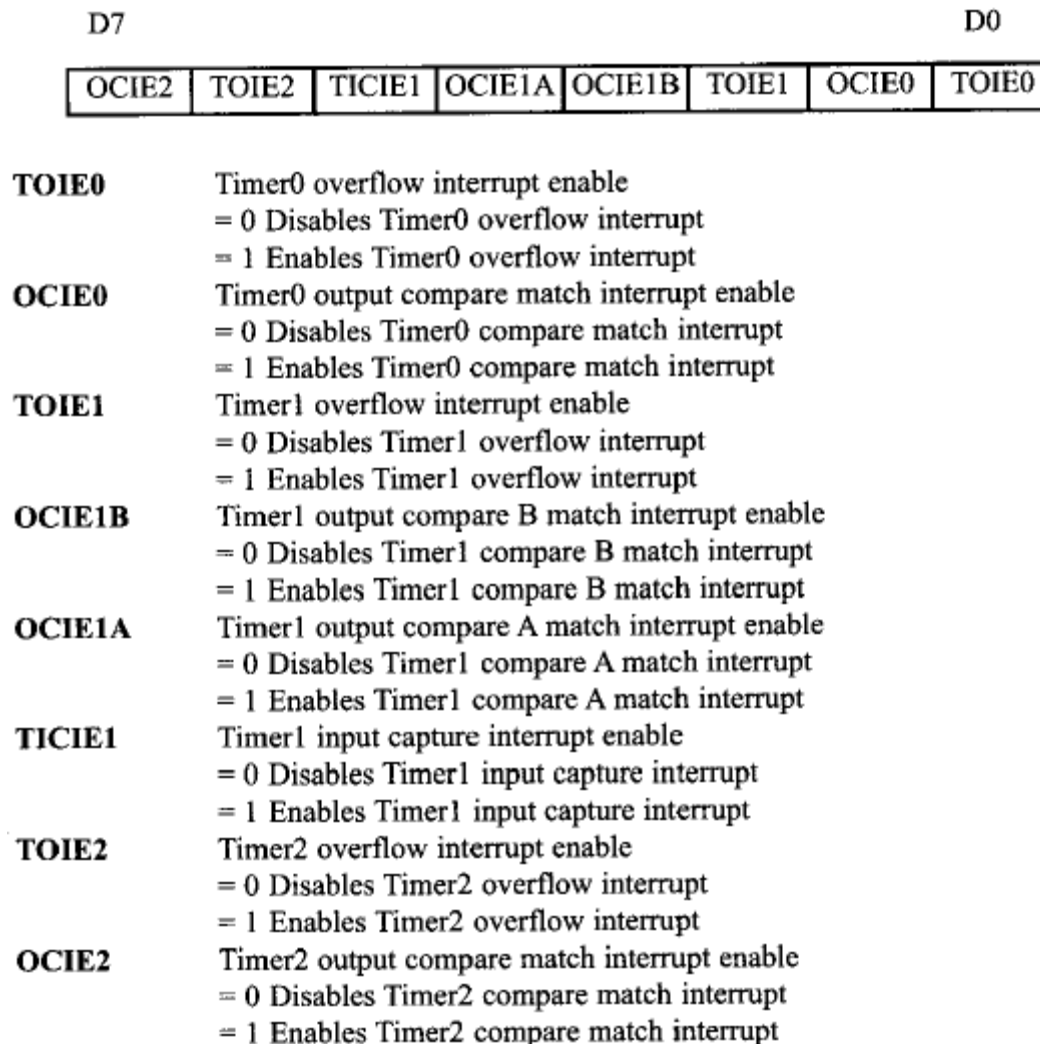
Enabling and disabling an interrupt

- 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 (unmasked) by software in order for the microcontroller to respond to them.
 - The D7 bit of the SREG (Status Register) register is responsible for enabling and disabling the interrupts globally.
 - The I bit makes the job of disabling all the interrupts easy. With a single instruction "CLI" (Clear Interrupt), we can make I= 0 during the operation of a critical task.

Steps in enabling an interrupt

- To enable any one of the interrupts, we take the following steps:
 - Bit D7 (I) of the SREG register must be set to HIGH to allow the interrupts to happen. This is done with the "SEI" (Set Interrupt) instruction.
 - If $I = 1$, each interrupt is enabled by setting to HIGH the interrupt enable (IE) flag bit for that interrupt.
 - There are some I/O registers holding the interrupt enable bits.
 - If $I = 0$, no interrupt will be responded to, even if the corresponding interrupt enable bit is high.

TIMSK (Timer Interrupt Mask) Register



Example 10-1

Show the instructions to (a) enable (unmask) the Timer0 overflow interrupt and Timer2 compare match interrupt, and (b) disable (mask) the Timer0 overflow interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

Solution:

- (a) LDI R20, (1<<TOIE0) | (1<<OCIE2) ;TOIE0 = 1, OCIE2 = 1
 OUT TIMSK,R20 ;enable Timer0 overflow and Timer2 compare match
 SEI ;allow interrupts to come in
- (b) IN R20,TIMSK ;R20 = TIMSK
 ANDI R20,0xFF^(1<<TOIE0) ;TOIE0 = 0
 OUT TIMSK,R20 ;mask (disable) Timer0 interrupt

We can perform the above actions with the following instructions, as well:

- IN R20,TIMSK ;R20 = TIMSK
 CBR R20,1<<TOIE0 ;TOIE0 = 0
 OUT TIMSK,R20 ;mask (disable) Timer0 interrupt
- (c) CLI ;mask all interrupts globally

Notice that in part (a) we can use “LDI, 0x81” in place of the following instruction:
“LDI R20, (1<<TOIE0) | (1<<OCIE2)”

PROGRAMMING TIMER INTERRUPTS

- **Rollover timer flag and interrupt**
- If the timer interrupt in the interrupt register is enabled, TOV0 is raised whenever the timer rolls over and the microcontroller 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.
- To use an interrupt in place of polling, first we must enable the interrupt because all the interrupts are masked upon reset.
- The TOIE_x bit enables the interrupt for a given timer. TOIE_x bits are held by the TIMSK register.

PROGRAMMING TIMER INTERRUPTS

Table 10-2: Timer Interrupt Flag Bits and Associated Registers

Interrupt Overflow Register Flag Bit			Enable Bit	Register
Timer0	TOV0	TIFR	TOIE0	TIMSK
Timer1	TOV1	TIFR	TOIE1	TIMSK
Timer2	TOV2	TIFR	TOIE2	TIMSK

PROGRAMMING TIMER INTERRUPTS

- We must avoid using the memory space allocated to the interrupt vector table.
 - Therefore, we place all the initialization codes in memory starting at an address such as \$100.
 - The JMP instruction is the first instruction that the AVR executes when it is awakened at address 0000 upon reset.
 - The JMP instruction at address 0000 redirects the controller away from the interrupt vector table.
- In the MAIN program, we enable (unmask) the Timer0 interrupt with the following instructions

```
LDI    R16,1<<TOV0
OUT    TIMSK,R16    ;enable Timer0 overflow interrupt
SEI    ;set I (enable interrupts globally)
```

PROGRAMMING TIMER INTERRUPTS

- In the MAIN program, we initialize the Timer0 register and then enter into an infinite loop to keep the CPU busy.
- The loop could be replaced with a real world application being executed by the CPU.
 - For example, the loop gets data from PORTC and sends it to PORTD. While the PORTC data is brought in and issued to PORTD continuously, the TOIE0 flag is raised as soon as Timer0 rolls over, and the microcontroller gets out of the loop and goes to \$0016 to execute the JSR associated with Timer0.
 - At this point, the AVR clears the I bit (D7 of SREG) to indicate that it is currently serving an interrupt and cannot be interrupted again; in other words, no interrupt inside the interrupt.

PROGRAMMING TIMER INTERRUPTS

- The ISR for Timer0 is located starting at memory location \$200 because it is too large to fit into address space \$16--\$18, the address allocated to the Timer0 overflow interrupt in the interrupt vector table.
- RETI must be the last instruction of the ISR. Upon execution of the RETI instruction, the AVR automatically enables the I bit (D7 of the SREG register) to indicate that it can accept new interrupts.
- In the ISR for Timer0, notice that there is no need for clearing the TOV0 flag since the AVR clears the TOV0 flag internally upon jumping to the interrupt vector table.

Example- Program 10-1

- For this program, we assume that PORTC is connected to 8 switches and PORTD to 8 LEDs. This program uses Timer0 to generate a square wave on pin PORTB.5, while at the same time data is being transferred from PORTC to PORTD.

Example- Program 10-1

```
;Program 10-1
.INCLUDE "M32DEF.INC"
.ORG 0x0          ;location for reset
    JMP    MAIN
.ORG 0x16         ;location for Timer0 overflow (see Table 10.1)
    JMP    T0_OV_ISR      ;jump to ISR for Timer0
;-main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20      ;initialize stack
      SBI    DDRB,5       ;PB5 as an output
      LDI    R20,(1<<TOIE0)
      OUT    TIMSK,R20    ;enable Timer0 overflow interrupt
      SEI                    ;set I (enable interrupts globally)
      LDI    R20,-32      ;timer value for 4  $\mu$ s
      OUT    TCNT0,R20    ;load Timer0 with -32
      LDI    R20,0x01
      OUT    TCCR0,R20    ;Normal, internal clock, no prescaler
      LDI    R20,0x00
      OUT    DDRC,R20     ;make PORTC input
      LDI    R20,0xFF
      OUT    DDRD,R20     ;make PORTD output
;----- Infinite loop
HERE: IN     R20,PINC      ;read from PORTC
      OUT    PORTD,R20    ;give it to PORTD
      JMP    HERE        ;keeping CPU busy waiting for interrupt
```

Example- Program 10-1

```
;-----ISR for Timer0 (it is executed every 4  $\mu$ s)
.ORG 0x200
T0_OV_ISR:
    IN     R16,PORTB    ;read PORTB
    LDI    R17,0x20     ;00100000 for toggling PB5
    EOR    R16,R17
    OUT    PORTB,R16    ;toggle PB5
    LDI    R16,-32      ;timer value for 4  $\mu$ s
    OUT    TCNT0,R16    ;load Timer0 with -32 (for next round)
    RETI               ;return from interrupt
```

Example- Program 10-2

- Use Timer0 and Timer1 interrupts simultaneously, to generate square waves on pins PB1 and PB7 respectively, while data is being transferred from PORTC to PORTD.

Example- Program 10-2

```
;Program 10-2
.INCLUDE "M32DEF.INC"
.ORG 0x0                ;location for reset
    JMP    MAIN          ;bypass interrupt vector table
.ORG 0x12               ;ISR location for Timer1 overflow
    JMP    T1_OV_ISR     ;go to an address with more space
.ORG 0x16               ;ISR location for Timer0 overflow
    JMP    T0_OV_ISR     ;go to an address with more space
;----main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20      ;initialize stack point
      SBI    DDRB,1       ;PB1 as an output
      SBI    DDRB,7       ;PB7 as an output
      LDI    R20,(1<<TOIE0)|(1<<TOIE1)
      OUT    TIMSK,R20    ;enable Timer0 overflow interrupt
      SEI                    ;set I (enable interrupts globally)
      LDI    R20,-160     ;value for 20 µs
      OUT    TCNT0,R20    ;load Timer0 with -160
      LDI    R20,0x01
      OUT    TCCR0,R20    ;Normal mode, int clk, no prescaler
      LDI    R20,HIGH(-640) ;the high byte
      OUT    TCNT1H,R20   ;load Timer1 high byte
```

Example- Program 10-2

```
LDI R20,LOW(-640) ;the low byte
OUT TCNT1L,R20 ;load Timer1 low byte
LDI R20,0x00
OUT TCCR1A,R20 ;Normal mode
LDI R20,0x01
OUT TCCR1B,R20 ;internal clk, no prescaler
LDI R20,0x00
OUT DDRC,R20 ;make PORTC input
LDI R20,0xFF
OUT DDRD,R20 ;make PORTD output
;----- Infinite loop
HERE: IN R20,PINC ;read from PORTC
OUT PORTD,R20 ;and give it to PORTD
JMP HERE ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0 (It comes here after elapse of 20 µs time)
.ORG 0x200
T0_OV_ISR:
LDI R16,-160 ;value for 20 µs
OUT TCNT0,R16 ;load Timer0 with -160 (for next round)
IN R16,PORTB ;read PORTB
LDI R17,0x02 ;00000010 for toggling PB1
EOR R16,R17
OUT PORTB,R16 ;toggle PB1
RETI ;return from interrupt
;-----ISR for Timer1 (It comes here after elapse of 80 µs time)
.ORG 0x300
T1_OV_ISR:
LDI R18,HIGH(-640)
OUT TCNT1H,R18 ;load Timer1 high byte
LDI R18,LOW(-640)
OUT TCNT1L,R18 ;load Timer1 low byte (for next round)
IN R18,PORTB ;read PORTB
LDI R19,0x80 ;10000000 for toggling PB7
EOR R18,R19
OUT PORTB,R18 ;toggle PB7
RETI ;return from interrupt
```

Example- Program 10-3

- Program 10-3 has two interrupts: (1) PORTA counts up every time Timer1 overflows. It overflows once per second. (2) A pulse is fed into Timer0, where Timer0 is used as counter and counts up. Whenever the counter reaches 200, it will toggle the pin PORTB.6

Example- Program 10-3

```
;Program 10-3
.INCLUDE "M32DEF.INC"
.ORG 0x0          ;location for reset
    JMP  MAIN      ;bypass interrupt vector table
.ORG 0x12         ;ISR location for Timer1 overflow
    JMP  T1_OV_ISR  ;go to an address with more space
.ORG 0x16         ;ISR location for Timer0 overflow
    JMP  T0_OV_ISR  ;go to an address with more space
```

Example- Program 10-3

```
;---main program for initialization and keeping CPU busy
.ORG 0x40
MAIN: LDI R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI R20,LOW(RAMEND)
      OUT SPL,R20 ;initialize SP

      LDI R18,0 ;R18 = 0
      OUT PORTA,R18 ;PORTA = 0
      LDI R20,0
      OUT DDRC,R20 ;PORTC as input
      LDI R20,0xFF
      OUT DDRA,R20 ;PORTA as output
      OUT DDRD,R20 ;PORTD as output
      SBI DDRB,6 ;PB6 as an output
      SBI PORTB,0 ;activate pull-up of PB0

      LDI R20,0x06
      OUT TCCR0,R20 ;Normal, T0 pin falling edge, no scale
      LDI R16,-200
      OUT TCNT0,R16 ;load Timer0 with -200
      LDI R19,HIGH(-31250) ;timer value for 1 second
      OUT TCNT1H,R19 ;load Timer1 high byte
      LDI R19,LOW(-31250)
      OUT TCNT1L,R19 ;load Timer1 low byte
      LDI R20,0
      OUT TCCR1A,R20 ;Timer1 Normal mode
      LDI R20,0x04
      OUT TCCR1B,R20 ;int clk, prescale 1:256
      LDI R20,(1<<TOIE0)|(1<<TOIE1)
      OUT TIMSK,R20 ;enable Timer0 & Timer1 overflow ints
      SEI ;set I (enable interrupts globally)
```

Example- Program 10-3

```
;----- Infinite loop
HERE: IN      R20,PINC      ;read from PORTC
      OUT     PORTD,R20     ;and send it to PORTD
      JMP     HERE         ;waiting for interrupt
;-----ISR for Timer0 to toggle after 200 clocks
.ORG 0x200
T0_OV_ISR:
      IN      R16,PORTB     ;read PORTB
      LDI     R17,0x40      ;0100 0000 for toggling PB7
      EOR     R16,R17
      OUT     PORTB,R16     ;toggle PB6
      LDI     R16,-200      ;setup for next round
      OUT     TCNT0,R16     ;load Timer0 with -200 for next round
      RETI                ;return from interrupt
;-----ISR for Timer1 (It comes here after elapse of 1s time)
.ORG 0x300
T1_OV_ISR:
      INC     R18           ;increment upon overflow
      OUT     PORTA,R18     ;display it on PORTA
      LDI     R19,HIGH(-31250)
      OUT     TCNT1H,R19    ;load Timer1 high byte
      LDI     R19,LOW(-31250)
      OUT     TCNT1L,R19    ;load Timer1 low byte (for next round)
      RETI                ;return from interrupt
```

Compare match timer flag and interrupt

- The programs can be written using the CTC mode and compare match (OCF) flag.
- To do so, we load the OCR register with the proper value and initialize the timer to the CTC mode.
- When the content of TCNT matches with OCR, the OCF flag is set. CTC Mode causes the compare match interrupt to occur

Example 10-3

Using Timer0, write a program that toggles pin PORTB.5 every 40 μ s, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 1 MHz.

Solution:

$1/1 \text{ MHz} = 1 \mu\text{s}$ and $40 \mu\text{s}/1 \mu\text{s} = 40$. That means we must have $\text{OCR0} = 40 - 1 = 39$

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP     MAIN
.ORG 0x14    ;ISR location for Timer0 compare match
    JMP     T0_CM_ISR
;main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI     R20,HIGH(RAMEND)
      OUT     SPH,R20
      LDI     R20,LOW(RAMEND)
      OUT     SPL,R20    ;set up stack
      SBI     DDRB,5      ;PB5 as an output
      LDI     R20,(1<<OCIE0)
      OUT     TIMSK,R20   ;enable Timer0 compare match interrupt
      SEI                          ;set I (enable interrupts globally)
      LDI     R20,39
      OUT     OCR0,R20    ;load Timer0 with 39
      LDI     R20,0x09
      OUT     TCCR0,R20   ;start Timer0, CTC mode, int clk, no prescaler
      LDI     R20,0x00
      OUT     DDRC,R20    ;make PORTC input
      LDI     R20,0xFF
      OUT     DDRD,R20    ;make PORTD output
```

Example 10-3

```
;----- Infinite loop
HERE: IN    R20,PINC    ;read from PORTC
      OUT    PORTD,R20  ;and send it to PORTD
      JMP    HERE      ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0 (it is executed every 40  $\mu$ s)
TO_CM_ISR:
      IN     R16,PORTB  ;read PORTB
      LDI    R17,0x20   ;00100000 for toggling PB5
      EOR    R16,R17
      OUT    PORTB,R16  ;toggle PB5
      RETI             ;return from interrupt
```

Example 10-4

Using Timer1, write a program that toggles pin PORTB.5 every second, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 8 MHz.

Solution:

For prescaler = 1024 we have $T_{\text{Clock}} = (1 / 8 \text{ MHz}) \times 1024 = 128 \mu\text{s}$ and $1 \text{ s} / 128 \mu\text{s} = 7812$. That means we must have $\text{OCR1A} = 7811 = 0x1E83$

```
.INCLUDE "M32DEF.INC"
.ORG 0x0 ;location for reset
JMP MAIN
.ORG 0x14 ;location for Timer1 compare match
JMP T1_CM_ISR
;-----main program for initialization and keeping CPU busy
MAIN: LDI R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI R20,LOW(RAMEND)
      OUT SPL,R20 ;set up stack
      SBI DDRB,5 ;PB5 as an output
      LDI R20,(1<<OCIE1A)
      OUT TIMSK,R20 ;enable Timer1 compare match interrupt
      SEI ;set I (enable interrupts globally)
      LDI R20,0x00
      OUT TCCR1A,R20
      LDI R20,0xD
      OUT TCCR1B,R20 ;prescaler 1:1024, CTC mode
      LDI R20,HIGH(7811) ;the high byte
      OUT OCR1AH,R20 ;Temp = 0x1E (high byte of 7811)
      LDI R20,LOW(7811) ;the low byte
      OUT OCR1AL,R20 ;OCR1A = 7811
      LDI R20,0x00
      OUT DDRC,R20 ;make PORTC input
      LDI R20,0xFF
      OUT DDRD,R20 ;make PORTD output
```

Example 10-4

```
;----- Infinite loop
HERE: IN    R20,PINC    ;read from PORTC
      OUT   PORTD,R20   ;PORTD = R20
      JMP   HERE        ;keeping CPU busy waiting for interrupt
;---ISR for Timer1 (It comes here after elapse of 1 second time)
T1_CM_ISR:
      IN    R16,PORTB
      LDI   R17,0x20     ;00100000 for toggling PB5
      EOR   R16,R17
      OUT   PORTB,R16    ;toggle PB5
      RETI               ;return from interrupt
```


PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

- The number of external hardware interrupt interrupts varies in different AVR.
- The ATmega32 has three external hardware interrupts: pins PD2 (PORTD.2), PD3 (PORTD.3), and PB2 (PORTB.2), designated as INT0, INT1, and INT2, respectively.
- Upon activation of these pins, the AVR is interrupted in whatever it is doing and jumps to the vector table to perform the interrupt service routine.

External interrupts

INT0, INT1, and INT2

- The interrupt vector table locations \$2, \$4, and \$6 are set aside for INT0, INT1, and INT2, respectively.
- The hardware interrupts must be enabled before they can take effect.
- This is done using the INTx bit located in the GICR register.

```
LDI    R20, 0x40
OUT    GICR, R20
```

External interrupts INT0, INT1, and INT2

D7							D0
INT1	INT0	INT2	-	-	-	IVSEL	IVCE

INT0 External Interrupt Request 0 Enable
 = 0 Disables external interrupt 0
 = 1 Enables external interrupt 0

INT1 External Interrupt Request 1 Enable
 = 0 Disables external interrupt 1
 = 1 Enables external interrupt 1

INT2 External Interrupt Request 2 Enable
 = 0 Disables external interrupt 2
 = 1 Enables external interrupt 2

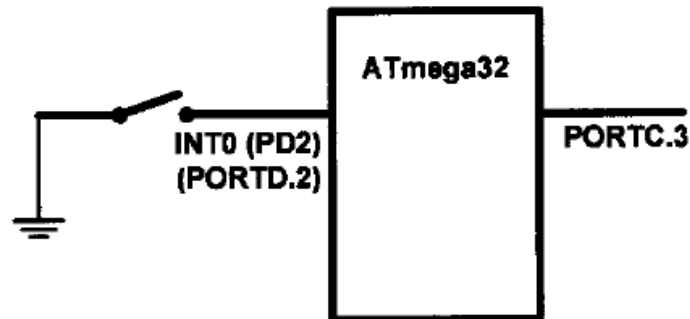
These bits, along with the I bit, must be set high for an interrupt to be responded to.

Example 10-5

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3 whenever the INT0 pin goes low.

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0                      ;location for reset
    JMP     MAIN
.ORG 0x02                   ;vector location for external interrupt 0
    JMP     EX0_ISR
MAIN: LDI     R20,HIGH(RAMEND)
    OUT     SPH,R20
    LDI     R20,LOW(RAMEND)
    OUT     SPL,R20          ;initialize stack
    SBI     DDRC,3           ;PORTC.3 = output
    SBI     PORTD,2          ;pull-up activated
    LDI     R20,1<<INT0     ;enable INT0
    OUT     GICR,R20
    SEI                     ;enable interrupts
HERE:JMP     HERE           ;stay here forever
EX0_ISR:
    IN      R21,PINC         ;read PINC
    LDI     R22,0x08         ;00001000
    EOR     R21,R22
    OUT     PORTC,R21
    RETI
```



Edge-triggered vs. level-triggered interrupts

- There are two types of activation for the external hardware interrupts:
 - level triggered
 - edge triggered
 - INT2 is only edge triggered, while INT0 and INT1 can be level or edge triggered.
 - Upon reset INT0 and INT1 are low-level-triggered interrupts.
 - The bits of the MCUCR register indicate the trigger options of INT0 and INT1.

Edge-triggered vs. level-triggered interrupts





MCUCR register

D7





D0

SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
----	-----	-----	-----	-------	-------	-------	-------

ISC01, ISC00 (Interrupt Sense Control bits) These bits define the level or edge on the external INT0 pin that activates the interrupt, as shown in the following table:

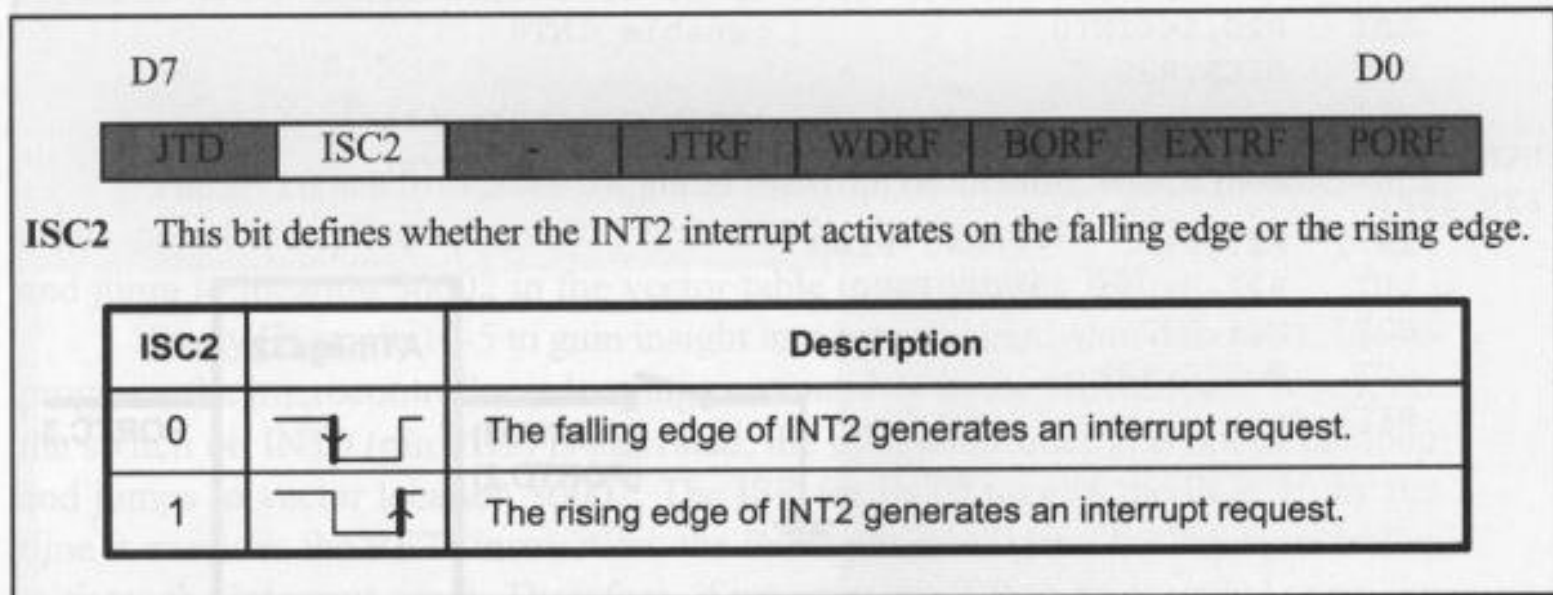
ISC01	ISC00		Description
0	0		The low level of INT0 generates an interrupt request.
0	1		Any logical change on INT0 generates an interrupt request.
1	0		The falling edge of INT0 generates an interrupt request.
1	1		The rising edge of INT0 generates an interrupt request.

ISC11, ISC10 These bits define the level or edge that activates the INT1 pin.

ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

Edge-triggered vs. level-triggered interrupts

- The ISC2 bit of the MCUCSR register defines whether INT2 activates in the falling edge or the rising edge.
- Upon reset ISC2 is 0, meaning that the external hardware interrupt of INT2 is falling edge triggered.



Example 10-6

Show the instructions to (a) make INT0 falling edge triggered, (b) make INT1 triggered on any change, and (c) make INT2 rising edge triggered.

Solution:

```
(a)  LDI    R20, 0x02  
      OUT    MCUCR, R20
```

```
(b)  LDI    R20, 1<<ISC10      ;R20 = 0x04  
      OUT    MCUCR, R20
```

```
(c)  LDI    R20, 1<<ISC2      ;R20 = 0x40  
      OUT    MCUCSR, R20
```


Example 10-7

Rewrite Example 10-5, so that whenever INT0 goes low, it toggles PORTC.3 only once

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0                                ;location for reset
    JMP    MAIN
.ORG 0x02                             ;location for external interrupt 0
    JMP    EX0_ISR
MAIN: LDI    R20,HIGH(RAMEND)
    OUT    SPH,R20
    LDI    R20,LOW(RAMEND)
    OUT    SPL,R20                    ;initialize stack
    LDI    R20,0x2                    ;make INT0 falling edge triggered
    OUT    MCUCR,R20
    SBI    DDRC,3                     ;PORTC.3 = output
    SBI    PORTD,2                    ;pull-up activated
    LDI    R20,1<<INT0               ;enable INT0
    OUT    GICR,R20
    SEI                                ;enable interrupts
HERE: JMP    HERE
EX0_ISR:
    IN     R21,PORTC
    LDI    R22,0x08                   ;00001000 for toggling PC3
    EOR    R21,R22
    OUT    PORTC,R21
    RETI
```

Sampling the edge-triggered and level-triggered interrupts

- The edge interrupt (the falling edge, the rising edge, or the change level) is latched by the AVR and is held by the INTFx bits of the GIFR register.
- This means that when an external interrupt is in an edge-triggered mode (falling edge, rising edge, or change level), upon triggering an interrupt request, the related INTFx flag becomes set.
- GIFR (General Interrupt Flag Register) Register:

Bit	D7							D0
	INTF1	INTF0	INTF2	-	-	-	-	-

Sampling the edge-triggered and level-triggered interrupts

- If the interrupt is active (the INTx bit is set and the I-bit in SREG is one), the AVR will jump to the corresponding interrupt vector location and the INTFx flag will be cleared automatically, otherwise the flag remains set.
- The flag can be cleared by writing a one to it.

```
LDI    R20, (1<<INTF1)    ;R20 = 0x80
OUT    GIFR,R20            ;clear the INTF1 flag
```

Sampling the edge-triggered and level-triggered interrupts

- Notice that in edge-triggered interrupts (falling edge, rising edge, and change level interrupts), the pulse must last at least 1 instruction cycle to ensure that the transition is seen by the microcontroller. This means that pulses shorter than 1 machine cycle are not guaranteed to generate an interrupt.

Sampling the edge-triggered and level-triggered interrupts

- When an external interrupt is in level-triggered mode, the interrupt is not latched, meaning that the INTFx flag remains unchanged when an interrupt occurs, and the state of the pin is read directly. As a result, when an interrupt is in level-triggered mode, the pin must be held low for a minimum time of 5 machine cycles to be recognized.

INTERRUPT PRIORITY IN THE AVR

- If two interrupts are activated at the same time, the interrupt with the higher priority is served first.
- The priority of each interrupt is related to the address of that interrupt in the interrupt vector.
- The interrupt that has a lower address, has a higher priority.
 - For example, the address of external interrupt 0 is 2, while the address of external interrupt 2 is 6; thus, external interrupt 0 has a higher priority, and if both of these interrupts are activated at the same time, external interrupt 0 is served first.

Interrupt inside an interrupt

- What happens if the AVR is executing an ISR belonging to an interrupt and another interrupt is activated?
- When the AVR begins to execute an ISR, it disables the I bit of the SREG register, causing all the interrupts to be disabled, and no other interrupt occurs while serving the interrupt.
- When the RETI instruction is executed, the AVR enables the I bit, causing the other interrupts to be served.
- If you want another interrupt (with any priority) to be served while the current interrupt is being served you can set the I bit using the SEI instruction. But do it with care.
 - For example, in a low-level-triggered external interrupt, enabling the I bit while the pin is still active will cause the ISR to be reentered infinitely, causing the stack to overflow with unpredictable consequences

Context saving in task switching

- In multitasking systems, the CPU serves one task (job or process) at a time and then moves to the next one.
- In simple systems, the tasks can be organized as the interrupt service routine.
 - For example, in Example 10-3, the program does two different tasks
 - copying the contents of PORTC to PORTD
 - toggling PORTC.2 every 5 μ s

Context saving in task switching

- While writing a program for a multitasking system, we should manage the resources carefully so that the tasks do not conflict with each other.
- For example, consider a system that should perform the following tasks:
 - increasing the contents of PORTC continuously
 - increasing the content of PORTD once every 5 μ s

Context saving in task switching

```
MAIN: LDI    R20, HIGH(RAMEND)
      OUT    SPH, R20
      LDI    R20, LOW(RAMEND)
      OUT    SPL, R20      ;set up stack
      SBI    DDRB, 5       ;PB5 as an output
      LDI    R20, (1<<OCIE0)
      OUT    TIMSK, R20    ;enable Timer0 compare match interrupt
      SEI                      ;set I (enable interrupts globally)
      LDI    R20, 160
      OUT    OCR0, R20     ;load Timer0 with 160
      LDI    R20, 0x09
      OUT    TCCR0, R20    ;CTC mode, int clk, no prescaler
      LDI    R20, 0xFF
      OUT    DDRC, R20     ;make PORTC output
      OUT    DDRD, R20     ;make PORTD output
      LDI    R20, 0
HERE: OUT    PORTC, R20     ;PORTC = R20
      INC    R20
      JMP    HERE          ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0
T0_CM_ISR:
      IN     R20, PIND
      INC    R20
      OUT    PORTD, R20    ;PORTD = R20
      RETI                ;return from interrupt
```

Context saving in task switching

- The tasks do not work properly, since they have resource conflict and they interfere with each other.
- R20 is used and changed by both tasks, which causes the program not to work properly.
 - For example, consider the following scenario:
 - The content of R20 increases in the main program, at first becoming 0, then 1, and so on. When the timer interrupt occurs, R20 is 95, and PORTC is 95 as well. In the ISR, the R20 is loaded with the content of PORTD, which is 0. So, when it goes back to the main program, the content of R20 is 1 and PORTC will be loaded by 2. But if the program worked properly, PORTC would be loaded with 96.

Context saving in task switching

- We can solve such problems in the following two ways:
 1. Using different registers for different tasks. In the program discussed above, if we use different registers in the main program and in the ISR, the program will work properly

Context saving in task switching

```
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20      ;set up stack
      SBI    DDRB,5       ;PB5 as an output
      LDI    R20,(1<<OCIE0)
      OUT    TIMSK,R20    ;enable Timer0 compare match interrupt
      SEI                    ;set I (enable interrupts globally)
      LDI    R20,160
      OUT    OCR0,R20     ;load Timer0 with 160
      LDI    R20,0x09
      OUT    TCCR0,R20    ;start timer,CTC mode,int clk,no prescaler
      LDI    R20,0xFF
      OUT    DDRC,R20     ;make PORTC output
      OUT    DDRD,R20     ;make PORTD output
      LDI    R20, 0
      HERE: OUT    PORTC,R20 ;PORTC = R20
      INC    R20
      JMP    HERE        ;keeping CPU busy waiting for int.
;-----ISR for Timer0
T0_CM_ISR:
      IN     R21,PIND
      INC    R21
      OUT    PORTD,R21    ;toggle PB5
      RETI                ;return from interrupt
```

Context saving in task switching

2. Context saving:

- In big programs we might not have enough registers to use separate registers for different tasks.
- In these cases, we can save the contents of registers on the stack before execution of each task, and reload the registers at the end of the task.
- This saving of the CPU contents before switching to a new task is called *context saving* (or *context switching*).

Context saving in task switching

```
T0_CM_ISR:
    PUSH    R20            ;save R20 on stack
    IN      R20,PIND
    INC     R20
    OUT     PORTD,R20      ;toggle PB5
    POP     R20            ;restore value for R20
    RETI                  ;return from interrupt
```

- Notice that using the stack as a place to save the CPU's contents is tedious, time consuming, and slow. So, we might want to use the first solution, whenever we have enough registers

Saving flags of the SREG register

- The flags of SREG are important especially when there are conditional jumps in our program. We should save the SREG register if the flags are changed in a task.

```
Sample_ISR:
    PUSH    R20
    IN      R20,SREG
    PUSH    R20
    ...
    POP     R20
    OUT     SREG,R20
    POP     R20
    RETI
```


Interrupt latency

- The time from the moment an interrupt is activated to the moment the CPU starts to execute the task is called the *interrupt latency*.
- This latency is 4 machine cycle times.
- During this time the PC register is pushed on the stack and the I bit of the SREG register clears, causing all the interrupts to be disabled.
- The duration of an interrupt latency can be affected by the type of instruction that the CPU is executing when the interrupt comes in, since the CPU finishes the execution of the current instruction before it serves the interrupt.
 - It takes slightly longer in cases where the instruction being executed lasts for two (or more) machine cycles (e.g., MUL) compared to the instructions that last for only one instruction cycle (e.g., ADD).