

به نام خدا

برنامه‌ریزی وقفه‌ها در AVR

آشنایی با وقفه‌ها

Dr. Aref Karimafshar
A.karimafshar@ec.iut.ac.ir



Interrupts vs. Polling

- Two methods of receiving service from μC
 - Polling
 - The μC continuously monitors the status of a given device
 - When the status condition is met
 - » It performs the service
 - Interrupts
 - Whenever the device needs μC 's service
 - Notifies the μC by sending an interrupt signal
 - » Upon receiving the interrupt signal, the μC stops whatever it is doing and serves the device
 - The program associated with the interrupt is called
 - » Interrupt Service Routine (ISR) or Interrupt Handler

Interrupts vs. Polling

- Polling
 - Can monitor the status of several devices
 - Serve each of them as certain conditions are met
 - Moves on to the next until each one is serviced
 - It is not an efficient use of μC
- Interrupts
 - μC can service many devices (not all at the same time, of course)
 - Each device can get the attention of the μC based on the priority assigned to it (polling cannot assign priority because it checks all devices in round-robin fashion)
 - The μC can ignore (mask) a device request (not possible in polling)

Interrupts vs. Polling

- Polling
 - Wastes much of μC 's time by polling devices that do not need services
- Interrupts
 - Avoid tying down the μC

```
AGAIN:IN    R20,TIFR    ;read TIFR
          SBRS  R20,TOV0 ;if TOV0 is set skip next instruction
          RJMP  AGAIN
```

```
while ((TIFR&0x1)==0); //wait for TF0 to roll over
```

Interrupts

- For every Interrupt
 - There must be an Interrupt Service Routine (ISR) or Interrupt Handler
 - There is a fixed location in memory that holds the address of its ISR
 - The group of memory locations set aside to hold the ISR addresses is called
 - Interrupt Vector Table
- When an interrupt is invoked
 - The μ C runs the ISR

Steps in Executing an Interrupt

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

1. It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.
2. 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).
3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
4. 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.

Interrupt Vector Table ATmega32

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

Interrupts in the AVR

Notice from Step 4 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal.

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.

```
                .ORG 0      ;wake-up ROM reset location
                JMP  MAIN    ;bypass interrupt vector table

;---- the wake-up program
                .ORG $100
MAIN:          ....        ;enable interrupt flags
                ....
```


Sources of Interrupts in the AVR

There are many sources of interrupts in the AVR, depending on which peripheral is incorporated into the chip. The following are some of the most widely used sources of interrupts in the AVR:

1. There are at least two interrupts set aside for each of the timers, one for overflow and another for compare match.
2. 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.
3. Serial communication's USART has three interrupts, one for receive and two interrupts for transmit.
4. The SPI interrupts.
5. The ADC (analog-to-digital converter).

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.



Steps in Enabling an Interrupt

To enable any one of the interrupts, we take the following steps:

1. 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.
2. 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.

Figure below shows that the TIMSK register has interrupt enable bits for Timer0, Timer1, and Timer2.



Steps in Enabling an Interrupt

D7							D0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

TOIE0	Timer0 overflow interrupt enable = 0 Disables Timer0 overflow interrupt = 1 Enables Timer0 overflow interrupt
OCIE0	Timer0 output compare match interrupt enable = 0 Disables Timer0 compare match interrupt = 1 Enables Timer0 compare match interrupt
TOIE1	Timer1 overflow interrupt enable = 0 Disables Timer1 overflow interrupt = 1 Enables Timer1 overflow interrupt
OCIE1B	Timer1 output compare B match interrupt enable = 0 Disables Timer1 compare B match interrupt = 1 Enables Timer1 compare B match interrupt
OCIE1A	Timer1 output compare A match interrupt enable = 0 Disables Timer1 compare A match interrupt = 1 Enables Timer1 compare A match interrupt
TICIE1	Timer1 input capture interrupt enable = 0 Disables Timer1 input capture interrupt = 1 Enables Timer1 input capture interrupt
TOIE2	Timer2 overflow interrupt enable = 0 Disables Timer2 overflow interrupt = 1 Enables Timer2 overflow interrupt
OCIE2	Timer2 output compare match interrupt enable = 0 Disables Timer2 compare match interrupt = 1 Enables Timer2 compare match interrupt

Example

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.

```
(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

we showed how to monitor the timer flag TOV0, TOV1 and TOV2 with the instruction “SBRSC R20, TOV0”. In polling TOV0, we have to wait until TOV0 is raised. The problem with this method is that the microcontroller is tied down waiting for TOV0 to be raised, and cannot do anything else. Using interrupts avoids tying down the controller. 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

Timer Interrupt Flag Bits and Associated Registers				
Interrupt	Overflow Flag Bit	Register	Enable Bit	Register
Timer0	TOV0	TIFR	TOIE0	TIMSK
Timer1	TOV1	TIFR	TOIE1	TIMSK
Timer2	TOV2	TIFR	TOIE2	TIMSK

Example

```
.INCLUDE "M32DEF.INC"
.ORG 0x0           ;location for reset
    JMP    MAIN
.ORG 0x16          ;location for Timer0 overflow
    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 15
```

Example cnt.

```
;-----ISR for Timer0 (it is executed every 4 µs)
.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 µs
    OUT    TCNT0,R16    ;load Timer0 with -32 (for next round)
    RETI               ;return from interrupt
```

Example: 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 cnt.

1. 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.
2. 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)
```
3. 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. In this case, 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 ISR 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.

Example cnt.

4. 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.
5. 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.
6. 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

What is the difference between the RET and RETI instructions? Explain why we cannot use RET instead of RETI as the last instruction of an ISR.

Both perform the same actions of popping off the top bytes of the stack into the program counter, and making the AVR return to where it left off. However, RETI also performs the additional task of setting the I flag, indicating that the servicing of the interrupt is over and the AVR now can accept a new interrupt. If you use RET instead of RETI as the last instruction of the interrupt service routine, you simply block any new interrupt after the first interrupt, because the I would indicate that the interrupt is still being serviced.

Example: Program below uses Timer0 and Timer1 interrupts simultaneously, to generate square waves on pins PB1 and PB7 respectively, while data is being transferred from PORTC to PORTD.

```
.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 cnt.

```
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  $\mu$ s time)
.ORG 0x200
T0_OV_ISR:
      LDI   R16,-160      ;value for 20  $\mu$ s
      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  $\mu$ s 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
```

Compare Match Timer Flag and Interrupt

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.

$1/1 \text{ MHz} = 1 \mu\text{s}$ 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    TO_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
;----- 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
```

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.

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.

For example, the following instructions enable INT0:

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



General Interrupt Control Register (GICR)

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

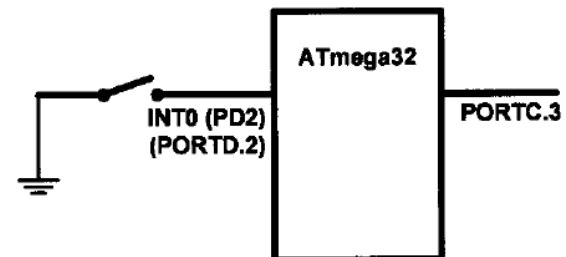
These bits, along with the I bit, must be set high for an interrupt to be responded to.

The INT0 is a low-level-triggered interrupt by default, which means, when a low signal is applied to pin PD2 (PORTD.2), the controller will be interrupted and jump to location \$0002 in the vector table to service the ISR.

Example

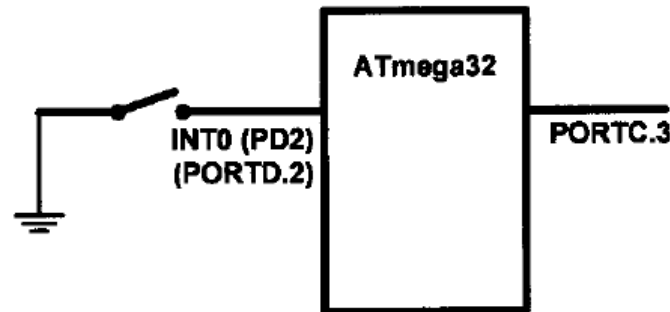
Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3 whenever the INT0 pin goes low.

```
.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
```



Example _{cnt.}

the microcontroller is looping continuously in the HERE loop. Whenever the switch on INT0 (pin PD2) is activated, the microcontroller gets out of the loop and jumps to vector location \$0002. The ISR for INT0 toggles the PC0. If, by the time it executes the RETI instruction, the INT0 pin is still low, the microcontroller initiates the interrupt again. Therefore, if we want the ISR to be executed once, the INT0 pin must be brought back to high before RETI is executed, or we should make the interrupt edge-triggered.



Edge-Triggered vs. Level-Triggered Interrupts

There are two types of activation for the external hardware interrupts: (1) level triggered, and (2) edge triggered. INT2 is only edge triggered, while INT0 and INT1 can be level or edge triggered.





As stated before, upon reset INT0 and INT1 are low-level-triggered interrupts. The bits of the MCUCR register indicate the trigger options of INT0 and INT1.







(Interrupt Sense Control bits) These bits define the level or edge on the external interrupt.

MCU Control Register (MCUCR)

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.

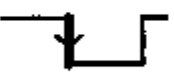
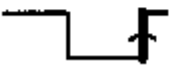
ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

MCU Control and Status Register (MCUCSR)

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



ISC2 This bit defines whether the INT2 interrupt activates on the falling edge or the rising edge.

ISC2		Description
0		The falling edge of INT2 generates an interrupt request.
1		The rising edge of INT2 generates an interrupt request.

Example

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

```
(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

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

```
.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 the 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. 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. For example, the INTF1 flag can be cleared using the following instructions:

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



Sampling the Edge-Triggered and the 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.

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

- What happens when two interrupts are activated at the same time?
 - Priority
 - 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 lower address has a higher priority

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, such as multitasking real-time operating systems (RTOS), 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, the program does two different tasks:

- (1) copying the contents of PORTC to PORTD,
- (2) toggling PORTC.2 every 5 μ s

While writing a program for a multitasking system, we should manage the resources carefully so that the tasks do not conflict with each other.

Example

consider a system that should perform the following tasks: (1) increasing the contents of PORTC continuously, and (2) increasing the content of PORTD once every 5 μ s. Read the following program. Does it work?

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP    MAIN
.ORG 0x14   ;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    R20,160
      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
```

Example cnt.

consider a system that should perform the following tasks: (1) increasing the contents of PORTC continuously, and (2) increasing the content of PORTD once every 5 μ s. Read the following program. Does it work?

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.

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.

Example cnt.

```
.INCLUDE "M32DEF.INC"
.ORG 0x0                ;location for reset
    JMP    MAIN

.ORG 0x14               ;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    R20,160
      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
```

Example cnt. (Context Saving)

(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*). See the following program:

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP    MAIN
.ORG 0x14    ;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    R20,160
      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:
    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
```


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 programming in C

In C language there is no instruction to manage the interrupts. So, in WinAVR the following have been added to manage the interrupts:

1. **Interrupt include file:** We should include the interrupt header file if we want to use interrupts in our program. Use the following instruction:
`#include <avr\interrupt.h>`
2. **cli() and sei():** In Assembly, the CLI and SEI instructions clear and set the I bit of the SREG register, respectively. In WinAVR, the `cli()` and `sei()` macros do the same tasks.
3. **Defining ISR:** To write an ISR (interrupt service routine) for an interrupt we use the following structure:

```
ISR(interrupt vector name)  
{  
    //our program  
}
```

Interrupt programming in C

Interrupt Vector Name for the ATmega32/ATmega16 in WinAVR

Interrupt	Vector Name in WinAVR
External Interrupt request 0	INT0_vect
External Interrupt request 1	INT1_vect
External Interrupt request 2	INT2_vect
Time/Counter2 Compare Match	TIMER2_COMP_vect
Time/Counter2 Overflow	TIMER2_OVF_vect
Time/Counter1 Capture Event	TIMER1_CAPT_vect
Time/Counter1 Compare Match A	TIMER1_COMPA_vect
Time/Counter1 Compare Match B	TIMER1_COMPB_vect
Time/Counter1 Overflow	TIMER1_OVF_vect
Time/Counter0 Compare Match	TIMER0_COMP_vect
Time/Counter0 Overflow	TIMER0_OVF_vect
SPI Transfer complete	SPI_STC_vect
USART, Receive complete	USART0_RX_vect
USART, Data Register Empty	USART0_UDRE_vect
USART, Transmit Complete	USART0_TX_vect
ADC Conversion complete	ADC_vect
EEPROM ready	EE_RDY_vect
Analog Comparator	ANALOG_COMP_vect
Two-wire Serial Interface	TWI_vect
Store Program Memory Ready	SPM_RDY_vect

Example

Using Timer0 generate a square wave on pin PORTB.5, while at the same time transferring data from PORTC to PORTD.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRB |= 0x20;           //DDRB.5 = output

    TCNT0 = -32;            //timer value for 4  $\mu$ s
    TCCR0 = 0x01;           //Normal mode, int clk, no prescaler

    TIMSK = (1<<TOIE0);    //enable Timer0 overflow interrupt
    sei ();                 //enable interrupts

    DDRC = 0x00;            //make PORTC input
    DDRD = 0xFF;            //make PORTD output

    while (1)               //wait here
        PORTD = PINC;

    ISR (TIMER0_OVF_vect)   //ISR for Timer0 overflow
    {
        TCNT0 = -32;
        PORTB ^= 0x20;      //toggle PORTB.5
    }
}
```

Example

Using Timer0 and Timer1 interrupts, generate square waves on pins PB1 and PB7 respectively, while transferring data from PORTC to PORTD.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ( )
{
    DDRB |= 0x82;          //make DDRB.1 and DDRB.7 output
    DDRC = 0x00;           //make PORTC input
    DDRD = 0xFF;           //make PORTD output

    TCNT0 = -160;
    TCCR0 = 0x01;          //Normal mode, int clk, no prescaler

    TCNT1H = (-640)>>8;    //the high byte
    TCNT1L = (-640);       //the low byte
    TCCR1A = 0x00;
    TCCR1B = 0x01;
    TIMSK = (1<<TOIE0)|(1<<TOIE1); //enable Timers 0 and 1 int.
    sei ();                //enable interrupts

    while (1)              //wait here
        PORTD = PINC;

    ISR (TIMER0_OVF_vect)  //ISR for Timer0 overflow
    {
        TCNT0 = -160;      //TCNT0 = -160 (reload for next round)
        PORTB ^= 0x02;     //toggle PORTB.1
    }

    ISR (TIMER1_OVF_vect)  //ISR for Timer0 overflow
    {
        TCNT1H = (-640)>>8;
        TCNT1L = (-640);   //TCNT1 = -640 (reload for next round)

        PORTB ^= 0x80;     //toggle PORTB.7
    }
}
```

Example

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.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRB |= 0x20;           //make DDRB.5 output

    OCR0 = 40;
    TCCR0 = 0x09;           //CTC mode, internal clk, no prescaler

    TIMSK = (1<<OCIE0);    //enable Timer0 compare match int.
    sei ();                 //enable interrupts

    DDRC = 0x00;           //make PORTC input
    DDRD = 0xFF;           //make PORTD output

    while (1)              //wait here
        PORTD = PINC;
}

ISR (TIMER0_COMP_vect)    //ISR for Timer0 compare match
{
    PORTB ^= 0x20;         //toggle PORTB.5
}
```

Example

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3, whenever INT0 pin goes low. Use the external interrupt in level-triggered mode.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRC = 1<<3;           //PC3 as an output
    PORTD = 1<<2;           //pull-up activated
    GICR = (1<<INT0);       //enable external interrupt 0
    sei ();                 //enable interrupts

    while (1);              //wait here
}

ISR (INT0_vect)             //ISR for external interrupt 0
{
    PORTC ^= (1<<3);        //toggle PORTC.3
}
```

پایان

موفق و پیروز باشید