AVR Microcontroller

Microprocessor Course

Chapter 10

AVR INTERRUPT PROGRAMMING

IN ASSEMBLY AND C

Day 1401(version 1.3)

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

Interrupts vs. polling

There are two methods by which devices receive service from the microcontroller: interrupts or polling.

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.

In the polling method, 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.

10.1 AVR INTERRUPT

Interrupt Service Routine (ISR)

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.

Interrupt Vector Table (IVT)

The group of memory locations set aside to hold the addresses of ISRs is called the Interrupt Vector Table (IVT).

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

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

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

Table 18. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

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.

10.1 AVR INTERRUPT

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).

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

The AVR has many more interrupts than the list shows. Notice in Table 10-1. that 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.

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

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

Figure 10-1. Redirecting the AVR from the Interrupt Vector Table at Power-up

10.1 AVR INTERRUPT

Enabling and disabling an interrupt

Upon reset, all interrupts are disabled (masked), 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.

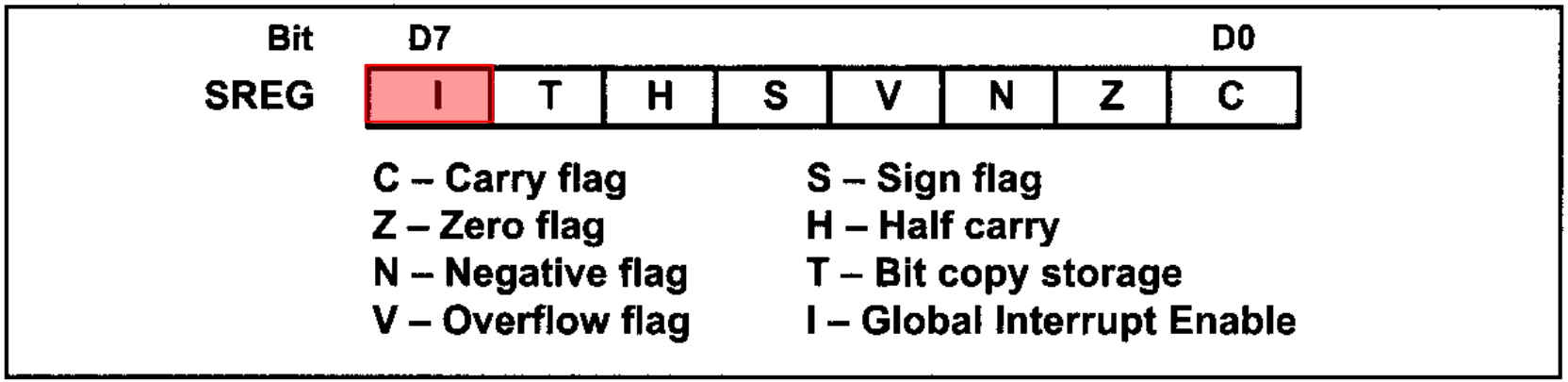


Figure 10-2. Bits of Status Register (SREG)

10.1 AVR INTERRUPT

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 10-3 shows that the TIMSK register has interrupt enable bits for Timed, Timer1, and Timer2. It must be noted that if I = 0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.1 AVR INTERRUPT

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:

```
1   (a)
2           LDI     R20,(1<<TOIE0)|(1<<OCIE2)       ;TOIE0 = 1, OCIE2 = 1
3           OUT     TIMSK,R20                       ;enable Timer0 overflow and Time
4           SEI                                     ;allow interrupts to come in
5   (b)
6           IN      R20,TIMSK                       ;R20 = TIMSK
7           ANDI    R20,0xFF ^ (1<<TOIE0)           ;TOIE0 = 0
8           OUT     TIMSK,R20                       ;mask (disable) Timer0 interrupt
```

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

```
10          IN      R20,TIMSK                       ;R20 = TIMSK
11          CBR     R20,1<<TOIE0                    ;TOIE0 = 0
12          OUT     TIMSK,R20                       ;mask (disable) Timer0 interrupt
13
14  (c)     CLI                                     ;mask all interrupts globally
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.1 AVR INTERRUPT

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable
When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow Interrupt is enabled. The corresponding Interrupt Vector is executed when the TOV1 Flag, located in TIFR, is set.

Bit 4 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable
When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1A Flag, located in TIFR, is set.

10.2 PROGRAMMING TIMER INTERRUPTS

Rollover timer flag and interrupt

In polling TOV0, we have to wait until TOV0 is raised.

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.

The TOIEx bit enables the interrupt for a given timer. TOIEx bits are held by the TIMSK register.

Table 10-2: 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

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

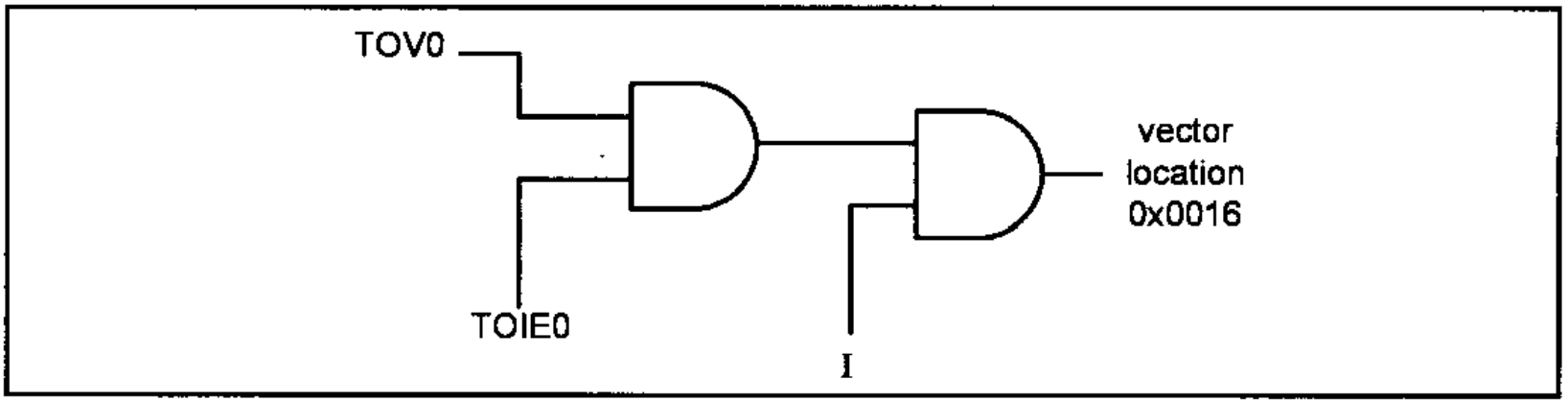


Figure 10-4. The Role of Timer Overflow Interrupt Enable (TOIE0)

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

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.

```
1   ;Program 10-1
2   .INCLUDE "M32DEF.INC"
3   .ORG 0x0                          ;location for reset
4           JMP     MAIN
5   .ORG 0x16                         ;location for Timer0 overflow (see Table 10.1)
6           JMP     TOV_ISR           ;jump to ISR for Timer0
7           ;-main program for initialization and keeping CPU busy
8           .ORG 0x100
9           MAIN:
10          LDI     R20,HIGH(RAMEND)
11          OUT     SPH,R20
12          LDI     R20,LOW(RAMEND)
13          OUT     SPL,R20           ;initialize stack
14          SBI     DDRB,5            ;PBS as an output
15          LDI     R20,(1<<TOIE0)
16          OUT     TIMSK,R20         ;enable Timer() overflow interrupt
17          SEI                       ;set I (enable interrupts globally)
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
17              SEI                     ;set I (enable interrupts globally)
18              LDI     R20,-32         ;timer value for 4 micros
19              OUT     TONT0,R20       ;load Timer0 with -32
20              LDI     R20,0x01
21              OUT     TCCR0,R20       ;Normal, internal clock, no prescaler
22              LDI     R20,0x00
23              OUT     DDRC,R20        ;make PORTC input
24              LDI     R20,0xFF
25              OUT     DDRD,R20        ;make PORTD output
26      ;------------------------------ Infinite loop
27      HERE:   IN      R20, PINC       ;read from PORTC
28              OUT     PORTD,R20       ;give it to PORTD
29              JMP     HERE            ;keeping CPU busy waiting for interrupt
30
31      ;------------------------------ ISR for Timer() (it is executed every 4 micros)
32      .ORG 0x200
33      TOV_ISR:
34              IN      R16,PORTB       ;read PORTB
35              LDI     R17,0x20        ;00100000 for toggling PB5
36              EOR     R16,R17         ;toggle PB5
37              OUT     PORTB,R16
38              LDI     R16,-32         ;timer value for 4 micross
39              OUT     TCNT0,R16       ;load Timer0 with -32 (for next round)
40              RETI                    ;return from interrupt
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

Notice the following points about Program 10- 1:

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. 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.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

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.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

Example 10-2

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

Solution:

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.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

Program 10-2 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.

```
1   ;Program 10-3
2   .INCLUDE "M32DEF.INC"
3   .ORG    0x0                     ;location for reset
4           JMP     MAIN            ;bypass interrupt vector table
5   .ORG    0x12                    ;ISR location for Timer0 overflow
6           JMP     T1_OV_ISR       ;go to an address with more space
7   .ORG    0x16
8           JMP     T0 OV ISR
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
10  ;---------- main program for initialization and keeping CPU busy
11  .ORG 0x100
12  MAIN:   LDI     R20,HIGH(RAMEND)
13          OUT     SPH,R20
14          LDI     R20,LOW(RAMEND)
15          OUT     SPL,R20
16          ;Init Timer 0
17          LDI     R20,-160
18          OUT     TCNT0,R20
19          LDI     R20,0x01
20          OUT     TCCR0,R20
21          ;Init Timer 1
22          LDI     R20,HIGH(-640)
23          OUT     TCNT1H,R20
24          LDI     R20,LOW(-640)
25          OUT     TCNT1L,R20
26          LDI     R20,0x00
27          OUT     TCCR1A,R20
28          LDI     R20,0x01
29          OUT     TCCR1B,R20
30          LDI     R20,(1<<TOIE0)|(1<<TOIE1)
31          OUT     TIMSK,R20
32          SEI
```

```
33          ;Init PORT B & C & I
34          SBI     DDRB,1
35          SBI     DDRB,7
36          LDI     R20,0
37          OUT     DDRC,R20
38          LDI     R20,0xFF
39          OUT     DDRD,R20
40          OUT     PORTC,R20
41  ;---------- Infinite loop
42  HERE:   IN      R20,PINC
43          OUT     PORTD,R20
44          RJMP    HERE
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
45  ;---------- ISR for Timer0
46  .ORG 0x200
47  T0_OV_ISR:
48          LDI     R16,-160
49          OUT     TCNT0,R16
50          IN      R17,PINB
51          LDI     R16,0x02
52          EOR     R16,R17
53          OUT     PORTB,R16
54          RETI
55  ;---------- ISR for Timer1
56  .ORG 0x220
57  T1_OV_ISR:
58          LDI     R18,HIGH(-640)
59          OUT     TCNT1H,R18
60          LDI     R18,LOW(-640)
61          OUT     TCNT1L,R18
62          IN      R18,PINB
63          LDI     R19,0x80
64          EOR     R18,R19
65          OUT     PORTB,R18
66          RETI
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

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.

```
1   ;Program 10-03
2   .INCLUDE "M32DEF.INC"
3   .ORG 0x0
4           JMP     MAIN
5   .ORG 0x12
6           JMP     T1_OV_ISR
7   .ORG 0x16
8           JMP     T0_OV_ISR
9
10  .ORG 0x40
11  MAIN:   LDI     R20,HIGH(RAMEND)
12          OUT     SPH,R20
13          LDI     R20,LOW(RAMEND)
14          OUT     SPL,R20
15          LDI     R18,0
16          OUT     PORTA,R18
17          LDI     R20,0
18          OUT     DDRC,R20
19          LDI     R20,0xFF
20          OUT     DDRA,R20
21          OUT     DDRD,R20
22          SBI     DDRB,6
23          SBI     PORTB,0
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
25          LDI     R20,0x06
26          OUT     TCCR0,R20
27          LDI     R16,-200
28          OUT     TCNT0,R16
29          LDI     R19,HIGH(-31250)
30          OUT     TCNT1H,R19
31          LDI     R19,LOW(-31250)
32          OUT     TCNT1L,R19
33          LDI     R20,0
34          OUT     TCCR1A,R20
35          LDI     R20,4
36          OUT     TCCR1B,R20
37          LDI     R20,(1<<TOIE0)|(1<<TOIE1)
38          OUT     TIMSK,R20
39          SEI
40  ;---------- Infinite loop
41  HERE:   IN      R20,PINC
42          OUT     PORTD,R20
43          JMP     HERE
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
;---------- ISR for Timer0
.ORG 0x200
T0_OV_ISR:
        IN      R16,PINB
        LDI     R17,0x40
        EOR     R16,R17
        OUT     PORTB,R16
        LDI     R16,-200
        OUT     TCNT0,R16
        RETI
;---------- ISR for Timer1
.ORG 0x300
T1_OV_ISR:
        INC     R18
        OUT     PORTA,R18
        LDI     R19,HIGH(-31250)
        OUT     TCNT1H,R19
        LDI     R19,LOW(-31250)
        OUT     TCNT1L,R19
        RETI
```

Compare match timer flag and interrupt

Sometimes a task should be done periodically, as in the previous examples. 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, which causes the compare match interrupt to occur.

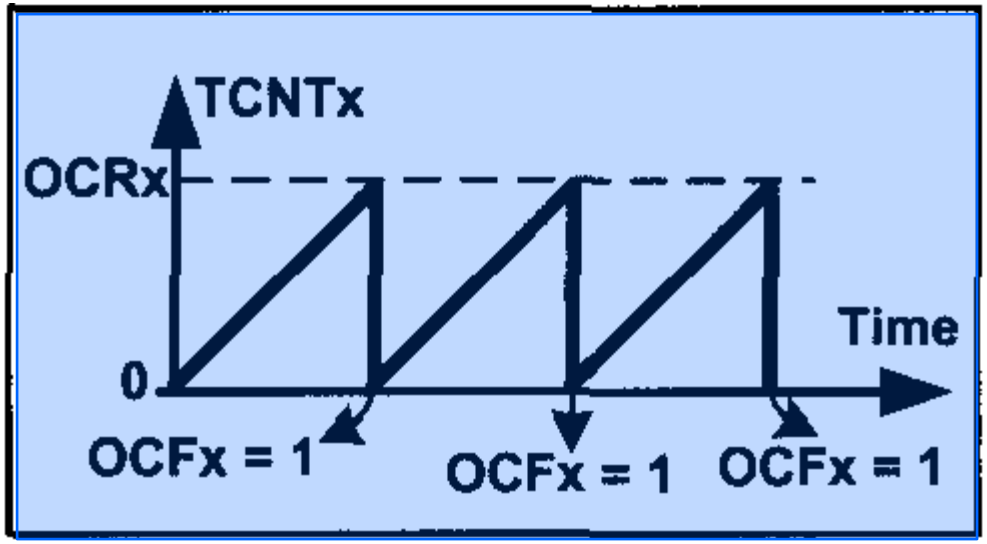


Figure 10-5. CTC Mode

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

Example 10-3

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

Solution:

1/1 MHz = 1 us and 40 us/1 us = 40.

That means we must have OCR0 = 40 - 1 = 39

```
1   .INCLUDE "M32DEF.INC"
2   .ORG 0x0                        ;location for reset
3           JMP     MAIN
4   .ORG 0x14                       ;ISR location for Timer0 compare match
5           JMP     T0_OV_ISR       ;main program for initialization and keeping CPU busy
6   .ORG 0x100
7   MAIN:   LDI     R20,HIGH(RAMEND)
8           OUT     SPH,R20
9           LDI     R20,LOW(RAMEND)
10          OUT     SPL,R20         ;set up stack
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.2 PROGRAMMING TIMER INTERRUPTS

```
11          SBI     DDRB,5          ;PB5 as an output
12          LDI     R20,(1<<OCIE0)
13          OUT     TIMSK,R20       ;enable Timer0 compare match interrupt
14          SEI                     ;set I (enable interrupts globally)
15          LDI     R20,39
16          OUT     OCR0,R20        ;load Timer0 with 39
17          LDI     R20,0x09
18          OUT     TCCR0,R20       ;start Timer0, CTC mode, int clk, no prescaler
19          LDI     R20,0x00
20          OUT     DDRC,R20        ;make PORTC input
21          LDI     R20,0xFF
22          OUT     DDRD,R20        ;make PORTD output
23  ;---------- Infinite loop
24  HERE:   IN      R20,PINC        ;read from PORTC
25          OUT     PORTD,R20       ;and send it to PORTD
26          JMP     HERE            ;keeping CPU busy waiting for interrupt
27  ;---------- ISR for Timer0 (it is executed every 40 us)
28  T0_OV_ISR:
29          IN      R16,PORTB       ;read PORTB
30          LDI     R17,0x20        ;00100000 for toggling PB5
31          EOR     R16,R17
32          OUT     PORTB,R16       ;toggle PB5
33          RETI    ;return from interrupt
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.2 PROGRAMMING TIMER INTERRUPTS

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_{clock} = (1/8 MHz) × 1024 = 128 µs and 1s/128 µs = 7812. That means we must have OCR1A = 7811 = 0x1E83

```
1   .INCLUDE "M32DEF.INC"
2   .ORG 0x0                              ;location for reset
3           JMP     MAIN
4   .ORG OC1Aaddr                             ;location for Timer1 compare match
5           JMP     T1_OC_ISR
6   ;---------- main program for initialization and keeping CPU busy
7   MAIN:   LDI     R20,HIGH(RAMEND)
8           OUT     SPH,R20
9           LDI     R20,LOW(RAMEND)
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

```
11          SBI     DDRB,5              ;PB5 as an output
12          LDI     R20,(1<<OCIE1A)
13          OUT     TIMSK,R20           ;enable Timer1 compare match interrupt
14          SEI                         ;set I (enable interrupts globally)
15          LDI     R20,0x00
16          OUT     TCCR1A,R20
17          LDI     R20,0xD
18          OUT     TCCR1B,R20          ;prescaler 1:1024, CTC mode
19          LDI     R20,HIGH(7811)      ;the high byte
20          OUT     OCR1AH,R20          ;Temp = Ox1E (high byte of 7811)
21          LDI     R20,LOW(7811)       ;the low byte
22          OUT     OCR1AL,R20          ;OCR1A = 7811
23          LDI     R20,0x00
24          OUT     DDRC,R20            ;make PORTC input
25          LDI     R20,0xFF
26          OUT     DDRD,R20            ;make PORTD output
27  ;---------- Infinite loop
28  HERE:   IN      R20,PINC            ;read from PORTC
29          OUT     PORTD,R20           ;PORTD = R20
30          JMP     HERE                ;keeping CPU busy waiting for interrupt
31  ;---------- ISR for Timer1 (It comes here after elapse of 1 second time)
32  T1_OC_ISR:
33          IN      R16, PORTB
34          LDI     R17,0x20            ;00100000 for toggling PB5
35          EOR     R16,R17
36          OUT     PORTB,R16           ;toggle PB5
37          RETI                        ;return from interrupt
```

10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

The number of external hardware interrupt interrupts varies in different AVRs. The ATmega32 has three external hardware interrupts:

1. Pin PD2 (PORTD.2), INT0 ----- Vector location 0x02
2. Pin PD3 (PORTD.3), INT1 ----- Vector location 0x04
3. Pin PB2 (PORTB.2), INT2 ----- Vector location 0x06

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 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
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

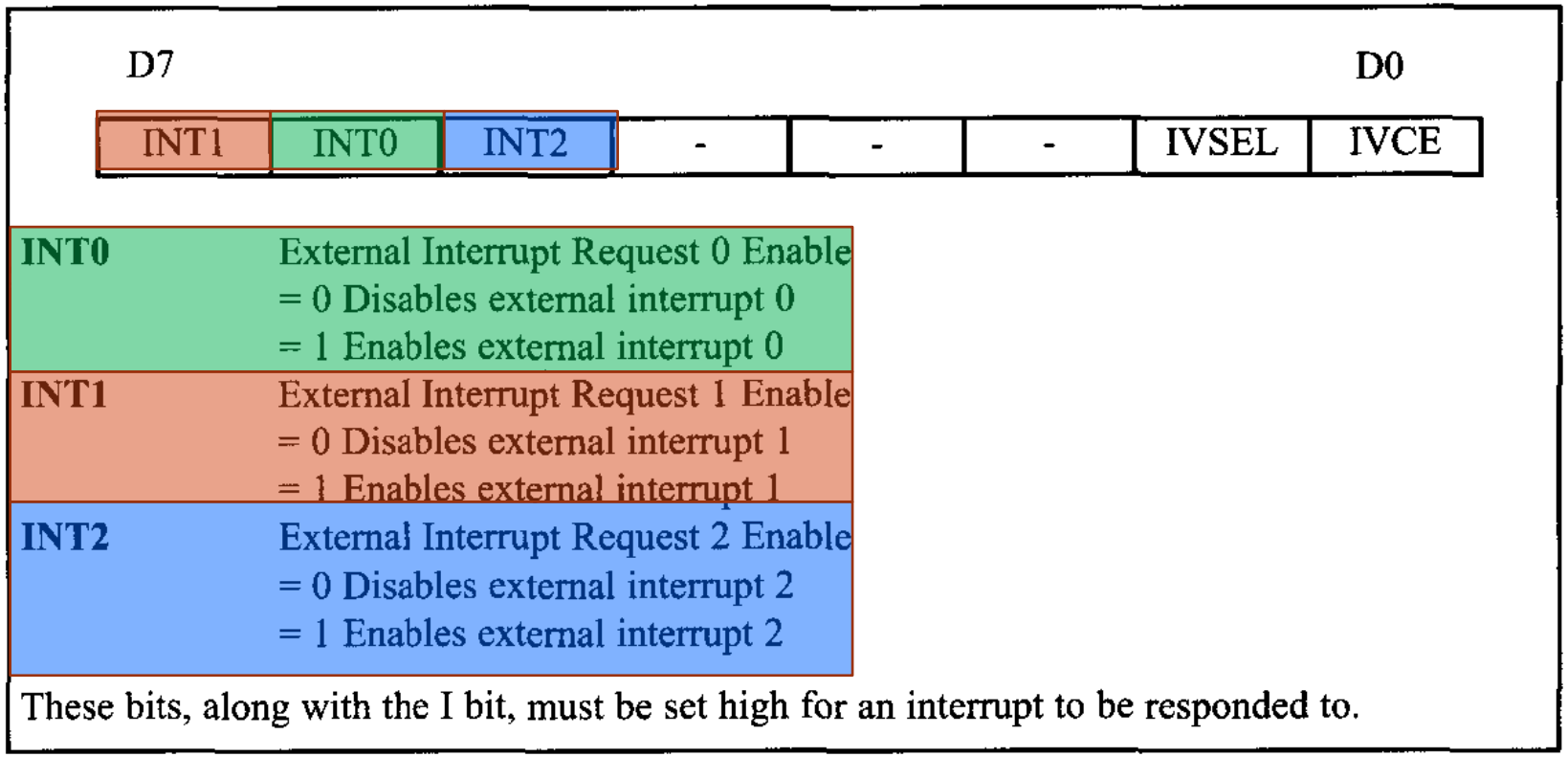


Figure 10-6. GICR (General Interrupt Control Register) Register

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

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.

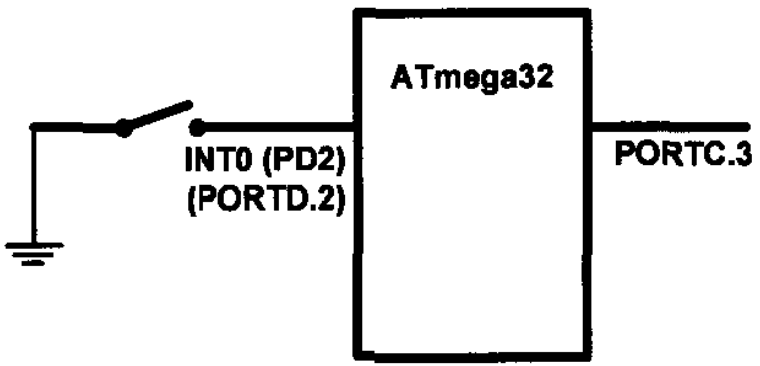
In this program, the microcontroller is looping continuously in the HERE loop. Whenever the switch on INT0 (pin PD2) is activated, the microcontroller gets out of the loop and jumps to vector location $0002. The ISR for INT0 toggles the PC0. If, by the time it executes the RETI instruction, the INT0 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.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

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.

```
1   .INCLUDE "M32DEF.INC"
2   .ORG    0                       ;location for reset
3           JMP     MAIN
4   .ORG INT0addr                   ;vector location for external interrupt 0
5           JMP     EX0_ISR
6   MAIN:   LDI     R20,HIGH(RAMEND)
7           OUT     SPH,R20
8           LDI     R20,LOW(RAMEND)
9           OUT     SPL,R20         ;initialize stack
10          SBI     DDRC,3
11          SBI     PORTD,2
12          LDI     R20,1<<INT0
13          OUT     GICR,R20
14          SEI
15  HERE:   JMP     HERE
16
17  EX0_ISR:
18          IN      R21,PINC
19          LDI     R22,0x08
20          EOR     R21,R22
21          OUT     PORTC,R21
22          RETI
```



Edge-triggered vs. level-triggered interrupts

There are two types of activation for the external hardware interrupts:

(1) level triggered, INT0 and INT1 can be level or edge triggered.
(2) edge triggered, INT2 is only 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.

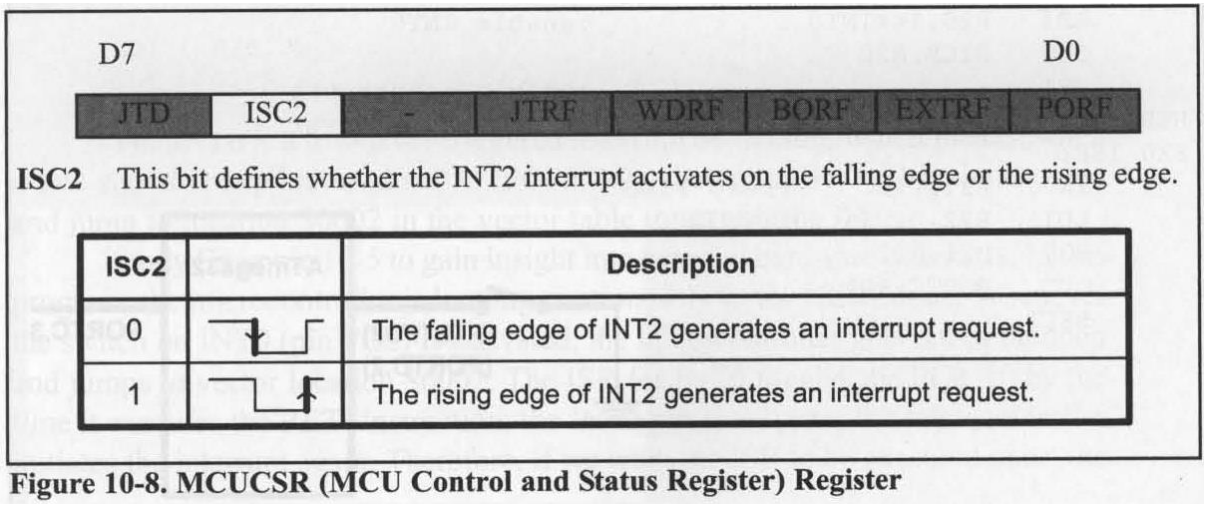
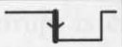
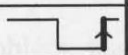


Figure 10-8. MCUCSR (MCU Control and Status Register) Register

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

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.

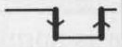
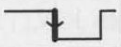
ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

Figure 10-7. MCUCR (MCU Control Register) Register

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

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.

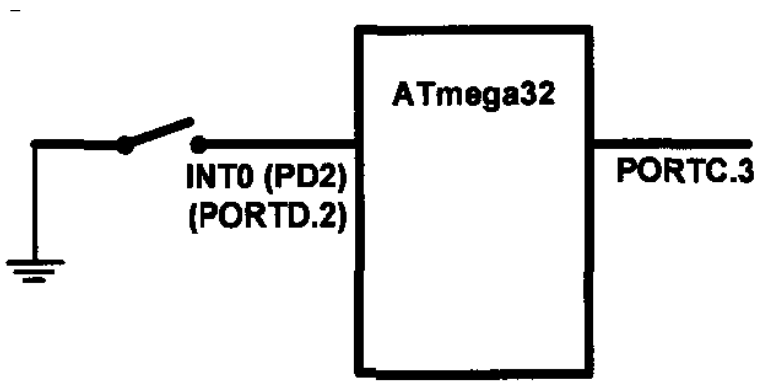
Solution:

```
(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
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

Example 10-7

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




```
1   .INCLUDE "M32DEF.INC"
2   .ORG    0
3           JMP     MAIN
4   .ORG 0x02
5           JMP     EX0_ISR
6   MAIN:   LDI     R20,HIGH(RAMEND)
7           OUT     SPH,R20
8           LDI     R20,LOW(RAMEND)
9           OUT     SPL,R20
10          LDI     R20,0x02
11          OUT     MCUCR,R20
12          SBI     DDRC,3
13          SBI     PORTD,2
14          LDI     R20,1<<INT0
15          OUT     GICR,R20
16          SEI
17  HERE:   JMP     HERE
18
19  EX0_ISR:
20          IN      R21,PINC
21          LDI     R22,0x08
22          EOR     R21,R22
23          OUT     PORTC,R21
24          RETI
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

Sampling the edge-triggered and level-triggered interrupts

Examine Figure 10-9. 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.

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.

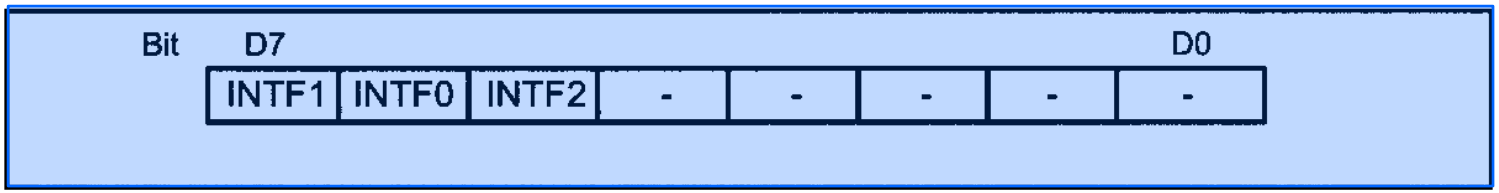


Figure 10-9. GIFR (General Interrupt Flag Register) Register

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.3 PROGRAMMING EXTERNAL HARDWARE 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.

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.

10.4 INTERRUPT PRIORITY IN THE AVR

Interrupt priority

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 0x0002, while the address of external interrupt 2 is 0x0006; thus, external interrupt 0 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

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, in Example 10-3, the program does two different tasks:

1. copying the contents of PORTC to PORTD,
2. toggling PORTB.2 every 5 μs

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:

(1) increasing the contents of PORTC continuously, and
(2) increasing the content of PORTB once every 5 μs.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.4 INTERRUPT PRIORITY IN THE AVR

Read the following program. Does it work?

```
; Program 10-4
.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
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.4 INTERRUPT PRIORITY IN THE AVR

```
25    HERE:   OUT     PORTC,R20           ;PORTC = R20
26            INC     R20
27            JMP     HERE                ;keeping CPU busy waiting for interrupt
28    ;---------- ISR for Timer0
29    T0_CM_ISR:
30            IN      R20,PIND
31            INC     R20
32            OUT     PORTD,R20           ;PORTD = R20
33            RETI                        ;return from interrupt
```

The task do not work properly, since they have resource conflict and they interfere with each other.

R20 is used by both tasks, which causes the program not to work properly.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.4 INTERRUPT PRIORITY IN THE AVR

We can solve such problems in the following two ways:

1. Using different registers for different tasks. If we use different registers in the main program and in the ISR, the program will work properly.

```
28  ;---------- ISR for Timer0
29  T0_CM_ISR:
30          IN      R20,PIND
31          INC     R20
32          OUT     PORTD,R20           ;PORTD = R20
33          RETI                        ;return from interrupt
```

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).***

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.4 INTERRUPT PRIORITY IN THE AVR

```
1   ; Program 10-6
2   .INCLUDE "M32DEF.INC"
3   .ORG 0x0                                ;location for reset
4           JMP     MAIN
5   .ORG 0x14                               ;location for Timer0 compare match
6           JMP     T0_CM_ISR
7   ;--------- main program for initialization and keeping CPU busy
8   .ORG 0x100
9   MAIN:   LDI     R20,HIGH(RAMEND)
10          OUT     SPH,R20
11          LDI     R20,LOW(RAMEND)
12          OUT     SPL,R20                 ;set up stack
13          SBI     DDRB,5                  ;PB5 as an output
14          LDI     R20,(1<<OCIE0)
15          OUT     TIMSK,R20               ;enable Timer0 compare match interrupt
16          SEI                             ;set I (enable interrupts globally)
17          LDI     R20,160
18          OUT     OCR0,R20                ;load Timer0 with 160
19          LDI     R20,0x09
20          OUT     TCCR0,R20               ;CTC mode, int clk, no prescaler
21          LDI     R20,0xFF
22          OUT     DDRC,R20                ;make PORTC output
23          OUT     DDRD,R20                ;make PORTD output
24          LDI     R20,0
25  HERE:   OUT     PORTC,R20               ;PORTC = R20
26          INC     R20
27          JMP     HERE                    ;keeping CPU busy waiting for interrupt
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.4 INTERRUPT PRIORITY IN THE AVR

```
28  ;---------- ISR for Timer0
29  T0_CM_ISR:
30          PUSH    R20                 ;save R20 on stack
31          IN      R20,PIND
32          INC     R20
33          OUT     PORTD,R20           ;PORTD = R20
34          POP     R20                 ;restore value for R20
35          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 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
```

Figure 10-10. Saving the SREG Register

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 olny one instruction cycle.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 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.

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

Table 10-3: 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

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

3. Defining ISR: To write an ISR (interrupt service routine) for an interrupt we use the following structure:

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

For the interrupt vector name we must use the ISR names in Table 10-3. For example, the following ISR serves the Timer0 compare match interrupt:

```
ISR (TIMER0_COMP_vect)
{
}
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

Example 10-8 (C version of program 10-1)

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

Solution:

```
1   #include navr/io.h"
2   #include "avr/interrupt.h"
3   int main ()
4   {
5           DDRB |= 0x20;                   //DDRB.5 = output
6           TCNT0 = -32;                    //timer value for 4 us
7           TCCR0 = 0x01;                   //Normal mode, int clk, no prescaler
8           TIMSK = (1<<TOIE0);             //enable Timer0 overflow interrupt
9           sei();                          //enable interrupts
10          DDRC = 0x00;                    //make PORTC input
11          DDRD = 0xFF;                    //make PORTD output
12          while (1)                       //wait here
13              PORTD = PINC;
14  }
15  ISR (TIMER0_OVF_vect)           //ISR for Timer0 overflow
16  {
17          TCNT0 = -32;
18          PORTB ^= 0x20;          //toggle PORTB.5
19  }
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

Context saving

The C compiler automatically adds instructions to the beginning of the ISRs, which save the contents of all of the general purpose registers and the SREG register on the stack. Some instructions are also added to the end of the ISRs to reload the registers.

10.5 INTERRUPT PROGRAMMING IN C

Example 10-9 (C version of Program 10-2)

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

```
1   #include navr/io.h"
2   #include "avr/interrupt.h"
3   int main()
4   {
5           DDRB |= 0x82;
6           DDRC = 0x00;
7           DDRD = 0xFF;
8           TCNT0 = -160;
9           TCCR0 = 0x01; //Normal mode, int clk, no prescaler
10          TCNT1H = (-640)>>8;
11          TCNT1L = (-640);
12          TCCR1A = 0x00;
13          TCCR1B = 0x01;
14          TIMSK = (1<<TOIE0)|(1<<TOIE1);
15          sei();
16          while (1)
17              PORTD = PINC;
18  }
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

```
19  ISR (TIMER0_OVF_vect)    //ISR for Timer0 overflow
20  {
21      TCNT0 = -160;
22      PORTB ^= 0x02;
23  }
24  ISR (TIMER1_OVF_vect) //ISR for Timer1 overflow
25  {
26      TCNT1H = (-640)>>8;
27      TCNT1L = (-640);
28      PORTB ^= 0x80;
29  }
```

Note: We can use "**TCNT1 = -640;**" in place of the following instructions:

```
TCNT1H = (-640)>>8;
TCNT1L = (-640) ;
```

10.5 INTERRUPT PROGRAMMING IN C

Example 10-10 (C version of program 10-3)

Using Timer0 and Timer1 interrupts, write a program in which:

(a) PORTA counts up every time Timer1 overflows. It overflows once per second.

(b) A pulse is fed into Time0 where Timer0 is used as counter and counts up. whenever the counter reaches 200, it will toggle the pin PORTB.6.

```
#include "avr/io.h"
#include "avr/interrupt.h"
int main()
{
        DDRA = 0xFF;
        DDRC = 0x00;
        DDRD = 0xFF;
        DDRB |= 0x40;
        PORTB |= 0x01;
        TCNT0 = -200;
        TCCR0 = 0x06;
        TCNT1H = (-31250)>>8;
        TCNT1L = (-31250)& 0xFF;
        TCCR1A = 0x00;
        TCCR1B = 0x04;
        TIMSK = (1<<TOIE0)|(1<<TOIE1);
        sei();

            while (1)
            PORTD = PINC;
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C
10.5 INTERRUPT PROGRAMMING IN C

Example 10-10 (C version of program 10-3)

```
22  ISR (TIMER0_OVF_vect)//ISR for Timer() overflow
23  {
24      TCNT0 = -200;
25      PORTB ^= 0x40;
26  }
27  ISR (TIMER1_OVF_vect)
28  {
29      TCNT1H = (-31250)>>8;
30      TCNTIL = (-31250)&0xFF;
31      PORTA ++;
32  }
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.5 INTERRUPT PROGRAMMING IN C

Example 10-11 (C version of program 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.

```
1   #include <mega32.h>
2   // Timer 0 output compare interrupt service routine
3   interrupt [TIM1_COMPA] void timer1_compa_isr(void)
4   {
5       PORTB ^= 0x20;
6   }
7   void main(void)
8   {
9   DDRB |= 0x20;
10  DDRC=0x00;
11  DDRD=0xFF;
12  TCCR1A=0x00;
13  TCCR1B=0x0C;
14  OCR1AH=(65535-31250)>>8;
15  OCR1AL=(65535-31250)&0xFF;
16  TIMSK=(1<<OCIE1A);
17  #asm("sei")
18
19  while (1)
20      {
21      PORTD = PINC;
22      }
23  }
```

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.5 INTERRUPT PROGRAMMING IN C

Example 10-12 (C version of program 10-5)

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

```
1   #include "avr/io.h"
2   #include "avr/interrupt.h"
3   int main()
4   {
5           DDRC = 1<<3;              //PC3 as an output
6           PORTD = 1<<2;             //pull-up activated
7           GICR = (1<<INT0);         //enable external interrupt 0
8           sei();                    //enable interrupts
9
10          while(1);                 //wait here
11  }
12  ISR (INT0_vect)                   //ISR for external interrupt 0
13  {
14          PORTC ^= (1<<3);          //toggle PORTC.3
15  }
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR INTERRUPT PROGRAMMING In ASSEMBLY AND C

10.5 INTERRUPT PROGRAMMING IN C

Example 10-13 (C version of program 10-7)

Rewrite Example 10-12 so that whenever INT0 goes low, it toggles PORTC.3 only once.

```
1   #include "avr/io.h"
2   #include "avr/interrupt.h"
3   int main()
4   {
5           DDRC = 1<<3;                    //PC3 as an output
6           PORTD = 1<<2;                   //pull-up activated
7           MCUCR = 0x02;                   //make INT0 falling edge triggered
8           GICR = (1<<INT0);               //enable external interrupt 0
9           sei();                          //enable interrupts
10          while (1);                      //wait here
11  }
12  ISR (INT0_vect)                         //ISR for external interrupt 0
13  {
14          PORTC ^= (1<<3);                //toggle PORTC.3
15  }
16
```