

CHAPTER 10

AVR INTERRUPT PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> Contrast and compare interrupts versus polling
- >> Explain the purpose of the ISR (interrupt service routine)
- >> List all the major interrupts of the AVR
- >> Explain the purpose of the interrupt vector table
- >> Enable or disable AVR interrupts
- >> Program the AVR timers using interrupts
- >> Describe the external hardware interrupts of the AVR
- >> Program the pin change interrupts
- >> Define the interrupt priority of the AVR
- >> Program AVR interrupts in C

In this chapter we explore the concept of the interrupt and interrupt programming. In Section 10.1, the basics of AVR interrupts are discussed. In Section 10.2, interrupts belonging to timers are discussed. External hardware interrupts are discussed in Section 10.3. In Section 2.4, we discuss Pin change interrupts. In Section 10.5, we cover interrupt priority. In Section 10.6, we provide AVR interrupt programming examples in C.

SECTION 10.1: AVR INTERRUPTS

In this section, we first examine the difference between polling and interrupt and then describe the various interrupts of the AVR.

Interrupts vs. polling

A single microcontroller can serve several devices. There are two ways to do that: interrupts or polling. In the *interrupt* method, whenever any device needs service, the device notifies the CPU by sending an interrupt signal. Upon receiving an interrupt signal, the CPU stops whatever it is doing and serves the device. In *polling*, the CPU 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 everyone is serviced. See Figure 10-1.

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 CPU time. The polling method wastes much of the CPU's time by polling devices when they do not need service. So, in order to avoid tying down the CPU, interrupts are used. For example, in discussing timers in Chapter 9 we used the bit test instruction "SBIS TIFR0, TOV0" and waited until the timer rolled over, and while we were waiting we could not do anything else. That is a waste of the CPU's time that could have been used to perform some useful tasks. In the case of the timer, if we use the interrupt method, the CPU can go about doing other tasks, and when the TOV0 flag is raised, the timer will interrupt the CPU to let it know that the time is elapsed.

Interrupt service routine

For every interrupt there must be a program associated with it. When an interrupt occurs this program is executed to perform certain services for the interrupt. This program is commonly referred to as an *interrupt service routine* (ISR). The interrupt service routine is also called the *interrupt handler*. When an interrupt occurs, the CPU runs the interrupt service routine. Now the question is how the ISR gets executed?

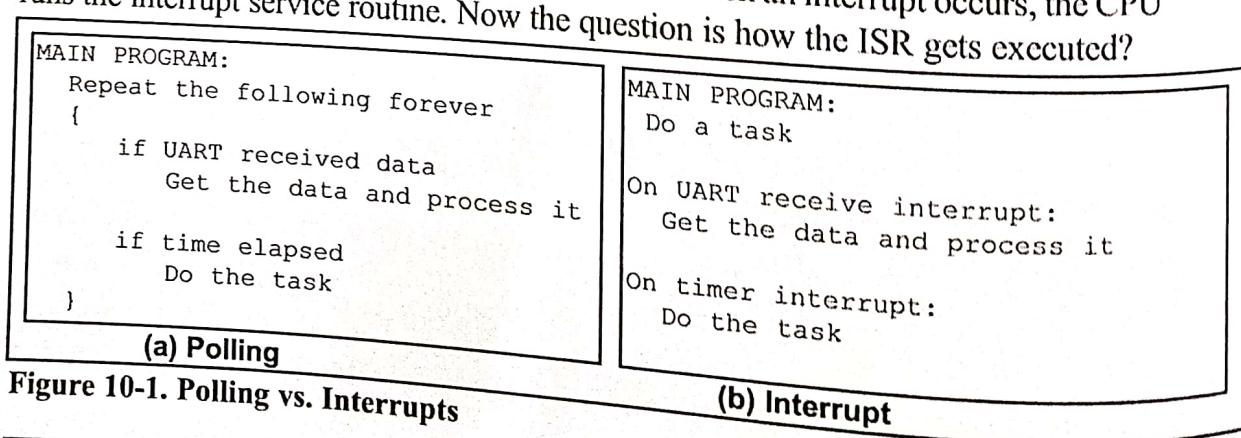


Figure 10-1. Polling vs. Interrupts

As shown in Figure 10-2, in most CPUs there are pins that are associated with interrupt controllers. They are input signals into the CPU. When the signals are triggered, CPU pushes the PC register onto the stack and loads the PC register with the address of the interrupt service routine. This causes the ISR to get executed.

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*. See Table 10-1.

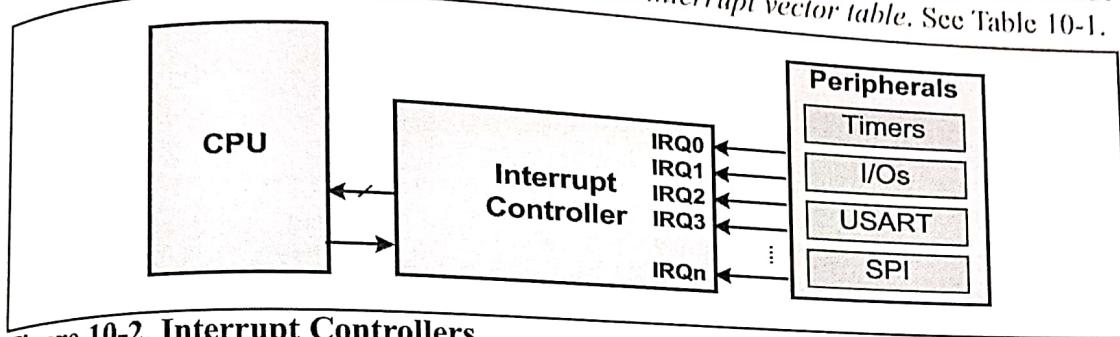


Figure 10-2. Interrupt Controllers

Table 10-1: Interrupt Vector Table for the ATmega328 AVR

Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt Request 0	0002
External Interrupt Request 1	0004
Pin Change Interrupt Request 0	0006
Pin Change Interrupt Request 1	0008
Pin Change Interrupt Request 2	000A
Watchdog Time-out Interrupt	000C
Time/Counter2 Compare Match A	000E
Time/Counter2 Compare Match B	0010
Time/Counter2 Overflow	0012
Time/Counter1 Capture Event	0014
Time/Counter1 Compare Match A	0016
Time/Counter1 Compare Match B	0018
Time/Counter1 Overflow	001A
Time/Counter0 Compare Match A	001C
Time/Counter0 Compare Match B	0020
Time/Counter0 Overflow	0022
SPI Serial Transfer complete	0024
USART, Receive Complete	0026
USART, Data Register Empty	0028
USART, Transmit Complete	002A
ADC Conversion Complete	002C
EEPROM Ready	002E
Analog Comparator	0030
Two-wire Serial Interface (I2C)	0032

Note: In Atmega328, you can move the interrupt vector table to the beginning of boot Loader Section. For more information see the datasheet.

Steps in executing an interrupt

Upon activation of an interrupt, the AVR 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.

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.

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. See Section 10.2.
2. In ATmega328, two interrupts are set aside for external hardware interrupts. Pins PD2 (PORTD.2) and PD3 (PORTD.3) are for the external hardware interrupts INT0 and INT1, respectively. See Section 10.3.
3. Serial communication's USART has three interrupts, one for receive and two interrupts for transmit. See Chapter 11.
4. The SPI interrupts. See Chapter 17.
5. The ADC (analog-to-digital converter). See Chapter 13.

The AVR has many more interrupts than the list shows. We will cover them throughout the book as we study the peripherals of the AVR. 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), locations 0020 and 0021, 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. In upcoming sections of this chapter, we will see many examples of interrupt programming that clarify these concepts.

From Table 10-1, also notice that only 2 words (4 bytes) of ROM space are assigned to the reset pin. They are ROM address locations 0–1. For this reason, in our program we put the JMP as the first instruction and redirect the processor away from the interrupt vector table, as shown in Figure 10-3. In the next section we will see how this works in the context of some examples.

```

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

;----- the wake-up program
.ORG $100

MAIN: .... ;enable interrupt flags
    ....

```

Figure 10-3. Redirecting the AVR from the Interrupt Vector Table at Power-up

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. Figure 10-4 shows the SREG register. 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.

Bit	D7	D0							
SREG	I	T	H	S	V	N	Z	C	
C – Carry flag					S – Sign flag				
Z – Zero flag					H – Half carry				
N – Negative flag					T – Bit copy storage				
V – Overflow flag					I – Global Interrupt Enable				

Figure 10-4. Bits of Status Register (SREG)

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 10-5 shows that the TIMSKn registers have interrupt enable bits for Timer0, Timer1, and Timer2. As we study each of peripherals throughout the book we will examine the registers holding the interrupt enable bits. It must be noted that if I = 0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high. To understand this important point look at Example 10-1.

Example 10-1

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

Example 10-1 (Cont.)

Solution:

(a) LDI R20, (1<<TOIE0) | (1<<OCIE0A) ;TOIE0 = 1, OCIE0A = 1
STS TIMSK0,R20 ;enable Timer0 overflow and compare match A
SEI ;allow interrupts to come in

(b) LDS R20,TIMSK0 ;R20 = TIMSK0
ANDI R20, 0xFF^(1<<TOIE0) ;TOIE0 = 0
STS TIMSK0,R20 ;mask (disable) Timer0 interrupt

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

LDS R20,TIMSK0 ;R20 = TIMSK
CBR R20, 1<<TOIE0 ;TOIE0 = 0
STS TIMSK0,R20 ;mask (disable) Timer0 interrupt

(c) CLI ;mask all interrupts globally

Notice that in part (a) we can use "LDI R20, 0x03" in place of the following instruction: "LDI R20, (1<<TOIE0) | (1<<OCIE0A)"

Review Questions

1. Of the interrupt and polling methods, which one avoids tying down the microcontroller?
2. Give the name of the interrupts in the TIMSK0 register.
3. Upon power-on reset of the ATmega328, what memory area is assigned to the interrupt vector table? Can the programmer change the memory space assigned to the table?
4. What is the content of D7 (I) of the SREG register upon reset, and what does it mean?
5. Show the instructions needed to enable the Timer1 compare A match interrupt.
6. What address in the interrupt vector table is assigned to the Timer1 overflow and INT0 interrupts?

SECTION 10.2: PROGRAMMING TIMER INTERRUPTS

In Chapter 9 we discussed how to use Timers 0, 1, and 2 with the polling method. In this section we use interrupts to program the AVR timers. Please review Chapter 9 before you study this section.

Rollover timer flag and interrupt

In Chapter 9 we stated that the timer overflow flag is raised when the timer rolls over. In that chapter, we also showed how to monitor the timer flag with the instruction "SBIS TIFR0, 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

	D7									D0
TIMSK0:	-	-	-	-	-	OCIE0B	OCIE0A	TOI0E		
TOI0E	Timer0 overflow interrupt enable = 0 Disables Timer0 overflow interrupt = 1 Enables Timer0 overflow interrupt									
OCIE0A	Timer0 output compare A match interrupt enable = 0 Disables Timer0 compare A match interrupt = 1 Enables Timer0 compare A match interrupt									
OCIE0B	Timer0 output compare B match interrupt enable = 0 Disables Timer0 compare B match interrupt = 1 Enables Timer0 compare B match interrupt									
	D7									D0
TIMSK1:	-	-	-	-	-	OCIE1B	OCIE1A	TOI1E		
TOI1E	Timer0 overflow interrupt enable = 0 Disables Timer0 overflow interrupt = 1 Enables Timer0 overflow interrupt									
OCIE1A	Timer0 output compare A match interrupt enable = 0 Disables Timer0 compare A match interrupt = 1 Enables Timer0 compare A match interrupt									
OCIE1B	Timer0 output compare B match interrupt enable = 0 Disables Timer0 compare B match interrupt = 1 Enables Timer0 compare B match interrupt									
ICIE1	Timer1 input capture interrupt enable = 0 Disables Timer1 input capture interrupt = 1 Enables Timer1 input capture interrupt									
	D7									D0
TIMSK2:	-	-	-	-	-	OCIE2B	OCIE2A	TOI2E		
TOI2E	Timer2 overflow interrupt enable = 0 Disables Timer2 overflow interrupt = 1 Enables Timer2 overflow interrupt									
OCIE2A	Timer2 output compare A match interrupt enable = 0 Disables Timer2 compare A match interrupt = 1 Enables Timer2 compare A match interrupt									
OCIE2B	Timer2 output compare B match interrupt enable = 0 Disables Timer2 compare B match interrupt = 1 Enables Timer2 compare B match interrupt									

These bits, along with the I bit, must be set high for an interrupt to be responded to.
Upon activation of the interrupt, the I bit is cleared by the AVR itself to make sure another interrupt cannot interrupt the microcontroller while it is servicing the current one. At the end of the ISR, the RETI instruction will make I = 1 to allow other interrupts to come in.

Figure 10-5. TIMSKn (Timer Interrupt Mask) Registers

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_n bit enables the interrupt for a given timer. TOIE_n bits are held by the TIMSK_n registers as shown in Table 10-2. See Figure 10-6 and Program 10-1.

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

Interrupt	Overflow Register	Interrupt Enable Bit	Register
Timer0	TOV0	TOIE0	TIMSK0
Timer1	TOV1	TOIE1	TIMSK1
Timer2	TOV2	TOIE2	TIMSK2

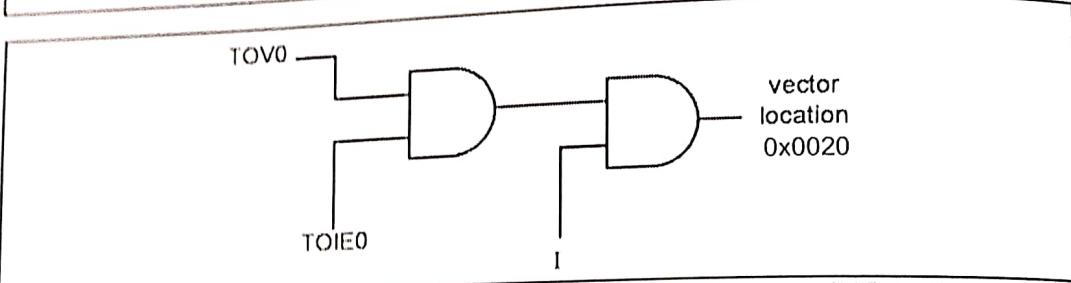


Figure 10-6. The Role of Timer Overflow Interrupt Enable (TOIE0)

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    R20, (1<<TOIE0)
STS    TIMSK0, R20 ;enable Timer0 overflow interrupt
SET              ;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 \$0020 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. In Section 10.6, we show how to allow an interrupt inside an interrupt.
4. The ISR for Timer0 is located starting at memory location \$200 because it is too large to fit into address space \$20-\$21, 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.

Program 10-1

```

/* Program10_1: For this program, we assume that PORTC is connected
   to 6 switches and PORTD to 6 LEDs. This program uses Timer0 to gen-
   erate a square wave on pin PORTB.5, while at the same time data is
   being transferred from PORTC to PORTD. */

.ORG 0x0          ;location for reset
JMP  MAIN

.ORG 0x20         ;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)
      STS  TIMSK0,R20        ;enable Timer0 overflow interrupt
      SEI
      LDI  R20,-32           ;set I (enable interrupts globally)
      OUT TCNT0,R20          ;timer value for 2 us
      LDI  R20,0x00
      OUT TCCR0A,R20
      LDI  R20,0x01
      OUT TCCR0B,R20          ;Normal, internal clock, no prescaler
      LDI  R20,0x00
      OUT DDRC,R20           ;make PORTC input
      LDI  R20,0xFF
      OUT PORTC,R20          ;enable pull-up resistors
      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

;-----ISR for Timer0 (it is executed every 4 µs)
.ORG 0x200
T0_OV_ISR:
      IN   R16,PORTB          ;read PORTB
      LDI  R17,(1<<5)        ;00100000 for toggling PB5
      EOR  R16,R17
      OUT PORTB,R16           ;toggle PB5
      LDI  R16,-32             ;timer value for 2 us
      OUT TCNT0,R16           ;load Timer0 with -32 (for next round)
      RETI                     ;return from interrupt

```

See Example 10-2 to understand the difference between RET and RETI.

Example 10-2

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.

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.

See Program 10-2. Program 10-2 uses Timer0 and Timer1 interrupts simultaneously, to generate square waves on pins PB1 and PB3 respectively, while data is being transferred from PORTC to PORTD.

Program 10-2

```
.ORG 0x0          ;location for reset
    JMP MAIN      ;bypass interrupt vector table
.ORG 0x1A         ;ISR location for Timer1 overflow
    JMP T1_OV_ISR ;go to an address with more space
.ORG 0x20         ;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,3        ;PB3 as an output
      LDI R20,0x00
      OUT DDRC,R20     ;make PORTC input
      LDI R20,0xFF
      OUT PORTC,R20    ;enable the pull-up resistors
      OUT DDRD,R20     ;make PORTD output

      LDI R20,-160      ;value for 10 us
      OUT TCNT0,R20    ;load Timer0 with -160
      LDI R20,0x00
      OUT TCCR0A,R20
      LDI R20,0x01
      OUT TCCR0B,R20    ;Normal mode, int clk, no prescaler
      LDI R20,HIGH(-1600) ;the high byte
      STS TCNT1H,R20    ;load Timer1 high byte
      LDI R20,LOW(-1600) ;the low byte
      STS TCNT1L,R20    ;load Timer1 low byte
      LDI R20,0x00
      STS TCCR1A,R20   ;Normal mode
      LDI R20,0x01
      STS TCCR1B,R20   ;internal clk, no prescaler
      LDI R20,(1<<TOIE0)
      STS TIMSK0,R20    ;enable Timer0 overflow interrupt
      LDI R20,(1<<TOIE1)
```

Program 10-2 (Cont.)

```

STS  TIMSK1,R20      ;enable Timer0 overflow interrupt
SET
;----- Infinite loop ;set I (enable interrupts globally)
HERE: IN  R20,PTINC   ;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 10 us time)
.ORG 0x200
T0_OV_ISR:
LDI  R16,-160        ;value for 10 us
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 100 us time)
.ORG 0x300
T1_OV_ISR:
LDI  R18,HIGH(-1600)
STS  TCNT1H,R18        ;load Timer1 high byte
LDI  R18,LOW(-1600)
STS  TCNT1L,R18        ;load Timer1 low byte (for next round)
IN   R18,PORTB        ;read PORTB
LDI  R19,(1<<3)       ;00001000 for toggling PB3
EOR  R18,R19
OUT PORTB,R18         ;toggle PB3
RETI                   ;return from interrupt

```

Notice that the addresses \$0100, \$0200, and \$0300 that we used in Program 10-2 are all arbitrary and can be changed to any addresses we want. The only addresses that we cannot change are the reset location of 0000, the Timer0 overflow address of \$0020, and the Timer1 overflow address of \$001A in the interrupt vector table because they were fixed at the time of the ATmega328 design.

Program 10-3 has two interrupts: (1) PORTC 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 counts 200 pulses, it will toggle the pin PORTB.5. (3) It toggles PORTB.1, continuously.

Program 10-3

```

;Program 10-3
.ORG 0x00                  ;location for reset
.JMP MAIN                    ;bypass interrupt vector table
.ORG 0x1A                  ;ISR location for Timer1 overflow
.JMP T1_OV_ISR              ;go to an address with more space
.ORG 0x20                  ;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 0x40
MAIN: LDI  R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI  R20,LOW(RAMEND)
      OUT SPL,R20      ;initialize SP

```

Program 10-3

```

LDI    R20, 0xFF      ; PORTC as output
OUT   DDRC, R20      ; R18 = 0
LDI    R18, 0          ; PORTC = 0
OUT   PORTC, R18      ; PB5 as output
SBI   DDRB, 5         ; activate pull-up of PD4 (T0)

SBI   PORTD, 4         ; activate pull-up of PD4 (T0)
LDI   R16, -200        ; load Timer0 with -200
OUT   TCNT0, R16        ; load Timer0 with -200
LDI   R20, 0x00        ; Normal, T0 pin falling edge, no scale
OUT   TCCR0A, R20        ; Normal, T0 pin falling edge, no scale
LDI   R20, 0x06        ; Normal, T0 pin falling edge, no scale
OUT   TCCR0B, R20        ; Normal, T0 pin falling edge, no scale
LDI   R20, (1<<TOIE0)
STS   TIMSK0, R20        ; enable Timer0 overflow interrupt

LDI   R19, HIGH(-62500)  ; timer value for 1 second
STS   TCNT1H, R19        ; load Timer1 high byte
LDI   R19, LOW(-62500)
STS   TCNT1L, R19        ; load Timer1 low byte
LDI   R20, 0
STS   TCCR1A, R20        ; Timer1 Normal mode
LDI   R20, 0x04        ; int clk, prescale 1:256
STS   TCCR1B, R20        ; int clk, prescale 1:256
LDI   R20, (1<<TOIE1)
STS   TIMSK1, R20        ; enable Timer1 overflow interrupt
SEI           ; set I (enable interrupts globally)

SBI   DDRB, 1         ; PB1 as an output
;----- Infinite loop
HERE: SBI   PORTB, 1
      CBI   PORTB, 1
      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, 1<<5    ; 0010 0000 for toggling PB5
      EOR   R16, R17
      OUT  PORTB, R16    ; toggle PB4
      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  PORTC, R18    ; display it on PORTC
      LDI   R19, HIGH(-62500)
      STS   TCNT1H, R19    ; load Timer1 high byte
      LDI   R19, LOW(-62500)
      STS   TCNT1L, R19    ; load Timer1 low byte (for next round)
      RETI             ; return from interrupt

```

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. Because the timer is in the CTC mode, the timer will be loaded with zero as well. So, the compare match interrupt occurs periodically. See Figure 10-7 and Examples 10-3 and 10-4. Notice that the AVR chip clears the OCF flag upon jumping to the interrupt vector table.

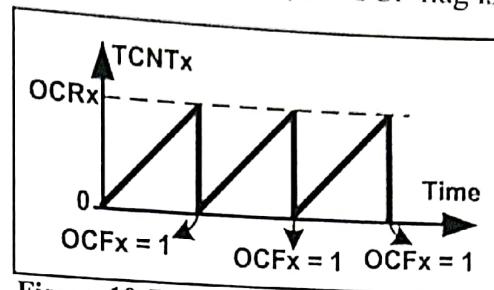


Figure 10-7. CTC Mode

Example 10-3

Using Timer0, write a program that toggles pin PORTB.5 every 15 μ s, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 16 MHz.

Solution:

$1/16 \text{ MHz} = 0.0625 \mu\text{s}$ and $15 \mu\text{s}/0.0625 \mu\text{s} = 240$. That means we must have $\text{OCR0A} = 240 - 1 = 239$

```
.ORG 0x0 ;location for reset
JMP MAIN
.ORG 0x1C ;ISR location for Timer0 compare match A
JMP T0_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, 239
OUT OCR0A, R20 ;load Timer0 with 239
LDI R20, (1<<WGM01)
OUT TCCR0A, R20
LDI R20, 0x01
OUT TCCR0B, R20 ;start Timer0, CTC mode, no prescaler int clk
LDI R20, (1<<OCIE0A)
STS TIMSK0, R20 ;enable Timer0 compare match interrupt
SEI ;set I (enable interrupts globally)

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
```

Example 10-3 (Cont.)

```
;-----ISR for Timer0 (it is executed every 40 µs)
TO_CM_ISR:
    IN    R16, PORTB      ;read PORTB
    LDI   R17, 1<<5      ;00100000 for toggling PB5
    EOR   R16, R17        ;toggle PB5
    OUT   PORTB, R16      ;return from interrupt
    RETI
```

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 = 16 MHz.

Solution:

For prescaler = 1024 we have $T_{Clock} = (1 / 16 \text{ MHz}) \times 1024 = 64 \mu\text{s}$ and $1 \text{ s}/64 \mu\text{s} = 15625$. That means we must have $OCR1A = 15624 = 0x3D08$

```
.ORG 0x0      ;location for reset
JMP  MAIN
.ORG 0x16     ;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,HIGH(15624)  ;the high byte
STS  OCR1AH,R20      ;Temp = 0x3D (high byte of 15624)
LDI  R20,LOW(15624)  ;the low byte
STS  OCR1AL,R20      ;OCR1A = 15624
LDI  R20,0x00
STS  TCCR1A,R20
LDI  R20,0xD
STS  TCCR1B,R20
LDI  R20,(1<<OCIE1A) ;prescaler 1:1024, CTC mode
STS  TIMSK1,R20
SEI               ;enable Timer1 compare match interrupt
                  ;set I (enable interrupts globally)

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      ;PORTD = R20
JMP  HERE           ;keeping CPU busy waiting for interrupt
```

Example 10-4 (Cont.)

```
;---ISR for Timer1 (It comes here after elapse of 1 second time)
T1_CM_ISR:
    IN    R16, PORTB
    LDI   R17, (1<<5) ;00100000 for toggling PB5
    EOR   R16, R17
    OUT   PORTB, R16 ;toggle PB5
    RETI              ;return from interrupt
```

Review Questions

1. True or false. There is a single interrupt in the interrupt vector table assigned to both Timer0 and Timer1.
2. What address in the interrupt vector table is assigned to Timer0 overflow?
3. Which register does TOIE1 belong to? Show how it is enabled.
4. Assume that Timer0 is programmed in Normal mode, TCNT0 = 0xF1, and the TOIE0 bit is enabled. Explain how the interrupt for the timer works.
5. True or false. The last two instructions of the ISR for Timer0 are:
OUT TIFR0, 1<<TOV0 ;clear TOV0 flag
RETI
6. Assume that Timer0 is programmed in CTC mode, OCR0A = 21, and the compare match interrupt is enabled. Explain how the interrupt for the timer works.
7. In the previous problem, assume XTAL = 8 MHz, and the timer is in no prescaler mode. How often is the ISR executed?

SECTION 10.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

The number of external hardware interrupts varies in different AVR. The ATmega328 has two external hardware interrupts: pins PD2 (PORTD.2) and PD3 (PORTD.3), designated as INT0 and INT1, 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. In this section we study these two external hardware interrupts of the AVR with some examples in Assembly language.

External interrupts INT0 and INT1

There are two external hardware interrupts in the ATmega328: INT0 and INT1. They are located on pins PD2 and PD3, respectively. As we saw in Table 10-1, the interrupt vector table locations \$2 and \$4 are set aside for INT0 and INT1, respectively. The hardware interrupts must be enabled before they can take effect. This is done using the INTx bit located in the EIMSK register. See Figure 10-8. For example, the following instructions enable INT0:

```
LDI   R20, 0x01
STS   EIMSK, R20
```

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.

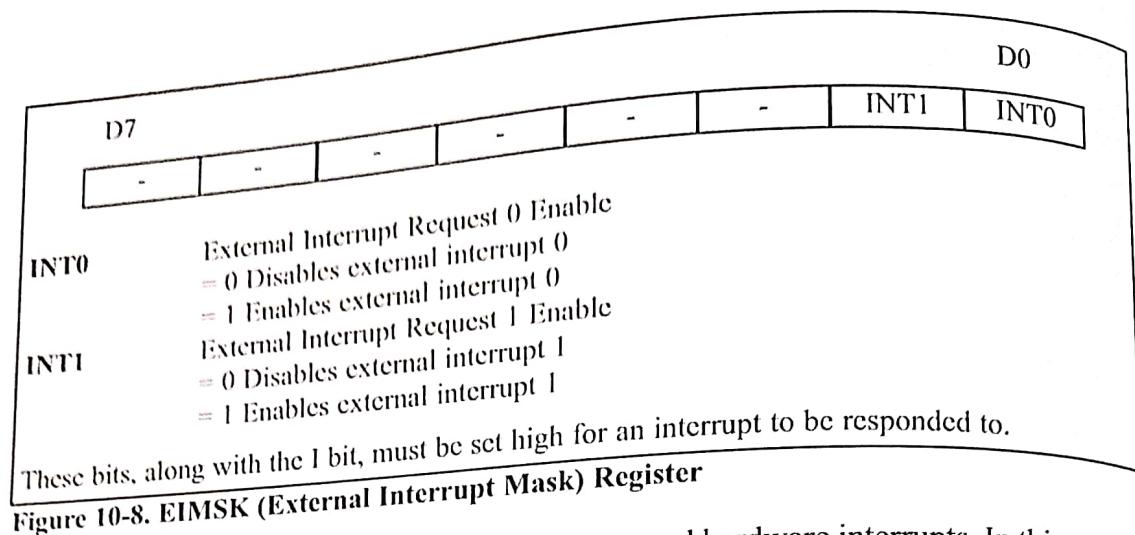


Figure 10-8. EIMSK (External Interrupt Mask)

Study Example 10-5 to gain insight into external hardware interrupts. 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, as discussed next.

Example 10-5

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTB.5 whenever the INT0 pin goes low.

Solution:

```

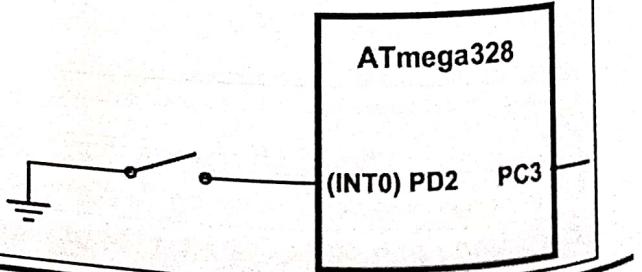
.ORG 0
        JMP    MAIN           ;location for reset

.ORG 0x02
        JMP    EX0_ISR        ;location for external interrupt 0

MAIN: LDI    R20, HIGH(RAMEND)
      OUT   SPH, R20
      LDI    R20, LOW(RAMEND)
      OUT   SPL, R20
      SBI   DDRB, 5          ;initialize stack
      SBI   PORTD, 2         ;PORTB.5 = output
      LDI    R20, 1<<INT0
      OUT   EIMSK, R20       ;pull-up activated
      SEI
HERE: JMP    HERE           ;enable interrupts

EX0_ISR:
      IN    R21, PORTB
      LDI   R22, (1<<5)     ;000100000
      EOR   R21, R22
      OUT   PORTB, R21
      RETI

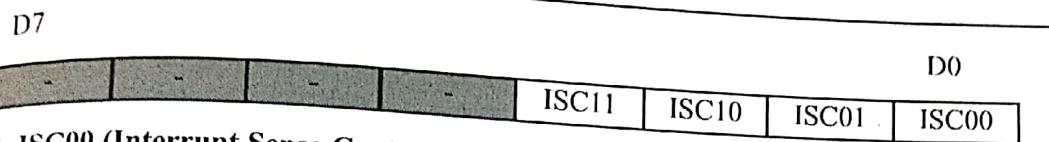
```



Edge-triggered vs. level-triggered interrupts

There are two types of activation for the external hardware interrupts: (1) level triggered, and (2) edge triggered. 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 EICRA register indicate the trigger options of INT0 and INT1, as shown in Figure 10-9. See Examples 10-6 and 10-7.



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 10-9. EICRA (External Interrupt Control Register A) Register

Example 10-6

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

Solution:

- (a) LDI R20, 1<<ISC01 ;R20 = 0x02
STS EICRA, R20
- (b) LDI R20, 1<<ISC10 ;R20 = 0x04
STS EICRA, R20
- (c) LDI R20, (1<<ISC11) | (1<<ISC10) ;R20 = 0x0C
STS EICRA, R20

Example 10-7

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

Solution:

```

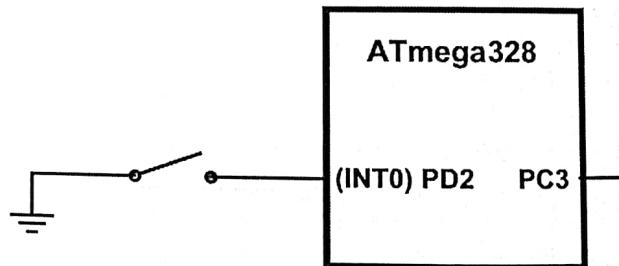
.ORG 0          ;location for reset
    JMP MAIN
.ORG 0x02       ;location for external interrupt 0
    JMP EX0_ISR

MAIN:
    LDI R20, HIGH(RAMEND)
    OUT SPH, R20
    LDI R20, LOW(RAMEND)
    OUT SPL, R20
    LDI R20, 0x2
    STS EICRA, R20
    SBI DDRB, 5
    SBI PORTD, 2
    LDI R20, 1<<INT0
    OUT EIMSK, R20
    SEI

HERE: JMP HERE

EX0_ISR:
    IN  R21, PORTB
    LDI R22, (1<<5) ;00100000 for toggling PB5
    EOR R21, R22
    OUT PORTB, R21
    RETI

```



In Example 10-7, notice that the only difference between it and the program in Example 10-5 is in the following instructions:

```

LDI R20, 0x2           ;make INT0 falling edge triggered
STS EICRA, R20

```

which makes INT0 an edge-triggered interrupt. When the falling edge of the signal is applied to pin INT0, PORTB.5 will toggle. To toggle the LED again, another high-to-low pulse must be applied to INT0. This is the opposite of Example 10-5. In Example 10-5, due to the level-triggered nature of the interrupt, as long as INT0 is kept at a low level, PORTB.5 toggles. But in this example, to turn on PORTB.5 again, the INT0 pulse must be brought back high and then low

Sampling the edge-triggered and level-triggered interrupts

Examine Figure 10-10. 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 EIFR 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 = 0x02  
OUT    EIFR, R20            ;clear the INTF1 flag
```

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.

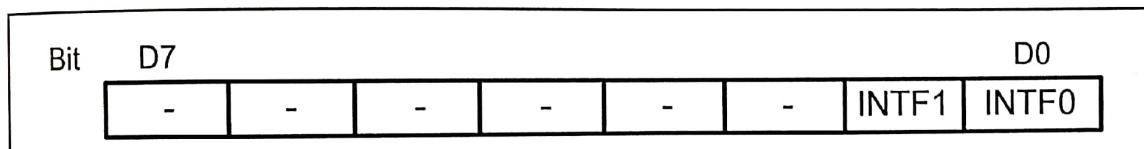


Figure 10-10. EIFR (External Interrupt Flag Register) Register

Review Questions

1. True or false. Upon reset, the external hardware interrupts INT0 and INT1 are edge triggered.
2. For ATmega328, what pins are assigned to INT0 and INT1?
3. Show how to enable the INT1 interrupt.
4. Assume that the external hardware interrupt INT0 is enabled, and is set to the low-edge trigger. Explain how this interrupt works when it is activated.
5. True or false. Upon reset, the INT1 interrupt is edge triggered.
6. Assume that INT0 is falling edge triggered. How do we make sure that a single interrupt is not recognized as multiple interrupts?
7. Using polling and INT0, write a program that upon falling edges toggles PORTC.3. Compare it with Example 10-7; which program is better?

SECTION 10.4: PIN CHANGE INTERRUPT

In ATmega328, for each port there is a change interrupt. If the pin change interrupt of a port is enabled, any changes on values of pins leads rising an interrupt.

rupt. See Figure 10-11. The PCICR register contains the enable bits for the ports. Figure 10-12 shows the simplified circuit for a PCINT pin. See Figures 10-12 and 10-13. For each PCINT pin there is an enable bit. If the enable bit of a pin is set, changes of the pin triggers an interrupt request.

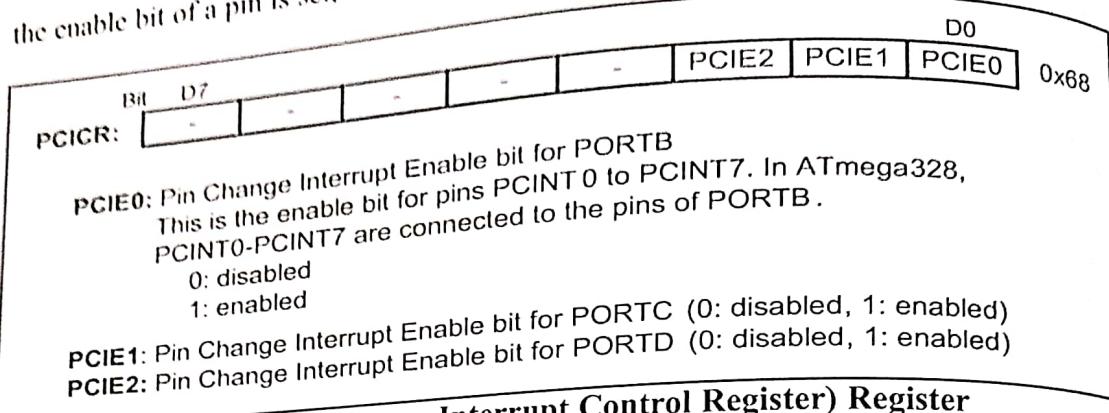


Figure 10-11. PCICR (Pin Change Interrupt Control Register) Register

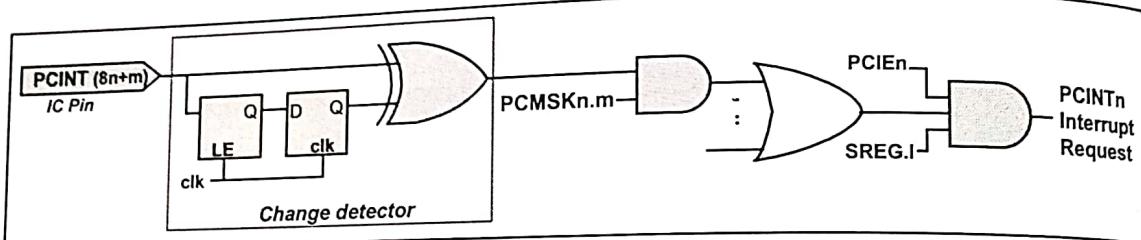


Figure 10-12. Simplified Circuit of a PCINT Pin

Bit	D7									D0
PCMSK0: (for PORTB)		PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	0x6B
PCMSK1: (for PORTC)		PCINT15	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	0x6C
PCMSK2: (for PORTD)		PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	0x6D

PCINTx: It is the enable bit for pin PCINTx. If the bit is set, and the PCIE bit of the port is enabled, any changes of the pin makes an interrupt request.

Figure 10-13. PCMSK (Pin Change Mask) Registers

Using pin change interrupts

In order to use the pin change interrupts the followings must be set to one:

- (1) proper bits of PCMSK_n registers,
- (2) the PCIE_n bits of the related ports,
- (3) the I flag of SREG. See Example 10-8.

Review Questions

1. True or false. Upon reset, the pin change interrupts are enabled.
2. Show how to enable the pin change interrupt for pins PD3 and PD4.

Example 10-8

The PB0, PB2, and PB3 pins are connected to switches. Write a program that makes PORTB.5 high whenever any of the switches changes state.

Solution:

To enable the pin change interrupts of pins PB0, PB2, and PB3 the following bits must be set:

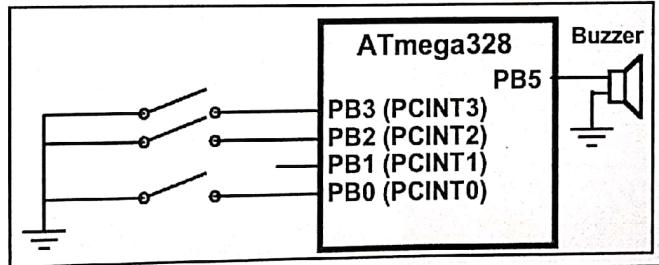
- (1) Bits 0, 2, and 3 of PCMSK0 (According to Figure 10-13, PCMSK0 is for PORTB)
- (2) PCIE0 of PCICR (See Figure 10-11. PCIE0 is for PORTB)
- (3) the I flag

```
.ORG 0          ;location for reset
    JMP MAIN
.ORG 0x06       ;location for pin change interrupt 0
    JMP PCINT0_ISR

MAIN: LDI R20, HIGH (RAMEND)
      OUT SPH, R20
      LDI R20, LOW (RAMEND)
      OUT SPL, R20           ;initialize stack
      SBI DDRB, 5            ;PB5 as output
      CBI PORTB, 5           ;PB5 = low
      LDI R20, 0b000001101
      STS PCMSK0, R20        ;enable PB0, PB2, and PB3 in PCMSK0
      OUT PORTB, R20         ;enable pull-up resistors
      LDI R20, (1<<PCIE0)
      STS PCICR, R20         ;enable PORTB change interrupt
      SEI                   ;enable interrupts

HERE: JMP HERE   ;stay here

PCINT0_ISR:
      SBI PORTB, 5 ;PB5=high
      RETI          ;return
```



SECTION 10.5: INTERRUPT PRIORITY IN THE AVR

The next topic that we must deal with is what happens when two interrupts are activated at the same time. Which of these two interrupts is responded to first?

Interrupt priority

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. See Table 10-1. For example, the address of external interrupt 0 is 2, while the address of external interrupt 1 is 4; 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, 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 PORTC.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 PORTD once every 10 µs. Read the following program. Does it work?

Program 10-4

```
;Program 10-4
.ORG 0x0 ;location for reset
JMP MAIN
.ORG 0x1C ;ISR location for Timer0 compare match A
JMP T0_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
LDI R20,159
OUT OCR0A,R20 ;load Timer0 with 159
LDI R20,(1<<WGM01)
OUT TCCR0A,R20
LDI R20,0x01
OUT TCCR0B,R20 ;start Timer0, CTC mode, no prescaler clk
LDI R20,(1<<OCIE0A)
STS TIMSK0,R20 ;enable Timer0 compare match interrupt
SEI ;set I (enable interrupts globally)
LDI R20,0xFF
OUT DDRC,R20 ;make PORTC output
OUT DDRD,R20 ;make PORTD output
```

Program 10-4 (Cont.)

```
;----- Infinite loop
HERE: OUT  PORTC, R20 ;PORTC = R20
      INC  R20
      JMP  HERE           ;keeping CPU busy waiting for interrupt

;-----ISR for Timer0 (it is executed every 40 µs)
T0_CM_ISR:
      IN   R20, PIND
      INC R20
      OUT PORTD, R20 ;PORTD = R20
      RETI            ;return from interrupt
```

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. See Program 10-5.

Program 10-5

```
;Program 10-5
.ORG 0x0 ;location for reset
      JMP MAIN
.ORG 0x1C ;ISR location for Timer0 compare match A
      JMP T0_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

      LDI R20, 159
      OUT OCR0A, R20 ;load Timer0 with 159
      LDI R20, (1<<WGM01)
      OUT TCCR0A, R20
      LDI R20, 0x01
      OUT TCCR0B, R20 ;start Timer0, CTC mode, no prescaler clk
      LDI R20, (1<<OCIE0A)
      STS TIMSK0, R20 ;enable Timer0 compare match interrupt
      SEI               ;set I (enable interrupts globally)

      LDI R20, 0xFF
      OUT DDRC, R20 ;make PORTC output
      OUT DDRD, R20 ;make PORTD output
```

Program 10-5 (Cont.)

```
;----- Infinite loop
HERE: OUT PORTC,R20 ;PORTC = R20
      INC R20
      JMP HERE ;keeping CPU busy waiting for interrupt
;----- ISR for Timer0 (it is executed every 40 µs)
T0_CM_ISR:
      IN R21,PIND
      INC R21
      OUT PORTD,R21 ;PORTD = R21
      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*). See the following program:

Program 10-6

```
;Program 10-6
;----- location for reset
.ORG 0x00
JMP MAIN
;----- ISR location for Timer0 compare match A
.ORG 0x1C
JMP T0_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

      LDI R20,159
      OUT OCR0A,R20 ;load Timer0 with 159
      LDI R20,(1<<WGM01)
      OUT TCCR0A,R20
      LDI R20,0x01
      OUT TCCR0B,R20 ;start Timer0, CTC mode, no prescaler int clk
      LDI R20,(1<<OCIE0A)
      STS TIMSK0,R20 ;enable Timer0 compare match interrupt
      SEI ;set I (enable interrupts globally)

      LDI R20,0xFF
      OUT DDRC,R20 ;make PORTC output
      OUT DDRD,R20 ;make PORTD output
;----- Infinite loop
HERE: OUT PORTC,R20 ;PORTC = R20
      INC R20
      JMP HERE ;keeping CPU busy waiting for interrupt
;----- ISR for Timer0 (it is executed every 40 µs)
T0_CM_ISR:
```

Program 10-6 (Cont.)

```
PUSH R20 ; save R20 on stack
IN  R20, PIND
INC R20
OUT PORTD, R20 ; PORTD = R20
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. See Figure 10-14.

Sample_ISR:

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

Figure 10-14. 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 only one instruction cycle (e.g., ADD). See the AVR datasheet for the timing.

Review Questions

1. True or false. In ATmega328, if the Timer1 and Timer0 interrupts are activated at the same time, the Timer0 interrupt is served first.
2. What happens if two interrupts are activated at the same time?
3. What happens if an interrupt is activated while the CPU is serving another interrupt?
4. What is context saving?

SECTION 10.5: INTERRUPT PROGRAMMING IN C

So far all the programs in this chapter have been written in Assembly. In this section we show how to program the AVR's interrupts in C language. In C language there is no instruction to manage the interrupts. So, in Atmel Studio 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 Atmel Studio, 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
}
```

Table 10-3: Interrupt Vector Name for the ATmega328 in Atmel Studio

Interrupt	Vector Name in Atmel Studio
External Interrupt request 0	INT0_vect
External Interrupt request 1	INT1_vect
Pin Change Interrupt Request 0	PCINT0_vect
Pin Change Interrupt Request 1	PCINT1_vect
Pin Change Interrupt Request 2	PCINT2_vect
Watchdog Time-out Interrupt	WDT_vect
Time/Counter2 Compare Match A	TIMER2_COMPA_vect
Time/Counter2 Compare Match B	TIMER2_COMPB_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 A	TIMER0_COMPA_vect
Time/Counter0 Compare Match B	TIMER0_COMPB_vect
Time/Counter0 Overflow	TIMER0_OVF_vect
SPI Transfer complete	SPI_STC_vect
USART, Receive complete	USART_RX_vect
USART, Data Register Empty	USART_UDRE_vect
USART, Transmit Complete	USART_TX_vect
ADC Conversion complete	ADC_vect
EEPROM ready	EE_READY_vect
Analog Comparator	ANALOG_COMP_vect
Two-wire Serial Interface	TWI_vect
Store Program Memory Ready	SPM_READY_vect

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 (TIMERO_COMP_vect)
{
}
```

See Example 10-9.

Example 10-9 (C version of Program 10-1)

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

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR (TIMER0_OVF_vect) //ISR for Timer0 overflow
{
    TCNT0 = -32;
    PORTB ^= (1<<5); //toggle PORTB.5
}

int main ()
{
    DDRB |= (1<<5); //DDRB.5 = output

    TCNT0 = -32; //timer value for 2 µs
    TCCR0A = 0x00;
    TCCR0B = 0x01; //Normal mode, int clk, no prescaler

    TIMSK0 = (1<<TOIE0); //enable Timer0 overflow interrupt
    sei(); //enable interrupts

    DDRC = 0x00; //make PORTC input
    PORTC = 0xFF; //enable pull-up resistors
    DDRD = 0xFF; //make PORTD output

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

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. See Examples 10-10 through 10-16.

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

Using Timer0 and Timer1 interrupts, generate square waves on pins PB1 and PB3 respectively, while transferring data from PORTC to PORTD.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ()
{
    DDRB |= (1<<1) | (1<<3); //make DDRB.1 and DDRB.3 output
    DDRC = 0x00; //make PORTC input
    PORTC = 0xFF; //enable pull-up resistors
    DDRD = 0xFF; //make PORTD output

    TCNT0 = -160;
    TCCR0A = 0x00; //Normal mode, int clk, no prescaler
    TCCR0B = 0x01;

    TCNT1H = (-1600)>>8; //the high byte
    TCNT1L = (-1600); //the low byte
    TCCR1A = 0x00;
    TCCR1B = 0x01;

    TIMSK0 = (1<<TOIE0); //enable Timers 0 interrupt
    TIMSK1 = (1<<TOIE1); //enable Timers 1 interrupt
    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 ^= (1<<1); //toggle PORTB.1
}

ISR (TIMER1_OVF_vect) //ISR for Timer0 overflow
{
    TCNT1H = (-1600)>>8; //TCNT1 = -1600 (reload for next round)
    TCNT1L = (-1600);
    PORTB ^= (1<<3); //toggle PORTB.3
```

Note: We can use "TCNT1 = -1600;" in place of the following instructions:
TCNT1H = (-1600)>>8;
TCNT1L = (-1600);

Example 10-11 (C version of Program 10-3)

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

- (a) PORTC counts up everytime Timer1 overflows. It overflows once per second.
- (b) 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.5.
- (c) It toggles PORTB.1 continuously.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ()
{
    DDRC = 0xFF;                      //make PORTC output
    DDRB |= (1<<5);                  //PORTB.5 as an output
    DDRB |= (1<<1);                  //PORTB.1 as an output

    PORTD |= (1<<4);                //activate pull-up

    TCNT0 = -200;                     //load Timer0 with -200
    TCCR0A = 0x00;
    TCCR0B = 0x06;                   //Normal mode, falling edge, no prescaler
    TIMSK0 = (1<<TOIE0);           //enable Timer 0 interrupt.

    TCNT1H = (-62500)>>8;          //the high byte
    TCNT1L = (-62500)&0xFF;          //overflow after 31250 clocks
    TCCR1A = 0x00;                   //Normal mode
    TCCR1B = 0x04;                   //internal clock, prescaler 1:256

    TIMSK1 = (1<<TOIE1);           //enable Timer 1 interrupt
    sei ();                          //enable interrupts

    while (1)                        //wait here
        PORTB ^= (1<<1);           //toggle PB1
}

ISR (TIMER0_OVF_vect)           //ISR for Timer0 overflow
{
    TCNT0 = -200;                  //TCNT0 = -200
    PORTB ^= (1<<5);             //toggle PORTB.5
}

ISR (TIMER1_OVF_vect)           //ISR for Timer1 overflow
{
    TCNT1H = (-62500)>>8;        //the high byte
    TCNT1L = (-62500)&0xFF;        //overflow after 62500 clocks
    PORTC++;                       //increment PORTC
}
```

Example 10-12 (C version of Example 10-3)

Using Timer0, write a program that toggles pin PORTB.5 every 15 μ s, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 16 MHz.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>
int main () {
    DDRB |= 0x20; //make DDRB.5 output

    OCR0A = 239; //CTC mode, internal clk, no prescaler
    TCCR0A = (1<<WGM01);
    TCCR0B = 0x01;
    TIMSK0 = (1<<OCIE0A); //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_COMPA_vect) //ISR for Timer0 compare match
{
    PORTB ^= 0x20; //toggle PORTB.5
}
```

Example 10-13 (C version of 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 = 16 MHz.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>
int main () {
    DDRB |= (1<<5); //make DDRB.5 output

    OCR1A = 15624;
    TCCR1A = 0x00; //CTC mode, internal clk, prescaler=1024
    TCCR1B = 0x0D;
    TIMSK1 = (1<<OCIE1A); //enable Timer1 compare match A int.
    sei (); //enable interrupts

    DDRC = 0x00;
    DDRD = 0xFF; //make PORTC input
    while (1) //make PORTD output
        PORTD = PINC; //wait here

    ISR (TIMER1_COMPA_vect) //ISR for Timer1 compare match A
    {
        PORTB ^= (1<<5); //toggle PORTB.5
    }
}
```

Example 10-14 (C version of Example 10-5)

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTB.5, whenever INT0 pin goes low. Use the external interrupt in level-triggered mode.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ()
{
    DDRB = 1<<5;           //PB5 as an output
    PORTD = 1<<2;          //pull-up activated
    EIMSK = (1<<INT0);    //enable external interrupt 0
    sei ();                 //enable interrupts

    while (1);              //wait here
}

ISR (INT0_vect)           //ISR for external interrupt 0
{
    PORTB ^= (1<<5);     //toggle PORTB.5
}
```

Example 10-15 (C version of Example Example 10-7)

Rewrite Example 10-14 so that whenever INT0 goes low, it toggles PORTB.5 only once.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ()
{
    DDRB = 1<<5;           //PB5 as an output
    PORTD = 1<<2;          //pull-up activated
    EICRA = 0x2;             //make INT0 falling edge triggered

    EIMSK = (1<<INT0);    //enable external interrupt 0
    sei ();                 //enable interrupts

    while (1);              //wait here
}

ISR (INT0_vect)           //ISR for external interrupt 0
{
    PORTB ^= (1<<5);     //toggle PORTB.5
}
```

Example 10-16 (C version of Example Example 10-8)

The PB0, PB2, and PB3 pins are connected to switches. Write a program that makes PORTB.5 high whenever any of the switches changes state.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ()
{
    DDRB = 1<<5;                                //PB5 as an output
    PCMSK0 = (1<<0) | (1<<2) | (1<<3);        //pull-up activated
    PORTB = (1<<0) | (1<<2) | (1<<3);          //enable PORTB change interrupt
    PCICR = (1<<PCIE0);                         //enable interrupts
    sei ();                                         //wait here
    while (1);
}

ISR (PCINT0_vect)                            //ISR for Pin Change Interrupt 0
{
    PORTB |= (1<<5);                          //PORTB.5 = HIGH
}
```

SUMMARY

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service. Every interrupt has a program associated with it called the ISR, or interrupt service routine. The AVR has many sources of interrupts, depending on the family member. Some of the most widely used interrupts are for the timers, external hardware interrupts, and serial communication. When an interrupt is activated, the related interrupt flag bit is raised.

The AVR can be programmed to enable (unmask) or disable (mask) an interrupt, which is done with the help of the I (global interrupt enable) and IE (interrupt enable) bits. This chapter also showed how to program AVR interrupts in both Assembly and C languages.

PROBLEMS

SECTION 10.1: AVR INTERRUPTS

1. Which technique, interrupt or polling, avoids tying down the microcontroller?
2. List some of the interrupt sources in the AVR.
3. In the ATmega328 what memory area is assigned to the interrupt vector table?
4. True or false. The AVR programmer cannot change the memory address location assigned to the interrupt vector table.
5. What memory address is assigned to the Timer0 overflow interrupt in the interrupt vector table?

6. What memory address is assigned to the Timer1 overflow interrupt in the interrupt vector table?
7. Do we have a memory address assigned to the Timer0 compare match A interrupt in the interrupt vector table?
8. Do we have a memory address assigned to the external INT0 interrupt in the interrupt vector table?
9. To which register does the I bit belong?
10. Why do we put a JMP instruction at address 0?
11. What is the state of the I bit upon power-on reset, and what does it mean?
12. Show the instruction to enable the Timer0 compare match interrupt.
13. Show the instruction to enable the Timer1 overflow interrupt.
14. The TOIE0 bit belongs to register _____.
15. True or false. The TIMSK register is not a bit-addressable register.
16. With a single instruction, show how to disable all the interrupts.
17. Show how to disable the INT0 interrupt.
18. True or false. Upon reset, all interrupts are enabled by the AVR.
19. In the AVR, how many bytes of program memory are assigned to the reset?

SECTION 10.2: PROGRAMMING TIMER INTERRUPTS

20. True or false. For each of Timer0 and Timer1, there is a unique address in the interrupt vector table.
21. What address in the interrupt vector table is assigned to Timer2 overflow?
22. Show how to enable the Timer2 overflow interrupt.
23. Which bit of TIMSK0 belongs to the Timer0 overflow interrupt? Show how it is enabled.
24. Assume that Timer0 is programmed in Normal mode, TCNT0 = \$E0, and the TOIE0 bit is enabled. Explain how the interrupt for the timer works.
25. True or false. The last three instructions of the ISR for Timer0 are:

```

LDI    R20, 1<<TOV0
OUT    TIFR0, R20 ; clear TOV0 flag
RETI

```

26. Assume that Timer1 is programmed for CTC mode, TCNT1H = \$01, TCNT1L = \$00, OCR1AH = \$01, OCR1AL = \$F5, and the OCIE1A bit is enabled. Explain how the interrupt is activated.
27. Assume that Timer1 is programmed for Normal mode, TCNT1H = \$FF, TCNT1L = \$E8, and the TOIE1 bit is enabled. Explain how the interrupt is activated.
28. Write a program using the Timer1 interrupt to create a square wave of 1 Hz on pin PB7 while sending data from PORTC to PORTD. Assume XTAL = 8 MHz.
29. Write a program using the Timer0 interrupt to create a square wave of 3 kHz on pin PB7 while sending data from PORTC to PORTD. Assume XTAL = 1 MHz.

SECTION 10.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

30. True or false. An address location is assigned to each of the external hardware interrupts INT0, INT1, and INT2.
31. What address in the interrupt vector table is assigned to INT0 and INT1? How

- about the pins?
32. To which register does the INT0 bit belong? Show how it is enabled.
33. To which register does the INT1 bit belong? Show how it is enabled.
34. Show how to enable all three external hardware interrupts.
35. Assume that the INT0 bit for external hardware interrupt is enabled and is negative edge-triggered. When is the interrupt activated? How does this interrupt work when it is activated.
36. True or false. Upon reset, all the external hardware interrupts are negative edge triggered.
37. The INTF0 bit belongs to the _____ register.
38. The INTF1 bit belongs to the _____ register.
39. Explain the role of INTF0 and INT0 in the execution of external interrupt 0.
40. Explain the role of I in the execution of external interrupts.
41. True or false. Upon power-on reset, INT0 and INT1 are positive edge triggered.
42. Explain the difference between low-level and falling edge-triggered interrupts.
43. Show how to make the external INT0 negative edge triggered.
44. True or false. INT0 and INT1 must be configured as an input pin for a hardware interrupt to come in.
45. Assume that the INT0 pin is connected to a switch. Write a program in which, whenever it goes low, the content of PORTC increases by one.
46. Assume that the INT0 and INT1 are connected to two switches named S1 and S2. Write a program in which, whenever S1 goes low, the content of PORTC increases by one; and when S2 goes low, the content of PORTC decreases by one. When the value of PORTC is bigger than 100, PD7 is high; otherwise, it is low.

SECTION 10.4: PIN CHANGE INTERRUPT

47. True or false. ATmega328 does not have pin change interrupt.
48. True or false. In ATmega328, pin change interrupt is available only on PORTB.
49. Using pin change interrupt, write a program that sends the value of PORTC to PORTD whenever the value of PORTC changes.
50. Write a program that toggles PB2 whenever the value of PORTD changes.

SECTION 10.5: INTERRUPT PRIORITY IN THE AVR

51. Explain what happens if both INT1F and INT2F are activated at the same time.
52. Assume that the Timer1 and Timer0 overflow interrupts are both enabled. Explain what happens if both TOV1 and TOV0 are activated at the same time.
53. Explain what happens if an interrupt is activated while the AVR is serving an interrupt.
54. True or false. In the AVR, an interrupt inside an interrupt is not allowed.

ANSWERS TO REVIEW QUESTIONS

SECTION 10.1: AVR INTERRUPTS

1. Interrupt
2. Timer0 overflow, Timer0 compare match A, and Timer0 compare match B.
3. Address locations 0x00 to 0x32. Yes.
4. I = 0 means that all interrupts are masked, and as a result no interrupts will be responded to by the AVR.
5. Assuming I = 1, we need:
LDI R16, (1<<OCIE1A)
STS TIMSK1, R16
6. \$1A for Timer1 overflow interrupt and 0x02 for INT0.

SECTION 10.2: PROGRAMMING TIMER INTERRUPTS

1. False. For each of the interrupts there is a separate address.
2. 0x20
3. TIMSK1
LDI R16, (1<<TOIE1)
STS TIMSK1, R16
4. After Timer0 is started, the timer will count up from \$F1 to \$FF on its own while the AVR is executing other tasks. Upon rolling over from \$FF to 00, the TOV0 flag is raised, which will interrupt the AVR in whatever it is doing and force it to jump to memory location \$0020 to execute the ISR belonging to this interrupt.
5. False. There is no need to clear the TOV0 flag since the AVR clears the TOV0 flag internally upon jumping to the interrupt vector table.
6. The timer counts from 0 to 21. Then TCNT0 is loaded with 0 and the OCF0A flag is set. If Timer0 compare match A interrupt is enabled, the ISR of the compare match A interrupt is executed on each compare match.
7. $1/8 \text{ MHz} = 125 \text{ ns} \rightarrow 125 \text{ ns} \times (21 + 1) = 2.75 \mu\text{s}$

SECTION 10.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

1. False. Only INT2 is in edge-triggered mode.
2. Bits PD2 (PORTD.2) and PD3 (PORTD.3) are assigned to INT0 and INT1, respectively.
3. LDI R20, (1<<INT1)
OUT EIMSK, R20
4. Upon application of a high-to-low pulse to pin PD2, the INTF0 flag will be set; as a result, the AVR is interrupted in whatever it is doing, clears the INTF0 flag, and jumps to ROM location 0x02 to execute the ISR.
5. False
6. When the CPU jumps to the interrupt vector to execute the ISR, it clears the flag that has caused the interrupt (the INTF0 flag in this case). The INTF0 flag will be set only if a new high-to-low pulse is applied to the pin.

```
7. LDI R16, 0x2          ;make INT0 falling edge triggered
    STS EICRA, R16      ;skip next instruct. if the INTF0 bit is set
L1: SBIS EIFR, INTF0   ;go to L1
    RJMP L1              ;R21 = PORTC
    IN R21, PORTC
    LDI R22, 0x08         ;R21 = R21 xor 0x08 (toggle bit 3)
    EOR R21, R22
    OUT PORTC, R21       ;PORTC = R21
    LDI R20, 1<<INTF0   ;clear INTF0 flag
    OUT EIFR, R20
    RJMP L1
```

SECTION 10.4: PIN CHANGE INTERRUPT

```
1. False  
2. LDI R20, (1<<3) | (1<<4) ; enable PD3 and PD4 in PCMSK2  
STS PCMSK2, R20  
LDI R20, (1<<PCIE2) ; enable PORTD change interrupt  
STS PCICR, R20 ; enable interrupts  
SEI
```

SECTION 10.5: INTERRUPT PRIORITY IN THE AVR

1. False. As shown in Table 10-1, the address of the Timer0 overflow interrupt is \$20, while the address of Timer1 overflow is \$1A. Thus, the Timer1 overflow has a higher priority.
2. The interrupt whose vector is first in the interrupt vector is served first.
3. The flag of the interrupt will be set, but since I is 0, the new interrupt will not be served. The last instruction of the old interrupt is RETI, which causes the I flag to be set and the new interrupt to be served.
4. Context saving is the saving of the CPU contents before switching to a new task.