

---

# CHAPTER 9

---

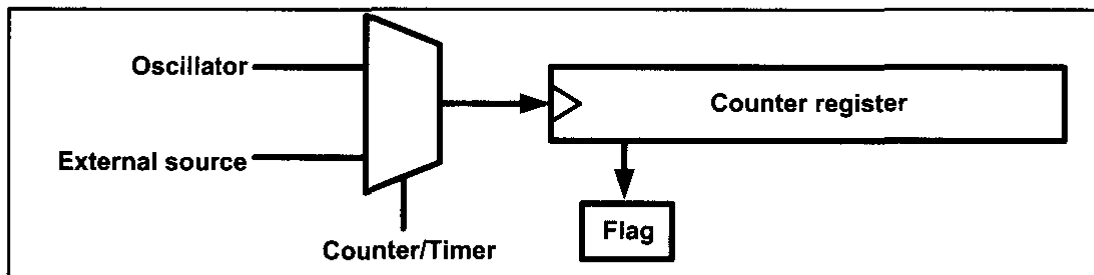
## AVR TIMER PROGRAMMING IN ASSEMBLY AND C

### OBJECTIVES

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

- >> List the timers of the ATmega32 and their associated registers
- >> Describe the Normal and CTC modes of the AVR timers
- >> Program the AVR timers in Assembly and C to generate time delays
- >> Program the AVR counters in Assembly and C as event counters

Many applications need to count an event or generate time delays. So, there are counter registers in microcontrollers for this purpose. See Figure 9-1. When we want to count an event, we connect the external event source to the clock pin of the counter register. Then, when an event occurs externally, the content of the counter is incremented; in this way, the content of the counter represents how many times an event has occurred. When we want to generate time delays, we connect the oscillator to the clock pin of the counter. So, when the oscillator ticks, the content of the counter is incremented. As a result, the content of the counter register represents how many ticks have occurred from the time we have cleared the counter. Since the speed of the oscillator in a microcontroller is known, we can calculate the tick period, and from the content of the counter register we will know how much time has elapsed.



**Figure 9-1. A General View of Counters and Timers in Microcontrollers**

So, one way to generate a time delay is to clear the counter at the start time and wait until the counter reaches a certain number. For example, consider a microcontroller with an oscillator with frequency of 1 MHz; in the microcontroller, the content of the counter register increments once per microsecond. So, if we want a time delay of 100 microseconds, we should clear the counter and wait until it becomes equal to 100.

In the microcontrollers, there is a flag for each of the counters. The flag is set when the counter overflows, and it is cleared by software. The second method to generate a time delay is to load the counter register and wait until the counter overflows and the flag is set. For example, in a microcontroller with a frequency of 1 MHz, with an 8-bit counter register, if we want a time delay of 3 microseconds, we can load the counter register with \$FD and wait until the flag is set after 3 ticks. After the first tick, the content of the register increments to \$FE; after the second tick, it becomes \$FF; and after the third tick, it overflows (the content of the register becomes \$00) and the flag is set.

The AVR has one to six timers depending on the family member. They are referred to as Timers 0, 1, 2, 3, 4, and 5. They can be used as timers to generate a time delay or as counters to count events happening outside the microcontroller.

In the AVR some of the timers/counters are 8-bit and some are 16-bit. In ATmega32, there are three timers: Timer0, Timer1, and Timer2. Timer0 and Timer2 are 8-bit, while Timer1 is 16-bit. In this chapter we cover Timer0 and Timer2 as 8-bit timers, and Timer1 as a 16-bit timer.

If you learn to use the timers of ATmega32, you can easily use the timers of other AVR. You can use the 8-bit timers like the Timer0 of ATmega32 and the 16-bit timers like the Timer1 of ATmega32.

## SECTION 9.1: PROGRAMMING TIMERS 0, 1, AND 2

Every timer needs a clock pulse to tick. The clock source can be internal or external. If we use the internal clock source, then the frequency of the crystal oscillator is fed into the timer. Therefore, it is used for time delay generation and consequently is called a *timer*. By choosing the external clock option, we feed pulses through one of the AVR's pins. This is called a *counter*. In this section we discuss the AVR timer, and in the next section we program the timer as a counter.

### Basic registers of timers

Examine Figure 9-2. In AVR, for each of the timers, there is a TCNTn (timer/counter) register. That means in ATmega32 we have TCNT0, TCNT1, and TCNT2. The TCNTn register is a counter. Upon reset, the TCNTn contains zero. It counts up with each pulse. The contents of the timers/counters can be accessed using the TCNTn. You can load a value into the TCNTn register or read its value.

Each timer has a TOVn (Timer Overflow) flag, as well. When a timer overflows, its TOVn flag will be set.

Each timer also has the TCCRn (timer/counter control register) register for setting modes of operation. For example, you can specify Timer0 to work as a timer or a counter by loading proper values into the TCCR0.

Each timer also has an OCRn (Output Compare Register) register. The content of the OCRn is compared with the content of the TCNTn. When they are equal the OCFn (Output Compare Flag) flag will be set.

The timer registers are located in the I/O register memory. Therefore, you can read or write from timer registers using IN and OUT instructions, like the other I/O registers. For example, the following instructions load TCNT0 with 25:

```
LDI R20,25      ;R20 = 25
OUT TCNT0,R20   ;TCNT0 = R20
```

or "IN R19,TCNT2" copies TCNT2 to R19.

The internal structure of the ATmega32 timers is shown in Figure 9-3. Next, we discuss each timer separately in more detail.

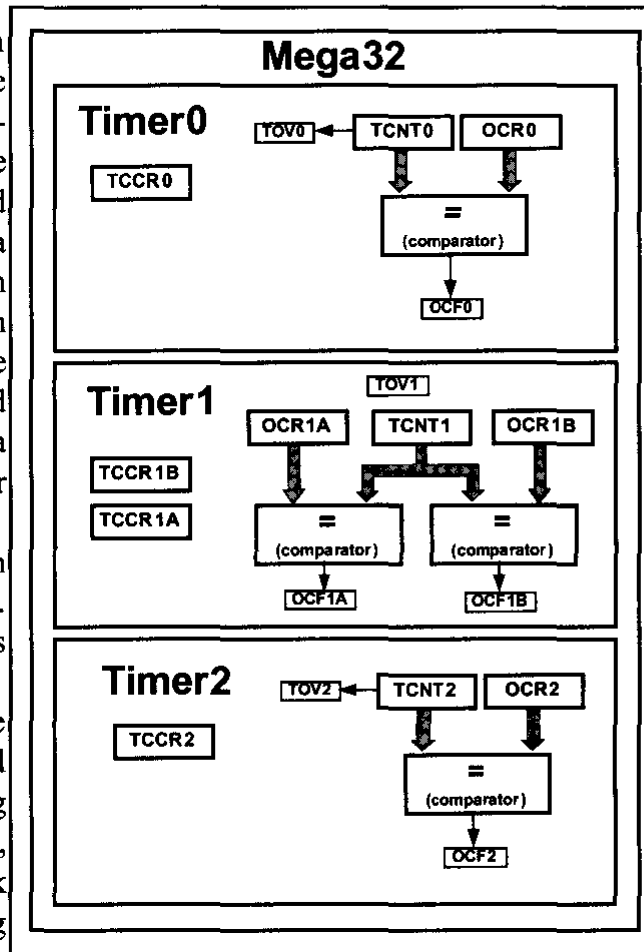


Figure 9-2. Timers in ATmega32

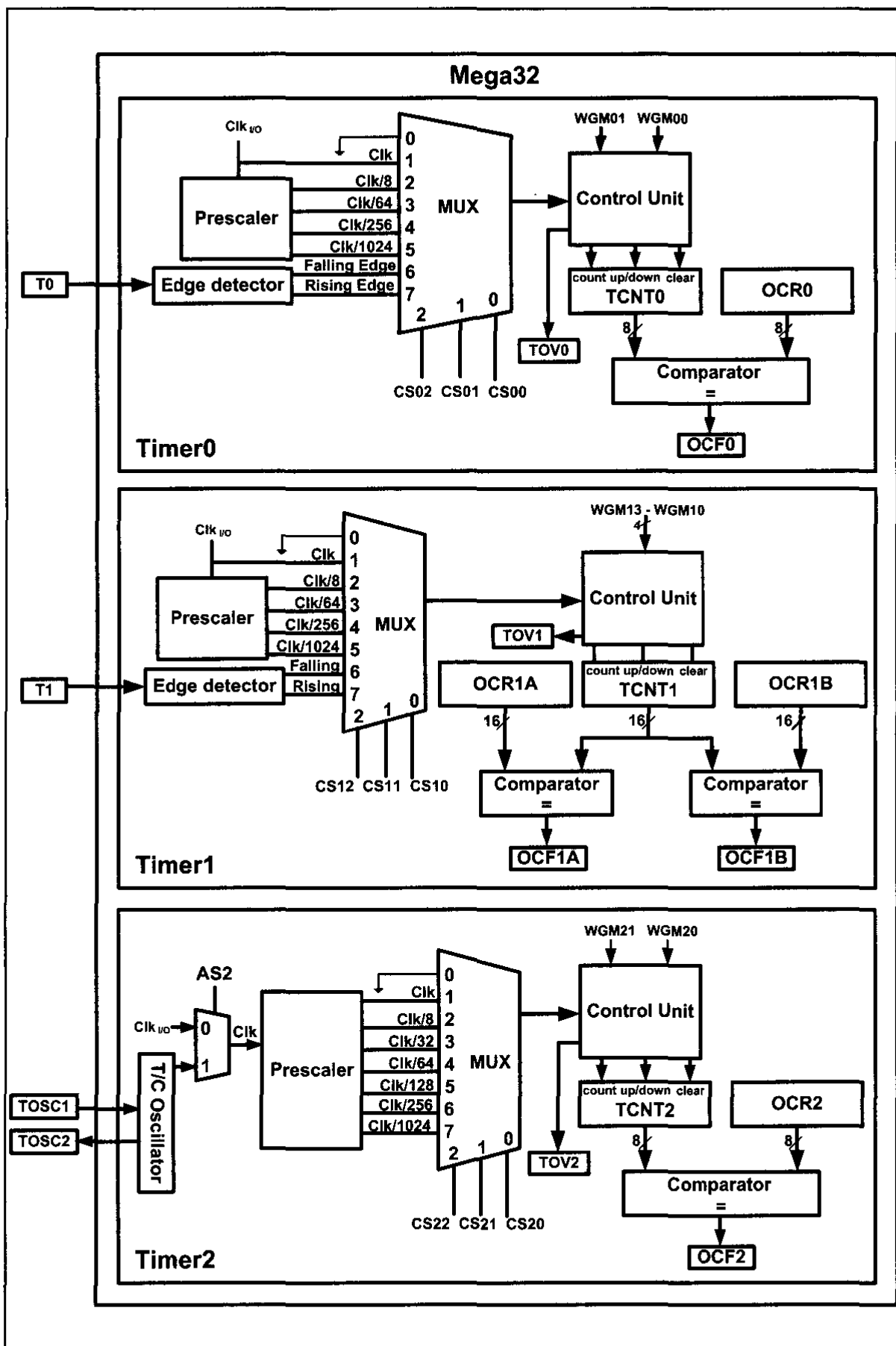


Figure 9-3. Timers in ATmega32

## Timer0 programming

Timer0 is 8-bit in ATmega32; thus, TCNT0 is 8-bit as shown in Figure 9-4.

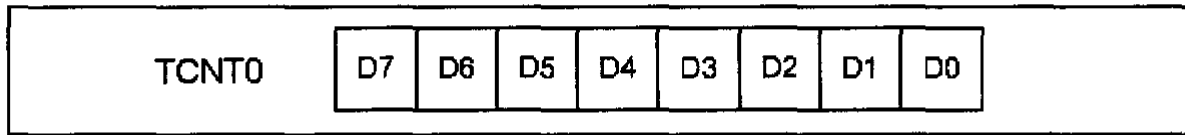


Figure 9-4. Timer/Counter 0 Register

### TCCR0 (Timer/Counter Control Register) register

TCCR0 is an 8-bit register used for control of Timer0. The bits for TCCR0 are shown in Figure 9-5.

#### CS02:CS00 (Timer0 clock source)

These bits in the TCCR0 register are used to choose the clock source. If CS02:CS00 = 000, then the counter is stopped. If CS02–CS00 have values between 001 and 101, the oscillator is used as clock source and the timer/counter acts as a timer. In this case, the timers are often used for time delay generation. See Figure 9-3 and then see Examples 9-1 and 9-2.

Bit	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Read/Write	W	RW	RW	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0
<b>FOC0</b>	D7	Force compare match: This is a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.						
<b>WGM00, WGM01</b>	D6	D3	Timer0 mode selector bits					
	0	0	Normal					
	0	1	CTC (Clear Timer on Compare Match)					
	1	0	PWM, phase correct					
	1	1	Fast PWM					
<b>COM01:00</b>	D5	D4	Compare Output Mode: These bits control the waveform generator (see Chapter 15).					
<b>CS02:00</b>	D2	D1	D0	Timer0 clock selector				
	0	0	0	No clock source (Timer/Counter stopped)				
	0	0	1	clk (No Prescaling)				
	0	1	0	clk / 8				
	0	1	1	clk / 64				
	1	0	0	clk / 256				
	1	0	1	clk / 1024				
	1	1	0	External clock source on T0 pin. Clock on falling edge.				
	1	1	1	External clock source on T0 pin. Clock on rising edge.				

Figure 9-5. TCCR0 (Timer/Counter Control Register) Register

### Example 9-1

Find the value for TCCR0 if we want to program Timer0 in Normal mode, no prescaler. Use AVR's crystal oscillator for the clock source.

**Solution:**

TCCR0 =	0	0	0	0	0	0	0	1
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

### Example 9-2

Find the timer's clock frequency and its period for various AVR-based systems, with the following crystal frequencies. Assume that no prescaler is used.

- (a) 10 MHz      (b) 8 MHz      (c) 1 MHz

**Solution:**

(a)  $F = 10 \text{ MHz}$  and  $T = 1/10 \text{ MHz} = 0.1 \mu\text{s}$

(b)  $F = 8 \text{ MHz}$  and  $T = 1/8 \text{ MHz} = 0.125 \mu\text{s}$

(c)  $F = 1 \text{ MHz}$  and  $T = 1/1 \text{ MHz} = 1 \mu\text{s}$

If CS02–CS00 are 110 or 111, the external clock source is used and it acts as a counter. We will discuss Counter in the next section.

### WGM01:00

Timer0 can work in four different modes: Normal, phase correct PWM, CTC, and Fast PWM. The WGM01 and WGM00 bits are used to choose one of them. We will discuss the PWM options in Chapter 16.

## TIFR (Timer/counter Interrupt Flag Register) register

The TIFR register contains the flags of different timers, as shown in Figure 9-6. Next, we discuss the TOV0 flag, which is related to Timer0.

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
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
TOV0	D0	Timer0 overflow flag bit 0 = Timer0 did not overflow. 1 = Timer0 has overflowed (going from \$FF to \$00).						
OCF0	D1	Timer0 output compare flag bit 0 = compare match did not occur. 1 = compare match occurred.						
TOV1	D2	Timer1 overflow flag bit						
OCF1B	D3	Timer1 output compare B match flag						
OCF1A	D4	Timer1 output compare A match flag						
ICF1	D5	Input Capture flag						
TOV2	D6	Timer2 overflow flag						
OCF2	D7	Timer2 output compare match flag						

Figure 9-6. TIFR (Timer/Counter Interrupt Flag Register)

### TOV0 (Timer0 Overflow)

The flag is set when the counter overflows, going from \$FF to \$00. As we will see soon, when the timer rolls over from \$FF to 00, the TOV0 flag is set to 1 and it remains set until the software clears it. See Figure 9-6. The strange thing about this flag is that in order to clear it we need to write 1 to it. Indeed this rule applies to all flags of the AVR chip. In AVR, when we want to clear a given flag of a register we write 1 to it and 0 to the other bits. For example, the following program clears TOV0:

```
LDI    R20,0x01
OUT    TIFR,R20    ;TIFR = 0b00000001
```

### Normal mode

In this mode, the content of the timer/counter increments with each clock. It counts up until it reaches its max of 0xFF. When it rolls over from 0xFF to 0x00, it sets high a flag bit called TOV0 (Timer Overflow). This timer flag can be monitored. See Figure 9-7.

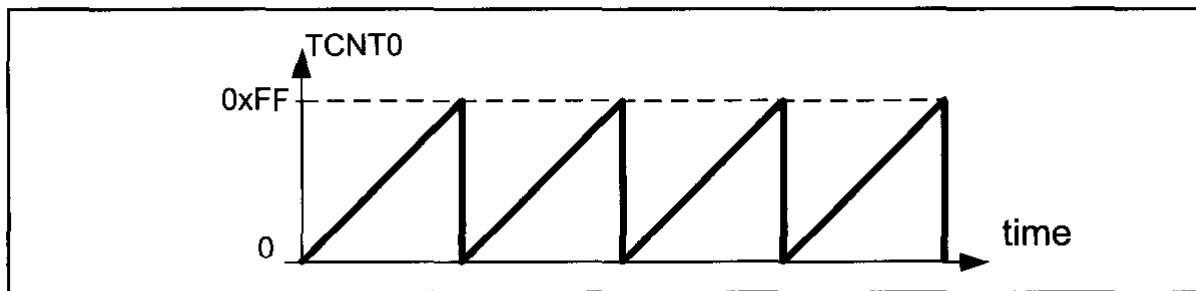


Figure 9-7. Timer/Counter 0 Normal Mode

### Steps to program Timer0 in Normal mode

To generate a time delay using Timer0 in Normal mode, the following steps are taken:

1. Load the TCNT0 register with the initial count value.
2. Load the value into the TCCR0 register, indicating which mode (8-bit or 16-bit) is to be used and the prescaler option. When you select the clock source, the timer/counter starts to count, and each tick causes the content of the timer/counter to increment by 1.
3. Keep monitoring the timer overflow flag (TOV0) to see if it is raised. Get out of the loop when TOV0 becomes high.
4. Stop the timer by disconnecting the clock source, using the following instructions:

```
LDI    R20,0x00
OUT    TCCR0,R20    ;timer stopped, mode=Normal
```

5. Clear the TOV0 flag for the next round.
6. Go back to Step 1 to load TCNT0 again.

To clarify the above steps, see Example 9-3.

### Example 9-3

In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PORTB.5 bit. Timer0 is used to generate the time delay. Analyze the program.

```
.INCLUDE "M32DEF.INC"
.MACRO      INITSTACK      ;set up stack
    LDI     R20,HIGH(RAMEND)
    OUT     SPH,R20
    LDI     R20,LOW(RAMEND)
    OUT     SPL,R20
.ENDMACRO

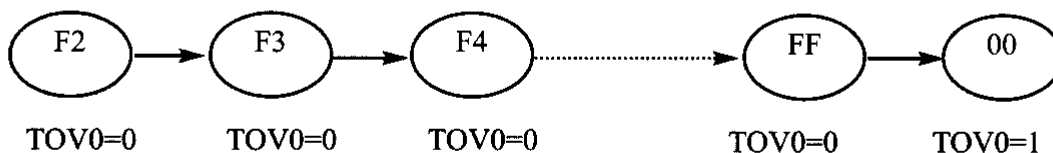
INITSTACK
LDI     R16,1<<5      ;R16 = 0x20 (0010 0000 for PB5)
SBI     DDRB,5        ;PB5 as an output
LDI     R17,0
OUT     PORTB,R17     ;clear PORTB
BEGIN:RCALL DELAY      ;call timer delay
EOR     R17,R16        ;toggle D5 of R17 by Ex-Oring with 1
OUT     PORTB,R17     ;toggle PB5
RJMP    BEGIN

;-----Time0 delay
DELAY:LDI     R20,0xF2    ;R20 = 0xF2
OUT     TCNT0,R20        ;load timer0
LDI     R20,0x01
OUT     TCCR0,R20        ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN      R20,TIFR    ;read TIFR
SBRs     R20,TOV0        ;if TOV0 is set skip next instruction
RJMP     AGAIN
LDI     R20,0x0
OUT     TCCR0,R20        ;stop Timer0
LDI     R20,(1<<TOV0)
OUT     TIFR,R20        ;clear TOV0 flag by writing a 1 to TIFR
RET
```

### Solution:

In the above program notice the following steps:

1. 0xF2 is loaded into TCNT0.
2. TCCR0 is loaded and Timer0 is started.
3. Timer0 counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of F3, F4, F5, F6, F7, F8, F9, FA, FB, and so on until it reaches 0xFF. One more clock rolls it to 0, raising the Timer0 flag (TOV0 = 1). At that point, the "SBRs R20,TOV0" instruction bypasses the "RJMP AGAIN" instruction.
4. Timer0 is stopped.
5. The TOV0 flag is cleared.





To calculate the exact time delay and the square wave frequency generated on pin PB5, we need to know the XTAL frequency. See Examples 9-4 and 9-5.

#### Example 9-4

In Example 9-3, calculate the amount of time delay generated by the timer. Assume that XTAL = 8 MHz.

#### Solution:

We have 8 MHz as the timer frequency. As a result, each clock has a period of  $T = 1 / 8 \text{ MHz} = 0.125 \mu\text{s}$ . In other words, Timer0 counts up each  $0.125 \mu\text{s}$  resulting in delay = number of counts  $\times 0.125 \mu\text{s}$ .

The number of counts for the rollover is  $0xFF - 0xF2 = 0x0D$  (13 decimal). However, we add one to 13 because of the extra clock needed when it rolls over from FF to 0 and raises the TOV0 flag. This gives  $14 \times 0.125 \mu\text{s} = 1.75 \mu\text{s}$  for half the pulse.

#### Example 9-5

In Example 9-3, calculate the frequency of the square wave generated on pin PORTB.5. Assume that XTAL = 8 MHz.

#### Solution:

To get a more accurate timing, we need to add clock cycles due to the instructions.

	<u>Cycles</u>
LDI R16, 0x20	
SBI DDRB, 5	
LDI R17, 0	
OUT PORTB, R17	
BEGIN: RCALL DELAY	3
EOR R17, R16	1
OUT PORTB, R17	1
RJMP BEGIN	2
DELAY: LDI R20, 0xF2	1
OUT TCNT0, R20	1
LDI R20, 0x01	1
OUT TCCR0, R20	1
AGAIN: IN R20, TIFR	1
SBRS R20, 0	1 / 2
RJMP AGAIN	2
LDI R20, 0x0	1
OUT TCCR0, R20	1
LDI R20, 0x01	1
OUT TIFR, R20	1
RET	4
	24

$T = 2 \times (14 + 24) \times 0.125 \mu\text{s} = 9.5 \mu\text{s}$  and  $F = 1 / T = 105.263 \text{ kHz}$ .

(a) in hex	(b) in decimal
(FF - XX + 1) × 0.125 μs where XX is the TCNT0, initial value. Notice that XX value is in hex.	Convert XX value of the TCNT0 register to decimal to get a NNN decimal number, then (256 - NNN) × 0.125 μs

**Figure 9-8. Timer Delay Calculation for XTAL = 8 MHz with No Prescaler**

We can develop a formula for delay calculations using the Normal mode of the timer for a crystal frequency of XTAL = 8 MHz. This is given in Figure 9-8. The scientific calculator in the Accessories menu directory of Microsoft Windows can help you find the TCNT0 value. This calculator supports decimal, hex, and binary calculations. See Example 9-6.

### Example 9-6

Find the delay generated by Timer0 in the following code, using both of the methods of Figure 9-8. Do not include the overhead due to instructions. (XTAL = 8 MHz)

```
.INCLUDE "M32DEF.INC"
    INITSTACK          ;add its definition from Example 9-3
    LDI    R16,0x20
    SBI    DDRB,5       ;PB5 as an output
    LDI    R17,0
    OUT    PORTB,R17
BEGIN:RCALL DELAY
    EOR    R17,R16      ;toggle D5 of R17
    OUT    PORTB,R17    ;toggle PB5
    RJMP   BEGIN
DELAY:LDI    R20,0x3E
    OUT    TCNT0,R20    ;load timer0
    LDI    R20,0x01
    OUT    TCCR0,R20     ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN     R20,TIFR    ;read TIFR
    SBRS   R20,TOV0      ;if TOV0 is set skip next instruction
    RJMP   AGAIN
    LDI    R20,0x00
    OUT    TCCR0,R20     ;stop Timer0
    LDI    R20,(1<<TOV0) ;R20 = 0x01
    OUT    TIFR,R20      ;clear TOV0 flag
    RET
```

### Solution:

(a) (FF - 3E + 1) = 0xC2 = 194 in decimal and 194 × 0.125 μs = 24.25 μs.

(b) Because TCNT0 = 0x3E = 62 (in decimal) we have 256 - 62 = 194. This means that the timer counts from 0x3E to 0xFF. This plus rolling over to 0 goes through a total of 194 clock cycles, where each clock is 0.125 μs in duration. Therefore, we have 194 × 0.125 μs = 24.25 μs as the width of the pulse.

## Finding values to be loaded into the timer

Assuming that we know the amount of timer delay we need, the question is how to find the values needed for the TCNT0 register. To calculate the values to be loaded into the TCNT0 registers, we can use the following steps:

1. Calculate the period of the timer clock using the following formula:

$$T_{\text{clock}} = 1/F_{\text{Timer}}$$

where  $F_{\text{Timer}}$  is the frequency of the clock used for the timer. For example, in no prescaler mode,  $F_{\text{Timer}} = F_{\text{oscillator}}$ .  $T_{\text{clock}}$  gives the period at which the timer increments.

2. Divide the desired time delay by  $T_{\text{clock}}$ . This says how many clocks we need.
3. Perform  $256 - n$ , where  $n$  is the decimal value we got in Step 2.
4. Convert the result of Step 3 to hex, where  $xx$  is the initial hex value to be loaded into the timer's register.
5. Set  $\text{TCNT0} = xx$ .

Look at Examples 9-7 and 9-8, where we use a crystal frequency of 8 MHz for the AVR system.

### Example 9-7

Assuming that  $\text{XTAL} = 8 \text{ MHz}$ , write a program to generate a square wave with a period of  $12.5 \mu\text{s}$  on pin  $\text{PORTB.3}$ .

#### Solution:

For a square wave with  $T = 12.5 \mu\text{s}$  we must have a time delay of  $6.25 \mu\text{s}$ . Because  $\text{XTAL} = 8 \text{ MHz}$ , the counter counts up every  $0.125 \mu\text{s}$ . This means that we need  $6.25 \mu\text{s} / 0.125 \mu\text{s} = 50$  clocks.  $256 - 50 = 206 = 0x\text{CE}$ . Therefore, we have  $\text{TCNT0} = 0x\text{CE}$ .

```
.INCLUDE "M32DEF.INC"
    INITSTACK                ;add its definition from Example 9-3
    LDI    R16,0x08
    SBI    DDRB,3            ;PB3 as an output
    LDI    R17,0
    OUT    PORTB,R17
BEGIN:RCALL DELAY
    EOR    R17,R16          ;toggle D3 of R17
    OUT    PORTB,R17        ;toggle PB3
    RJMP   BEGIN
;----- Timer0 Delay
DELAY:LDI    R20,0xCE
    OUT    TCNT0,R20        ;load Timer0
    LDI    R20,0x01
    OUT    TCCR0,R20        ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN     R20,TIFR        ;read TIFR
    SBRS   R20,TOV0         ;if TOV0 is set skip next instruction
    RJMP   AGAIN
    LDI    R20,0x00
    OUT    TCCR0,R20        ;stop Timer0
    LDI    R20,(1<<TOV0)
    OUT    TIFR,R20        ;clear TOV0 flag
    RET
```

### Example 9-8

Assuming that XTAL = 8 MHz, modify the program in Example 9-7 to generate a square wave of 16 kHz frequency on pin PORTB.3.

#### Solution:

Look at the following steps.

- (a)  $T = 1 / F = 1 / 16 \text{ kHz} = 62.5 \mu\text{s}$  the period of the square wave.
- (b)  $1/2$  of it for the high and low portions of the pulse is  $31.25 \mu\text{s}$ .
- (c)  $31.25 \mu\text{s} / 0.125 \mu\text{s} = 250$  and  $256 - 250 = 6$ , which in hex is 0x06.
- (d) TCNT0 = 0x06.

### Using the Windows calculator to find TCNT0

The scientific calculator in Microsoft Windows is a handy and easy-to-use tool to find the TCNT0 value. Assume that we would like to find the TCNT0 value for a time delay that uses 135 clocks of  $0.125 \mu\text{s}$ . The following steps show the calculation:

1. Bring up the scientific calculator in MS Windows and select decimal.
2. Enter 135.
3. Select hex. This converts 135 to hex, which is 0x87.
4. Select +/- to give -135 decimal (0x79).
5. The lowest two digits (79) of this hex value are for TCNT0. We ignore all the Fs on the left because our number is 8-bit data.

### Prescaler and generating a large time delay

As we have seen in the examples so far, the size of the time delay depends on two factors, (a) the crystal frequency, and (b) the timer's 8-bit register. Both of

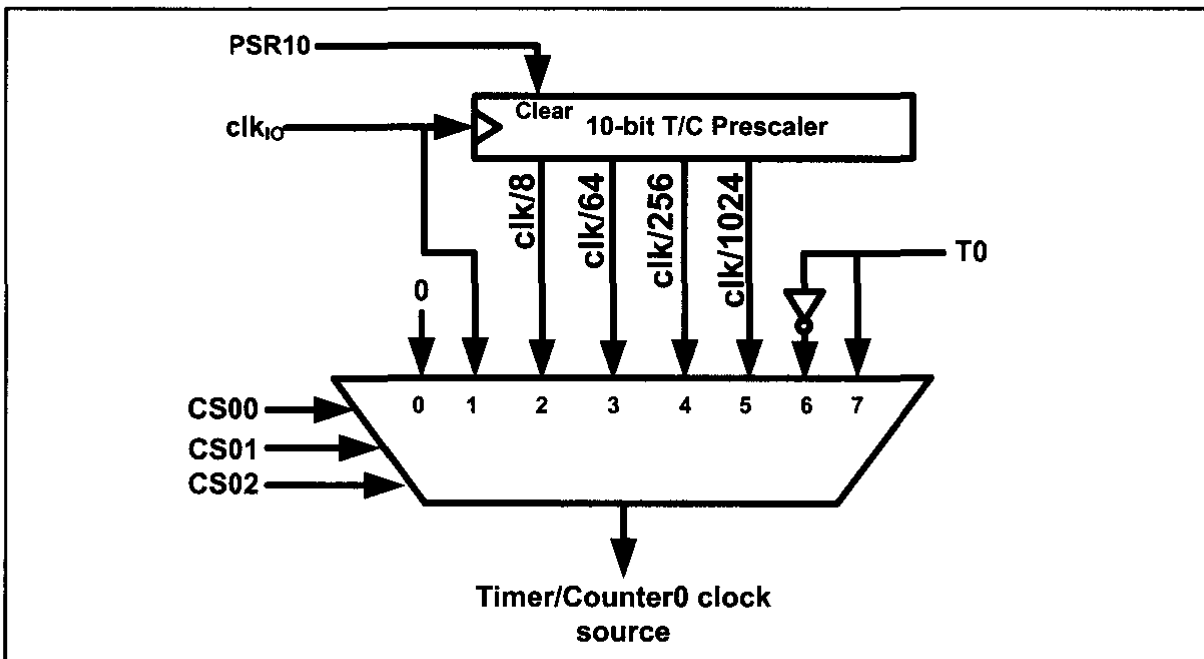


Figure 9-9. Timer/Counter 0 Prescaler

### Example 9-9

Modify TCNT0 in Example 9-7 to get the largest time delay possible. Find the delay in ms. In your calculation, exclude the overhead due to the instructions in the loop.

#### Solution:

To get the largest delay we make TCNT0 zero. This will count up from 00 to 0xFF and then roll over to zero.

```
.INCLUDE "M32DEF.INC"
    INITSTACK          ;add its definition from Example 9-3
    LDI    R16,0x08
    SBI    DDRB,3       ;PB3 as an output
    LDI    R17,0
    OUT    PORTB,R17
BEGIN:RCALL DELAY
    EOR    R17,R16      ;toggle D3 of R17
    OUT    PORTB,R17    ;toggle PB3
    RJMP   BEGIN
;----- Timer0 Delay
DELAY:LDI    R20,0x00
    OUT    TCNT0,R20    ;load Timer0 with zero
    LDI    R20,0x01
    OUT    TCCR0,R20    ;Timer0, Normal mode, int clk, no prescaler
AGAIN:IN     R20,TIFR    ;read TIFR
    SBRS   R20,TOV0     ;if TOV0 is set skip next instruction
    RJMP   AGAIN
    LDI    R20,0x00
    OUT    TCCR0,R20    ;stop Timer0
    LDI    R20,(1<<TOV0)
    OUT    TIFR,R20     ;clear TOV0 flag
    RET
```

Making TCNT0 zero means that the timer will count from 00 to 0xFF, and then will roll over to raise the TCNT0 flag. As a result, it goes through a total of 256 states. Therefore, we have delay =  $(256 - 0) \times 0.125 \mu\text{s} = 32 \mu\text{s}$ . That gives us the smallest frequency of  $1 / (2 \times 32 \mu\text{s}) = 1 / (64 \mu\text{s}) = 15.625 \text{ kHz}$ .

these factors are beyond the control of the AVR programmer. We saw in Example 9-9 that the largest time delay is achieved by making TCNT0 zero. What if that is not enough? We can use the prescaler option in the TCCR0 register to increase the delay by reducing the period. The prescaler option of TCCR0 allows us to divide the instruction clock by a factor of 8 to 1024 as was shown in Figure 9-5. The prescaler of Timer/Counter 0 is shown in Figure 9-9.

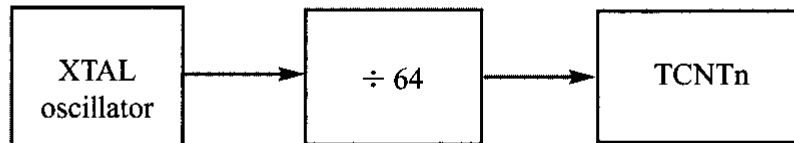
As we have seen so far, with no prescaler enabled, the crystal oscillator frequency is fed directly into Timer0. If we enable the prescaler bit in the TCCR0 register, however, then we can divide the clock before it is fed into Timer0. The lower 3 bits of the TCCR0 register give the options of the number we can divide by. As shown in Figure 9-9, this number can be 8, 64, 256, and 1024. Notice that the lowest number is 8 and the highest number is 1024. Examine Examples 9-10 through 9-14 to see how the prescaler options are programmed.

### Example 9-10

Find the timer's clock frequency and its period for various AVR-based systems, with the following crystal frequencies. Assume that a prescaler of 1:64 is used.

- (a) 8 MHz                      (b) 16 MHz                      (c) 10 MHz

**Solution:**



- (a)  $1/64 \times 8 \text{ MHz} = 125 \text{ kHz}$  due to 1:64 prescaler and  $T = 1/125 \text{ kHz} = 8 \mu\text{s}$   
(b)  $1/64 \times 16 \text{ MHz} = 250 \text{ kHz}$  due to prescaler and  $T = 1/250 \text{ kHz} = 4 \mu\text{s}$   
(c)  $1/64 \times 10 \text{ MHz} = 156.2 \text{ kHz}$  due to prescaler and  $T = 1/156 \text{ kHz} = 6.4 \mu\text{s}$

### Example 9-11

Find the value for TCCR0 if we want to program Timer0 in Normal mode with a prescaler of 64 using internal clock for the clock source.

**Solution:**

From Figure 9-5 we have TCCR0 = 0000 0011; XTAL clock source, prescaler of 64.

TCCR0 =	0	0	0	0	0	0	1	1
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

### Example 9-12

Examine the following program and find the time delay in seconds. Exclude the overhead due to the instructions in the loop. Assume XTAL = 8 MHz.

```
.INCLUDE "M32DEF.INC"
    INITSTACK          ;add its definition from Example 9-3
    LDI    R16,0x08
    SBI    DDRB,3       ;PB3 as an output
    LDI    R17,0
    OUT    PORTB,R17
BEGIN:RCALL DELAY
    EOR    R17,R16      ;toggle D3 of R17
    OUT    PORTB,R17    ;toggle PB3
    RJMP   BEGIN
;----- Timer0 Delay
DELAY:LDI    R20,0x10
    OUT     TCNT0,R20    ;load Timer0
    LDI     R20,0x03
    OUT     TCCR0,R20    ;Timer0, Normal mode, int clk, prescaler 64
AGAIN:IN     R20,TIFR    ;read TIFR
    SBRS    R20,TOV0     ;if TOV0 is set skip next instruction
    RJMP    AGAIN
    LDI     R20,0x00
```

**Example 9-12 (Cont.)**

```

OUT    TCCR0,R20    ;stop Timer0
LDI     R20,1<<TOV0
OUT     TIFR,R20     ;clear TOV0 flag
RET

```

**Solution:**

TCNT0 = 0x10 = 16 in decimal and  $256 - 16 = 240$ . Now  $240 \times 64 \times 0.125 \mu\text{s} = 1920 \mu\text{s}$ , or from Example 9-10, we have  $240 \times 8 \mu\text{s} = 1920 \mu\text{s}$ .

**Example 9-13**

Assume XTAL = 8 MHz. (a) Find the clock period fed into Timer0 if a prescaler option of 1024 is chosen. (b) Show what is the largest time delay we can get using this prescaler option and Timer0.

**Solution:**

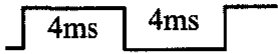
- (a)  $8 \text{ MHz} \times 1/1024 = 7812.5 \text{ Hz}$  due to 1:1024 prescaler and  $T = 1/7812.5 \text{ Hz} = 128 \text{ ms} = 0.128 \text{ ms}$
- (b) To get the largest delay, we make TCNT0 zero. Making TCNT0 zero means that the timer will count from 00 to 0xFF, and then roll over to raise the TOV0 flag. As a result, it goes through a total of 256 states. Therefore, we have delay =  $(256 - 0) \times 128 \mu\text{s} = 32,768 \mu\text{s} = 0.032768 \text{ seconds}$ .

**Example 9-14**

Assuming XTAL = 8 MHz, write a program to generate a square wave of 125 Hz frequency on pin PORTB.3. Use Timer0, Normal mode, with prescaler = 256.

**Solution:**

Look at the following steps:

- (a)  $T = 1 / 125 \text{ Hz} = 8 \text{ ms}$ , the period of the square wave.
- (b) 1/2 of it for the high and low portions of the pulse = 4 ms 
- (c)  $(4 \text{ ms} / 0.125 \mu\text{s}) / 256 = 125$  and  $256 - 125 = 131$  in decimal, and in hex it is 0x83.
- (d) TCNT0 = 83 (hex)

```

.INCLUDE "M32DEF.INC"
.MACRO INITSTACK      ;set up stack
LDI     R20,HIGH(RAMEND)
OUT     SPH,R20
LDI     R20,LOW(RAMEND)
OUT     SPL,R20
.ENDMACRO

```

### Example 9-14 (Cont.)

```
INITSTACK
LDI    R16,0x08
SBI    DDRB,3      ;PB3 as an output
LDI    R17,0
BEGIN:OUT    PORTB,R17 ;PORTB = R17
CALL   DELAY
EOR    R17,R16     ;toggle D3 of R17
RJMP   BEGIN

;----- Timer0 Delay
DELAY:LDI    R20,0x83
OUT     TCNT0,R20  ;load Timer0
LDI     R20,0x04
OUT     TCCR0,R20  ;Timer0, Normal mode, int clk, prescaler 256

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

LDI     R20,0x0
OUT     TCCR0,R20    ;stop Timer0
LDI     R20,1<<TOV0
OUT     TIFR,R20     ;clear TOV0 flag
RET
```

## Assemblers and negative values

Because the timer is in 8-bit mode, we can let the assembler calculate the value for TCNT0. For example, in the “LDI R20, -100” instruction, the assembler will calculate the  $-100 = 9C$  and make  $R20 = 9C$  in hex. This makes our job easier. See Examples 9-15 and 9-16.

### Example 9-15

Find the value (in hex) loaded into TCNT0 for each of the following cases.

- |     |                |     |                |     |                |
|-----|----------------|-----|----------------|-----|----------------|
| (a) | LDI R20, -200  | (b) | LDI R17, -60   | (c) | LDI R25, -12   |
|     | OUT TCNT0, R20 |     | OUT TCNT0, R17 |     | OUT TCNT0, R25 |

#### Solution:

You can use the Windows scientific calculator to verify the results provided by the assembler. In the Windows calculator, select decimal and enter 200. Then select hex, then +/- to get the negative value. The following is what we get.

<i>Decimal</i>	<i>2's complement (TCNT0 value)</i>
-200	0x38
-60	0xC4
-12	0xF4



### Example 9-16

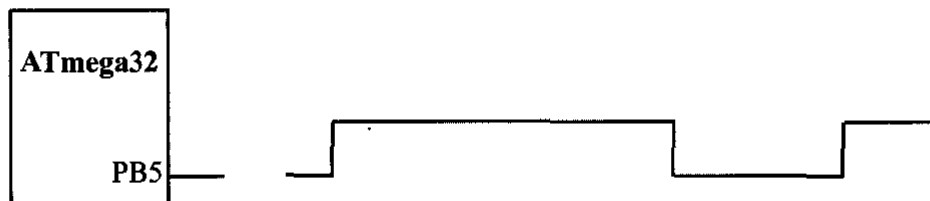
Find (a) the frequency of the square wave generated in the following code, and (b) the duty cycle of this wave. Assume XTAL = 8 MHz.

```
.INCLUDE "M32DEF.INC"
    LDI    R16,HIGH(RAMEND)
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16           ;initialize stack pointer
    LDI    R16,0x20
    SBI    DDRB,5           ;PB5 as an output
    LDI    R18,-150
BEGIN:SBI    PORTB,5         ;PB5 = 1
    OUT    TCNT0,R18        ;load Timer0 byte
    CALL   DELAY
    OUT    TCNT0,R18        ;reload Timer0 byte
    CALL   DELAY
    CBI    PORTB,5         ;PB5 = 0
    OUT    TCNT0,R18        ;reload Timer0 byte
    CALL   DELAY
    RJMP   BEGIN

;----- Delay using Timer0
DELAY:LDI    R20,0x01
    OUT     TCCR0,R20       ;start Timer0, Normal mode, int clk, no prescaler
AGAIN:IN     R20,TIFR       ;read TIFR
    SBRS    R20,TOV0       ;monitor TOV0 flag and skip if high
    RJMP    AGAIN
    LDI     R20,0x0
    OUT     TCCR0,R20       ;stop Timer0
    LDI     R20,1<<TOV0
    OUT     TIFR,R20       ;clear TOV0 flag bit
    RET
```

#### Solution:

For the TCNT0 value in 8-bit mode, the conversion is done by the assembler as long as we enter a negative number. This also makes the calculation easy. Because we are using 150 clocks, we have time for the DELAY subroutine =  $150 \times 0.125 \mu\text{s} = 18.75 \mu\text{s}$ . The high portion of the pulse is twice the size of the low portion (66% duty cycle). Therefore, we have:  $T = \text{high portion} + \text{low portion} = 2 \times 18.75 \mu\text{s} + 18.75 \mu\text{s} = 56.25 \mu\text{s}$  and frequency =  $1 / 56.25 \mu\text{s} = 17.777 \text{ kHz}$ .



## Clear Timer0 on compare match (CTC) mode programming

Examining Figure 9-2 once more, we see the OCR0 register. The OCR0 register is used with CTC mode. As with the Normal mode, in the CTC mode, the timer is incremented with a clock. But it counts up until the content of the TCNT0 register becomes equal to the content of OCR0 (compare match occurs); then, the timer will be cleared and the OCF0 flag will be set when the next clock occurs. The OCF0 flag is located in the TIFR register. See Figure 9-10 and Examples 9-17 through 9-21.

### Example 9-17

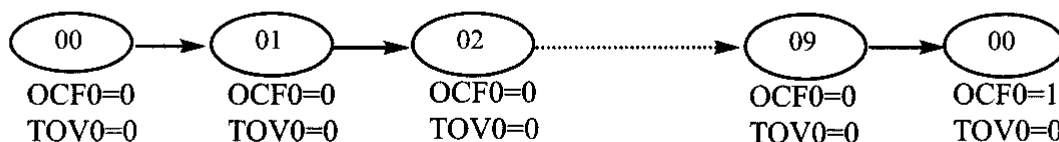
In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PORTB.5 bit. Timer0 is used to generate the time delay. Analyze the program.

```
.INCLUDE "M32DEF.INC"
    INITSTACK                ;add its definition from Example 9-3
    LDI    R16,0x08
    SBI    DDRB,3            ;PB3 as an output
    LDI    R17,0
BEGIN:OUT    PORTB,R17        ;PORTB = R17
    RCALL  DELAY
    EOR    R17,R16           ;toggle D3 of R17
    RJMP   BEGIN
;----- Timer0 Delay
DELAY:LDI    R20,0
    OUT    TCNT0,R20
    LDI    R20,9
    OUT    OCR0,R20          ;load OCR0
    LDI    R20,0x09
    OUT    TCCR0,R20         ;Timer0, CTC mode, int clk
AGAIN:IN     R20,TIFR         ;read TIFR
    SBRS   R20,OCF0          ;if OCF0 is set skip next inst.
    RJMP   AGAIN
    LDI    R20,0x0
    OUT    TCCR0,R20         ;stop Timer0
    LDI    R20,1<<OCF0
    OUT    TIFR,R20          ;clear OCF0 flag
    RET
```

### Solution:

In the above program notice the following steps:

1. 9 is loaded into OCR0.
2. TCCR0 is loaded and Timer0 is started.
3. Timer0 counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of 00, 01, 02, 03, and so on until it reaches 9. One more clock rolls it to 0, raising the Timer0 compare match flag (OCF0 = 1). At that point, the "SBRS R20,OCF0" instruction bypasses the "RJMP AGAIN" instruction.
4. Timer0 is stopped.
5. The OCF0 flag is cleared.



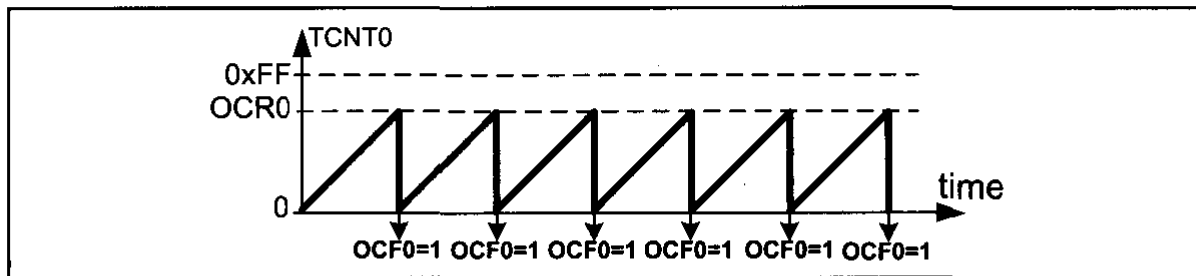


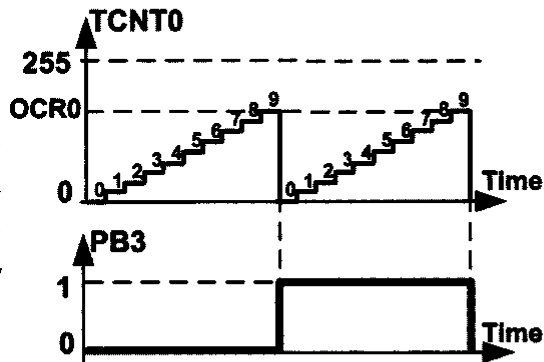
Figure 9-10. Timer/Counter 0 CTC Mode

### Example 9-18

Find the delay generated by Timer0 in Example 9-17. Do not include the overhead due to instructions. (XTAL = 8 MHz)

#### Solution:

OCR0 is loaded with 9 and TCNT0 is cleared; Thus, after 9 clocks TCNT0 becomes equal to OCR0. On the next clock, the OCF0 flag is set and the reset occurs. That means the TCNT0 is cleared after  $9 + 1 = 10$  clocks. Because XTAL = 8 MHz, the counter counts up every  $0.125 \mu\text{s}$ . Therefore, we have  $10 \times 0.125 \mu\text{s} = 1.25 \mu\text{s}$ .



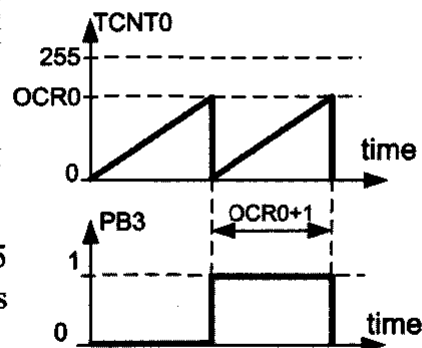
### Example 9-19

Find the delay generated by Timer0 in the following program. Do not include the overhead due to instructions. (XTAL = 8 MHz)

```
.INCLUDE "M32DEF.INC"
LDI    R16,0x08
SBI    DDRB,3        ;PB3 as an output
LDI    R17,0
OUT    PORTB,R17
LDI    R20,89
OUT    OCR0,R20      ;load Timer0
BEGIN:LDI    R20,0x0B
OUT    TCCR0,R20     ;Timer0, CTC mode, prescaler = 64
AGAIN:IN    R20,TIFR  ;read TIFR
SBRs    R20,OCF0     ;if OCF0 flag is set skip next instruction
RJMP    AGAIN
LDI    R20,0x0
OUT    TCCR0,R20     ;stop Timer0 (This line can be omitted)
LDI    R20,1<<OCF0
OUT    TIFR,R20      ;clear OCF0 flag
EOR    R17,R16       ;toggle D3 of R17
OUT    PORTB,R17     ;toggle PB3
RJMP    BEGIN
```

#### Solution:

Due to prescaler = 64 each timer clock lasts  $64 \times 0.125 \mu\text{s} = 8 \mu\text{s}$ . OCR0 is loaded with 89; thus, after 90 clocks OCF0 is set. Therefore we have  $90 \times 8 \mu\text{s} = 720 \mu\text{s}$ .



### Example 9-20

Assuming XTAL = 8 MHz, write a program to generate a delay of 25.6 ms. Use Timer0, CTC mode, with prescaler = 1024.

#### Solution:

Due to prescaler = 1024 each timer clock lasts  $1024 \times 0.125 \mu\text{s} = 128 \mu\text{s}$ . Thus, in order to generate a delay of 25.6 ms we should wait  $25.6 \text{ ms} / 128 \mu\text{s} = 200$  clocks. Therefore the OCR0 register should be loaded with  $200 - 1 = 199$ .

```
DELAY:LDI    R20,0
          OUT    TCNT0,R20
          LDI    R20,199
          OUT    OCR0,R20           ;load OCR0
          LDI    R20,0x0D
          OUT    TCCR0,R20         ;Timer0, CTC mode, prescaler = 1024
AGAIN:IN    R20,TIFR               ;read TIFR
          SBRS   R20,OCF0          ;if OCF0 is set skip next inst.
          RJMP   AGAIN
          LDI    R20,0x0
          OUT    TCCR0,R20         ;stop Timer0
          LDI    R20,1<<OCF0
          OUT    TIFR,R20         ;clear OCF0 flag
          RET
```

### Example 9-21

Assuming XTAL = 8 MHz, write a program to generate a delay of 1 ms.

#### Solution:

As XTAL = 8 MHz, the different outputs of the prescaler are as follows:

Prescaler	Timer Clock	Timer Period	Timer Value
None	8 MHz	$1/8 \text{ MHz} = 0.125 \mu\text{s}$	$1 \text{ ms}/0.125 \mu\text{s} = 8000$
8	$8 \text{ MHz}/8 = 1 \text{ MHz}$	$1/1 \text{ MHz} = 1 \mu\text{s}$	$1 \text{ ms}/1 \mu\text{s} = 1000$
64	$8 \text{ MHz}/64 = 125 \text{ kHz}$	$1/125 \text{ kHz} = 8 \mu\text{s}$	$1 \text{ ms}/8 \mu\text{s} = 125$
256	$8 \text{ MHz}/256 = 31.25 \text{ kHz}$	$1/31.25 \text{ kHz} = 32 \mu\text{s}$	$1 \text{ ms}/32 \mu\text{s} = 31.25$
1024	$8 \text{ MHz}/1024 = 7.8125 \text{ kHz}$	$1/7.8125 \text{ kHz} = 128 \mu\text{s}$	$1 \text{ ms}/128 \mu\text{s} = 7.8125$

From the above calculation we can only use the options Prescaler = 64, Prescaler = 256, or Prescaler = 1024. We should use the option Prescaler = 64 since we cannot use a decimal point. To wait 125 clocks we should load OCR0 with  $125 - 1 = 124$ .

```
DELAY:LDI    R20,0
          OUT    TCNT0,R20         ;TCNT0 = 0
          LDI    R20,124
          OUT    OCR0,R20         ;OCR0 = 124
          LDI    R20,0x0B
          OUT    TCCR0,R20         ;Timer0, CTC mode, prescaler = 64
AGAIN:IN    R20,TIFR               ;read TIFR
          SBRS   R20,OCF0          ;if OCF0 is set skip next instruction
          RJMP   AGAIN
          LDI    R20,0x0
          OUT    TCCR0,R20         ;stop Timer0
          LDI    R20,1<<OCF0
          OUT    TIFR,R20         ;clear OCF0 flag
          RET
```

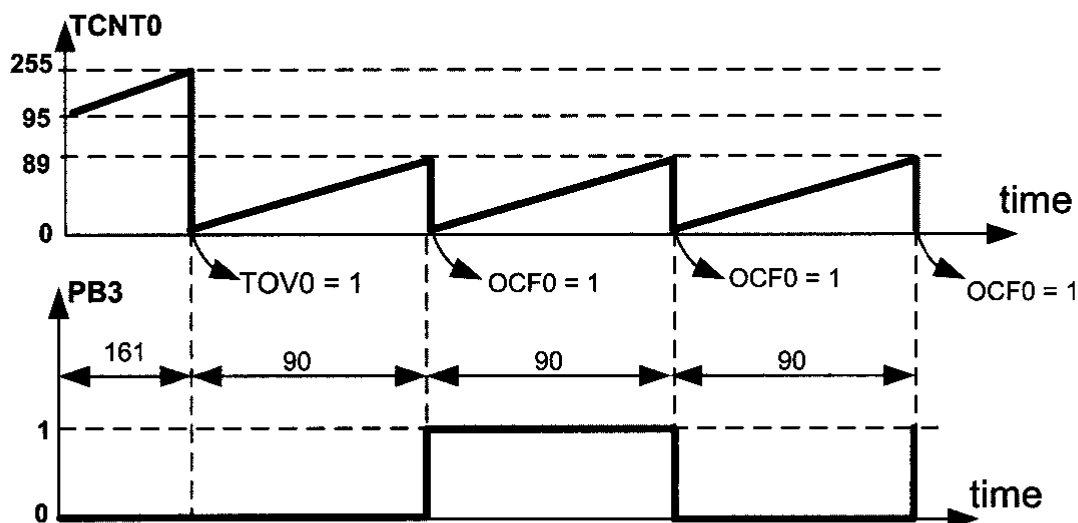
Notice that the comparator checks for equality; thus, if we load the OCR0 register with a value that is smaller than TCNT0's value, the counter will miss the compare match and will count up until it reaches the maximum value of \$FF and rolls over. This causes a big delay and is not desirable in many cases. See Example 9-22.

### Example 9-22

In the following program, how long does it take for the PB3 to become one? Do not include the overhead due to instructions. (XTAL = 8 MHz)

```
.INCLUDE "M32DEF.INC"
    SBI    DDRB,3           ;PB3 as an output
    CBI    PORTB,3          ;PB3 = 0
    LDI    R20,89
    OUT    OCR0,R20         ;OCR0 = 89
    LDI    R20,95
    OUT    TCNT0,R20        ;TCNT0 = 95
BEGIN:LDI    R20,0x09
    OUT    TCCR0,R20        ;Timer0, CTC mode, prescaler = 1
AGAIN:IN    R20,TIFR        ;read TIFR
    SBRS   R20,OCF0         ;if OCF0 flag is set skip next inst.
    RJMP   AGAIN
    LDI    R20,0x0
    OUT    TCCR0,R20        ;stop Timer0 (This line can be omitted)
    LDI    R20,1<<OCF0
    OUT    TIFR,R20         ;clear OCF0 flag
    EOR    R17,R16          ;toggle D3 of R17
    OUT    PORTB,R17        ;toggle PB3
    RJMP   BEGIN
```

### Solution:



Since the value of TCNT0 (95) is bigger than the content of OCR0 (89), the timer counts up until it gets to \$FF and rolls over to zero. The TOV0 flag will be set as a result of the overflow. Then, the timer counts up until it becomes equal to 89 and compare match occurs. Thus, the first compare match occurs after  $161 + 90 = 251$  clocks, which means after  $251 \times 0.125 \mu\text{s} = 31.375 \mu\text{s}$ . The next compare matches occur after 90 clocks, which means after  $90 \times 0.125 \mu\text{s} = 11.25 \mu\text{s}$ .

## Timer2 programming

See Figure 9-12. Timer2 is an 8-bit timer. Therefore it works the same way as Timer0. But there are two differences between Timer0 and Timer2:

1. Timer2 can be used as a real time counter. To do so, we should connect a crystal of 32.768 kHz to the TOSC1 and TOSC2 pins of AVR and set the AS2 bit. See Figure 9-12. For more information about this feature, see the AVR datasheet.

2. In Timer0, when CS02–CS00 have values 110 or 111, Timer0 counts the external events. But in Timer2, the multiplexer selects between the different scales of the clock. In other words, the same values of the CS bits can have different meanings for Timer0 and Timer2. Compare Figure 9-11 with Figure 9-5 and examine Examples 9-23 through 9-25.

Bit	7	6	5	4	3	2	1	0
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
Read/Write	W	RW	RW	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0
<b>FOC2</b>	D7	Force compare match: a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.						
<b>WGM20, WGM21</b>	D6	D3	Timer2 mode selector bits					
	0	0	Normal					
	0	1	CTC (Clear Timer on Compare Match)					
	1	0	PWM, phase correct					
	1	1	Fast PWM					
<b>COM21:20</b>	D5	D4	Compare Output Mode: These bits control the waveform generator (see Chapter 15).					
<b>CS22:20</b>	D2	D1	D0	Timer2 clock selector				
	0	0	0	No clock source (Timer/Counter stopped)				
	0	0	1	clk (No Prescaling)				
	0	1	0	clk / 8				
	0	1	1	clk / 32				
	1	0	0	clk / 64				
	1	0	1	clk / 128				
	1	1	0	clk / 256				
	1	1	1	clk / 1024				

Figure 9-11. TCCR2 (Timer/Counter Control Register) Register

### Example 9-23

Find the value for TCCR2 if we want to program Timer2 in normal mode with a prescaler of 64 using internal clock for the clock source.

#### Solution:

From Figure 9-11 we have TCCR2 = 0000 0100; XTAL clock source, prescaler of 64.

TCCR2 =

0	0	0	0	0	1	0	0
FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20

Compare the answer with Example 9-11.

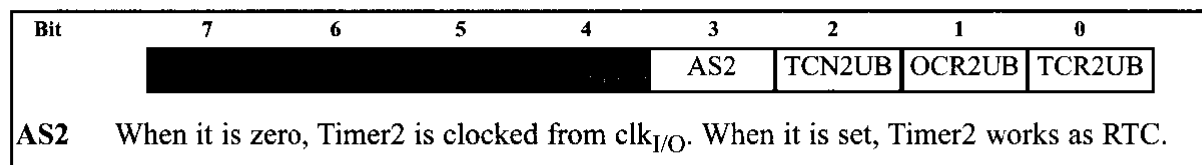


Figure 9-12. ASSR (Asynchronous Status Register)

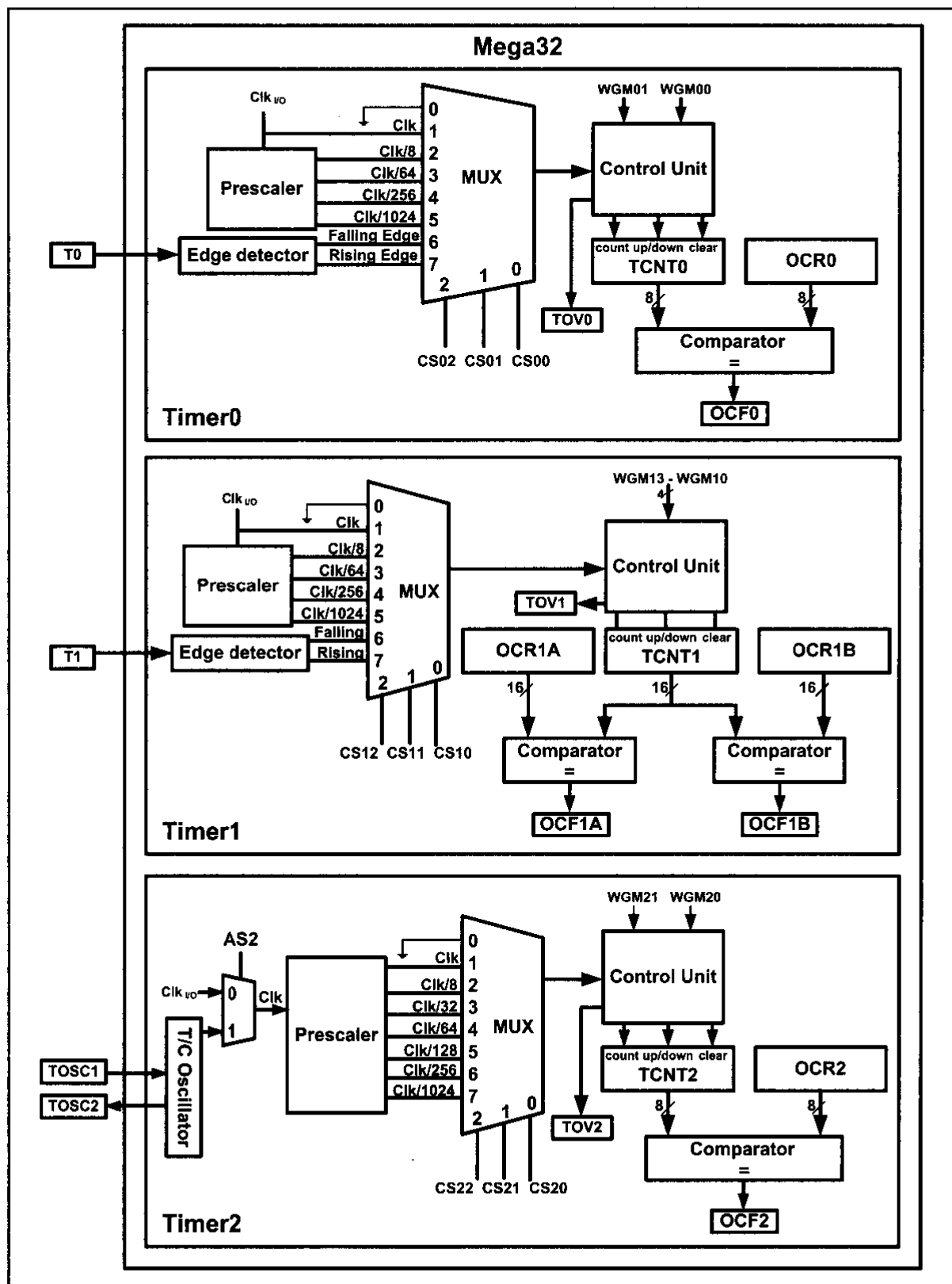


Figure 9-13. Timers in ATmega32

### Example 9-24

Using a prescaler of 64, write a program to generate a delay of 1920  $\mu$ s. Assume XTAL = 8 MHz.

#### Solution:

Timer clock = 8 MHz/64 = 125 kHz  $\rightarrow$  Timer Period = 1 / 125 kHz = 8  $\mu$ s  $\rightarrow$

Timer Value = 1920  $\mu$ s / 8  $\mu$ s = 240

```
;----- Timer2 Delay
DELAY:LDI    R20,-240    ;R20 = 0x10
          OUT    TCNT2,R20    ;load Timer2
          LDI    R20,0x04
          OUT    TCCR2,R20    ;Timer2, Normal mode, int clk, prescaler 64
AGAIN:IN     R20,TIFR     ;read TIFR
          SBRS   R20,TOV2    ;if TOV2 is set skip next instruction
          RJMP   AGAIN
          LDI    R20,0x0
          OUT    TCCR2,R20    ;stop Timer2
          LDI    R20,1<<TOV2
          OUT    TIFR,R20     ;clear TOV2 flag
          RET
```

Compare the above program with the DELAY subroutine in Example 9-12. There are two differences between the two programs:

1. The register names are different. For example, we use TCNT2 instead of TCNT0.
2. The values of TCCRn are different for the same prescaler.

### Example 9-25

Using CTC mode, write a program to generate a delay of 8 ms. Assume XTAL = 8 MHz.

#### Solution:

As XTAL = 8 MHz, the different outputs of the prescaler are as follows:

<u>Prescaler</u>	<u>Timer Clock</u>	<u>Timer Period</u>	<u>Timer Value</u>
None	8 MHz	1/8 MHz = 0.125 $\mu$ s	8 ms / 0.125 $\mu$ s = 64 k
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 $\mu$ s	8 ms / 1 $\mu$ s = 8000
32	8 MHz/32 = 250 kHz	1/250 kHz = 4 $\mu$ s	8 ms / 4 $\mu$ s = 2000
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 $\mu$ s	8 ms / 8 $\mu$ s = 1000
128	8 MHz/128 = 62.5 kHz	1/62.5 kHz = 16 $\mu$ s	8 ms / 16 $\mu$ s = 500
<b>256</b>	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 $\mu$ s	8 ms / 32 $\mu$ s = <b>250</b>
<b>1024</b>	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz = 128 $\mu$ s	8 ms / 128 $\mu$ s = <b>62.5</b>

From the above calculation we can only use options Prescaler = 256 or Prescaler = 1024. We should use the option Prescaler = 256 since we cannot use a decimal point. To wait 250 clocks we should load OCR2 with 250 - 1 = 249.



### Example 9-25 (Cont.)

```
TCCR2 = 

|      |       |       |       |       |      |      |      |
|------|-------|-------|-------|-------|------|------|------|
| 0    | 0     | 0     | 0     | 1     | 1    | 1    | 0    |
| FOC2 | WGM20 | COM21 | COM20 | WGM21 | CS22 | CS21 | CS20 |


```

----- Timer2 Delay

```
DELAY:LDI    R20,0
      OUT    TCNT2,R20          ;TCNT2 = 0
      LDI    R20,249
      OUT    OCR0,R20          ;OCR0 = 249
      LDI    R20,0x0E
      OUT    TCCR0,R20          ;Timer0,CTC mode,prescaler = 256
AGAIN:IN     R20,TIFR           ;read TIFR
      SBRS   R20,OCF2          ;if OCF2 is set skip next inst.
      RJMP   AGAIN
      LDI    R20,0x0
      OUT    TCCR2,R20          ;stop Timer2
      LDI    R20,1<<OCF2
      OUT    TIFR,R20          ;clear OCF2 flag
      RET
```

## Timer1 programming

Timer1 is a 16-bit timer and has lots of capabilities. Next, we discuss Timer1 and its capabilities.

Since Timer1 is a 16-bit timer its 16-bit register is split into two bytes. These are referred to as TCNT1L (Timer1 low byte) and TCNT1H (Timer1 high byte). See Figure 9-15. Timer1 also has two control registers named TCCR1A (Timer/counter 1 control register) and TCCR1B. The TOV1 (timer overflow) flag bit goes HIGH when overflow occurs. Timer1 also has the prescaler options of 1:1, 1:8, 1:64, 1:256, and 1:1024. See Figure 9-14 for the Timer1 block diagram and Figures 9-15 and 9-16 for TCCR1 register options. There are two OCR registers in Timer1: OCR1A and OCR1B. There

are two separate flags for each of the OCR registers, which act independently of each other. Whenever TCNT1 equals OCR1A, the OCF1A flag will be set on

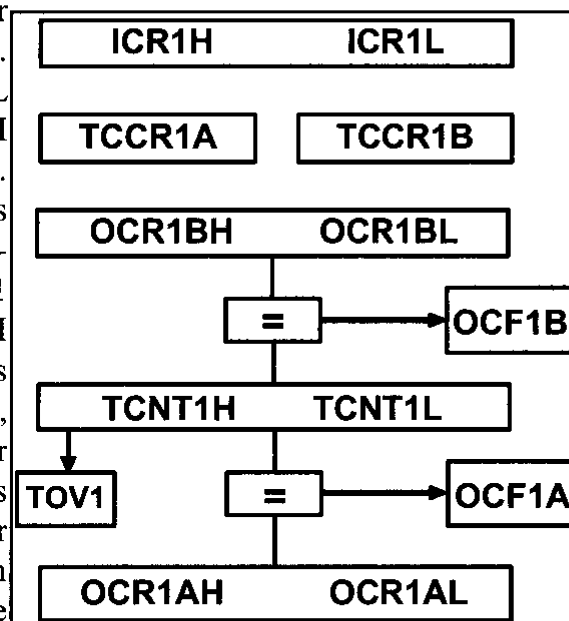


Figure 9-14. Simplified Diagram of Timer1

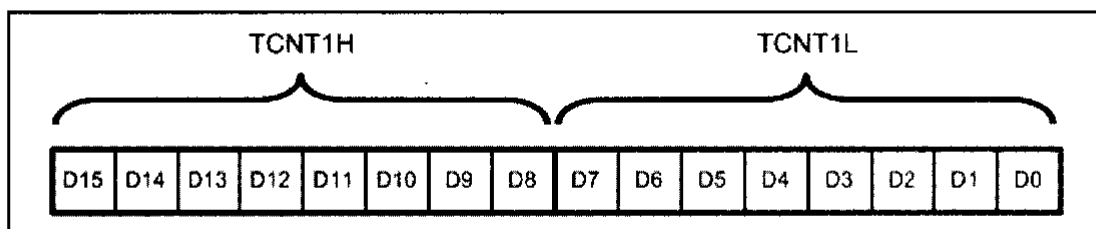


Figure 9-15. Timer1 High and Low Registers

the next timer clock. When TCNT equals OCR1B, the OCF1B flag will be set on the next clock. As Timer1 is a 16-bit timer, the OCR registers are 16-bit registers as well and they are made of two 8-bit registers. For example, OCR1A is made of OCR1AH (OCR1A high byte) and OCR1AL (OCR1A low byte). For a detailed view of Timer1 see Figure 9-13.

The TIFR register contains the TOV1, OCF1A, and OCF1B flags. See Figure 9-16.

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
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
<b>TOV0</b>	D0	Timer0 overflow flag bit 0 = Timer0 did not overflow. 1 = Timer0 has overflowed (going from \$FF to \$00).						
<b>OCF0</b>	D1	Timer0 output compare flag bit 0 = compare match did not occur. 1 = compare match occurred.						
<b>TOV1</b>	D2	Timer1 overflow flag bit						
<b>OCF1B</b>	D3	Timer1 output compare B match flag						
<b>OCF1A</b>	D4	Timer1 output compare A match flag						
<b>ICF1</b>	D5	Input Capture flag						
<b>TOV2</b>	D6	Timer2 overflow flag						
<b>OCF2</b>	D7	Timer2 output compare match flag						

**Figure 9-16. TIFR (Timer/Counter Interrupt Flag Register)**

There is also an auxiliary register named ICR1, which is used in operations such as capturing. ICR1 is a 16-bit register made of ICR1H and ICR1L, as shown in Figure 9-19.

Bit	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0
<b>COM1A1:COM1A0</b>	D7 D6	Compare Output Mode for Channel A (discussed in Section 9-3)						
<b>COM1B1:COM1B0</b>	D5 D4	Compare Output Mode for Channel B (discussed in Section 9-3)						
<b>FOC1A</b>	D3	Force Output Compare for Channel A (discussed in Section 9-3)						
<b>FOC1B</b>	D2	Force Output Compare for Channel B (discussed in Section 9-3)						
<b>WGM11:10</b>	D1 D0	Timer1 mode (discussed in Figure 9-18)						

**Figure 9-17. TCCR1A (Timer 1 Control ) Register**

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
<b>ICNC1</b>	D7 Input Capture Noise Canceler 0 = Input Capture is disabled. 1 = Input Capture is enabled.								
<b>ICES1</b>	D6 Input Capture Edge Select 0 = Capture on the falling (negative) edge 1 = Capture on the rising (positive) edge								
	D5 Not used								
<b>WGM13:WGM12</b>	D4 D3 Timer1 mode								
Mode	WGM13	WGM12	WGM11	WGM10	Timer/Counter Mode of Operation	Top	Update of OCR1x	TOV1 Flag Set on	
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX	
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM	
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM	
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM	
4	0	1	0	0	CTC	OCR1A	Immediate	MAX	
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP	
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP	
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP	
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM	
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM	
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM	
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM	
12	1	1	0	0	CTC	ICR1	Immediate	MAX	
13	1	1	0	1	Reserved	-	-	-	
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP	
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP	
<b>CS12:CS10</b>	D2D1D0 Timer1 clock selector 0 0 0 No clock source (Timer/Counter stopped) 0 0 1 clk (no prescaling) 0 1 0 clk / 8 0 1 1 clk / 64 1 0 0 clk / 256 1 0 1 clk / 1024 1 1 0 External clock source on T1 pin. Clock on falling edge. 1 1 1 External clock source on T1 pin. Clock on rising edge.								

Figure 9-18. TCCR1B (Timer 1 Control) Register

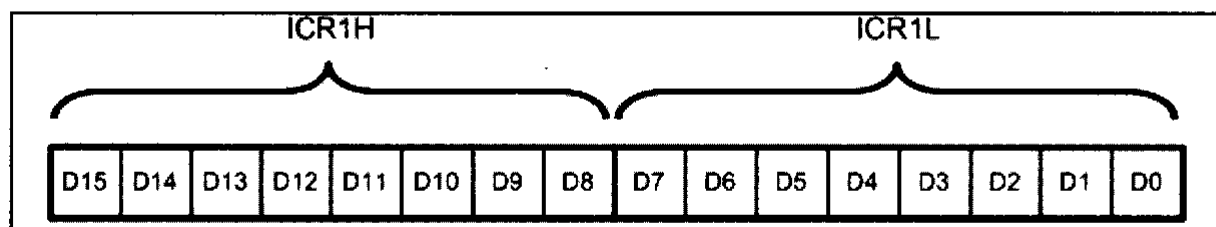


Figure 9-19. Input Capture Register (ICR) for Timer1

## WGM13:10

The WGM13, WGM12, WGM11, and WGM10 bits define the mode of Timer1, as shown in Figure 9-18. Timer1 has 16 different modes. One of them (mode 13) is reserved (not implemented). In this chapter, we cover mode 0 (Normal mode) and mode 4 (CTC mode). The other modes will be covered in Chapters 15 and 16.

### Timer1 operation modes

#### Normal mode (WGM13:10 = 0000)

In this mode, the timer counts up until it reaches \$FFFF (which is the maximum value) and then it rolls over from \$FFFF to 0000. When the timer rolls over from \$FFFF to 0000, the TOV1 flag will be set. See Figure 9-20 and Examples 9-26 and 9-27. In Example 9-27, a delay is generated using Normal mode.

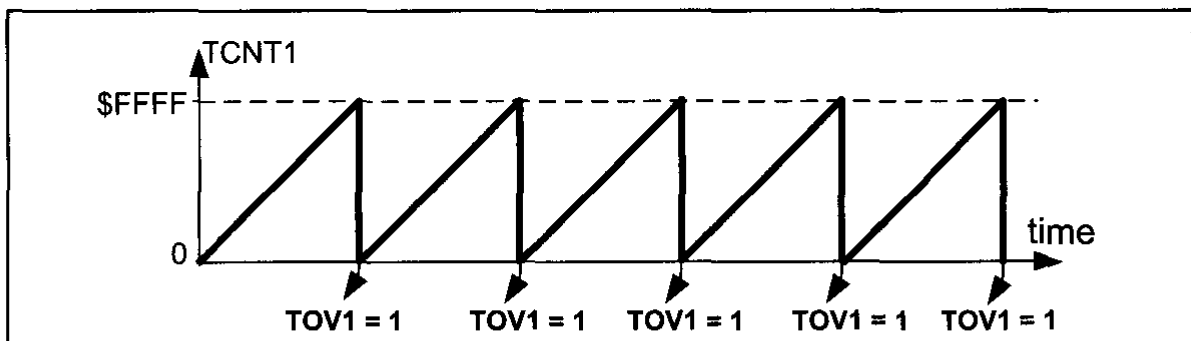


Figure 9-20. TOV in Normal and Fast PWM

#### CTC mode (WGM13:10 = 0100)

In mode 4, the timer counts up until the content of the TCNT1 register becomes equal to the content of OCR1A (compare match occurs); then, the timer will be cleared when the next clock occurs. The OCF1A flag will be set as a result of the compare match as well. See Figure 9-21 and Examples 9-28 and 9-29.

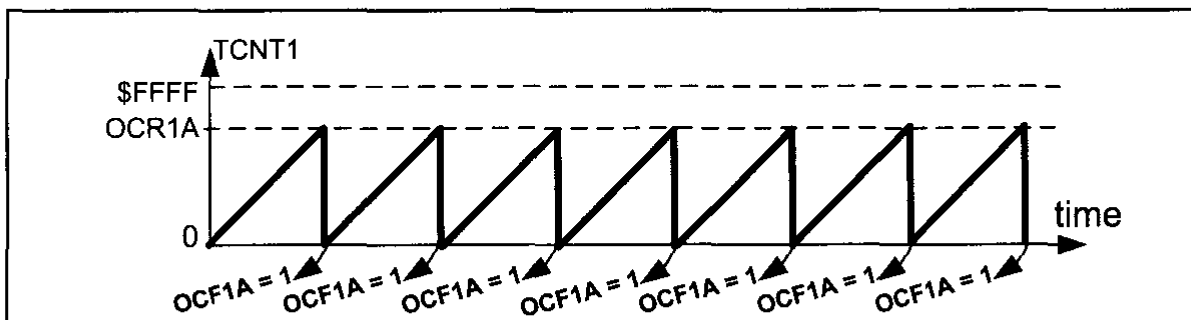


Figure 9-21. OCF1A in CTC Mode

#### Example 9-26

Find the values for TCCR1A and TCCR1B if we want to program Timer1 in mode 0 (Normal), with no prescaler. Use AVR's crystal oscillator for the clock source.

##### Solution:

TCCR1A = 0000 0000 WGM11 = 0, WGM10 = 0

TCCR1B = 0000 0001 WGM13 = 0, WGM12 = 0, oscillator clock source, no prescaler

### Example 9-27

Find the frequency of the square wave generated by the following program if XTAL = 8 MHz. In your calculation do not include the overhead due to instructions in the loop.

```
.INCLUDE "M32DEF.INC"
    INITSTACK          ;add its definition from Example 9-3
    LDI    R16,0x20
    SBI    DDRB,5       ;PB5 as an output
    LDI    R17,0
    OUT    PORTB,R17    ;PB5 = 0
BEGIN:RCALL DELAY
    EOR    R17,R16      ;toggle D5 of R17
    OUT    PORTB,R17    ;toggle PB5
    RJMP   BEGIN
;----- Timer1 delay
DELAY:LDI    R20,0xD8
    OUT    TCNT1H,R20   ;TCNT1H = 0xD8
    LDI    R20,0xF0
    OUT    TCNT1L,R20   ;TCNT1L = 0xF0
    LDI    R20,0x00
    OUT    TCCR1A,R20   ;WGM11:10 = 00
    LDI    R20,0x01
    OUT    TCCR1B,R20   ;WGM13:12 = 00, Normal mode, prescaler = 1
AGAIN:IN     R20,TIFR   ;read TIFR
    SBRS   R20,TOV1     ;if TOV1 is set skip next instruction
    RJMP   AGAIN
    LDI    R20,0x00
    OUT    TCCR1B,R20   ;stop Timer1
    LDI    R20,0x04
    OUT    TIFR,R20     ;clear TOV1 flag
    RET
```

#### Solution:

WGM13:10 = 0000 = 0x00, so Timer1 is working in mode 0, which is Normal mode, and the top is 0xFFFF.

FFFF + 1 - D8F0 = 0x2710 = 10,000 clocks, which means that it takes 10,000 clocks. As XTAL = 8 MHz each clock lasts  $1/(8M) = 0.125 \mu s$  and delay =  $10,000 \times 0.125 \mu s = 1250 \mu s = 1.25 ms$  and frequency =  $1 / (1.25 ms \times 2) = 400 Hz$ .

In this calculation, the overhead due to all the instructions in the loop is not included.

Notice that instead of using hex numbers we can use HIGH and LOW directives, as shown below:

```
LDI    R20,HIGH (65536-10000)    ;load Timer1 high byte
OUT    TCNT1H,R20   ;TCNT1H = 0xD8
LDI    R20,LOW (65536-10000)     ;load Timer1 low byte
OUT    TCNT1L,R20   ;TCNT1L = 0xF0
```

or we can simply write it as follows:

```
LDI    R20,HIGH (-10000)         ;load Timer1 high byte
OUT    TCNT1H,R20   ;TCNT1H = 0xD8
LDI    R20,LOW (-10000)         ;load Timer1 low byte
OUT    TCNT1L,R20   ;TCNT1L = 0xF0
```

**Example 9-28**

Find the values for TCCR1A and TCCR1B if we want to program Timer1 in mode 4 (CTC, Top = OCR1A), no prescaler. Use AVR's crystal oscillator for the clock source.

**Solution:**

TCCR1A = 0000 0000 WGM11 = 0, WGM10 = 0

TCCR1B = 0000 1001 WGM13 = 0, WGM12 = 1, oscillator clock source, no prescaler

**Example 9-29**

Find the frequency of the square wave generated by the following program if XTAL = 8 MHz. In your calculation do not include the overhead due to instructions in the loop.

```
.INCLUDE "M32DEF.INC"
      SBI   DDRB,5           ;PB5 as an output
BEGIN:SBI   PORTB,5          ;PB5 = 1
      RCALL DELAY
      CBI   PORTB,5          ;PB5 = 0
      RCALL DELAY
      RJMP  BEGIN
;----- Timer1 delay
DELAY:LDI   R20,0x00
      OUT   TCNT1H,R20
      OUT   TCNT1L,R20      ;TCNT1 = 0
      LDI   R20,0
      OUT   OCR1AH,R20
      LDI   R20,159
      OUT   OCR1AL,R20      ;OCR1A = 159 = 0x9F
      LDI   R20,0x0
      OUT   TCCR1A,R20      ;WGM11:10 = 00
      LDI   R20,0x09
      OUT   TCCR1B,R20      ;WGM13:12 = 01, CTC mode, prescaler = 1
AGAIN:IN    R20,TIFR         ;read TIFR
      SBRS  R20,OCF1A        ;if OCF1A is set skip next instruction
      RJMP  AGAIN
      LDI   R20,1<<OCF1A
      OUT   TIFR,R20         ;clear OCF1A flag
      LDI   R19,0
      OUT   TCCR1B,R19       ;stop timer
      OUT   TCCR1A,R19
      RET
```

**Solution:**

WGM13:10 = 0100 = 0x04 therefore, Timer1 is working in mode 4, which is a CTC mode, and max is defined by OCR1A.

$159 + 1 = 160$  clocks

XTAL = 8 MHz, so each clock lasts  $1/(8\text{M}) = 0.125 \mu\text{s}$ .

Delay =  $160 \times 0.125 \mu\text{s} = 20 \mu\text{s}$  and frequency =  $1 / (20 \mu\text{s} \times 2) = 25 \text{ kHz}$ .

In this calculation, the overhead due to all the instructions in the loop is not included.

## Accessing 16-bit registers

The AVR is an 8-bit microcontroller, which means it can manipulate data 8 bits at a time, only. But some Timer1 registers, such as TCNT1, OCR1A, ICR1, and so on, are 16-bit; in this case, the registers are split into two 8-bit registers, and each one is accessed individually. This is fine for most cases. For example, when we want to load the content of SP (stack pointer), we first load one half and then the other half, as shown below:

```
LDI R16, 0x12
OUT SPL, R16
LDI R16, 0x34
OUT SPH, R16 ;SP = 0x3412
```

In 16-bit timers, however, we should read/write the entire content of a register at once, otherwise we might have problems. For example, imagine the following scenario:

The TCNT1 register contains 0x15FF. We read the low byte of TCNT1, which is 0xFF, and store it in R20. At the same time a timer clock occurs, and the content of TCNT1 becomes 0x1600; now we read the high byte of TCNT1, which is now 0x16, and store it in R21. If we look at the value we have read, R21:R20 = 0x16FF. So, we believe that TCNT1 contains 0x16FF, although it actually contains 0x15FF.

This problem exists in many 8-bit microcontrollers. But the AVR designers have resolved this issue with an 8-bit register called TEMP, which is used as a buffer. See Figure 9-22. When we write or read the high byte of a 16-bit register, the value will be written into the TEMP register. When we write into the low byte of a 16-bit register, the content of TEMP will be written into the high byte of the 16-bit register as well. For example, consider the following program:

```
LDI R16, 0x15
OUT TCNT1H, R16 ;store 0x15 in TEMP of Timer1
LDI R16, 0xFF
OUT TCNT1L, R16 ;TCNT1L = R16, TCNT1H = TEMP
```

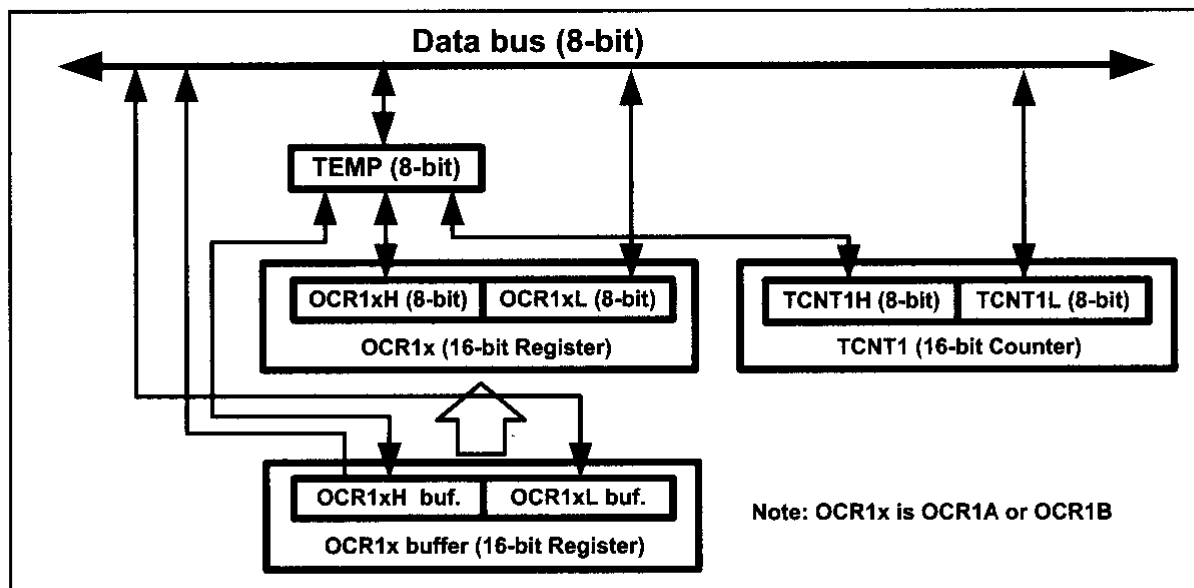


Figure 9-22. Accessing 16-bit Registers through TEMP

After the execution of “OUT TCNT1H, R16”, the content of R16, 0x15, will be stored in the TEMP register. When the instruction “OUT TCNT1L, R16” is executed, the content of R16, 0xFF, is loaded into TCNT1L, and the content of the TEMP register, 0x15, is loaded into TCNT1H. So, 0x15FF will be loaded into the TCNT1 register at once.

Notice that according to the internal circuitry of the AVR, we should first write into the high byte of the 16-bit registers and then write into the lower byte. Otherwise, the program does not work properly. For example, the following code:

```
LDI    R16, 0xFF
OUT    TCNT1L, R16      ;TCNT1L = R16, TCNT1H = TEMP
LDI    R16, 0x15
OUT    TCNT1H, R16      ;store 0x15 in TEMP of Timer1
```

does not work properly. This is because, when the TCNT1L is loaded, the content of TEMP will be loaded into TCNT1H. But when the TCNT1L register is loaded, TEMP contains garbage (improper data), and this is not what we want.

When we read the low byte of 16-bit registers, the content of the high byte will be copied to the TEMP register. So, the following program reads the content of TCNT1:

```
IN     R20, TCNT1L      ;R20 = TCNT1L, TEMP = TCNT1H
IN     R21, TCNT1H      ;R21 = TEMP of Timer1
```

We must pay attention to the order of reading the high and low bytes of the 16-bit registers. Otherwise, the result is erroneous.

Notice that reading the OCR1A and OCR1B registers does not involve using the temporary register. You might be wondering why. It is because the AVR microcontroller does not update the content of OCR1A nor OCR1B unless we update them. For example, consider the following program:

```
IN     R20, OCR1AL      ;R20 = OCR1L
IN     R21, OCR1AH      ;R21 = OCR1H
```

The above code reads the low byte of the OCR1A and then the high byte, and between the two readings the content of the register remains unchanged. That is why the AVR does not employ the TEMP register while reading the OCR1A / OCR1B registers.

Examine Examples 9-29 through 9-31 to see how to generate time delay in different modes.



### Example 9-30

Assuming XTAL = 8 MHz, write a program that toggles PB5 once per millisecond.

#### Solution:

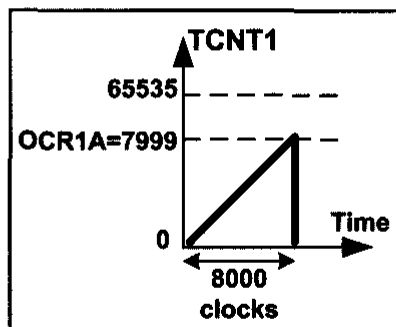
XTAL = 8 MHz means that each clock takes 0.125  $\mu$ s. Now for 1 ms delay, we need 1 ms/0.125  $\mu$ s = 8000 clocks = 0x1F40 clocks. We initialize the timer so that after 8000 clocks the OCF1A flag is raised, and then we will toggle the PB5.

```
.INCLUDE "M32DEF.INC"
    LDI    R16,HIGH(RAMEND)
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16           ;initialize the stack
    SBI    DDRB,5            ;PB5 as an output
BEGIN:SBI    PORTB,5         ;PB5 = 1
    RCALL  DELAY_1ms
    CBI    PORTB,5           ;PB5 = 0
    RCALL  DELAY_1ms
    RJMP   BEGIN

;-----Timer1 delay
DELAY_1ms:
    LDI    R20,0x00
    OUT    TCNT1H,R20        ;TEMP = 0
    OUT    TCNT1L,R20        ;TCNT1L = 0, TCNT1H = TEMP

    LDI    R20,HIGH(8000-1)
    OUT    OCR1AH,R20        ;TEMP = 0x1F
    LDI    R20,LOW(8000-1)
    OUT    OCR1AL,R20        ;OCR1AL = 0x3F, OCR1AH = TEMP

    LDI    R20,0x0
    OUT    TCCR1A,R20        ;WGM11:10 = 00
    LDI    R20,0x09
    OUT    TCCR1B,R20        ;WGM13:12 = 01, CTC mode, CS = 1
AGAIN:
    IN     R20,TIFR          ;read TIFR
    SBRS   R20,OCF1A         ;if OCF1A is set skip next instruction
    RJMP   AGAIN
    LDI    R20,1<<OCF1A
    OUT    TIFR,R20          ;clear OCF1A flag
    LDI    R19,0
    OUT    TCCR1B,R19        ;stop timer
    OUT    TCCR1A,R19
    RET
```



### Example 9-31

Rewrite Example 9-30 using the TOV1 flag.

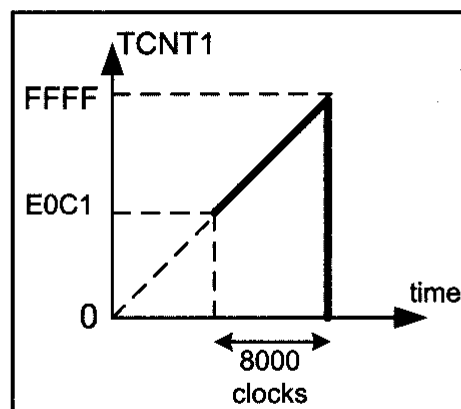
#### Solution:

To wait 1 ms we should load the TCNT1 register so that it rolls over after 8000 = 0x1F40 clocks. In Normal mode the top value is 0xFFFF = 65535.

$65535 + 1 - 8000 = 57536 = 0xE0C0$ . Thus, we should load TCNT1 with 57536, or 0xE0C0 in hex, or we can simply use  $65536 - 8000$ , as shown below:

```
.INCLUDE "M32DEF.INC"
    LDI    R16,HIGH(RAMEND)    ;initialize stack pointer
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16
    SBI    DDRB,5              ;PB5 as an output
BEGIN:SBI    PORTB,5           ;PB5 = 1
    RCALL  DELAY_1ms
    CBI    PORTB,5             ;PB5 = 0
    RCALL  DELAY_1ms
    RJMP   BEGIN

;-----Timer1 delay
DELAY_1ms:
    LDI    R20,HIGH(65536-8000)    ;R20 = high byte of 57536
    OUT    TCNT1H,R20              ;TEMP = 0xE0
    LDI    R20,LOW(65536-8000)     ;R20 = low byte of 57536
    OUT    TCNT1L,R20              ;TCNT1L = 0xC1, TCNT1H = TEMP
    LDI    R20,0x0
    OUT    TCCR1A,R20              ;WGM11:10 = 00
    LDI    R20,0x1
    OUT    TCCR1B,R20              ;WGM13:12 = 00, Normal mode, CS = 1
AGAIN:
    IN     R20,TIFR                ;read TIFR
    SBRS   R20,TOV1                ;if OCF1A is set skip next instruction
    RJMP   AGAIN
    LDI    R20,1<<TOV1
    OUT    TIFR,R20                ;clear TOV1 flag
    LDI    R19,0
    OUT    TCCR1B,R19              ;stop timer
    OUT    TCCR1A,R19
    RET
```



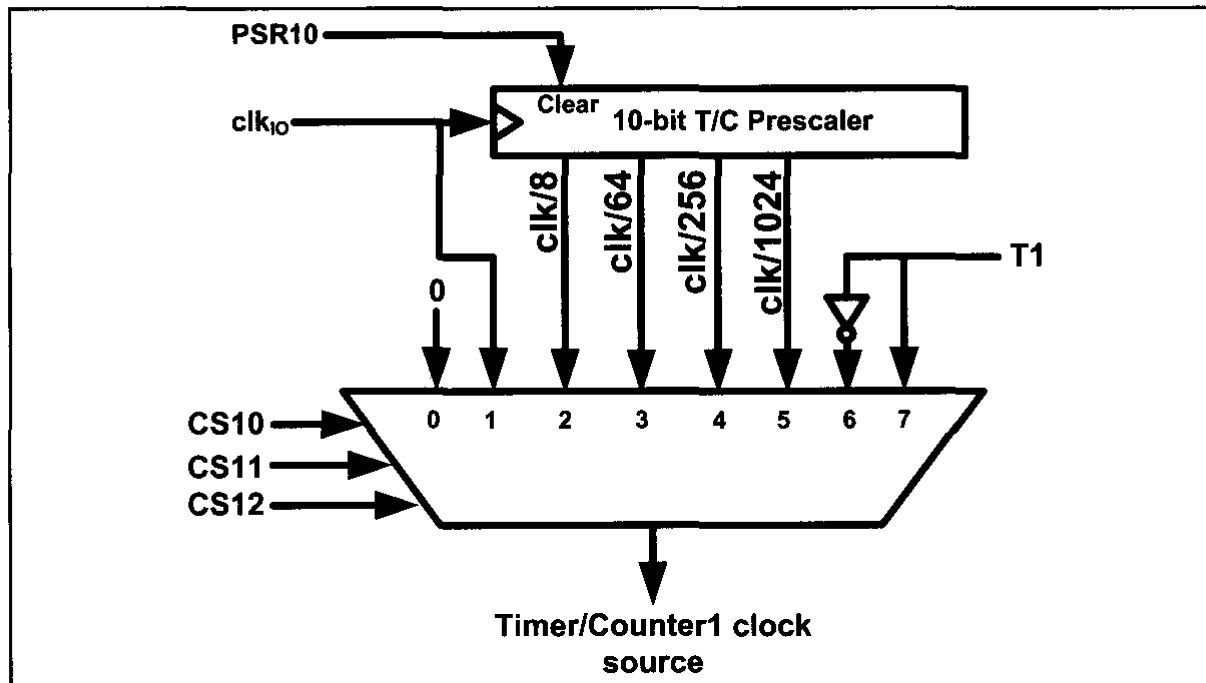


Figure 9-23. Timer/Counter 1 Prescaler

## Generating a large time delay using prescaler

As we have seen in the examples so far, the size of the time delay depends on two factors: (a) the crystal frequency, and (b) the timer's 16-bit register. Both of these factors are beyond the control of the AVR programmer. We can use the prescaler option in the TCCR1B register to increase the delay by reducing the period. The prescaler option of TCCR1B allows us to divide the instruction clock by a factor of 8 to 1024, as was shown in Figure 9-16. The prescaler of Timer/Counter 1 is shown in Figure 9-23.

As we have seen so far, with no prescaler enabled, the crystal oscillator frequency is fed directly into Timer1. If we enable the prescaler bit in the TCCR1B register, then we can divide the instruction clock before it is fed into Timer1. The lower 3 bits of the TCCR1B register give the options of the number we can divide the clock by before it is fed to timer. As shown in Figure 9-23, this number can be 8, 64, 256, or 1024. Notice that the lowest number is 8, and the highest number is 1024. Examine Examples 9-32 and 9-33 to see how the prescaler options are programmed.

## Review Questions

1. How many timers do we have in the ATmega32?
2. True or false. Timer0 is a 16-bit timer.
3. True or false. Timer1 is a 16-bit timer.
4. True or false. The TCCR0 register is a bit-addressable register.
5. In Normal mode, when the counter rolls over it goes from \_\_\_\_ to \_\_\_\_.
6. In CTC mode, the counter rolls over when the counter reaches \_\_\_\_.
7. To get a 5-ms delay, what numbers should be loaded into TCNT1H and TCNT1L using Normal mode and the TOV1 flag? Assume that XTAL = 8 MHz.
8. To get a 20-μs delay, what number should be loaded into the TCNT0 register using Normal mode and the TOV0 flag? Assume that XTAL = 1 MHz.

### Example 9-32

An LED is connected to PC4. Assuming XTAL = 8 MHz, write a program that toggles the LED once per second.

#### Solution:

As XTAL = 8 MHz, the different outputs of the prescaler are as follows:

Scaler	Timer Clock	Timer Period	Timer Value
None	8 MHz	1/8 MHz = 0.125 $\mu$ s	1 s/0.125 $\mu$ s = 8 M
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 $\mu$ s	1 s/1 $\mu$ s = 1 M
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 $\mu$ s	1 s/8 $\mu$ s = 125,000
256	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 $\mu$ s	1 s/32 $\mu$ s = 31,250
1024	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz = 128 $\mu$ s	1 s/128 $\mu$ s = 7812.5

From the above calculation we can use only options 256 or 1024. We should use option 256 since we cannot use a decimal point.

```
.INCLUDE "M32DEF.INC"
    LDI    R16,HIGH(RAMEND)    ;initialize stack pointer
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16
    SBI    DDRC,4              ;PC4 as an output
BEGIN:SBI    PORTC,4           ;PC4 = 1
    RCALL  DELAY_1s
    CBI    PORTC,4            ;PC4 = 0
    RCALL  DELAY_1s
    RJMP   BEGIN

;----- Timer1 delay
DELAY_1s:
    LDI    R20,HIGH (31250-1)
    OUT    OCR1AH,R20         ;TEMP = $7A (since 31249 = $7A11)
    LDI    R20,LOW (31250-1)
    OUT    OCR1AL,R20         ;OCR1AL = $11 (since 31249 = $7A11)
    LDI    R20,0
    OUT    TCNT1H,R20         ;TEMP = 0x00
    OUT    TCNT1L,R20         ;TCNT1L = 0x00, TCNT1H = TEMP
    LDI    R20,0x00
    OUT    TCCR1A,R20         ;WGM11:10 = 00
    LDI    R20,0x4
    OUT    TCCR1B,R20         ;WGM13:12 = 00, Normal mode,CS = CLK/256
AGAIN:IN    R20,TIFR          ;read TIFR
    SBRS   R20,OCF1A          ;if OCF1A is set skip next instruction
    RJMP   AGAIN
    LDI    R20,1<<OCF1A
    OUT    TIFR,R20           ;clear OCF1A flag
    LDI    R19,0
    OUT    TCCR1B,R19         ;stop timer
    OUT    TCCR1A,R19
    RET
```

**Example 9-33**

Assuming XTAL = 8 MHz, write a program to generate 1 Hz frequency on PC4.

**Solution:**

With 1 Hz we have  $T = 1 / F = 1 / 1 \text{ Hz} = 1 \text{ second}$ , half of which is high and half low. Thus we need a delay of 0.5 second duration.

Since XTAL = 8 MHz, the different outputs of the prescaler are as follows:

<u>Scaler</u>	<u>Timer Clock</u>	<u>Timer Period</u>	<u>Timer Value</u>
None	8 MHz	$1/8 \text{ MHz} = 0.125 \mu\text{s}$	$0.5 \text{ s} / 0.125 \mu\text{s} = 4 \text{ M}$
8	$8 \text{ MHz} / 8 = 1 \text{ MHz}$	$1/1 \text{ MHz} = 1 \mu\text{s}$	$0.5 \text{ s} / 1 \mu\text{s} = 500 \text{ k}$
64	$8 \text{ MHz} / 64 = 125 \text{ kHz}$	$1/125 \text{ kHz} = 8 \mu\text{s}$	$0.5 \text{ s} / 8 \mu\text{s} = 62,500$
256	$8 \text{ MHz} / 256 = 31.25 \text{ kHz}$	$1/31.25 \text{ kHz} = 32 \mu\text{s}$	$0.5 \text{ s} / 32 \mu\text{s} = 15,625$
1024	$8 \text{ MHz} / 1024 = 7.8125 \text{ kHz}$	$1/7.8125 \text{ kHz} = 128 \mu\text{s}$	$0.5 \text{ s} / 128 \mu\text{s} = 3906.25$

From the above calculation we can use options 64 or 256. We choose 64 in this Example.

```
.INCLUDE "M32DEF.INC"
    LDI    R16,HIGH(RAMEND)    ;initialize stack pointer
    OUT    SPH,R16
    LDI    R16,LOW(RAMEND)
    OUT    SPL,R16
    SBI    DDRC,4              ;PC4 as an output
BEGIN:SBI    PORTC,4           ;PC4 = 1
    RCALL  DELAY_1s
    CBI    PORTC,4             ;PC4 = 0
    RCALL  DELAY_1s
    RJMP   BEGIN

;----- Timer1 delay
DELAY_1s:
    LDI    R20,HIGH (62500-1)
    OUT    OCR1AH,R20          ;TEMP = $F4 (since 62499 = $F423)
    LDI    R20,LOW (62500-1)
    OUT    OCR1AL,R20          ;OCR1AL = $23 (since 62499 = $F423)
    LDI    R20,0x00
    OUT    TCNT1H,R20          ;TEMP = 0x00
    OUT    TCNT1L,R20          ;TCNT1L = 0x00, TCNT1H = TEMP
    LDI    R20,0x00
    OUT    TCCR1A,R20          ;WGM11:10 = 00
    LDI    R20,0x3
    OUT    TCCR1B,R20          ;WGM13:12 = 00, Normal mode, CS = CLK/64
AGAIN:IN    R20,TIFR           ;read TIFR
    SBRS   R20,OCF1A           ;if OCF1A is set skip next instruction
    RJMP   AGAIN
    LDI    R20,1<<OCF1A
    OUT    TIFR,R20           ;clear OCF1A flag
    LDI    R19,0
    OUT    TCCR1B,R19          ;stop timer
    OUT    TCCR1A,R19
    RET
```

## SECTION 9.2: COUNTER PROGRAMMING

In the previous section, we used the timers of the AVR to generate time delays. The AVR timer can also be used to count, detect, and measure the time of events happening outside the AVR. The use of the timer as an event counter is covered in this section. When the timer is used as a timer, the AVR's crystal is used as the source of the frequency. When it is used as a counter, however, it is a pulse outside the AVR that increments the TCNTx register. Notice that, in counter mode, registers such as TCCR, OCR0, and TCNT are the same as for the timer discussed in the previous section; they even have the same names.

### CS00, CS01, and CS02 bits in the TCCR0 register

Recall from the previous section that the CS bits (clock selector) in the TCCR0 register decide the source of the clock for the timer. If CS02:00 is between 1 and 5, the timer gets pulses from the crystal oscillator. In contrast, when CS02:00 is 6 or 7, the timer is used as a counter and gets its pulses from a source outside the AVR chip. See Figure 9-24. Therefore, when CS02:00 is 6 or 7, the TCNT0 counter counts up as pulses are fed from pin T0 (Timer/Counter 0 External Clock input). In ATmega32/ATmega16, T0 is the alternative function of PORTB.0. In the

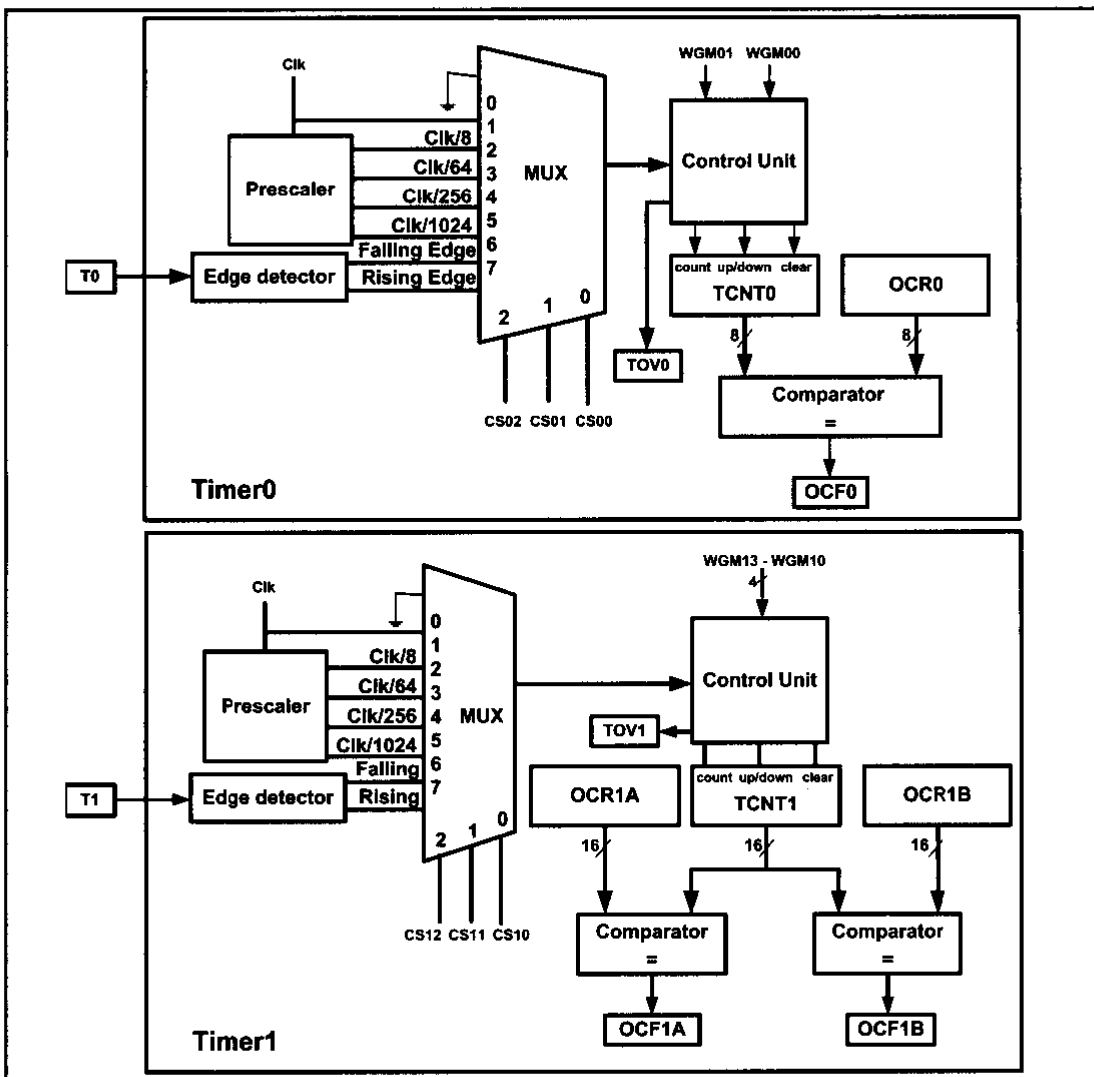


Figure 9-24. Timer/Counters 0 and 1 Prescalers

### Example 9-34

Find the value for TCCR0 if we want to program Timer0 as a Normal mode counter. Use an external clock for the clock source and increment on the positive edge.

#### Solution:

TCCR0 = 0000 0111 Normal, external clock source, no prescaler

case of Timer0, when CS02:00 is 6 or 7, pin T0 provides the clock pulse and the counter counts up after each clock pulse coming from that pin. Similarly, for Timer1, when CS12:10 is 6 or 7, the clock pulse coming in from pin T1 (Timer/Counter 1 External Clock input) makes the TCNT1 counter count up. When CS12:10 is 6, the counter counts up on the negative (falling) edge. When CS12:10 is 7, the counter counts up on the positive (rising) edge. In ATmega32/ATmega16, T1 is the alternative function of PORTB.1. See Example 9-34.

In Example 9-35, we are using Timer0 as an event counter that counts up as clock pulses are fed into PB0. These clock pulses could represent the number of people passing through an entrance, or of wheel rotations, or any other event that can be converted to pulses.

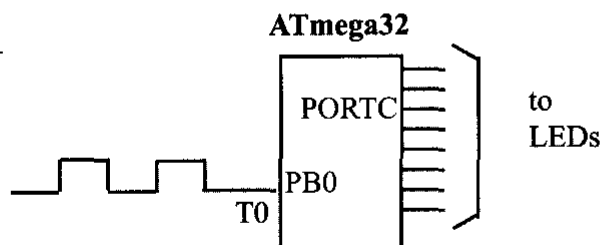
### Example 9-35

Assuming that a 1 Hz clock pulse is fed into pin T0 (PB0), write a program for Counter0 in normal mode to count the pulses on falling edge and display the state of the TCNT0 count on PORTC.

#### Solution:

```
.INCLUDE "M32DEF.INC"
CBI   DDRB,0           ;make T0 (PB0) input
LDI   R20,0xFF
OUT   DDRC,R20         ;make PORTC output
LDI   R20,0x06
OUT   TCCR0,R20        ;counter, falling edge
AGAIN:
IN    R20,TCNT0
OUT   PORTC,R20        ;PORTC = TCNT0
IN    R16,TIFR
SBRS  R16,TOV0         ;monitor TOV0 flag
RJMP  AGAIN            ;keep doing if Timer0 flag is low
LDI   R16,1<<TOV0
OUT   TIFR,R16         ;clear TOV0 flag
RJMP  AGAIN            ;keep doing it
```

PORTC is connected to 8 LEDs  
and input T0 (PB0) to 1 Hz pulse.



In Example 9-35, the TCNT0 data was displayed in binary. In Example 9-36, the TCNT0 register is extended to a 16-bit counter using the TOV0 flag. See Examples 9-37 and 9-38.

As another example of the application of the counter, we can feed an external square wave of 60 Hz frequency into the timer. The program will generate the second, the minute, and the hour out of this input frequency and display the result on an LCD. This will be a nice looking digital clock, although not a very accurate one.

Before we finish this section, we need to state an important point. You might think monitoring the TOV and OCR flags is a waste of the microcontroller's time. You are right. There is a solution to this: the use of interrupts. Using interrupts enables us to do other things with the microcontroller. When a timer Interrupt flag such as TOV0 is raised it will inform us. This important and powerful feature of the AVR is discussed in Chapter 10.

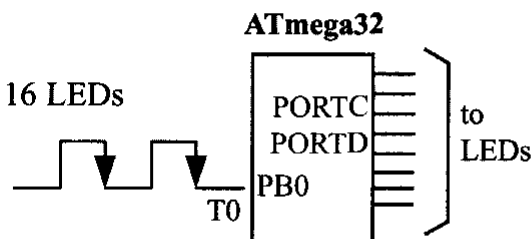
### Example 9-36

Assuming that a 1 Hz clock pulse is fed into pin T0, use the TOV0 flag to extend Timer0 to a 16-bit counter and display the counter on PORTC and PORTD.

#### Solution:

```
.INCLUDE "M32DEF.INC"
    LDI    R19,0                ;R19 = 0
    CBI    DDRB,0               ;make T0 (PB0) input
    LDI    R20,0xFF
    OUT    DDRC,R20             ;make PORTC output
    OUT    DDRD,R20             ;make PORTD output
    LDI    R20,0x06
    OUT    TCCR0,R20            ;counter, falling edge
AGAIN:
    IN     R20,TCNT0
    OUT    PORTC,R20             ;PORTC = TCNT0
    IN     R16,TIFR
    SBRS   R16,TOV0
    RJMP   AGAIN                ;keep doing it
    LDI    R16,1<<TOV0         ;clear TOV0 flag
    OUT    TIFR,R16
    INC    R19                  ;R19 = R19 + 1
    OUT    PORTD,R19            ;PORTD = R19
    RJMP   AGAIN                ;keep doing it
```

PORTC and PORTD are connected to 16 LEDs and input T0 (PB0) to 1 Hz pulse.





### Example 9-37

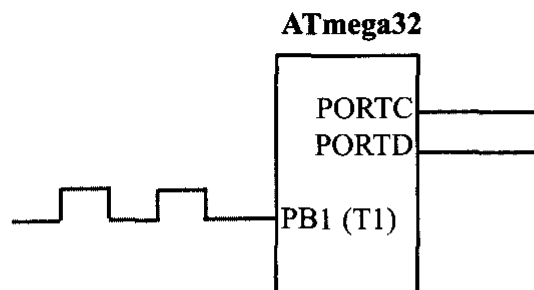
Assuming that clock pulses are fed into pin T1 (PB1), write a program for Counter1 in Normal mode to count the pulses on falling edge and display the state of the TCNT1 count on PORTC and PORTD.

#### Solution:

```
.INCLUDE "M32DEF.INC"

CBI  DDRB,1           ;make T1 (PB1) input
LDI  R20,0xFF
OUT  DDRC,R20         ;make PORTC output
OUT  DDRD,R20         ;make PORTD output
LDI  R20,0x0
OUT  TCCR1A,R20
LDI  R20,0x06
OUT  TCCR1B,R20       ;counter, falling edge

AGAIN:
IN   R20,TCNT1L       ;R20 = TCNT1L, TEMP = TCNT1H
OUT  PORTC,R20        ;PORTC = TCNT0
IN   R20,TCNT1H       ;R20 = TEMP
OUT  PORTD,R20        ;PORTD = TCNT0
IN   R16,TIFR
SBRS R16,TOV1
RJMP AGAIN            ;keep doing it
LDI  R16,1<<TOV1      ;clear TOV1 flag
OUT  TIFR, R16
RJMP AGAIN            ;keep doing it
```



### Example 9-38

Assuming that clock pulses are fed into pin T1 (PB1) and a buzzer is connected to pin PORTC.0, write a program for Counter 1 in CTC mode to sound the buzzer every 100 pulses.

#### Solution:

To sound the buzzer every 100 pulses, we set the OCR1A value to 99 (63 in hex), and then the counter counts up until it reaches OCR1A. Upon compare match, we can sound the buzzer by toggling the PORTC.0 pin.

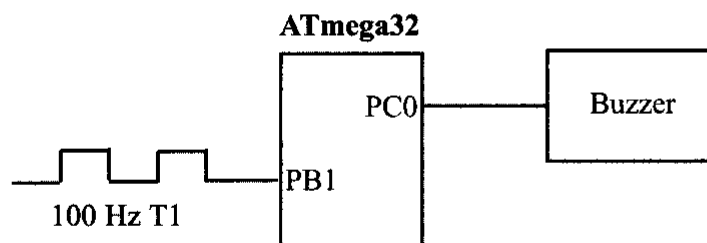
```
.INCLUDE "M32DEF.INC"

CBI   DDRB,1           ;make T1 (PB1) input
SBI   DDRC,0           ;PC0 as an output
LDI   R16,0x1
LDI   R17,0

LDI   R20,0x0
OUT   TCCR1A,R20
LDI   R20,0x0E
OUT   TCCR1B,R20       ;CTC, counter, falling edge
AGAIN:
LDI   R20,0
OUT   OCR1AH,R20       ;TEMP = 0
LDI   R20,99
OUT   OCR1AL,R20       ;ORC1L = R20, OCR1H = TEMP
L1:   IN    R20,TIFR
SBRs  R20,OCF1A
RJMP  L1               ;keep doing it
LDI   R20,1<<OCF1A    ;clear OCF1A flag
OUT   TIFR, R20

EOR   R17,R16          ;toggle D0 of R17
OUT   PORTC,R17        ;toggle PC0
RJMP  AGAIN            ;keep doing it
```

PC0 is connected to a buzzer and input T1 to a pulse.



## Review Questions

1. Which resource provides the clock pulses to AVR timers if CS02:00 = 6?
2. For Counter 0, which pin is used for the input clock?
3. To allow PB1 to be used as an input for the Timer1 clock, what must be done, and why?
4. Do we have a choice of counting up on the positive or negative edge of the clock?

## SECTION 9.3: PROGRAMMING TIMERS IN C

In Chapter 7 we showed some examples of C programming for the AVR. In this section we show C programming for the AVR timers. As we saw in the examples in Chapter 7, the general-purpose registers of the AVR are under the control of the C compiler and are not accessed directly by C statements. All of the SFRs (Special Function Registers), however, are accessible directly using C statements. As an example of accessing the SFRs directly, we saw how to access ports PORTB–PORTD in Chapter 7.

In C we can access timer registers such as TCNT0, OCR0, and TCCR0 directly using their names. See Example 9-39.

### Example 9-39

Write a C program to toggle all the bits of PORTB continuously with some delay. Use Timer0, Normal mode, and no prescaler options to generate the delay.

#### Solution:

```
#include "avr/io.h"
void T0Delay ( );
int main ( )
{
    DDRB = 0xFF;        //PORTB output port

    while (1)
    {
        PORTB = 0x55;    //repeat forever
        T0Delay ( );     //delay size unknown
        PORTB = 0xAA;    //repeat forever
        T0Delay ( );
    }
}

void T0Delay ( )
{
    TCNT0 = 0x20;        //load TCNT0
    TCCR0 = 0x01;        //Timer0, Normal mode, no prescaler
    while ((TIFR&0x1)==0); //wait for TF0 to roll over
    TCCR0 = 0;
    TIFR = 0x1;          //clear TF0
}
```

## Calculating delay length using timers

As we saw in the last two sections, the delay length depends primarily on two factors: (a) the crystal frequency, and (b) the prescaler factor. A third factor in the delay size is the C compiler because various C compilers generate different hex code sizes, and the amount of overhead due to the instructions varies by compiler. Study Examples 9-40 through 9-42 and verify them using an oscilloscope.

### Example 9-40

Write a C program to toggle only the PORTB.4 bit continuously every 70  $\mu$ s. Use Timer0, Normal mode, and 1:8 prescaler to create the delay. Assume XTAL = 8 MHz.

#### Solution:

XTAL = 8MHz  $\rightarrow$   $T_{\text{machine cycle}} = 1/8 \text{ MHz}$

Prescaler = 1:8  $\rightarrow$   $T_{\text{clock}} = 8 \times 1/8 \text{ MHz} = 1 \mu\text{s}$

$70 \mu\text{s} / 1 \mu\text{s} = 70 \text{ clocks} \rightarrow 1 + 0xFF - 70 = 0x100 - 0x46 = 0xBA = 186$

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

void T0Delay ( );

int main ( )
{
    DDRB = 0xFF;          //PORTB output port

    while (1)
    {
        T0Delay ( );      //Timer0, Normal mode
        PORTB = PORTB ^ 0x10; //toggle PORTB.4
    }
}

void T0Delay ( )
{
    TCNT0 = 186;          //load TCNT0
    TCCR0 = 0x02;         //Timer0, Normal mode, 1:8 prescaler
    while ((TIFR & (1<<TOV0)) == 0); //wait for TOV0 to roll over

    TCCR0 = 0;            //turn off Timer0
    TIFR = 0x1;           //clear TOV0
}
```

**Example 9-41**

Write a C program to toggle only the PORTB.4 bit continuously every 2 ms. Use Timer1, Normal mode, and no prescaler to create the delay. Assume XTAL = 8 MHz.

**Solution:**

XTAL = 8 MHz  $\rightarrow T_{\text{machine cycle}} = 1/8 \text{ MHz} = 0.125 \mu\text{s}$

Prescaler = 1:1  $\rightarrow T_{\text{clock}} = 0.125 \mu\text{s}$

2 ms/0.125  $\mu\text{s} = 16,000 \text{ clocks} = 0x3E80 \text{ clocks}$

$1 + 0xFFFF - 0x3E80 = 0xC180$

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

void T1Delay ( );

int main ( )
{
    DDRB = 0xFF;          //PORTB output port

    while (1)
    {
        PORTB = PORTB ^ (1<<PB4); //toggle PB4
        T1Delay ( );             //delay size unknown
    }
}

void T1Delay ( )
{
    TCNT1H = 0xC1;        //TEMP = 0xC1
    TCNT1L = 0x80;

    TCCR1A = 0x00;        //Normal mode
    TCCR1B = 0x01;        //Normal mode, no prescaler

    while ((TIFR & (0x1<<TOV1)) == 0);    //wait for TOV1 to roll over

    TCCR1B = 0;
    TIFR = 0x1<<TOV1;    //clear TOV1
}
```

**Example 9-42 (C version of Example 9-32)**

Write a C program to toggle only the PORTB.4 bit continuously every second. Use Timer1, Normal mode, and 1:256 prescaler to create the delay. Assume XTAL = 8 MHz.

**Solution:**

$$\text{XTAL} = 8 \text{ MHz} \rightarrow T_{\text{machine cycle}} = 1/8 \text{ MHz} = 0.125 \mu\text{s} = T_{\text{clock}}$$

$$\text{Prescaler} = 1:256 \rightarrow T_{\text{clock}} = 256 \times 0.125 \mu\text{s} = 32 \mu\text{s}$$

$$1 \text{ s}/32 \mu\text{s} = 31,250 \text{ clocks} = 0x7A12 \text{ clocks} \rightarrow 1 + 0xFFFF - 0x7A12 = 0x85EE$$

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

void T1Delay ( );

int main ( )
{
    DDRB = 0xFF;        //PORTB output port

    while (1)
    {
        PORTB = PORTB ^ (1<<PB4); //toggle PB4
        T1Delay ( );           //delay size unknown
    }
}

void T1Delay ( )
{
    TCNT1H = 0x85;        //TEMP = 0x85
    TCNT1L = 0xEE;

    TCCR1A = 0x00;        //Normal mode
    TCCR1B = 0x04;        //Normal mode, 1:256 prescaler

    while ((TIFR & (0x1<<TOV1)) == 0); //wait for TF0 to roll over

    TCCR1B = 0;
    TIFR = 0x1<<TOV1;     //clear TOV1
}
```

**C programming of Timers 0 and 1 as counters**

In Section 9.2 we showed how to use Timers 0 and 1 as event counters. Timers can be used as counters if we provide pulses from outside the chip instead of using the frequency of the crystal oscillator as the clock source. By feeding pulses to the T0 (PB0) and T1 (PB1) pins, we use Timer0 and Timer1 as Counter 0 and Counter 1, respectively. Study Examples 9-43 and 9-44 to see how Timers 0 and 1 are programmed as counters using C language.

### Example 9-43 (C version of Example 9-36)

Assuming that a 1 Hz clock pulse is fed into pin T0, use the TOV0 flag to extend Timer0 to a 16-bit counter and display the counter on PORTC and PORTD.

#### Solution:

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

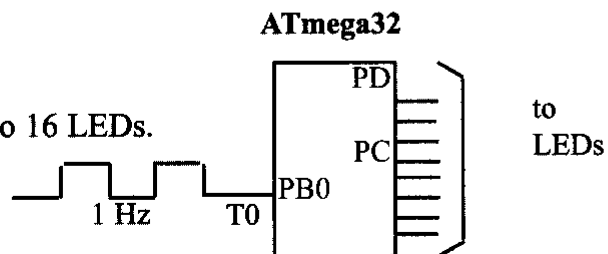
int main ( )
{
    PORTB = 0x01;           //activate pull-up of PB0
    DDRC = 0xFF;            //PORTC as output
    DDRD = 0xFF;            //PORTD as output

    TCCR0 = 0x06;           //output clock source
    TCNT0 = 0x00;

    while (1)
    {
        do
        {
            PORTC = TCNT0;
        } while((TIFR & (0x1 << TOV0)) == 0); //wait for TOV0 to roll over

        TIFR = 0x1 << TOV0; //clear TOV0
        PORTD ++;            //increment PORTD
    }
}
```

PORTC and PORTD are connected to 16 LEDs.  
T0 (PB0) is connected to a  
1-Hz external clock.



**Example 9-44 (C version of Example 9-37)**

Assume that a 1-Hz external clock is being fed into pin T1 (PB1). Write a C program for Counter1 in rising edge mode to count the pulses and display the TCNT1H and TCNT1L registers on PORTD and PORTC, respectively.

**Solution:**

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

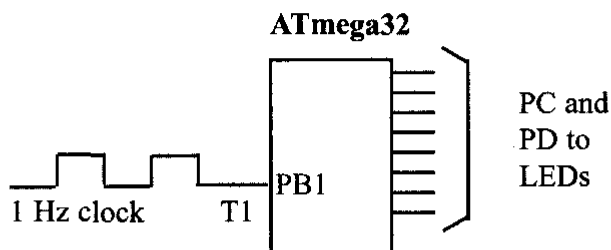
int main ( )
{
    PORTB = 0x01;           //activate pull-up of PB0
    DDRC = 0xFF;            //PORTC as output
    DDRD = 0xFF;            //PORTD as output

    TCCR1A = 0x00;          //output clock source
    TCCR1B = 0x06;          //output clock source

    TCNT1H = 0x00;          //set count to 0
    TCNT1L = 0x00;          //set count to 0

    while (1)               //repeat forever
    {
        do
        {
            PORTC = TCNT1L;
            PORTD = TCNT1H;    //place value on pins
        } while((TIFR & (0x1 << TOV1)) == 0); //wait for TOV1

        TIFR = 0x1 << TOV1;    //clear TOV1
    }
}
```





## SUMMARY

The AVR has one to six timers/counters depending on the family member. When used as timers, they can generate time delays. When used as counters, they can serve as event counters.

Some of the AVR timers are 8-bit and some are 16-bit. The 8-bit timers are accessed as TCNTn (like TCNT0 for Timer0), whereas 16-bit timers are accessed as two 8-bit registers (TCNTnH, TCNTnL).

Each timer has its own TCCR (Timer/Counter Control Register) register, allowing us to choose various operational modes. Among the modes are the prescaler and timer/counter options. When the timer is used as a timer, the AVR crystal is used as the source of the frequency; however, when it is used as a counter, it is a pulse outside of the AVR that increments the TCNT register.

This chapter showed how to program the timers/counters to generate delays and count events using Normal and CTC modes.

## PROBLEMS

### SECTION 9.1: PROGRAMMING TIMERS 0, 1, AND 2

1. How many timers are in the ATmega32?
2. Timer0 of the ATmega32 is \_\_\_\_-bit, accessed as \_\_\_\_.
3. Timer1 of the ATmega32 is \_\_\_\_-bit, accessed as \_\_\_\_ and \_\_\_\_.
4. Timer0 supports the highest prescaler value of \_\_\_\_.
5. Timer1 supports the highest prescaler value of \_\_\_\_.
6. The TCCR0 register is a(n) \_\_\_\_-bit register.
7. What is the job of the TCCR0 register?
8. True or false. TCCR0 is a bit-addressable register.
9. True or false. TIFR is a bit-addressable register.
10. Find the TCCR0 value for Normal mode, no prescaler, with the clock coming from the AVR's crystal.
11. Find the frequency and period used by the timer if the crystal attached to the AVR has the following values:
  - (a) XTAL = 8 MHz
  - (b) XTAL = 16 MHz
  - (c) XTAL = 1 MHz
  - (d) XTAL = 10 MHz
12. Which register holds the TOV0 (timer overflow flag) and TOV1 bits?
13. Indicate the rollover value (in hex and decimal) of the timer for each of the following cases:
  - (a) Timer0 and Normal mode
  - (b) Timer1 and Normal mode
14. Indicate when the TOVx flag is raised for each of the following cases:
  - (a) Timer0 and Normal mode
  - (b) Timer1 and Normal mode
15. True or false. Both Timer0 and Timer1 have their own timer overflow flags.
16. True or false. Both Timer0 and Timer1 have their own timer compare match flags.

17. Assume that XTAL = 8 MHz. Find the TCNT0 value needed to generate a time delay of 20  $\mu$ s. Use Normal mode, no prescaler mode.
18. Assume that XTAL = 8 MHz. Find the TCNT0 value needed to generate a time delay of 5 ms. Use Normal mode, and the largest prescaler possible.
19. Assume that XTAL = 1 MHz. Find the TCNT1H,TCNT1L value needed to generate a time delay of 2.5 ms. Use Normal mode, no prescaler mode.
20. Assume that XTAL = 1 MHz. Find the OCR0 value needed to generate a time delay of 0.2 ms. Use CTC mode, no prescaler mode.
21. Assume that XTAL = 1 MHz. Find the OCR1H,OCR1L value needed to generate a time delay of 2 ms. Use CTC mode, and no prescaler mode.
22. Assuming that XTAL = 8 MHz, and we are generating a square wave on pin PB7, find the lowest square wave frequency that we can generate using Timer1 in Normal mode.
23. Assuming that XTAL = 8 MHz, and we are generating a square wave on pin PB2, find the highest square wave frequency that we can generate using Timer1 in Normal mode.
24. Repeat Problems 22 and 23 for Timer0.
25. Assuming that TCNT0 = \$F1, indicate which states Timer0 goes through until TOV0 is raised. How many states is that?
26. Program Timer0 to generate a square wave of 1 kHz. Assume that XTAL = 8 MHz.
27. Program Timer1 to generate a square wave of 3 kHz. Assume that XTAL = 8 MHz.
28. State the differences between Timer0 and Timer1.
29. Find the value (in hex) loaded into R16 in each of the following:
 

(a) LDI R16,-12	(b) LDI R16,-22
(c) LDI R16,-34	(d) LDI R16,-92
(e) LDI R16,-120	(f) LDI R16,-104

## SECTION 9.2: COUNTER PROGRAMMING

30. To use a timer as an event counter we must set the \_\_\_\_\_ bits in the TCCR register to \_\_\_\_\_.
31. Can we use both Timer0 and Timer1 as event counters?
32. For Counter 0, which pin is used for the input clock?
33. For Counter 1, which pin is used for the input clock?
34. Program Timer1 to be an event counter. Use Normal mode, and display the binary count on PORTC and PORTD continuously. Set the initial count to 20,000.
35. Program Timer0 to be an event counter. Use Normal mode and display the binary count on PORTC continuously. Set the initial count to 20.

## SECTION 9.3: PROGRAMMING TIMERS IN C

36. Program Timer0 in C to generate a square wave of 1 kHz. Assume that XTAL = 1 MHz.

37. Program Timer1 in C to generate a square wave of 1 kHz. Assume that XTAL = 8 MHz.
38. Program Timer0 in C to generate a square wave of 3 kHz. Assume that XTAL = 16 MHz.
39. Program Timer1 in C to generate a square wave of 3 kHz. Assume that XTAL = 10 MHz.
40. Program Timer1 in C to be an event counter. Use Normal mode and display the binary count on PORTB and PORTD continuously. Set the initial count to 20,000.
41. Program Timer0 in C to be an event counter. Use Normal mode and display the binary count on PORTD continuously. Set the initial count to 20.

## ANSWERS TO REVIEW QUESTIONS

### SECTION 9.1: PROGRAMMING TIMERS 0, 1, AND 2

1. 3
2. False
3. True
4. False
5. Max (\$FFFF for 16-bit timers and \$FF for 8-bit timers), 0000
6. OCR1A
7.  $\$10000 - (5000 \times 8) = 25536 = 63C0$ , TCNT1H = 0x64 and TCNT1L = 0xC0
8. XTAL = 1 MHz  $\rightarrow T_{\text{machine cycle}} = 1/1 \text{ M} = 1 \mu\text{s} \rightarrow 20 \mu\text{s} / 1 \mu\text{s} = 20$   
 $-20 = \$100 - 20 = 256 - 20 = 236 = 0xEC$

### SECTION 9.2: COUNTER PROGRAMMING

1. External clock (falling edge)
2. PORTB.0 (T0)
3. DDRB.0 must be cleared to turn the output circuit off and use the pin as input.
4. Yes

---

# 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
- >> 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 10.4, we cover interrupt priority. In Section 10.5, 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 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 *polling*, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller. The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it. The polling method cannot assign priority because it checks all devices in a round-robin fashion. More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This also is not possible with the polling method. The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service. So interrupts are used to avoid tying down the microcontroller. For example, in discussing timers in Chapter 9 we used the bit test instruction "SBRSC R20, TOV0" and waited until the timer rolled over, and while we were waiting we could not do anything else. That is a waste of microcontroller time that could have been used to perform some useful tasks. In the case of the timer, if we use the interrupt method, the microcontroller can go about doing other tasks, and when the TOV0 flag is raised, the timer will interrupt the microcontroller in whatever it is doing.

### Interrupt service routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. Generally, in most microprocessors, for every interrupt there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the *interrupt vector table*, as shown in Table 10-1.

## Steps in executing an interrupt

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

1. It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.
2. It jumps to a fixed location in memory called the *interrupt vector table*. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
4. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

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.

**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

## 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. 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. 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), 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. 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 10-1. In the next section we will see how this works in the context of some examples.

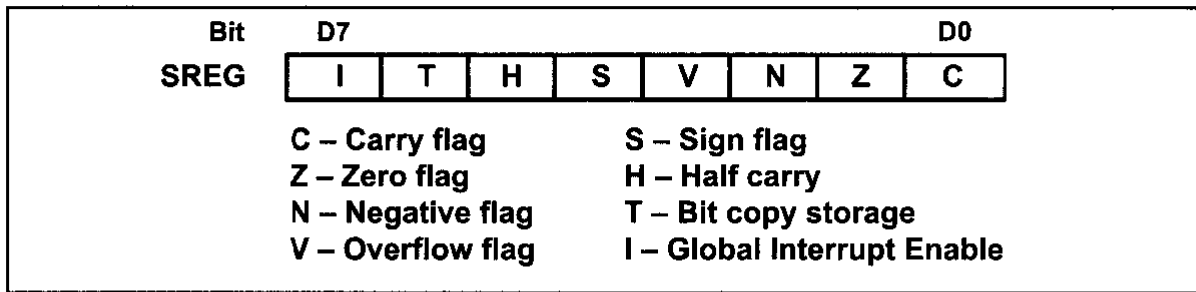
```
                .ORG 0      ;wake-up ROM reset location
                JMP  MAIN   ;bypass interrupt vector table

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

**Figure 10-1. 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 10-2 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.



**Figure 10-2. 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 10-3 shows that the TIMSK register has 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 Timer2 compare match interrupt, and (b) disable (mask) the Timer0 overflow interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

#### Solution:

```
(a)  LDI R20, (1<<TOIE0) | (1<<OCIE2) ;TOIE0 = 1, OCIE2 = 1
      OUT TIMSK,R20 ;enable Timer0 overflow and Timer2 compare match
      SEI ;allow interrupts to come in
```

```
(b)  IN R20,TIMSK ;R20 = TIMSK
      ANDI R20,0xFF^(1<<TOIE0) ;TOIE0 = 0
      OUT TIMSK,R20 ;mask (disable) Timer0 interrupt
```

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

```
IN R20,TIMSK ;R20 = TIMSK
CBR R20,1<<TOIE0 ;TOIE0 = 0
OUT TIMSK,R20 ;mask (disable) Timer0 interrupt

(c)  CLI ;mask all interrupts globally
```

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



## 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 TIMSK register.
3. Upon power-on reset of the ATmega32, 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?

D7				D0			
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
<b>TOIE0</b> Timer0 overflow interrupt enable = 0 Disables Timer0 overflow interrupt = 1 Enables Timer0 overflow interrupt							
<b>OCIE0</b> Timer0 output compare match interrupt enable = 0 Disables Timer0 compare match interrupt = 1 Enables Timer0 compare match interrupt							
<b>TOIE1</b> Timer1 overflow interrupt enable = 0 Disables Timer1 overflow interrupt = 1 Enables Timer1 overflow interrupt							
<b>OCIE1B</b> Timer1 output compare B match interrupt enable = 0 Disables Timer1 compare B match interrupt = 1 Enables Timer1 compare B match interrupt							
<b>OCIE1A</b> Timer1 output compare A match interrupt enable = 0 Disables Timer1 compare A match interrupt = 1 Enables Timer1 compare A match interrupt							
<b>TICIE1</b> Timer1 input capture interrupt enable = 0 Disables Timer1 input capture interrupt = 1 Enables Timer1 input capture interrupt							
<b>TOIE2</b> Timer2 overflow interrupt enable = 0 Disables Timer2 overflow interrupt = 1 Enables Timer2 overflow interrupt							
<b>OCIE2</b> Timer2 output compare match interrupt enable = 0 Disables Timer2 compare match interrupt = 1 Enables Timer2 compare 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 another interrupt to come in.							

**Figure 10-3. TIMSK (Timer Interrupt Mask) Register**

## 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 “SBRs R20, TOV0”. In polling TOV0, we have to wait until TOV0 is raised. The problem with this method is that the microcontroller is tied down waiting for TOV0 to be raised, and cannot do anything else. Using interrupts avoids tying down the controller. If the timer interrupt in the interrupt register is enabled, TOV0 is raised whenever the timer rolls over and the microcontroller jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other things until it is notified that the timer has rolled over. To use an interrupt in place of polling, first we must enable the interrupt because all the interrupts are masked upon reset. The TOIEx bit enables the interrupt for a given timer. TOIEx bits are held by the TIMSK register as shown in Table 10-2. See Figure 10-4 and Program 10-1.

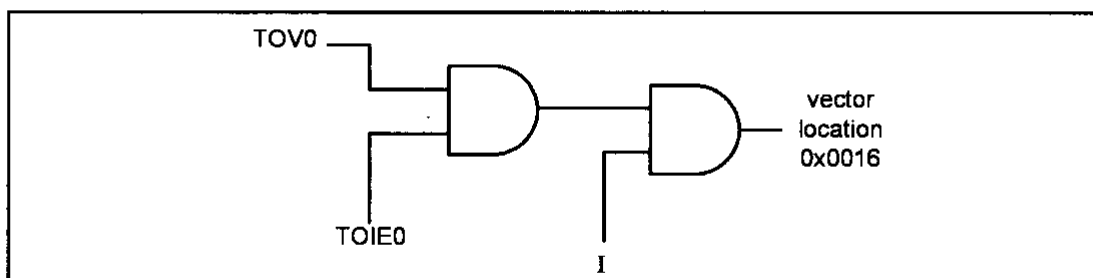
**Table 10-2: Timer Interrupt Flag Bits and Associated Registers**

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

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)
```



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

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

**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.

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

```

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

```

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 PB7 respectively, while data is being transferred from PORTC to PORTD.

```

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

```

```

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

```

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 \$0016, and the Timer1 overflow address of \$0012 in the interrupt vector table because they were fixed at the time of the ATmega32 design.

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.

```

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

```

```

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

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

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

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

```

## Compare match timer flag and interrupt

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.

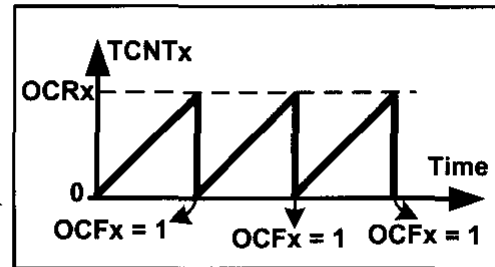


Figure 10-5. CTC Mode

### Example 10-3

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

#### Solution:

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

```
.INCLUDE "M32DEF.INC"
.ORG 0x0 ;location for reset
    JMP MAIN
.ORG 0x14 ;ISR location for Timer0 compare match
    JMP T0_CM_ISR
;main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI R20,LOW(RAMEND)
      OUT SPL,R20 ;set up stack
      SBI DDRB,5 ;PB5 as an output
      LDI R20,(1<<OCIE0)
      OUT TIMSK,R20 ;enable Timer0 compare match interrupt
      SEI ;set I (enable interrupts globally)
      LDI R20,39
      OUT OCR0,R20 ;load Timer0 with 39
      LDI R20,0x09
      OUT TCCR0,R20 ;start Timer0, CTC mode, int clk, no prescaler
      LDI R20,0x00
      OUT DDRC,R20 ;make PORTC input
      LDI R20,0xFF
      OUT DDRD,R20 ;make PORTD output
;----- Infinite loop
HERE: IN R20,PINC ;read from PORTC
      OUT PORTD,R20 ;and send it to PORTD
      JMP HERE ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0 (it is executed every 40  $\mu$ s)
T0_CM_ISR:
      IN R16,PORTB ;read PORTB
      LDI R17,0x20 ;00100000 for toggling PB5
      EOR R16,R17
      OUT PORTB,R16 ;toggle PB5
      RETI ;return from interrupt
```

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 10-5 and Examples 10-3 and 10-4. Notice that the AVR chip clears the OCF flag upon jumping to the interrupt vector table.

#### Example 10-4

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

#### Solution:

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

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP     MAIN
.ORG 0x14    ;location for Timer1 compare match
    JMP     T1_CM_ISR
;-----main program for initialization and keeping CPU busy
MAIN: LDI    R20,HIGH(RAMEND)
    OUT     SPH,R20
    LDI    R20,LOW(RAMEND)
    OUT     SPL,R20    ;set up stack
    SBI     DDRB,5      ;PB5 as an output
    LDI    R20,(1<<OCIE1A)
    OUT     TIMSK,R20   ;enable Timer1 compare match interrupt
    SEI                      ;set I (enable interrupts globally)
    LDI    R20,0x00
    OUT     TCCR1A,R20
    LDI    R20,0xD
    OUT     TCCR1B,R20   ;prescaler 1:1024, CTC mode
    LDI    R20,HIGH(7811) ;the high byte
    OUT     OCR1AH,R20   ;Temp = 0x1E (high byte of 7811)
    LDI    R20,LOW(7811) ;the low byte
    OUT     OCR1AL,R20   ;OCR1A = 7811
    LDI    R20,0x00
    OUT     DDRC,R20     ;make PORTC input
    LDI    R20,0xFF
    OUT     DDRD,R20     ;make PORTD output
;----- Infinite loop
HERE: IN     R20,PINC     ;read from PORTC
    OUT     PORTD,R20    ;PORTD = R20
    JMP     HERE        ;keeping CPU busy waiting for interrupt
;---ISR for Timer1 (It comes here after elapse of 1 second time)
T1_CM_ISR:
    IN     R16,PORTB
    LDI    R17,0x20      ;00100000 for toggling PB5
    EOR    R16,R17
    OUT     PORTB,R16    ;toggle PB5
    RETI                ;return from interrupt
```



## 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    TIFR, 1<<TOV0    ;clear TOV0 flag
RETI
```
6. Assume that Timer0 is programmed in CTC mode, OCR0 = 0x21, 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 interrupt interrupts varies in different AVR. The ATmega32 has three external hardware interrupts: pins PD2 (PORTD.2), PD3 (PORTD.3), and PB2 (PORTB.2), designated as INT0, INT1, and INT2, respectively. Upon activation of these pins, the AVR is interrupted in whatever it is doing and jumps to the vector table to perform the interrupt service routine. In this section we study these three external hardware interrupts of the AVR with some examples in Assembly language.

### External interrupts INT0, INT1, and INT2

There are three external hardware interrupts in the ATmega32: INT0, INT1, and INT2. They are located on pins PD2, PD3, and PB2, respectively. As we saw in Table 10-1, the interrupt vector table locations \$2, \$4, and \$6 are set aside for INT0, INT1, and INT2, respectively. The hardware interrupts must be enabled before they can take effect. This is done using the INTx bit located in the GICR register. See Figure 10-6. For example, the following instructions enable INT0:

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

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.

D7				D0			
INT1	INT0	INT2	-	-	-	IVSEL	IVCE

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

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

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

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

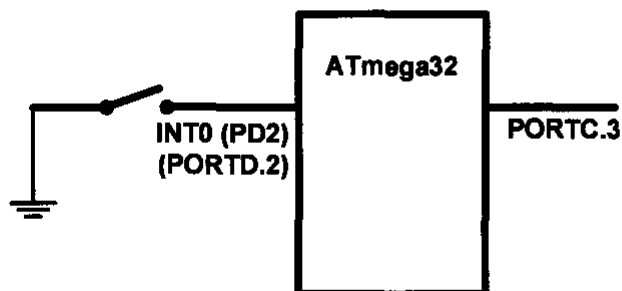
**Figure 10-6. GICR (General Interrupt Control Register) Register**

### Example 10-5

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




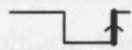
#### Solution:

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



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

## Edge-triggered vs. level-triggered interrupts

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

As stated before, upon reset INT0 and INT1 are low-level-triggered interrupts. The bits of the MCUCR register indicate the trigger options of INT0 and INT1, as shown in Figure 10-7.

D7				D0			
JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF

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


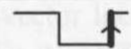
ISC2		Description
0		The falling edge of INT2 generates an interrupt request.
1		The rising edge of INT2 generates an interrupt request.

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

The ISC2 bit of the MCUCSR register defines whether INT2 activates in the falling edge or the rising edge (see Figure 10-8). Upon reset ISC2 is 0, meaning that the external hardware interrupt of INT2 is falling edge triggered. See Examples 10-6 and 10-7.

#### Example 10-6

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

#### Solution:

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

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

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

#### Example 10-7

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

#### Solution:

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

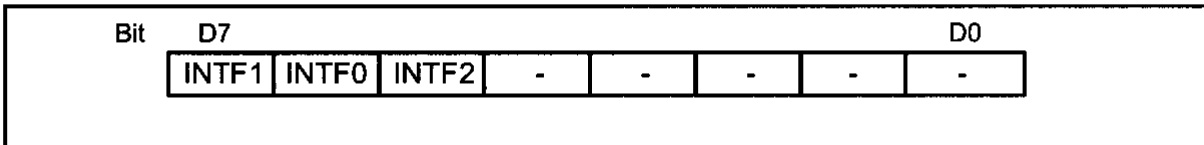
In Example 10-7, notice that the only difference between it and the program in Example 10-5 is in the following instructions:

```
LDI    R20,0x2                ;make INT0 falling edge triggered
OUT     MCUCR,R20
```

which makes INT0 an edge-triggered interrupt. When the falling edge of the signal is applied to pin INT0, PORTC.3 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, PORTC.3 toggles. But in this example, to turn on PORTC.3 again, the INT0 pulse must be brought back high and then low to create a falling edge to activate the interrupt.

## 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. 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,



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

the flag remains set. The flag can be cleared by writing a one to it. For example, the INTF1 flag can be cleared using the following instructions:

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

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.

## Review Questions

1. True or false. Upon reset, the external hardware interrupts INT0–INT2 are edge triggered.

2. For ATmega32, what pins are assigned to INT0–INT2?
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 INT2 interrupt is falling 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: 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 2 is 6; thus, external interrupt 0 has a higher priority, and if both of these interrupts are activated at the same time, external interrupt 0 is served first.

### Interrupt inside an interrupt

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

### Context saving in task switching

In multitasking systems, 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  $\mu$ s

While writing a program for a multitasking system, we should manage the resources carefully so that the tasks do not conflict with each other. For example, consider a system that should perform the following tasks: (1) increasing the con-

tents of PORTC continuously, and (2) increasing the content of PORTD once every 5  $\mu$ s. 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
HERE: OUT PORTC,R20 ;PORTC = R20
INC R20
JMP HERE ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0
T0_CM_ISR:
IN R20,PIND
INC R20
OUT PORTD,R20 ;PORTD = R20
RETI ;return from interrupt
```

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.

```
;Program 10-5
.INCLUDE "M32DEF.INC"
.ORG 0x0 ;location for reset
JMP MAIN
```

```

.ORG 0x14 ;location for Timer0 compare match
JMP T0_CM_ISR
;-----main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI R20,LOW(RAMEND)
      OUT SPL,R20 ;set up stack
      SBI DDRB,5 ;PB5 as an output
      LDI R20,(1<<OCIE0)
      OUT TIMSK,R20 ;enable Timer0 compare match interrupt
      SEI ;set I (enable interrupts globally)
      LDI R20,160
      OUT OCR0,R20 ;load Timer0 with 160
      LDI R20,0x09
      OUT TCCR0,R20 ;start timer,CTC mode,int clk,no prescaler
      LDI R20,0xFF
      OUT DDRC,R20 ;make PORTC output
      OUT DDRD,R20 ;make PORTD output
      LDI R20, 0
HERE: OUT PORTC,R20 ;PORTC = R20
      INC R20
      JMP HERE ;keeping CPU busy waiting for int.
;-----ISR for Timer0
T0_CM_ISR:
      IN R21,PIND
      INC R21
      OUT PORTD,R21 ;toggle PB5
      RETI ;return from interrupt

```

(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
.INCLUDE "M32DEF.INC"
.ORG 0x0 ;location for reset
JMP MAIN
.ORG 0x14 ;location for Timer0 compare match
JMP T0_CM_ISR
;main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI R20,HIGH(RAMEND)
      OUT SPH,R20
      LDI R20,LOW(RAMEND)
      OUT SPL,R20 ;set up stack
      SBI DDRB,5 ;PB5 as an output
      LDI R20,(1<<OCIE0)
      OUT TIMSK,R20 ;enable Timer0 compare match interrupt
      SEI ;set I (enable interrupts globally)
      LDI R20,160
      OUT OCR0,R20 ;load Timer0 with 160
      LDI R20,0x09

```



```

        OUT    TCCR0,R20    ;CTC mode, int clk, no prescaler
        LDI    R20,0xFF
        OUT    DDRC,R20    ;make PORTC output
        OUT    DDRD,R20    ;make PORTD output
        LDI    R20, 0
HERE:    OUT    PORTC,R20    ;PORTC = R20
        INC    R20
        JMP    HERE        ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0
T0_CM_ISR:
        PUSH   R20          ;save R20 on stack
        IN     R20,PIND
        INC    R20
        OUT    PORTD,R20    ;toggle PB5
        POP    R20          ;restore value for R20
        RETI               ;return from interrupt

```

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 10-10.

### 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.

<pre> Sample_ISR:         PUSH   R20         IN     R20,SREG         PUSH   R20         ...         POP    R20         OUT    SREG,R20         POP    R20         RETI </pre>
---

**Figure 10-10. Saving the SREG Register**

## Review Questions

1. True or false. In ATmega32, 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 WinAVR C language.

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.

**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

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)
{
}
```

See Example 10-8.

#### Example 10-8 (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"

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

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

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

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

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

}

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

## 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 10-9 through 10-13.

### 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 data from PORTC to PORTD.

#### Solution:

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

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

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

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

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

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

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

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

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

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

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

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

- (a) PORTA 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.6.

**Solution:**

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

int main ()
{
    DDRA = 0xFF;           //make PORTA output
    DDRD = 0xFF;           //make PORTD output
    DDRB |= 0x40;          //PORTB.6 as an output
    PORTB |= 0x01;         //activate pull-up

    TCNT0 = -200;          //load Timer0 with -200
    TCCR0 = 0x06;          //Normal mode, falling edge, no prescaler

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

    TIMSK = (1<<TOIE0)|(1<<TOIE1); //enable Timers 0 & 1 int.
    sei ();                 //enable interrupts

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

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

ISR (TIMER0_OVF_vect)      //ISR for Timer0 overflow
{
    TCNT0 = -200;          //TCNT0 = -200
    PORTB ^= 0x40;         //toggle PORTB.6
}

ISR (TIMER1_OVF_vect)      //ISR for Timer1 overflow
{
    TCNT1H = (-31250)>>8;  //the high byte
    TCNT1L = (-31250)&0xFF; //overflow after 31250 clocks
    PORTA ++;              //increment PORTA
}
```

**Example 10-11 (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 = 8 MHz.

**Solution:**

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

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

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

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

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

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

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

### Example 10-12 (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 PORTC.3, whenever INT0 pin goes low. Use the external interrupt in level-triggered mode.

#### Solution:

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

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

    while (1);              //wait here
}

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

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

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

#### Solution:

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

int main ()
{
    DDRC = 1<<3;           //PC3 as an output
    PORTD = 1<<2;           //pull-up activated
    MCUCR = 0x02;           //make INT0 falling edge triggered
    GICR = (1<<INT0);       //enable external interrupt 0
    sei ();                 //enable interrupts

    while (1);              //wait here
}

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

## 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 IF (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 ATmega32 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 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 TIMSK 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, 0x01
OUT    TIFR, 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, INT1 and INT2? 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, all of INT0–INT2 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–INT2 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: INTERRUPT PRIORITY IN THE AVR

47. Explain what happens if both INT1F and INT2F are activated at the same time.
48. 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.
49. Explain what happens if an interrupt is activated while the AVR is serving an interrupt.
50. 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, Timer1 overflow, Timer1 compare B match, Timer1 compare A match, Timer1 input capture, Timer2 overflow, Timer2 output compare match
3. Address locations 0x00 to 0x28. No. It is set when the processor is designed.
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)
OUT TIMSK, R16
```
6. \$12 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. 0x16
3. TIMSK  

```
LDI R16, (1<<TOIE0)
OUT TIMSK, 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 \$0016 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 OCF0 flag is set. If Timer0 compare match interrupt is enabled, the ISR of the compare match 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), PD3 (PORTD.3), and PB2 (PORTB.2) are assigned to INT0, INT1, and INT2, respectively.

3. LDI R20, (1<<INT1)  
OUT GICR,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. True
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. .INCLUDE "M32DEF.INC"  

```

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

```

#### SECTION 10.4: INTERRUPT PRIORITY IN THE AVR

1. False. As shown in Table 10-1, the address of the Timer0 overflow interrupt is \$16, while the address of Timer1 overflow is \$12. 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.