CHAPTER 8

AVR HARDWARE CONNECTION, HEX FILE, AND FLASH LOADERS

OBJECTIVES

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

- **>> Explain the function of the reset pin of the AVR microcontroller**
- **>> Show the hardware connection of the AVR chip**
- **>> Show the use of a crystal oscillator for a clock source**
- **>> Explain how to design an AVR-based system**
- **>> Explain the role of brown-out detection voltage (BOD) in system reset**
- **>> Explain the role of the fuse bytes in an AVR-based system**
- **>> Show the design of the AVR trainer**
- **>> Code a test program in Assembly and C for testing the AVR**
- **>> Show how to download programs into the AVR system using AVRISP**
- **>> Explain the hex file characteristics**

This chapter describes the process of physically connecting and testing AVR-based systems. In the first section we describe the functions of ATmega32 pins. The fuse bits of the AVR are explored in Section 8.2. In Section 8.3 we explain the characteristics of hex files that are produced by AVR Studio. In Section 8.4 we discuss the various methods of loading a program into the AVR microcontroller. It also shows the hardware connection for an AVR trainer using the ATmega16 or ATmega32 chips.

SECTION 8.1: ATMEGA32 PIN CONNECTION

The ATmega family members come in different packages, such as DIP (dual in-line package), MLF (Micro Lead Frame Package), and QFP (quad flat package). They all have many pins that are dedicated to various functions such as I/O, ADC, timer, and interrupts. Notice that Atmel provides a 28-pin version of the ATmega family (ATmega8) with a reduced number of I/O ports for less demanding applications. In Chapter 1 you can see members of the ATmega family and their characteristics. Because the vast majority of educational trainers use the 40-pin chip, we will concentrate on that. Figure 8-1 shows the pins for the ATmega32.

Examine Figure 8-1. Notice that of the 40 pins, a total of 32 are set aside for the four Ports A, B, C, and D, with their alternate functions. The rest of the pins are designated as VCC, AVCC, AREF, GND, XTAL1, XTAL2, and RESET. Next, we describe the function of each pin.

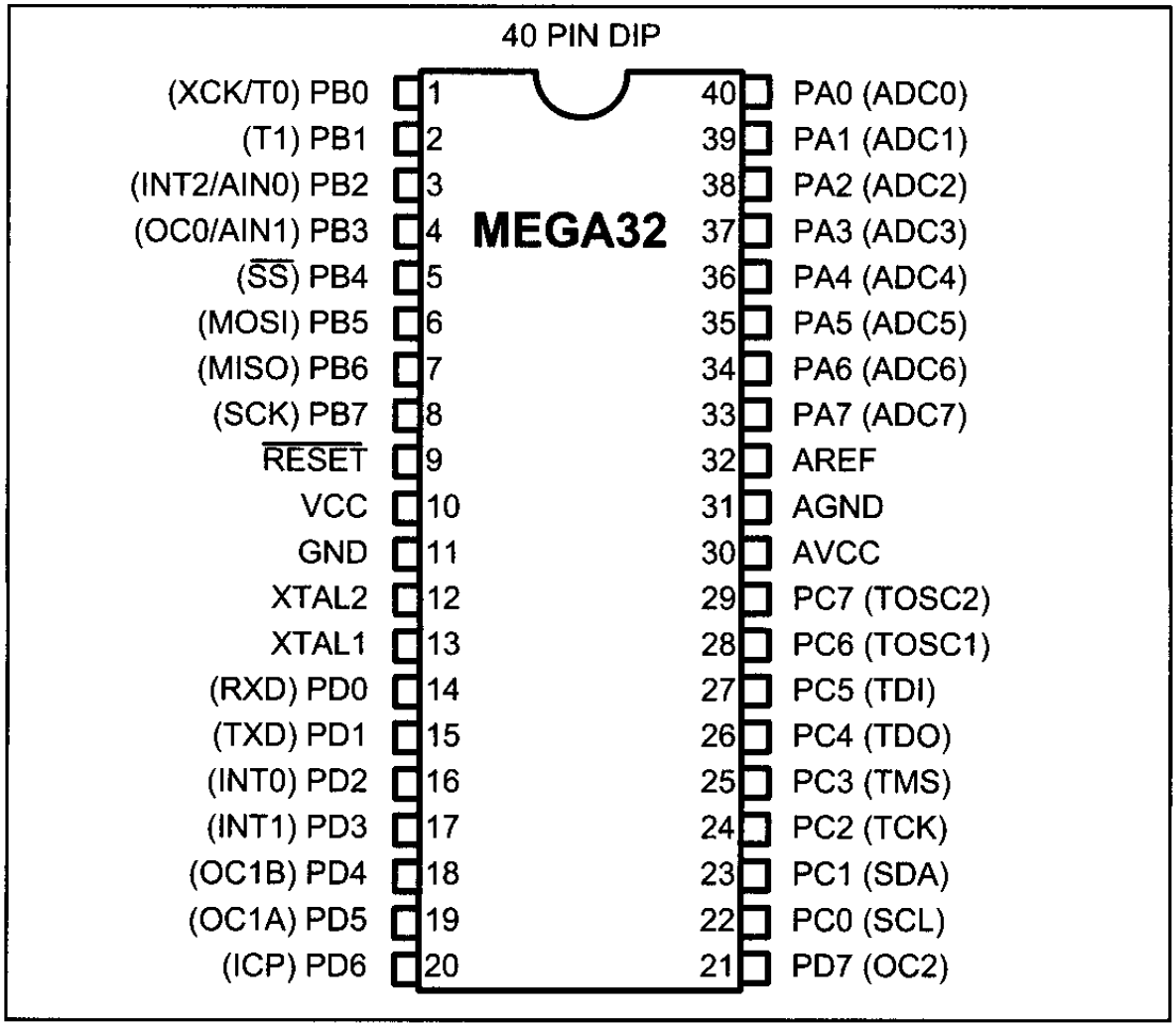


Figure 8-1. ATmega32 Pin Diagram

VCC

This pin provides supply voltage to the chip. The typical voltage source is +5 V. Some AVR family members have lower voltage for VCC pins in order to reduce the noise and power dissipation of the AVR system. For example, ATmega32L operation voltage is 2.7–5.5 V. We can choose other options for the operating voltage level by setting BOD fuse bits. The BOD fuse bits are discussed in the next section.

AVCC

AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to VCC, even if the ADC is not used. In Chapter 13 you will see how to connect this pin if you want to use ADC.

AREF

AREF is the analog reference pin for ADC. In Chapter 13 we will discuss it further.

GND

Two pins are also used for ground. In chips with 40 pins and more, it is common to have multiple pins for VCC and GND. This will help reduce the noise (ground bounce) in high-frequency systems.

XTAL1 and XTAL2

The ATmega32 has many options for the clock source. Most often a quartz crystal oscillator is connected to input pins XTAL1 and XTAL2. The quartz crystal oscillator connected to the XTAL1 and XTAL2 pins also needs two capacitors. One side of each capacitor is connected to the ground as shown in Figure 8-2. Notice that ATmega32 microcontrollers can have speeds of 0 Hz to 16 MHz.

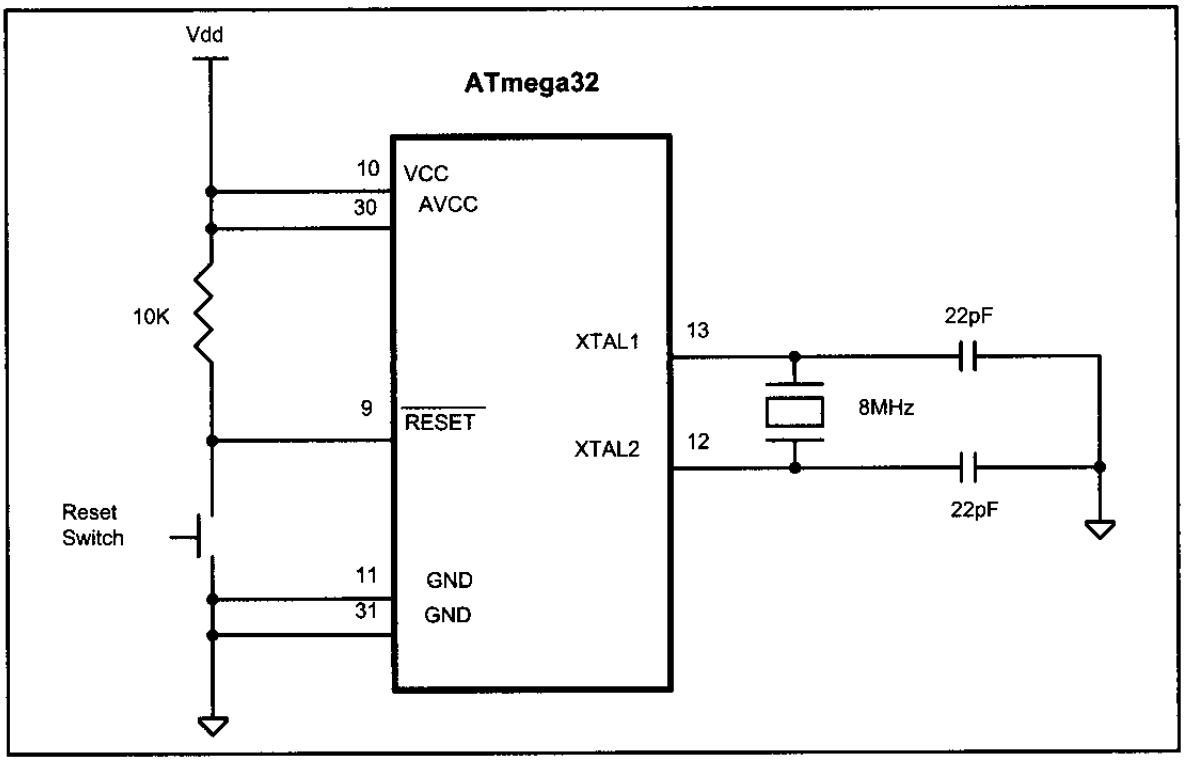


Figure 8-2. Minimum Connection for ATmega32

We can choose options for the clock source and frequency by setting some fuse bits. The fuse bits are discussed in the next section.

RESET

Pin 9 (in the ATmega32, 40-pin DIP) is the RESET pin. It is an input and is active-LOW (normally HIGH). When a LOW pulse is applied to this pin, the microcontroller will reset and terminate all activities. After applying reset, contents of all registers and SRAM locations will be cleared. Notice that after applying reset, all ports will be input because contents of all DDR registers are cleared. The CPU will start executing the program from run location 0x00000 after a brief delay when the RESET pin is forced low and then released.

Table 8-1: RESET Values for Some AVR Registers

Register	Reset Value (hex)
PC	000000
R0–R31	00
DDRx	00
PORTx	00

Figures 8-3a, 8-3b, and 8-3c show three ways of connecting the RESET pin. Figure 8-3b uses a momentary switch for reset circuitry. The most difficult time for any system is during the power-up. The CPU needs both a stable clock source and a stable voltage level to function properly. Some designers put a 10 nF capacitor between the RESET pin and GND to filter the noise during reset and working time. See Figure 3-8c. The diode protects the RESET pin from being powered by the capacitor when the power is off. The AVR chips come with some features that help the reset process. We can choose these features by setting the bits in the fuse bytes. The fuse bits for the reset are discussed in the next section. In addition to the RESET pin there are other sources of reset in the AVR family that will be discussed in future chapters.

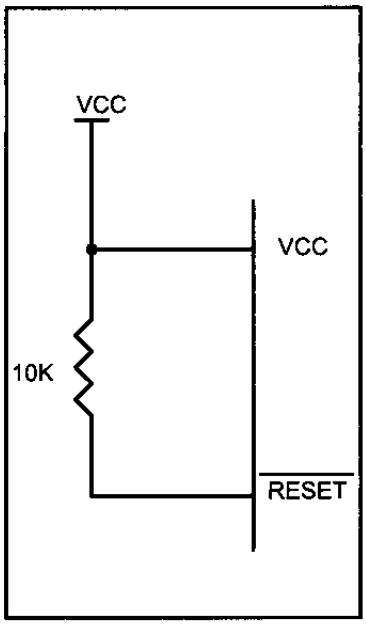


Figure 8-3a. Simple Power-On Reset Circuit

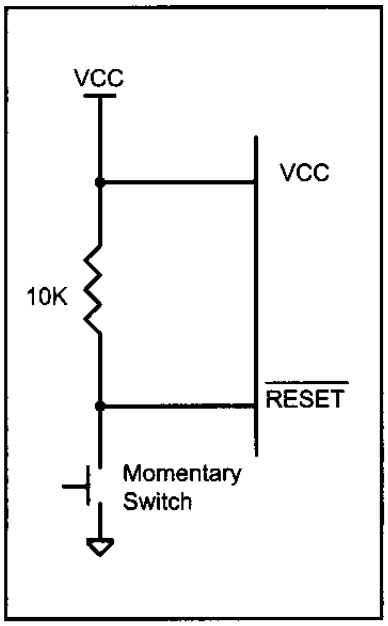


Figure 8-3b. Power-On Reset Circuit with Momentary Switch

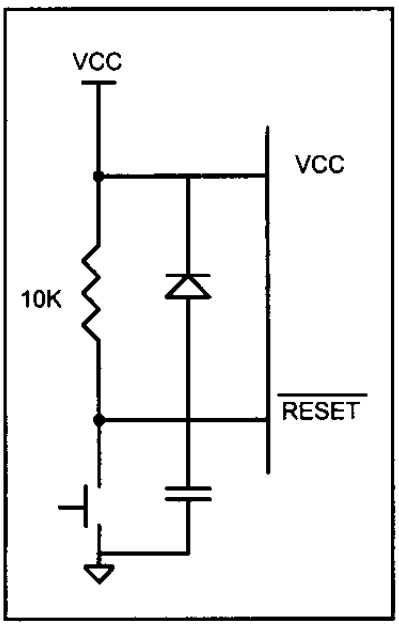


Figure 8-3c. Power-On Reset Circuit with Capacitor and Diode

The number of I/O ports varies among the AVR family members, as we saw in Chapter 4. The following is another look at them for the ATmega32.

Ports A, B, C, D, and E

As shown in Figure 8-1 (and discussed in Chapter 4), the Ports A, B, C, and D use a total of 32 pins. Tables 8-2 through 8-5 provide summaries of features of Ports A–D and their alternative functions. We will study the alternative functions of these pins in future chapters, as we discuss the AVR features.

Table 8-2: Port A Alternate Functions

Bit	Function
PA0	ADC0
PA1	ADC1
PA2	ADC2
PA3	ADC3
PA4	ADC4
PA5	ADC5
PA6	ADC6
PA7	ADC7

Table 8-3: Port B Alternate Functions

Bit	Function
PB0	XCK/T0
PB1	T1
PB2	INT2/AIN0
PB3	OC0/AIN1
PB4	SS
PB5	MOSI
PB6	MISO
PB7	SCK

Table 8-4: Port C Alternate Functions

Bit	Function
PC0	SCL
PC1	SDA
PC2	TCK
PC3	TMS
PC4	TDO
PC5	TDI
PC6	TOSC1
PC7	TOSC2

Table 8-5: Port D Alternate Functions

Bit	Function
PD0	RXD
PD1	TXD
PD2	INT0
PD3	INT1
PD4	OC1B
PD5	OC1A
PD6	ICP
PD7	OC2

Review Questions

1. Which pin is used to reset the ATmega32 chip?
2. Upon power-up, the R0–R31 registers have a value of ____.
3. True or false. Upon power-up, the CPU continues running the code from the line it was running before reset.
4. Reset is an active-_____ (LOW, HIGH) pin.
5. What is the operating voltage of ATmega32L?

SECTION 8.2: AVR FUSE BITS

There are some features of the AVR that we can choose by programming the bits of fuse bytes. These features will reduce system cost by eliminating any need for external components.

ATmega32 has two fuse bytes. Tables 8-6 and 8-7 give a short description of the fuse bytes. Notice that the default values can be different from production to production and time to time. In this section we examine some of the basic fuse bits. The Atmel website (http://www.atmel.com) provides the complete description of fuse bits for the AVR microcontrollers. It must be noted that if a fuse bit is incorrectly programmed, it can cause the system to fail. An example of this is changing the SPIEN bit to 0, which disables SPI programming mode. In this case you will not be able to program the chip any more! Also notice that the fuse bits are '0' if they are programmed and '1' when they are not programmed.

In addition to the fuse bytes in the AVR, there are 4 lock bits to restrict access to the Flash memory. These allow you to protect your code from being copied by others. In the development process it is not recommended to program lock bits because you may decide to read or verify the contents of Flash memory. Lock bits are set when the final product is ready to be delivered to market. In this book we do not discuss lock bits. To study more about lock bits you can read the data sheets for your chip at http://www.atmel.com.

Table 8-6: Fuse Byte (High)

Fuse High Byte	Bit No.	Description	Default Value
OCDEN	7	Enable OCD	1 (unprogrammed)
JTAGEN	6	Enable JTAG	0 (programmed)
SPIEN	5	Enable SPI serial program and data downloading	0 (programmed)
CKOPT	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the chip erase	1 (unprogrammed)
BOOTSZ1	2	Select boot size	0 (programmed)
BOOTSZ0	1	Select boot size	0 (programmed)
BOOTRST	0	Select reset vector	1 (unprogrammed)

Table 8-7: Fuse Byte (Low)

Fuse High Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out detector enable	1 (unprogrammed)
SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)
CKSEL3	3	Select clock source	0 (programmed)
CKSEL2	2	Select clock source	0 (programmed)
CKSEL1	1	Select clock source	0 (programmed)
CKSEL0	0	Select clock source	1 (unprogrammed)

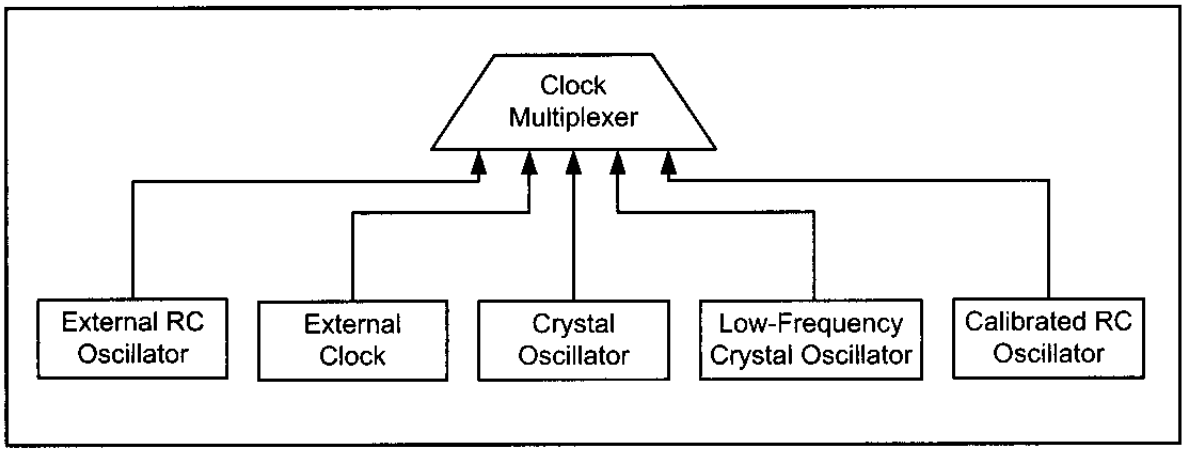


Figure 8-4. ATmega32 Clock Sources

Fuse bits and oscillator clock source

As you see in Figure 8-4, there are different clock sources in AVR. You can choose one by setting or clearing any of the bits CKSEL0 to CKSEL3.

CKSEL0–CKSEL3

The four bits of CKSEL3, CKSEL2, CKSEL1, and CKSEL0 are used to select the clock source to the CPU. The default choice is internal RC (0001), which uses the on-chip RC oscillator. In this option there is no need to connect an external crystal and capacitors to the chip. As you see in Table 8-8, by changing the values of CKSEL0–CKSEL3 we can choose among 1, 2, 4, or 8 MHz internal RC frequencies; but it must be noted that using an internal RC oscillator can cause about 3% inaccuracy and is not recommended in applications that need precise timing.

Table 8-8: Internal RC Oscillator Operation Modes

CKSEL3...0	Frequency
0001	1 MHz
0010	2 MHz
0011	4 MHz
0100	8 MHz

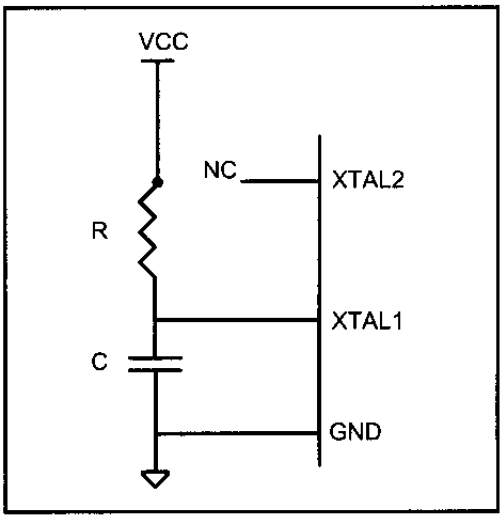


Figure 8-5 External RC

The external RC oscillator is another source to the CPU. As you see in Figure 8-5, to use the external RC oscillator, you have to connect an external resistor and capacitors to the XTAL1 pin. The values of R and C determine the clock speed. The frequency of the RC oscillator circuit is estimated by the equation f = 1/(3RC). When you need a variable clock source you can use the external RC and replace the resistor with a potentiometer. By turning the potentiometer you will be able to change the frequency. Notice that the capacitor value should be at least 22 pF. Also, notice that by programming the CKOPT fuse, you can enable an internal 36 pF capacitor between XTAL1 and GND, and remove the external capacitor. As you see in Table 8-9, by changing the values of CKSEL0–CKSEL3, we can choose different frequency ranges.

Table 8-9: External RC Oscillator Operation Modes

CKSEL3...0	Frequency (MHz)
0101	<0.9
0110	0.9–3.0
0111	3.0–8.0
1000	8.0–12.0

By setting CKSEL0...3 bits to 0000, we can use an external clock source for the CPU. In Figure 8-6a you see the connection to an external clock source.

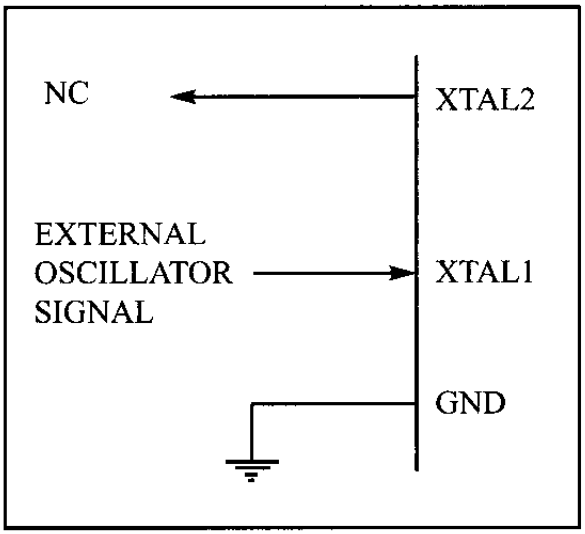


Figure 8-6a. XTAL1 Connection to an External Clock Source

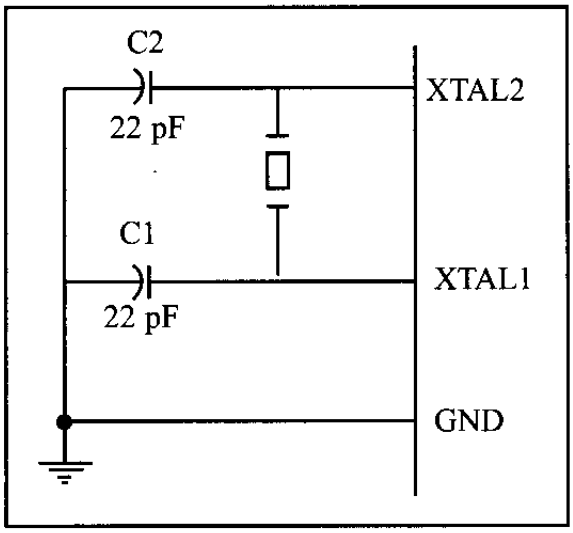


Figure 8-6b. XTAL1–XTAL2 Connection to Crystal Oscillator

The most widely used option is to connect the XTAL1 and XTAL2 pins to a crystal (or ceramic) oscillator, as shown in Figure 8-6b. In this mode, when CKOPT is programmed, the oscillator output will oscillate with a full rail-to-rail swing on the output, causing a more powerful clock signal. This is suitable when the chip drives a second clock buffer or operates in a very noisy environment. As you see in Table 8-10, this mode has a wide frequency range. When CKOPT is not programmed, the oscillator has a smaller output swing and a limited frequency range. This mode cannot be used to drive other clock buffers, but it does reduce power consumption considerably. There are four choices for the crystal oscillator option. Table 8-10 shows all of these choices. Notice that mode 101 cannot be used with crystals, and only ceramic resonators can be used. Example 8-1 shows the relation between crystal frequency and instruction cycle time.

Table 8-10: ATmega32 Crystal Oscillator Frequency Choices and Capacitor Range

CKOPT	CKSEL3...1	Frequency (MHz)	C1 and C2 (pF)
1	101	0.4–0.9	Not for crystals
1	110	0.9–3.0	12–22
1	111	3.0–8.0	12–22
0	101, 110, 111	More than 1.0	12–22

Example 8-1

Find the instruction cycle time for the ATmega32 chip with the following crystal oscillators connected to the XTAL1 and XTAL2 pins.
(a) 4 MHz (b) 8 MHz (c) 10 MHz

Solution:
(a) Instruction cycle time is 1/(4 MHz) = 250 ns
(b) Instruction cycle time is 1/(8 MHz) = 125 ns
(c) Instruction cycle time is 1/(10 MHz) = 100 ns

Fuse bits and reset delay

The most difficult time for a system is during power-up. The CPU needs both a stable clock source and a stable voltage level to function properly. In AVRs, after all reset sources have gone inactive, a delay counter is activated to make the reset longer. This short delay allows the power to become stable before normal operation starts. You can choose the delay time through the SUT1, SUT0, and CKSEL0 fuses. Table 8-11 shows start-up times for the different values of SUT1, SUT0, and CKSEL fuse bits and also the recommended usage of each combination. Notice that the third column of Table 8-11 shows start-up time from power-down mode. Power-down mode is not discussed in this book.

Brown-out detector

Occasionally, the power source provided to the V_{CC} pin fluctuates, causing the CPU to malfunction. The ATmega family has a provision for this, called *brown-out detection*. The BOD circuit compares VCC with BOD-Level and resets the chip if VCC falls below the BOD-Level. The BOD-Level can be either 2.7 V when the BODLEVEL fuse bit is one (not programmed) or 4.0 V when the BODLEVEL fuse is zero (programmed). You can enable the BOD circuit by programming the BODEN fuse bit. When VCC increases above the trigger level, the BOD circuit releases the reset, and the MCU starts working after the time-out period has expired.

A good rule of thumb

There is a good rule of thumb for selecting the values of fuse bits. If you are using an external crystal with a frequency of more than 1 MHz you can set the CKSEL3, CKSEL2, CKSEL1, SUT1, and SUT0 bits to 1 (not programmed) and clear CKOPT to 0 (programmed).

Table 8-11: Startup Time for Crystal Oscillator and Recommended Usage

CKSEL0	SUT1...0	Start-Up Time from Power-Down	Delay from Reset (VCC = 5)	Recommended Usage
0	00	258 CK	4.1	Ceramic resonator, fast rising power
0	01	258 CK	65	Ceramic resonator, slowly rising power
0	10	1K CK	-	Ceramic resonator, BOD enabled
0	11	1K CK	4.1	Ceramic resonator, fast rising power
1	00	1K CK	65	Ceramic resonator, slowly rising power
1	01	16K CK	-	Crystal oscillator, BOD enabled
1	10	16K CK	4.1	Crystal oscillator, fast rising power
1	11	16K CK	65	Crystal oscillator, slowly rising power

Putting it all together

Many of the programs we showed in the first seven chapters were intended to be simulated. Now that we know what we should write in the fuse bits and how we should connect the ATmega32 pins, we can download the hex output file provided by the AVR Studio assembler into the Flash memory of the AVR chip using an AVR programmer.

We can use the following skeleton source code for the programs that we intend to download into a chip. Notice that you have to modify the first line if you use a chip other than ATmega32. As you can see in the comments, if you want to enable interrupts you have to modify ".ORG 0", and if you do not use call the instruction in your code, you can omit the codes that set the stack pointer.

```
.INCLUDE "M32DEF.INC"    ;change it according to your chip
.ORG 0                   ;change it if you use interrupt
LDI   R16,HIGH(RAMEND)   ;set the high byte of stack pointer to
OUT   SPH,R16            ;the high address of RAMEND
LDI   R16,LOW(RAMEND)    ;set the low byte of stack pointer to
OUT   SPL,R16            ;low address of RAMEND

...                      ;place your code here
```

As an example, examine Program 8-1. It will toggle all the bits of Port B with some delay between the "on" and "off" states.

```
;Test Program 8-1: Toggling PORTB for the Atmega32
.INCLUDE "M32DEF.INC"          ;using Atmega32
.ORG 0
      LDI   R16,HIGH(RAMEND)   ;set up stack
      OUT   SPH,R16
      LDI   R16,LOW(RAMEND)
      OUT   SPL,R16
      LDI   R16,0xFF           ;load R16 with 0xFF
      OUT   DDRB,R16           ;Port B is output
BACK:
      COM   R16                ;complement R16
      OUT   PORTB,R16          ;send it to Port B
      CALL  DELAY              ;time delay
      RJMP  BACK               ;keep doing this indefinitely

DELAY:
      LDI   R20,16
L1:   LDI   R21,200
L2:   LDI   R22,250
L3:
      NOP
      NOP
      DEC   R22
      BRNE  L3
      DEC   R21
      BRNE  L2

      DEC   R20
      BRNE  L1
      RET
```

Program 8-1: Toggling Port B in Assembly

Toggle program in C

In Chapter 7 we covered C programming of the AVR using the AVR GCC compiler. Program 8-2 shows the toggle program written in C. It will toggle all the bits of Port B with some delay between the "on" and "off" states.

```
#include <avr/io.h>                    //standard AVR header
#include <util/delay.h>

void delay_ms(int d);

int main(void)
{
      DDRB = 0xFF;                     //Port B is output
      while (1)
      {                                //do forever
            PORTB = 0x55;
            delay_ms(1000);            //delay 1 second
            PORTB = 0xAA;
            delay_ms(1000);            //delay 1 second
      }
      return 0;
}

void delay_ms(int d)
{
      _delay_ms(d);                    //delay 1000 us
}
```

Program 8-2: Toggling Port B in C

Review Questions

1. A given ATmega32-based system has a crystal frequency of 16 MHz. What is the instruction cycle time for the CPU?
2. How many fuse bytes are available in ATmega32?
3. True or false. Upon power-up, both voltage and frequency are stable instantly.
4. The internal RC oscilator works for the frequency range of _____ to _____ MHz.
5. Which fuse bit is used to disable the BOD?
6. True or false. Upon power-up, the CPU starts working immediately.
7. What is the rule of thumb for ATmega32 fuse bits?
8. The brown-out detection voltage can be set at _____ or _____ by_____ fuse bit.
9. True or false. The higher the clock frequency for the system, the lower the power dissipation.

SECTION 8.3: EXPLAINING THE HEX FILE FOR AVR

Intel Hex is a widely used file format designed to standardize the loading (transferring) of executable machine code into a chip. Therefore, the loaders that come with every ROM burner (programmer) support the Intel Hex file format. In many Windows-based assemblers such as AVR Studio, the Intel Hex file is produced according to the settings you set. In the AVR Studio environment, the object file is fed into the linker program to produce the Intel hex file. The hex file is used by a programmer such as the AVRISP to transfer (load) the file into the Flash memory. The AVR Studio assembler can produce three types of hex files. They are (a) Intel Intellec 8/MDS (Intel Hex), (b) Motorola S-record, and (c) Generic. See Table 8-12. In this section we will explain Intel Hex with some examples. We recommend that you do not use AVR GCC if you want to test the programs in this section on your computer. It is better to use a simple .asm file like toggle.asm to understand this concept better.

Table 8-12: Intel Hex File Formats Produced by AVR Studio

Format Name	File Extension	Max. ROM Address
Extended Intel Hex file	.hex	20-bit address
Motorola S-record	.mot	32-bit address
Generic	.gen	24-bit address

Analyzing the Intel Hex file

We choose the hex type of Intel Hex, Motorola S-record, or Generic by using the command-line invocation options or setting the options in the AVR Studio assembler itself. If we do not choose one, the AVR Studio assembler selects Intel Hex by default. Intel Hex supports up to 16-bit addressing and is not applicable for programs more than 64K bytes in size. To overcome this limitation AVR Studio uses extended Intel Hex files, which support type 02 records to extend address space to 1M. We will explain extended Intel Hex file format in this section. Figure 8-10 shows the Intel Hex file of the test program whose list file is given in Figure 8-8. Since the programmer (loader) uses the Hex file to download the opcode into Flash, the hex file must provide the following: (1) the number of bytes of information to be loaded, (2) the information itself, and (3) the starting address where the information must be placed. Each record (line) of the Hex file consists of six parts as follows:

```
:BBAAAATTHHHHH.....HHHHCC
```

The following describes each part:

1. ":" Each line starts with a colon.
2. BB, the count byte. This tells the loader how many bytes are in the line.
3. AAAA is for the record address. This is a 16-bit address. The loader places the first byte of record data into this Flash location. This is the case in files that are less than 64 KB. For files that are more than 64 KB the address field shows the record address in the current segment.

4. TT is for type. This field is 00, 01, or 02. If it is 00, it means that there are more lines to come after this line. If it is 01, it means that this is the last line and the loading should stop after this line. If it is 02, it indicates the current segment address. To calculate the absolute address of each record (line), we have to shift the current segment address 4 bits to left and then add it to the record address. Examples 8-2 and 8-3 show how to calculate the absolute address of a record in extended Intel hex file.
5. HH.....H is the real information (data or code). The loader places this information into successive memory locations of Flash. The information in this field is presented as low byte followed by the high byte.
6. CC is a single byte. This last byte is the checksum byte for everything in that line. The checksum byte is used for error checking. Checksum bytes are discussed in detail in Chapters 6 and 7. Notice that the checksum byte at the end of each line represents the checksum byte for everything in that line, and not just for the data portion.

Example 8-2

What is the absolute address of the first byte of a record that has 0025 in the address field if the last type 02 record before it has the segment address 0030?

Solution:

To calculate the absolute address of each record (line), we have to shift the segment address (0030) four bits to the left and then add it to the record address (0025):

```
0030 (2 bytes segment address) shifted 4 bits to the left  -->     00300
0025 (record address)                                          +    25
                                                               --------
=>   (absolute address)                                          00325
```

Example 8-3

What is the absolute address of the first byte of the second record below?

```
:02000002000FC
:10000000008E00EBF0FE50DBF0FEF07BB05E500953C
```

Solution:

To calculate the absolute address of the first byte of the second record, we have to shift left the segment address (0000, as you see in the first record) four bits and then add it to the second record address (0000, as you see in the second record).

```
0000 (segment address) shift 4 bits to the left  -->  00000
                                                    + 0000      (record address)
                                                    --------
                                                      000000    (absolute address)
```

Analyzing the bytes in the Flash memory vs. list file

The data in the Flash memory of the AVR is recorded in a way that is called *Little-endian*. This means that the high byte of the code is located in the higher address location of Flash memory, and the low byte of the code is located in the lower address location of Flash memory. Compare the first word of code (`e008`) in Figure 8-8 with the first two bytes of Flash memory (`08e0`) in Figure 8-7. As you see, 08, which is the low byte of the first instruction (`LDI R16,HIGH(RAMEND)`) in the code, is placed in the lower location of Flash memory, and e0, which is the high byte of the instruction in the code, is placed in the next location of program space just after 08.

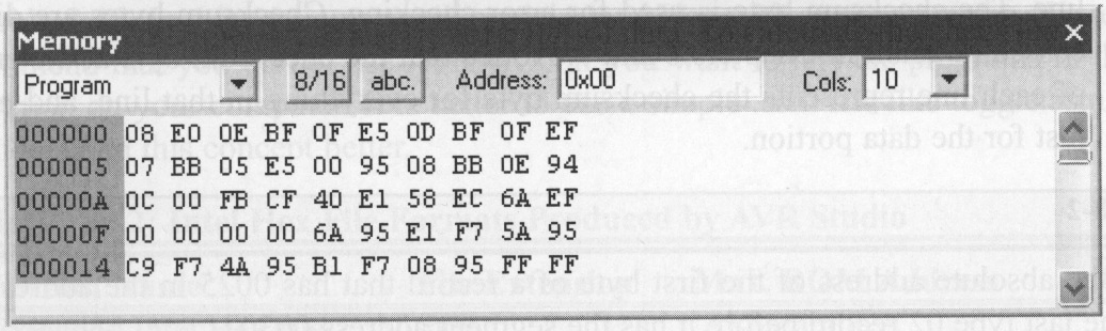


Figure 8-7. AVR Flash Memory Contents

```
LOC    OBJ           LINE
                     .ORG 0x000
000000 e008                LDI   R16,HIGH(RAMEND)
000001 bf0e                OUT   SPH,R16
000002 e50f                LDI   R16,LOW(RAMEND)
000003 bf0d                OUT   SPL,R16

000004 ef0f                LDI   R16,0xFF
000005 bb07                OUT DDRB,R16
000006 e505                LDI   R16,0x55
                     BACK:
000007 9500                COM         R16
000008 bb08                OUT   PORTB,R16
000009 940e 000c           CALL  DELAY_1S
00000b cffb                RJMP  BACK
                     DELAY_1S:
00000c e140                LDI   R20,16
00000d ec58          L1:   LDI   R21,200
00000e ef6a          L2:   LDI   R22,250
                     L3:
00000f 0000                NOP
000010 0000                NOP
000011 956a                DEC   R22
000012 f7e1                BRNE  L3
000013 955a                DEC   R21
000014 f7c9                BRNE  L2
000015 954a                DEC   R20
000016 f7b1                BRNE  L1
000017 9508                RET
```

Figure 8-8. List File for Test Program
(Comments and other lines are deleted, and some spaces are added for simplicity.)

As we mentioned in Chapter 2, each Flash location in the AVR is 2 bytes long. So, for example, the first byte of Flash location #2 is Byte #4 of the code. See Figure 8-9.

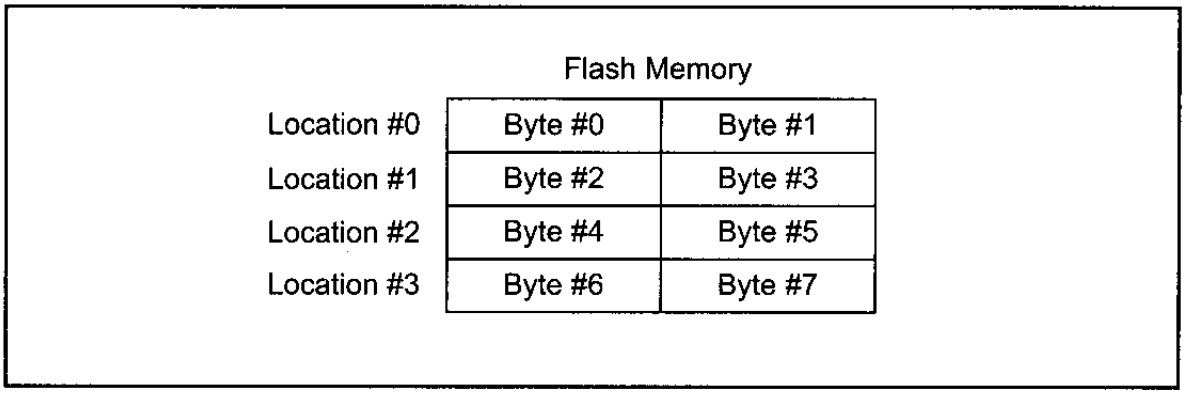


Figure 8-9. AVR Flash Memory Locations

In Figure 8-10 you see the hex file of the toggle code. The first record (line) is a type 02 record and indicates the current segment address, which is `0000`. The next record (line) is a type 00 record and contains the data (the code to be loaded into the chip). After ':' the record starts with `10`, which means that the data field contains 10 (16 decimal) bytes of data. The next field is the address field (`0000`), and it indicates that the first byte of the data field will be placed in address location 0 in the current segment. So the first byte of code will be loaded into location 0 of Flash memory. (Reexamine Example 8-3 if needed.) Also, notice the use of `.ORG 0x000` in the code. The next field is the data field, which contains the code to be loaded into the chip. The first byte of the data field is `08`, which is the low byte of the first instruction (`LDI R16,HIGH(RAMEND)`). See Figure 8-8. The last field of the record is the checksum byte of the record. Notice that the checksum byte at the end of each line represents the checksum byte for everything in that line, and not just for the data portion.

Pay attention to the address field of the next record (`0010`) in Figure 8-10 and compare it with the address of the `bb08` instruction in the list file in Figure 8-8. As you can see, the address in the list file is `000008`, which is exactly half of the address of the `bb08` instruction in the hex file, which is `0010`. That is because each Flash location (word) contains 2 bytes.

```
:020000020000FC
:1000000008E00EBF0FE50DBF0FEF07BB05E500953C
:1000100008BB0E940C00FBCF40E158EC6AEF0000E7
:1000200000006A95E1F75A95C9F74A95B1F7089526
:00000001FF

Separating the fields, we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH          CC
:02 0000 02 0000                                      FC
:10 0000 00 08E00EBF0FE50DBF0FEF07BB05E50095          3C
:10 0010 00 08BB0E940C00FBCF40E158EC6AEF0000          E7
:10 0020 00 00006A95E1F75A95C9F74A95B1F70895          26
:00 0000 01                                           FF
```

Figure 8-10. Intel Hex File Test Program with the Intel Hex Option

Examine Examples 8-4 through 8-6 to gain insight into the Intel Hex file format.

Example 8-4

From Figure 8-10, analyze the six parts of line 3.

Solution:

After the colon (:), we have 10, which means that 16 bytes of data are in this line. 0010H is the record address, and means that 08, which is the first byte of the record, is placed in address location 10H (16 decimal). Next, 00 means that this is not the last line of the record. Then the data, which is 16 bytes, is as follows: `08BB0E940C00FBCF40E158EC6AEF0000`. Finally, the last byte, E7, is the checksum byte.

Example 8-5

Compare the data portion of the Intel Hex file of Figure 8-10 with the opcodes in the list file of the test program given in Figure 8-8. Do they match?

Solution:

In the second line of Figure 8-10, the data portion starts with 08E0H, where the low byte is followed by the high byte. That means it is E008, the opcode for the instruction "`LDI R16,HIGH(RAMEND)`", as shown in the list file of Figure 8-8. The last byte of the data in line 5 is 0895, which is the opcode for the "`RET`" instruction in the list file.

Example 8-6

(a) Verify the checksum byte for line 3 of Figure 8-10. (b) Verify also that the information is not corrupted.

Solution:

(a) `10 + 00 + 00 + 00 + 08 + E0 + 0E + BF + 0F + E5 + 0D + BF + 0F + EF + 07 + BB + 05 + E5 + 00 + 95 = 6C4` in hex. Dropping the carries (6) gives C4H, and its 2's complement is 3CH, which is the last byte of line 3.

(b) If we add all the information in line 2, including the checksum byte, and drop the carries we should get `10 + 00 + 00 + 00 + 08 + E0 + 0E + BF + 0F + E5 + 0D + BF + 0F + EF + 07 + BB + 05 + E5 + 00 + 95 + 3C = 700`. Dropping the carries (7) gives 00H, which means OK.

Review Questions

1. True or false. The Intel Hex file format does not use the checksum byte method to ensure data integrity.
2. The first byte of a line in an Intel Hex file represents ____.
3. The last byte of a line in an Intel Hex file represents ____.
4. In the TT field of an Intel Hex file, we have 00. What does it indicate?
5. Find the checksum byte for the following values: 22H, 76H, 5FH, 8CH, 99H.
6. In Question 5, add all the values and the checksum byte. What do you get?

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

In this section, we show various ways of loading a hex file into the AVR microcontroller. We also discuss the connection for a simple AVR trainer.

Atmel has skillfully designed AVR microcontrollers for maximum flexibility of loading programs. The three primary ways to load a program are:

1. Parallel programming. In this way a device burner loads the program into the microcontroller separate from the system. This is useful on a manufacturing floor where a gang programmer is used to program many chips at one time. Most mainstream device burners support the AVR families: EETools is a popular one. The device programming method is straightforward: The chip is programmed before it is inserted into the circuit. Or, the chip can be removed and reprogrammed if it is in a socket. A ZIF (zero insertion force) socket is even quicker and less damaging than a standard socket. When removing and reinserting, we must observe ESD (electrostatic discharge) procedures. Although AVR devices are rugged, there is always a risk when handling them. Using this method allows all of the device's resources to be utilized in the design. No pins are shared, nor are internal resources of the chip used as is the case in the other two methods. This allows the embedded designer to use the minimum board space for the design.
2. An in-circuit serial programmer (ISP) allows the developer to program and debug their microcontroller while it is in the system. This is done by a few wires with a system setup to accept this configuration. In-circuit serial programming is excellent for designs that change or require periodic updating. AVR has two methods of ISP. They are SPI and JTAG. Most of the ATmega family supports both methods. The SPI uses 3 pins, one for send, one for receive, and one for clock. These pins can be used as I/O after the device is programmed. The designer must make sure that these pins do not conflict with the programmer. Notice that SPI stands for "serial peripheral interface" and is a protocol. But ISP stands for "in-circuit serial programming" and is a method of code loading. AVRISP and many other devices support ISP. To connect AVRISP to your device you also need to connect VCC, GND, and RESET pins. You must bring the pins to a header on the board so that the programmer can connect to it. Figure 8-11 shows the pin connections.

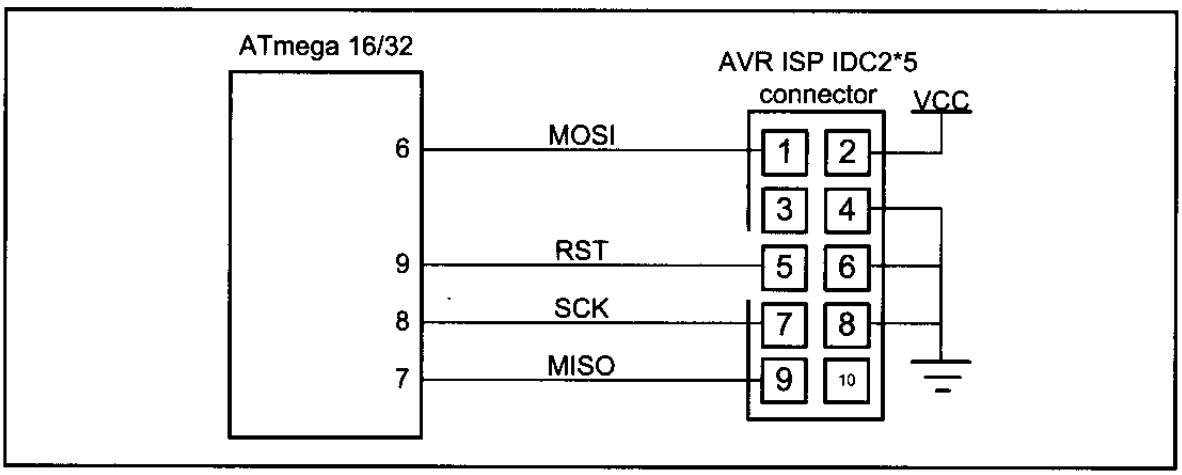


Figure 8-11. ISP 10-pin Connections (See www.Atmel.com for 6-pin version)

Another method of ISP is JTAG. JTAG is another protocol that supports in-circuit programming and debugging. It means that in addition to programming you can trace your program on the chip line by line and watch or change the values of memory locations, ports, or registers while your program is running on the chip.

3. A boot loader is a piece of code burned into the microcontroller's program Flash. Its purpose is to communicate with the user's board to load the program. A boot loader can be written to communicate via a serial port, a CAN port, a USB port, or even a network connection. A boot loader can also be designed to debug a system, similar to the JTAG. This method of programming is excellent for the developer who does not always have a device programmer or a JTAG available. There are several application notes on writing boot loaders on the Web. The main drawback of the boot loader is that it does require a communication port and program code space on the microcontroller. Also, the boot loader has to be programmed into the device before it can be used, usually by one of the two previous ways.

   The boot loader method is ideal for the developer who needs to quickly program and test code. This method also allows the update of devices in the field without the need of a programmer. All one needs is a computer with a port that is compatible with the board. (The serial port is one of the most commonly used and discussed, but a CAN or USB boot loader can also be written.) This method also consumes the largest amount of resources. Code space must be reserved and protected, and external devices are needed to connect and communicate with the PC. Developing projects using this method really helps programmers test their code. For mature designs that do not change, the other two methods are better suited.

AVR trainers

There are many popular trainers for the AVR chip. The vast majority of them have a built-in ISP programmer. See the following website for more information and support about the AVR trainers. For more information about how to use an AVR trainer you can visit the www.MicroDigitalEd.com website.

Review Questions

1. Which method(s) to program the AVR microcontroller is/are the best for the manufacturing of large-scale boards?
2. Which method(s) allow(s) for debugging a system?
3. Which method(s) would allow a small company to develop a prototype and test an embedded system for a variety of customers?
4. True or false. The ATmega32 has Flash program ROM.
5. Which pin is used for reset in the ATmega32?
6. What is the status of the RESET pin when it is not activated?

The information about the trainer board can be found at:
www.MicroDigitalEd.com

SUMMARY

This chapter began by describing the function of each pin of the ATmega32. A simple connection for ATmega32 was shown. Then, the fuse bytes were discussed. We use fuse bytes to enable features such as BOD and clock source and frequency. We also explained the Intel Hex file format and discussed each part of a record in a hex file using an example. Then, we explained list files in detail. The various ways of loading a hex file into a chip were discussed in the last section. The connections to a ISP device were shown.

PROBLEMS

SECTION 8.1: ATMEGA32 PIN CONNECTION

1. The ATmega32 DIP package is a(n) ____-pin package.
2. Which pins are assigned to VCC and GND?
3. In the ATmega32, how many pins are designated as I/O port pins?
4. The crystal oscillator is connected to pins ____ and ____ .
5. True or false. AVR chips comes only in DIP packages.
6. Indicate the pin number assigned to RESET in the DIP package.
7. The RESET pin is normally _____ (LOW, HIGH) and needs a _____ (LOW, HIGH) signal to be activated.
8. In the ATmega32, how many pins are set aside for the VCC?
9. In the ATmega32, how many pins are set aside for the GND?
10. True or false. In connecting VCC pins to power both must be connected.
11. RESET is an _____ (input, output) pin.
12. How many pins are designated as Port A and what are those in the 40-pin DIP package?
13. How many pins are designated as Port B and what are those in the 40-pin DIP package?
14. How many pins are designated as Port C and what are those in the 40-pin DIP package?
15. How many pins are designated as Port D and what are those in the 40-pin DIP package?
16. Upon reset, all the bits of ports are configured as _____ (input, output).

SECTION 8.2: AVR FUSE BITS

17. How many clock sources does the AVR have?
18. What fuse bits are used to select clock source?
19. Which clock source do you suggest if you need a variable clock source?
20. Which clock source do you suggest if you need to build a system with minimum external hardware?
21. Which clock source do you suggest if you need a precise clock source?
22. How many fuse bytes are there in the AVR?

23. Which fuse bit is used to set the brown-out detection voltage for the ATmega32?
24. Which fuse bit is used to enable and disable the brown-out detection voltage for the ATmega32?
25. If the brown-out detection voltage is set to 4.0 V, what does it mean to the system?

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

26. True or false. The Hex option can be set in AVR Studio.
27. True or false. The extended Intel Hex file can be used for ROM sizes of less than 64 kilobytes.
28. True or false. The extended Intel Hex file can be used for ROM sizes of more than 64 kilobytes.
29. Analyze the six parts of line 3 of Figure 8-10.
30. Verify the checksum byte for line 3 of Figure 8-10. Verify also that the information is not corrupted.
31. What is the difference between Intel Hex files and extended Intel Hex files?

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

32. True or false. To use a parallel programmer, we must remove the AVR chip from the system and place it into the programmer.
33. True or false. ISP can work only with Flash chips.
34. What are the different ways of loading a code into an AVR chip?
35. True or false. A boot loader is a kind of parallel programmer.

ANSWERS TO REVIEW QUESTIONS

SECTION 8.1: ATMEGA32 PIN CONNECTION

1. 9
2. 0
3. False
4. LOW
5. 2.7 V–5.5 V

SECTION 8.2: AVR FUSE BITS

1. 1/16 MHz = 62.5 ns
2. 16 bits = 2 bytes
3. False
4. 1, 8
5. BODEN
6. False
7. If you are using an external crystal with a frequency of more than 1 MHz you can set the CKSEL3, CKSEL2, CKSEL1, SUT1, and SUT0 bits to 1 (not programmed) and clear CKOPT to 0 (programmed).
8. 2.7 V, 4 V, BODLEVEL
9. False

SECTION 8.3: EXPLAINING THE INTEL HEX FILE FOR AVR

1. False
2. The number of bytes of data in the line
3. The checksum byte of all the bytes in that line
4. 00 means this is not the last line and that more lines of data follow.
5. 22H + 76H + 5FH + 8CH + 99H = 21CH. Dropping the carries we have 1CH and its 2's complement, which is E4H.
6. 22H + 76H + 5FH + 8CH + 99H + E4H = 300H. Dropping the carries, we have 00, which means that the data is not corrupted.

SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD

1 Device burner
2. JTAG and boot loader
3. ISP
4. True
5. Pin 9
6. HIGH

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.

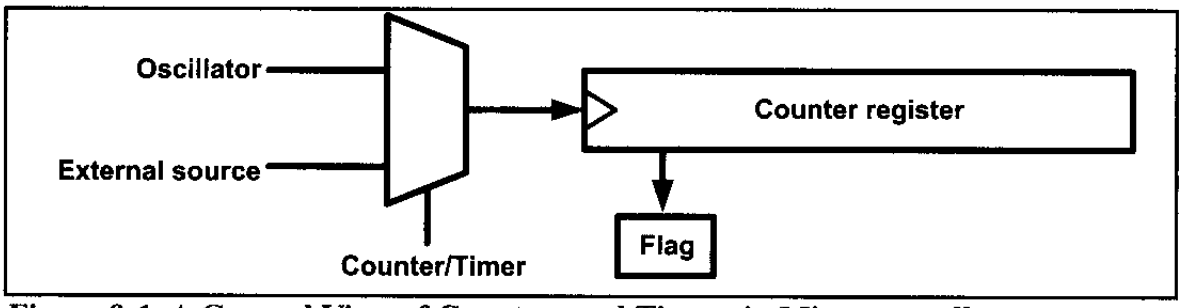


Figure 9-1. A General View of Counters and Timers in Microcontrollers

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 AVRs. 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.

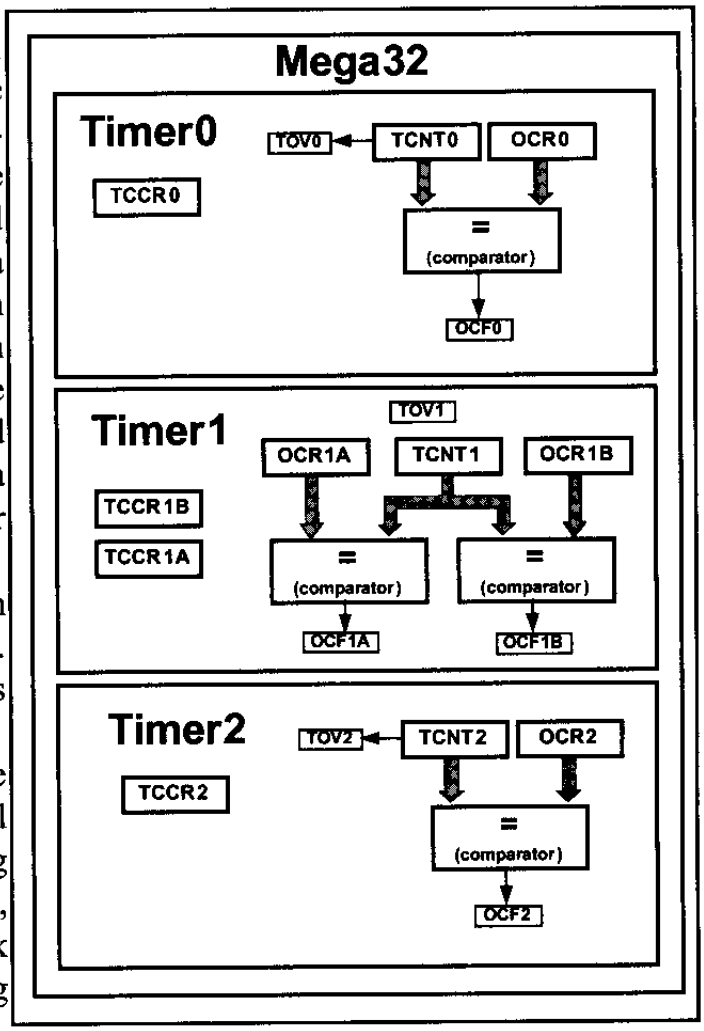


Figure 9-2. Timers in ATmega32

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.

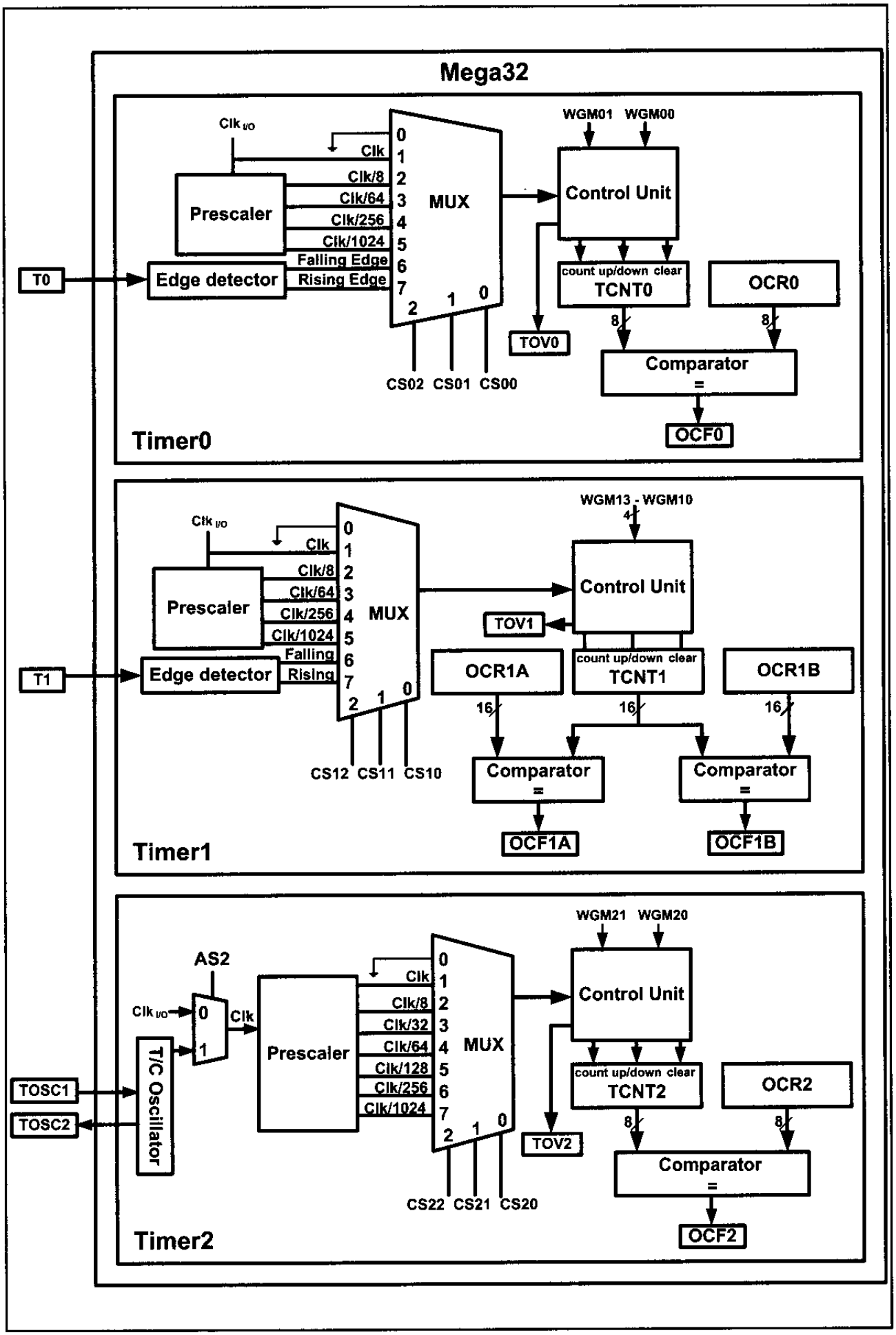


Figure 9-3. Timers in ATmega32

Timer0 programming

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

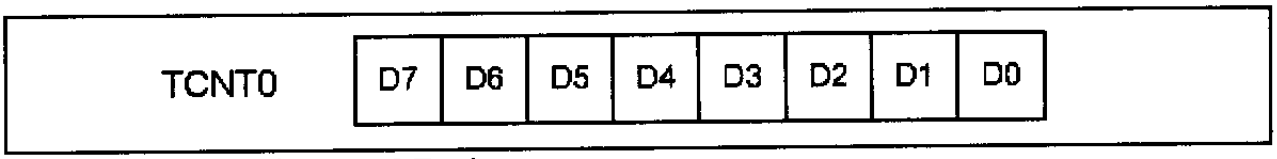


Figure 9-4. Timer/Counter 0 Register

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

FOC0 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.

WGM00, WGM01

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

COM01:00 D5 D4 Compare Output Mode:
These bits control the waveform generator (see Chapter 15).

CS02:00

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 MHz and T = 1/10 MHz = 0.1 µs
(b) F = 8 MHz and T = 1/8 MHz = 0.125 µs
(c) F = 1 MHz and T = 1/1 MHz = 1 µs

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.

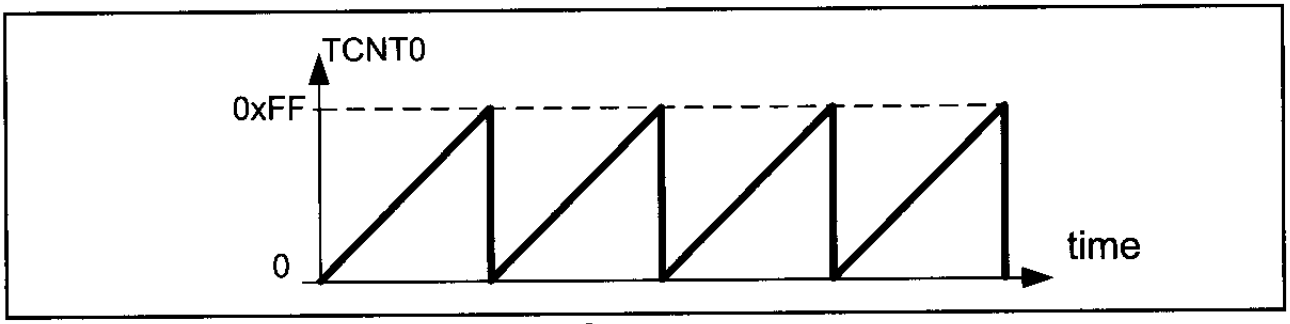


Figure 9-7. Timer/Counter 0 Normal Mode

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

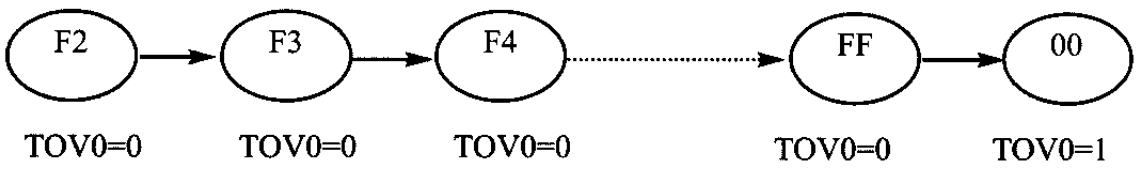
In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PORTB.5 bit. Timer0 is used to generate the time delay. Analyze the program.

```
.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 MHz = 0.125 μs. In other words, Timer0 counts up each 0.125 μs resulting in delay = number of counts × 0.125 μs.
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 × 0.125 μs = 1.75 μs 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.

```
                                    Cycles
      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 × (14 + 24) × 0.125 μs = 9.5 μs and F = 1 / T = 105.263 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:
   $T_{clock} = 1/F_{Timer}$
   where F_{Timer} is the frequency of the clock used for the timer. For example, in no prescaler mode, $F_{Timer} = F_{oscillator}$. T_{clock} gives the period at which the timer increments.
2. Divide the desired time delay by T_{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 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 XTAL = 8 MHz, write a program to generate a square wave with a period of 12.5 μs on pin PORTB.3.

Solution:

For a square wave with T = 12.5 μs we must have a time delay of 6.25 μs. Because XTAL = 8 MHz, the counter counts up every 0.125 μs. This means that we need 6.25 μs / 0.125 μs = 50 clocks. 256 - 50 = 206 = 0xCE. Therefore, we have TCNT0 = 0xCE.

```
.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 kHz = 62.5 μs the period of the square wave.
(b) 1/2 of it for the high and low portions of the pulse is 31.25 μs.
(c) 31.25 μs / 0.125 μs = 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 μs. 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

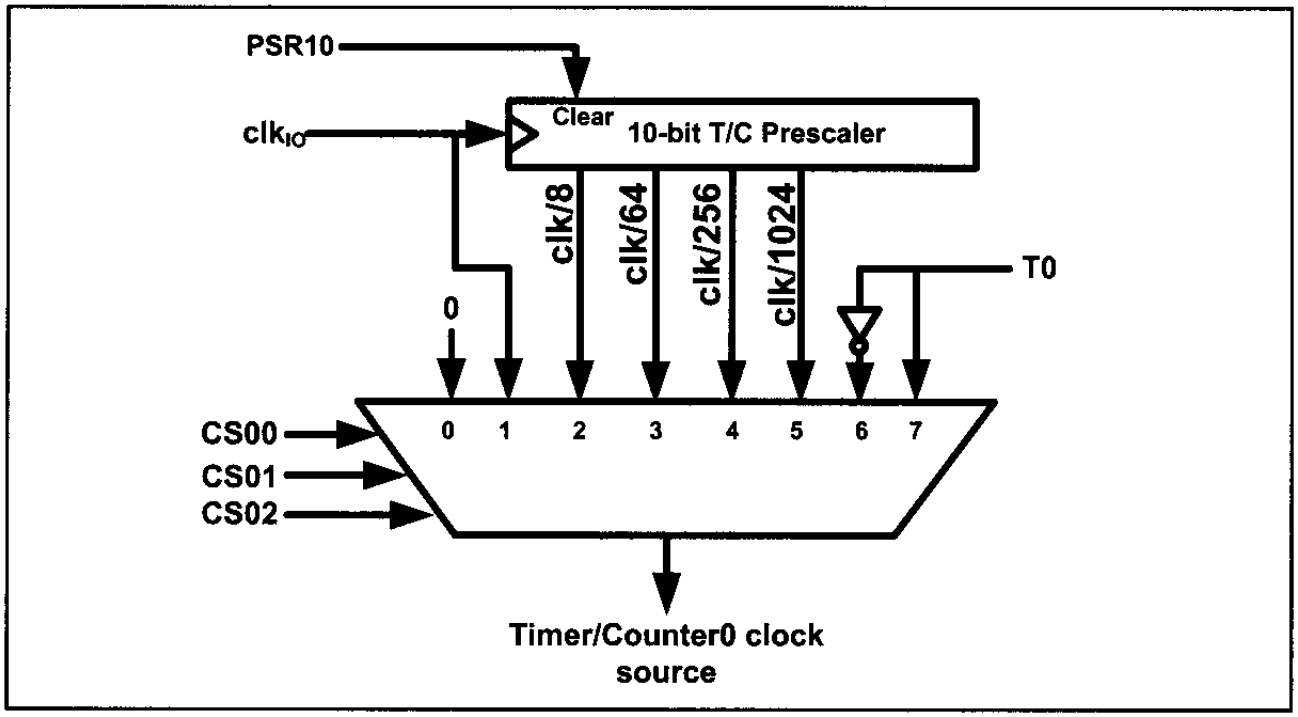


Figure 9-9. Timer/Counter 0 Prescaler

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) × 0.125 μs = 32 μs. That gives us the smallest frequency of 1 / (2 × 32 μs) = 1 / (64 μs) = 15.625 kHz.

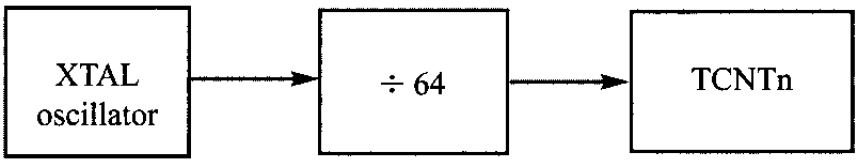
these factors are beyond the control of the AVR programmer. We saw in Example 9-9 that the largest time delay is achieved by making TCNT0 zero. What if that is not enough? We can use the prescaler option in the TCCR0 register to increase the delay by reducing the period. The prescaler option of TCCR0 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 × 8 MHz = 125 kHz due to 1:64 prescaler and T = 1/125 kHz = 8 μs
(b) 1/64 × 16 MHz = 250 kHz due to prescaler and T = 1/250 kHz = 4 μs
(c) 1/64 × 10 MHz = 156.2 kHz due to prescaler and T = 1/156 kHz = 6.4 μs

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   R20,0x0
```

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 × 64 × 0.125 μs = 1920 μs, or from Example 9-10, we have 240 × 8 μs = 1920 μs.

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 MHz × 1/1024 = 7812.5 Hz due to 1:1024 prescaler and T = 1/7812.5 Hz = 128 ms = 0.128 ms

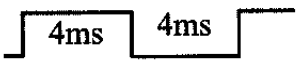
(b) To get the largest delay, we make TCNT0 zero. Making TCNT0 zero means that the timer will count from 00 to 0xFF, and then roll over to raise the TOV0 flag. As a result, it goes through a total of 256 states. Therefore, we have delay = (256 - 0) × 128 μs = 32,768 μs = 0.032768 seconds.

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 Hz = 8 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 ms / 0.125 μs) / 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
OUT TCNT0,R20
```

(b)
```
LDI R17,-60
OUT TCNT0,R17
```

(c)
```
LDI R25,-12
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.

Decimal	*2's complement (TCNT0 value)*
-200	0x38
-60	0xC4
-12	0xF4

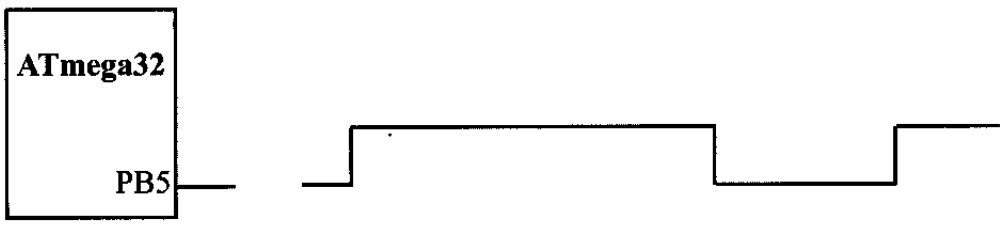
Example 9-16

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 × 0.125 μs = 18.75 μs. The high portion of the pulse is twice the size of the low portion (66% duty cycle). Therefore, we have: T = high portion + low portion = 2 × 18.75 μs + 18.75 μs = 56.25 μs and frequency = 1 / 56.25 μs = 17.777 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

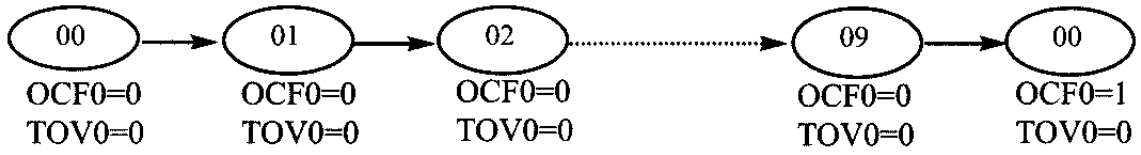
In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PORTB.5 bit. Timer0 is used to generate the time delay. Analyze the program.

```
.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.



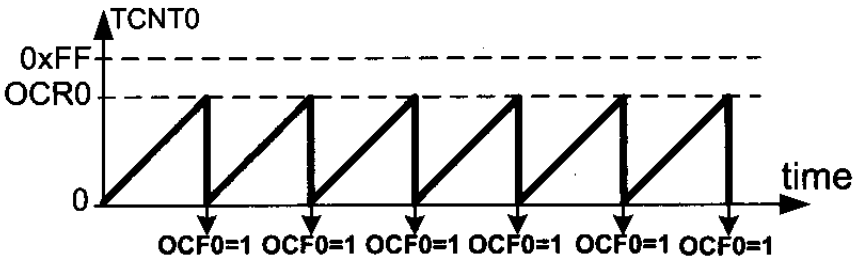


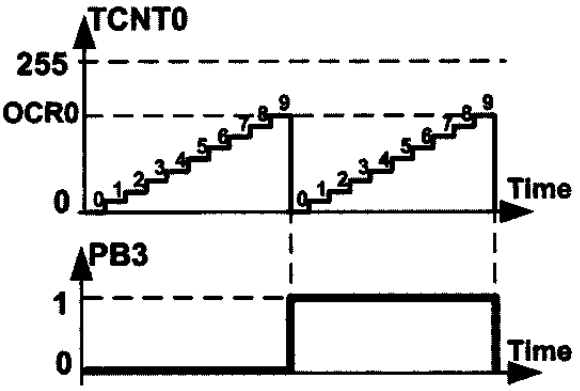
Figure 9-10. Timer/Counter 0 CTC Mode

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:

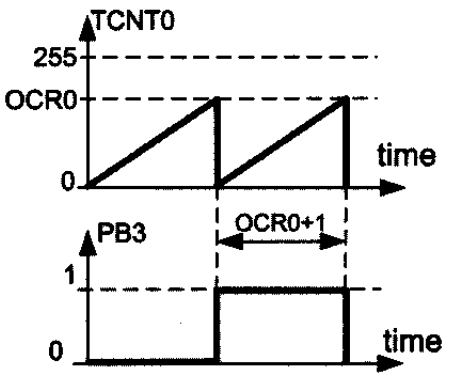
OCR0 is loaded with 9 and TCNT0 is cleared; Thus, after 9 clocks TCNT0 becomes equal to OCR0. On the next clock, the OCF0 flag is set and the reset occurs. That means the TCNT0 is cleared after 9 + 1 = 10 clocks. Because XTAL = 8 MHz, the counter counts up every 0.125 µs. Therefore, we have 10 × 0.125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



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 × 0.125 µs = 8 µs. OCR0 is loaded with 89; thus, after 90 clocks OCF0 is set. Therefore we have 90 × 8 µs = 720 µs.

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 × 0.125 μs = 128 μs. Thus, in order to generate a delay of 25.6 ms we should wait 25.6 ms / 128 μs = 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 MHz = 0.125 μs	1 ms/0.125 μs = 8000
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 μs	1 ms/1 μs = 1000
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 μs	1 ms/8 μs = **125**
256	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 μs	1 ms/32 μs = **31.25**
1024	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz= 128 μs	1 ms/128 μs = **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
```

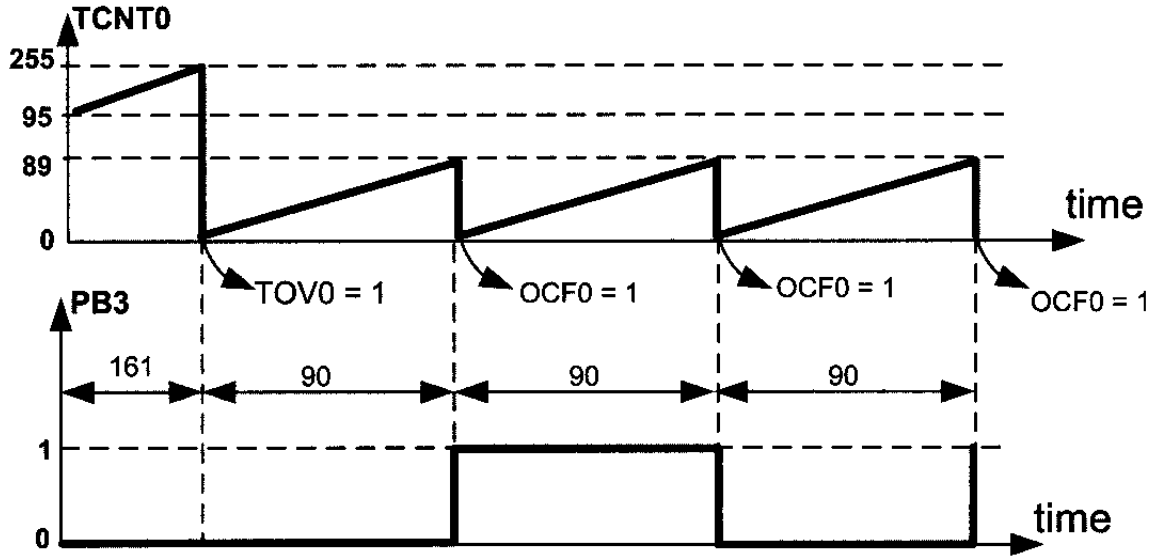
Notice that the comparator checks for equality; thus, if we load the OCR0 register with a value that is smaller than TCNT0's value, the counter will miss the compare match and will count up until it reaches the maximum value of $FF and rolls over. This causes a big delay and is not desirable in many 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 × 0.125 μs = 31.375 μs. The next compare matches occur after 90 clocks, which means after 90 × 0.125 μs = 11.25 μs.

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

FOC2 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.

WGM20, WGM21

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

COM21:20 D5 D4 Compare Output Mode:
These bits control the waveform generator (see Chapter 15).

CS22:20

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.

Bit	7	6	5	4	3	2	1	0
					AS2	TCN2UB	OCR2UB	TCR2UB

AS2 When it is zero, Timer2 is clocked from $clk_{I/O}$. When it is set, Timer2 works as RTC.

Figure 9-12. ASSR (Asynchronous Status Register)

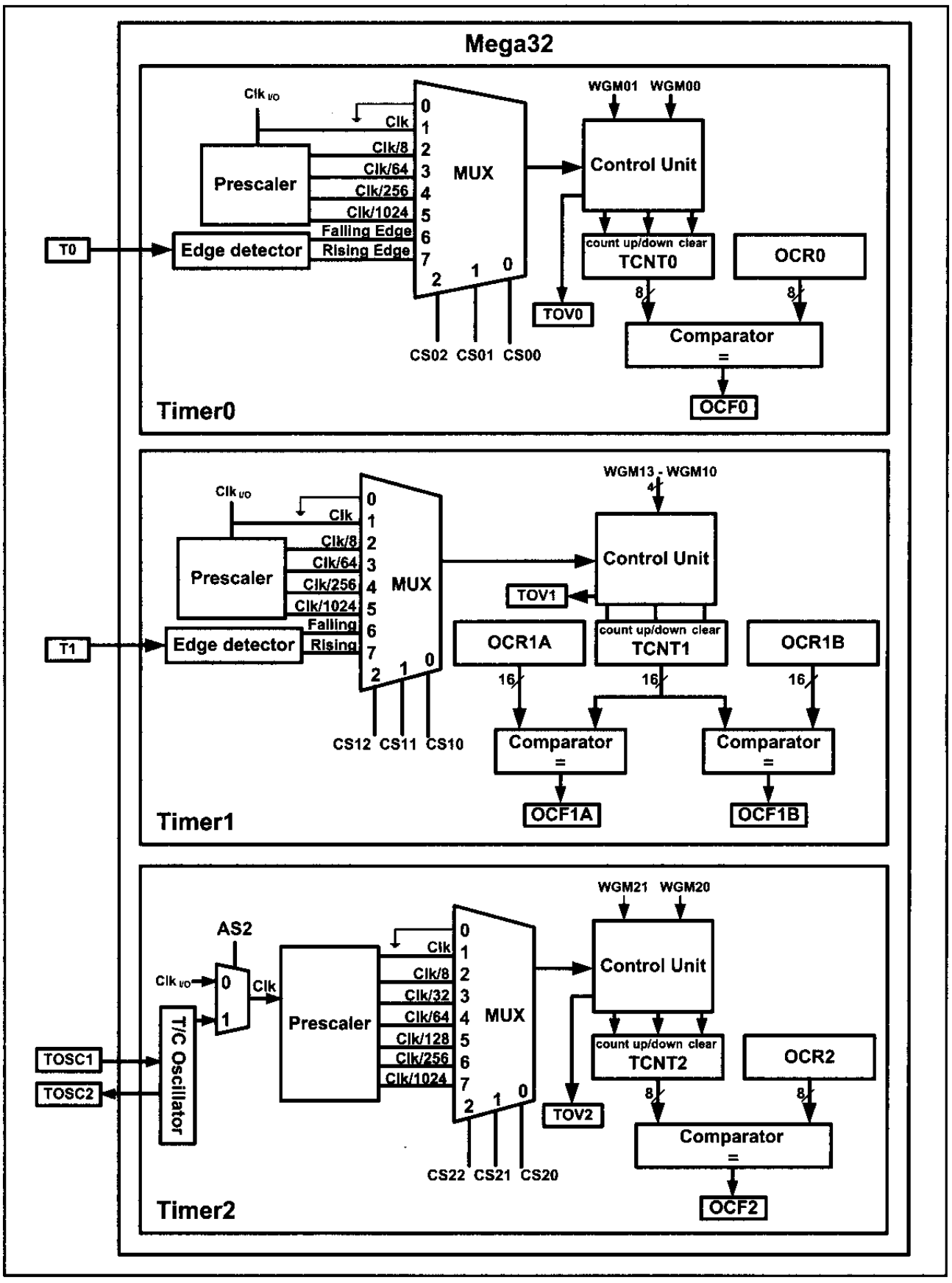


Figure 9-13. Timers in ATmega32

Example 9-24

Using a prescaler of 64, write a program to generate a delay of 1920 μs. Assume XTAL = 8 MHz.

Solution:

Timer clock = 8 MHz/64 = 125 kHz ➔ Timer Period = 1 / 125 kHz = 8 μs ➔ Timer Value = 1920 μs / 8 μs = 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:

Prescaler	Timer Clock	Timer Period	Timer Value
None	8 MHz	1/8 MHz = 0.125 μs	8 ms / 0.125 μs = 64 k
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 μs	8 ms / 1 μs = 8000
32	8 MHz/32 = 250 kHz	1/250 kHz = 4 μs	8 ms / 4 μs = 2000
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 μs	8 ms / 8 μs = 1000
128	8 MHz/128 = 62.5 kHz	1/62.5 kHz = 16 μs	8 ms / 16 μs = 500
256	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 μs	8 ms / 32 μs = **250**
1024	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz= 128 μs	8 ms / 128 μs = **62.5**

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

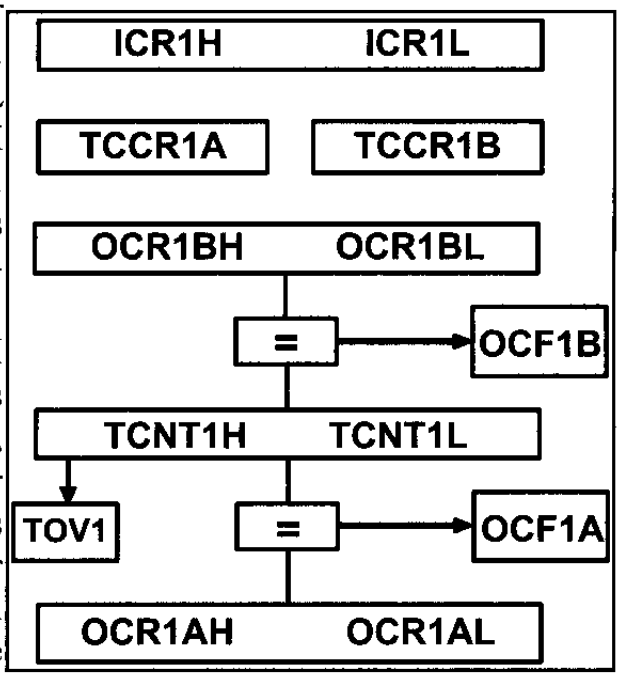


Figure 9-14. Simplified Diagram of Timer1

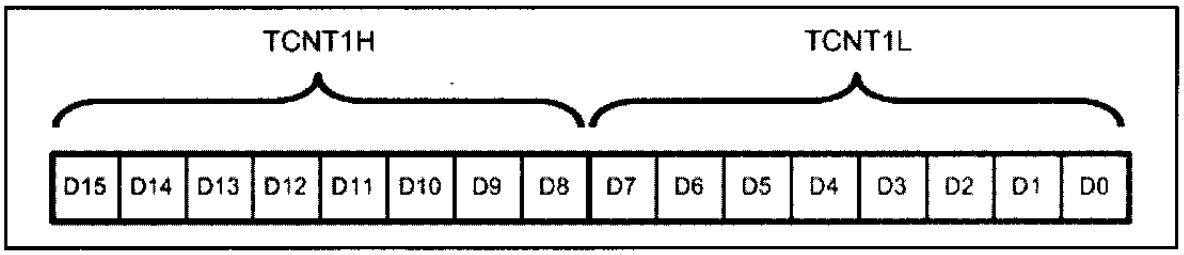


Figure 9-15. Timer1 High and Low Registers

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

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 9-16. 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

COM1A1:COM1A0 D7 D6 Compare Output Mode for Channel A (discussed in Section 9-3)

COM1B1:COM1B0 D5 D4 Compare Output Mode for Channel B (discussed in Section 9-3)

FOC1A D3 Force Output Compare for Channel A (discussed in Section 9-3)

FOC1B D2 Force Output Compare for Channel B (discussed in Section 9-3)

WGM11:10 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	

ICNC1 D7 Input Capture Noise Canceler
0 = Input Capture is disabled.
1 = Input Capture is enabled.

ICES1 D6 Input Capture Edge Select
0 = Capture on the falling (negative) edge
1 = Capture on the rising (positive) edge

D5 Not used

WGM13:WGM12 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

CS12:CS10 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

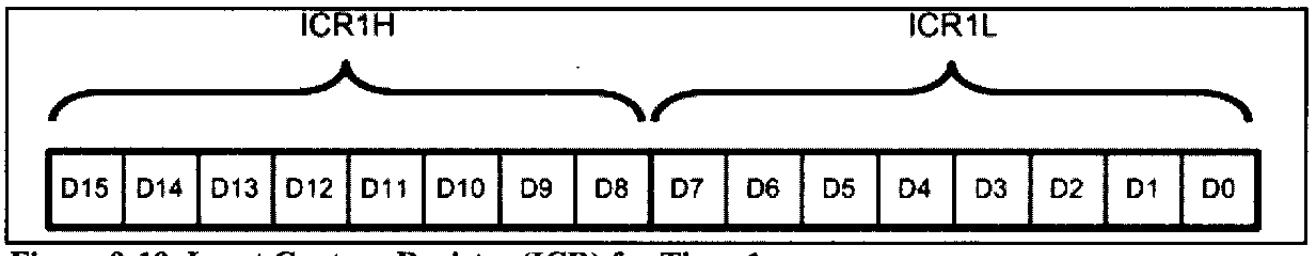


Figure 9-19. Input Capture Register (ICR) for Timer1

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.

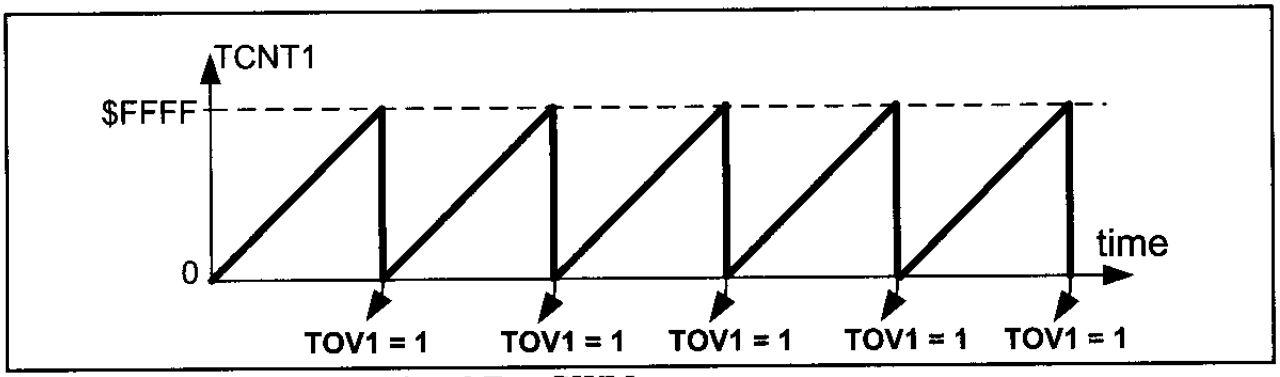


Figure 9-20. TOV in Normal and Fast PWM

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.

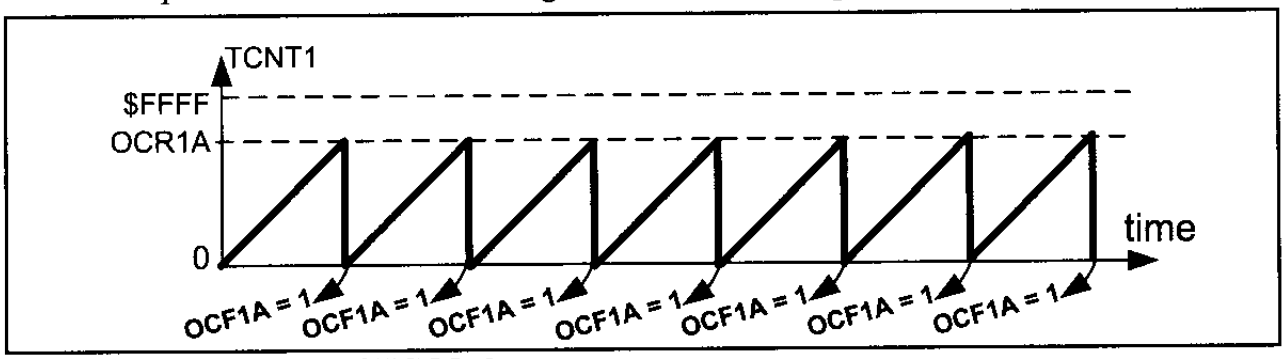


Figure 9-21. OCF1A in CTC Mode

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 μs and delay = 10,000 × 0.125 μs = 1250 μs = 1.25 ms and frequency = 1 / (1.25 ms × 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/(8M) = 0.125 μs.
Delay = 160 × 0.125 μs = 20 μs and frequency = 1 / (20 μs × 2) = 25 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
```

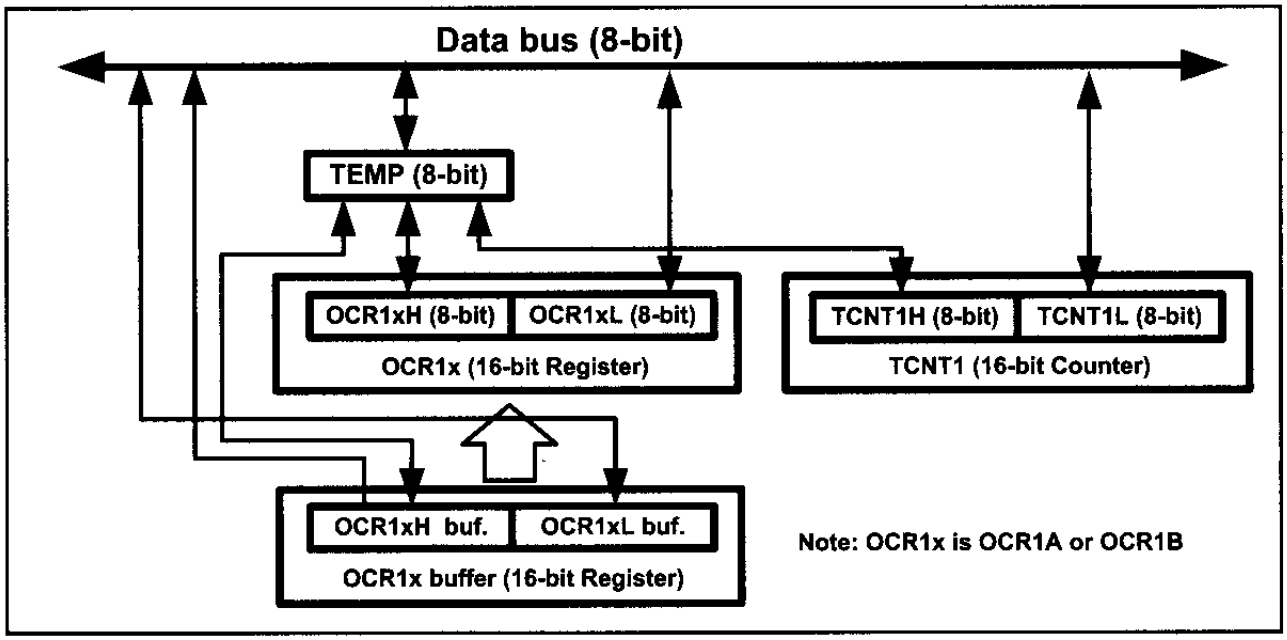


Figure 9-22. Accessing 16-bit Registers through TEMP

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.

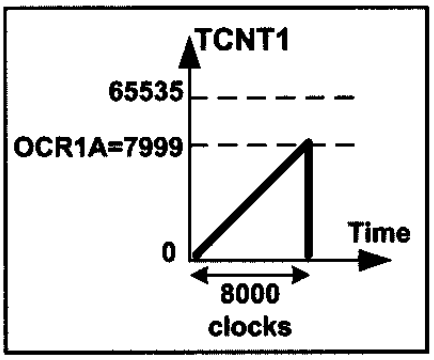
Example 9-30

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

Solution:

XTAL = 8 MHz means that each clock takes 0.125 µs. Now for 1 ms delay, we need 1 ms/0.125 µs = 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
```



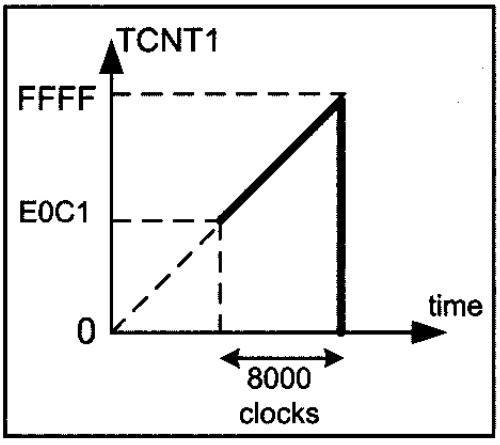
Example 9-31

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
```



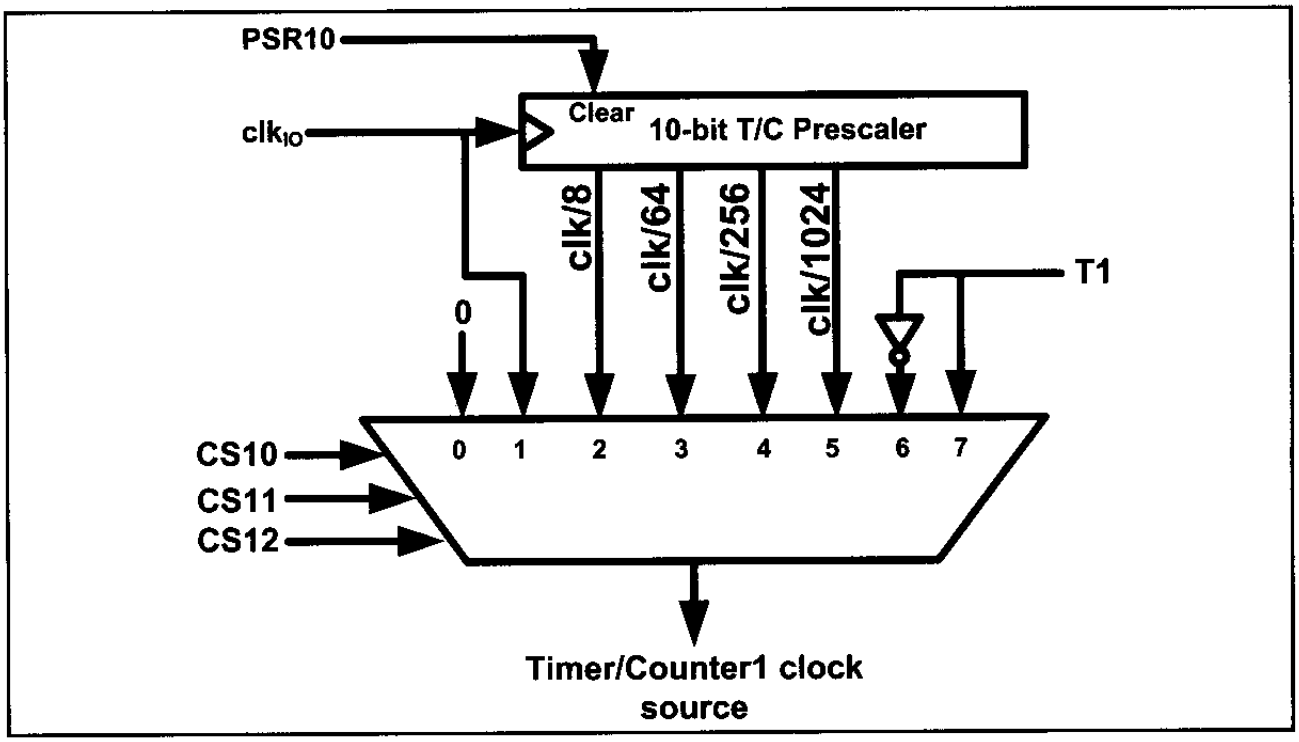


Figure 9-23. Timer/Counter 1 Prescaler

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 μs	1 s/0.125 μs = 8 M
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 μs	1 s/1 μs = 1 M
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 μs	1 s/8 μs = 125,000
256	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 μs	1 s/32 μs = 31,250
1024	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz = 128 μs	1 s/128 μs = 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 Hz = 1 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:

Scaler	Timer Clock	Timer Period	Timer Value
None	8 MHz	1/8 MHz = 0.125 µs	0.5 s/0.125 µs = 4 M
8	8 MHz/8 = 1 MHz	1/1 MHz = 1 µs	0.5 s/1 µs = 500 k
64	8 MHz/64 = 125 kHz	1/125 kHz = 8 µs	0.5 s/8 µs = 62,500
256	8 MHz/256 = 31.25 kHz	1/31.25 kHz = 32 µs	0.5 s/32 µs = 15,625
1024	8 MHz/1024 = 7.8125 kHz	1/7.8125 kHz = 128 µs	0.5 s/128 µs = 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

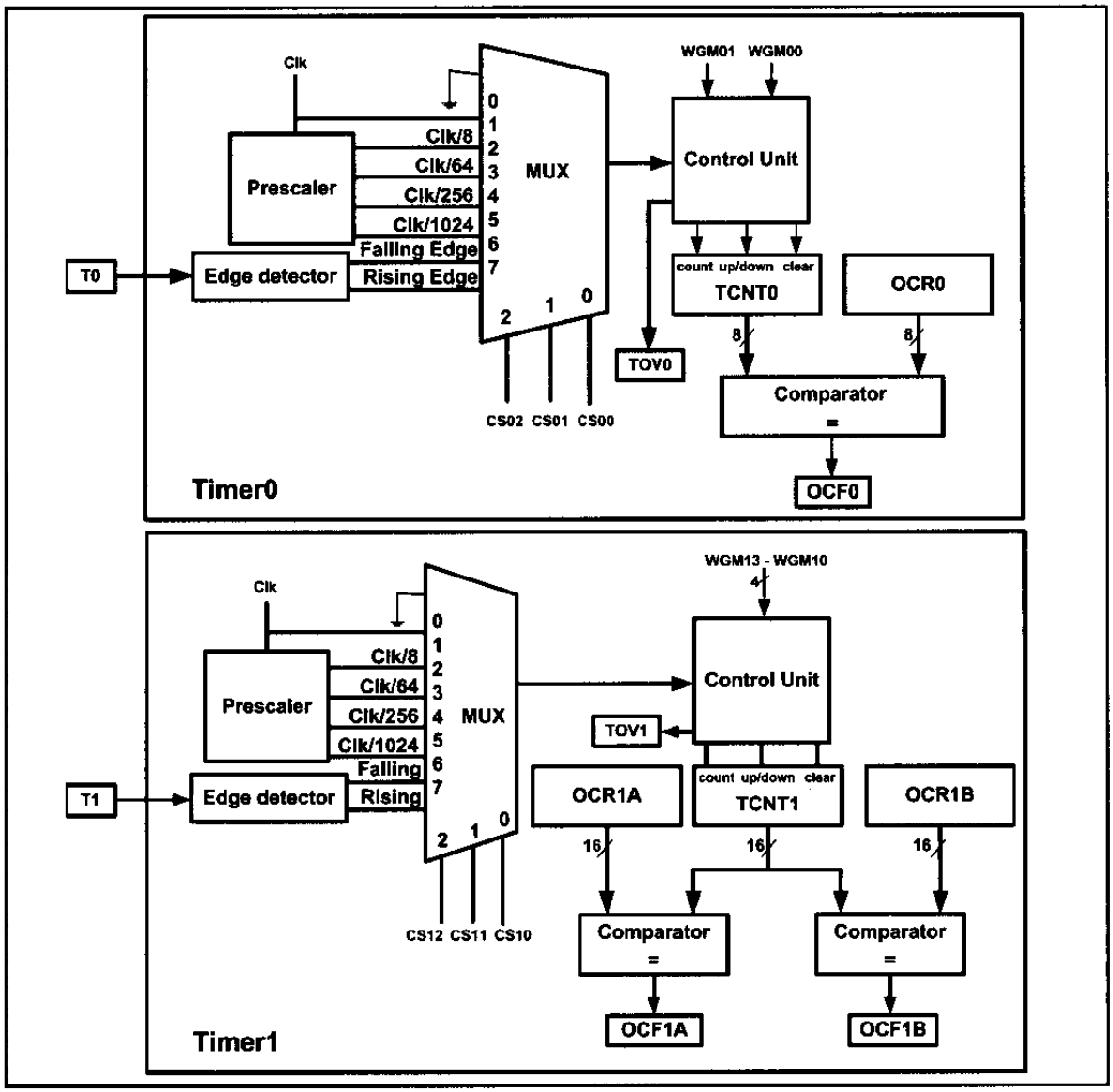


Figure 9-24. Timer/Counters 0 and 1 Prescalers

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.

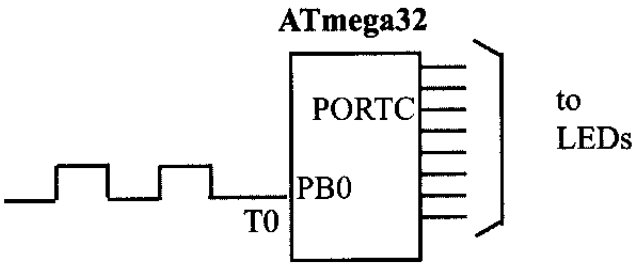
Example 9-35

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.

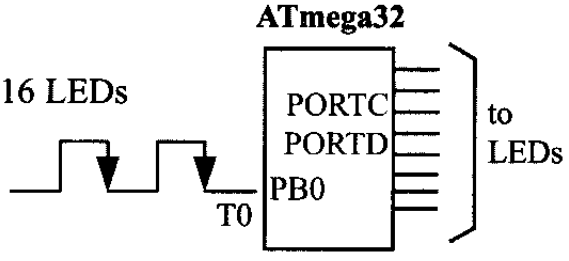
Example 9-36

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.

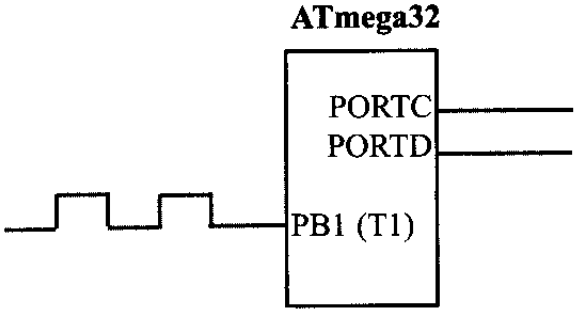


Example 9-37

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

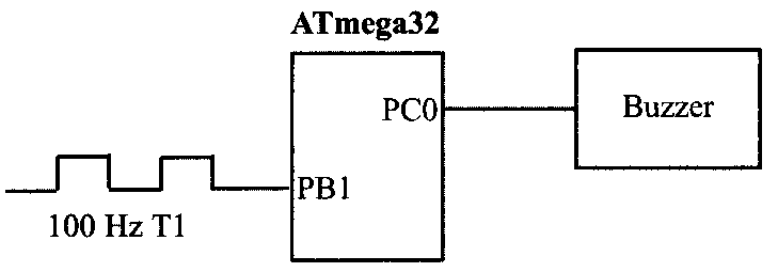
Assuming that clock pulses are fed into pin T1 (PB1) and a buzzer is connected to pin PORTC.0, write a program for Counter 1 in CTC mode to sound the buzzer every 100 pulses.

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 primorily 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 μs. Use Timer0, Normal mode, and 1:8 prescaler to create the delay. Assume XTAL = 8 MHz.

Solution:

XTAL = 8MHz ➔ $T_{machine\ cycle}$ = 1/8 MHz
Prescaler = 1:8 ➔ T_{clock} = 8 × 1/8 MHz = 1 μs
70 μs/1 μs = 70 clocks ➔ 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 ➔ $T_{machine\ cycle}$ = 1/8 MHz = 0.125 µs
Prescaler = 1:1 ➔ T_{clock}= 0.125 µs
2 ms/0.125 µs = 16,000 clocks = 0x3E80 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:

XTAL = 8 MHz ➔ $T_{machine\ cycle}$ = 1/8 MHz = 0.125 µs = T_{clock}
Prescaler = 1:256 ➔ T_{clock} = 256 × 0.125 µs = 32 µs
1 s/32 µs = 31,250 clocks = 0x7A12 clocks ➔ 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.

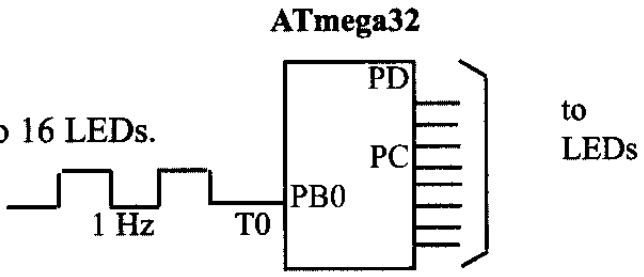
Example 9-43 (C version of Example 9-36)

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.

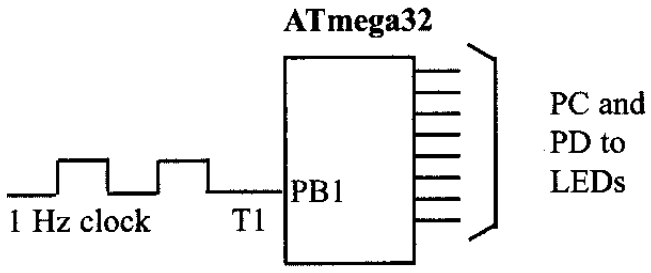


Example 9-44 (C version of Example 9-37)

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 μs. 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 × 8) = 25536 = 63C0, TCNT1H = 0x64 and TCNT1L = 0xC0
8. XTAL = 1 MHz ➔ $T_{machine\ cycle}$ = 1/1 M = 1 μs ➔ 20 μs / 1 μs = 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 "`SBRS 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.

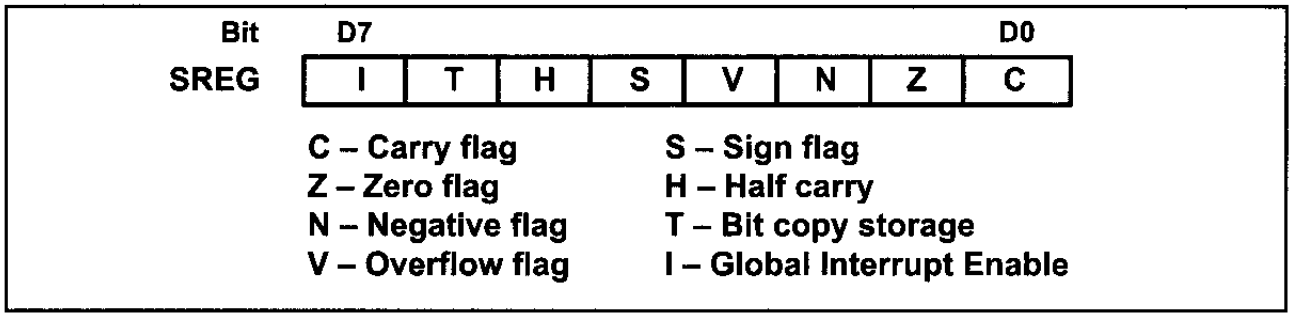


Figure 10-2. Bits of Status Register (SREG)

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

TOIE0 Timer0 overflow interrupt enable
= 0 Disables Timer0 overflow interrupt
= 1 Enables Timer0 overflow interrupt

OCIE0 Timer0 output compare match interrupt enable
= 0 Disables Timer0 compare match interrupt
= 1 Enables Timer0 compare match interrupt

TOIE1 Timer1 overflow interrupt enable
= 0 Disables Timer1 overflow interrupt
= 1 Enables Timer1 overflow interrupt

OCIE1B Timer1 output compare B match interrupt enable
= 0 Disables Timer1 compare B match interrupt
= 1 Enables Timer1 compare B match interrupt

OCIE1A Timer1 output compare A match interrupt enable
= 0 Disables Timer1 compare A match interrupt
= 1 Enables Timer1 compare A match interrupt

TICIE1 Timer1 input capture interrupt enable
= 0 Disables Timer1 input capture interrupt
= 1 Enables Timer1 input capture interrupt

TOIE2 Timer2 overflow interrupt enable
= 0 Disables Timer2 overflow interrupt
= 1 Enables Timer2 overflow interrupt

OCIE2 Timer2 output compare match interrupt enable
= 0 Disables Timer2 compare match interrupt
= 1 Enables Timer2 compare match interrupt

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)
```

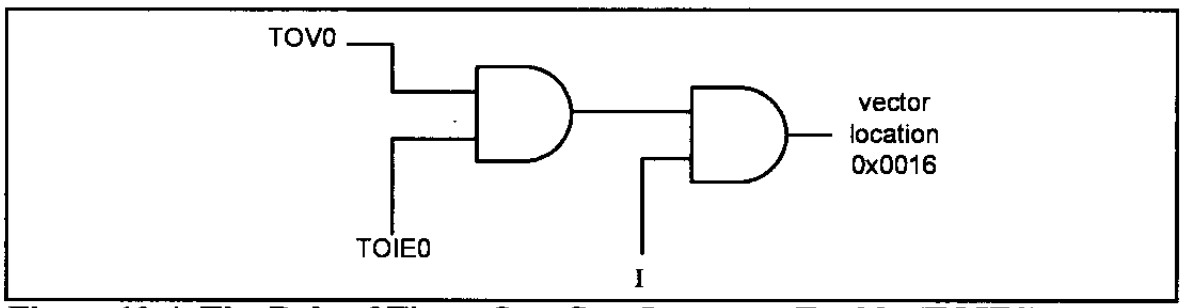


Figure 10-4. The Role of Timer Overflow Interrupt Enable (TOIE0)

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 µs
      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.

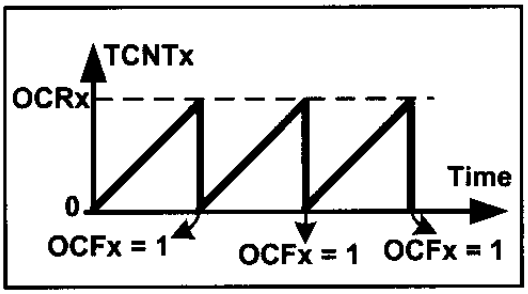


Figure 10-5. CTC Mode

Example 10-3

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

Solution:

1/1 MHz = 1 µs and 40 µs/1 µs = 40. That means we must have 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 µs)
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_{Clock}= (1 / 8 MHz) × 1024 = 128 μs and 1 s/128 μs = 7812. That means we must have 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 AVRs. 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.

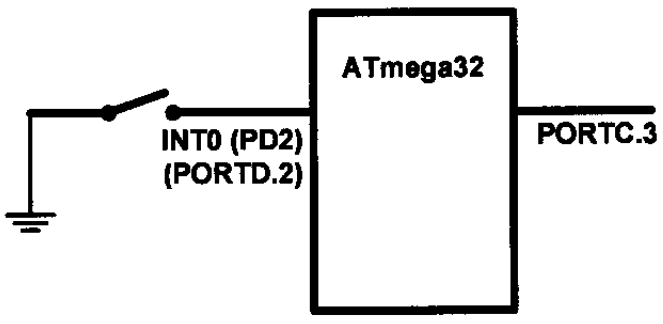
Figure 10-6. GICR (General Interrupt Control Register) Register

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.

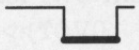
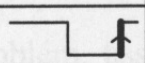
Solution:

```
.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.

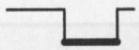
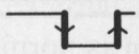
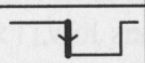
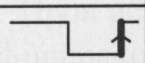
ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

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.

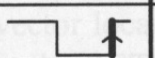
ISC2		Description
0		The falling edge of INT2 generates an interrupt request.
1		The rising edge of INT2 generates an interrupt request.

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,

Bit	D7							D0
	INTF1	INTF0	INTF2	-	-	-	-	-

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 μs

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

tents of PORTC continuously, and (2) increasing the content of PORTD once every 5 μs. 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.

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

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 µs
      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 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 Time0 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 MHz = 125 ns → 125 ns × (21 + 1) = 2.75 μs

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.