AVR Interrupts

By
D.BALAKRISHNA,
Research Assistant, IIIT-H

A single microcontroller can serve several devices. There are two methods by which devices receive service from the microcontroller: **interrupts or polling.**

Interrupts Vs 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.

www.sakshieducation.com

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 below

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

Fig: Interrupt vector table for ATmega 32

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).

www.sakshieducation.com

3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
4. Upon executing the RETI Instruction, the microcontroller returns to the place where it was interrupted.
    - First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC.
    - Then it starts to execute from that address.

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

Sources of interrupts in the AVR:

There are many sources of interrupts in the AVR, depending on which peripheral is incorporated into the chip.

The following are some of the most widely used sources of interrupts in the AVR:

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

Enabling and disabling an interrupt:

Upon reset, all interrupts are disabled (masked). The interrupts must be enabled (unmasked) by software in order for the microcontroller to respond to them.

www.sakshieducation.com

The D7 bit of the SREG (Status Register) register is responsible for enabling and disabling the interrupts globally. The I-bit makes the job of disabling all the interrupts easy. With a single instruction “CLI” (Clear Interrupt), we can make I = 0 during the operation of a critical task.

Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
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

I - Global Interrupt Enable
C - carry flag
N - Negative Flag
S - Sign Bit, S = N EXOR V

T - Bit Copy Storage
Z - Zero Flag
V - Two’s Complement Overflow Flag
H - Half Carry Flag

Fig: Status Register of ATmega 32

- Bit D7 (1) of the SREG register must be set to HIGH to allow the interrupts to happen.
  - This is done with the “SEI” (Set Interrupt) instruction.
- 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.
  - It must be noted that if I =0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high.

External Interrupts

Three external hardware interrupts are there in ATmega AVR.

- Pins PD2 (PORTD.2),
- PD3 (PORTD.3),
- PB2 (PORTB.2)

These are for the external hardware interrupts INT0, INT1, and INT2, respectively

www.sakshieducation.com

PROGRAMMING EXTERNAL HARDWARE INTERRUPTS:

The number of external hardware interrupt interrupts varies in different AVRs. There are three external hardware interrupts in the ATmega32: **INTO, INT1, and INT2.**

They are located on pins PD2, PD3, and PB2, respectively. As shown in Table below, the interrupt vector table locations $2, $4, and $6 are set aside for INTO, INT1, and LNT2, respectively.

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

Fig: Interrupt vector table.

The hardware interrupts must be enabled before they can take effect. These interrupts are controlled by the following registers.

- GICR
- GIFR
- MCUCR
- MCUCSR

www.sakshieducation.com

GICR:

Bit Number	7	6	5	4	3	2	1	0
GICR	INT1	INT0	INT2	—	—	—	IVSEL	IVCE
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

INT0:

When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit) the External **Interrupt 0 is enabled**. The ISC01 and ISC00 of MCUCR register control the interrupt when to be activated (i.e. on rising edge or falling edge or level sensed).

The **INTO 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 TSR.

INT1: When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit) the External Interrupt 1 is enabled. The ISC11 and ISC10 of MCUCR register control the interrupt when to be activated (i.e. on rising edge or falling edge or level sensed).

INT2: When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit) the External Interrupt 2 is enabled. ISC2 bit of MCUCSR register control the interrupt when to be activated (i.e. on rising edge or falling edge).

There are 2 types of activation for the external hardware interrupts.

- Level triggered
- Edge triggered

INT0 & INT1 can be edge or level triggered, but INT2 can be edge triggered only.

The MCUCR & MCUCSR registers decides the triggering options of the external hardware interrupts INT0, INT1, and INT2.

www.sakshieducation.com

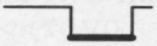
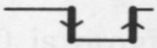
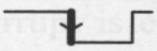
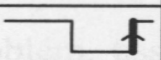
MCUCR:

This register decides the triggering options of the external hardware interrupts INT0 and INT1.

Bit Number	7	6	5	4	3	2	1	0
MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
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

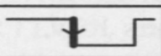
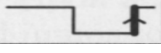
ISC01 & ISC00 (Interrupt Sense Control Bits):

These bits define the level or edge on the external **INT0** pin that activate the interrupt as shown in table below.

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 on the external **INT1** pin that activate the interrupt as shown in table below.

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.

MCUSCR:

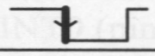
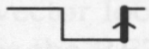
This register decides the triggering options of the external hardware interrupt INT2.

www.sakshieducation.com

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

ISC2: This bit controls the INT2 interrupt trigger condition.

- ISC2 = 0: the interrupt is detected on falling edge.
- ISC2 = 1: the interrupt is detected on rise edge.

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

GIFR:

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.

Bit Number	7	6	5	4	3	2	1	0
GIFR	INTF1	INTF0	INTF2	–	–	–	–	–
Read/Write	R/W	R/W	R/W	R	R	R	R	R
Initial Value	0	0	0	0	0	0	0	0

If the interrupt is active (the INTx bit is set and the I-bit in SREG is one), the AVR will jump to the corresponding interrupt vector location and the INTFx flag will be cleared automatically, otherwise, the flag remains set. The flag can be cleared by writing a one to it.

In other words

- **INTF1:** When '1' on this bit trigger INT1 Interrupt when INT1 bit of GICR and I bit of SREG is one.
- **INTF0:** When '1' on this bit trigger INT0 Interrupt when INT0 bit of GICR and I bit of SREG is one.
- **INTF2:** When '1' on this bit trigger INT2 Interrupt when INT2 bit of GICR and I bit of SREG is one.

www.sakshieducation.com

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.

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, extern al 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 current interrupt.

When the RETI instruction is execute d, the AVR enables the I-bit, causing the other interrupts are 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.

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.

www.sakshieducation.com

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 the 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**).

INTERRUPT PROGRAMMING IN C:

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

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>

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.

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 shown below.

For example, the following TSR serves the Timer0 compare match interrupt:

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


www.sakshieducation.com

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

Fig: interrupt Vector Names for WinAVR

Example 1:

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/irtterrupt.h>
int main ()
```


www.sakshieducation.com

```
{
        DDRC = 1<<3;    //PC3 as an output
        PORTD = 1<<2;   //pull-up activated
        GICR = (1<<INTO);    //enable external interrupt 0
        sei ();                    //enable interrupts
        while (1);                 //wait here
}
ISR (INT0_vect)          //ISR for external interrupt 0
{
        PORTC ^= (l<<3);       //toggle PORTC.3
}
```

Example 2:

Rewrite Example 1, so that whenever INT0 goes low, it toggles PORTC.3 only once.

Solution:

```
#include <avr/io.h>
#include <avr/irtterrupt.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<<INTO);       //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
}
```

www.sakshieducation.com

TIMER Interrupts

AVR Timers:

AVR timers have a lot of complex uses, but their essential purpose is to measure time.

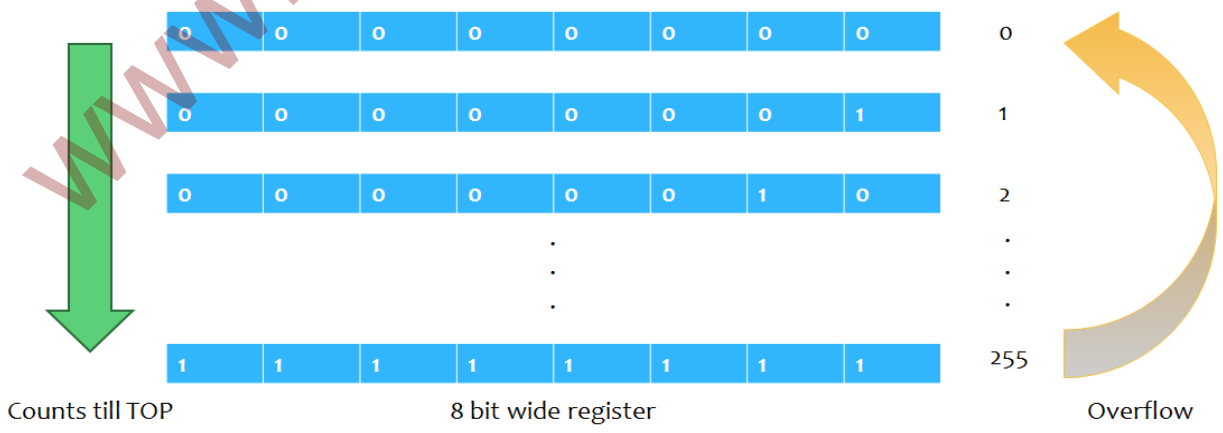
They work in an asynchronous manner, i.e. they run parallel to the microcontroller's core code. This is possible only because the timers have a separate circuit for their function.

The smallest amount of time that a timer can measure is determined by the frequency of the clock source which the microcontroller uses. For example if the microcontroller uses a 4MHz crystal as the clock source, then the smallest time it can measure is 1/4000000th of a second.

Timers as registers:

So basically, a timer is a register, but not a normal one. The value of this register increases/decreases automatically.

- In AVR, timers are of two types:
  - 8-bit and 16-bit timers.
- In an 8-bit timer, the register used is 8-bit wide
- In 16-bit timer, the register width is of 16 bits.
- This means that the 8-bit timer is capable of counting 2^8=256 steps from 0 to 255



www.sakshieducation.com

- 16 bit timer is capable of counting 2^16=65536 steps from 0 to 65535.
- Due to this feature, **timers are also known as counters**.
- Once they reach their MAX value it returns to its initial value of zero.
    - We say that the timer/counter **overflows**.
    - Shown in above figure.

In ATMEGA32, we have three different kinds of timers:

- **TIMER0** - 8-bit timer
- **TIMER1** – 16-bit timer
- **TIMER2** – 8-bit timer

The timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention.

Apart from normal operation, these three timers can be either operated in

- **Normal** mode
- **CTC** mode
- **PWM** mode

Timer Concepts:

Basic Concepts: We know the following formula:

$$Time\ Period = \frac{1}{Frequency}$$

Now let's assume that we have an external crystal XTAL of 4 MHz, Hence, the CPU clock frequency is 4 MHz

- As we discussed that the timer counts from 0 to TOP.
- For an 8-bit timer, it counts from 0 to 255
- For a 16-bit timer it counts from 0 to 65535.
- After that, they overflow.

www.sakshieducation.com

Let's the timer's value is zero now.

To go from 0 to 1, it takes one clock pulse. To go from 1 to 2, it takes another clock pulse. To go from 2 to 3, it takes one more clock pulse. And so on.

For F_CPU = 4 MHz, time period T = 1/4M = 0.00025 ms. Thus for every transition (0 to 1, 1 to 2, etc), it takes *only* 0.00025 ms

Let us assume we need a delay of 10 ms. This maybe a very short delay, but for the microcontroller which has a resolution of 0.00025 ms, it's quite a long delay.

To get an idea of *how long* it takes, let's calculate the timer count from the following formula:

$$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$

Substitute *Required Delay = 10 ms* and *Clock Time Period = 0.00025 ms*, and we will get***Timer Count = 39999***

Now, to achieve this, we definitely cannot use an 8-bit timer (as it has an upper limit of 255, after which it overflows). Hence, we use a 16-bit timer (which is capable of counting up to 65535) to achieve this delay.

To achieve this, we cannot use an 8-bit timer (as it has an upper limit of 255). Hence, we use a 16-bit timer (which is capable of counting up to 65535) to achieve this delay.

The Prescaler:

Assuming F_CPU = 4 MHz and a 16-bit timer (MAX = 65535), and substituting in the above formula, we can get a maximum delay of 16.384 ms.

Now what if we need a greater delay,

Example: For 20 ms.

Suppose if we decrease the F_CPU from 4 MHz to 0.5 MHz (i.e. 500 kHz), then the clock time period increases to 1/500k = 0.002 ms.*Now* if we substitute *Required Delay = 20 ms* and *Clock Time Period = 0.002 ms*, we get

www.sakshieducation.com

Timer Count = 9999. As we can see, this can easily be achieved using a 16-bit timer. At this frequency, a maximum delay of 131.072 ms can be achieved.

This technique of frequency division is called ***prescaling***. We do not reduce the actual F_CPU. The actual F_CPU remains the same (at 4 MHz in this case). So basically, we *derive* a frequency from it to run the timer. Thus, while doing so, we divide the frequency and use it.There is a provision to do so in AVR by setting some bits which we will discuss later.

We cannot use prescaler freely.**There is a trade-off between resolution and duration**.The resolution has also increased from 0.00025 ms to 0.002 ms. this means each tick will take 0.002 ms that causes reduction of accuracy.

Choosing Prescalers:

The AVR offers us the following prescaler values to choose from: 8, 64, 256 and 1024. A prescaler of 8 means the effective clock frequency will be F_CPU/8.

Now substituting each of these values into the above formula, we get different values of timer value.

Let us assume required delay: 184 ms and F_CPU: 4 M Hz.

The results are summarized as below:

Required Delay = 184 ms
F_CPU = 4 MHz

Prescaler	Clock Frequency	Timer Count
8	500 kHz	91999
64	62.5 kHz	11499
256	15.625 kHz	2874
1024	3906.25 Hz	717.75

Now out of these four prescalers, 8 cannot be used as the timer value exceeds the limit of 65535. Also, since the timer always takes up integer values, we cannot choose 1024 as the timer count is a decimal digit. Hence, we see that prescaler values of 64 and 256 are feasible. But out of these two, we choose 64 as it provides us with greater resolution. We can choose 256 if we need the timer for a greater duration elsewhere.

www.sakshieducation.com

Thus, we always choose prescalar which gives the counter value within the feasible limit (255 or 65535) **and the counter value should always be an integer.**

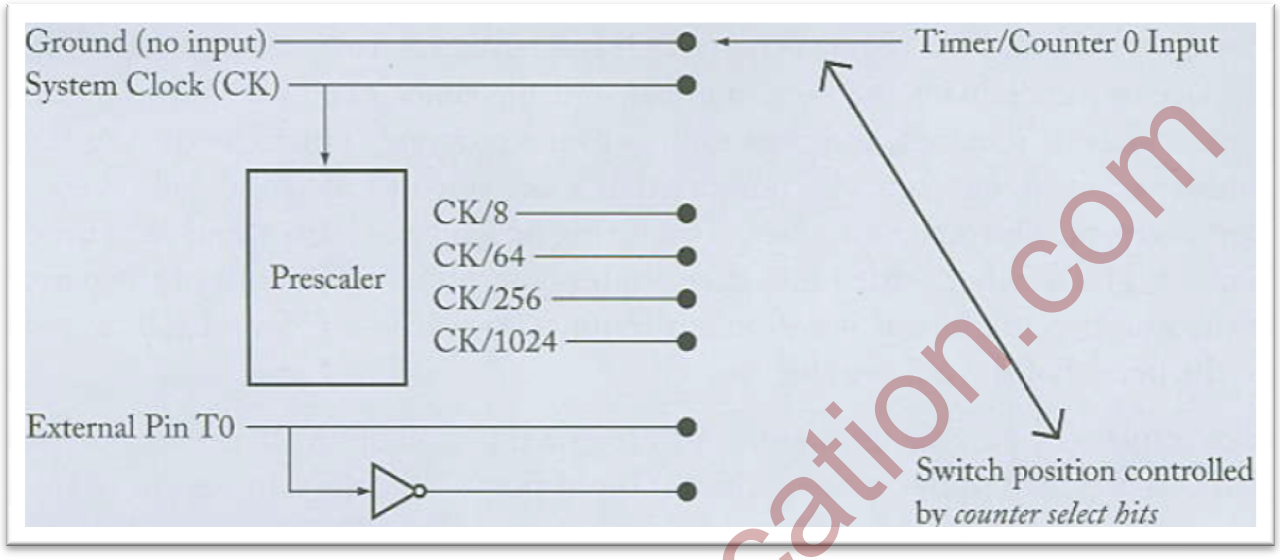


Fig: Timer0 Prescalar/ Selector

AVR Timers – TIMER0:

Since timer is a peripheral, it can be activated by setting some bits in some registers.

TCNT0 Register:

The **Timer/Counter Register,** shown in figure below

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
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	

Fig: TCNT0 Register

The value of the counter is stored here and increases/decreases automatically. Data can be both read/written from this register.

www.sakshieducation.com

Now we know where the counter value lies. But this register won't be activated unless we activate the timer! Thus we need to set the timer up by using Timer Counter Control Register.

TCCR0 Register:

The **Timer/Counter Control Register,**shown in figure below

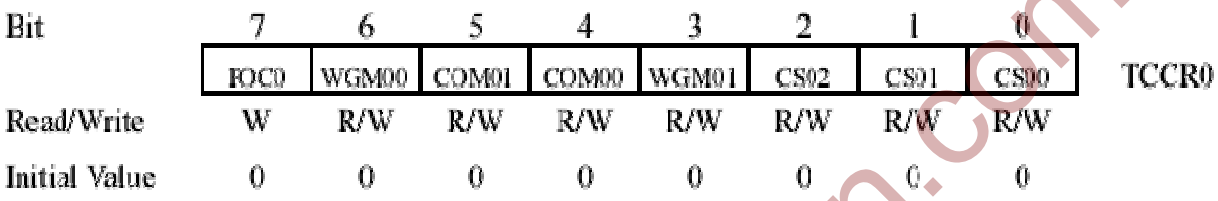


Fig: TCCR0 Register

- **Clock Select Bits (CS 02: 00):** Used toset the timer up by choosing proper prescaler. The possible combinations are shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$Clk_{I/O}$/1 (No prescaling)
0	1	0	$Clk_{I/O}$/8 (From prescaler)
0	1	1	$Clk_{I/O}$/64 (From prescaler)
1	0	0	$Clk_{I/O}$/256 (From prescaler)
1	0	1	$Clk_{I/O}$/1024 (From prescaler)
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.

Fig: TCCR0 PrescalerDefinitions

- Example code:
  - **TCCR0 |= (1 << CS00);** // Initializing the counter in **Noprecaling**
  - **TCCR0 |= (1 << CS02)|(1 << CS00);** // set up timer with prescaler = 1024

www.sakshieducation.com

- **Bit 6,3 – WGM01:00 – Wave Generation Mode** – Just like in TIMER1, we choose the type of wave mode from here as follows

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

Fig: Wave Generation Mode Bit Description

- **Bit 5:4 – COM01:00 – Compare Match Output Mode –**
  - Controls the behavior of the OC0 (PB3) pin depending upon the WGM mode –
    - *non-PWM,*
    - *Phase Correct PWM mode and*
    - *Fast PWM mode*.
  - The selection options of non-PWM mode are as follows.

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Fig: Compare Output Mode, non-PWM

- **Bit 7 – FOC0 – Force Output Compare –**
  - When set to ‘1’
    - Forces an immediate compare match and affects the behavior of OC0 pin.
  - When clear to ‘0’
    - To ensure compatibility with future devices, this bit must be set to ‘0’.

www.sakshieducation.com

OCR0 Register:

The **Output Compare Register**– OCR0 Register is shown in figure below.

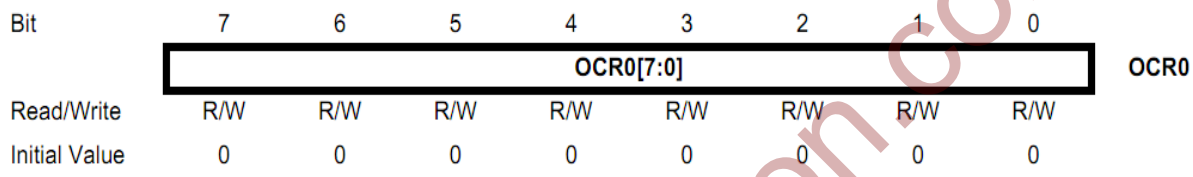


Fig: OCR0 Register

The value to be compared (max 255) is stored in this register.

TIMSK Register:

The **Timer/Counter Interrupt Mask**Register, shown in figure below:

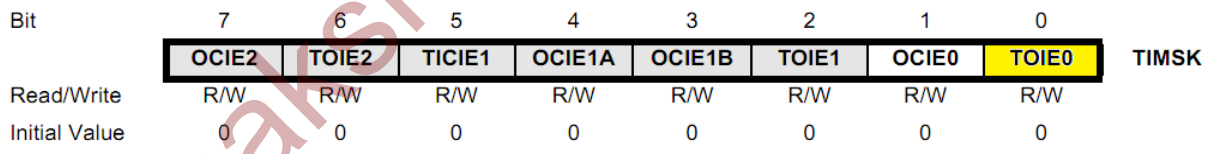


Fig: TIMSK Register

It is a common register for all the three timers.

Bits (1:0):

- Correspond to TIMER0
- **Bit0:**
  - Setting the bit**TOIE0** to ’1′ enables the TIMER0 overflow interrupt.
- **Bit 1:**

www.sakshieducation.com

- **OCIE0 – Timer/Counter0 Output Compare Match Interrupt Enable**
  - Enables the firing of interrupt whenever a compare match occurs.

Bits (5:2):

- Correspond to TIMER1.
- **Bit 2 – TOIE1 – Timer/Counter1 Overflow Interrupt Enable** bit
  - Enables the overflow interrupt of TIMER1.
- Other bits are related to CTC mode
  - **Bit 4:3 –OCIE1A: B – Timer/Counter1, Output Compare A/B Match Interrupt Enable** bits.
    - Enabling it ensures that an interrupt is fired whenever a match occurs.
    - Since there are two CTC channels, we have two different bits OCIE1A and OCIE1B for them.
  - **Bit 5- TICIE1 Timer 1 Input Capture Interrupt Enable**
    - TICIE1= 0 Disables Timerl input capture interrupt
    - TICIE1= I Enables Timer 1 input capture interrupt

Bits (7:6):

- Correspond to TIMER2
- Setting the bit **TOIE2** to ’1′ enables the TIMER0 overflow interrupt.
- **OCIE0 – Timer/Counter0 Output Compare Match Interrupt Enable**
  - Enables the firing of interrupt whenever a compare match occurs.

TIFR Register:

www.sakshieducation.com

The **Timer/Counter Interrupt Flag Register,** shown in figure below.

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

Fig: TIFR Register

This is also a register shared by all the timers.

Bits (1:0):

- Correspond to TIMER0
- **Bit 0:**

o **TOV0(Timer/Counter1 Overflow Flag)** bit is set (one) whenever TIMER0 overflows.

▪ This bit is reset (zero) whenever the Interrupt Service Routine (ISR) is executed.

▪ If there is no ISR to execute, we can clear it manually by writing one to it.

- **Bit 1**:

o **OCF0 – Output Compare Flag 0**

▪ Sets whenever a compare match occurs.

▪ It is cleared automatically whenever the corresponding ISR is executed.

▪ Alternatively it is cleared by writing '1' to it.

Bits (5:2):

- Correspond to TIMER1.
- **Bit 2 – TOV1 – Timer/Counter1 Overflow Flag** bit is set to 1 whenever the timer overflows

o This bit is reset (zero) whenever the Interrupt Service Routine (ISR) is executed.

o If there is no ISR to execute, we can clear it manually by writing one to it.

www.sakshieducation.com

- **Bit 4:3 – OCF1A: B – Timer/Counter1, Output Compare A/B Match Flag Bit**.

o This bit is set (one) by the AVR whenever a match occurs

- TCNT1 becomes equal to OCR1A (or OCR1B).
- It is cleared automatically whenever the corresponding Interrupt Service Routine (ISR) is executed.
- Alternatively, it can be cleared by writing '1′ to it!

Bits (7:6):

- Correspond to TIMER2
- **TOV2** bit is set (one) whenever TIMER2 overflows.

o This bit is reset (zero) whenever the Interrupt Service Routine (ISR) is executed.

o If there is no ISR to execute, we can clear it manually by writing one to it.

Example: (Without Using interrupts)

To flash an LED every 8 ms and we have an XTAL of 16 MHz. We can use a prescaler of 1024. Now refer to the descriptions of clock select bits as shown in the TCCR0 register.

Code:

```
#include <avr/io.h>
void timer0_init()
{
    // set up timer with prescaler = 1024
    TCCR0 |= (1 << CS02)|(1 << CS00);
    // initialize counter
    TCNT0 = 0;
}
int main(void)
{
    // connect led to pin PC0
    DDRC |= (1 << 0);
    // initialize timer
```


www.sakshieducation.com

```
        timer0_init();
        // loop forever
        while(1)
        {
                // check if the timer count reaches 124
                if (TCNT0 >= 124)
                {
                PORTC ^= (1 << 0);    // toggles the led
                TCNT0 = 0;          // reset counter
                }
        }
}
```

Example: (Using Interrupts)

To flash the LED every 50 ms. With CPU frequency 16 MHz,

Even a maximum delay of 16.384 ms can be achieved using a 1024 prescaler.The concept here is that the hardware generates an interrupt every time the timer overflows. Since the required delay is greater than the maximum possible delay, obviously the timer will overflow. And whenever the timer overflows, an interrupt is fired. Now the question is *how many times should the interrupt be fired?*

For this, let's do some calculation. Let's choose a prescaler, say 256. Thus, as per the calculations, it should take 4.096 ms for the timer to overflow. Now as soon as the timer overflows, an interrupt is fired and an Interrupt Service Routine (ISR) is executed. Now,

50 ms ÷ 4.096 ms = 12.207

Thus, in simple terms, by the time the timer has overflown 12 times, 49.152 ms would have passed. After that, when the timer undergoes 13th iteration, it would achieve a delay of 50 ms. Thus, in the 13th iteration, we need a delay of 50 – 49.152 = 0.848 ms. At a frequency of 62.5 kHz (prescaler = 256), each tick takes 0.016 ms. Thus to achieve a delay of 0.848 ms, it would require 53 ticks. Thus, in the 13th iteration, we only allow the timer to count up to 53, and then reset it. All this can be achieved in the ISR as follows:

Code:

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


www.sakshieducation.com

```
#include <avr/interrupt.h>
        // global variable to count the number of overflows
        volatile uint8_t tot_overflow;
        // initialize timer, interrupt and variable
        void timer0_init()
        {
                TCCR0 |= (1 << CS02);// set up timer with prescaler = 256
                TCNT0 = 0;// initialize counter
                TIMSK |= (1 << TOIE0);// enable overflow interrupt
                sei();// enable global interrupts
                tot_overflow = 0;// initialize overflow counter variable
        }
        // TIMER0 overflow interrupt service routine called whenever TCNT0
overflows
        ISR(TIMER0_OVF_vect)
        {
                // keep a track of number of overflows
                tot_overflow++;
        }
        int main(void)
        {
                // connect led to pin PC0
                DDRC |= (1 << 0);
                // initialize timer
                timer0_init();
                 // loop forever
                while(1)
                {
                        // check if no. of overflows = 12
                        if (tot_overflow>= 12)  // NOTE: '>=' is used
                        {
                        // check if the timer count reaches 53
                        if (TCNT0 >= 53)
                        {
                                PORTC ^= (1 << 0);    // toggles the led
                                TCNT0 = 0;          // reset counter
                                tot_overflow = 0;     // reset overflow counter
                        }
                        }
                }
```

www.sakshieducation.com

}

AVR Timers – TIMER1:

In addition to the usual timer/counter, Timer 1 contains one 16 bit input capture register and two 16-bit outputs compare registers.

The input capture register is used for measuring pulse widths or capturingtimes. The output compare registers are used for producing frequencies or pulses from thetimer/counter to an output pin on the microcontroller.

TCNT1 Register:

The **Timer/Counter1 Register,** shown in figure below.

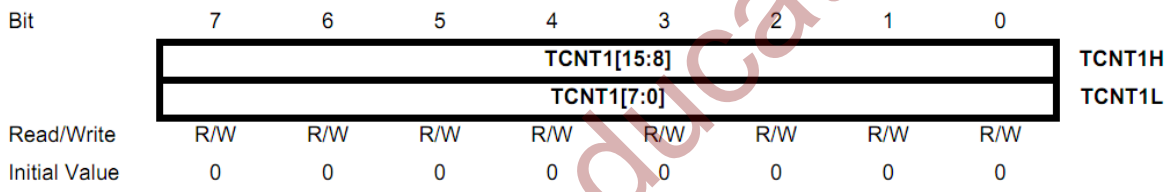


Fig: TCNT1 Register.

It is 16 bits wide since the TIMER1 is a 16-bit register. **TCNT1H** represents the HIGH byte whereas **TCNT1L** represents the LOW byte. The timer/counter value is stored in these bytes.

Timer/counter control register 1 (TCCR1): the ATMegal6 timer /control register for Timer 1 is actually composed oftwo registers, **TCCR1A and TCCR1B.**

TCCR1A controls the compare modes and the pulse width modulation modes of Timer1.

TCCR1B controls the prescaler and input multiplexer for Timer 1, as well asthe input capture modes.

TCCR1A Register:

The **Timer/Counter1 Control RegisterA,**shown in figure below.

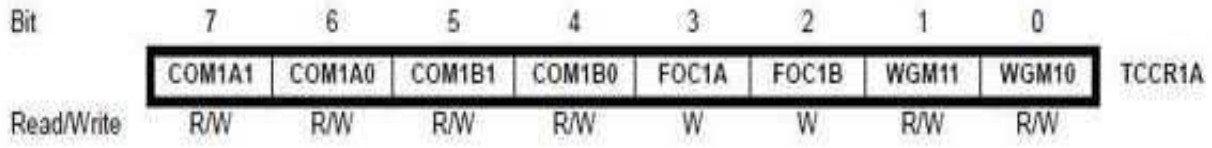


Fig: TCCR1A Register

- The behavior changes depending upon the following modes:
  - Non-PWM mode (normal / CTC mode)
  - Fast PWM mode
  - Phase Correct / Phase & Frequency Correct PWM mode
- **Bit 7:6 – COM1A1:0** and **Bit 5:4 – COM1B1:0**
  - **Compare Output Mode for Compare Unit A/B**.
  - These bits control the behavior of the Output Compare (OC) pins.

COM1A1/ COM1B1	**COM1A0/ COM1B0**	**Description**
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on compare match.
1	0	Clear OC1A/OC1B on compare match (Set output to low level)
1	1	Set OC1A/OC1B on compare match (Set output to high level)

Table: Compare Output Mode, non-PWM

- **Bit 3:2 – FOC1A: B – Force Output Compare for Compare Unit A/B**.
  - These bits are *write only* bits.
  - They are active only in non-PWM mode.
  - For ensuring compatibility with future devices, these bits must be set to zero (which they *already* are by default).

www.sakshieducation.com

- Setting them to '1' will result in an immediate forced compare match and the effect will be reflected in the OC1A/OC1B pins.
- **The thing to be noted is that FOC1A/FOC1B will *not* generate any interrupt, *nor* will it clear the timer in CTC mode.**

TCCR1B Register:

The **Timer/Counter1 Control Register B**shown in figure below.

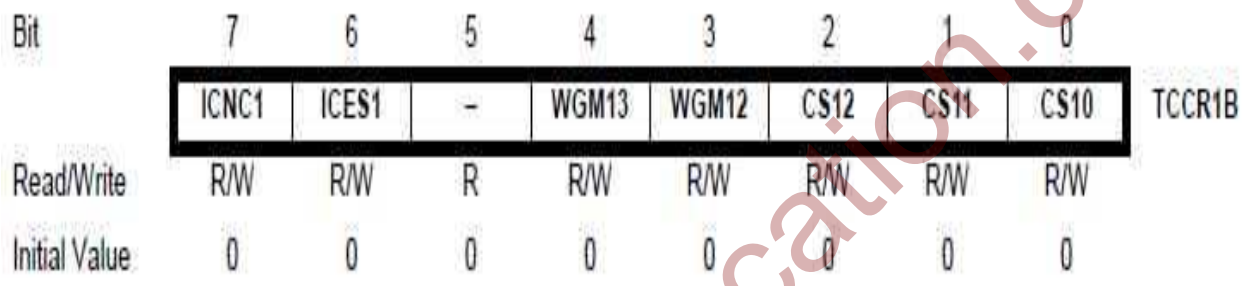


Fig: TCCR1B Register

The **bit 2:0 – CS12:10** are the **Clock Select Bits** of TIMER1. Their selection is as follows.

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$Clk_{I/O}$/1 (No prescaling)
0	1	0	$Clk_{I/O}$/8 (From prescaler)
0	1	1	$Clk_{I/O}$/64 (From prescaler)
1	0	0	$Clk_{I/O}$/256 (From prescaler)
1	0	1	$Clk_{I/O}$/1024 (From prescaler)
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.

Fig: TCCR1B PrescalerDefinitions

Bits 7:6 in TCCR1B:

- **ICNC1(Input Capture Noise Canceller)** (1 = enabled)

www.sakshieducation.com

- **ICES1(Input Capture Edge Select)** (1 = rising edse, 0 = falling edge)

Bits 1:0 in TCCR1A **(WGM11 & WGM11)**and

Bits 4:3 in TCCR1B **(WGM13&WGM12)**are **Wave Generation Mode Bits** which are used to select mode shown in figure below.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	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 & Frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase & 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

Fig:Wave Generation Mode Bit Description

In pin configuration of ATMEGA16/32, we can see the pins PB3, PD4, PD5 and PD7. Their special functions are mentioned in the brackets (OC0, OC1A, OC1B and OC2). These are the Output Compare pins of TIMER0, TIMER1 and TIMER2 respectively shown in figure below.

www.sakshieducation.com

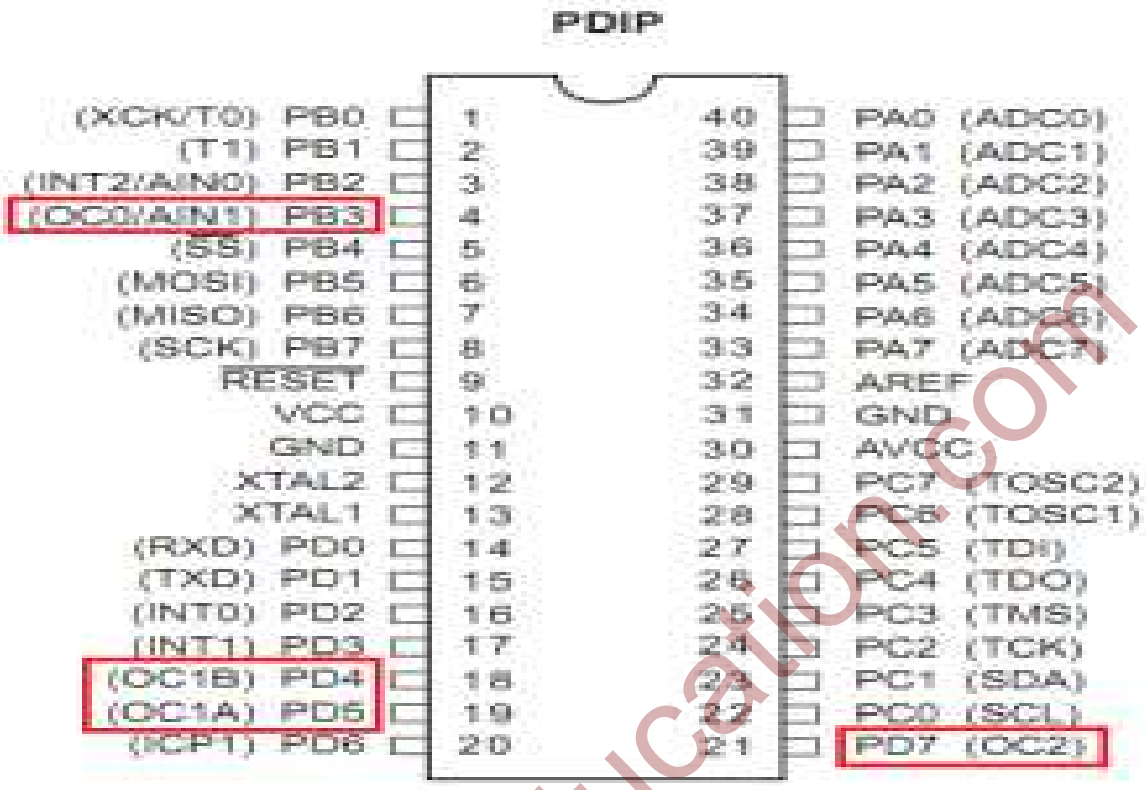


Fig: ATmega 16/ 32 Pin description

OCR1A and OCR1B Registers:

We must tell the AVR to reset the timer as soon as its value reaches *such and such value*. So, the question is, how do we set *such and such values?* The **Output Compare Register 1A** – OCR1A and the **Output Compare Register 1B** – OCR1B are utilized for this purpose.

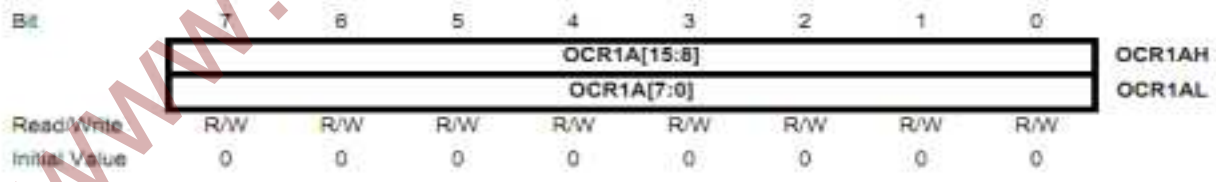


Fig: OCR1A Register

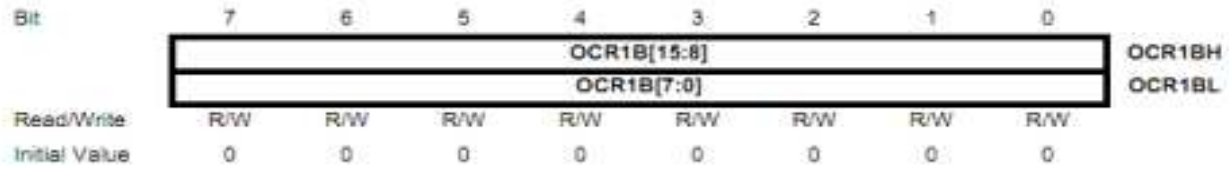


Fig: OCR1B Register

Since the compare value will be a 16-bit value (in between 0 and 65535), OCR1A and OCR1B are 16-bit registers. In ATMEGA16/32, there are two CTC channels – A and B. We can use any one of them or both. Let's use OCR1A.

Example:

OCR1A = 24999; // timer compare value

Example:

To flash an LED every 2 seconds, i.e. at a frequency of 0.5 Hz. We have an XTAL of 16 MHz.

Code:

```
#include <avr/io.h>
#include <avr/interrupt.h>
    // global variable to count the number of overflows
    volatile uint8_t tot_overflow;
    // initialize timer, interrupt and variable
    void timer1_init()
    {
        // set up timer with prescaler = 8
        TCCR1B |= (1 << CS11);
        // initialize counter
        TCNT1 = 0;
      // enable overflow interrupt
        TIMSK |= (1 << TOIE1);
       sei();// enable global interrupts
      // initialize overflow counter variable
        tot_overflow = 0;
    }
    // TIMER1 overflow interrupt service routine called whenever TCNT1 overflows
    ISR(TIMER1_OVF_vect)
```

```
{
        // keep a track of number of overflows
            tot_overflow++;
            // check for number of overflows here itself
            // 61 overflows = 2 seconds delay (approx.)
            if (tot_overflow >= 61) // NOTE: '>=' used instead of '=='
            {
                    PORTC ^= (1 << 0);  // toggles the led
                // no timer reset required here as the timer is reset every
                time it overflows
                    tot_overflow = 0;   // reset overflow counter
            }
}
int main(void)
{
        // connect led to pin PC0
            DDRC |= (1 << 0);
            timer1_init();// initialize timer
            while(1)// loop forever
            {
                // do nothing
                    // comparison is done in the ISR itself
            }
}
```

AVR Timers – TIMER2:

TIMER2 is an 8-bit timer (like TIMER0); most of the registers are similar to that of TIMER0 registers. Apart from that, TIMER2 offers a special feature which other timers don't – **Asynchronous Operation**.

TCNT2 Register:

In the **Timer/Counter register**shown in figure below.

Bit	7	6	5	4	3	2	1	0	
	TCNT2[7:0]								TCNT2
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	

Fig: TCNT2 Register

www.sakshieducation.com

TCCR2 Register:

The **Timer/Counter Control Register**is shown in figure below.

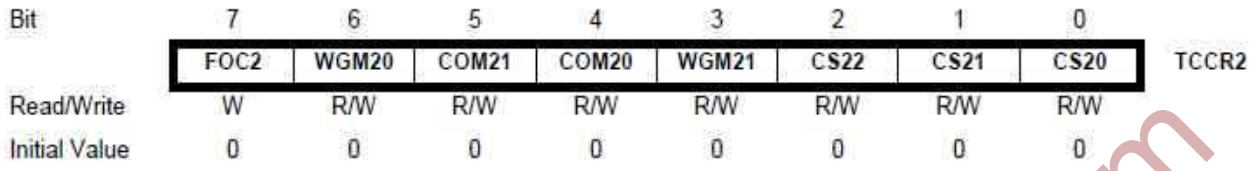


Fig: TCCR2 Register

In TIMER0/1 the prescalers available are 8, 64, 256 and 1024, whereas in TIMER2, we have 8, 32, 64, 128, 256 and 1024.

The **bit 2:0 – CS22:20** are the **Clock Select Bits** of TIMER2. Their selection is as follows.

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	Clk_{T2S}/1 (No prescaling)
0	1	0	Clk_{T2S}/8 (From prescaler)
0	1	1	Clk_{T2S}/32 (From prescaler)
1	0	0	Clk_{T2S}/64 (From prescaler)
1	0	1	Clk_{T2S}/128 (From prescaler)
1	1	0	Clk_{T2S}/256 (From prescaler)
1	1	1	Clk_{T2S}/1024 (From prescaler)

Fig: Clock Select Bit Description

Example:

To flash an LED every 50 ms. We have an XTAL of 16 MHz.

Code:

```
#include <avr/io.h>
#include <avr/interrupt.h>
// global variable to count the number of overflows
volatile uint8_t tot_overflow;
// initialize timer, interrupt and variable
void timer2_init()
{
```


www.sakshieducation.com

```
        // set up timer with prescaler = 256
           TCCR2 |= (1 << CS22)|(1 << CS21);
        // initialize counter
           TCNT2 = 0;
           TIMSK |= (1 << TOIE2);    // enable overflow interrupt
            sei();// enable global interrupts
           // initialize overflow counter variable
           tot_overflow = 0;
}
// TIMER0 overflow interrupt service routine called whenever TCNT0 overflows
ISR(TIMER2_OVF_vect)
{
        tot_overflow++;// keep a track of number of overflows
}
int main(void)
{
        DDRC |= (1 << 0); // connect led to pin PC0
      timer2_init();// initialize timer
      while(1) // loop forever
        {
                // check if no. of overflows = 12
                if (tot_overflow >= 12)  // NOTE: '>=' is used
                {
                  if (TCNT2 >= 53) // check if the timer count reaches 53
                        {
                                PORTC ^= (1 << 0);    // toggles the led
                                TCNT2 = 0;          // reset counter
                                tot_overflow  =  0;       //  reset  overflow
                        counter
                         }
                }
        }
}
```

AVR Timers – CTC Mode:

It is a special mode of operation – **Clear Timer on Compare (CTC) Mode**.

We had two timer values with us – **Set Point** (SP) and **Process Value** (PV).

In everytime, we used to compare the process value with the set point. Once the process value becomes equal (or exceeds) the set point, the process value is reset.

Example:

```
max = 39999;   // max timer value set <--- set point
// some code here
// ...
// ...
// TCNT1 <--- process value
if (TCNT1 >= max)   // process value compared with the set point
{
   TCNT1 = 0;      // process value is reset
}
// ...
```

Since TIMER1 is a 16-bit timer, it can count up to a maximum of 65535. Here, what we desire is that the timer (process value) should reset as soon as its value becomes equal to (or greater than) the set point (Maximum Value) of 39999.

So basically, the CTC Mode implements the same thing, but unlike the above example, it implements it in hardware. Which means that we no longer need to worry about comparing the process value with the set point every time! This will not only avoid unnecessary wastage of cycles, but also ensure greater accuracy (i.e. no missed compares, no double increment, etc).

Hence, this comparison takes place in the hardware itself, inside the AVR CPU! **Once the process value becomes equal to the set point, a flag in the status register is set and the timer is reset *automatically!* Thus goes the name –**

www.sakshieducation.com

CTC – *Clear* Timer on *Compare!* Thus, all we need to do is to take care of the flag, which is much faster to execute.

CTC mode - Timer 1:

Example:

Let's take up a problem to understand this concept. We need to flash an LED every 100 ms. we have a crystal of XTAL 16 MHz.

$$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$

Now, given **XTAL = 16 MHz,** with a prescaler of **64**, the frequency of the clock pulse reduces to **250 kHz**. With a Required Delay = **100 ms,** we get the Timer Count to be equal to **24999**. Up until now, we would have let the value of the timer increment, and check its value every iteration, whether it's equal to 24999 or not, and then reset the timer. *Now*, the same will be done in hardware! We won't check its value every time in software! We will simply check whether the flag bit is set or not, that's all.

Using CTC Mode:

TCCR1A and TCCR1B Registers:

We are already aware of the **Clock Select Bits – CS12:10** in TCCR1B. Hence, right now, we are concerned with the **Wave Generation Mode Bits – WGM13:10**. These bits are spread across both the TCCR1 registers (A and B). Thus we need to be a bit careful while using them. Their selection is as follows:

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	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

www.sakshieducation.com

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 & Frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase & 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

Fig: Wave Generation Mode Bit Description

We can see that there are two possible selections for CTC Mode. Practically, both are the same, except the fact that we store the timer compare value in different registers. Right now, let's move on with the first option (0100). Thus, the initialization of TCCR1A and TCCR1B is as follows.

TCCR1A |= 0; // not required since WGM11:0, both are zero (0)

TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10); // Mode = CTC, Prescaler = 64

OCR1A and OCR1B Registers:

We must tell the AVR to reset the timer as soon as its value reaches *such and such value.* So, the question is, how do we set *such and such values?* The **Output Compare Register 1A** – OCR1A and the **Output Compare Register 1B** – OCR1B are utilized for this purpose.

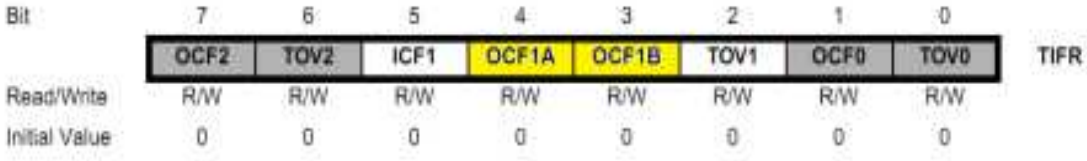
Since the compare value will be a 16-bit value (in between 0 and 65535), OCR1A and OCR1B are 16-bit registers. In ATMEGA16/32, there are two CTC channels – A and B. We can use any one of them or both. Let's use OCR1A.

Example:

OCR1A = 24999; // timer compare value

TIFR Register:

www.sakshieducation.com



Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

We are interested in **Bit 4:3 – OCF1A: B – Timer/Counter1, Output Compare A/B Match Flag Bit**. This bit is set (one) by the AVR whenever a match occurs i.e. TCNT1 becomes equal to OCR1A (or OCR1B). It is cleared automatically whenever the corresponding Interrupt Service Routine (ISR) is executed. Alternatively, it can be cleared by writing ’1′ to it!

Code:

```
#include <avr/io.h>
// initialize timer, interrupt and variable
void timer1_init()
{
// set up timer with prescaler = 64 and CTC mode
   TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);
// initialize counter
   TCNT1 = 0;
   // initialize compare value
   OCR1A = 24999;
}

int main(void)
{
// connect led to pin PC0
   DDRC |= (1 << 0);
// initialize timer
   timer1_init();
// loop forever
while(1)
   {
// check whether the flag bit is set if set, it means that there has been a compare match and the timer has been cleared use this opportunity to toggle the led
if (TIFR & (1 << OCF1A)) // NOTE: '>=' used instead of '=='
      {
```


www.sakshieducation.com

```
            PORTC ^= (1 << 0); // toggles the led
        }
        // wait! we are not done yet!
        // clear the flag bit manually since there is no ISR to execute
        // clear it by writing '1' to it (as per the datasheet)
        TIFR |= (1 << OCF1A);
// yeah, now we are done!
    }
}
```

Using Interrupts with CTC Mode:

In the previous methodology, we simply used the CTC Mode of operation. We used to *check* every time for the flag bit (OCF1A). Now let's shift this responsibility to the AVR itself! Yes, now we *do not need to check* for the flag bit at all! The AVR will compare TCNT1 with OCR1A. Whenever a match occurs, it sets the flag bit OCF1A, and *also* fires an interrupt! We just need to attend to that interrupt, that's it.

There are three kinds of interrupts in AVR –

- *Overflow,*
- *Compare*
- *Capture*.

We have already discussed the *overflow* interrupt.

TIMSK Register:

The **Bit 4:3 –OCIE1A: B – Timer/Counter1, Output Compare A/B Match Interrupt Enable** bits are of our interest here. Enabling it ensures that an interrupt is fired whenever a match occurs. Since there are two CTC channels, we have two different bits OCIE1A and OCIE1B for them.

www.sakshieducation.com

Whenever a match occurs (TCNT1 becomes equal to OCR1A = 24999), an interrupt is fired (as OCIE1A is set) and the OCF1A flag is set. Now since an interrupt is fired, we need an Interrupt Service Routine (ISR) to attend to the interrupt. Executing the ISR clears the OCF1A flag bit automatically and the timer value (TCNT1) is reset.

Code:

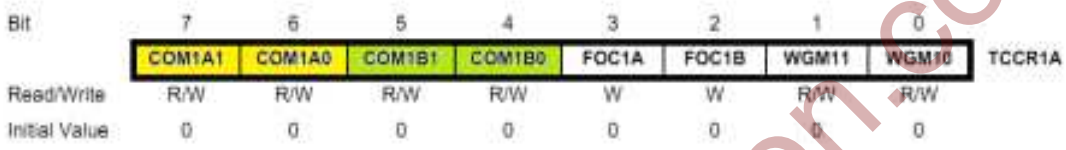
```
#include <avr/io.h>
#include <avr/interrupt.h>
    // initialize timer, interrupt and variable
    void timer1_init()
    {
        // set up timer with prescaler = 64 and CTC mode
          TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);
          TCNT1 = 0; // initialize counter
          OCR1A = 24999;// initialize compare value
          TIMSK |= (1 << OCIE1A);// enable compare interrupt
        sei();// enable global interrupts
    }
    // this ISR is fired whenever a match occurs hence, toggle led here itself..
    ISR (TIMER1_COMPA_vect)
    {
        PORTC ^= (1 << 0);// toggle led here
    }
    int main(void)
    {
        DDRC |= (1 << 0);// connect led to pin PC0
        timer1_init();// initialize timer
        while(1)    // loop forever
        {
            // do nothing
             // whenever a match occurs, ISR is fired
            // toggle the led in the ISR itself
            // no need to keep track of any flag bits here
        }
    }
```

www.sakshieducation.com

Using Hardware CTC Mode:

In the pin configuration of ATMEGA16/32,we can see the pins PB3, PD4, PD5 and PD7. Their special functions are mentioned in the brackets (OC0, OC1A, OC1B and OC2). These are the Output Compare pins of TIMER0, TIMER1 and TIMER2 respectively.

Here **TCCR1A Register**plays major role to operate this mode.



Now time for us to concentrate on **Bit 7:6 – COM1A1:0** and **Bit 5:4 – COM1B1:0 – Compare Output Mode for Compare Unit A/B**. These bits control the behavior of the Output Compare (OC) pins. The behavior changes depending upon the following modes:

- Non-PWM mode (normal / CTC mode)
- Fast PWM mode
- Phase Correct / Phase & Frequency Correct PWM mode

Right now we are concerned only with the CTC mode.

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on compare match.
1	0	Clear OC1A/OC1B on compare match (Set output to low level)
1	1	Set OC1A/OC1B on compare match (Set output to high level)

We choose the second option (01).No need to check any flag bit, no need to attend to any interrupts, nothing. Just set the timer to this mode. Whenever a compare match occurs, the OC1A pin is automatically toggled.

But we need to compromise on the hardware. *Only* PD5 or PD4 (OC1A or OC1B) can be controlled this way, which means that we should connect the LED to PD5 (since we are using channel A) instead of PC0 or else.

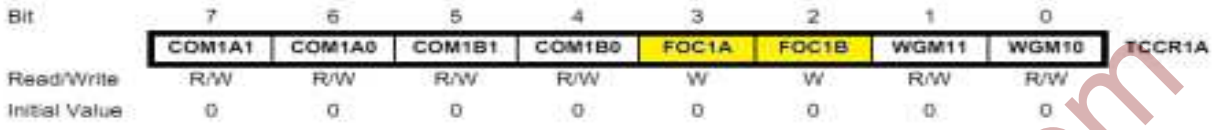
Code:

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

// initialize timer, interrupt and variable
void timer1_init()
{
        // set up timer with prescaler = 64 and CTC mode
           TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);
        // set up timer OC1A pin in toggle mode
           TCCR1A |= (1 << COM1A0);
        // initialize counter
           TCNT1 = 0;
           // initialize compare value
           OCR1A = 24999;
}

int main(void)
{
           DDRD |= (1 << 5);    // connect led to pin PD5
        timer1_init();// initialize timer
        while(1) // loop forever
           {
                   // do nothing
                   // whenever a match occurs
                   // OC1A is toggled automatically!
                   // no need to keep track of any flag bits or ISR
           }
}
```

www.sakshieducation.com

Forcing Compare Match:

Bit 3:2in TCCR1A – FOC1A: B – Force Output Compare for Compare Unit A/B.



Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- These bits are *write only* bits.
- They are active only in non-PWM mode.
- For ensuring compatibility with future devices, these bits must be set to zero (which they *already* are by default).
- Setting them to '1' will result in an immediate forced compare match and the effect will be reflected in the OC1A/OC1B pins.
- **The thing to be noted is that FOC1A/FOC1B will *not* generate any interrupt, *nor* will it clear the timer in CTC mode.**

CTC mode - Timer 0/ Timer 2:

In this section we will discuss about the registers only.CTC mode of TIMER 0/ 2 is exactly in the same way of TIMER 1. So we will discuss about TIMER 0 now.

TCCR0 Register:

The **Timer/Counter0 Control Register**– TCCR10 Register is as follows:

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 6:3 – WGM01:00 – Wave Generation Mode** – Just like in TIMER1, we choose the type of wave mode from here as follows.

- Choose 10 for CTC mode.

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

Table: Wave Generation Mode Bit Description

- **Bit 5:4 – COM01:00 – Compare Match Output Mode –** They control the behavior of the OC0 (PB3) pin
  - depending upon the WGM mode –
    - *non-PWM,*
    - *Phase Correct PWM mode*
    - *Fast PWM mode*.
  - The selection options of non-PWM mode are as follows.
  - Choose 01 to toggle the LED.

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Table: Compare Output Mode, non-PWM

- **Bit 7 – FOC0 – Force Output Compare –** This bit,
  - When set to '1' forces an immediate compare match and affects the behavior of OC0 pin.

www.sakshieducation.com

- For ensuring compatibility with future devices, this bit must be set to '0'.

- **Bit 2:0 – CS02:0 – Clock Select Bits**

OCR0 Register:

The **Output Compare Register**– OCR0 Register is as follows:

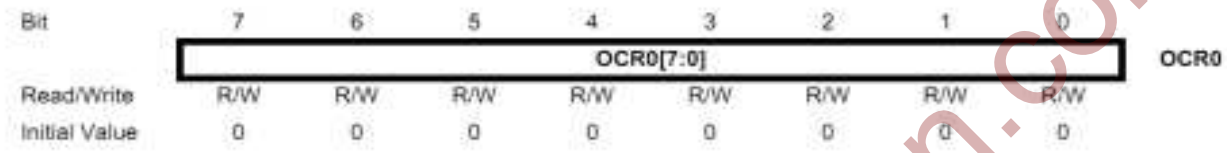


Fig: OCR0 Register

The value to be compared (max 255) is stored in this register.

TIMSK Register:

The **Timer/Counter Interrupt Mask**– TIMSK Register is as follows:

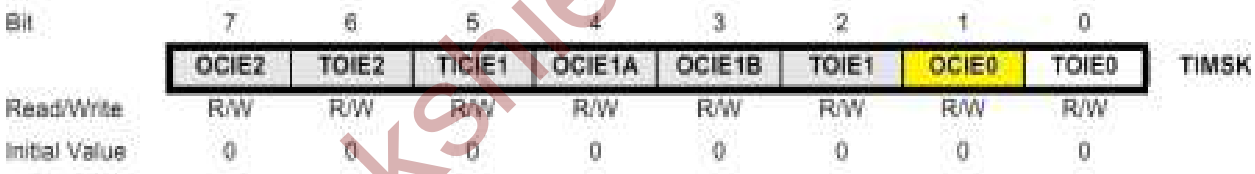


Fig: TIMSK Register

The **Bit 1 – OCIE0 – Timer/Counter0 Output Compare Match Interrupt Enable** enables the firing of interrupt whenever a compare match occurs.

TIFR Register:

The **Timer/Counter Flag Register**– TIFR is as follows:

www.sakshieducation.com

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

Fig: TIFR Register

The **Bit 1 – OCF0 – Output Compare Flag 0** is set whenever a compare match occurs. It is cleared automatically whenever the corresponding ISR is executed. Alternatively it is cleared by writing '1' to it.

AVR Timers – PWM Mode:

PWM is the technique used to generate analogue signals from a digital device like a MCU. Almost all modern MCUs have dedicated hardware for PWM signal generation.

PWM can be used to control servo motors, perform DAC (Digital to Analogue Conversion) etc.

PWM: Pulse Width Modulation:

It is basically a modulation technique, in which the width of the carrier pulse is varied in accordance with the analog message signal.

In PWM, we generate square waves whose **duty cycle** can be varied. Duty cycle refers to the fraction of the time period of the wave for which the signal is in high state (or simply ON state).

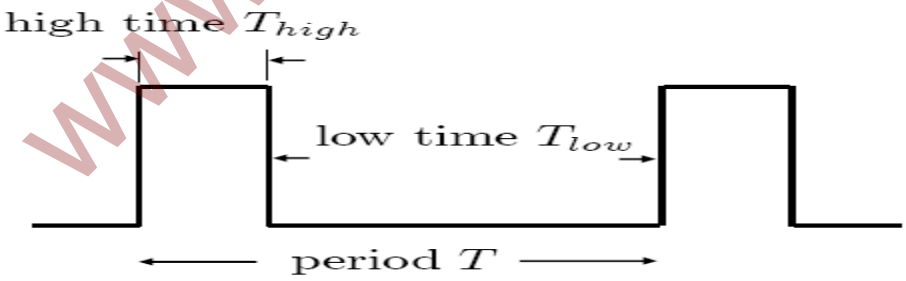


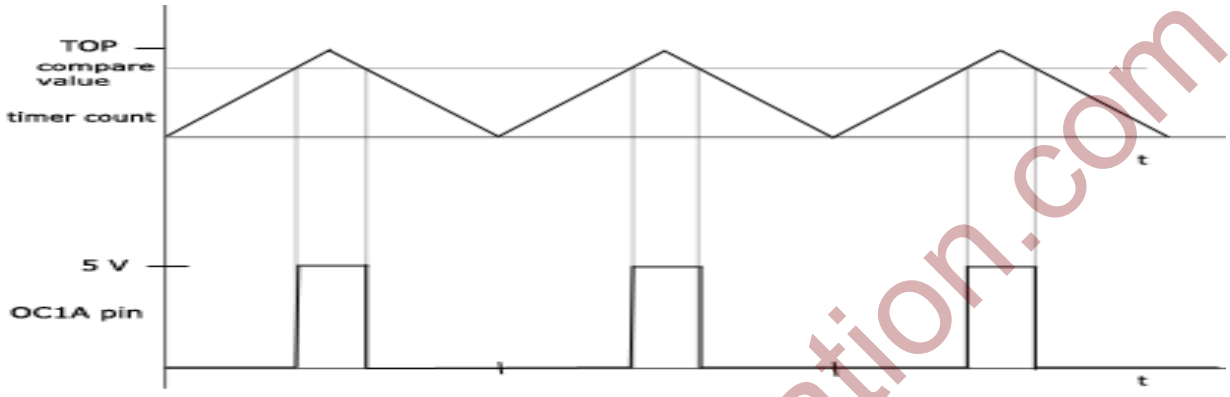
Fig: A PWM Waveform.

A PWM signal is a periodic rectangular pulse.

Frequency = (1/T)

Duty Cycle = (T_{high}/T)

The simplest way to generate a PWM signal is by comparing the predetermined waveform with a fixed voltage level as shown below.



It has three **compare output modes** of operation:

- **Inverted Mode -** In this mode, if the waveform value is greater than the compare level, then the output is set high, or else the output is low.
- **Non-Inverted Mode -** In this mode, the output is high whenever the compare level is greater than the waveform level and low otherwise.
- **Toggle Mode -** In this mode, the output toggles whenever there is a compare match. If the output is high, it becomes low, and vice-versa.

But it's always not necessary that we have a fixed compare level. Those who have had exposure in the field of analog/digital communication must have come across cases where a **saw tooth carrier wave is compared with a sinusoidal message signal** as shown below.

www.sakshieducation.com

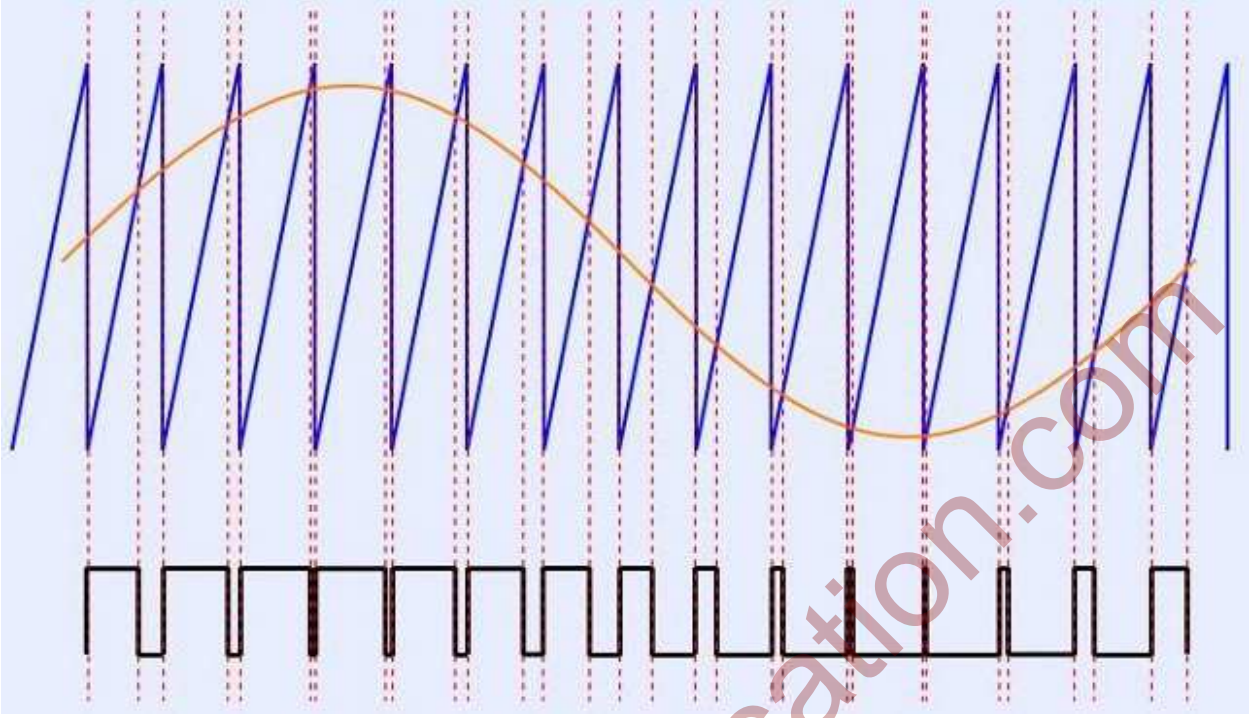
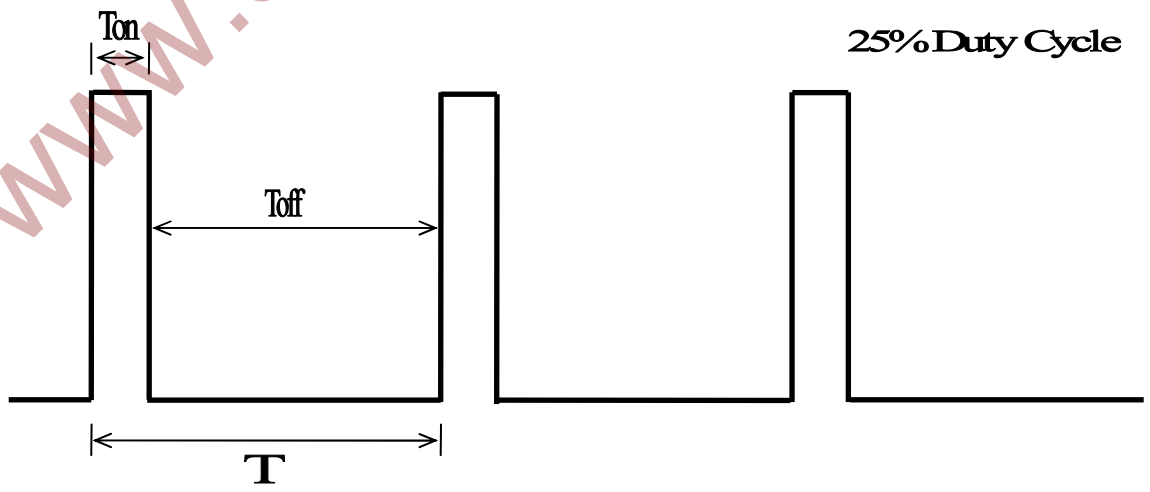


Fig: PWM Modulation

Duty Cycle:

The Duty Cycle of a PWM Waveform is given by

$$Duty\ Cycle = \frac{T_{on}}{T_{on} + T_{off}} \times 100\ \%$$



www.sakshieducation.com

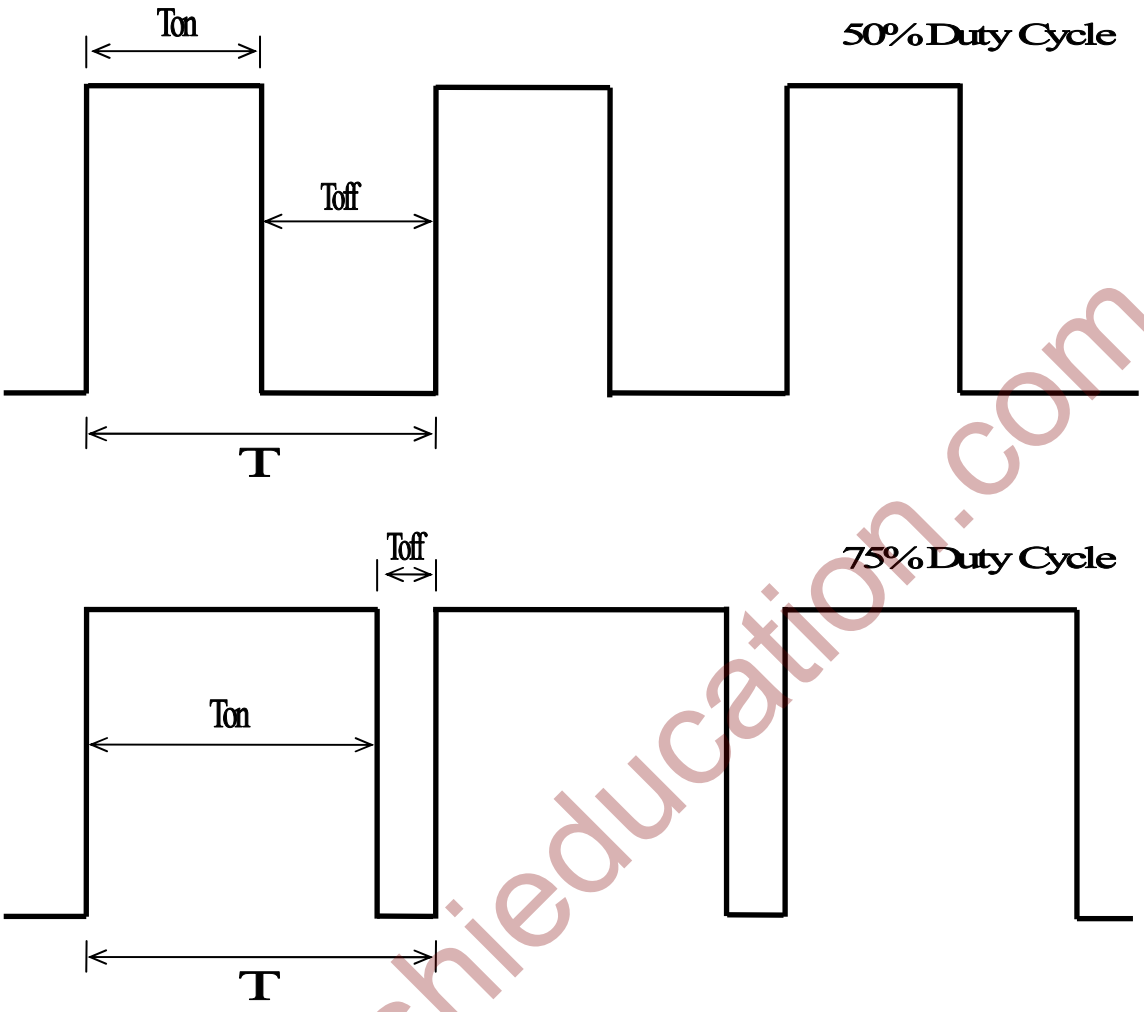
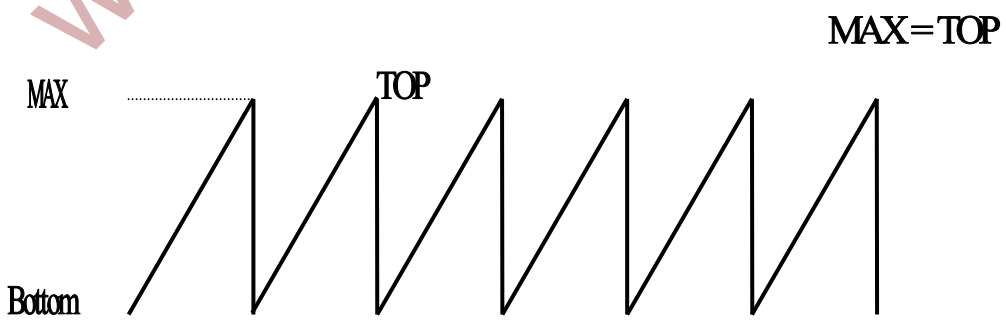


Fig: PWM with Different Duty cycles.

We are very well aware that the AVR provides us with an option of 8 and 16 bit timers. 8bit timers count from 0 to 255, then back to zero and so on. 16bit timers count from 0 to 65535, then back to zero. Thus for a 8bit timer, MAX = 255 and for a 16bit timer, MAX = 65535.



www.sakshieducation.com

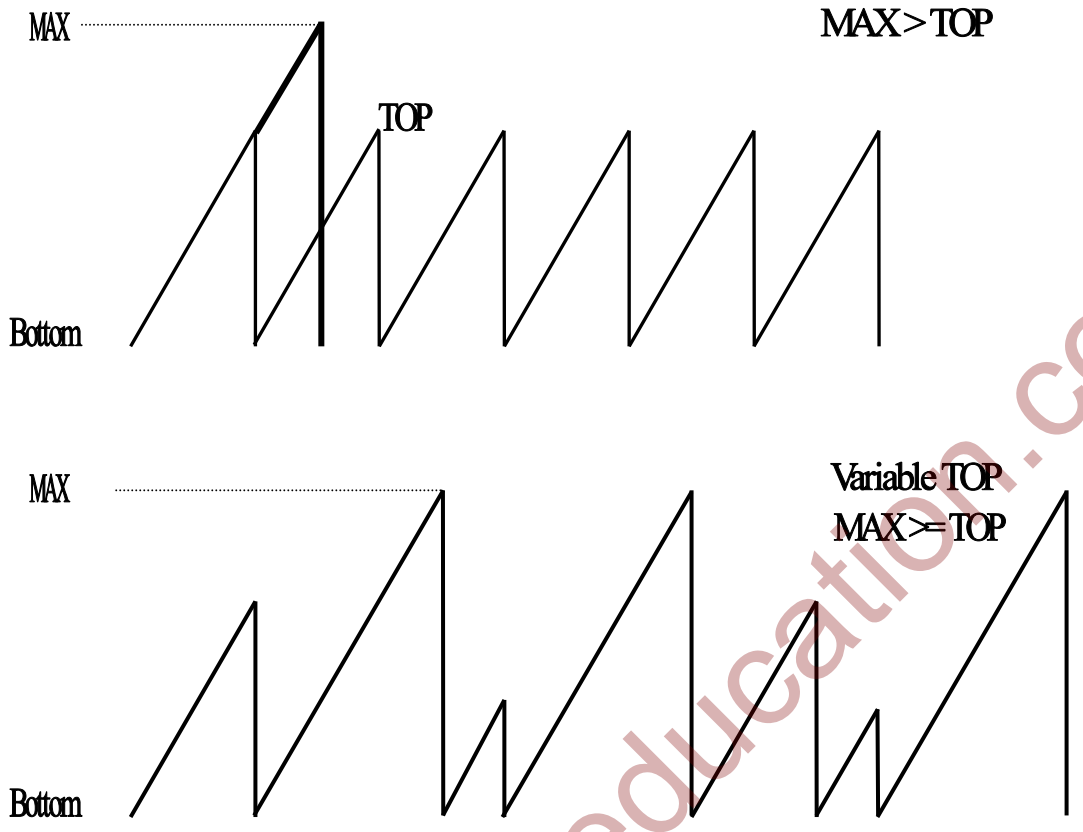


Fig: Fixed and Variable TOP in Timers

Note: **TOP never exceeds MAX. TOP <= MAX.**

Before going to PWM concepts in timers we have to aware about **TOP, Bottom and MAX**.

The timer *always* counts from **0 to TOP**, then overflows back to zero. The1st figure shown above, **TOP = MAX**.

We knew in CTC Mode, in which we can clear the timer whenever a compare match occurs. Due to this, the value of TOP can be reduced as shown in 2nd figure. The thick line shows how the timer would have gone in normal mode.

Now, the CTC Mode can be extended to introduce variable TOP as shown in 3rd figure.

PWM Modes of Operation:

www.sakshieducation.com

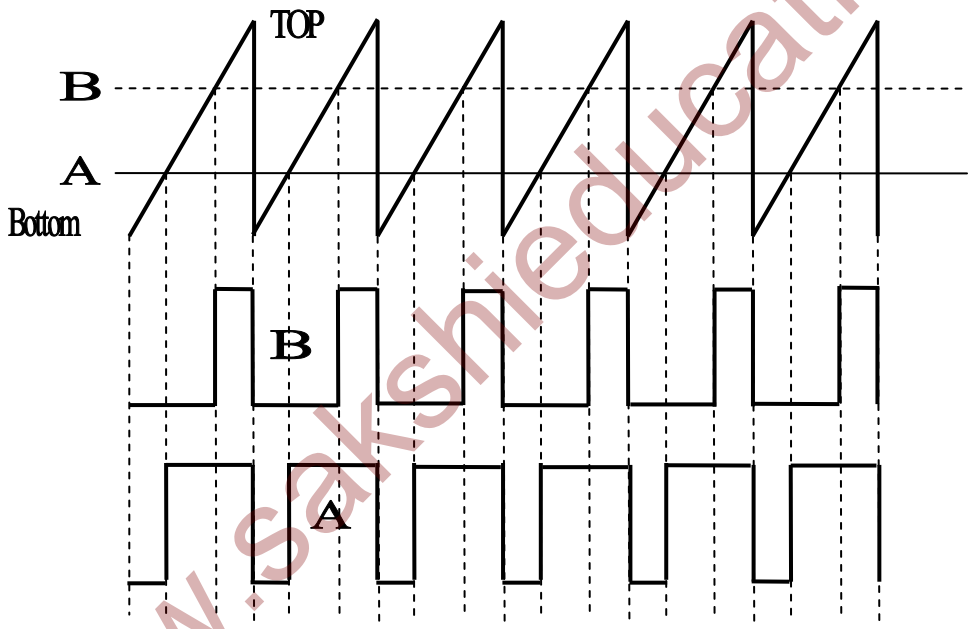
In general, there are three modes of operation of PWM Timers:

- Fast PWM
- Phase Correct PWM
- Frequency and Phase Correct PWM

Fast PWM:

In simple terms, *this* is Fast PWM! We have a saw tooth waveform, and we compare it with a fixed voltage level (say A), and thus we get a PWM output as shown (in A).

Now suppose we increase the compare voltage level (to, say B). In this case, as we can see, the pulse width has reduced, and hence the duty cycle.

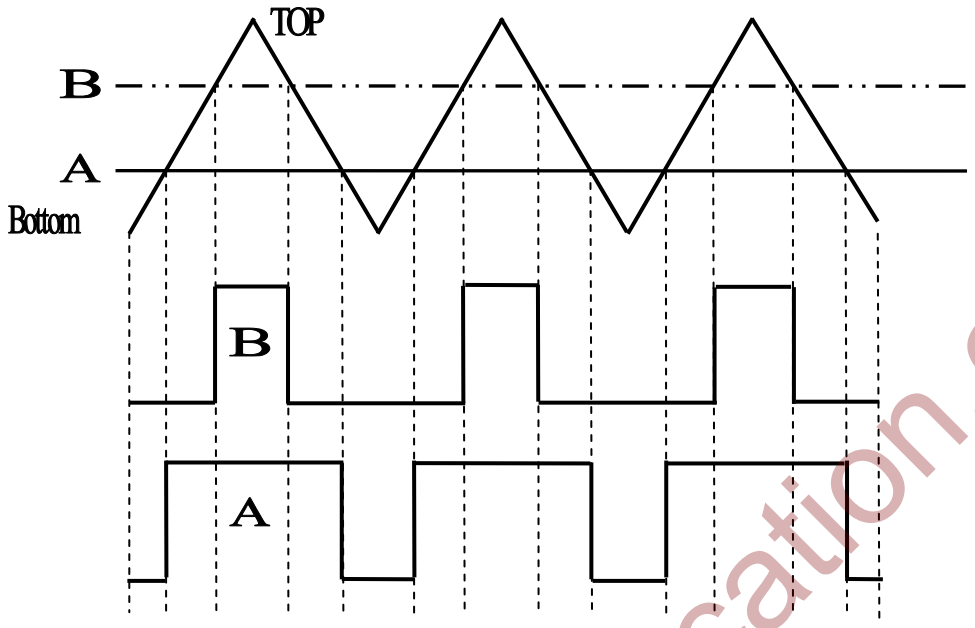


But, as you can see, both the pulses (A and B) end at the same time irrespective of their starting time.

In this mode, since saw tooth waveform is used, the timer counter TCNTn (n = 0,1,2) counts from BOTTOM to TOP and then it is simply allowed to overflow (or cleared at a compare match) to BOTTOM.

www.sakshieducation.com

Phase Correct PWM:



Here instead of a saw tooth waveform, we have used a triangular waveform. Even here, you can see how PWM is generated. We can see that upon increasing the compare voltage level, the duty cycle reduces. But unlike Fast PWM, the phase of the PWM is maintained. Thus it is called *Phase Correct* PWM.

By visual inspection, we can clearly see that the frequency of Fast PWM is twice that of Phase Correct PWM.

Frequency and Phase Correct PWM:

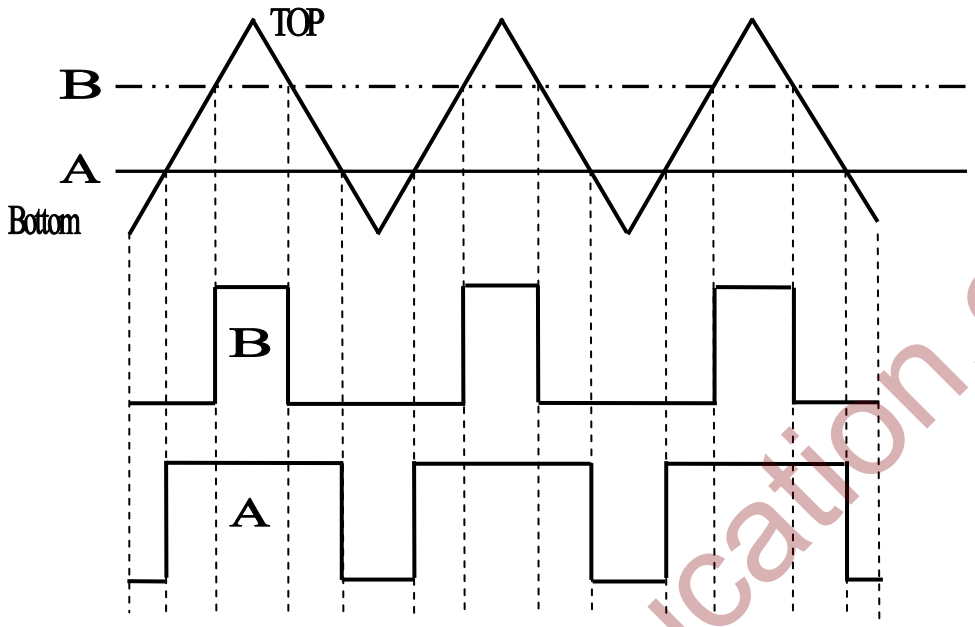
The datasheets say that there is **no difference between 'phase correct' and 'phase and frequency correct' modes** if we are not changing the value of TOP on the fly. Since TOP is dictating our repeating frequency then we aren't changing it so these two modes are interchangeable and analogous.

So we will cover both of them with one discussion and will refer to them both collectively as 'any phase correct' mode.

The major difference is that 'fast PWM mode' counted repeatedly from BOTTOM to TOP to generate a saw tooth waveform whereas these 'any phase

www.sakshieducation.com

correct modes' will count up from BOTTOM to TOP, and then from TOP to BOTTOM so rather than a saw tooth they generate a triangular waveform:



Thus, for this, we need Frequency and Phase Correct PWM. Since in most cases the value of TOP remains same, it doesn't matter which one we are choosing – Phase Correct or Frequency and Phase Correct PWM.

Making Choices:

Now that we are familiar with all the PWM concepts, it's up to you to decide

- Which timer to choose?
- Which mode of operation to choose?
- Which compare output mode to choose?

Choosing Timer:

In AVR, PWM Mode is available in all timers. TIMER0 and TIMER2 provide 8bit accuracy whereas TIMER1 provides 16bit accuracy. In 8bit accuracy, we have 256 individual steps, whereas in 16bit accuracy, we have 65536 steps.

Now suppose we want to control the speed of a DC motor. In this case, having 65536 steps is totally useless. Thus we can use an 8bit timer for this.

www.sakshieducation.com

Even 8bit is too much, but there is no other choice. Obviously there isn't much difference in speed between 123/256th and 124/256th of full speed in case of a motor.

But if we use servo motors, you have to use 16bit timer. If we need quite high resolution in your application, go for 16bit timer.

Choosing Mode of Operation

If we want to control the speed of DC motors or brightness of LEDs, go for any one of them. But if we are using it for telecommunication purposes, or for signal sampling, fast PWM would be better. For general applications, phase correct PWM would do.

Choosing Compare Output Modes

Out of the three modes,

- inverted,
- non-inverted
- toggle mode,

Non-inverted mode is the most reasonable. This is because upon increasing the compare voltage, the duty cycle increases. However, you can choose any of them.

Example: Let us take a problem statement. We need to generate a 50 Hz PWM signal having 45% duty cycle.

Analysis:

Given that

Frequency = 50 Hz

In other words, the time period, T

T = T (on) + T (off) = 1/50 = 0.02 s = 20 ms

Also, given that

www.sakshieducation.com

Duty Cycle = 45%

Thus, solving according to equation given above, we get

T (on) = 9 ms

T (off) = 11 ms

Now, this can be achieved in two ways:

- Use Timer in CTC Mode
- Use Timer in PWM Mode

Methodology – CTC Mode:

- Firstly, choose a suitable timer.
  - For this application, we can choose any of the three timers available in ATMEGA32.
- Choose a suitable prescaler.
- Then set up the timer and proceed as usual.

Note: The catch lies here is that you need to update the compare value of OCRx register every time.

Code:

```
#include <avr/io.h>
#include <avr/interrupt.h>
uint8_t count = 0;              // global counter
voidtimerX_init()// initialize timer, interrupt and variable
{
        // set up timerX with suitable prescaler and CTC mode
        // initialize counter
        // initialize compare value
        // enable compare interrupt
        // enable global interrupts
}
ISR (TIMERx_COMPA_vect)// process the ISR that is fired
{
      // do whatever you want to do here say, increment the global
      counter
```


www.sakshieducation.com

```
            count++;
            // check for the global counter
               // if count == odd, delay required = 11 ms
               // if count == even, delay required = 9 ms
               // thus, the value of the OCRx should be constantly updated
            if (count % 2 == 0)
                  OCRx = 9999;      // calculate and substitute appropriate
            value
            else
                  OCRx = 10999;     // calculate and substitute appropriate
            value
      }
      int main(void)
      {
            DDRC |= (1 << 0);// initialize the output pin, say PC0
            timerX_init();   // initialize timerX
            while(1)   // loop forever
              {
                  // do nothing
              }
      }
```

Methodology – PWM Mode:

The PWM Mode in AVR is hardware controlled. This means that *"everything"*, is done by the AVR CPU. All we need to do is to initialize and start the timer, and set the duty cycle.

We can choose any timer of AVR microcontroller, here we are using TIMER0.

TCCR0 – Timer/Counter0 Control Register:

Here, we will learn how to set appropriate bits to run the timer in PWM mode.

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	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	

Fig: TCCR0 Register

We will discuss only those bits which are of interest to us now.

- **Bit 6, 3 – WGM01, 00 – Waveform Generation Mode** - These bits can be set to either "00" or "01" depending upon the type of PWM you want to generate.

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

Fig: Waveform Generation Mode Bit Description

- **Bit 5, 4 – COM01:0 – Compare Match Output Mode** - These bits are set in order to control the behavior of Output Compare pin (OC0) in accordance with the WGM01:00 bits.

The following look up table to determine the operations of OC0 pin for Fast PWM mode.

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Fig:Compare Output Mode, Fast PWM Mode

Now let's have a look at the Fast PWM waveforms.

www.sakshieducation.com

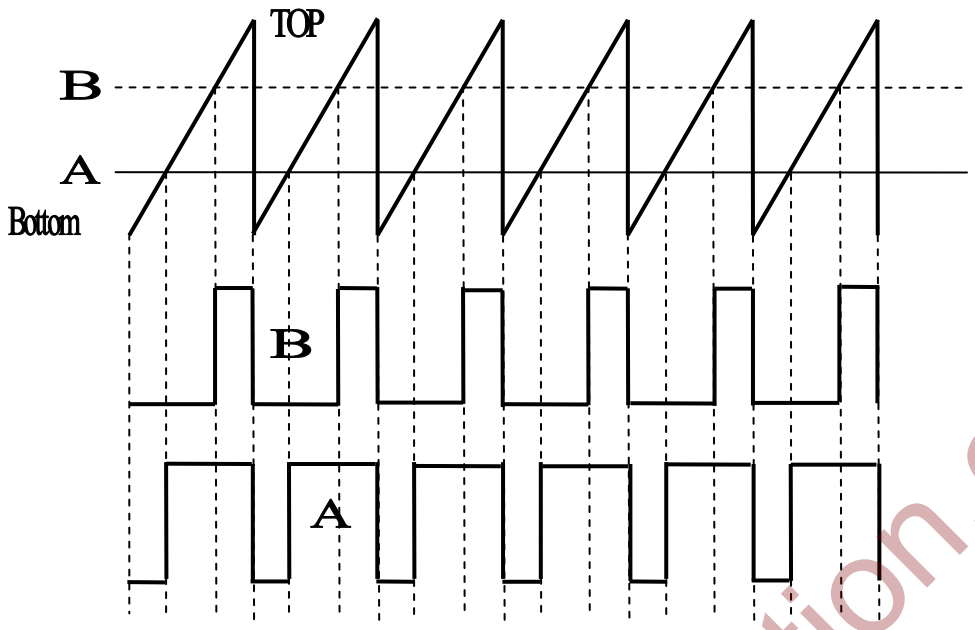


Fig: Fast PWM

Now let me remind you that the AVR PWM is fully hardware controlled, which means that even the timer compare operation is done by the AVR CPU. All we need to do is to *tell* the CPU *what* to do once a match occurs.

The COM01:00 pins come into play here. We see that by setting it to "10″ or "11″, the output pin OC0 is either set or cleared (in other words, it determines whether the PWM is in inverted mode, or in non-inverted mode).

Similarly for Phase Correct PWM, the look up table and the waveforms go like this.

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Reserved
1	0	Clear OC0 on compare match when up-counting. Set OC0 on compare match when down-counting.
1	1	Set OC0 on compare match when up-counting. Clear OC0 on compare match when down-counting.

Fig: Compare Output Mode, Phase Correct PWM Mode

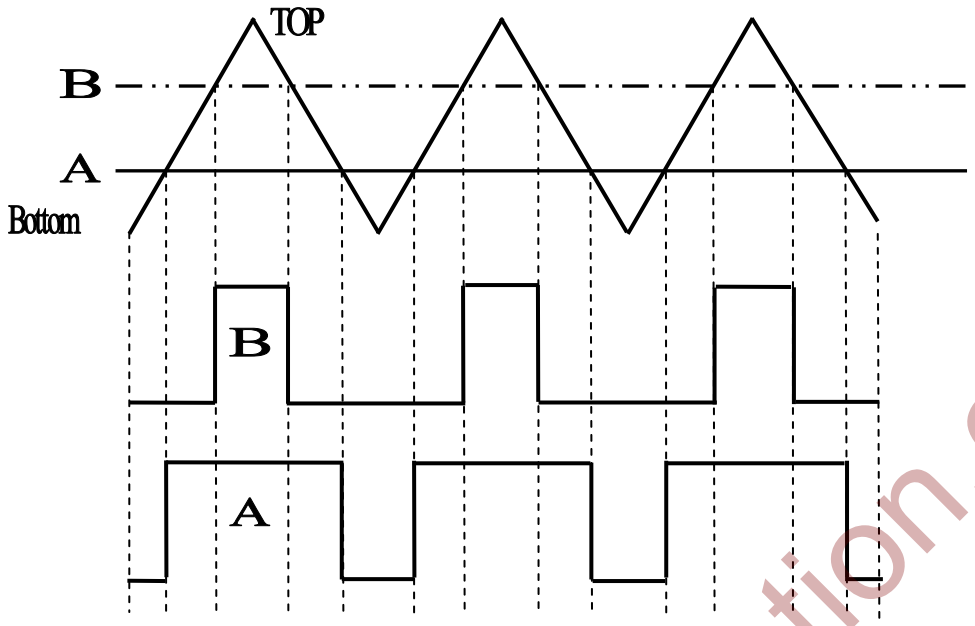


Fig: Phase Correct PWM

Setting of COM01:00 to “10″ or “11″ determines the behavior of OC0 pin. As shown in the waveforms, there are two instances – one during up-counting, and other during down-counting. The behavior is clearly described in the look up table.

Please note that OC0 is an output pin. Thus, the effects of WGM and COM won’t come into play unless the DDRx register is set properly.

- **Bit 2:0 – CS02:0 – Clock Select Bits** - These bits are used to select prescaler.

OCR0 – Output Compare Register

We use this register to store the compare value. But when we use Timer0 in PWM mode, the value stored in it acts as the duty cycle (obviously!). In the problem statement, it’s given that the duty cycle is 45%, which means

OCR0 = 45% of 255 = 114.75 = 115

Edit: Note

www.sakshieducation.com

The following code discusses how to create a PWM signal of a desired duty cycle. If we want to change its frequency, you need to alter the TOP value, which can be done using the ICRx register (which is not supported by 8-bit timers). For 16-bit Timer1, it can be varied using ICR1A.

Code:

```
#include <avr/io.h>
#include <util/delay.h>
        voidpwm_init()
        {
                // initialize TCCR0 as per requirement, say as follows
                   TCCR0                                                    |=
        (1<<WGM00)|(1<<COM01)|(1<<WGM01)|(1<<CS00);
                // make sure to make OC0 pin (pin PB3 for atmega32) as output
        pin
                   DDRB |= (1<<PB3);
        }
        void main()
        {
                uint8_t duty;
                duty = 115;      // duty cycle = 45% of 255 = 114.75 = 115
                   // initialize timer in PWM mode
                pwm_init();
                   // run forever
                while(1)
                   {
                    OCR0 = duty;
                   }
        }
```

www.sakshieducation.com