3.1 Comparison of Embedded C and assembly language programming, AVR programming in C, AVR Data types., Arithmetic and Logic Operations programming
3.2 I/O port programming.
3.3 Timer programming, PWM programming.
3.4 ADC programming

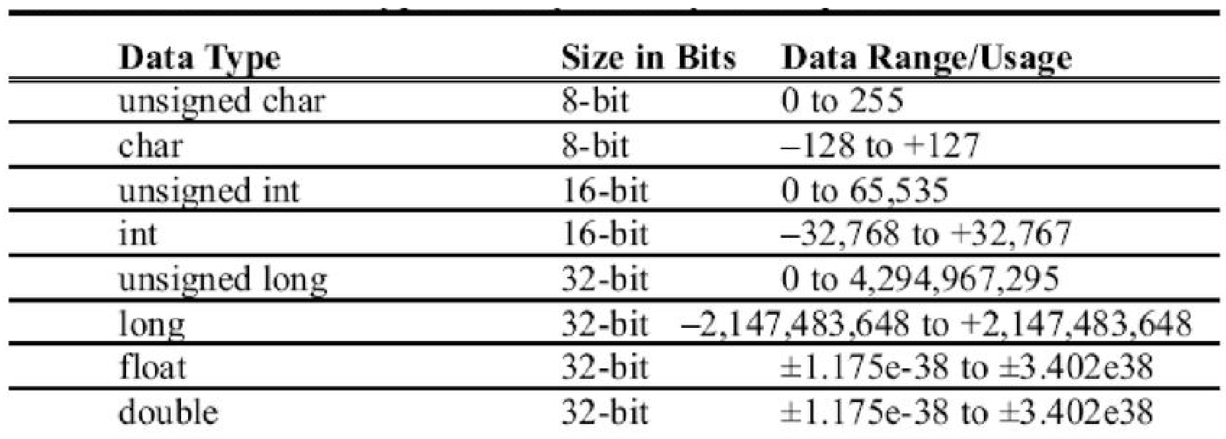
**3.1 Comparison of Embedded C and assembly language programming, AVR programming in C, AVR Data types, Arithmetic and Logic Operations programming**

a. **Comparison of Embedded C and assembly language programming**

| **Parameter** | **Assembly language** | **Embedded C** |
|---|---|---|
| Execution time | Faster(Less execution time required) | Slower(More execution time required) |
| Time for coding | More time is required for coding | Less time required for coding and code is more efficient |
| Hex file size | less | More |
| Debugging | Not so easy | Easy |
| Portability | Assembly does not provide portability and source code is specific to a processor | Portable, the code written in c could be easily reused on a different platform |

b. **AVR Data types**

AVR Data Types with their ranges



| Data Type | Size in Bits | Data Range/Usage |
|---|---|---|
| unsigned char | 8-bit | 0 to 255 |
| char | 8-bit | –128 to +127 |
| unsigned int | 16-bit | 0 to 65,535 |
| int | 16-bit | –32,768 to +32,767 |
| unsigned long | 32-bit | 0 to 4,294,967,295 |
| long | 32-bit | –2,147,483,648 to +2,147,483,648 |
| float | 32-bit | ±1.175e-38 to ±3.402e38 |
| double | 32-bit | ±1.175e-38 to ±3.402e38 |

c. **Arithmetic and Logic Operations programming**

Arithmetic Operators:

+:  adds up two values
–:  substracts up two values
*:  Multiply two values
/:  Divides two values
%:  finds the remainder of the division.
++ :  increments the value by one. For example- a++
—:  decrements the value by one. For example- a–

**Program to perform arithmetic operations**

```
#include <avr/io.h>
int main(void)
{
unsigned int a;
unsigned int b;
DDRA = 0xFF; // make Port A output
DDRB = 0xFF; // make Port B output
DDRC = 0xFF; // make Port C output
DDRD = 0xFF; // make Port D output
a=0X04;
b=0X05;
PORTA = a + b; //ADDITION
PORTB = a - b; //SUBTRACTION
PORTC = a *b; //MULTIPLICATION
PORTD= a /b; //DIVISION
while (1);
}
```

Write AVR C program to perform addition, subtraction and multiplication operations on two constant data and outputs the result to Ports B with 5s delay between each result.

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
```

```
DDRB = 0xFF; // make Port B output
unsigned int a;
unsigned int b;
DDRB = 0xFF; // make Port B output
a=0X04;
b=0X05;
PORTB = a + b; //ADDITION
_delay_ms(5000);
PORTB = a - b; //SUBTRACTION
_delay_ms(5000);
PORTB = a *b; //MULTIPLICATION
while (1);
}
```

**Logical Operations:**

|  |  | AND | OR | EX-OR | Inverter |
|---|---|---|---|---|---|
| A | B | A&B | A\|B | A^B | Y=~B |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |  |
| 1 | 1 | 1 | 1 | 0 |  |

|  |  |
|---|---|
| 1. 0x35 & 0x0F = 0x05 | /* ANDing */ |
| 2. 0x04 \| 0x68 = 0x6C | /* ORing */ |
| 3. 0x54 ^ 0x78 = 0x2C | /* XORing */ |
| 4. ~0x55 = 0xAA | /* Inverting 55H */ |

**Program for Logical Operations:**

```
#include <avr/io.h>
int main(void)
{
unsigned int a;
unsigned int b;
DDRA = 0xFF; // make Port A output
DDRB = 0xFF; // make Port B output
DDRC = 0xFF; // make Port C output
DDRD = 0xFF; // make Port D output
a=0X04;
b=0X05;
PORTA = a & b; //AND operation
PORTB = a | b; //OR operation
PORTC = a ^b; //XOR operation
PORTD= ~a; //NOT operation
while (1);
}
```

**Write AVR C program to perform AND, OR, XOR operations on two constant data and outputs the result to Ports B with 5 seconds delay between each result.**

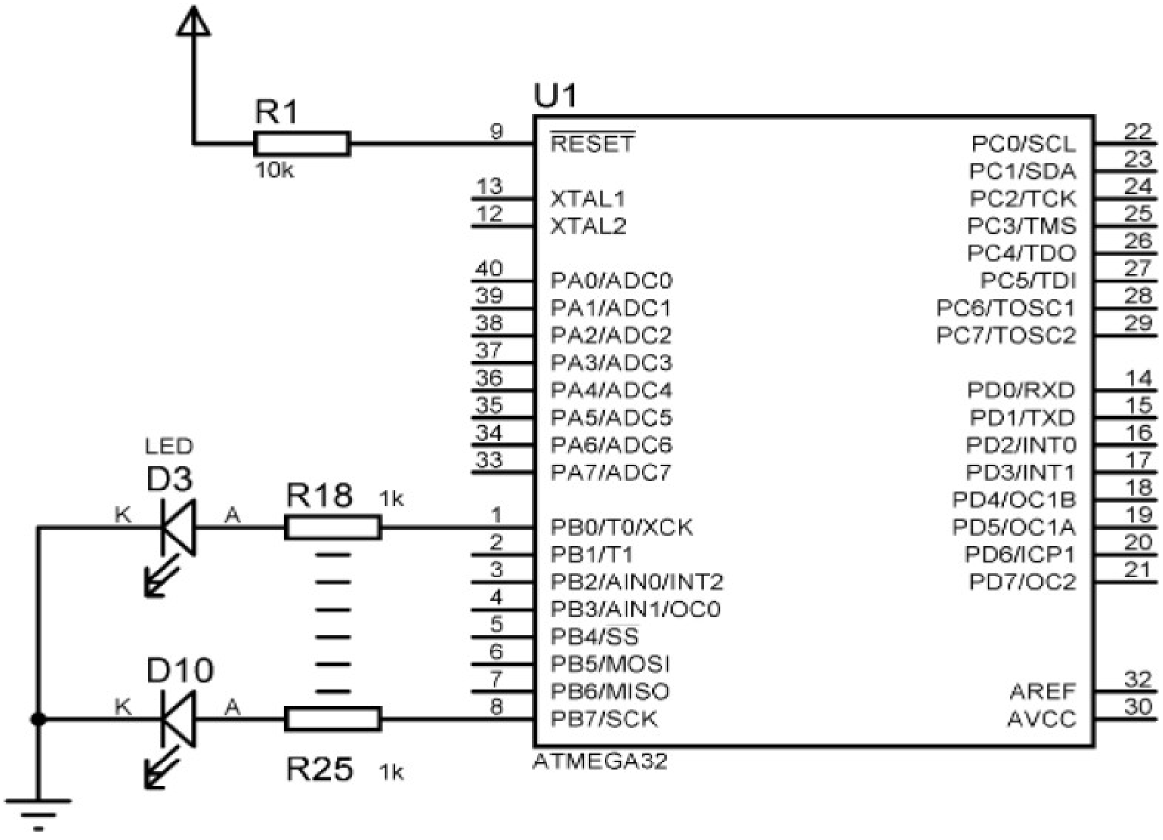
```
#include <avr/io.h>
#include<util/delay.h>
int main(void)
{
unsigned int a;
unsigned int b;
DDRB = 0xFF; // make Port B output
a=0X35;
b=0X05;
PORTB = a & b; //ANDing
_delay_ms(5000);
PORTB = a | b; //Oring
_delay_ms(5000);
PORTB = a ^ b; //XORing
_delay_ms(5000);
while (1);
}
```

**3.2 I/O port programming.**

1. **Write an AVR C program to send one byte of data to PORTB of ATMEGA32 to understand the operation of PORT as output.**

ATMEGA ports are 8 bit wide. Each port has 3 eight bit registers associated. Each bit in these

registers configures pins of associated port. Bit 0 of these registers is associated with Pin 0 of the
port, Bit1 of these registers is associated with Pin1 and so on.
These three registers are
– DDRx register
– PORTx register
– PINx register
X may be replaced by A,B,C or D based on the PORT you are using.
DDRx register
DDRx (Data Direction Register) configures data direction of the port pins. Which, writing 0 to a bit
in DDRx makes corresponding port pin as input, while writing 1 to a bit in DDRx makes the
corresponding port pin as output.
EXAMPLE:
• to make all pins of port B as input, DDRA = 0b00000000;
• to make all pins of port A as output pins : DDRB= 0b11111111;
• to make lower nibble of port B as output and higher nibble as input :
DDRB = 0b00001111; In hexadecimal representation, it can be written as DDRB = 0x0F;



```
#include <avr/io.h> // standard AVR header
int main(void)
{
DDRB = 0xff; // setting PORTB as output
PORTB = 0B01010101; // setting value for PORTB
while (1); // indefinite loop
}
```

2. **To send data from 00 to FF to PORTB of ATMEGA32 by giving sufficient delay between out operation. Write program in C.**

A data variable z is defined and initialized with zero, and during each iteration of the for loop, the value of z is given to PORTB and increment z by one. The program never exits the for loop because if you increment an unsigned char variable when it is 0xFF, it will become zero.

```
#include <avr/io.h> // standard AVR header
#include<util/delay.h> // delay header file
#define F_CPU 1000000UL /* it tells the compiler that the MC running at 1MHz. */
int main(void)
{
unsigned char z; // defining data variable z
DDRB = 0xFF; // setting PORTB as output
for(z=0;z<=255;z++) // data z varies from 0 to 255
{
PORTB = z; // sending z to PORTB
_delay_ms(10000); // time in ms
}
return 0;
}
```

3. **Write an AVR C program to blink LED connecetd to PORTB of ATMEGA32 ON and OFF at 1sec interval with internal RC oscillator at 1Mhz.**

PORTB is set as output and all the bits of PORTB is initialized as ON. A delay of one second is set by using the timer function _delay_ms. After assigning the delay the value in PORTB is negated and assigned to PORTB by using the command PORTB = ~ PORTB. This two commands are placed in an infinite while loop.

```
#define F_CPU 1000000UL
#include <avr/io.h> // standard AVR header
#include <util/delay.h> // delay loop function
```

```
int main(void)
{
DDRB = 0xFF; // setting PORTB as output
PORTB = 0b11111111; // setting value for PORTB
while (1) // indefinite loop
{
_delay_ms(1000);
PORTB = ~ PORTB;
}
}
```

4. **BIT-WISE OPERATORS : To blink LED connected to PB1 at 1sec interval while keeping LED on PB7 on using AND, OR operations**

Previous concept of PORT setting by moving values to the entire port cannot be used In practical cases as moving can change values of pins which are not desired to be affected.
Hence usually AND and OR logic operations are used to manipulate bits.
In order to set PB7 in this case
PORTB |= 0B10000000; // setting PB7
In order to clear PB1
PORTB &= 0B11111101; // clear PB1
This can be rewritten as
In order to set PB7 in this case
PORTB |= (PORTB <<7); // setting PB7
Or as
PORTB |= (PORTB <<PB7); // setting PB7
PORTB &= ~(PORTB <<1); // clear PB1
Or as
PORTB &= ~(PORTB <<PB1); // clear PB1

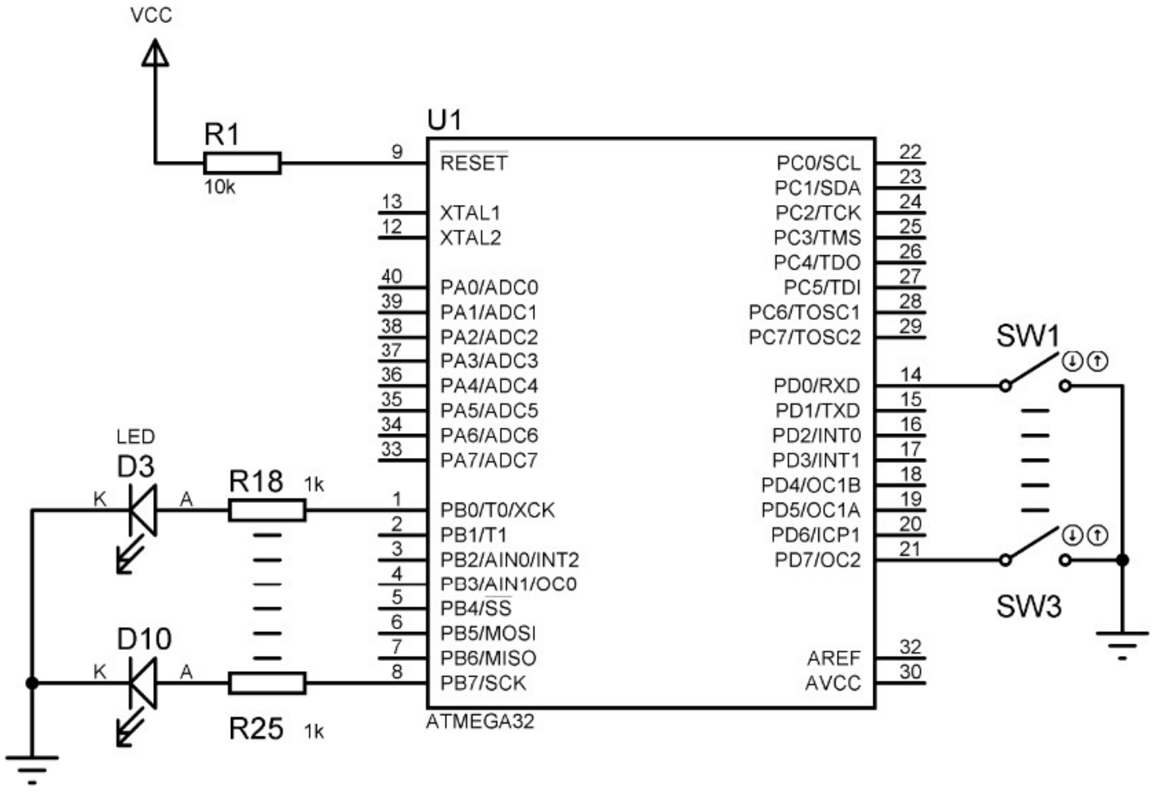
**Program**

```
#include <avr/io.h> // standard AVR header
#include <util/delay.h>
int main(void)
{
DDRB = 0xff; // setting PORTB as output
PORTB |= (1<<PB7); // setting value for PORTB
while (1) // indefinite loop
{
PORTB |= (1<<PB1); // setting Pin 1 of PORTB
_delay_ms(10000);
PORTB &= ~(1<<PB1); // resetting Pin 1 of PORTB
_delay_ms(10000);
}
}
```

## 5. PORT PIN AS INPUT

To read switches connected to PORTD and output the contents to LEDs connected to PORTB. Write a program in C.

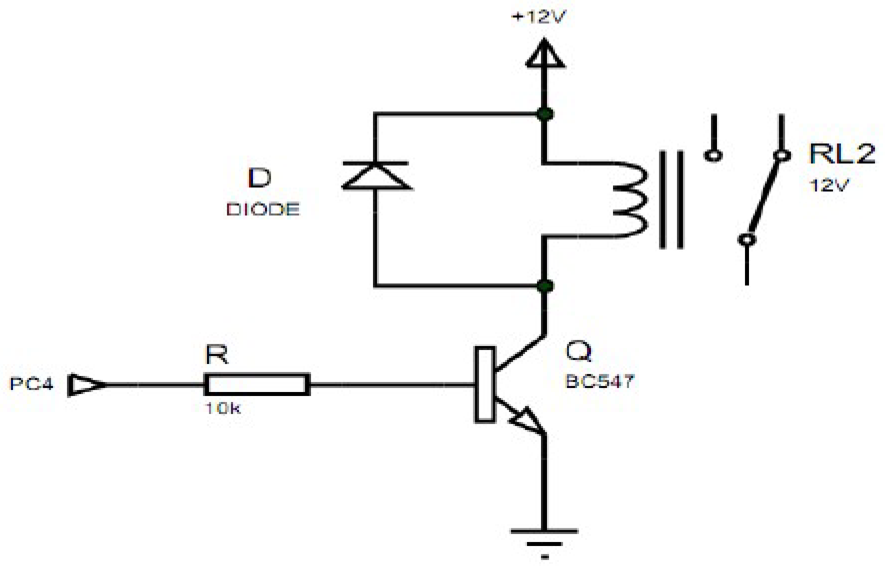
PINx register : PINx (Port IN) register is used to read data from port pins. In order to read the data from the port pin, first you have to change the port's data direction to input. This is done by setting bits in DDRx to zero. If the port is made output, then reading the PINx register will give you data that has been output on port pins. There are two input modes. Either you can use port pins as tri stated inputs or you can activate internal pull up. In order to enable internal pull up, you have to write 1 to the corresponding PORTx register.



```
#include <avr/io.h>
#include<util/delay.h>
#define F_CPU 1000000UL
```

```
int main(void)
{
DDRB = 0xFF; // DDRB as output
DDRD = 0x00; // DDRD as input
PORTD = 0xFF; //ACTIVATE INTERNAL PULL-UP RESISTORS
while(1)
{
_delay_ms(10); //Calling Delay
/* When a key is pressed corresponding bit of PIND
register becomes zero. so to send a HIGH to LED for that key press, we need to give
complement of PIND value to the output port B. */
PORTB = ~PIND;
}
}
```

6. **Relay Interfacing**



```
#define F_CPU 8000000UL // clock at 8MHz
#include <avr/io.h> // standard AVR header
#include <util/delay.h> // delay loop function
#define relay PC4
int main(void)
{
DDRC = 0xFF; // PORTC pins are defined as output
PORTC = 0X00; // Initializing PORTC with 0
while (1) // indefinite loop
{
_delay_ms(3000);
PORTC |= (1<<relay); // relay ON
_delay_ms(3000);
PORTC &= ~(1<<relay); // relay OFF
}
```

**3.3 Timer programming, PWM programming.**

A. **Timer Programming**

**Steps to Program Delay using Timer0**

1. Load the TCNT0 register with the initial value (let's take 0x25).

2. For normal mode and the pre-scaler option of the clock, set the value in the TCCR0 register. As soon as the clock Prescaler value gets selected, the timer/counter starts to count, and each clock tick causes the value of the timer/counter to increment by 1.

3.Timer keeps counting up, so keep monitoring for timer overflow i.e. TOV0 (Timer0 Overflow) flag to see if it is raised.

4. Stop the timer by putting 0 in the TCCR0 i.e. the clock source will get disconnected and the timer/counter will get stopped.

5. Clear the TOV0 flag. Note that we have to write 1 to the TOV0 bit to clear the flag.

6. Return to the main function.

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

void T0delay();

int main(void)
{
        DDRB = 0xFF;                    /* PORTB as output*/

        while(1)                /* Repeat forever*/
        {
                PORTB=0x55;
                T0delay();      /* Give some delay */
                PORTB=0xAA;
                T0delay();
        }
}

void T0delay()
```

```
{
        TCNT0 = 0x25;                  /* Load TCNT0*/
        TCCR0 = 0x01;                  /* Timer0, normal mode, no pre-scalar */

        while((TIFR&0x01)==0);  /* Wait for TOV0 to roll over */
        TCCR0 = 0X00;
        TIFR = 0x01;                   /* Clear TOV0 flag*/
}
```

The time delay generated by above code

As Fosc = 8 MHz

T = 1 / Fosc = 0.125 µs

Therefore, the count increments by every 0.125 µs.

In above code, the number of cycles required to roll over are:

0xFF - 0x25= 0xDA i.e. decimal 218

Add one more cycle as it takes to roll over and raise TOV0 flag: 219

Total Delay = 219 x 0.125 µs = 27.375 µs

**Example2**

Let us generate a square waveform having 10 ms high and 10 ms low time:

First, we have to create a delay of 10 ms using timer0.

*Fosc = 8 MHz

Use the pre-scalar 1024, so the timer clock source frequency will be,

8 MHz / 1024 = **7812.5 Hz**

Time of 1 cycle = 1 / 7812.5 = **128 µs**

Therefore, for a delay of 10 ms, number of cycles required will be,

10 ms / 128 µs = **78 (approx)**

We need 78 timer cycles to generate a delay of 10 ms. Put the value in TCNT0 accordingly.

Value to load in TCNT0 = 256 – 78 (78 clock ticks to overflow the timer)

**= 178 i.e. 0xB2 in hex**

Thus, if we load 0xB2 in the TCNT0 register, the timer will overflow after 78 cycles i.e. precisely after a delay of 10 ms.

*Note - All calculations are done by considering 8 MHz CPU frequency. If you are using another value of CPU frequency, modify the calculations accordingly; otherwise, the delay will mismatch.

Program for 10ms Delay Using Timer0

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

void T0delay();

int main(void)
{
        DDRB = 0xFF;                /* PORTB as output */
        PORTB=0;
        while(1)                /* Repeat forever */
        {
                PORTB= ~ PORTB;
                T0delay();
        }
}
void T0delay()
{
        TCCR0 = 0X05;    /* Timer0, normal mode, /1024 prescalar */
        TCNT0 = 0xB2;                /* Load TCNT0, count for 10ms */
        while((TIFR&0x01)==0);  /* Wait for TOV0 to roll over */
        TCCR0 = 0X00;
        TIFR = 0x01;                /* Clear TOV0 flag */
}
```

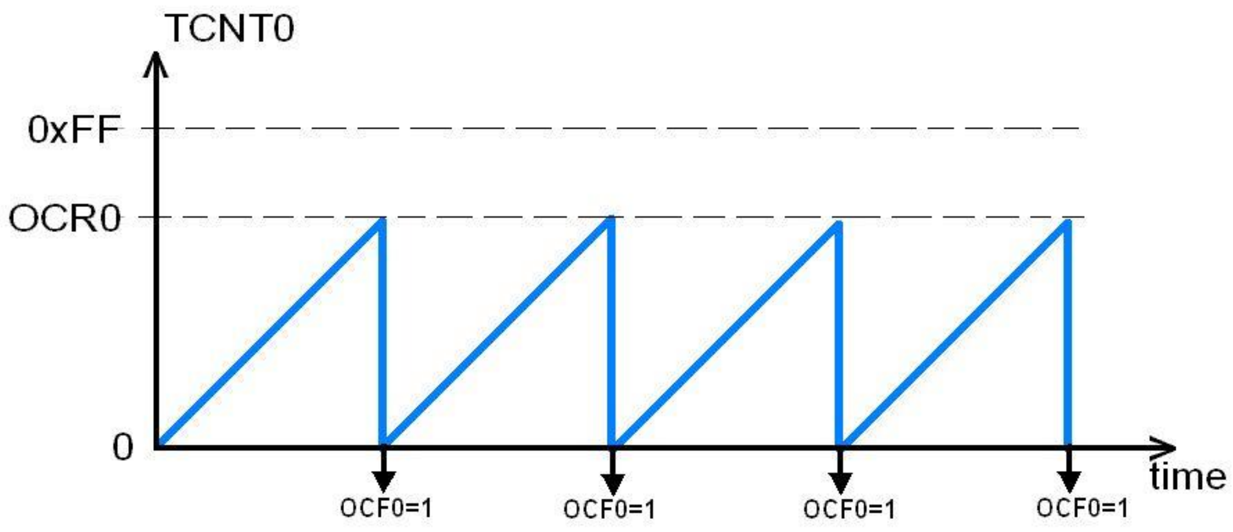
## Clear Timer on Compare Match (CTC mode) in AVR ATmega16/ATmega32

Generally, compare mode is used for generating periodic events or for generating waveforms.

In compare mode, there is one compare register, where we can set the value to compare with the Timer/counter register value. Once the compare value matches with the timer/counter register value, a compare match occurs. This compare match event can be used for waveform generation.

In ATmega 16 / 32, the Timer counts up until the value of the TCNT0 (Timer/counter register) register becomes equal to the content of OCR0 (Compare register). As soon as TCNT0 becomes equal to the OCR0, a compare match occurs, and then the timer will get cleared and the OCF0 flag will get set.

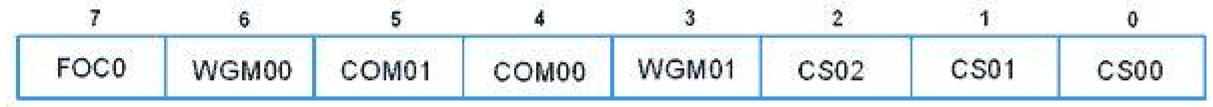
OCF0 flag is located in the TIFR register.



## Waveform Generation

Using Normal mode & CTC mode:

**TCCR0: Timer / counter control register**



| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 |

**Bit 7 - FOC0:** Force compare match

After setting this bit, the timer forced to match occur. i.e. setting output compare flag.

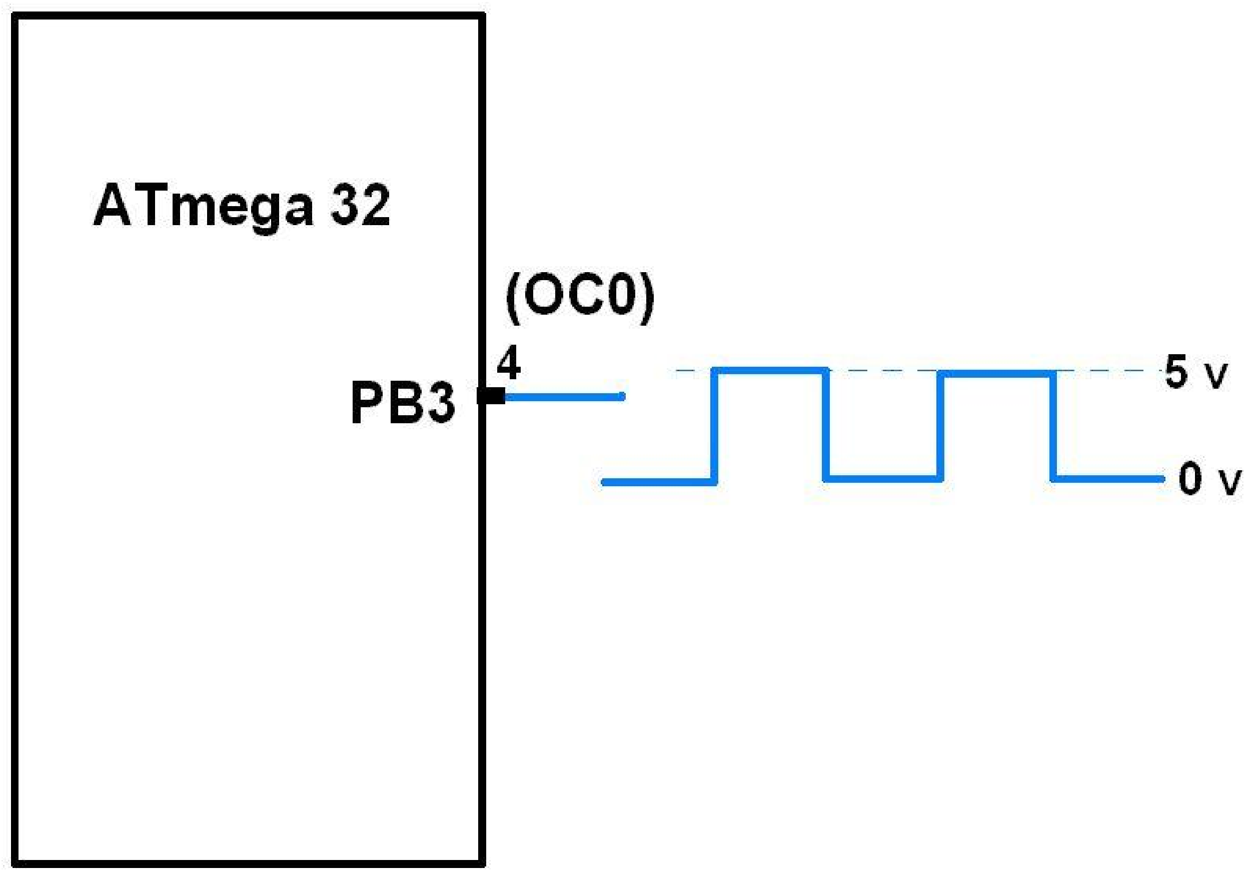
**Bit 6, 3- WGM00, WGM01-** Timer0 mode selection bit

| WGM00 | WGM01 | Timer0 mode selection bit |
|---|---|---|
| 0 | 0 | Normal |
| 0 | 1 | CTC (Clear timer on Compare Match) |
| 1 | 0 | PWM, Phase correct |
| 1 | 1 | Fast PWM |

So, we can generate a square wave with PWM waveforms on pin OC0 (output compare pin) using different modes.

**Bit 5, 4- COM01:00 (compare output mode)**

COM01:00 controls OC0 pin behavior, however, the DDR bit of corresponding OC0 pin must be set to make OC0 pin as an output.



Waveform Generation on OC0 Pin of ATmega16/32

The following table defines the behavior of OC0 pin for the Normal and CTC mode (i.e. WGM00: WGM01= 00   & 01)

| COM01 | COM00 | Description |
|---|---|---|
| 0 | 0 | The 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 |

**Bit 2:0 - CS02:CS00: Clock Source Select**

These bits are used to select a clock source. When CS02: CS00 = 000, then timer is stopped. As it gets a value between 001 to 101, it gets a clock source and starts as the timer.

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| **0** | **0** | **0** | No clock source (Timer / Counter stopped) |
| **0** | **0** | **1** | clk (no pre-scaling) |
| **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. |

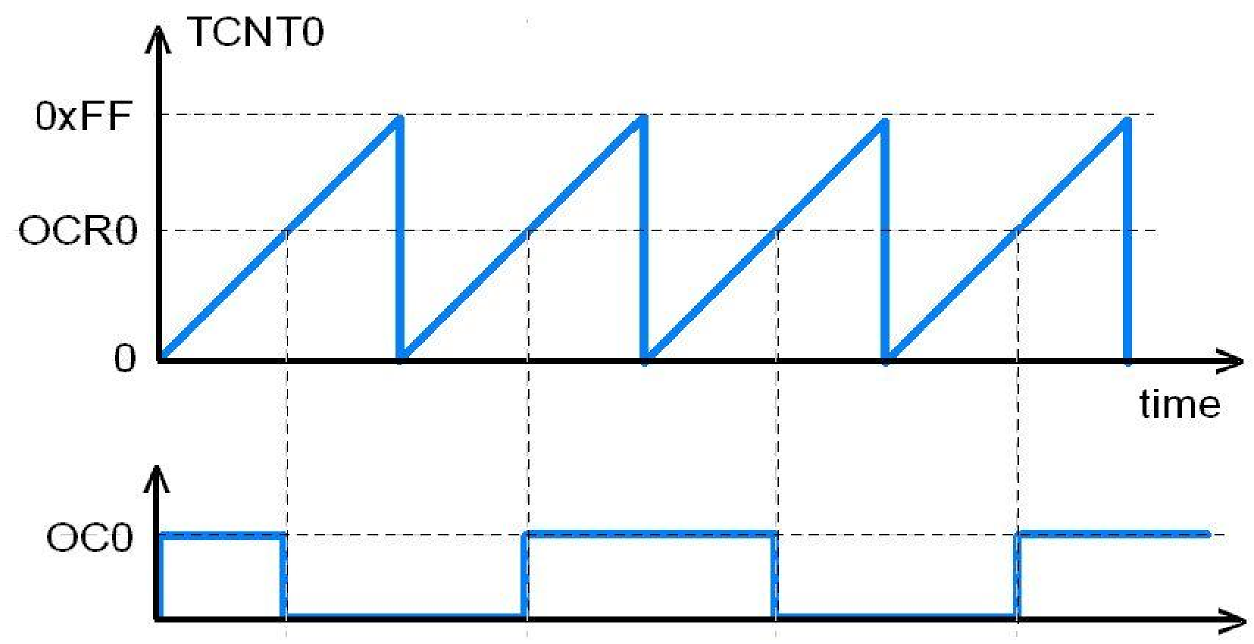
## Generating square wave

1. **Square wave using the normal mode using OCR register:**

To generate a square wave in normal mode, we can set COM bit as toggle mode (COM01:00=01), so OC0 pin will be toggle on each compare match and the square wave will be generated.

## Normal Mode Waveform Generation Program

```
#include "avr/io.h"
int main ()
{
      DDRB = DDRB|(1<<3);
      TCCR0 = 0x11;     /* normal mode, clk- no pre-scaling */
      OCR0 = 100;       /* compare value */
      while (1);
      return 0;
}
```



In normal mode, when a match occurs, the OC0 pin toggles and the timer continues to count up until it reaches to the top value.

Frequency of square wave:

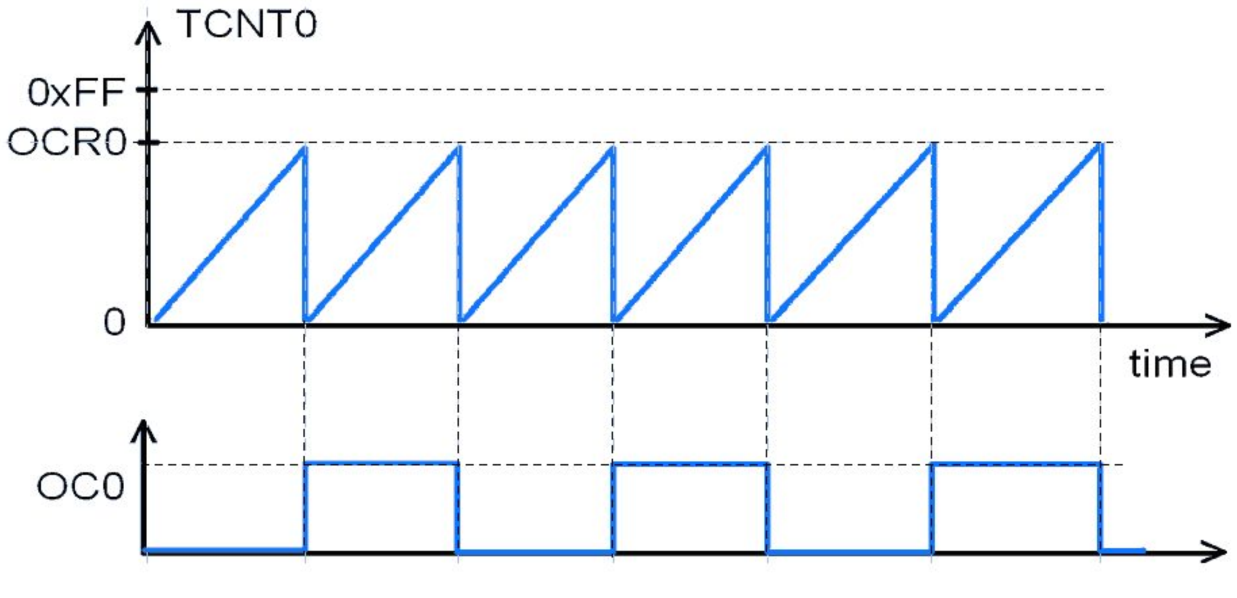
Assuming Fosc=8MHz waveform, T=0.125 µs.

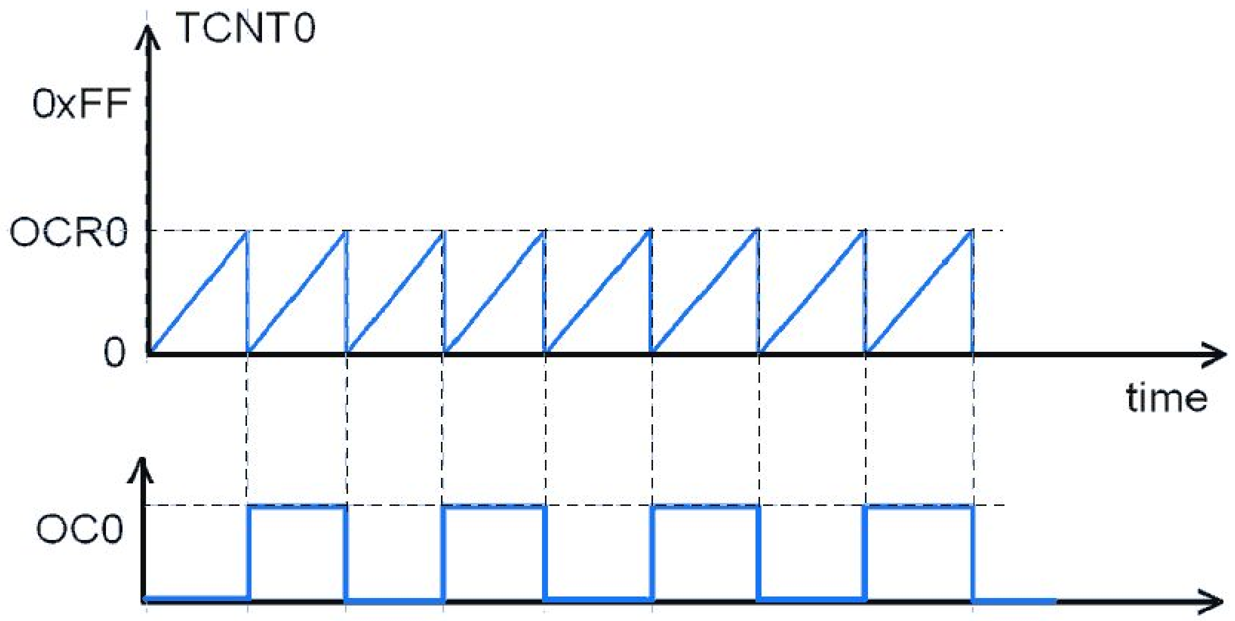
Time period of square wave: 2 x 256 x 0.125 µs   = 64 µs

Frequency of wave= 1/64 µs = 15,625 kHz.

**2. Square wave Using CTC mode:**

**This is a better mode than the normal mode for generating square waves because the frequency of the wave can be easily adjusted using the OCR0 register. See the figure**





**Waveform Generation Using CTC Mode**

As you can see, when a compare match occurs, the timer value becomes zero.

```
#include "avr/io.h"
int main ( )
{
        DDRB = DDRB|(1<<3);         /* PB3 (OC0) as output */
        TCCR0 = 0x19;               /* CTC mode, toggle on compare match,
                        clk- no pre-scaling */
        OCR0 = 200;         /* compare value */
        while (1);
}
```

Square wave frequency calculations:

Assuming Fosc=8MHz waveform, T=0.125 µs.

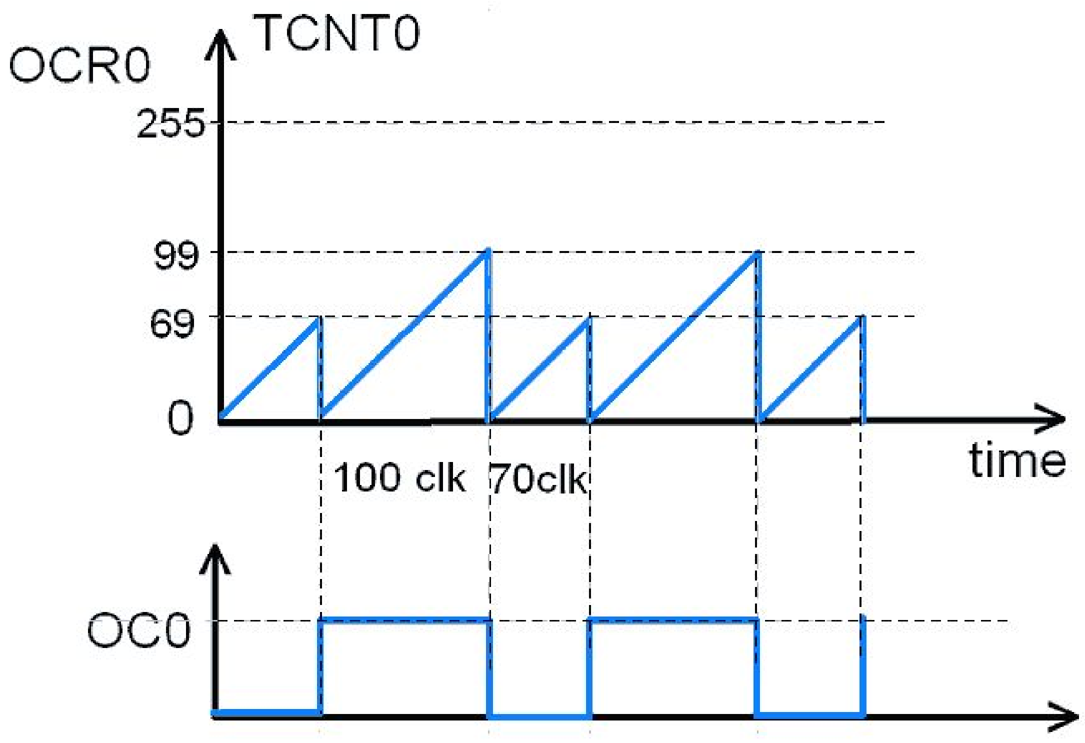
Time period of square wave: 2 x (OCR0+1) x 0.125 µs      = 2 x 201 x 0.125 µs =50.25 µs

Frequency of square wave = 1/50.25 µs = 19.9 KHz.

**Here we have changed the value of OCR0 in runtime, to generate different pulses in the below program.**

```
#include "avr/io.h"
int main ( )
{
        DDRB |= (1<<3);                 /*PB3 (OC0) as output */
        while (1)
        {
                OCR0 = 69;
                TCCR0 = 0x39;           /* CTC, set on match, no prescaler */
                while ((TIFR&(1<<OCF0)) == 0);  /* monitor OCF0 flag */
                TIFR = (1<<OCF0);/* Clear OCF0 by writing 1 */
                OCR0 = 99;
                TCCR0 = 0x29;           /* CTC, clear on match, no prescaler */
                while ((TIFR&(1<<OCF0)) == 0);
                TIFR = (1<<OCF0);/* Clear OCF0 by writing 1 */
        }
}
```

Output:



# PWM in AVR ATmega16/ATmega32

Pulse Width Modulation (PWM) is a technique by which the width of a pulse is varied while keeping the frequency constant.

Why do we need to do this? Let’s take an example of controlling DC motor speed, more the Pulse width more the speed. Also, there are applications like controlling light intensity by PWM.

A period of a pulse consists of an ON cycle (5V) and an OFF cycle (0V). The fraction for which the signal is ON over a period is known as the duty cycle.

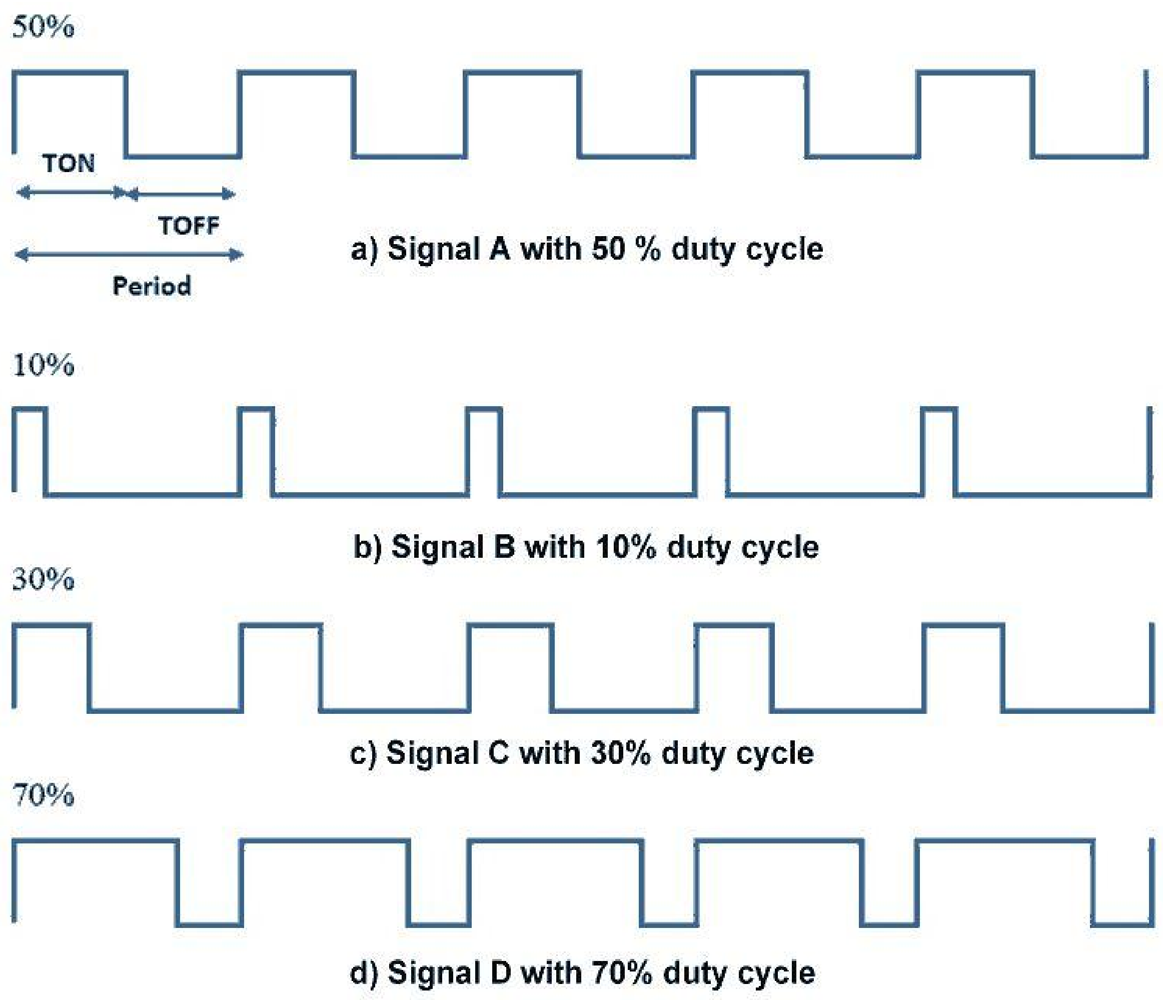
Duty Cycle (In %) = $\frac{T_{ON}}{Total Period} * 100$

E.g. Consider a pulse with a period of 10ms which remains ON (high) for 2ms.The duty cycle of this pulse will be

D = 2ms / 10ms = 20%

Through the PWM technique, we can control the power delivered to the load by using the ON-OFF signal.

Pulse Width Modulated signals with different duty cycle are shown below


50%
TON
TOFF
Period
a) Signal A with 50 % duty cycle
10%
b) Signal B with 10% duty cycle
30%
c) Signal C with 30% duty cycle
70%
d) Signal D with 70% duty cycle

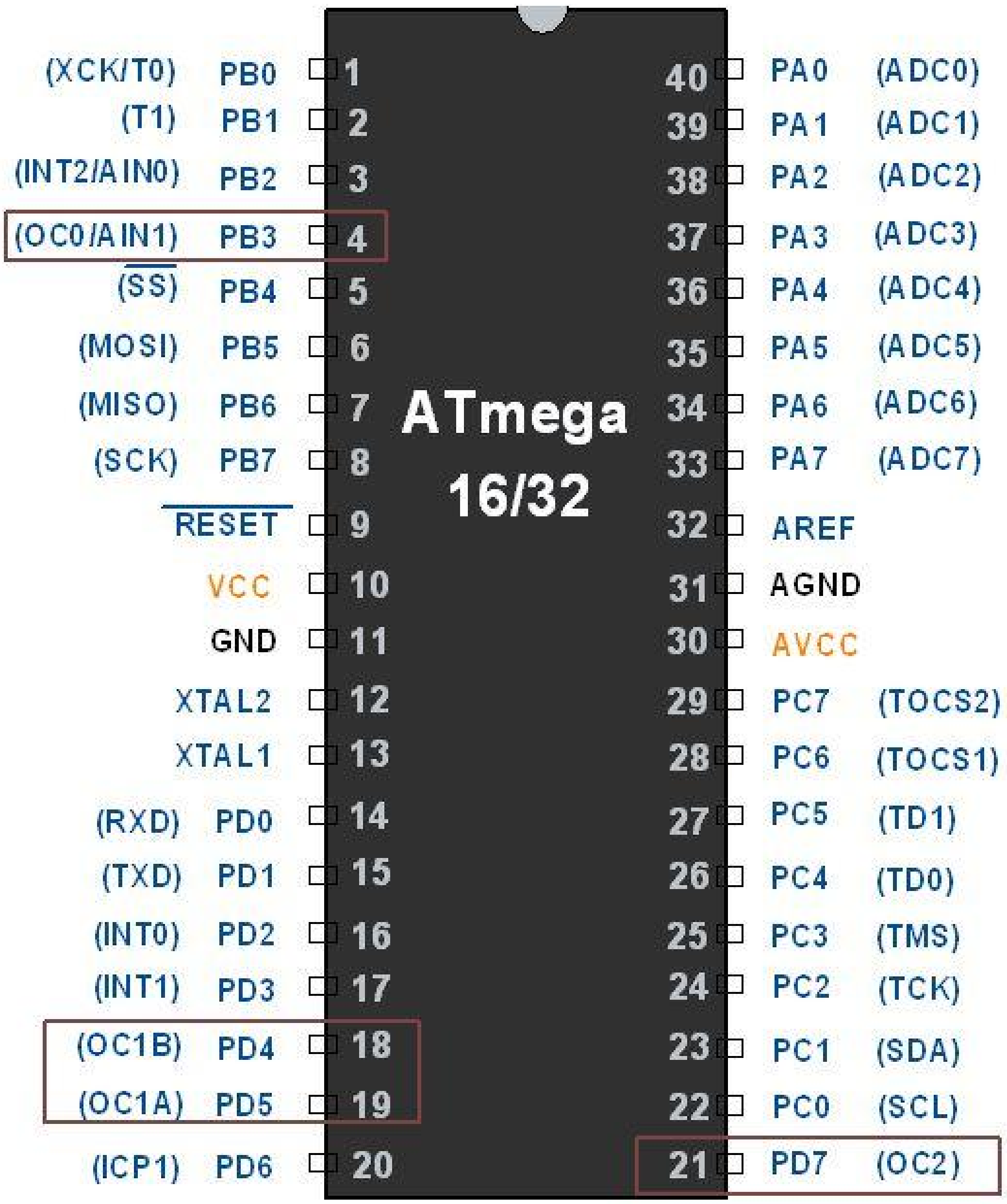
PWM Duty Cycle Waveforms

AVR ATmega PWM

ATmega has an inbuilt PWM unit. As we know, ATmega has 3 Timers T0, T1, and T2 which can be used for PWM generation. Mainly there are two modes in PWM.

1. Fast PWM
2. Phase correct PWM

We need to configure the Timer Register for generating PWM. PWM output will be generated on the corresponding Timer's output compare pin (OCx).



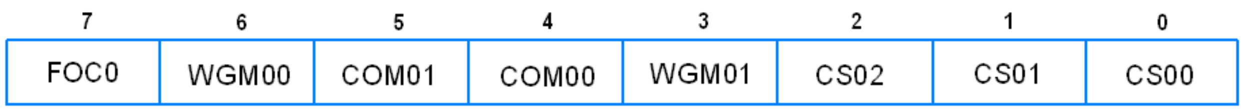
AVR

ATmega16/32 PWM Pins

Configuring Timer0 for PWM generation

It is simple to configure PWM mode in Timer. We just need to set some bits in the TCCR0 register.

TCCR0: Timer Counter Control Register 0



| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 |

Bit 7- FOC0: Force compare match

Write only bit, which can be used while generating a wave. Writing 1 to this bit will force the wave generator to act as if a compare match has occurred.

Bit 6, 3 - WGM00, WGM01: Waveform Generation Mode

| WGM00 | WGM01 | Timer0 mode selection bit |
|---|---|---|
| 0 | 0 | Normal |
| 0 | 1 | CTC (Clear timer on Compare Match) |
| 1 | 0 | PWM, Phase correct |
| 1 | 1 | Fast PWM |

Bit 5:4 - COM01:00:

1. When WGM00: WGM01= 11 i.e. Fast PWM. Compare Output Mode

waveform generator on OC0 pin

| COM01 | COM00 | Mode Name | Description |
|---|---|---|---|
| 0 | 0 | Disconnected | The normal port operation, OC0 disconnected |
| 0 | 1 | Reserved | Reserved |
| 1 | 0 | Non-inverted | Clear OC0 on compare match, set OC0 at TOP |
| 1 | 1 | Inverted PWM | Set OC0 on compare match, clear OC0 at TOP |

2. When WGM00: WGM01= 10 i.e. Phase correct PWM. Compare Output Mode waveform generator on OC0 pin

| COM01 | COM00 | Description |
|---|---|---|
| 0 | 0 | The 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 |

Bit 2:0 - CS02:CS00: Clock Source Select

These bits are used to select a clock source. When CS02: CS00 = 000, then timer is stopped. As it gets a value between 001 to 101, it gets a clock source and starts as the timer.

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer / Counter stopped) |
| 0 | 0 | 1 | clk (no pre-scaling) |
| 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. |

## Fast PWM mode

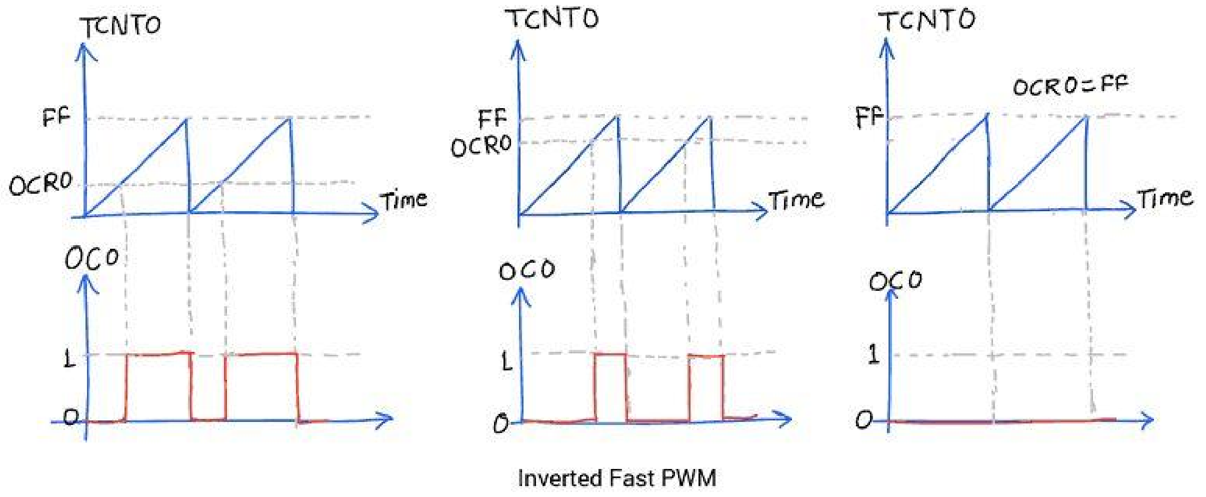
To set Fast PWM mode, we have to set WGM00: 01= 11. To generate a PWM waveform on the OC0 pin, we need to set COM01:00= 10 or 11.

COM01:00= 10 will generate Noninverting PWM output waveform and COM01:00= 11 will generate Inverting PWM output waveform. See fig.

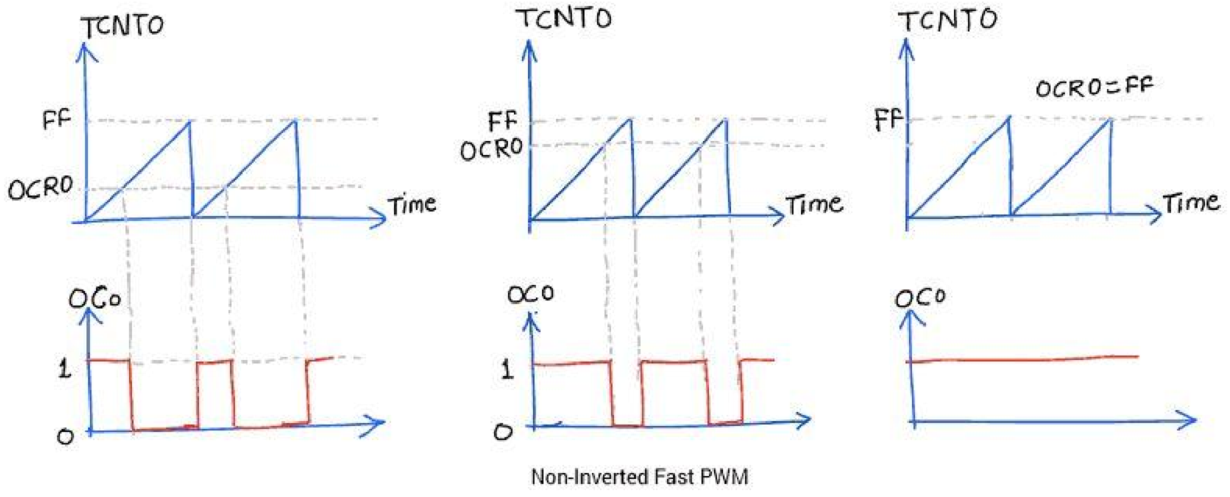
```
void PWM_init()
{
	/*set fast PWM mode with non-inverted output*/
	TCCR0 = (1<<WGM00) | (1<<WGM01) | (1<<COM01) | (1<<CS00);
	DDRB|=(1<<PB3);  /*set OC0 pin as output*/
}
```

Setting Duty cycle: we have to load value in the OCR0 register to set the duty cycle.

255 value for 100% duty cycle and 0 for 0% duty cycle. Accordingly, if we load value 127 in OCR0, the Duty cycle will be 50%.



Inverted Fast PWM

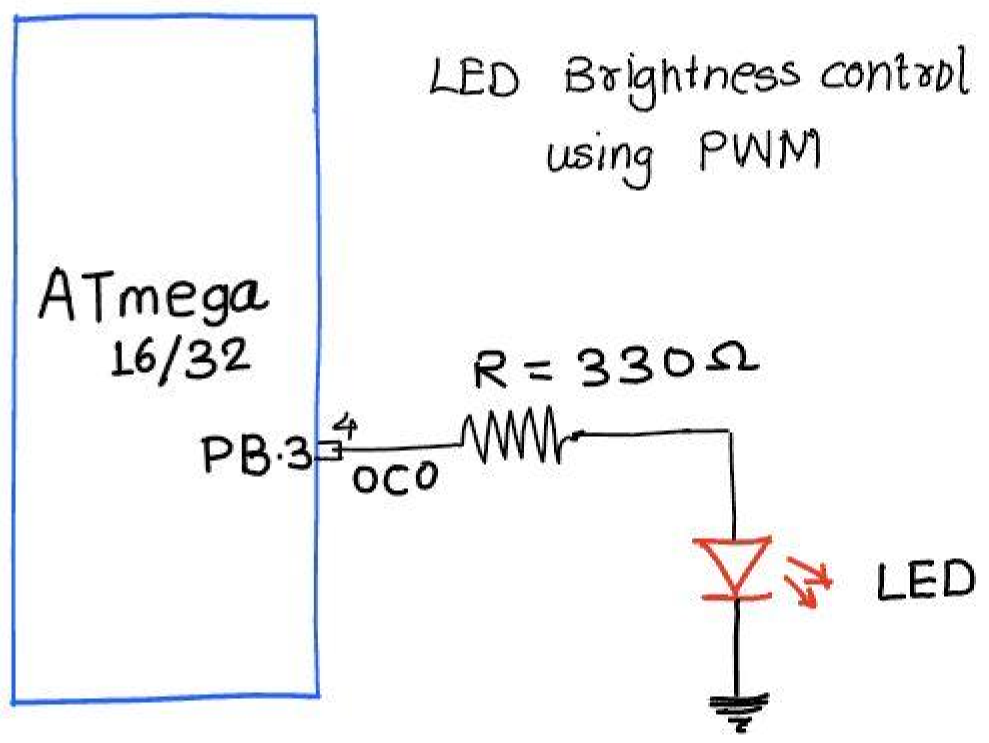


Non-Inverted Fast PWM

The advantage of using PWM mode in AVR is that it is an inbuilt hardware unit for waveform generation and once we set the PWM mode and duty cycle, this unit starts generating PWM and the controller can do other work.

Example

Control LED brightness using Fast PWM.



PWM LED

Interfacing With ATmega16/32

```
#define F_CPU 8000000UL
#include "avr/io.h"
#include <util/delay.h>


void PWM_init()
{
	/*set fast PWM mode with non-inverted output*/
	TCCR0 = (1<<WGM00) | (1<<WGM01) | (1<<COM01) | (1<<CS00);
	DDRB|=(1<<PB3);  /*set OC0 pin as output*/
}


int main ()
{
	unsigned char duty;

	PWM_init();
```

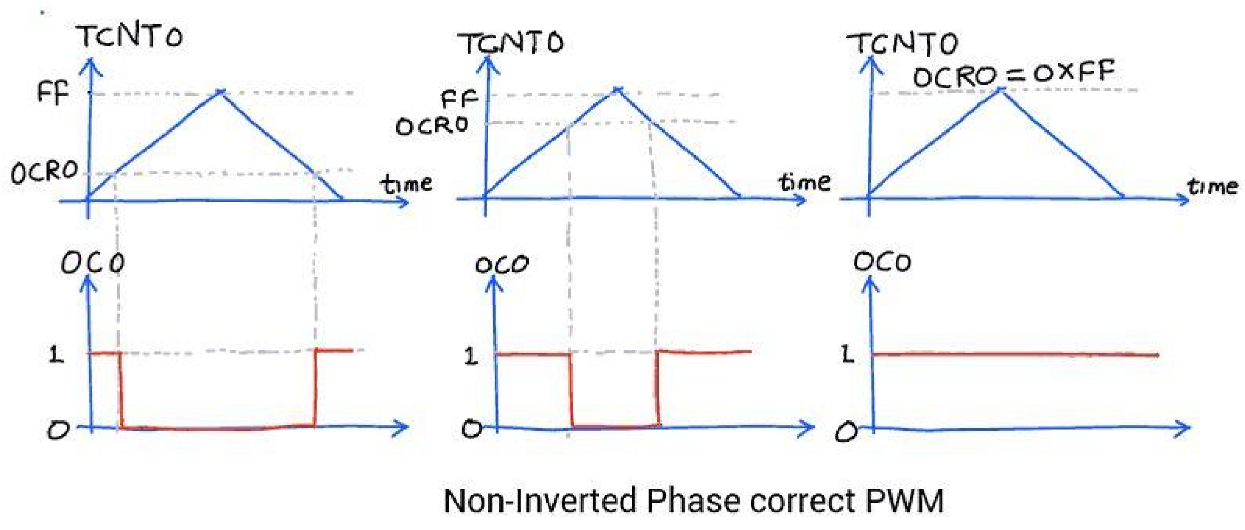
```
        while (1)
        {
                for(duty=0; duty<255; duty++)
                {
                        OCR0=duty;  /*increase the LED light intensity*/
                        _delay_ms(8);
                }
                for(duty=255; duty>1; duty--)
                {
                        OCR0=duty;  /*decrease the LED light intensity*/
                        _delay_ms(8);
                }
        }
}
```

Phase correct PWM mode

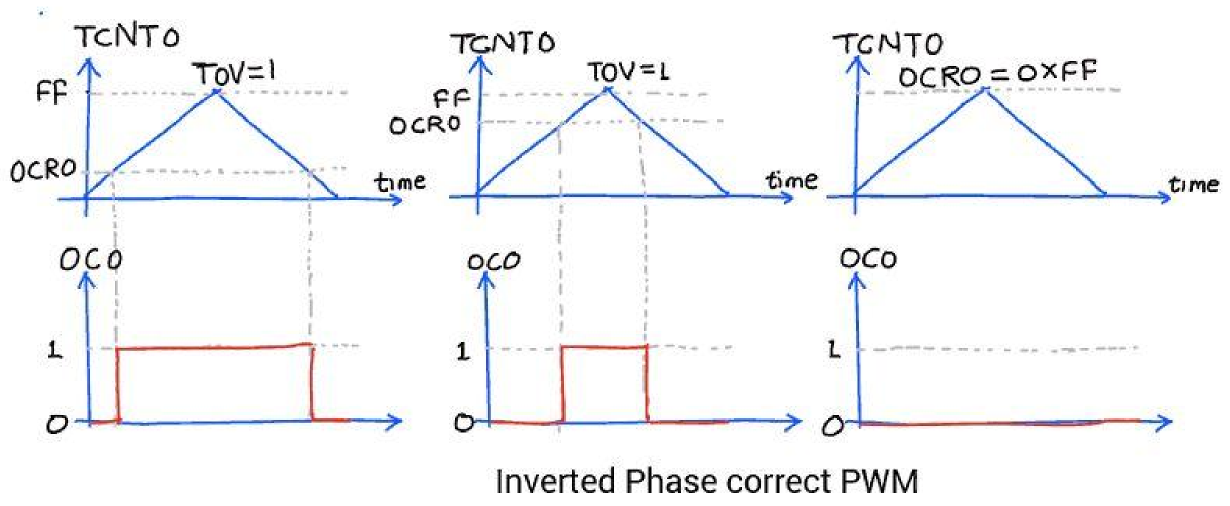
To set Phase correct PWM, we just have to set the TCCRO register as follows.

We can set the output waveform as inverted or non-inverted. See fig.

TCCR0 = (1<<WGM00) | (1<<COM01) | (1<<CS00);



Non-Inverted Phase Correct PWM



Inverted Phase Correct PWM

## 3.4 ADC programming

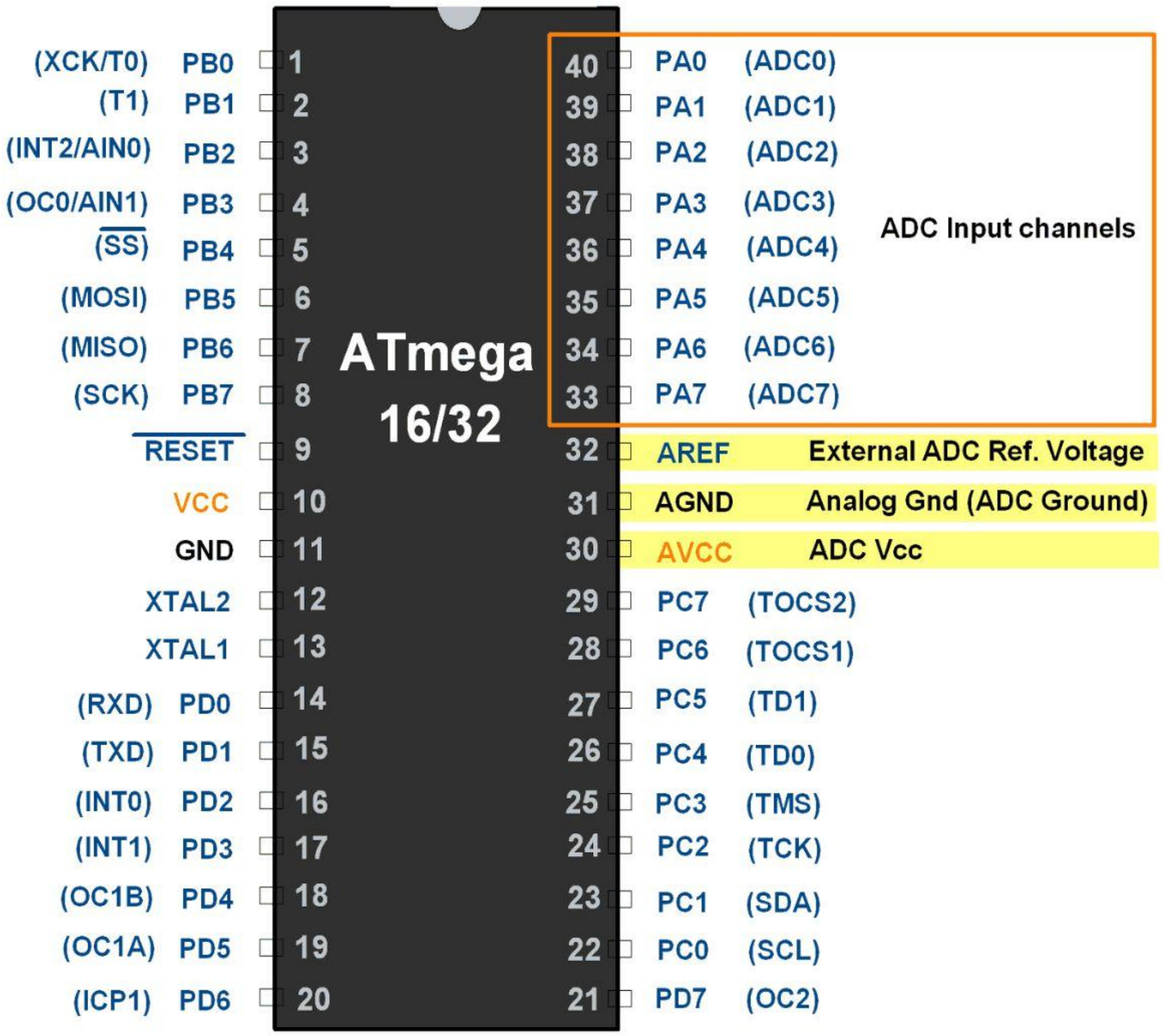
ADC in AVR ATmega16/ATmega32

Introduction to ATmega ADC

ADC (Analog to Digital converter) is the most widely used device in embedded systems which is designed especially for data acquisition. In the AVR ATmega series normally 10-bit ADC is inbuilt in the controller.

Let us see how to use the ADC of AVR ATmega16 / ATmega32.

ATmega16/32 supports eight ADC channels, which means we can connect eight analog inputs at a time. ADC channel 0 to channel 7 are present on PORTA. i.e. Pin no.33 to 40.



ADC Pins of ATmega16/32

The controller has 10 bit ADC, which means we will get digital output 0 to 1023.

i.e. When the input is 0V, the digital output will be 0V & when input is 5V (and Vref=5V), we will get the highest digital output corresponding to 1023 steps, which is 5V.

So controller ADC has 1023 steps and

- Step size with Vref=5V :  5/1023 = 4.88 mV.
- Step size with Vref=2.56 :  2.56/1023 = 2.5 mV.

So Digital data output will be Dout = Vin / step size.

## ATmega16/32 ADC Features

- It is 10-bit ADC
- Converted output binary data is held in two special functions 8-bit register ADCL (result Low) and ADCH (result in High).
- ADC gives 10-bit output, so (ADCH: ADCL) only 10-bits are useful out of 16-bits.
- We have options to use this 10-bits as upper bits or lower bits.
- We also have three options for Vref. 1. AVcc (analog Vcc), 2. Internal 2.56 v3. External Aref. Pin.
- The total conversion time depends on crystal frequency and ADPS0: 2 (frequency devisor)
- If you decided to use AVcc or Vref pin as ADC voltage reference, you can make it more stable and increase the precision of ADC by connecting a capacitor between that pin and GND.

## ATmega16/32 ADC Registers

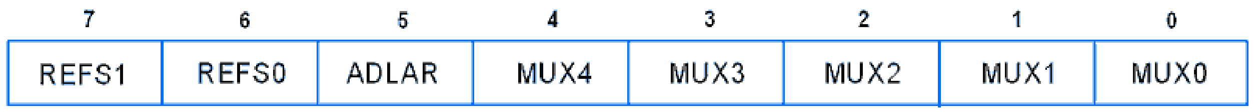
In AVR ADC, we need to understand four main register -

1. ADCH: Holds digital converted data higher byte
2. ADCL: Holds digital converted data lower byte
3. ADMUX: ADC Multiplexer selection register
4. ADCSRA: ADC Control and status register

ADCH: ADCL register

First, two-register holds the digital converted data, which is 10-bit.

ADMUX Register



| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 |

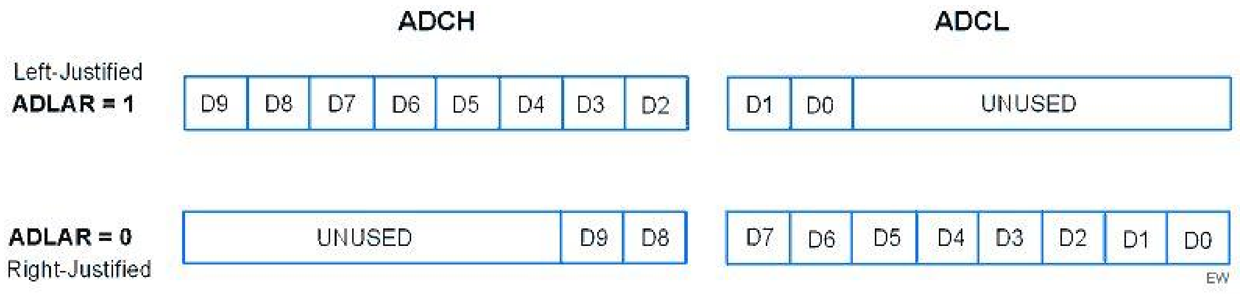
Bit 7: 6 – REFS1 : 0: Reference Selection Bits

Reference voltage selection for ADC

| REFS1 | REFS0 | Vref to ADC |
|---|---|---|
| 0 | 0 | AREF pin |
| 0 | 1 | AVCC pin i.e. Vcc 5 V |
| 1 | 0 | Reserved |
| 1 | 1 | Internal 2 |

Bit 5 – ADLAR: ADC Left Adjust Result

Use 10-bits output as upper bits or lower bits in ADCH & ADCL.



Bits 4 : 0 – MUX4 : 0: Analog Channel and Gain Selection Bits

We can select input channel ADC0 to ADC7 by using these bits. These bits are also used to select comparator (inbuilt in AVR) inputs with various gain. We will cover these comparator operations in another part.

Selecting a channel is very easy, just put the channel number in MUX4 : 0.

Suppose you are connecting the input to ADC channel 2 then put 00010 in MUX4 : 0.

Suppose you are connecting the input to ADC channel 5 then put 00101 in MUX4 : 0.

ADCSRA Register:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 |

- Bit 7 – ADEN: ADC Enable

Writing one to this bit enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- Bit 6 – ADSC: ADC Start Conversion

Writing one to this bit starts the conversion.

- Bit 5 – ADATE: ADC Auto Trigger Enable

Writing one to this bit, results in Auto Triggering of the ADC is enabled.

- Bit 4 – ADIF: ADC Interrupt Flag

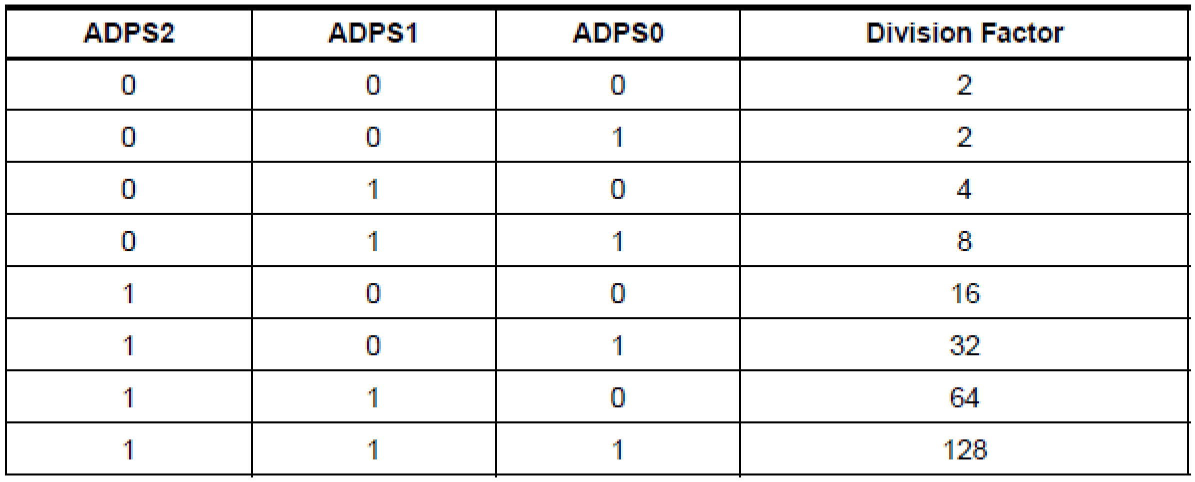
This bit is set when an ADC conversion completes and the Data Registers are updated.

- Bit 3 – ADIE: ADC Interrupt Enable

Writing one to this bit, the ADC Conversion Complete Interrupt is activated.

- Bits 2 : 0 – ADPS2 : 0: ADC Prescaler Select Bits

These bits determine the division factor between the XTAL frequency and the input clock to the ADC



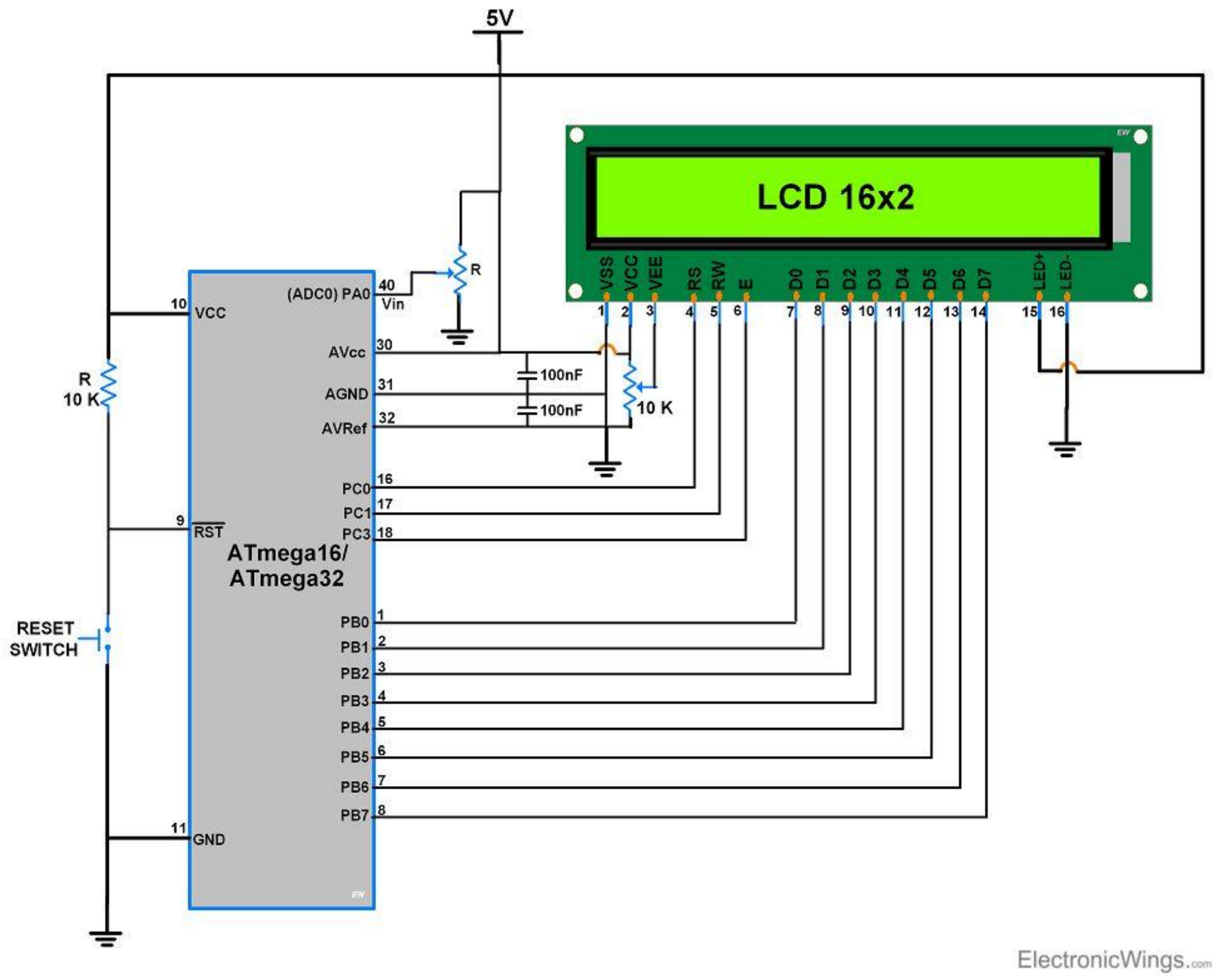
| ADPS2 | ADPS1 | ADPS0 | Division Factor |
|---|---|---|---|
| 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 2 |
| 0 | 1 | 0 | 4 |
| 0 | 1 | 1 | 8 |
| 1 | 0 | 0 | 16 |
| 1 | 0 | 1 | 32 |
| 1 | 1 | 0 | 64 |
| 1 | 1 | 1 | 128 |

We can select any divisor and set frequency Fosc/2, Fosc/4, etc. for ADC, But in AVR, ADC requires an input clock frequency less than 200KHz for max. accuracy. So we have to always take care of not exceeding ADC frequency more than 200KHz.

Suppose your clock frequency of AVR is 8MHz, then we must have to use devisor 64 or 128. Because it gives 8MHz/64 = 125KHz, which is lesser than 200KHz.

## ATmega16/32 ADC Interfacing Diagram

Here we are displaying ADC channel 0 values on the 16x2 LCD using ATmega16/32 Microcontroller



Circuit Diagram For Using ADC0 Of ATmega16/32

Steps to Program ATmega16/32 ADC

1. Make the ADC channel pin as an input.
2. Set ADC enable bit in ADCSRA, select the conversion speed using ADPS2 : 0. For example, we will select devisor 128.
3. Select ADC reference voltage using REFS1: REFS0 in ADMUX register, for example, we will use AVcc as a reference voltage.
4. Select the ADC input channel using MUX4 : 0 in ADMUX, for example, we will use channel 0.
5. So our value in register ADCSRA = 0x87 and ADMUX = 0x40.
6. Start conversion by setting bit ADSC in ADCSRA. E.g.ADCSRA |= (1<<ADSC);

7. Wait for conversion to complete by polling ADIF bit in ADCSRA register.
8. After the ADIF bit gone high, read ADCL and ADCH register to get digital output.
9. Notice that read ADCL before ADCH; otherwise result will not be valid.

## Program to read O/P digital data to PortC

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

unsigned int ADC_read(unsigned char chnl)
{

chnl= chnl & 0b00000111; // select adc channel between 0 to 7 (restricting to&

ADMUX = 0x40;        //channel A0 selected

ADCSRA|=(1<<ADSC);   // start conversion

while(!(ADCSRA & (1<<ADIF)));   // wait for ADIF conversion complete return


ADCSRA|=(1<<ADIF);   // clear ADIF when conversion complete by writing 1


return (ADC); //return calculated ADC value


}




int main(void)


{


PORTC = 0xFF;         //make PORTC as output to connect 8 leds


ADMUX=(1<<REFS0);      // Selecting internal reference voltage


ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);     // Enable ADC also set
Prescaler as 128
int i = 0; // define an integer to save adc read value

    while (1)
    {

  i = ADC_read(0);   //save adc read value in integer
          PORTC = i;        // send adc value to all pins of portc
    }
}
```