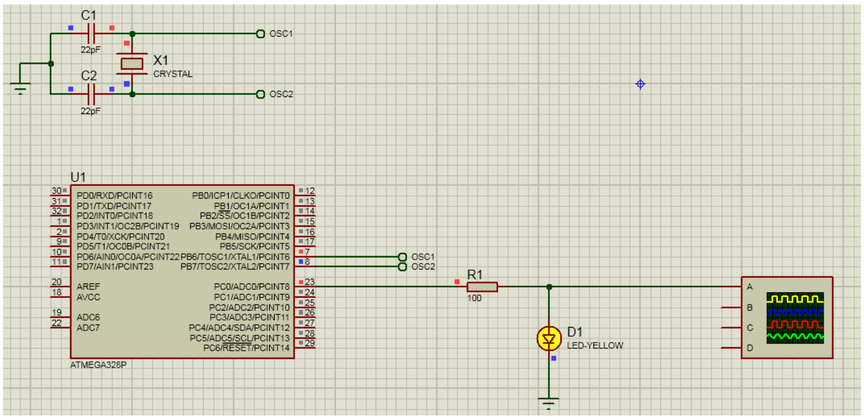
Contents

Basic Programs

1.1 BasicLedBlink

1.1.1 Circuit



1.1.2 Code

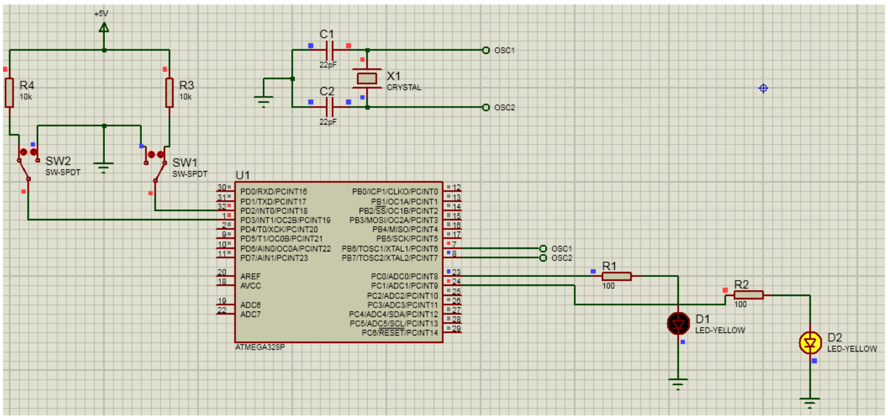
```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
        DDRC |= (1<<0);
        PORTC &= ~(1<<0);

    while(1)
    {
                PORTC |= (1<<0);
                _delay_ms(1000);
                PORTC &= ~(1<<0);
                _delay_ms(1000);
    }
}
```

1.1.3 Output

The Output can be seen @ *PC0*.

1.2 InterruptsExternal

1.2.1 Circuit



1.2.2 Code

```c
#define F_CPU 16000000L
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

void externalInterruptINT0()
{
        // making PD2 as input for INT0, though not neeced
        DDRD &= ~(1<<2);
        // enabling the internal pull-up register for PD2 for INT0
        PORTD |= (1<<2);
        // making EICRA's ISC01 and ISC00 as 10 for falling edge detection at INT0
        EICRA |= (1<<ISC01);
        EICRA &= ~(1<<ISC00);
        // making EIMSK's INT0 as 1 to enable External Interrput Request for INT0
        EIMSK |= (1<<INT0);
        // Enabling global Interrupts
        sei();
}
void externalInterruptINT1()
{
        // making PD3 as input for INT1, though not neeced
        DDRD &= ~(1<<3);
        // enabling the internal pull-up register for PD3 for INT1
        PORTD |= (1<<3);
        // making EICRA's ISC21 and ISC20 as 11 for rising edge detection at INT1
        EICRA |= ((1<<ISC11) | (1<<ISC10));
        // making EIMSK's INT2 as 1 to enable External Interrput Request for INT1
        EIMSK |= (1<<INT1);
        // Enabling global Interrupts
        sei();
}
int main(void)
{
        // making PC[1:0] as output for led
        DDRC |= 0X03;
        // PC[1:0] is made 0
        PORTC &= 0XFC;
        externalInterruptINT0();
        externalInterruptINT1();
        while(1)
        {
```

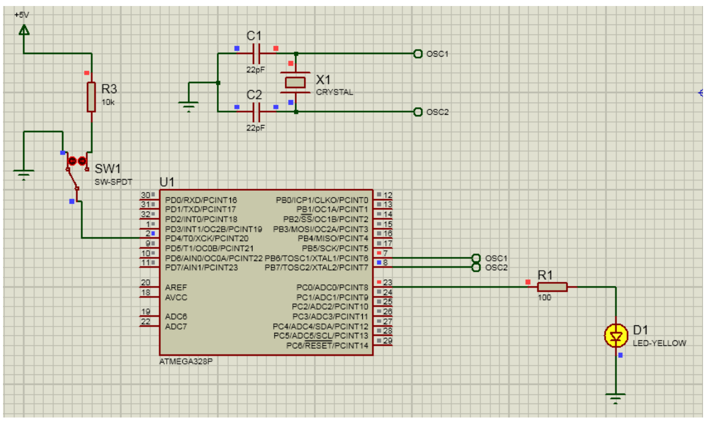
```
        }
        return 0;
}

ISR(INT0_vect)
{
        // INT0 interrupt as occured
        if((EIFR & (1<<INTF0)) != 0)
        {
                //toggle Led at pinc 0
                PINC |= (1<<0);
        }
}
ISR(INT1_vect)
{        // INT1 interrupt as occured
        if((EIFR & (1<<INTF1)) != 0)
        {
                //toggle Led at pinc 1
                PINC |= (1<<1);
        }
}
```

1.2.3 Output

The Output can be seen @ *PC0* and *PC1* when falling edge @ *INT0* and rising edge @ *INT1* occurs.

1.3 InterruptsPinChange

1.3.1 Circuit



1.3.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

void pinChangeInterrupt_PCINT20()
{
        // making PD4 as input for PCI20
        DDRD &= ~(1<<4);
        // enabling the internal pull-up register for PD4 for PCI20
        PORTD |= (1<<4);
         // Selecting the PCINT20 for PCI2 intterupt
        PCMSK2 |= (1<<PCINT20);
```

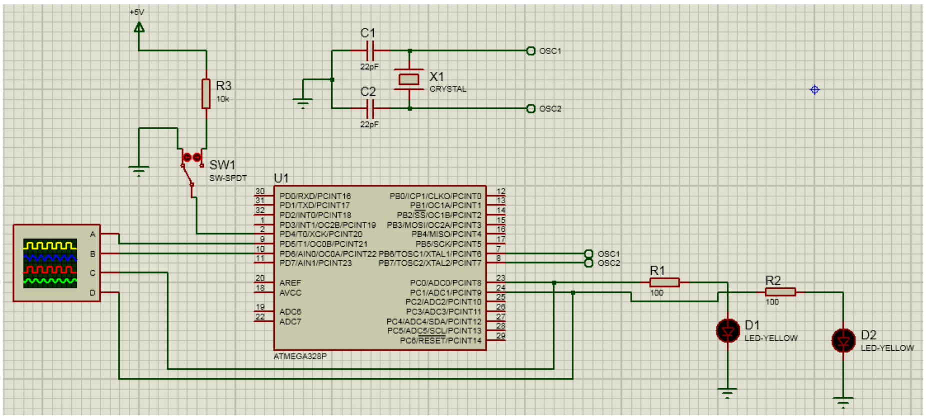
```
        // Enabling the PCI2 interrupt
        PCICR |= (1<<PCIE2);
        // Enabling global Interrupts
        sei();
}
int main(void)
{
        DDRC |= 0X01; // making PC0 as output for led
        PORTC &= 0XFE; // PC0 is made 0
        pinChangeInterrupt_PCINT20();
        while(1)
        {
        }
        return 0;
}

ISR(PCINT2_vect)
{
        // PCI2 interrupt as occured
        if((PCIFR & (1<<PCIF2)) != 0)
        {
                //toggle Led at pinc 0
                PINC |= (1<<0);
        }
}
```

1.3.3 Output

The Output can be seen @ *PC0* when pin change @ *PCINT20* occurs.

1.4 TimerCounter0_NormalMode

1.4.1 Circuit



1.4.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer0_asTimer()
{
        /* TCNT0 starts from 0X00 goes upto 0XFF and restarts */
        /* No possible use case as it just goes upto 0xFF and restarts */
        // MOde of operation to Normal Mode -- WGM0[2:0] === 000
```

```
        /* WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A,
         WGM0[0](bit0)  from TCCR0A*/
        TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
        TCCR0B = TCCR0B & ~(1<<3);
        /* What to do when timer reaches the MAX(0xFF) value */
        // toggle OC0A and OC0B on each time when reaches the MAX(0xFF)
        // which is reflected in PD6 and PD5
        // Output OC0A to toglle when reaches MAX -- COM0A[1:0] === 01
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<7);
        TCCR0A = TCCR0A | (1<<6);
    // Output OC0B to toglle when reaches MAX -- COM0B1:0] === 01
        // COM0B[1](bit7) from TCCR0A, COM0B[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<5);
        TCCR0A = TCCR0A | (1<<4);
        //Enable Interrupt of OVERFLOW flag so that interrupt can be generated
        TIMSK0 = TIMSK0 | (1<<0);
        // start timer by setting the clock prescalar
        // DIVIDE BY 8 from I/O clock
        // DIVIDE BY 8-- CS0[2:0] === 010
        /* CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,
        CS0[0](bit0) from TCCR0B*/
        TCCR0B = TCCR0B | (1<<1);
        TCCR0B = TCCR0B & (~(1<<0) & ~(1<<2));
        // enabling global interrupt
        sei();
        // SO ON TIME = max_count / (F_CPU / PRESCALAR)
        // ON TIME = 0xFF / (16000000/8) = 128us
        // since symmetric as toggling OFF TIME = 128us
        // hence, we get a square wave of fequency 1 / 256us = 3.906kHz
}
void Timer0_asCounter()
{
        /* TCNT0 starts from 0X00 goes upto 0XFF and restarts */
        /* No possible use case as it just goes upto 0xFF and restarts */
        // MOde of operation to Normal Mode -- WGM0[2:0] === 000
        /* WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A,
        WGM0[0](bit0)  from TCCR0A*/
        TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
        TCCR0B = TCCR0B & ~(1<<3);
        /* to count external event -we must connect source to T0 (PD4) */
        // THE CLK IS CLOCKED FROM external source
        // Falling edge of T0(PD4) -- CS0[2:0] === 110
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<2);
        TCCR0B = TCCR0B | (1<<1);
        TCCR0B = TCCR0B & ~(1<<0);
}
void Timer0_asDelay()
{
        /* TCNT0 starts from 0X00 goes upto 0XFF and restarts */
        /* No possible use case as it just goes upto 0xFF and restarts */
        // MOde of operation to Normal Mode -- WGM0[2:0] === 000
        /* WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A,
         WGM0[0](bit0)  from TCCR0A*/
        TCCR0A = TCCR0A & (~(1<<0) & ~(1<<1));
        TCCR0B = TCCR0B & ~(1<<3);
        /* What to do when timer reaches the MAX(0xFF) value */
        // nothing should be done on OC0A for delay
        // nothing  -- COM0A[1:0] === 00
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<7);
        TCCR0A = TCCR0A & ~(1<<6);
```

```c
        /* The delay possible = 0xff / (F_CPU/prescalar) */
        // lowest delay = 0xff / (16000000 / 1) = 16us
        // when prescalar == 8 --> delay = 0xff / (16000000 / 8) = 128us
        // when prescalar == 64 --> delay = 0xff / (16000000 / 64) = 1.024ms
        // when prescalar == 256 --> delay = 0xff / (16000000 / 256) = 4.096ms
        // highest delay possible = 0xff / (16000000 / 1024) = 16.38ms

        // start timer by setting the clock prescalar
        // DIVIDE BY 8 use the same clock from I/O clock
        // DIVIDE BY 8-- CS0[2:0] === 010
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B & ~(1<<0);
        TCCR0B = TCCR0B | (1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        // actual delaying - wait until delay happens
        while((TIFR0 & 0x01) == 0x00); // checking overflow flag when overflow happns
        // clearing the overflag so that we can further utilize
        TIFR0 = TIFR0 | 0x01;
}
int main(void)
{
        // making the PD5 and PD6 as output
        DDRD = DDRD | (1<<6) | (1<<5);
        DDRD = DDRD & ~(1<<4);
        DDRC |= (1<<0) | (1<<1);
        PORTC &= ~(1<<0);
        // Timer0_asTimer();
        // Timer0_asCounter();
        while(1)
        {
                PORTC &= ~(1<<0);
                Timer0_asDelay();
                PORTC |= (1<<0);
                Timer0_asDelay();
        }
}
ISR(TIMER0_OVF_vect)
{
        // toggle PC1 when overflows
        PINC |= (1<<1);
}
```

1.4.3 Output

Timer0_asTimer

- The output can be seen @ *OC0A* and *OC0B* pins with a on time of 128μs and off time of 128μs ($\frac{0xFF*8}{16000000} = 127.5\mu s$).
- Also, *PC1* toglles for the overflow Timer0.

Timer0_asCounter

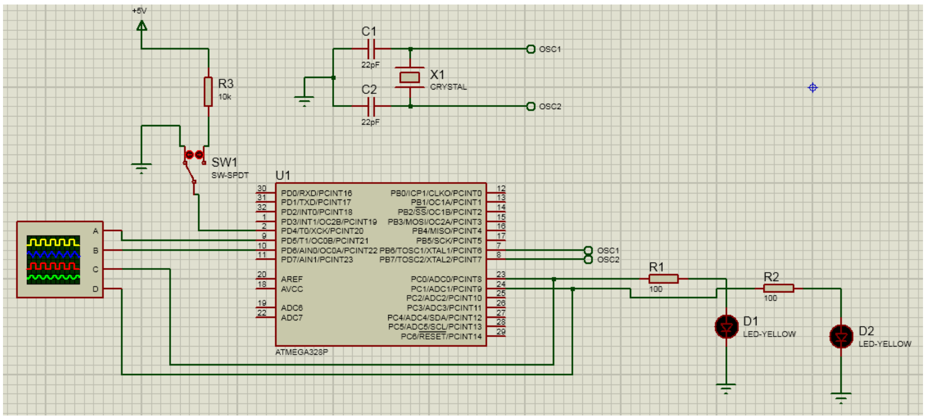
- The output can be seen @ Watch Window and see the **TCNT0** register when pulsed @ *T0* pin.

Timer0_asDelay

- The output can be seen *PC0* pin.

1.5 TimerCounter0_CTC

1.5.1 Circuit



1.5.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer0_asTimer()
{
        /* TCNT0 starts from 0X00 goes upto OCR0A and restarts */
        // MOde of operation to CTC Mode -- WGM0[2:0] === 010
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A & ~(1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
        /* What to do when timer reaches the OCR0A */
        // toggle OC0A on each time when reaches the OCR0A
        // which is reflected in PD6
        // Output OC0A to toglle when reaches MAX -- COM0A[1:0] === 01
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<7);
        TCCR0A = TCCR0A | (1<<6);
        // Output OC0B to toglle when reaches MAX -- COM0B[1:0] === 01
        // COM0B[1](bit7) from TCCR0A, COM0B[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<5);
        TCCR0A = TCCR0A | (1<<4);
        // Enable Interrupt when counter matches OCR0A Rgister
        //  OCIE0A bit is enabled
        TIMSK0 = TIMSK0 | (1<<1);
        // setting the value till the counter should reach in OCR0A
        // for toggling of OC0A pin
        OCR0A = 0x32;
        // start timer by setting the clock prescalar
        // DIVIDE BY 8 from I/O clock
        // DIVIDE BY 8-- CS0[2:0] === 010
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<1);
        TCCR0B = TCCR0B & (~(1<<0) & ~(1<<2));
        // enabling global interrupt
        sei();
        // SO ON TIME = (1 + OCR0A) / (F_CPU / PRESCALAR)
        // ON TIME = 0X32 / (16000000/8) = 25.5us
        // since symmetric as toggling OFF TIME = 25.5us
        // hence, we get a square wave of fequency 1 / 50us = 20kHz
}
```

```
void Timer0_asCounter()
{
        /* TCNT0 starts from 0X00 goes upto OCR0A and restarts */
        // MOde of operation to CTC Mode -- WGM0[2:0] === 010
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A & ~(1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
        // Disbale Interrupt when counter matches OCR0A Rgister
        //  OCIE0A bit is disabled
        TIMSK0 = TIMSK0 | (1<<1);
        //we count till OCR0A register value and reset and continue
        OCR0A = 0xA;
        /* to count external event -we must connect source to T0 (PD4) */
        // THE CLK IS CLOCKED FROM external source
        // Falling edge of T0(PD4) -- CS0[2:0] === 110
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<2);
        TCCR0B = TCCR0B | (1<<1);
        TCCR0B = TCCR0B & ~(1<<0);
        //enable global interrupt
        sei();
}
void Timer0_asDelayIn_ms(uint32_t delayInMs)
{
        // minimum delay being 4us -- choose like that
        // PRESCALAR OF 1 -- 3us - 16us -- usage 3us - 16us -- factor=0 -- CS0[2:0]=1
        // PRESCALAR OF 8 -- 3us - 128us -- usage 17us - 128us -- factor=3 -- CS0[2:0]=2
        // PRESCALAR OF 64 -- 4us - 1.024ms -- usage 129us - 1024us -- factor=6 -- CS0[2:0]=3
        // PRESCALAR OF 256 -- 16us - 4.096ms -- usage 1025us - 4096us -- factor=8 -- CS0[2:0]=4
        // MOde of operation to ctc Mode -- WGM0[2:0] === 010
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A & ~(1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
        while(delayInMs--)
        {
                // for 1ms delay
                OCR0A = 249;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS0[2:0] === 011
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B | (1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
                // actual delaying - wait until delay happens
                // checking OCF0A (compare match flag A) flag when match happns
                while((TIFR0 & 0x02) == 0x00);
                // clearing the compare match flag so that we can further utilize
                TIFR0 = TIFR0 | 0x02;
        }
}
int main(void)
{
        // making the PD5 and PD6 as output
        DDRD = DDRD | (1<<6) | (1<<5);
        DDRD = DDRD & ~(1<<4);
        DDRB |= (1<<0);
        DDRC |= (1<<0) | (1<<1);
        PORTC &= ~(1<<0);
        // Timer0_asTimer();
        // Timer0_asCounter();
```

```
        while(1)
        {
                PORTC &= ~(1<<0);
                Timer0_asDelayIn_ms(100);
                PORTC |= (1<<0);
                Timer0_asDelayIn_ms(100);
        }
}

ISR(TIMER0_COMPA_vect)
{
        // toggle PC1 when matches
        PINC |= (1<<1);
}
```

1.5.3 Output

Timer0_asTimer

- The output can be seen @ *OC0A* and *OC0B* pins with a on time of $25.5\mu s$ and off time of $25.5\mu s$ ($\frac{(0x32+1)*8}{16000000} = 25.5\mu s$).
- Also, *PC1* toglles for the **TCNT0** matches **OCR0A**.

Timer0_asCounter

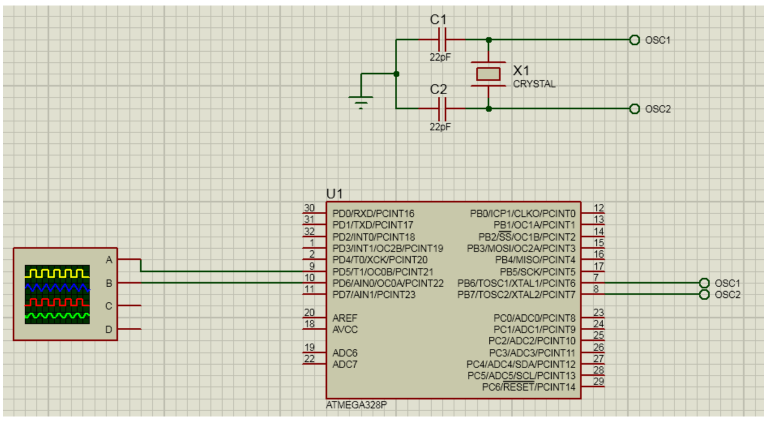
- The output can be seen @ Watch Window and see the **TCNT0** register when pulsed @ *T0* pin.
- Also, the *PC1* pin toggles for every 10 changes at *T0* pin.

Timer0_asDelayIn_ms

- The output can be seen *PC0* pin.

1.6 TimerCounter0_FastPWM

1.6.1 Circuit



1.6.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer0_NonInverting_TOP_at_MAX()
{
        /* TCNT0 starts from 0x00 to TOP and reaches to 0X00 \
        Here, TOP is defined by WGM0[2] bit
```

```
        0 -- TOP = 0xFF
        1 -- TOP = OCR0A
        for top begin max we select WGM0[2:0] = 011        */
        /* The frequnecy of PWM is fixed based just on the prescalare
        becase, the TCN0 reaches from 0X00 to 0XFF
        hence, Based on the prescalling possiblily{1,8,64,256,1024}
        we have just 5 Frequnecies possible */
        /* But, we get two PWM's using OCR0A and OCR0B
        A) choosing,  10 - Clear OC0A on compare match. Set OC0A at BOTTOM.
        will lead to on-time = OCR0A
        B) choosing,  11 - Set OC0A on compare match. Clear OC0A at BOTTOM.
         will lead to off-time = OCR0A
        A) choosing,  10 - Clear OC0B on compare match. Set OC0B at BOTTOM.
         will lead to on-time = OCR0B
        B) choosing,  11 - Set OC0B on compare match. Clear OC0B at BOTTOM.
         will lead to off-time = OCR0B*/
        // MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 011
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
    // here we set COM0A[1:0] as 10 for non-inverting
        // here we set COM0B[1:0] as 10 for non-inverting
        // which is reflected in PD6
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A | (1<<7);
        TCCR0A = TCCR0A & ~(1<<6);
        // which is reflected in PD65
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
    // Enable Interrupt when TCN0 overflows TOP - here 0xFF
        //  TOV0 bit is enabled
        TIMSK0 = TIMSK0 | (1<<0);
        /* we use OCF0A flag - which is set at every time TCN0 reaches OCR0A
           here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR0A*/
        TIMSK0 = TIMSK0 | (1<<1);
        /* we use OCF0B flag - which is set at every time TCN0 reaches OCR0B
           here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR0B*/
        TIMSK0 = TIMSK0 | (1<<2);
        // Next we set values for OCR0A and OCR0B
        /* Since, TCNT0 goes till max(0xFF), we can choose OCR0A
         and OCR0B to any value below max(0xFFF)*/
        OCR0A = 0x19; // for 10% duty clcle
        OCR0B = 0xC0; // for 75% duty clcle
    // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_Inverting_TOP_at_MAX()
{
    // MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 011
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
    // here we set COM0A[1:0] as 11 for inverting
```

```
        // here we set COMOB[1:0] as 11 for inverting
        // which is reflected in PD6
        // COMOA[1](bit7) from TCCROA, COMOA[0](bit6) from TCCROA
        TCCROA = TCCROA | (1<<7);
        TCCROA = TCCROA | (1<<6);
        // which is reflected in PD65
        // COMOB[1](bit5) from TCCROA, COMOB[0](bit4) from TCCROA
        TCCROA = TCCROA | (1<<5);
        TCCROA = TCCROA | (1<<4);
    // Enable Interrupt when TCNO overflows TOP - here 0xFF
        //  TOVO bit is enabled
        TIMSKO = TIMSKO | (1<<0);
        /* we use OCFOA flag - which is set at every time TCNO reaches OCROA
           here we clear led(PC1),  so that we obtain the PWM when TCNO reaches OCROA*/
        TIMSKO = TIMSKO | (1<<1);
        /* we use OCFOB flag - which is set at every time TCNO reaches OCROB
           here we clear led(PC2),  so that we obtain the PWM when TCNO reaches OCROB*/
        TIMSKO = TIMSKO | (1<<2);
        // Next we set values for OCROA and OCROB
        /* Since, TCNTO goes till max(0xFF), we can choose OCROA
        and OCROB to any value below max(0xFFF)*/
        OCROA = 0x19; // for 10% duty clcle
        OCROB = 0xC0; // for 75% duty clcle
    // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCROB,CS0[1](bit1) from TCCROB,CS0[0](bit0) from TCCROB
        TCCROB = TCCROB | (1<<0);
        TCCROB = TCCROB & ~(1<<1);
        TCCROB = TCCROB & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_NonInverting_TOP_at_OCROA()
{
    // MOde of operation to fast_pwm_top_max Mode -- WGMO[2:0] === 111
        // WGMO[2](bit3) from TCCROB, WGMO[1](bit1)  from TCCROA, WGMO[0](bit0)  from TCCROA
        TCCROA = TCCROA | (1<<0);
        TCCROA = TCCROA | (1<<1);
        TCCROB = TCCROB | (1<<3);
        // here we set COMOB[1:0] as 10 for non-inverting
        // which is reflected in PD5
        // COMOB[1](bit5) from TCCROA, COMOB[0](bit4) from TCCROA
        TCCROA = TCCROA | (1<<5);
        TCCROA = TCCROA & ~(1<<4);
        // Next we set values for OCROA and OCROB
        // Since, TCNTO goes till OCROA, we can choose OCROB to any value below OCROA
        OCROA = 0x70; // for freqeuncy
        OCROB = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCROB,CS0[1](bit1) from TCCROB,CS0[0](bit0) from TCCROB
        TCCROB = TCCROB | (1<<0);
        TCCROB = TCCROB & ~(1<<1);
        TCCROB = TCCROB & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_Inverting_TOP_at_OCROA()
{
    // MOde of operation to fast_pwm_top_max Mode -- WGMO[2:0] === 111
    // WGMO[2](bit3) from TCCROB, WGMO[1](bit1)  from TCCROA, WGMO[0](bit0)  from TCCROA
```

```c
    TCCR0A = TCCR0A | (1<<0);
    TCCR0A = TCCR0A | (1<<1);
    TCCR0B = TCCR0B | (1<<3);
    // here we set COM0B[1:0] as 11 for inverting
    // which is reflected in PD5
    // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
    TCCR0A = TCCR0A | (1<<5);
    TCCR0A = TCCR0A | (1<<4);
    // Next we set values for OCR0A and OCR0B
    // Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
    OCR0A = 0x70; // for freqeuncy
    OCR0B = 0x60; // for pwm duty cylc
    // start the timer by selecting the prescalr
    //  use the same clock from I/O clock
    //  CS0[2:0] === 001
    // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
    TCCR0B = TCCR0B | (1<<0);
    TCCR0B = TCCR0B & ~(1<<1);
    TCCR0B = TCCR0B & ~(1<<2);
    //enabled global interrupt
    sei();
}
void Timer0_OC0A_Square()
{
    // MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
    // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
    TCCR0A = TCCR0A | (1<<0);
    TCCR0A = TCCR0A | (1<<1);
    TCCR0B = TCCR0B | (1<<3);
    // here we set COM0B[1:0] as 01 for toggling of OC0A
    // which is reflected in PD6
    // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
    TCCR0A = TCCR0A & ~(1<<7);
    TCCR0A = TCCR0A | (1<<6);
    // Next we set values for OCR0A and OCR0B
    // Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
    OCR0A = 0x70; // for freqeuncy
    // start the timer by selecting the prescalr
    //  use the same clock from I/O clock
    //  CS0[2:0] === 001
    // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
    TCCR0B = TCCR0B | (1<<0);
    TCCR0B = TCCR0B & ~(1<<1);
    TCCR0B = TCCR0B & ~(1<<2);
    //enabled global interrupt
    sei();
}
void Timer0_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
        uint32_t total_time = on_time_us + off_time_us;
        // MOde of operation to fast_pwm_top_max Mode -- WGM0[2:0] === 111
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A | (1<<1);
        TCCR0B = TCCR0B | (1<<3);
        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock
                OCR0A = 0;
```

```
				// start timer by setting the clock prescalar
				//  use the same clock from I/O clock
				//  CS0[2:0] === 001
				// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
				TCCR0B = TCCR0B & ~(1<<0);
				TCCR0B = TCCR0B & ~(1<<1);
				TCCR0B = TCCR0B & ~(1<<2);
		}
		else if((3 < total_time)  && (total_time <= 16))
		{
				OCR0A = ((total_time * 16) >> 0) - 1;
				OCR0B = ((on_time_us * 16) >> 0) - 1;
				// start timer by setting the clock prescalar
				//  use the same clock from I/O clock
				//  CS0[2:0] === 001
				// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
				TCCR0B = TCCR0B | (1<<0);
				TCCR0B = TCCR0B & ~(1<<1);
				TCCR0B = TCCR0B & ~(1<<2);
		}
		else if((16 < total_time)  && (total_time <= 128))
		{
				OCR0A = ((total_time * 16) >> 3) - 1;
				OCR0B = ((on_time_us * 16) >> 3) - 1;
				// start timer by setting the clock prescalar
				//  dived by 8 from I/O clock
				//  CS0[2:0] === 010
				// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
				TCCR0B = TCCR0B & ~(1<<0);
				TCCR0B = TCCR0B | (1<<1);
				TCCR0B = TCCR0B & ~(1<<2);
		}
		else if((128 < total_time)  && (total_time <= 1024))
		{
				OCR0A = ((total_time * 16) >> 6) - 1;
				OCR0B = ((on_time_us * 16) >> 6) - 1;
				// start timer by setting the clock prescalar
				//  dived by 64 from I/O clock
				//  CS0[2:0] === 011
				// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
				TCCR0B = TCCR0B | (1<<0);
				TCCR0B = TCCR0B | (1<<1);
				TCCR0B = TCCR0B & ~(1<<2);
		}
		else if((1024 < total_time)  && (total_time <= 4096))
		{
				OCR0A = ((total_time * 16) >> 8) - 1;
				OCR0B = ((on_time_us * 16) >> 8) - 1;
				// start timer by setting the clock prescalar
				//  divide by256 from I/O clock
				//  CS0[2:0] === 100
				// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
				TCCR0B = TCCR0B & ~(1<<0);
				TCCR0B = TCCR0B & ~(1<<1);
				TCCR0B = TCCR0B | (1<<2);
		}
		else if(total_time > 4096)
		{
				// dont' cross more than 4.096ms
		}
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
```

```c
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }
        // max time = 4ms -- min freqency = 250 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer0_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
        DDRD = DDRD | (1<<6) | (1<<5);
        // Timer0_NonInverting_TOP_at_MAX();
        // Timer0_Inverting_TOP_at_MAX();
    // Timer0_NonInverting_TOP_at_OCR0A();
    // Timer0_Inverting_TOP_at_OCR0A();
    // Timer0_OC0A_Square();
    PWMGeneration(12, 1000);
    while(1)
    {
    }
}
ISR(TIMER0_OVF_vect)
{
}
ISR(TIMER0_COMPA_vect)
{
}
ISR(TIMER0_COMPB_vect)
{
}
```

1.6.3 Output

Timer0_NonInverting_TOP_at_MAX

- The output can be seen @ *OC0A* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of 10% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC0B* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of 75% ($\frac{75}{100} = 0xC0$).

Timer0_Inverting_TOP_at_MAX

- The output can be seen @ *OC0A* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC0B* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of (100 - 75)% ($\frac{75}{100} = 0xC0$).

Timer0_NonInverting_TOP_at_OCR0A

- The output can be seen @ *OC0B* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$) and duty cycle of 85% ($\frac{85}{100} = 0x60$).

Timer0_Inverting_TOP_at_OCR0A

- The output can be seen @ *OC0B* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$) and duty cycle of (100 - 85)% ($\frac{85}{100} = 0x60$).

Timer0_OC0A_Square

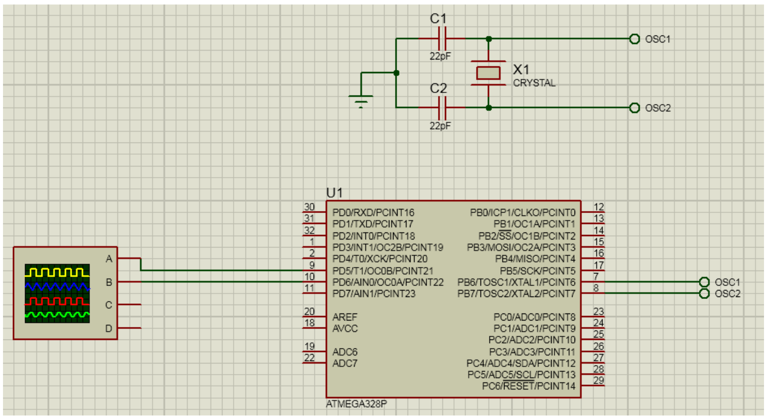
- The output can be seen @ *OC0A* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$).

PWMGeneration

- The output can be seen @ *OC0B*.

1.7 TimerCounter0_PhaseCorrectedPWM

1.7.1 Circuit



1.7.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer0_NonInverting_TOP_at_MAX()
{
        /*  Since dual slope the frequency is twice that of the frequency -
        - see documentation
        TCNTO up counts from 0x00 to TOP and down counts to 0X00
        Here, TOP is defined by WGM0[2] bit
        0 -- TOP = 0xFF
        1 -- TOP = OCR0A
        // for top begin max we select WGM0[2:0] = 001*/
        /* The frequncy of PWM is fixed frrquency based just on
        the prescalar because, the TCN0 up counts from 0X00 to 0XFF
         and from down counts from 0XFF to 0x00
        hence, Based on the prescalling possiblily{1,8,64,256,1024}
         we have just 5 Frequnecies possible */
        /* But, we get two PWM's using OCR0A and OCR0B
        A) choosing,  10 - Clear OC0A on compare match. Set OC0A at BOTTOM.
        will lead to on-time = OCR0A
        B) choosing,  11 - Set OC0A on compare match. Clear OC0A at BOTTOM.
        will lead to off-time = OCR0A
        A) choosing,  10 - Clear OC0B on compare match. Set OC0B at BOTTOM.
        will lead to on-time = OCR0B
        B) choosing,  11 - Set OC0B on compare match. Clear OC0B at BOTTOM.
        will lead to off-time = OCR0B        */
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 001
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
        /* in timer0_phase_pwm_top_max, only two possiblites are there
         for COM0B[1:0] and COM0A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
        // here we set COM0A[1:0] as 10 for non-inverting
        // here we set COM0B[1:0] as 10 for non-inverting
        // which is reflected in PD6
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A | (1<<7);
```

```
        TCCR0A = TCCR0A & ~(1<<6);
        // which is reflected in PD65
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
        /* we use overflow flag -- which is set at every time
        TCN0 reaches TOP here 0xFF
        here, we toggle an led(PC0) at every overflow interrupt -
         this led(PC0) would give the frequency of PWM being generated --
         done by PINC = PINC | 0X01;
        Also, we set the other leds(PC1 and PC2) so
        that they are make one when TCN0 reaches 0x00 */
        // Enable Interrupt when TCN0 overflows TOP - here 0xFF
        //  TOV0 bit is enabled
        TIMSK0 = TIMSK0 | (1<<0);
        // Next we set values for OCR0A and OCR0B
        /* Since, TCNT0 goes till max(0xFF), we can choose
         OCR0A and OCR0B to any value below max(0xFFF)*/
        OCR0A = 0x19; // for 10% duty clcle
        OCR0B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_Inverting_TOP_at_MAX()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 001
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<3);
        /* in timer0_phase_pwm_top_max, only two possiblites are there for COM0B[1:0]
         and COM0A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
        // here we set COM0A[1:0] as 11 for inverting
        // here we set COM0B[1:0] as 11 for inverting
        // which is reflected in PD6
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A | (1<<7);
        TCCR0A = TCCR0A & ~(1<<6);
        // which is reflected in PD65
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
        /* we use overflow flag -- which is set at every time TCN0 reaches TOP here 0xFF
        here, we toggle an led(PC0) at every overflow interrupt - this led(PC0) would
        give the frequency of PWM being generated -- done by PINC = PINC | 0X01;
        Also, we set the other leds(PC1 and PC2) so that they are
        make one when TCN0 reaches 0x00 */
        // Enable Interrupt when TCN0 overflows TOP - here 0xFF
        //  TOV0 bit is enabled
        TIMSK0 = TIMSK0 | (1<<0);
        // Next we set values for OCR0A and OCR0B
        // Since, TCNT0 goes till max(0xFF), we can choose OCR0A and
        OCR0B to any value below max(0xFFF)
        OCR0A = 0x19; // for 10% duty clcle
        OCR0B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
```

```c
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_NonInverting_TOP_at_OCR0A()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B | (1<<3);
        // here we set COM0A[1:0] as 10 for non-inverting
        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
        // Next we set values for OCR0A and OCR0B
        // Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
        OCR0A = 0x70; // for freqeuncy
        OCR0B = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_Inverting_TOP_at_OCR0A()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B | (1<<3);
        // here we set COM0A[1:0] as 11 for inverting
        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A | (1<<4);
        // Next we set values for OCR0A and OCR0B
        // Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
        OCR0A = 0x70; // for freqeuncy
        OCR0B = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_OC0A_Square()
{
```

```
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM0[2:0] === 101
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B | (1<<3);
        // here we set COM0B[1:0] as 01 for toggling of OC0A
        // which is reflected in PD6
        // COM0A[1](bit7) from TCCR0A, COM0A[0](bit6) from TCCR0A
        TCCR0A = TCCR0A & ~(1<<7);
        TCCR0A = TCCR0A | (1<<6);
        // Next we set values for OCR0A and OCR0B
        // Since, TCNT0 goes till OCR0A, we can choose OCR0B to any value below OCR0A
        OCR0A = 0x70; // for freqeuncy
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS0[2:0] === 001
        // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
        TCCR0B = TCCR0B | (1<<0);
        TCCR0B = TCCR0B & ~(1<<1);
        TCCR0B = TCCR0B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer0_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
        // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
        uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
        uint32_t on_time_us = On_time_us >> 1;
        // MOde of operation to phase_corrected_phase_top_max Mode -- WGM0[2:0] === 101
        // WGM0[2](bit3) from TCCR0B, WGM0[1](bit1)  from TCCR0A, WGM0[0](bit0)  from TCCR0A
        TCCR0A = TCCR0A | (1<<0);
        TCCR0A = TCCR0A & ~(1<<1);
        TCCR0B = TCCR0B | (1<<3);
        // which is reflected in PD5
        // COM0B[1](bit5) from TCCR0A, COM0B[0](bit4) from TCCR0A
        TCCR0A = TCCR0A | (1<<5);
        TCCR0A = TCCR0A & ~(1<<4);
        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock
                OCR0A = 0;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B & ~(1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR0A = ((total_time * 16) >> 0) - 1;
                OCR0B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS0[2:0] === 001
                // CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
                TCCR0B = TCCR0B | (1<<0);
                TCCR0B = TCCR0B & ~(1<<1);
                TCCR0B = TCCR0B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
```

```
		OCR0A = ((total_time * 16) >> 3) - 1;
		OCR0B = ((on_time_us * 16) >> 3) - 1;
		// start timer by setting the clock prescalar
		//  dived by 8 from I/O clock
		//  CS0[2:0] === 010
		// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
		TCCR0B = TCCR0B & ~(1<<0);
		TCCR0B = TCCR0B | (1<<1);
		TCCR0B = TCCR0B & ~(1<<2);
	}
	else if((128 < total_time)  && (total_time <= 1024))
	{
		OCR0A = ((total_time * 16) >> 6) - 1;
		OCR0B = ((on_time_us * 16) >> 6) - 1;
		// start timer by setting the clock prescalar
		//  dived by 64 from I/O clock
		//  CS0[2:0] === 011
		// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
		TCCR0B = TCCR0B | (1<<0);
		TCCR0B = TCCR0B | (1<<1);
		TCCR0B = TCCR0B & ~(1<<2);
	}
	else if((1024 < total_time)  && (total_time <= 4096))
	{
		OCR0A = ((total_time * 16) >> 8) - 1;
		OCR0B = ((on_time_us * 16) >> 8) - 1;
		// start timer by setting the clock prescalar
		//  divide by256 from I/O clock
		//  CS0[2:0] === 100
		// CS0[2](bit2) from TCCR0B,CS0[1](bit1) from TCCR0B,CS0[0](bit0) from TCCR0B
		TCCR0B = TCCR0B & ~(1<<0);
		TCCR0B = TCCR0B & ~(1<<1);
		TCCR0B = TCCR0B | (1<<2);
	}
	else if(total_time > 4096)
	{
		// dont' cross more than 4.096ms
	}
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
	double total_time_us = (1000000.0/freqeuncy);
	double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
	if (on_time_us<1.0)
	{
		on_time_us = 1;		}

	// max time = 8ms -- min freqency = 125 Hz
	//  min time = 8us -- max frequency = 250000 = 125khz
	Timer0_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
	DDRD = DDRD | (1<<6) | (1<<5);
	// Timer0_NonInverting_TOP_at_MAX();
	// Timer0_Inverting_TOP_at_MAX();
    // Timer0_NonInverting_TOP_at_OCR0A();
    // Timer0_Inverting_TOP_at_OCR0A();
    Timer0_OC0A_Square();
    // PWMGeneration(12, 1000);
    while(1)
    {
    }
```

```
}
ISR(TIMER0_OVF_vect)
{
}
ISR(TIMER0_COMPA_vect)
{
}
ISR(TIMER0_COMPB_vect)
{
}
```

1.7.3 Output

Timer0_NonInverting_TOP_at_MAX

- The output can be seen @ *OC0A* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of 10% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC0B* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of 75% ($\frac{75}{100} * 0xFF = 0xC0$).

Timer0_Inverting_TOP_at_MAX

- The output can be seen @ *OC0A* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC0B* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$)and duty cycle of (100 - 75)% ($\frac{75}{100} * 0xFF = 0xC0$).

Timer0_NonInverting_TOP_at_OCR0A

- The output can be seen @ *OC0B* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$) and duty cycle of 85% ($\frac{85}{100} * 0x70 = 0x60$).

Timer0_Inverting_TOP_at_OCR0A

- The output can be seen @ *OC0B* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$) and duty cycle of (100 - 85)% ($\frac{85}{100} * 0x70 = 0x60$).

Timer0_OC0A_Square

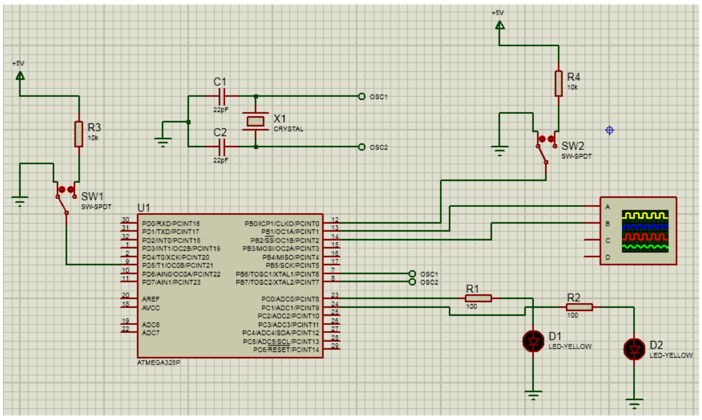
- The output can be seen @ *OC0A* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$).

PWMGeneration

- The output can be seen @ *OC0B*.

1.8 TimerCounter1_NormalMode

1.8.1 Circuit



1.8.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer1_asTimer()
{
        /* TCNT1 starts from 0X0000 goes upto 0XFFFF and restarts */
        /* No possible use case as it just goes upto 0xFFFF and restarts */
        // MOde of operation to Normal Mode -- WGM1[3:0] === 0000
        /* WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B,
         WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)  from TCCR1A*/
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* What to do when timer reaches the MAX(0xFFFF) value */
        // toggle OC1A on each time when reaches the MAX(0xFFFF)
        // which is reflected in PB1
        // Output OC1A to toglle when reaches MAX -- COM1A[1:0] === 01
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A & ~(1<<COM1A1);
        TCCR1A = TCCR1A | (1<<COM1A0);
        // toggle OC1B on each time when reaches the MAX(0xFFFF)
        // which is reflected in PB2
        // Output OC1B to toglle when reaches MAX -- COM1B[:0] === 01
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A & ~(1<<COM1B1);
        TCCR1A = TCCR1A | (1<<COM1B0);
        //Enable Interrupt of OVERFLOW flag so that interrupt can be generated
        TIMSK1 = TIMSK1 | (1<<TOV1);
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        // enabling global interrupt
        sei();
        // SO ON TIME = max_count / (F_CPU / PRESCALAR)
        // ON TIME = 0xFFFF / (16000000/1) = 4.096ms
        // since symmetric as toggling OFF TIME = 4.096ms
```

```
        // hence, we get a square wave of fequency 1 / 8.192ms = 122.07Hz

}
void Timer1_asCounter()
{
        // MOde of operation to Normal Mode -- WGM1[3:0] === 0000
        /* WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)
        from TCC1RA, WGM1[0](bit0)  from TCCR1A        */
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* to count external event -we must connect source to T1 (PD5) */
        // THE CLK IS CLOCKED FROM external source
        // Falling edge of T1(PD5) -- CS1[2:0] === 110
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<CS10);
        TCCR1B = TCCR1B | (1<<CS11);
        TCCR1B = TCCR1B | (1<<CS12);
}
void Timer1_asInputCapture()
{
        // MOde of operation to Normal Mode -- WGM1[3:0] === 0000
        /* WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)
         from TCC1RA, WGM1[0](bit0)  from TCCR1A        */
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        // Select the edge for Input Capture
        // ICES1(bit6) from TCCR1B
        // Capture on Rising edge, ICES1 === 1
        TCCR1B |= (1<<ICES1);
        //Enable Interrupt of Input Capture Interrupt Enable so that interrupt can be generated
        TIMSK1 = TIMSK1 | (1<<ICIE1);
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        // enabling global interrupt
        sei();
}

void Timer1_asDelay()
{
        /* TCNT1 starts from 0X0000 goes upto 0XFFFF and restarts */
        /* No possible use case as it just goes upto 0xFFFF and restarts */
        // MOde of operation to Normal Mode -- WGM1[3:0] === 0000
        /* WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B, WGM1[1](bit1)
        from TCC1RA, WGM1[0](bit0)  from TCCR1A        */
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* What to do when timer reaches the MAX(0xFFFF) value */
        // nothing should be done on OC1A for delay
        // nothing  -- COM1A[1:0] === 00
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A & ~(1<<COM1A1);
        TCCR1A = TCCR1A & ~(1<<COM1A0);
```

```c
        /* The delay possible = 0xffff / (F_CPU/prescalar) */
        // lowest delay = 0xffff / (16000000 / 1) = 4.096ms
        // when prescalar == 8 --> delay = 0xffff / (16000000 / 8) = 32.768ms
        // when prescalar == 64 --> delay = 0xffff / (16000000 / 64) = 262.144ms
        // when prescalar == 256 --> delay = 0xffff / (16000000 / 256) = 1.048576s
        // highest delay possible = 0xffff / (16000000 / 1024) = 4.194304s
        // start timer by setting the clock prescalar
        // divede by 64 from I/O clock
        // divede by 64-- CS1[2:0] === 101
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B | (1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        // actual delaying - wait until delay happens
        while((TIFR1 & 0x01) == 0x00); // checking overflow flag when overflow happns
        // clearing the overflag so that we can further utilize
        TIFR1 = TIFR1 | 0x01;
}
volatile uint16_t capVal=0;
int main(void)
{
    DDRB = DDRB | (1<<1) | (1<<2);
        DDRD = DDRD & ~(1<<5);
        DDRC |= (1<<0) | (1<<1);
        PORTC &= ~(1<<0);
        // Timer1_asTimer();
    // Timer1_asCounter();
        Timer1_asInputCapture();
    while(1)
    {
                // PORTC &= ~(1<<0);
                // Timer1_asDelay();
                // PORTC |= (1<<0);
                // Timer1_asDelay();
    }
}

ISR(TIMER1_OVF_vect)
{
    // toggle PC1 when overflows
        PINC |= (1<<1);
}

ISR(TIMER1_CAPT_vect)
{
        if((TIFR1 & (1<<ICF1)) != 0)
        {
                capVal = ICR1L;
                capVal = (ICR1H<<8) | (capVal & 0xFF);
                // see datamemory
        }
}
```

1.8.3 Output

Timer1_asTimer

- The output can be seen @ *OC1A* and *OC1B* pins with a on time of 4.096 ms and off time of 4.096 ms ($\frac{0xFFFF*1}{16000000} = 4.096ms$).
- Also, *PC1* toggles for the overflow Timer1.

Timer1_asCounter

- The output can be seen @ Watch Window and see the **TCNT1** register when pulsed @ *T1* pin.

Timer1_asDelay

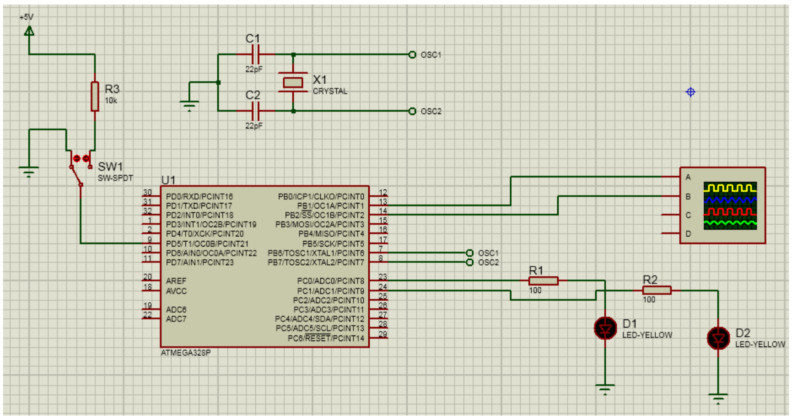
- The output can be seen *PC0* pin.

Timer1_asInputCapture

- The output can be seen @ Watch Window and see the **ICR1** register when pulsed @ *ICP1* pin.

1.9 TimerCounter1_CTC

1.9.1 Circuit



1.9.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer1_asTimer()
{
        /* TCNT1 starts from 0X0000 goes upto OCR1A or ICR1 and restarts
         MOde of operation to Normal Mode -- WGM1[3:0] ===
        0100(TOP = OCR1A) or 1100(TOP = ICR1)
         WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B,
        WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)  from TCCR1A*/
        // we take TOP to be OCR1A for custom frequency
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* What to do when timer reaches the OCR1A value */
        // toggle OC1A on each time when reaches the OCR1A
        // which is reflected in PB1
        // Output OC1A to toglle when reaches OCR1A -- COM1A[1:0] === 01
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1A0);
        TCCR1A = TCCR1A & ~(1<<COM1A1);
        // toggle OC1B on each time when reaches the OCR1A
        // which is reflected in PB2
        // Output OC1B to toglle when reaches OCR1A -- COM1B[1:0] === 01
        // COM1B[1](bi57) from TCCR1A, COM1B[0](bit64) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B0);
        TCCR1A = TCCR1A & ~(1<<COM1B1);
        // Enable Interrupt when counter matches OCR1A Rgister
        //  OCIE1A  bit is enabled
```

```
        TIMSK1 = TIMSK1 | (1<<OCIE1A);
        // setting the value till the counter should reach in OCR1A
        // for toggling of OC1A pin
        OCR1A = 0x4861;
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        // enabling global interrupt
        sei();
        // SO ON TIME = (1 + OCR1A) / (F_CPU / PRESCALAR)
        // ON TIME = 0x4861 / (16000000/1) = 1.15ms
        // since symmetric as toggling OFF TIME = 1.15ms
        // hence, we get a square wave of fequency 1 / 2.31ms = 431Hz
}
void Timer1_asCounter()
{
        /* MOde of operation to Normal Mode --
        WGM1[3:0] === 0100(TOP = OCR1A) or 1100(TOP = ICR1)
         WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from
        TCCR1B, WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)  from TCCR1A        */
        TCCR1A = TCCR1A & ~(1<<WGM10);
        TCCR1A = TCCR1A & ~(1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* What to do when timer reaches the OCR1A value */
        // toggle OC1A on each time when reaches the OCR1A
        // which is reflected in PB1
        // Output OC1A to toglle when reaches OCR1A -- COM1A[1:0] === 01
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1A0);
        TCCR1A = TCCR1A & ~(1<<COM1A1);
        //we count till OCR1A register value and toggle
        // lets' count 10 pulses
        OCR1A = 0x000a;
        // Enable Interrupt when counter matches OCR1A Rgister
        //  OCIE1A  bit is enabled
        TIMSK1 = TIMSK1 | (1<<OCIE1A);
        /* to count external event -we must connect source to T1 (PD5) */
        // THE CLK IS CLOCKED FROM external source
        // Falling edge of T1(PD5) -- CS1[2:0] === 110
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<CS10);
        TCCR1B = TCCR1B | (1<<CS11);
        TCCR1B = TCCR1B | (1<<CS12);
        // since for every rising edge the count increase
        // so to reach 10 count, it would take 0xa / (frequency of input at T1 pin or PD5)
        /* we wave used 5kHz so it would take ==>
         2ms to toggle as we have made OC1A toggle when overflows (by setting COMA[1:0])*/
        // also we canuse TCNT1 as edge counter
        // enabling global interrupt
        sei();
}
void Timer1_asDelayIn_us(uint32_t delay_in_us)
{
        /* minimum delay being 4us -- choose like that - because, of the the delay for execution,
        - we get us if we use toggling of pins OC1A or OC1B */
        // use PRESCALAR OF 1 -- 4us - 4.096ms -- usage 4us - 4ms -- factor=0 -- CS1[2:0]=1
        // use PRESCALAR OF 8 -- 4us - 32.768ms -- usage 5ms - 32ms -- factor=3 -- CS1[2:0]=2
        // use PRESCALAR OF 64 -- 4us - 262.144ms -- usage 33ms - 260ms -- factor=6 -- CS0[2:0]=3
```

```c
/* use PRESCALAR OF 256 -- 16us - 1.048s -- usage 261ms
 - 1.048s -- factor=8 -- CS0[2:0]=4*/

/* TCNT1 starts from 0X0000 goes upto OCR1A or ICR1 and restarts */
// MOde of operation to Normal Mode -- WGM1[3:0] === 0100(TOP = OCR1A) or 1100(TOP = ICR1)
/* WGM1[3](bit4) from TCCR1B, WGM1[2](bit3) from TCCR1B,
 WGM1[1](bit1)  from TCC1RA, WGM1[0](bit0)  from TCCR1A        */
// we take TOP to be OCR1A for custom frequency
TCCR1A = TCCR1A & ~(1<<WGM10);
TCCR1A = TCCR1A & ~(1<<WGM11);
TCCR1B = TCCR1B | (1<<WGM12);
TCCR1B = TCCR1B & ~(1<<WGM13);
/* What to do when timer reaches the MAX(0xFFFF) value */
// nothing should be done on OC1A for delay
// nothing  -- COM1A[1:0] === 00
// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
TCCR1A = TCCR1A & ~(1<<COM1A1);
TCCR1A = TCCR1A & ~(1<<COM1A0);
if(delay_in_us <=3)
{
        // if delay_in_us <= 3us -- so we stop clock

        OCR1A = 0;
        // stop clcok
        // stop clcok-- CS1[2:0] === 000
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((3 < delay_in_us)  && (delay_in_us <= 4000))
{
        OCR1A = ((delay_in_us * 16) >> 0) - 1;
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((4000 < delay_in_us)  && (delay_in_us <= 32000))
{
        OCR1A = ((delay_in_us * 16) >> 3) - 1;
        // start timer by setting the clock prescalar
        // divide by 8 from I/O clock
        // divide by 8 CS1[2:0] === 010
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<CS10);
        TCCR1B = TCCR1B | (1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
}
else if((32000 < delay_in_us)  && (delay_in_us <= 260000))
{
        OCR1A = ((delay_in_us * 16) >> 6) - 1;
        // start timer by setting the clock prescalar
        // divide by 64 from I/O clock
        // divide by 64 CS1[2:0] === 011
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B | (1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
}
```

```
        else if((260000 < delay_in_us)  && (delay_in_us <= 1000000))
        {
                OCR1A = ((delay_in_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                // divide by 256 from I/O clock
                // divide by 256 CS1[2:0] === 100
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<CS10);
                TCCR1B = TCCR1B & ~(1<<CS11);
                TCCR1B = TCCR1B | (1<<CS12);
        }
        else if(delay_in_us > 1000000)
        {
                Timer1_asDelayIn_us(delay_in_us - 1000000);
                OCR1A = ((1000000 * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                // divide by 256 from I/O clock
                //divide by 256 CS1[2:0] === 100
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<CS10);
                TCCR1B = TCCR1B & ~(1<<CS11);
                TCCR1B = TCCR1B | (1<<CS12);
        }
        // actual delaying - wait until delay happens
        while((TIFR1 & 0x02) == 0x00);
            // checking OCF1A (compare match flag A) flag when match happns
        // clearing the compare match flag so that we can further utilize
        TIFR1 = TIFR1 | 0x02;

}
int main(void)
{
    DDRB = DDRB | (1<<1) | (1<<2);
        DDRD = DDRD & ~(1<<5);
        DDRC |= (1<<0) | (1<<1);
        PORTC &= ~(1<<0);
        // Timer1_asTimer();
    // Timer1_asCounter();
    while(1)
    {
                PINC |= (1<<0);
                Timer1_asDelayIn_us(400);
    }
}

ISR(TIMER1_COMPA_vect)
{
    // toggle PC1 when TCNT1 matches OCR1A
        PINC |= (1<<1);
}
```

1.9.3 Output

Timer1_asTimer

- The output can be seen @ *OC1A* and *OC1B* pins with a on time of 1.15ms and off time of 1.15 ms ($\frac{(0x4861+1)*1}{16000000} = 1.15ms$).
- Also, *PC1* toglles for the **TCNT1** matches **OCR1A**.

Timer1_asCounter

- The output can be seen @ Watch Window and see the **TCNT1** register when pulsed @ *T1* pin.

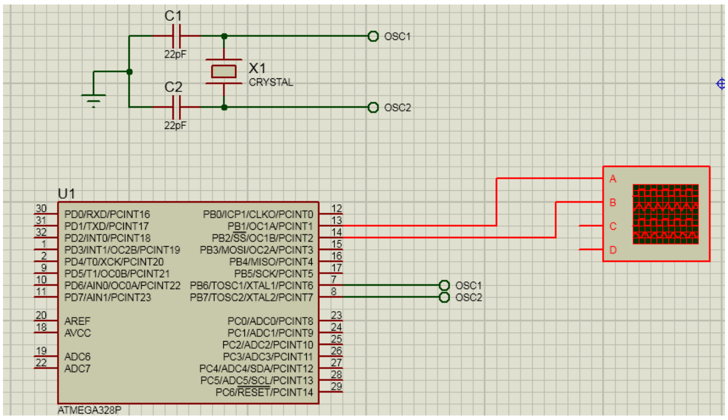
- Also, the *PC1* pin toggles for every 10 changes at *T1* pin.

Timer1_asDelayIn_us

- The output can be seen *PC0* pin.

1.10 TimerCounter1_FastPWM

1.10.1 Circuit



1.10.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer1_NonInverting_TOP_at_MAX()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
                WGM1[3:0] --> 0101 --        TOP--> 0X00FF
                WGM1[3:0] --> 0110 --        TOP--> 0x01FF
                WGM1[3:0] --> 0111 --        TOP--> 0x03FF
                WGM1[3:0] --> 1110 --        TOP--> ICR1
                WGM1[3:0] --> 1111 --        TOP--> OCR1A
        */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 0111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
    // here we set COM0A[1:0] as 10 for non-inverting
        // here we set COM0B[1:0] as 10 for non-inverting
        // which is reflected in PD6
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1A1);
        TCCR1A = TCCR1A & ~(1<<COM1A0);
        // which is reflected in PD65
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B1);
        TCCR1A = TCCR1A & ~(1<<COM1B0);
    // Enable Interrupt when TOV1 overflows TOP - here 0x03FF
        //  TOIE1 bit is enabled
        TIMSK1 = TIMSK1 | (1<<TOIE1);
        /* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
        TIMSK1 = TIMSK1 | (1<<OCIE1A);
        /* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
        TIMSK1 = TIMSK1 | (1<<OCIE1B);
```

```
        // Next we set values for OCR1A and OCR2B
        /* Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and
        OCR1B to any value below max(0x03FF)*/
        OCR1A = 102; // for 10% duty clcle
        OCR1B = 767; // for 75% duty clcle
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
        sei();
}
void Timer1_Inverting_TOP_at_MAX()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
        WGM1[3:0] --> 0101 --          TOP--> 0X00FF
        WGM1[3:0] --> 0110 --          TOP--> 0x01FF
        WGM1[3:0] --> 0111 --          TOP--> 0x03FF
        WGM1[3:0] --> 1110 --          TOP--> ICR1
        WGM1[3:0] --> 1111 --          TOP--> OCR1A          */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 0111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        // here we set COM0A[1:0] as 11 for inverting
        // here we set COM0B[1:0] as 11 for inverting
        // which is reflected in PD6
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1A1);
        TCCR1A = TCCR1A | (1<<COM1A0);
        // which is reflected in PD65
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B1);
        TCCR1A = TCCR1A | (1<<COM1B0);
        // Enable Interrupt when TOV1 overflows TOP - here 0x03FF
        //  TOIE1 bit is enabled
        TIMSK1 = TIMSK1 | (1<<TOIE1);
        /* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
        TIMSK1 = TIMSK1 | (1<<OCIE1A);
        /* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
        TIMSK1 = TIMSK1 | (1<<OCIE1B);
        // Next we set values for OCR1A and OCR2B
        /* Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B to
         any value below max(0x03FF)*/
        OCR1A = 102; // for 10% duty clcle
        OCR1B = 767; // for 75% duty clcle
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
        sei();
}
void Timer1_NonInverting_TOP_at_OCR1A()
```

```
{
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
                WGM1[3:0] --> 0101 --         TOP--> 0X00FF
                WGM1[3:0] --> 0110 --         TOP--> 0x01FF
                WGM1[3:0] --> 0111 --         TOP--> 0x03FF
                WGM1[3:0] --> 1110 --         TOP--> ICR1
                WGM1[3:0] --> 1111 --         TOP--> OCR1A         */
        // we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
        // for non-inverting on  OC1B we use 10 for and COM1B[1:0]
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A & ~(1<<COM1B0);
        TCCR1A = TCCR1A | (1<<COM1B1);
        // Next we set values for OCR1A and OCR1B
        // Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
        OCR1A = 0x7869; // for freqeuncy
        OCR1B = 0x1A20; // for pwm duty cylc
        // Enable interrupt when count reaches the overflow value
        TIMSK1 |= (1<<TOV1);
        // Enable interrupt when count reaches the OCR1B
        TIMSK1 |= (1<<OCF1B);
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //e enabel globalinterrupt
        sei();
}
void Timer1_Inverting_TOP_at_OCR1A()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
        WGM1[3:0] --> 0101 --         TOP--> 0X00FF
        WGM1[3:0] --> 0110 --         TOP--> 0x01FF
        WGM1[3:0] --> 0111 --         TOP--> 0x03FF
        WGM1[3:0] --> 1110 --         TOP--> ICR1
        WGM1[3:0] --> 1111 --         TOP--> OCR1A         */
        // we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
        // for ninverting on  OC1B we use 11 for and COM1B[1:0]
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B0);
        TCCR1A = TCCR1A | (1<<COM1B1);
        // Next we set values for OCR1A and OCR1B
        // Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
        OCR1A = 0x7869; // for freqeuncy
        OCR1B = 0x1A20; // for pwm duty cylc
        // Enable interrupt when count reaches the overflow value
        TIMSK1 |= (1<<TOV1);
        // Enable interrupt when count reaches the OCR1B
        TIMSK1 |= (1<<OCF1B);
```

```c
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //e enabel globalinterrupt
        sei();
}
void Timer1_OC1A_Square()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
        WGM1[3:0] --> 0101 --        TOP--> 0X00FF
        WGM1[3:0] --> 0110 --        TOP--> 0x01FF
        WGM1[3:0] --> 0111 --        TOP--> 0x03FF
        WGM1[3:0] --> 1110 --        TOP--> ICR1
        WGM1[3:0] --> 1111 --        TOP--> OCR1A        */
        // we take OCR1A for custom frequency
        // choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
    // here we set COM1B[1:0] as 01 for toggling of OC1A
    // which is reflected in PB1
    // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
    TCCR1A = TCCR1A & ~(1<<5);
    TCCR1A = TCCR1A | (1<<4);
    OCR1A = 0x7869; // for freqeuncy
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
    sei();
}
void Timer1_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
        uint32_t total_time = on_time_us + off_time_us;
        /* TCNT1 starts from 0X0000 goes upto TOP and restarts from 0X00*/
        /* Mode of operation:
                WGM1[3:0] --> 0101 --        TOP--> 0X00FF
                WGM1[3:0] --> 0110 --        TOP--> 0x01FF
                WGM1[3:0] --> 0111 --        TOP--> 0x03FF
                WGM1[3:0] --> 1110 --        TOP--> ICR1
                WGM1[3:0] --> 1111 --        TOP--> OCR1A
        */
        // we take OCR1A for custom frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1111 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B | (1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B0);
        TCCR1A = TCCR1A | (1<<COM1B1);
        if(total_time <4)
        {
```

```
        // if total_time <= 3us -- so we stop clock
        OCR1A = 0;
        OCR1B = 0;
        // start timer by setting the clock prescalar
        //  use the same clock from I/O clock
        //  CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
}
else if((3 < total_time)  && (total_time <= 4000))
{
        OCR1A = ((total_time * 16) >> 0) - 1;
        OCR1B = ((on_time_us * 16) >> 0) - 1;
        // start timer by setting the clock prescalar
        //  use the same clock from I/O clock
        //  CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
}
else if((4000 < total_time)  && (total_time <= 32000))
{
        OCR1A = ((total_time * 16) >> 3) - 1;
        OCR1B = ((on_time_us * 16) >> 3) - 1;
        // start timer by setting the clock prescalar
        //  dived by 8 from I/O clock
        //  CS1[2:0] === 010
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B | (1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
}
else if((32000 < total_time)  && (total_time <= 260000))
{
        OCR1A = ((total_time * 16) >> 6) - 1;
        OCR1B = ((on_time_us * 16) >> 6) - 1;
        // start timer by setting the clock prescalar
        //  dived by 64 from I/O clock
        //  CS1[2:0] === 011
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<0);
        TCCR1B = TCCR1B | (1<<1);
        TCCR1B = TCCR1B & ~(1<<2);
}
else if((260000 < total_time)  && (total_time <= 1000000))
{
        OCR1A = ((total_time * 16) >> 8) - 1;
        OCR1B = ((on_time_us * 16) >> 8) - 1;
        // start timer by setting the clock prescalar
        //  divide by256 from I/O clock
        //  CS1[2:0] === 100
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B & ~(1<<0);
        TCCR1B = TCCR1B & ~(1<<1);
        TCCR1B = TCCR1B | (1<<2);
}
else if(total_time > 1000000)
{
        // dont' cross more than 1s
}
```

```c
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }
        // max time = 1S -- min freqency = 1 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer1_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
        DDRB = DDRB | (1<<1) | (1<<2);
        // Timer1_NonInverting_TOP_at_MAX();
        // Timer1_Inverting_TOP_at_MAX();
    Timer1_NonInverting_TOP_at_OCR1A();
    // Timer1_Inverting_TOP_at_OCR1A();
    // Timer1_OC1A_Square();
    // PWMGeneration(12, 1000);
    while(1)
    {
    }
}
ISR(TIMER1_OVF_vect)
{
}
ISR(TIMER1_COMPA_vect)
{
}
ISR(TIMER1_COMPB_vect)
{
}
```

1.10.3 Output

Timer1_NonInverting_TOP_at_MAX

- The output can be seen @ *OC1A* with a frequency of 15.640 kHz($\frac{0x03FF*1}{16000000} = 64ms$) and duty cycle of 10% ($\frac{10}{100} * 0x3FF = 0x66$).
- The output can be seen @ *OC1B* with a frequency of 15.640 kHz($\frac{0x03FF*1}{16000000} = 64ms$) and duty cycle of 75% ($\frac{75}{100} * 0x3FF = 0x2FF$).

Timer1_Inverting_TOP_at_MAX

- The output can be seen @ *OC1A* with a frequency of 15.640 kHz($\frac{0x03FF*1}{16000000} = 64ms$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0x3FF = 0x66$).
- The output can be seen @ *OC1B* with a frequency of 15.640 kHz($\frac{0x03FF*1}{16000000} = 64ms$) and duty cycle of (100 - 75)% ($\frac{75}{100} * 0x3FF = 0x2FF$).

Timer1_NonInverting_TOP_at_OCR1A

- The output can be seen @ *OC1B* with a frequency of 0.5208 kHz($\frac{0x7869*1}{16000000} = 1.92ms$) and duty cycle of 21% ($\frac{21}{100} = 0x1A20$).

Timer1_Inverting_TOP_at_OCR1A

- The output can be seen @ *OC1B* with a frequency of 0.5208 kHz($\frac{0x7869*1}{16000000} = 1.92ms$) and duty cycle of (100 - 21)% ($\frac{21}{100} = 0x1A20$).

Timer1_OC1A_Square

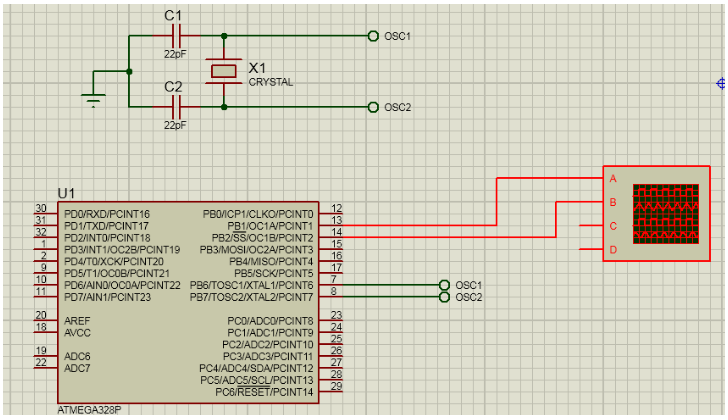
- The output can be seen @ *OC1A* with a frequency of 0.5208 kHz($\frac{0x70*1}{16000000} = 1.92ms$).

PWMGeneration

- The output can be seen @ *OC1B*.

1.11 TimerCounter1_PhaseCorrectedPWM

1.11.1 Circuit



1.11.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer1_NonInverting_TOP_at_MAX()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
        /* Mode of operation:
                WGM1[3:0] --> 0001 --        TOP--> 0X00FF
                WGM1[3:0] --> 0010 --        TOP--> 0x01FF
                WGM1[3:0] --> 0011 --        TOP--> 0x03FF
                WGM1[3:0] --> 1010 --        TOP--> ICR1
                WGM1[3:0] --> 1011 --        TOP--> OCR1A
        */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 0011 for 0x03FF as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B & ~(1<<WGM13);
        /* in timer0_phase_pwm_top_max, only two possiblites are there for
        COM0B[1:0] and COM0A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
        // here we set COM0A[1:0] as 10 for non-inverting
        // here we set COM0B[1:0] as 10 for non-inverting
        // which is reflected in PD6
        // COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1A1);
        TCCR1A = TCCR1A & ~(1<<COM1A0);
        // which is reflected in PD65
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B1);
        TCCR1A = TCCR1A & ~(1<<COM1B0);
    // Enable Interrupt when TOV1 overflows TOP - here 0x03FF
        //  TOIE1 bit is enabled
        TIMSK1 = TIMSK1 | (1<<TOIE1);
```

```
	/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
	TIMSK1 = TIMSK1 | (1<<OCIE1A);
	/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
	TIMSK1 = TIMSK1 | (1<<OCIE1B);
	// Next we set values for OCR1A and OCR2B
	/* Since, TCNT1 goes till max(0x3FF), we can choose OCR1A and OCR1B
	to any value below max(0x03FF)*/
	OCR1A = 102; // for 10% duty clcle
	OCR1B = 767; // for 75% duty clcle
	// start timer by setting the clock prescalar
	// SAME AS from I/O clock
	// same-- CS1[2:0] === 001
	// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
	TCCR1B = TCCR1B | (1<<CS10);
	TCCR1B = TCCR1B & ~(1<<CS11);
	TCCR1B = TCCR1B & ~(1<<CS12);
	//enabled global interrupt
	sei();
}
void Timer1_Inverting_TOP_at_MAX()
{
	/* TCNT1 starts from 0X0000 goes upto TOP and  from TOP to BOTTOM*/
	/* Mode of operation:
		WGM1[3:0] --> 0001 --       TOP--> 0X00FF
		WGM1[3:0] --> 0010 --       TOP--> 0x01FF
		WGM1[3:0] --> 0011 --       TOP--> 0x03FF
		WGM1[3:0] --> 1010 --       TOP--> ICR1
		WGM1[3:0] --> 1011 --       TOP--> OCR1A        */
	// we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
	// choose WGM1[3:0] --> 0011 for 0x03FFF as TOP for custom frequency
	TCCR1A = TCCR1A | (1<<WGM10);
	TCCR1A = TCCR1A | (1<<WGM11);
	TCCR1B = TCCR1B & ~(1<<WGM12);
	TCCR1B = TCCR1B & ~(1<<WGM13);
	/* in timer0_phase_pwm_top_max, only two possiblites are
	there for COM0B[1:0] and COM0A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
	// here we set COM0A[1:0] as 11 for inverting
	// here we set COM0B[1:0] as 11 for inverting
	// which is reflected in PD6
	// COM1A[1](bit7) from TCCR1A, COM1A[0](bit6) from TCCR1A
	TCCR1A = TCCR1A | (1<<COM1A1);
	TCCR1A = TCCR1A | (1<<COM1A0);
	// which is reflected in PD65
	// COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
	TCCR1A = TCCR1A | (1<<COM1B1);
	TCCR1A = TCCR1A | (1<<COM1B0);
    // Enable Interrupt when TOV1 overflows TOP - here 0x03FF
	//  TOIE1 bit is enabled
	TIMSK1 = TIMSK1 | (1<<TOIE1);
	/* we use OCF1A flag - which is set at every time TCN0 reaches OCR1A */
	TIMSK1 = TIMSK1 | (1<<OCIE1A);
	/* we use OCF1B flag - which is set at every time TCN0 reaches OCR1B */
	TIMSK1 = TIMSK1 | (1<<OCIE1B);
	// Next we set values for OCR1A and OCR1B
	/* Since, TCNT1 goes till max(0x3FF), we can choose OCR1A
	and OCR1B to any value below max(0x03FF) */
	OCR1A = 102; // for 10% duty clcle
	OCR1B = 767; // for 75% duty clcle
	// start timer by setting the clock prescalar
	// SAME AS from I/O clock
	// same-- CS1[2:0] === 001
	// CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
	TCCR1B = TCCR1B | (1<<CS10);
```

```
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
        sei();
}
void Timer1_NonInverting_TOP_at_OCR1A(){
        /* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
        /* Mode of operation:
                WGM1[3:0] --> 0001 --
                TOP--> 0X00FF
                WGM1[3:0] --> 0010 --
                TOP--> 0x01FF
                WGM1[3:0] --> 0011 --
                TOP--> 0x03FF
                WGM1[3:0] --> 1010 --
                TOP--> ICR1
                WGM1[3:0] --> 1011 --
                TOP--> OCR1A        */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
        // here we set COM1A[1:0] as 10 for non-inverting
        // which is reflected in PD5
        // COM1B[1](bit5) from TCCR1A, COMOB[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<5);
        TCCR1A = TCCR1A & ~(1<<4);
        // Next we set values for OCR1A and OCR1B
        // Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
        OCR1A = 0x7869; // for freqeuncy
        OCR1B = 0x1A20; // for pwm duty cylc
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
        sei();
}
void Timer1_Inverting_TOP_at_OCR1A()
{
        /* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
        /* Mode of operation:
                WGM1[3:0] --> 0001 --
                TOP--> 0X00FF
                WGM1[3:0] --> 0010 --
                TOP--> 0x01FF
                WGM1[3:0] --> 0011 --
                TOP--> 0x03FF
                WGM1[3:0] --> 1010 --
                TOP--> ICR1
                WGM1[3:0] --> 1011 --
                TOP--> OCR1A        */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
```

```
        // here we set COM1A[1:0] as 11 for inverting
        // which is reflected in PD5
        // COM1B[1](bit5) from TCCR1A, COMOB[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<5);
        TCCR1A = TCCR1A | (1<<4);
        // Next we set values for OCR1A and OCR1B
        // Since, TCNT1 goes till OCR1A, we can choose OCR1B to any value below OCR1A
        OCR1A = 0x7869; // for freqeuncy
        OCR1B = 0x1A20; // for pwm duty cylc
        // start timer by setting the clock prescalar
        // SAME AS from I/O clock
        // same-- CS1[2:0] === 001
        // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
        TCCR1B = TCCR1B | (1<<CS10);
        TCCR1B = TCCR1B & ~(1<<CS11);
        TCCR1B = TCCR1B & ~(1<<CS12);
        //enabled global interrupt
        sei();
}
void Timer1_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
        // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
        uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
        uint32_t on_time_us = On_time_us >> 1;
        /* TCNT1 starts from 0X0000 goes upto TOP and from TOP to BOTTOM*/
        /* Mode of operation:
                WGM1[3:0] --> 0001 --
                TOP--> 0X00FF
                WGM1[3:0] --> 0010 --
                TOP--> 0x01FF
                WGM1[3:0] --> 0011 --
                TOP--> 0x03FF
                WGM1[3:0] --> 1010 --
                TOP--> ICR1
                WGM1[3:0] --> 1011 --
                TOP--> OCR1A         */
        // we take 0x03FF for fixed frequency and OCR1B for PWM on time(duty cycle)
        // choose WGM1[3:0] --> 1011 for OCR1A as TOP for custom frequency
        TCCR1A = TCCR1A | (1<<WGM10);
        TCCR1A = TCCR1A | (1<<WGM11);
        TCCR1B = TCCR1B & ~(1<<WGM12);
        TCCR1B = TCCR1B | (1<<WGM13);
        // COM1B[1](bit5) from TCCR1A, COM1B[0](bit4) from TCCR1A
        TCCR1A = TCCR1A | (1<<COM1B0);
        TCCR1A = TCCR1A | (1<<COM1B1);
        if(total_time <4)
        {
                // if total_time <= 3us -- so we stop clock
                OCR1A = 0;
                OCR1B = 0;
                // start timer by setting the clock prescalar
                // use the same clock from I/O clock
                // CS1[2:0] === 001
                // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
                TCCR1B = TCCR1B & ~(1<<0);
                TCCR1B = TCCR1B & ~(1<<1);
                TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((3 < total_time) && (total_time <= 4000))
        {
                OCR1A = ((total_time * 16) >> 0) - 1;
                OCR1B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
```

```
            // use the same clock from I/O clock
            // CS1[2:0] === 001
            // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
            TCCR1B = TCCR1B | (1<<0);
            TCCR1B = TCCR1B & ~(1<<1);
            TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((4000 < total_time) && (total_time <= 32000))
        {
            OCR1A = ((total_time * 16) >> 3) - 1;
            OCR1B = ((on_time_us * 16) >> 3) - 1;
            // start timer by setting the clock prescalar
            // dived by 8 from I/O clock
            // CS1[2:0] === 010
            // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
            TCCR1B = TCCR1B & ~(1<<0);
            TCCR1B = TCCR1B | (1<<1);
            TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((32000 < total_time) && (total_time <= 260000))
        {
            OCR1A = ((total_time * 16) >> 6) - 1;
            OCR1B = ((on_time_us * 16) >> 6) - 1;
            // start timer by setting the clock prescalar
            // dived by 64 from I/O clock
            // CS1[2:0] === 011
            // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
            TCCR1B = TCCR1B | (1<<0);
            TCCR1B = TCCR1B | (1<<1);
            TCCR1B = TCCR1B & ~(1<<2);
        }
        else if((260000 < total_time) && (total_time <= 1000000))
        {
            OCR1A = ((total_time * 16) >> 8) - 1;
            OCR1B = ((on_time_us * 16) >> 8) - 1;
            // start timer by setting the clock prescalar
            // divide by256 from I/O clock
            // CS1[2:0] === 100
            // CS1[2](bit2) from TCCR1B,CS1[1](bit1) from TCCR1B,CS1[0](bit0) from TCCR1B
            TCCR1B = TCCR1B & ~(1<<0);
            TCCR1B = TCCR1B & ~(1<<1);
            TCCR1B = TCCR1B | (1<<2);
        }
        else if(total_time > 1000000)
        {
            // dont' cross more than 1s
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
            on_time_us = 1;
        }
        // max time = 8ms -- min freqency = 125 Hz
        //  min time = 8us -- max frequency = 250000 = 125khz
        Timer1_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
        DDRB = DDRB | (1<<1) | (1<<2);
```

```
        // Timer1_NonInverting_TOP_at_MAX();
        // Timer1_Inverting_TOP_at_MAX();
    Timer1_NonInverting_TOP_at_OCR1A();
    // Timer1_Inverting_TOP_at_OCR1A();
    // PWMGeneration(12, 1000);
    while(1)
    {
    }
}
ISR(TIMER1_OVF_vect)
{
}
ISR(TIMER1_COMPA_vect)
{
}
ISR(TIMER1_COMPB_vect)
{
}
```

1.11.3 Output

Timer1_NonInverting_TOP_at_MAX

- The output can be seen @ *OC1A* with a frequency of 7.812 kHz($\frac{(2*0x03FF)*1}{16000000} = 128ms$) and duty cycle of 10% ($\frac{10}{100} * 0x3FF = 0x66$).
- The output can be seen @ *OC1B* with a frequency of 7.812 kHz($\frac{(2*0x03FF)*1}{16000000} = 128ms$) and duty cycle of 75% ($\frac{75}{100} * 0x3FF = 0x2FF$).

Timer0_Inverting_TOP_at_MAX

- The output can be seen @ *OC1A* with a frequency of 7.812 kHz($\frac{(2*0x03FF)*1}{16000000} = 128ms$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0x3FF = 0x66$).
- The output can be seen @ *OC1B* with a frequency of 7.812 kHz($\frac{(2*0x03FF)*1}{16000000} = 128ms$) and duty cycle of (100 - 75)% ($\frac{75}{100} * 0x3FF = 0x2FF$).

Timer1_NonInverting_TOP_at_OCR1A

- The output can be seen @ *OC1B* with a frequency of 0.2604 kHz($\frac{(2*0x7869)*1}{16000000} = 3.84ms$) and duty cycle of 21% ($\frac{21}{100} = 0x1A20$).

Timer1_Inverting_TOP_at_OCR1A

- The output can be seen @ *OC1B* with a frequency of 0.2604 kHz($\frac{(2*0x7869)*1}{16000000} = 3.84ms$) and duty cycle of (100 - 21)% ($\frac{21}{100} = 0x1A20$).

Timer1_OC1A_Square

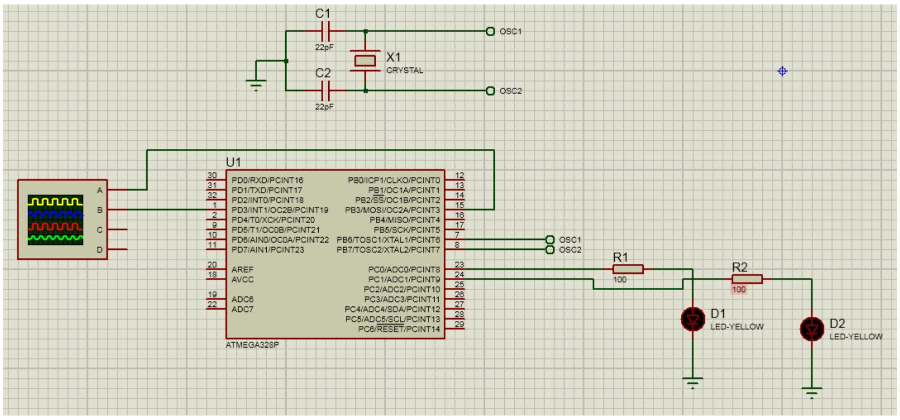
- The output can be seen @ *OC1A* with a frequency of 0.2604 kHz($\frac{(2*0x7869)*1}{16000000} = 3.84ms$).

PWMGeneration

- The output can be seen @ *OC1B*.

1.12 TimerCounter2_NormalMode

1.12.1 Circuit



1.12.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer2_asTimer()
{
        // MOde of operation to Normal Mode -- WGM2[2:0] === 000
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A & (~(1<<0) & ~(1<<1));
        TCCR2B = TCCR2B & ~(1<<3);
        /* What to do when timer reaches the MAX(0xFF) value */
        // toggle OC2A and OC2B on each time when reaches the MAX(0xFF)
        // which is reflected in PB3 and PD3
        // Output OC2A to togle when reaches MAX -- COM2A[1:0] === 01
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // Output OC2B to togle when reaches MAX -- COM2B1:0] === 01
        // COM2B[1](bit7) from TCCR2A, COM2B[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<5);
        TCCR2A = TCCR2A | (1<<4);
        //Enable Interrupt of OVERFLOW flag so that interrupt can be generated
        TIMSK2 = TIMSK2 | (1<<0);
        // start timer by setting the clock prescalar
        // DIVIDE BY 8 from I/O clock
        // DIVIDE BY 8-- CS2[2:0] === 010
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<1);
        TCCR2B = TCCR2B & (~(1<<0) & ~(1<<2));
        // enabling global interrupt
        sei();
        // SO ON TIME = max_count / (F_CPU / PRESCALAR)
        // ON TIME = 0xFF / (16000000/8) = 128us
        // since symmetric as toggling OFF TIME = 128us
        // hence, we get a square wave of fequency 1 / 256us = 3.906kHz
}

void Timer2_asDelay()
{
        /* TCNT2 starts from 0X00 goes upto 0XFF and restarts */
        /* No possible use case as it just goes upto 0xFF and restarts */
        // MOde of operation to Normal Mode -- WGM2[2:0] === 000
```

```
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A & (~(1<<0) & ~(1<<1));
        TCCR2B = TCCR2B & ~(1<<3);
        /* What to do when timer reaches the MAX(0xFF) value */
        // nothing should be done on OC2A for delay
        // nothing  -- COM2A[1:0] === 00
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<7);
        TCCR2A = TCCR2A & ~(1<<6);
        /* The delay possible = 0xff / (F_CPU/prescalar) */
        // lowest delay = 0xff / (16000000 / 1) = 16us
        // when prescalar == 8 --> delay = 0xff / (16000000 / 8) = 128us
        // when prescalar == 64 --> delay = 0xff / (16000000 / 64) = 1.024ms
        // when prescalar == 256 --> delay = 0xff / (16000000 / 256) = 4.096ms
        // highest delay possible = 0xff / (16000000 / 1024) = 16.38ms
        // start timer by setting the clock prescalar
        // DIVIDE BY 8 use the same clock from I/O clock
        // DIVIDE BY 8-- CS2[2:0] === 010
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B & ~(1<<0);
        TCCR2B = TCCR2B | (1<<1);
        TCCR2B = TCCR2B & ~(1<<2);

        // actual delaying - wait until delay happens
        while((TIFR2 & 0x01) == 0x00); // checking overflow flag when overflow happns
        // clearing the overflag so that we can further utilize
        TIFR2 = TIFR2 | 0x01;
}
int main(void)
{
    // making the PB2 and PD3 as output
    DDRD = DDRD | (1<<3);
    DDRB = DDRB | (1<<3);
        DDRC |= (1<<0) | (1<<1);
        PORTC &= ~(1<<0);
        //Timer2_asTimer();
    while(1)
    {
                PORTC &= ~(1<<0);
                Timer2_asDelay();
                PORTC |= (1<<0);
                Timer2_asDelay();
    }
}

ISR(TIMER2_OVF_vect)
{
    // do the thing when overflows.
                PINC |= (1<<1);
}
```

1.12.3 Output

Timer2_asTimer

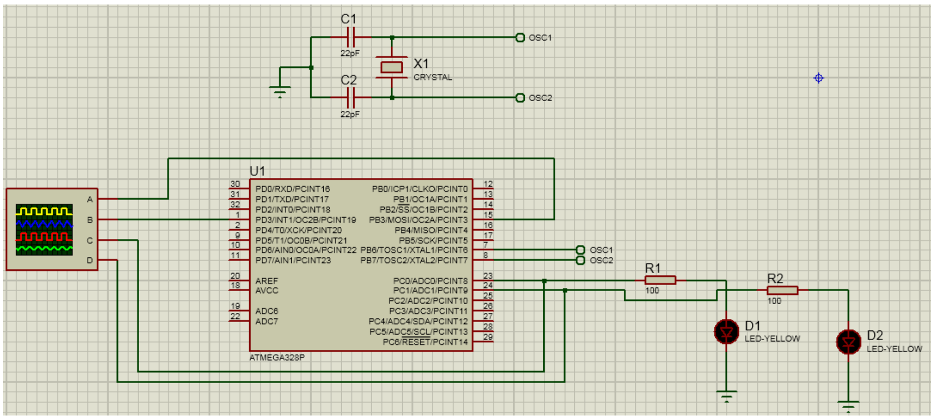
- The output can be seen @ *OC2A* and *OC2B* pins with a on time of 128μs and off time of 128μs ($\frac{0xFF*8}{16000000} = 127.5\mu s$).
- Also, *PC1* toglles for the overflow Timer2.

Timer2_asDelay

- The output can be seen *PC0* pin.

1.13 TimerCounter2_CTC

1.13.1 Circuit



1.13.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

void Timer2_asTimer()
{
        // MOde of operation to CTC Mode -- WGM2[2:0] === 010
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A & ~(1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        /* What to do when timer reaches the OCR2A */
        // toggle OC2A on each time when reaches the OCR2A
        // which is reflected in PB3
        // Output OC2A to toglle when reaches MAX -- COM2A[1:0] === 01
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // Output OC2B to toglle when reaches MAX -- COM2B1:0] === 01
        // COM2B[1](bit7) from TCCR2A, COM2B[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<5);
        TCCR2A = TCCR2A | (1<<4);
        // Enable Interrupt when counter matches OCR2A Rgister
        //  OCIE2A bit is enabled
        TIMSK2 = TIMSK2 | (1<<1);
        // setting the value till the counter should reach in OCR2A
        // for toggling of OC2A pin
        OCR2A = 0x32;
        // start timer by setting the clock prescalar
        // DIVIDE BY 8 from I/O clock
        // DIVIDE BY 8-- CS2[2:0] === 010
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<1);
        TCCR2B = TCCR2B & (~(1<<0) & ~(1<<2));
        // enabling global interrupt
        sei();
        // SO ON TIME = (1 + OCR2A) / (F_CPU / PRESCALAR)
        // ON TIME = 0X32 / (16000000/8) = 25.5us
        // since symmetric as toggling OFF TIME = 25.5us
        // hence, we get a square wave of fequency 1 / 50us = 20kHz
}
void Timer2_asDelayIn_ms(uint32_t delayInMs)
```

```
{

        // minimum delay being 4us -- choose like that
        // use PRESCALAR OF 1 -- 3us - 16us -- usage 3us - 16us -- factor=0 -- CS2[2:0]=1
        // use PRESCALAR OF 8 -- 3us - 128us -- usage 17us - 128us -- factor=3 -- CS2[2:0]=2
        // use PRESCALAR OF 64 -- 4us - 1.024ms -- usage 129us - 1024us -- factor=6 -- CS2[2:0]=3
        /* use PRESCALAR OF 256 -- 16us - 4.096ms.
         -- usage 1025us - 4096us -- factor=8 -- CS2[2:0]=4 */

        // MOde of operation to ctc Mode -- WGM2[2:0] === 010
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A & ~(1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        while(delayInMs--)
        {
                // for 1ms delay
                OCR2A = 249;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS2[2:0] === 011
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
                // actual delaying - wait until delay happens
                // checking OCF0A (compare match flag A) flag when match happns
                while((TIFR2 & 0x02) == 0x00);
                // clearing the compare match flag so that we can further utilize
                TIFR2 = TIFR2 | 0x02;
        }
}
int main(void)
{
    // making the PB3 and PD3 as output
    DDRD = DDRD | (1<<3);
    DDRB = DDRB | (1<<3);
        DDRC |= (1<<0);
        PORTC &= ~(1<<0);
        //Timer2_asTimer();
    while(1)
    {
                PORTC &= ~(1<<0);
                Timer2_asDelayIn_ms(10);
                PORTC |= (1<<0);
                Timer2_asDelayIn_ms(10);
    }
}

ISR(TIMER2_COMPA_vect)
{
        // toggle PC1 when matches
        PINC |= (1<<1);
}
```

1.13.3 Output

Timer2_asTimer

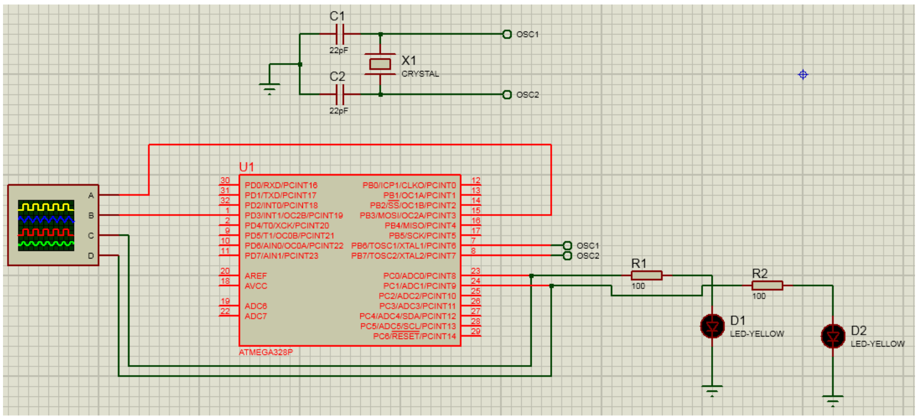
- The output can be seen @ *OC2A* and *OC2B* pins with a on time of 25.5μs and off time of 25.5μs ($\frac{(0x32+1)*8}{16000000} = 25.5\mu s$).
- Also, *PC1* toglles for the **TCNT2** matches **OCR2A**.

Timer2_asDelayIn_ms

- The output can be seen *PC0* pin.

1.14 TimerCounter2_FastPWM

1.14.1 Circuit



1.14.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer2_NonInverting_TOP_at_MAX()
{
        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 011
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        // here we set COM2A[1:0] as 10 for non-inverting
        // here we set COM2B[1:0] as 10 for non-inverting
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A | (1<<7);
        TCCR2A = TCCR2A & ~(1<<6);
        // which is reflected in PB35
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);
        // Enable Interrupt when TCN0 overflows TOP - here 0xFF
        //  TOV2 bit is enabled
        TIMSK2 = TIMSK2 | (1<<0);
        /* we use OCF0A flag - which is set at every time TCN0 reaches OCR2A
        here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR2A*/
        TIMSK2 = TIMSK2 | (1<<1);
        /* we use OCF0B flag - which is set at every time TCN0 reaches OCR2B
        here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR2B*/
        TIMSK2 = TIMSK2 | (1<<2);
        // Next we set values for OCR2A and OCR2B
        /* Since, TCNT2 goes till max(0xFF), we can choose
         OCR2A and OCR2B to any value below max(0xFFF)*/
        OCR2A = 0x19; // for 10% duty clcle
        OCR2B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
```

```
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_Inverting_TOP_at_MAX()
{
        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 011
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        // here we set COM2A[1:0] as 11 for inverting
        // here we set COM2B[1:0] as 11 for inverting
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A | (1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // which is reflected in PB35
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A | (1<<4);
        // Enable Interrupt when TCN0 overflows TOP - here 0xFF
        //  TOV2 bit is enabled
        TIMSK2 = TIMSK2 | (1<<0);
        /* we use OCF0A flag - which is set at every time TCN0 reaches OCR2A
                here we clear led(PC1),  so that we obtain the PWM when TCN0 reaches OCR2A*/
        TIMSK2 = TIMSK2 | (1<<1);
        /* we use OCF0B flag - which is set at every time TCN0 reaches OCR2B
                here we clear led(PC2),  so that we obtain the PWM when TCN0 reaches OCR2B*/
        TIMSK2 = TIMSK2 | (1<<2);
        // Next we set values for OCR2A and OCR2B
        /* Since, TCNT2 goes till max(0xFF), we can choose OCR2A
        and OCR2B to any value below max(0xFFF) */
        OCR2A = 0x19; // for 10% duty clcle
        OCR2B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_NonInverting_TOP_at_OCR2A()
{
        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2B[1:0] as 10 for non-inverting
        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        OCR2B = 0x60; // for pwm duty cylc
```

```
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_Inverting_TOP_at_OCR2A()
{
        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2B[1:0] as 11 for inverting
        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A | (1<<4);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        OCR2B = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_OC2A_Square()
{
        // MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A | (1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2B[1:0] as 01 for toggling of OC2A
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_FastPWMGeneration(uint32_t on_time_us, uint32_t off_time_us)
{
```

```
uint32_t total_time = on_time_us + off_time_us;
// MOde of operation to fast_pwm_top_max Mode -- WGM2[2:0] === 111
// WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
TCCR2A = TCCR2A | (1<<0);
TCCR2A = TCCR2A | (1<<1);
TCCR2B = TCCR2B | (1<<3);
// which is reflected in PD3
// COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
TCCR2A = TCCR2A | (1<<5);
TCCR2A = TCCR2A & ~(1<<4);

if(total_time <=3)
{
        // if total_time <= 3us -- so we stop clock
        OCR2A = 0;
        // start timer by setting the clock prescalar
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B & ~(1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
}
else if((3 < total_time)  && (total_time <= 16))
{
        OCR2A = ((total_time * 16) >> 0) - 1;
        OCR2B = ((on_time_us * 16) >> 0) - 1;
        // start timer by setting the clock prescalar
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
}
else if((16 < total_time)  && (total_time <= 128))
{
        OCR2A = ((total_time * 16) >> 3) - 1;
        OCR2B = ((on_time_us * 16) >> 3) - 1;
        // start timer by setting the clock prescalar
        //  dived by 8 from I/O clock
        //  CS2[2:0] === 010
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B & ~(1<<0);
        TCCR2B = TCCR2B | (1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
}
else if((128 < total_time)  && (total_time <= 1024))
{
        OCR2A = ((total_time * 16) >> 6) - 1;
        OCR2B = ((on_time_us * 16) >> 6) - 1;
        // start timer by setting the clock prescalar
        //  dived by 64 from I/O clock
        //  CS2[2:0] === 011
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B | (1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
}
else if((1024 < total_time)  && (total_time <= 4096))
{
        OCR2A = ((total_time * 16) >> 8) - 1;
        OCR2B = ((on_time_us * 16) >> 8) - 1;
```

```
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS2[2:0] === 100
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B | (1<<2);
        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }
        // max time = 4ms -- min freqency = 250 Hz
        //  min time = 4us -- max frequency = 250000 = 250khz
        Timer2_FastPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
        DDRD = DDRD | (1<<3);
        DDRB = DDRB | (1<<3);
        //Timer2_NonInverting_TOP_at_MAX();
         //Timer2_Inverting_TOP_at_MAX();
    //Timer2_NonInverting_TOP_at_OCR2A();
     //Timer2_Inverting_TOP_at_OCR2A();
    //Timer2_OC2A_Square();
    PWMGeneration(12, 1000);
    while(1)
    {
    }
}
ISR(TIMER2_OVF_vect)
{
}
ISR(TIMER2_COMPA_vect)
{
}
ISR(TIMER2_COMPB_vect)
{
}
```

1.14.3 Output

Timer2_NonInverting_TOP_at_MAX

- The output can be seen @ *OC2A* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of 10% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC2B* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of 75% ($\frac{75}{100} = 0xC0$).

Timer2_Inverting_TOP_at_MAX

- The output can be seen @ *OC2A* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0xFF = 0x19$).

- The output can be seen @ *OC2B* with a frequency of 62.74 kHz($\frac{0xFF*1}{16000000} = 15.9\mu s$) and duty cycle of (100 - 75)% ($\frac{75}{100} = 0xC0$).

Timer2_NonInverting_TOP_at_OCR2A

- The output can be seen @ *OC2B* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$) and duty cycle of 85% ($\frac{85}{100} = 0x60$).

Timer2_Inverting_TOP_at_OCR2A

- The output can be seen @ *OC2B* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$) and duty cycle of (100 - 85)% ($\frac{85}{100} = 0x60$).

Timer2_OC2A_Square

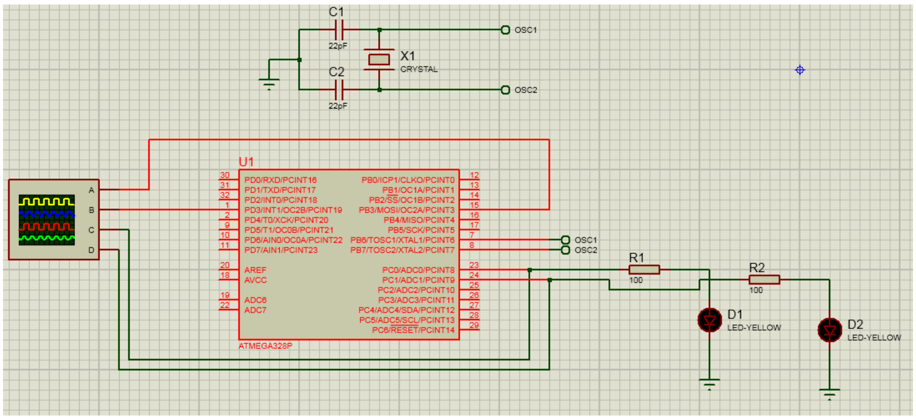
- The output can be seen @ *OC2A* with a frequency of 142.857 kHz($\frac{0x70*1}{16000000} = 7\mu s$).

PWMGeneration

- The output can be seen @ *O20B*.

1.15 TimerCounter2_PhaseCorrectedPWM

1.15.1 Circuit



2

1.15.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>
void Timer2_NonInverting_TOP_at_MAX()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 001
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        /* in TIMER2_phase_pwm_top_max, only two possiblites are there for
        COM2B[1:0] and COM2A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
        // here we set COM2A[1:0] as 10 for non-inverting
        // here we set COM2B[1:0] as 10 for non-inverting
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A | (1<<7);
        TCCR2A = TCCR2A & ~(1<<6);
        // which is reflected in PB35
```

```
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);
        /* we use overflow flag -- which is set at every time TCNO reaches TOP here 0xFF
        here, we toggle an led(PC0) at every overflow interrupt -
         this led(PC0) would give the frequency of PWM being
         generated -- done by PINC = PINC | 0X01;
        Also, we set the other leds(PC1 and PC2) so that they
        are make one when TCNO reaches 0x00 */
        // Enable Interrupt when TCNO overflows TOP - here 0xFF
        //  TOV2 bit is enabled
        TIMSK2 = TIMSK2 | (1<<0);
        // Next we set values for OCR2A and OCR2B
        /* Since, TCNT2 goes till max(0xFF), we can choose
        OCR2A and OCR2B to any value below max(0xFFF) */
        OCR2A = 0x19; // for 10% duty clcle
        OCR2B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_Inverting_TOP_at_MAX()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 001
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<3);
        /* in TIMER2_phase_pwm_top_max, only two possiblites are there for
        COM2B[1:0] and COM2A[1:0] i.e) 10(Inverting) and 11(Non- inverting) */
        // here we set COM2A[1:0] as 11 for inverting
        // here we set COM2B[1:0] as 11 for inverting
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A | (1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // which is reflected in PB35
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A | (1<<4);
        /* we use overflow flag -- which is set at every time TCNO
         reaches TOP here 0xFF
        here, we toggle an led(PC0) at every overflow interrupt -
        this led(PC0) would give the frequency of PWM being
         generated -- done by PINC = PINC | 0X01;
        Also, we set the other leds(PC1 and PC2) so that they are
        make one when TCNO reaches 0x00 */
        // Enable Interrupt when TCNO overflows TOP - here 0xFF
        //  TOV2 bit is enabled
        TIMSK2 = TIMSK2 | (1<<0);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till max(0xFF), we can choose OCR2A
        and OCR2B to any value below max(0xFFF)
        OCR2A = 0x19; // for 10% duty clcle
        OCR2B = 0xC0; // for 75% duty clcle
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
```

```
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global inerrupt
        sei();
}
void Timer2_NonInverting_TOP_at_OCR2A()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2A[1:0] as 10 for non-inverting
        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        OCR2B = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_Inverting_TOP_at_OCR2A()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2A[1:0] as 11 for inverting
        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A | (1<<4);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        OCR2B = 0x60; // for pwm duty cylc
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_OC2A_Square()
{
        // MOde of operation to phase_corrected_pwm_top_max Mode -- WGM2[2:0] === 101
```

```
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // here we set COM2B[1:0] as 01 for toggling of OC2A
        // which is reflected in PB3
        // COM2A[1](bit7) from TCCR2A, COM2A[0](bit6) from TCCR2A
        TCCR2A = TCCR2A & ~(1<<7);
        TCCR2A = TCCR2A | (1<<6);
        // Next we set values for OCR2A and OCR2B
        // Since, TCNT2 goes till OCR2A, we can choose OCR2B to any value below OCR2A
        OCR2A = 0x70; // for freqeuncy
        // start the timer by selecting the prescalr
        //  use the same clock from I/O clock
        //  CS2[2:0] === 001
        // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
        TCCR2B = TCCR2B | (1<<0);
        TCCR2B = TCCR2B & ~(1<<1);
        TCCR2B = TCCR2B & ~(1<<2);
        //enabled global interrupt
        sei();
}
void Timer2_PhaseCorrectedPWMGeneration(uint32_t On_time_us, uint32_t Off_time_us)
{
        // Since, it is dual slope, the time would be doubled for one cylce, so we divide by 2
        uint32_t total_time = (On_time_us>>1) + (Off_time_us>>1);
        uint32_t on_time_us = On_time_us >> 1;
        // MOde of operation to phase_corrected_phase_top_max Mode -- WGM2[2:0] === 101
        // WGM2[2](bit3) from TCCR2B, WGM2[1](bit1)  from TCCR2A, WGM2[0](bit0)  from TCCR2A
        TCCR2A = TCCR2A | (1<<0);
        TCCR2A = TCCR2A & ~(1<<1);
        TCCR2B = TCCR2B | (1<<3);
        // which is reflected in PD3
        // COM2B[1](bit5) from TCCR2A, COM2B[0](bit4) from TCCR2A
        TCCR2A = TCCR2A | (1<<5);
        TCCR2A = TCCR2A & ~(1<<4);
        if(total_time <=3)
        {
                // if total_time <= 3us -- so we stop clock
                OCR2A = 0;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((3 < total_time)  && (total_time <= 16))
        {
                OCR2A = ((total_time * 16) >> 0) - 1;
                OCR2B = ((on_time_us * 16) >> 0) - 1;
                // start timer by setting the clock prescalar
                //  use the same clock from I/O clock
                //  CS2[2:0] === 001
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((16 < total_time)  && (total_time <= 128))
        {
                OCR2A = ((total_time * 16) >> 3) - 1;
```

```c
                OCR2B = ((on_time_us * 16) >> 3) - 1;
                // start timer by setting the clock prescalar
                //  dived by 8 from I/O clock
                //  CS2[2:0] === 010
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((128 < total_time)  && (total_time <= 1024))
        {
                OCR2A = ((total_time * 16) >> 6) - 1;
                OCR2B = ((on_time_us * 16) >> 6) - 1;
                // start timer by setting the clock prescalar
                //  dived by 64 from I/O clock
                //  CS2[2:0] === 011
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B | (1<<0);
                TCCR2B = TCCR2B | (1<<1);
                TCCR2B = TCCR2B & ~(1<<2);
        }
        else if((1024 < total_time)  && (total_time <= 4096))
        {
                OCR2A = ((total_time * 16) >> 8) - 1;
                OCR2B = ((on_time_us * 16) >> 8) - 1;
                // start timer by setting the clock prescalar
                //  divide by256 from I/O clock
                //  CS2[2:0] === 100
                // CS2[2](bit2) from TCCR2B,CS2[1](bit1) from TCCR2B,CS2[0](bit0) from TCCR2B
                TCCR2B = TCCR2B & ~(1<<0);
                TCCR2B = TCCR2B & ~(1<<1);
                TCCR2B = TCCR2B | (1<<2);
                        }
        else if(total_time > 4096)
        {
                // dont' cross more than 4.096ms
        }
}
void PWMGeneration(double duty_cycle_percent,uint32_t freqeuncy)
{
        double total_time_us = (1000000.0/freqeuncy);
        double on_time_us = (duty_cycle_percent/100.0) * total_time_us;
        if (on_time_us<1.0)
        {
                on_time_us = 1;
        }
        // max time = 8ms -- min freqency = 125 Hz
        //  min time = 8us -- max frequency = 250000 = 125khz
        Timer2_PhaseCorrectedPWMGeneration(on_time_us, total_time_us - on_time_us);
}
int main(void)
{
        DDRD = DDRD | (1<<3);
        DDRB = DDRB | (1<<3);

        //Timer2_NonInverting_TOP_at_MAX();
        Timer2_Inverting_TOP_at_MAX();
    //Timer2_Inverting_TOP_at_OCR2A();
    //Timer2_NonInverting_TOP_at_OCR2A();
    //Timer2_OC2A_Square();
    //PWMGeneration(71, 1000);
    while(1)
    {
```

```
    }
}
ISR(TIMER2_OVF_vect)
{
}
ISR(TIMER2_COMPA_vect)
{
}
ISR(TIMER2_COMPB_vect)
{
}
```

1.15.3 Output

Timer2_NonInverting_TOP_at_MAX

- The output can be seen @ *OC2A* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of 10% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC2B* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of 75% ($\frac{75}{100} * 0xFF = 0xC0$).

Timer2_Inverting_TOP_at_MAX

- The output can be seen @ *OC2A* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$) and duty cycle of (100 - 10)% ($\frac{10}{100} * 0xFF = 0x19$).
- The output can be seen @ *OC2B* with a frequency of 31.372 kHz($\frac{510*1}{16000000} = 31.8\mu s$)and duty cycle of (100 - 75)% ($\frac{75}{100} * 0xFF = 0xC0$).

Timer0_NonInverting_TOP_at_OCR2A

- The output can be seen @ *OC2B* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$) and duty cycle of 85% ($\frac{85}{100} * 0x70 = 0x60$).

Timer2_Inverting_TOP_at_OCR2A

- The output can be seen @ *OC2B* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$) and duty cycle of (100 - 85)% ($\frac{85}{100} * 0x70 = 0x60$).

Timer2_OC2A_Square

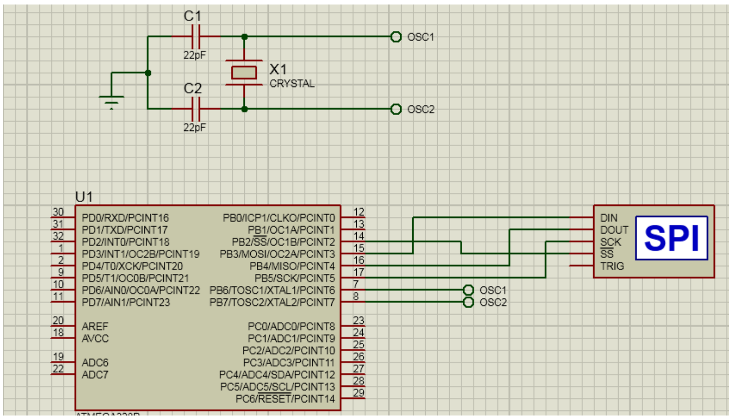
- The output can be seen @ *OC2TiA* with a frequency of 71.42 kHz($\frac{(2*0x70)*1}{16000000} = 14\mu s$).

PWMGeneration

- The output can be seen @ *OC2B*.

1.16 SPI

1.16.1 Circuit



1.16.2 Code

```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>
void SPI_Init()
{
    // making SCK, MOSI, SS' as outptut
    DDRB |= (1<<DDB2) | (1<<DDB3) | (1<<DDB5);
    // making MISO as input
    DDRB &= ~(1<<DDB4);
    // making SCK, MOSI, as low
    PORTB &= ~(1<<PORTB3) & ~(1<<PORTB5);
    // making SS' as high
    PORTB |= (1<<PORTB2);
    // Select MSB first or LSB first by DORD
    SPCR &= ~(1<<DORD);
    // Select this as Master
    SPCR |= (1<<MSTR);
    // Let the clock polarity be SCK is low when idle
    SPCR &= ~(1<<CPOL);
    // Sampled at Rising or Falling Edge
    // we choose rising edge
    SPCR &= ~(1<<CPHA);
    // Selecting a SCK frequnecy
    // we select Fosc/4 by 000
    SPSR &= ~(1<<SPI2X);
    SPCR &= ~(1<<SPR1);
    SPCR &= ~(1<<SPR0);
    // dISBALE SPIE bit for interrupt on Serial Transfer Completion
    SPCR &= ~(1<<SPIE);
    // Enabling SPI
    SPCR |= (1<<SPE);
}
uint8_t SPITransferReceive(uint8_t data_)
{
    SPDR = data_;
    // wait till serial transmission is complete by checking the SPI Interrupt Flag
    while((SPSR & (1<<SPIF)) == 0 ) {};
    // return the recieved data - can use it or ignore it
    return SPDR;
}
int main(void)
```

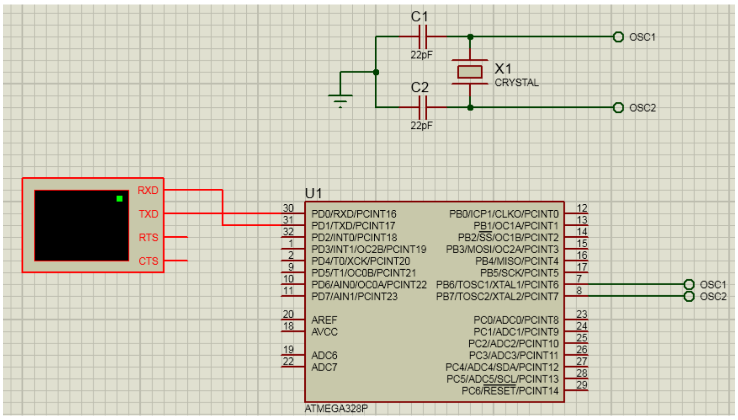
```
{
    SPI_Init();
    PORTB &= ~(1<<PORTB2);
    SPITransferReceive('A');
    while(1)
    {

    }
}
```

1.16.3 Output

The Output can be seen @ the SPI debugger.

1.17 USART0

1.17.1 Circuit



1.17.2 Code

```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>

void USART0init()
{
    // Setting up the Mode
    // Select the Asyncronous Master Mode.
    // Setting UMSEL0[1:0] in UCSR0C to 00
    UCSR0C &= ~(1<<UMSEL00);
    UCSR0C &= ~(1<<UMSEL01);

    // setting up the Buad rate
    // Due to The Clock rate being 8MHz, for a buad rate of 9600
    // UBRR0 = (fosc / (16*BAUD)) -1
    // So UBRR0 = (16000000 / (16 * 9600)) - 1 = 0x67
    UBRR0H = 0x00;
        UBRR0L = 0x67;

    // setting up the Frame Format
    // Let's select 8-bit data bits, no parity, and 1 stop bit
    // 8 - bit data bits
    // By selecting UCSZ0[2:0] in UCSR0C and UCSR0B register to be 011
        UCSR0B &= ~(1<<UCSZ02);
        UCSR0C |= (1<<UCSZ01);
```

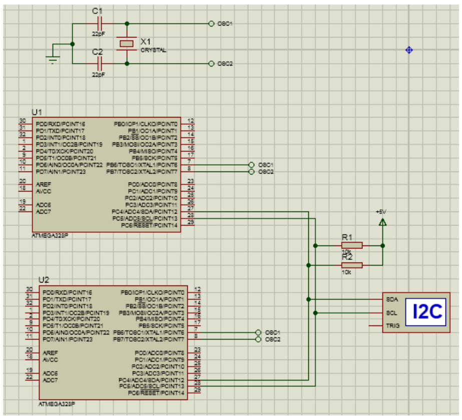
```
        UCSR0C |= (1<<UCSZ00);
    // No parity
    // By selecting UPM0[1:0] in UCSR0C to 00
    UCSR0C &= ~(1<<UPM01);
    UCSR0C &= ~(1<<UPM00);
    // 1 stop bit
    // By selecting USBS0 in UCSR0C to 0
    UCSR0C &= ~(1<<USBS0);

    // Disabling any interrupts
    UCSR0B &= ~(1<<7);
    UCSR0B &= ~(1<<6);
    UCSR0B &= ~(1<<5);

    // Enabling Transmitter
    UCSR0B |= (1<<TXEN0);
    // Enabling Receiver
    UCSR0B |= (1<<RXEN0);

}
void USART0sendChar(uint8_t data_)
{
        //cHECKING if transmitet buffer is empty
        while((UCSR0A & (1<<UDRE0)) == 0x00){};
        UDR0 = data_;
}
void USART0sendString(uint8_t *c_data_)
{
        while(*c_data_ != '\0')
        {
                USART0sendChar(*c_data_++);
        }
}
uint8_t USART0receiveChar()
{
        // wait for thedate to be recied
        while((UCSR0A & (1<<RXC0)) == 0x00){};
        return UDR0;
}
void USART0receiverStringUntil(uint8_t *rec_buff,uint8_t deliminator)
{
        uint16_t i=0;
        uint8_t curr_char = USART0receiveChar();
        while(curr_char != deliminator)
        {
                rec_buff[i] = curr_char;
                curr_char = USART0receiveChar();
                i++;
        }
        rec_buff[i] = '\0';
}
int main(void)
{
        uint8_t rec_buff[1024];
        USART0init();
    USART0sendString((uint8_t *)"This is working\n\r");
    while(1)
    {
        USART0receiverStringUntil(rec_buff, '\n');
        _delay_ms(100);
    }
}
```

1.17.3 Output

The Output can be seen @ the Virtual Terminal.

1.18 TwinWireInterface

1.18.1 Circuit



1.18.2 Code

Master Code

```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>
uint8_t status = 0;
void I2C_Master_Init()
{
        // Intialize the I2C clock frequency to 100kHz
        // let the prescalr be 1
        // f_i2c =  F_CPU / (16 + (2*TWBR*Prescaler)) = 72
        // setting the TWBR register.
        TWBR = 72;

        // writing 1 to prscalre
        // setting the TWPS bits in TWSR to 00
        TWSR &= ~(1<<TWPS0);
        TWSR &= ~(1<<TWPS1);
}
uint8_t I2C_Master_Status()
{
        // Status value are available from TWSR[7:3]
        return TWSR & 0XF8;
}
uint8_t I2C_Master_START()
{
        // Enabling the TWI interface
        TWCR |= (1<<TWEN);
        // sending START condition
        TWCR |= (1<<TWSTA);
        // Do the transaction
        TWCR |= (1<<TWINT);
        // Checking if START condition is sent correctly
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        // checking status if START condition is sent correctily
        if(status == 0x08)
```

```c
        {
                // no error occured
                return 0;
        }
        else
        {
                // error occured
                return 0;
        }
}
uint8_t I2C_Master_STOP()
{
        // Removing Start condition on bit
        TWCR &= ~(1<<TWSTA);
        // sending STOP condition
        TWCR |= (1<<TWSTO);

        // Do the transaction
        TWCR |= (1<<TWINT);

        // disaabling stop and interface


        TWCR &= ~(1<<TWSTO);
        TWCR &= ~(1<<TWEN);

        return 0;
}
uint8_t I2C_Master_Mode(uint8_t slave_address, uint8_t transmiter0_receiver1)
{
        // Entering MASTER mode
        // Writing SLA+W into TWDR for transmiiter and SLA+R for receiver
        // slave address must be MSB first
        // slave address is left shifted by 1 in order to accompany the R/W bit
        TWDR = (slave_address<<1) | transmiter0_receiver1;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        // For transmitter the staus would have to be 0x18 and for receiver 0x40
        uint8_t status_val_checker = (transmiter0_receiver1==0) ? 0x18 : 0x40;
        if(status == status_val_checker)
        {
                // no error occured
                return 0;
        }
        else
        {
                // error occured
                return 0;
        }
}
uint8_t I2C_Master_DataTransmitByte(uint8_t data_)
{
        // Data packet is transmitted
        // Writing data intor TWDR
        TWDR = data_;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Master_Status();
        if(status == 0x28)
        {
```

```
                // ACK received and still data can be sent
                return 0;
        }
        else if(status == 0x30)
        {
                // NACK received and this is the last data so stop
                return 1;
        }
        else
        {
                // error occured
                return 2;
        }
}
void I2C_Master_DataTransmitString(uint8_t *cdata)
{
        while(*cdata != '\0')
        {
                status = I2C_Master_DataTransmitByte(*cdata++) ;
                if(status == 0)
                {
                        // ACK received and still data can be sent
                        // continue
                }
                else if(status == 1)
                {
                        // NACK received and this is the last data so stop
                        return;
                }
                else
                {
                        // error occured
                        return;
                }
        }
}

uint8_t I2C_Master_DataReceiveByte()
{
        uint8_t value_ = 0;

        // Data packet is recieved
        TWCR |= (1<<TWINT);
        // Do the transaction
        while((TWCR & (1<<TWINT )) == 0x00)
        {
                value_ = TWDR;
        }


        status = I2C_Master_Status();
        if(status == 0x58)
        {
                // no error occured
                return value_;
        }
        else
        {
                // error occured
                return 1;
        }
}
void I2C_Master_DataReceiveString(uint8_t *recData,uint8_t NUMBYTE)
```

```
{
        uint8_t i=0;
        recData[NUMBYTE] = '\0';
        while(i < NUMBYTE)
        {
                // Enabling the Acknowledment bit for replying positive ACK
                TWCR |= (1<<TWEA);
                if(i==(NUMBYTE-1))
                {
                        // disbale the Acknowledment bit for replying Negatice ACK for last byte
                        TWCR &= ~(1<<TWEA);
                }
                status = I2C_Master_DataReceiveByte();
                if(status==0xFF)
                        return;
                else
                        recData[i] = status;
                i++;
        }
}
#define SLAVE_ADDRESS 0b01010101

int main(void)
{
        I2C_Master_Init();
        DDRC |= (1<<0) | (1<<1) | (1<<2);
        PORTC |= (1<<0) | (1<<1) | (1<<2);


        I2C_Master_START();
        I2C_Master_Mode(SLAVE_ADDRESS, 0);
        I2C_Master_DataTransmitString((uint8_t *)"K");
        I2C_Master_STOP();

        I2C_Master_START();
        I2C_Master_Mode(SLAVE_ADDRESS, 0);
        I2C_Master_DataTransmitString((uint8_t *)"Narendiran");
        I2C_Master_STOP();

        // uint8_t recData[15];
        // I2C_Master_START();
        // I2C_Master_Mode(SLAVE_ADDRESS, 1);
        // I2C_Master_DataReceiveString(recData, 3);
        // I2C_Master_STOP();

        PINC |= (1<<2);
    while(1)
    {

    }
}
```

Slave Code

```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>
#include <string.h>
uint8_t status = 0;
void I2C_SlaveInit(uint8_t my_address)
```

```
{
        // slave address  and last LSB 0 is for general call
        TWAR = (my_address<<1) & 0xFE;
        // Enabling the TWI interface.
        TWCR |= (1<<TWEN);
        // Disabling Start and Stop conditon bits
        TWCR &= ~(1<<TWSTA);
        TWCR &= ~(1<<TWSTO);

}
uint8_t I2C_Status()
{
        // Status value are available from TWSR[7:3]
        return TWSR & 0XF8;
}

uint8_t I2C_SlaveMode( uint8_t transmiter0_receiver1)
{
        // Acknowldege the address
        TWCR |= (1<<TWEA);
        // Watiting for the Master to call this slave
        while((TWCR & (1<<TWINT )) == 0x00);
        status = I2C_Status();
        // For transmitter the staus would have to be 0xA8 and for receiver 0x60
        uint8_t status_val_checker = (transmiter0_receiver1==0) ? 0xA8 : 0x60;
        if(status == status_val_checker)
        {
                // Master called this slave
                return 0;
        }
        else
        {
                // error occured
                return 1;
        }
}
uint8_t I2C_Slave_DataTransmitByte(uint8_t data_)
{
        // Data packet is transmitted
        // Writing data intor TWDR
        TWDR = data_;
        // Do the transaction
        TWCR |= (1<<TWINT);
        while((TWCR & (1<<TWINT )) == 0x00);

        status = I2C_Status();
        if(status == 0xB8)
        {
                // ACK received and still data can be sent
                return 0;
        }
        else if(status == 0xC8)
        {
                // NACK received and this is the last data so stop
                return 1;
        }
        else
        {
                // error occured
                return 2;
        }
}
void I2C_Slave_DataTransmitString(char *cdata)
```

```
{
        uint8_t i = 0;
        while(cdata[i] != '\0')
        {
                status = I2C_Slave_DataTransmitByte(cdata[i]) ;
                i++;
                if(status == 0)
                {
                        // ACK received and still data can be sent
                        // continue
                }
                else if(status == 1)
                {
                        // NACK received and this is the last data so stop
                        return;
                }
                else
                {
                        // error occured
                        return;
                }
        }
}

uint8_t I2C_Slave_DataReceiveByte()
{
        uint8_t value_ = 0;

        // Data packet is recieved
        TWCR |= (1<<TWINT);
        // Do the transaction
        while((TWCR & (1<<TWINT )) == 0x00)
        {
                value_ = TWDR;
        }

        status = I2C_Status();
        if(status == 0x80)
        {
                // Data is sent and ACK has been returned
                return value_;
        }
        else if(status == 0x88)
        {
                // Data is sent and NACK has been returned for last byte
                return value_;
        }
        else
        {
                // error occured
                return 0xFF;
        }
}
void I2C_Slave_DataReceiveString(uint8_t *recData,uint8_t NUMBYTE)
{
        uint8_t i=0;
        recData[NUMBYTE] = '\0';
        while(NUMBYTE > 0)
        {
                NUMBYTE = NUMBYTE - 1;
                // Enabling the Acknowledment bit for replying positive ACK
                TWCR |= (1<<TWEA);
                if(NUMBYTE==0)
```

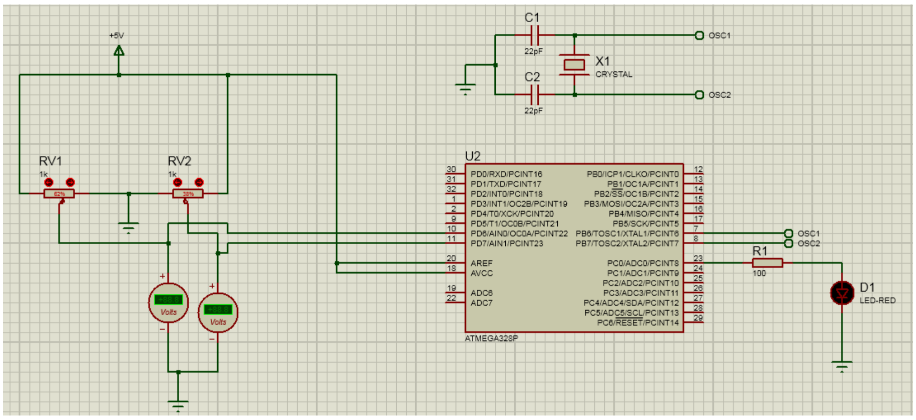
```
                {
                        // disbale the Acknowledment bit for replying Negatice ACK for last byte
                        TWCR &= ~(1<<TWEA);
                }
                status = I2C_Slave_DataReceiveByte();
                if(status==0xFF)
                        return;
                else
                        recData[i] = status;
                i++;
        }
}
#define SLAVE_ADDRESS 0b01010101


int main(void)
{
        DDRC |= (1<<0) | (1<<1) | (1<<2);
        PORTC |= (1<<0) | (1<<1) | (1<<2);


        I2C_SlaveInit(SLAVE_ADDRESS);
        I2C_SlaveMode(1);
        uint8_t recData[11];
        I2C_Slave_DataReceiveString(recData, 1);
        TWCR &= ~(1<<TWEN);
        if(recData[0]=='K')
                PINC |= (1<<0);

        I2C_SlaveInit(SLAVE_ADDRESS);
        I2C_SlaveMode(1);
        I2C_Slave_DataReceiveString(recData, 5);
        TWCR &= ~(1<<TWEN);
        if(strcmp((char *)recData, (char *)"Naren")==0)
                PINC |= (1<<1);

        // char *abcd ="AaBbCa";
        // I2C_SlaveInit(SLAVE_ADDRESS);
        // I2C_SlaveMode(0);
        // // Weird bug happens when keep more than 3
        // for(int j=0;j<2;j++)
        // I2C_Slave_DataTransmitByte(abcd[j]);
        // TWCR &= ~(1<<TWEN);
        // PINC |= (1<<2);
    while(1)
    {

    }
}
```

1.18.3 Output

The Output can be seen @ the I2C debugger.

1.19 AnalogComparator

1.19.1 Circuit



1.19.2 Code

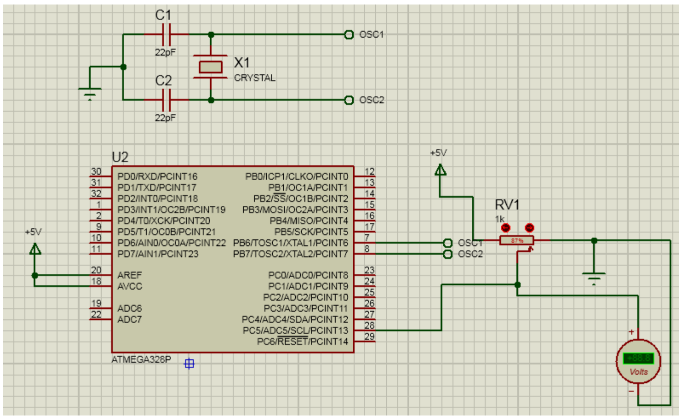
```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

void AnalogCompartorInit()
{
        /* Disabling the Analog Comparator Multiplexer Enable bit
         so that AIN1 is selected as positive input */
        ADCSRB &= ~(1<<ACME);
        /* Disabling the Analog Comparator Bandgap Select bit
        so that AIN0 is selected as negative input */
        ACSR &= ~(1<<ACBG);
        // Choosing the interrupt mode to toggle ACO bit
        // By selecting 00 to ACIS[1:0]
        ACSR &= ~(1<<ACIS1);
        ACSR &= ~(1<<ACIS0);
        // Enabling the Analog Comparator interrupt Enable to see the output
        ACSR |= (1<<ACIE);
        // enabling the Analog Comparator by clearing the Analog Comparator Disable bit
        ACSR &= ~(1<<ACD);
        sei();
}

int main(void)
{
        // making the AIN0(PD6) and AIN1(PD7) as input
        DDRD &= ~(1<<6);
        DDRD &= ~(1<<7);
        // making PC0 as output - to show output
        DDRC |= (1<<0);
    while(1)
    {
    }
}
ISR(ANALOG_COMP_vect)
{
    PINC |= (1<<0);
}
```

1.19.3 Output

The Output can be seen @ *PC0* by changing the voltages.

1.20 AnalogToDigital

1.20.1 Circuit



1.20.2 Code

```
#define F_CPU 8000000L

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
uint16_t ADC_SingleConversion(uint8_t channel_no)
{
        DDRC &= ~(1<<channel_no);
        // Selecting Voltage Referece
        // Lets use AREF pin
        // REFS[1:0] -- 00
        ADMUX &= ~(1<<REFS0);
        ADMUX &= ~(1<<REFS1);
        // Selecting the Presentation of ADC output
        // Right adjust - ADLAR == 0
        ADMUX &= ~(1<<ADLAR);
        // SELECTINT the channel for ADC
        // LET'S select channel_no
        // MUX[3:0]&0xF0 | channel_no
        ADMUX = (ADMUX & 0XF0) | channel_no;
        // for single conversion - disabling ADC auto trigger
        // ADATE == 0
        ADCSRA &= ~(1<<ADATE);
        // disable the interrrupt by disbaling ADIE bit
        // ADIE == 0
        ADCSRA &= ~(1<<ADIE);
        //  Prescaler be 64 so that we get 8Mhz/64 = 125kHz
        // ADPS[2:0] -- 110
        ADCSRA |= (1<<ADPS2) | (1<<ADPS1);
        ADCSRA &= ~(1<<ADPS0);
        // ENABLING adc
        ADCSRA |= (1<<ADEN);
        // STARTING CONVERSIOn
        ADCSRA |= (1<<ADSC);
                // since single conversion, we can check start conversion bit
        while((ADCSRA & (1<<ADSC)))
        {
        }
```

```
        // RESSETTING THE Flag
        // ADCSRA |= (1<<ADIF);
        return ADC;
}
volatile uint16_t free_running_value=0;
void ADC_FreeRunningInit(uint8_t channel_no)
{
        DDRC &= ~(1<<channel_no);
        // Selecting Voltage Referece
        // Lets use AREF pin
        // REFS[1:0] -- 00
        ADMUX &= ~(1<<REFS0);
        ADMUX &= ~(1<<REFS1);
        // Selecting the Presentation of ADC output
        // Right adjust - ADLAR == 0
        ADMUX &= ~(1<<ADLAR);
        // SELECTINT the channel for ADC
        // LET'S select channel_no
        // MUX[3:0]&0xF0 | channel_no
        ADMUX = (ADMUX & 0XF0) | channel_no;
        // Select the Auto Trigger source
        // for free running, use 000 for ADTS[2:0] in ADCSRB
        ADCSRB &= ~(1<<ADTS2);
        ADCSRB &= ~(1<<ADTS1);
        ADCSRB &= ~(1<<ADTS0);
        // for free runing conversion - enable ADC auto trigger
        // ADATE == 1
        ADCSRA |= (1<<ADATE);
        // enable the interrrupt by enabling ADIE bit
        // ADIE == 1
        ADCSRA |= (1<<ADIE);
        //  Prescaler be 64 so that we get 8Mhz/64 = 125kHz
        // ADPS[2:0] -- 110
        ADCSRA |= (1<<ADPS2) | (1<<ADPS1);
        ADCSRA &= ~(1<<ADPS0);
        // ENABLING adc
        ADCSRA |= (1<<ADEN);
        // STARTING CONVERSIOn
        ADCSRA |= (1<<ADSC);
        sei();
}
int main(void)
{
        ADC_FreeRunningInit(5);
    while(1)
    {
                // ADC_SingleConversion(5);
                // _delay_ms(100);
    }
}
ISR(ADC_vect)
{
        free_running_value = ADC;
        // ADCSRA |= (1<<ADIF);
}
```

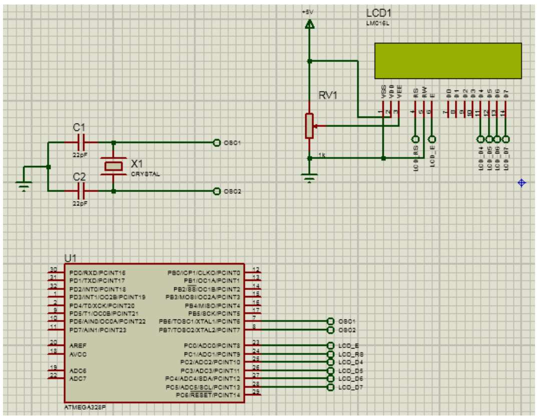
1.20.3 Output

The Output can be seen @ watch windows by seeing the **ADC** register by changing the voltages.

Applications

2.1 BasicLCD

2.1.1 Circuit



2.1.2 Code

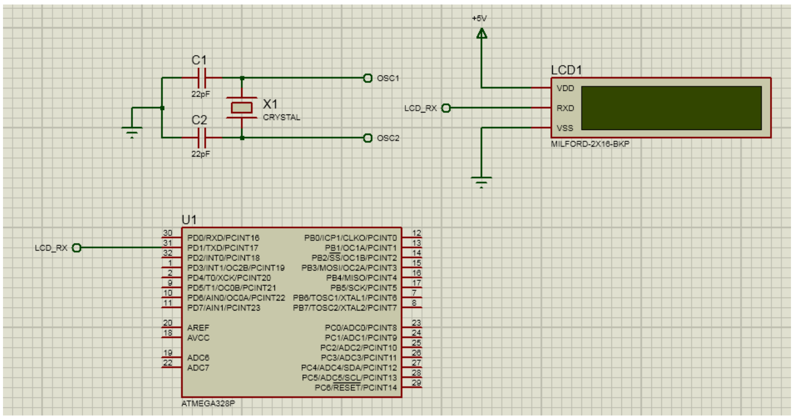
```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>


#include "LCDinclude.c"

int main(void)
{
        LCD_init();
        LCD_send_string("workIng?**--");
        LCD_display_on_cursor_on_blink_on();
        _delay_ms(1000);
        LCD_display_rightShift();
    while (1)
    {

    }
}
```

2.1.3 Output

The Output can be seen @ the LCD display.

2.2 UARTLCD

2.2.1 Circuit



2.2.2 Code

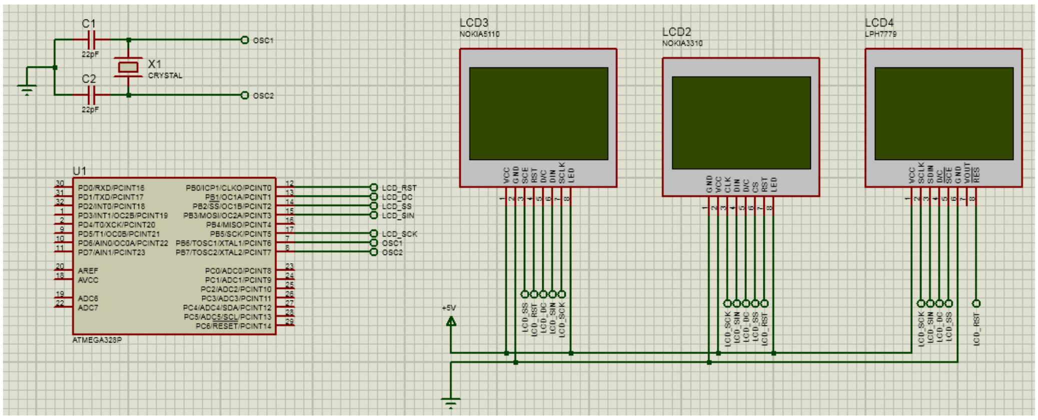
```
#define F_CPU 16000000L

#include <avr/io.h>
#include <util/delay.h>


#include "UARTLCDinclude.c"

int main(void)
{
        UARTLCD_init();
        UARTLCD_send_string("workIng?**--");
        UARTLCD_display_on_cursor_on_blink_on();
        _delay_ms(1000);
        UARTLCD_display_rightShift();
    while (1)
    {

    }
}
```

2.2.3 Output

The Output can be seen @ the LCD display.

2.3 SPILCD

2.3.1 Circuit



2.3.2 Code

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

// Reset
#define RST PORTB0
// Data/Command
#define DC PORTB1
// Chip Enable
#define SS PORTB2
// Data In
#define SIN PORTB3
// Serial Clock
#define SCK PORTB5

#define chip_enable PORTB &= ~(1<<SS)
#define chip_disable PORTB |= (1<<SS)
#define cmd_mode PORTB &= ~(1<<DC)
#define data_mode PORTB |= (1<<DC)


void SPILCD_init_pins()
{
        DDRB |= (1<<RST) | (1<<DC) | (1<<SS) | (1<<SIN) | (1<<SCK);

        PORTB |= (1<<RST); // initially let's reset
        PORTB &= ~(1<<DC); // initially command mode
        PORTB |= (1<<SS); // initially disable chip
        PORTB &= ~(1<<SIN); // initially SIN is 0
        PORTB &= ~(1<<SCK); // initially SCK is 0

}
void SPILCD_shiftOut(  uint8_t val)
{
        uint8_t i;

        for (i = 0; i<8; i++)
        {
                if ((val & (1<<(7-i))) == 0)
                {
                        PORTB &= ~(1<<SIN);
```

```c
                }
                else
                {
                        PORTB |= (1<<SIN);
                }

                PORTB |= (1<<SCK);
                PORTB &= ~(1<<SCK);
        }
}
void SPILCD_reset_procedure()
{
        PORTB |= (1<<RST); // initially let's reset

        chip_disable;
        PORTB &= ~(1<<RST); // remove reset
        _delay_ms(100);
        PORTB |= (1<<RST); // initially let's reset
}
void SPILCD_send_cmd(uint8_t cmd_)
{
        cmd_mode;

        chip_enable;

        SPILCD_shiftOut(cmd_);
        chip_disable;
}
void SPILCD_send_data(uint8_t data_)
{
        data_mode;

        chip_enable;
        SPILCD_shiftOut(data_);
        chip_disable;
}
void SPILCD_init()
{
        SPILCD_init_pins();
        SPILCD_reset_procedure();


        /* Function Set -- DB[7:0] ---> 0010_0(PD)(V)(H)
           PD = 1 - POWER DOWN mode
           PD = 0 - chip is active mode
           V = 0 - horizontal addressing
           V = 1 - vertical addressing
           H = 0 - Basic Instruction Set
           H = 1 - Extended Instruction Set */
        // We use chip active, horizontal addressing and extended instruction Set
        // Extended because we need to set bias, temperature and OPERATING voltage
        // So DB[7:0] ---> 0010_0001 = 0X21
        SPILCD_send_cmd(0x21);
        /* SEtting Vop - lcd voltage - operating voltage
        DB[7:0] ---> 1(Vop6)(Vop5)(Vop4)(Vop3)(Vop2)(Vop1)(Vop0)
        -- setting 5V we need == 100_0000 */
        // So DB[7:0] ---> 1100_0000 = 0XC0
        SPILCD_send_cmd(0xc0);

        /* Bias System -- DB[7:0] ---> 0001_0(BS2)(BS1)(BS0)
           -- setting bias voltage level (n = 4, 1:48) --- 011
           So DB[7:0] ---> 0001_0011 = 0X13 */
        SPILCD_send_cmd(0x13);
```

```c
        /* Temperature Control  -- DB[7:0] ---> 0000_01(TC1)(TC0)
           we take temparture coefficent 3 */
        // So PB[7:0] ---> 0000_0111 = 0X07
    SPILCD_send_cmd(0x07);

        /* Function Set -- DB[7:0] ---> 0010_0(PD)(V)(H)
           PD = 1 - POWER DOWN mode
           PD = 0 - chip is active mode
           V = 0 - horizontal addressing
           V = 1 - vertical addressing
           H = 0 - Basic Instruction Set
           H = 1 - Extended Instruction Set */
        // We use chip active, horizontal addressing and BASIC instruction Set
        // Basic because we are going to work now
        // So DB[7:0] ---> 0010_0000 = 0X20
        SPILCD_send_cmd(0x20);

        /* Display Control -- DB[7:0] ---> 0000_1(D)0(E)
           D | E ------- Mode ------- Description
           0   0      Display Blank    No Display on LCD
           0   1      Normal Mode      usual display on LCD
           1   0      All Segment on   every position in the LCD is on
           1   1      Inverse Mode     Display data is inverted */
        // So we select NOrmal mode
        // So DB[7:0] ---> 0000_1100 = 0x0c
        SPILCD_send_cmd(0x0C);

        /* setting X address varying pixel from 0 to 83[53H]
           PB[7:0] ---> 1(X6)(X5)(X4)(X3)(X2)(X1)(X0)
           X[6:0] is from 000_0000[00H] to 101_0011[53H] */
        // We select the middle column == 84/2 = 42[2A]
        // So PB[7:0] ---> 1010_1010 = 0xAA
        SPILCD_send_cmd(0xAA);

        /* setting Y address varying blank from 0 to 5[5H]
           PB[7:0] ---> 0100_0(Y2)(Y1)(Y0)
           Y[2:0] is from 000[0H] to 101[5H] */
        // We select the middle row == 0
        // So PB[7:0] ---> 0100_0000 = 0x40
        SPILCD_send_cmd(0x40);
        /* Sending data to draw
           DB[7:0] ---> (D7)(D6)(D5)(D4)(D3)(D2)(D1)(D0)
           each bit represent each pixel in a blank */
        // So lets' draw something
        SPILCD_send_data(0b10101010);
}
void SPILCD_pixel(uint8_t blank, uint8_t column)
{
        /* setting X address varying pixel from 0 to 83[53H]
        PB[7:0] ---> 1(X6)(X5)(X4)(X3)(X2)(X1)(X0)
        X[6:0] is from 000_0000[00H] to 101_0011[53H] */
        SPILCD_send_cmd(0x80 + column);

        /* setting Y address varying blank from 0 to 5[5H]
        PB[7:0] ---> 0100_0(Y2)(Y1)(Y0)
        Y[2:0] is from 000[0H] to 101[5H] */
        SPILCD_send_cmd(0x40 + blank);
}
void SPILCD_demo()
{
        SPILCD_pixel(0,0);
        SPILCD_send_data(0b11111111);
        SPILCD_pixel(0,2);
```

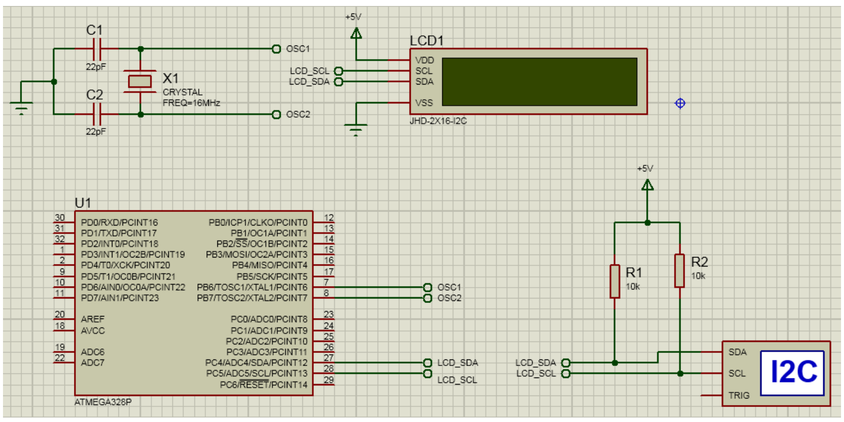
```c
        SPILCD_send_data(0b11111111);
        SPILCD_pixel(0,4);
        SPILCD_send_data(0b11110000);
        SPILCD_pixel(0,6);
        SPILCD_send_data(0b00001111);
        SPILCD_pixel(0,8);
        SPILCD_send_data(0b11001100);
        SPILCD_pixel(0,10);
        SPILCD_send_data(0b10010010);

        for (uint8_t i=0; i<84; i++)
        {
                SPILCD_pixel(2,0+i);
                SPILCD_send_data(0b00000001);
        }
        for (uint8_t i=0;i<6;i++)
        {
                SPILCD_pixel(0+i,12);
                SPILCD_send_data(0b11111111);
        }
        uint8_t heartine[]={ 0x18, 0x3c, 0x7c, 0x3c, 0x18};
        for(uint8_t i=0;i<5;i++)
        {
                SPILCD_pixel(3,42+i);
                SPILCD_send_data(heartine[i]);
        }
}
int main(void)
{
        //lcd_init();
        //lcd_send_string("hi guys");
        SPILCD_init();
        SPILCD_demo();
    while(1)
    {
        //TODO:: Please write your application code
    }
}
```

2.3.3 Output

The Output can be seen @ the LCD display.

2.4 I2CLCD

2.4.1 Circuit



2.4.2 Code

```
#define F_CPU 16000000L


#include <avr/io.h>
#include <util/delay.h>


#include "I2CMASTERinclude.c"

#define SLAVE_ADDRESS 0x3E // obtained by reversing the I2C address in PROTESS I2C lcd address

void I2CLCD_send_cmd(uint8_t cmd_)
{
        // start condtion
        I2C_Master_START();
         _delay_ms(10);
        // addressing the slave
        I2C_Master_Mode(SLAVE_ADDRESS, 0);
         _delay_ms(10);
        // sending command mode
        I2C_Master_DataTransmitByte(0x00);
         _delay_ms(10);
        // Sending actual command
        I2C_Master_DataTransmitByte(cmd_);
         _delay_ms(10);
        I2C_Master_STOP();
    _delay_ms(10);
}
void I2CLCD_send_data(uint8_t data_)
{
        // start condtion
        I2C_Master_START();
         _delay_ms(10);
        // addressing the slave
        I2C_Master_Mode(SLAVE_ADDRESS, 0);
         _delay_ms(10);
        // sending data mode
        I2C_Master_DataTransmitByte(0x40);
         _delay_ms(10);
        // Sending actual data
        I2C_Master_DataTransmitByte(data_);
         _delay_ms(10);
        I2C_Master_STOP();
    _delay_ms(10);
}
void I2CLCD_init()
{
        I2C_Master_Init();
        I2CLCD_send_cmd(0x38);
        I2CLCD_send_cmd(0x06); // cursoR left moment and display not shift
        I2CLCD_send_cmd(0x0E); // dsiply on cursor on blink on
        I2CLCD_send_data(0x80); // BGIGIND OF libne
}
int main(void)
{
        I2CLCD_init();
        I2CLCD_send_data('a');

        I2CLCD_send_data('v');
    while(1)
    {
```

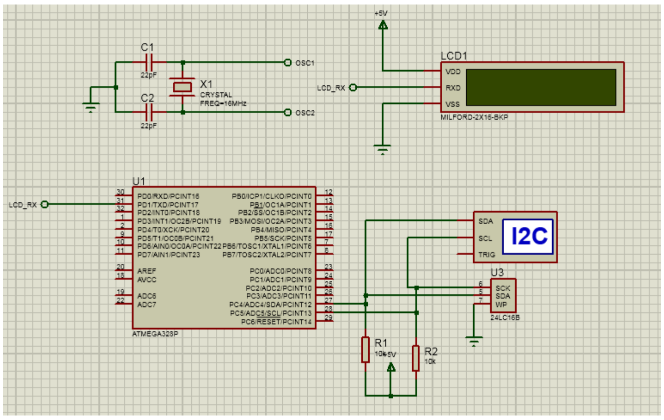
```
        //TODO:: Please write your application code
    }
}
```

2.4.3 Output

The Output can be seen @ the LCD display.

2.5 I2CEEPROM

2.5.1 Circuit



2.5.2 Code

```
#define F_CPU 16000000L

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#include "UARTLCDinclude.c"
#include "I2CMASTERinclude.c"

// This address is 7bit without any shifting for R/W bit as my implmentation of I2C takes care of it
#define SLAVE_ADDRESS 0b1010000

void EEPROM_writeByte(uint8_t data_byte, uint8_t block_addres, uint8_t word_address)
{
        // start condition
        I2C_Master_START();

        // Enter Master Transmiter Mode -0
        I2C_Master_Mode(SLAVE_ADDRESS | block_addres ,0);

        // Accoding to datasheet of EEPROM, the word address iss ent
        I2C_Master_DataTransmitByte(word_address);

        // Next the actual data is sent
        I2C_Master_DataTransmitByte(data_byte);

        // Leave Master Tranmitter Mode -0 and stop condition
        I2C_Master_STOP();
}
void EEPROM_writeString(uint8_t *my_str, uint8_t block_address, uint8_t word_address)
{
```

```c
        uint8_t string_count=0;
        // start condition
        I2C_Master_START();

        // Enter Master Transmiter Mode -0
        I2C_Master_Mode(SLAVE_ADDRESS | block_address ,0);

        // Accoding to datasheet of EEPROM, the word address iss ent
        I2C_Master_DataTransmitByte(word_address);

        while(*my_str != '\0')
        {
                string_count++;
                if (string_count==17)
                {
                        // if more than sixteen we change the page

                        string_count = 0;
                        I2C_Master_STOP();

                        I2C_Master_START();

                        I2C_Master_Mode(SLAVE_ADDRESS | block_address ,0);

                        I2C_Master_DataTransmitByte(word_address + 16);

                }
                I2C_Master_DataTransmitByte(*my_str++);
        }
        // Leave Master Tranmitter Mode -0 and stop condition
        I2C_Master_STOP();

}
uint8_t EEPROM_readByte(uint8_t block_address, uint8_t word_address)
{
        // start condition
        I2C_Master_START();

        // For reading first word address needs to be sent  so we enter master transmitter
        // Enter Master Transmiter Mode -0
        I2C_Master_Mode(SLAVE_ADDRESS | block_address,0);

        I2C_Master_DataTransmitByte(word_address);
        I2C_Master_STOP();

        //Next, we can read
        I2C_Master_START();
        // Enter Master Receiver Mode -0
        I2C_Master_Mode(SLAVE_ADDRESS | block_address,1);
        uint8_t value_ = I2C_Master_DataReceiveByte();
        I2C_Master_STOP();

        return value_;
}
void EEPROM_read_string(uint8_t *redval,uint8_t lenght_,uint8_t block_address, uint8_t word_address)
{
        uint8_t i;
        for (i=0;i<lenght_;i++)
        {
                 redval[i] = EEPROM_readByte(block_address,word_address + i);
        }
        redval[i] = '\0';
}
```

```c
int main(void)
{
	UARTLCD_init();
	I2C_Master_Init();

	// The device addressing is 101_0(B2,B1,B0)
	// B[2:0] -- Block Select bits
	// Since the memory size of 24LC16B is as eight 256x8-bit memory
	// Each block has 256Bytes of data
	// So total of 8block x 256 Bytes = 2KByte

	EEPROM_writeByte('A', 0, 0);
	UARTLCD_set_cursor(1,1);
	_delay_ms(10);
	char a=EEPROM_readByte(0,0);
	UARTLCD_send_data(a);

	EEPROM_writeString((uint8_t *)"abcdefghijklmnopqrstuvwxyz",0,0);
	_delay_ms(10);
	UARTLCD_set_cursor(2,1);
	uint8_t redval[50];
	EEPROM_read_string(redval,18,0,0);

	UARTLCD_send_string((char *)redval);
	while(1)
	{
		//TODO:: Please write your application code
	}
}
```

2.5.3 Output

The Output can be seen @ the LCD display and EEPROM memory.

APPENDIX

Basic Setup

- The programs are compiled using ***CompileAndProgram*** script.
- The proteus setup is as follows:
  - CLKDIV8 - Unprogrammed
  - CKOUT - Unprogrammed
  - RSTDISB - Unprogrammed
  - WDTON - Unprogrammed
  - BOOTRST - Unprogrammed
  - CKSEL Fuses - (0110) - External Full-swing Crystall
  - Boot Loader Size - 00
  - SUT Fuses - 10
  - Clock frequency - 160000000