# AVR Interfacing

# Timer

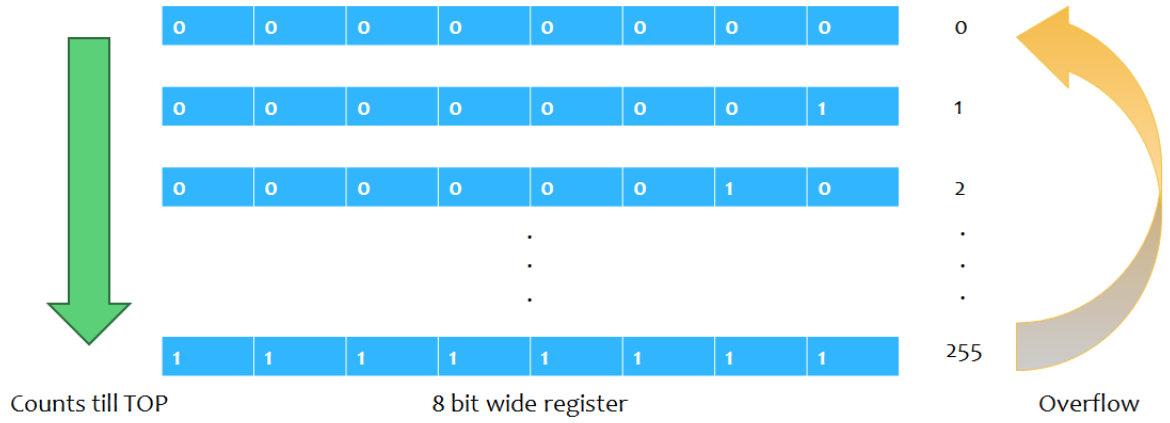
# Agenda

- Introduction to AVR Timer.
- AVR Overflow mode
- AVR Compare match mode (CTC)
- AVR Pulse width modulation mode (PWM).

# Timer Introduction

Timers is an important concept in the field of electronics. It can be generated a time base using a timer circuit, using a microcontroller, etc.

- In Microcontroller, timer is a register, but not a normal one.
- The value of this register increases/decreases automatically.
- In AVR, timers are of two types: 8-bit and 16-bit timers.
  - In an 8-bit timer, the register used is 8-bit wide whereas in 16-bit timer, the register width is of 16 bits.
  - The 8-bit timer is capable of counting 2^8=256 steps from 0 to 255 as demonstrated below. A 16 bit timer is capable of counting 2^16=65536 steps from 0 to 65535.



# Timer Introduction

- Timer can be used as
  - Precise timer
  - Counter
  - PWM (Pulse width modulation)
  - ICU (Input Capture unit)

- The best part is that the timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention, which makes the timer quite accurate.

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

- If Microcontroller works at frequency=4MHz, hence the timer will take 1/freq= 1/4M =0.00025 ms for the one count that called "system tick".
- To calculate the number of counts needed for a specific delay the following formula is used:

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

# Timer Introduction

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

- If required delay needs more counts than the max counts in the timer, prescaler (frequency division) can be used.
- For example:

Required Delay = 184 ms
F_CPU = 4 MHz

- $t_{sys\_tick} = \frac{1}{clock\ freq} = \frac{prescaler}{F\_cpu}$

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

- In 8 bit timer max counts=255 and in 16 bit timer max counts=65535. Therefore, To obtain the required delay, we can use prescaler 64, 256 or 1024. However, more the prescaler is smaller, more timing resolution (system tick time) is faster which is better.

# AVR Overflow mode

## Problem Statement

Let's define a problem statement for us. The simplest one being the LED flasher. Let's say, we need to flash an LED every 6 ms and we are have a CPU clock frequency of 32 kHz.

**Methodology**

Now, as per the following formula, with a clock frequency of 32 kHz and 8-bit counter, the maximum delay possible is of 8 ms. This is quite low. Hence for a delay of 6 ms, we need a timer count of 191. This can easily be achieved with an 8-bit counter (MAX = 255).

Thus, what we need to do is quite simple. We need to keep a track of the counter value. As soon as it reaches 191, we toggle the LED value and reset the counter.

# AVR Overflow mode

- In AVR, there are three types of timers Timer0, Timer1 and Timer2
- TIMER1 is a 16-bit timer whereas others are 8-bit timers.
- To configure the timer as we need. Knowing the registers details will the help.

## TCNT0 Register Timer/Counter Register

This is where the 8-bit counter of the timer resides. The value of the counter is stored here and it increases automatically each clock cycle. Data can be both read/written from this register. The initial value of the counter is set by writing it.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | TCNT0[7:0] | | | | | | | | TCNT0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

# AVR Overflow mode

TCCR0B Register – Timer/Counter Control Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x25 (0x45) | FOC0A | FOC0B | – | – | **WGM02** | **CS02** | **CS01** | **CS00** | TCCR0B |
| Read/Write | W | W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Right now, we will concentrate on the highlighted bits. The other bits will be discussed as and when necessary. By selecting these three **Clock Select Bits**, **CS02:00**, we set the timer up by choosing proper prescaler. The possible combinations are shown below.

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

# AVR Overflow mode

For the previous Problem statement  example: we need to flash an LED every 6 ms and we are have a CPU clock frequency of 32 kHz.

```
#include <avr/io.h>
void timer0_init(){
    // set up timer with no prescaling
     TCCR0B |= (1 << CS00);
    // initialize counter
    TCNT0 = 0;}
void main(void){
    // connect led to pin PC0
    DDRC |= (1 << 0);
    // initialize timer
    timer0_init();
    while(1){
        // check if the timer count reaches 191
        if (TCNT0 >= 191){
            PORTC ^= (1 << 0);     // toggles the led
            TCNT0 = 0;             // reset counter
        }
    }
}
```

# AVR Overflow mode

Problem Statement Redefined Again

Let's flash an LED every 50 ms with CPU frequency 16 MHz using Interrupt.

- The concept here is that the hardware generates an interrupt every time the timer overflows. Since the required delay is greater than the maximum possible delay, obviously the timer will overflow. And whenever the timer overflows, an interrupt is fired. Now the question is *how many times should the interrupt be fired?*

- For this, let's do some calculation. it should take 4.096 ms for the timer to overflow. Now as soon as the timer overflows, an interrupt is fired and an Interrupt Service Routine (ISR) is executed. Now,

$$50 \text{ ms} \div 4.096 \text{ ms} = 12.207$$

# AVR Overflow mode

## Problem Statement Redefined Again

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

```
// global variable to count the number of overflows
volatile uint8_t tot_overflow;
// TIMER0 overflow interrupt service routine
// called whenever TCNT0 overflows
ISR(TIMER0_OVF_vect)
{
    // keep a track of number of overflows
    tot_overflow++;
}
```

# AVR Overflow mode

```
void main(void)
{
    // connect led to pin PC0
    DDRC |= (1 << 0);
    // initialize timer
    timer0_init();
    // loop forever
    while(1){
        // check if no. of overflows = 12
        if (tot_overflow >= 12)  // NOTE: '>=' is used
        {
            // check if the timer count reaches 53
            if (TCNT0 >= 53){
                PORTC ^= (1 << 0);     // toggles the led
                TCNT0 = 0;             // reset counter
                tot_overflow = 0;      // reset overflow counter
            }
        }
    }
}
```

# AVR Overflow mode

- How to enable the interrupt feature. For this, you should be aware of the following registers.

- TIMSK0 Register – Timer/Counter Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6E) | – | – | – | – | – | OCIE0B | OCIE0A | TOIE0 | TIMSK0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

It is a common register for all the three timers. Setting **TOIE0 (**Timer/Counter0 Overflow Interrupt Enable) bit to ‘1’ enables the TIMER0 overflow interrupt.

- TIFR Register (Flag register) – Timer/Counter 0 Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x15 (0x35) | – | – | – | – | – | OCF0B | OCF0A | TOV0 | TIFR0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The bit TOV0 (Timer/Counter0 Overflow Flag) is set (one) when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector.

# AVR Overflow mode

- Enabling Global Interrupts

In the AVRs, there's only one single bit which handles all the interrupts. Thus, to enable it, we need to enable the global interrupts. This is done by calling a function named sei(). According to the previous regs the init function will be as follows:

```
#include <avr/io.h>
#include <avr/interrupt.h>
// initialize timer, interrupt and variable
void timer0_init(){
    // set up timer with prescaler = 256
    TCCR0B |= (1 << CS02);
    // initialize counter
    TCNT0 = 0;
    // enable overflow interrupt
    TIMSK0 |= (1 << TOIE0);
    // enable global interrupts
    sei();
    // initialize overflow counter variable
    tot_overflow = 0; }
```

# AVR Compare match mode (CTC)

AVR timers can operate not only in overflow mode but there are different modes it can be operated in as follows:

- Normal mode (Overflow).
- Clear Timer on Compare mode (CTC) or Compare Match mode.
- Pulse Width Modulation mode (PWM).

## Clear Timer on Compare (CTC) Mode:

Suppose that We had two timer values with us – Set Point (SP) and Process Value (PV). In every iteration, we used to compare the process value with the set point. Once the process value becomes equal (or exceeds) the set point, the process value is reset.

# AVR Compare match mode (CTC)

## Problem Statement

We need to flash an LED every 100 ms with CPU frequency 16 MHz.

*Methodology – Using CTC Mode*

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

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

# AVR Compare match mode (CTC)

Now, let me introduce you to the register bits which help you to implement this CTC Mode.

- TCCR1A and TCCR1B Registers (Timer/Counter1 Control Register)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x80) | COM1A1 | COM1A0 | COM1B1 | COM1B0 | – | – | WGM11 | WGM10 | TCCR1A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x81) | ICNC1 | ICES1 | – | WGM13 | WGM12 | CS12 | CS11 | CS10 | TCCR1B |
| Read/Write | R/W | R/W | R | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- We are already aware of the Clock Select Bits – CS12:10 which set the prescaler.
- Right now, we are concerned with the **Wave Generation Mode Bits – WGM13:10**. which configures the operating mode of the timer. We have discussed before normal mode (overflow mode)

# AVR Compare match mode (CTC)

- The following table shows how to configure the different modes by configure the values of **WGM13:0**

| Mode | WGM13 | WGM12 (CTC1) | WGM11 (PWM11) | WGM10 (PWM10) | Timer/Counter Mode of Operation | TOP | Update of OCR1x | TOV1 Flag Set on |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | Normal | 0xFFFF | Immediate | MAX |
| 1 | 0 | 0 | 0 | 1 | PWM, Phase Correct, 8-bit | 0x00FF | TOP | BOTTOM |
| 2 | 0 | 0 | 1 | 0 | PWM, Phase Correct, 9-bit | 0x01FF | TOP | BOTTOM |
| 3 | 0 | 0 | 1 | 1 | PWM, Phase Correct, 10-bit | 0x03FF | TOP | BOTTOM |
| 4 | 0 | 1 | 0 | 0 | CTC | OCR1A | Immediate | MAX |
| 5 | 0 | 1 | 0 | 1 | Fast PWM, 8-bit | 0x00FF | TOP | TOP |
| 6 | 0 | 1 | 1 | 0 | Fast PWM, 9-bit | 0x01FF | TOP | TOP |
| 7 | 0 | 1 | 1 | 1 | Fast PWM, 10-bit | 0x03FF | TOP | TOP |
| 8 | 1 | 0 | 0 | 0 | PWM, Phase and Frequency Correct | ICR1 | BOTTOM | BOTTOM |
| 9 | 1 | 0 | 0 | 1 | PWM, Phase and Frequency Correct | OCR1A | BOTTOM | BOTTOM |
| 10 | 1 | 0 | 1 | 0 | PWM, Phase Correct | ICR1 | TOP | BOTTOM |
| 11 | 1 | 0 | 1 | 1 | PWM, Phase Correct | OCR1A | TOP | BOTTOM |
| 12 | 1 | 1 | 0 | 0 | CTC | ICR1 | Immediate | MAX |
| 13 | 1 | 1 | 0 | 1 | Reserved | – | – | – |
| 14 | 1 | 1 | 1 | 0 | Fast PWM | ICR1 | TOP | TOP |
| 15 | 1 | 1 | 1 | 1 | Fast PWM | OCR1A | TOP | TOP |

- We have discussed before normal mode (overflow mode) which requires WGM13:0=0000. By default, these bit are zero unless it is not configured.

# AVR Compare match mode (CTC)

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

```
//Mode CTC and prescaler= 64
TCCR1A |=0;
TCCR1B |=(1<< WGM12)
         | (1<< CS00)
         | (1<< CS01);
```

| Mode | WGM13 | WGM12 (CTC1) | WGM11 (PWM11) | WGM10 (PWM10) | Timer/Counter Mode of Operation | TOP | Update of OCR1x | TOV1 Flag Set on |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | Normal | 0xFFFF | Immediate | MAX |
| 1 | 0 | 0 | 0 | 1 | PWM, Phase Correct, 8-bit | 0x00FF | TOP | BOTTOM |
| 2 | 0 | 0 | 1 | 0 | PWM, Phase Correct, 9-bit | 0x01FF | TOP | BOTTOM |
| 3 | 0 | 0 | 1 | 1 | PWM, Phase Correct, 10-bit | 0x03FF | TOP | BOTTOM |
| 4 | 0 | 1 | 0 | 0 | CTC | OCR1A | Immediate | MAX |
| 5 | 0 | 1 | 0 | 1 | Fast PWM, 8-bit | 0x00FF | TOP | TOP |
| 6 | 0 | 1 | 1 | 0 | Fast PWM, 9-bit | 0x01FF | TOP | TOP |
| 7 | 0 | 1 | 1 | 1 | Fast PWM, 10-bit | 0x03FF | TOP | TOP |
| 8 | 1 | 0 | 0 | 0 | PWM, Phase and Frequency Correct | ICR1 | BOTTOM | BOTTOM |
| 9 | 1 | 0 | 0 | 1 | PWM, Phase and Frequency Correct | OCR1A | BOTTOM | BOTTOM |
| 10 | 1 | 0 | 1 | 0 | PWM, Phase Correct | ICR1 | TOP | BOTTOM |
| 11 | 1 | 0 | 1 | 1 | PWM, Phase Correct | OCR1A | TOP | BOTTOM |
| 12 | 1 | 1 | 0 | 0 | CTC | ICR1 | Immediate | MAX |
| 13 | 1 | 1 | 0 | 1 | Reserved | – | – | – |
| 14 | 1 | 1 | 1 | 0 | Fast PWM | ICR1 | TOP | TOP |
| 15 | 1 | 1 | 1 | 1 | Fast PWM | OCR1A | TOP | TOP |

# AVR Compare match mode (CTC)

- OCR1A and OCR1B Registers Output Compare Register

OCR register is a register which we store the compared value with in it.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | OCR1A[15:8] | | | | | | | | OCR1AH |
| | OCR1A[7:0] | | | | | | | | OCR1AL |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | OCR1B[15:8] | | | | | | | | OCR1BH |
| | OCR1B[7:0] | | | | | | | | OCR1BL |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

# AVR Compare match mode (CTC)

- TIFR1 Register (Timer/Counter1 Interrupt Flag Register)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x16 (0x36) | – | – | ICF1 | – | – | OCF1B | OCF1A | TOV1 | TIFR1 |
| Read/Write | R | R | R/W | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

We are interested in **Bit 2:1 – OCF1A:B – Timer/Counter1, Output Compare A/B Match Flag Bit**. This bit is set (one) by the AVR whenever a match occurs i.e. TCNT1 becomes equal to OCR1A (or OCR1B). It is cleared automatically whenever the corresponding Interrupt Service Routine (ISR) is executed. Alternatively, it can be cleared by writing '1' to it!

❖ Now return back to our problem statement: We need to flash an LED every 100 ms with CPU frequency 16 MHz using timer1 in AVR.

```
#include <avr/io.h>
// initialize timer, interrupt and variable
void timer1_init(){
    // set up timer with prescaler = 64 and CTC mode
    TCCR1B |= (1 << WGM12)|(1 << CS01)|(1 << CS00);
    // initialize counter
    TCNT1 = 0;
    // initialize compare value
    OCR1A = 24999;
}
```

# AVR Compare match mode (CTC)

```
void main(void)
{
    // connect led to pin PC0
    DDRC |= (1 << 0);
    // initialize timer
    timer1_init();
    while(1)
    {
        // check whether the flag bit is set
        // if set, it means that there has been a compare match
        // and the timer has been cleared
        // use this opportunity to toggle the led
        if (TIFR1 & (1 << OCF1A)) // NOTE: '>=' used instead of '=='
        {
            PORTC ^= (1 << 0); // toggles the led
        }
        // wait! we are not done yet!
        // clear the flag bit manually since there is no ISR to execute
        // clear it by writing '1' to it (as per the datasheet)
        TIFR1 |= (1 << OCF1A);
    }
}
```

# AVR Compare match mode (CTC)

## Methodology – Using Interrupts with CTC Mode

- In the previous methodology, we simply used the CTC Mode of operation. We used to check every time for the flag bit (OCF1A). Now let's shift this responsibility to the AVR itself. now we *do not need to check* for the flag bit at all! The AVR will compare TCNT1 with OCR1A. Whenever a match occurs, it sets the flag bit OCF1A, and *also* fires an interrupt.
- There are three kinds of interrupts in AVR – *overflow, compare* and *capture*. We have already discussed the *overflow* interrupt. For this case, we need to enable the *compare* interrupt. The following register is used to enable interrupts.
  - TIMSK1 Register – Timer/Counter1 Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6F) | – | – | ICIE1 | – | – | OCIE1B | OCIE1A | TOIE1 | TIMSK1 |
| Read/Write | R | R | R/W | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

We have already come across TOIE1 bit. Now, the **Bit 2:1 – OCIE1A:B – Timer/Counter1, Output Compare A/B Match Interrupt Enable** bits are of our interest here. Enabling it ensures that an interrupt is fired whenever a match occurs. Since there are two CTC channels (unit) in Timer1, we have two different bits OCIE1A and OCIE1B for them.

# AVR Compare match mode (CTC)

- Now let's proceed to write an ISR for this. The ISR is defined as follows:
  - Note: Executing the ISR clears the OCF1A flag bit automatically and the timer value (TCNT1) is reset.

```
ISR (TIMER1_COMPA_vect)
{
   // toggle led here
   PORTB ^=(1<<0);
}
```

- Init function will be as follows:

```
#include <avr/io.h>
#include <avr/interrupt.h>
// initialize timer, interrupt and variable
void timer1_init(){
    // set up timer with prescaler = 64 and CTC mode
    TCCR1B |= (1 << WGM12)|(1 << CS01)|(1 << CS00);
    // initialize counter
    TCNT1 = 0;
    // initialize compare value
    OCR1A = 24999;
    // enable compare interrupt
    TIMSK1 |= (1 << OCIE1A);
    // enable global interrupts
    sei();
}
```

# AVR Compare match mode (CTC)

```
void main(void)
{
    // connect led to pin PC0
    DDRC |= (1 << 0);
    // initialize timer
    timer1_init();
    // loop forever
    while(1)
    {
        // do nothing
        // whenever a match occurs, ISR is fired
        // toggle the led every 100ms in the ISR itself
        // no need to keep track of any flag bits here
        // done!
    }
}
```

- Now, we have seen how to implement the CTC mode using interrupts, reducing the code size, comparisons and processing time.

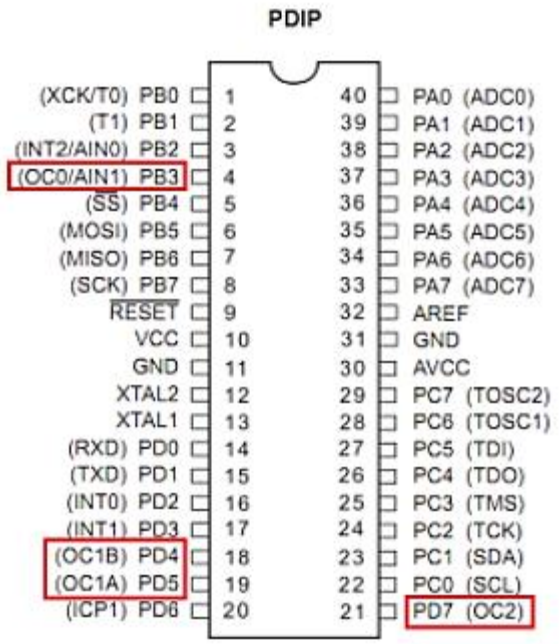
# AVR Compare match mode (CTC)

## Methodology – Using Hardware CTC Mode

- The pins PD6, PD5, PB1, PB2, PB3 and PD3 have their special functions are mentioned in the brackets (OC0A, OC0B, OC1A, OC1B , OC2A and OC2A). These are the Output Compare pins of TIMER0, TIMER1 and TIMER2

- **TCCR1A Register Timer/Counter1 Control Register A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x80) | COM1A1 | COM1A0 | COM1B1 | COM1B0 | – | – | WGM11 | WGM10 | TCCR1A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |




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

The behavior changes depending upon the following modes:

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

# AVR Compare match mode (CTC)

- The following options hold good for non-PWM mode

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

Compare Output Mode, non-PWM

- Since we need to toggle the LED, we choose the second option (01). Well, that's all we need to do! No need to check any flag bit, no need to attend to any interrupts, nothing. Just set the timer to this mode and we are done! Whenever a compare match occurs, the OC1A pin is automatically toggled.

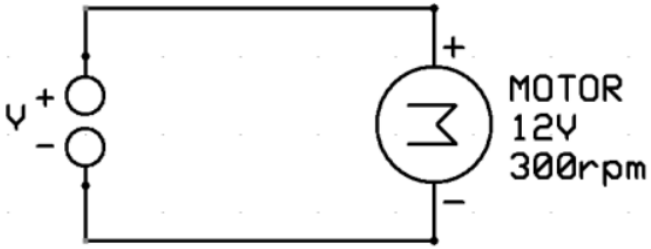
- But we need to compromise on the hardware. Only PB1 or PB2 (OC1A or OC1B) can be controlled this way, which means that we should connect the LED to PB1 (since we are using channel A) instead of PC0 (which we had been using in all the examples till now).

# AVR Compare match mode (CTC)

```
#include <avr/io.h>
#include <avr/interrupt.h>
// initialize timer, interrupt and variable
void timer1_init()
{
    // set up timer with prescaler = 64 and CTC mode
    TCCR1B |= (1 << WGM12)|(1 << CS01)|(1 << CS00);
    // set up timer OC1A pin in toggle mode
    TCCR1A |= (1 << COM1A0);
    // initialize counter
    TCNT1 = 0;
    // initialize compare value
    OCR1A = 24999;
}

void main(void){
    // connect led to pin PB1
    DDRB |= (1 << 1);
    // initialize timer
    timer1_init();
    while(1){
        // do nothing
        // whenever a match occurs
        // OC1A is toggled automatically!
        // no need to keep track of any flag bits or ISR
    }
}
```

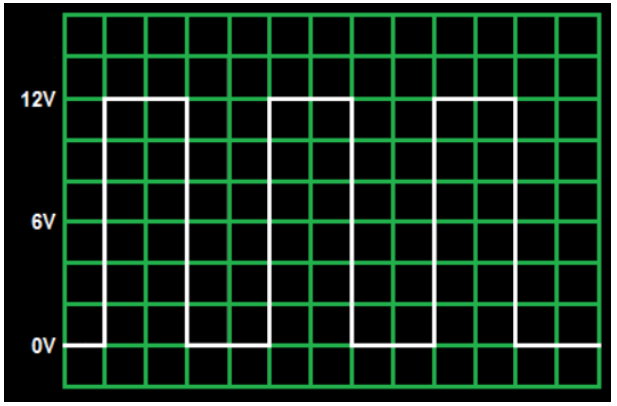
# AVR Pulse width modulation mode (PWM)

- Pulse Width Modulation (PWM) is a very common technique in telecommunication and power control. It is basically a modulation technique, in which the width of the carrier pulse is varied in accordance with the analog message signal.
- Let us assume that a DC motor is connected to the power supply as follows

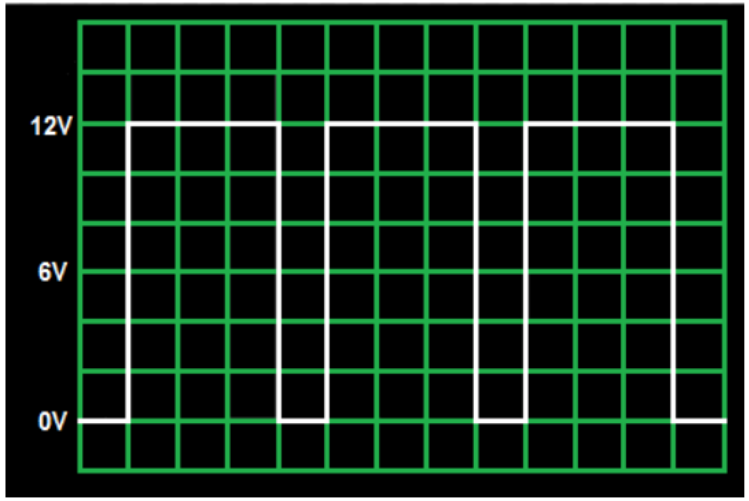



- The motor is rated 12V/300rpm. This means that (assuming ideal conditions) the motor will run at 300 rpm only when 12V DC is supplied to it. If we apply 6V, the motor will run at only 150 rpm.
- Let us ask what will happen if we apply a periodic signal as shown?

  It takes some time to reach its full speed. But before it happens, it is powered off, and so on. Thus, the overall effect of this action is that the motor rotates continuously, but at a lower speed. In this case, the motor will behave exactly as if a 6V DC is supplied to it, i.e. rotate at 150 rpm.



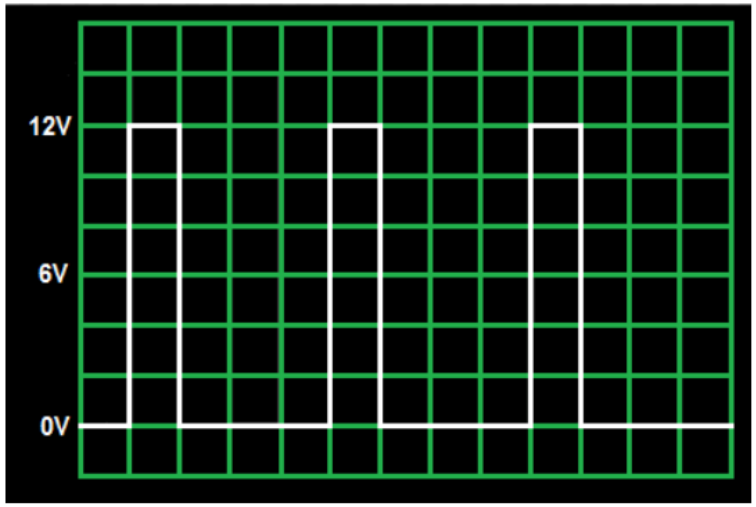
# AVR Pulse width modulation mode (PWM)




75% Duty Cycle PWM

Since the on-time is more than the off-time, the effective speed of the motor increased. In this case, the speed becomes 225 rpm (since on-time = ¾ off-time,

i.e. $speed = 300 * \frac{3}{4} = 225\ rpm$).




25% Duty Cycle PWM

Since the on-time is less than the off-time, the effective speed of the motor reduced. In this case, the speed becomes 75 rpm (since on-time = $\frac{1}{4}$ off-time,

i.e. $speed = 300 * \frac{1}{4} = 75\ rpm$).

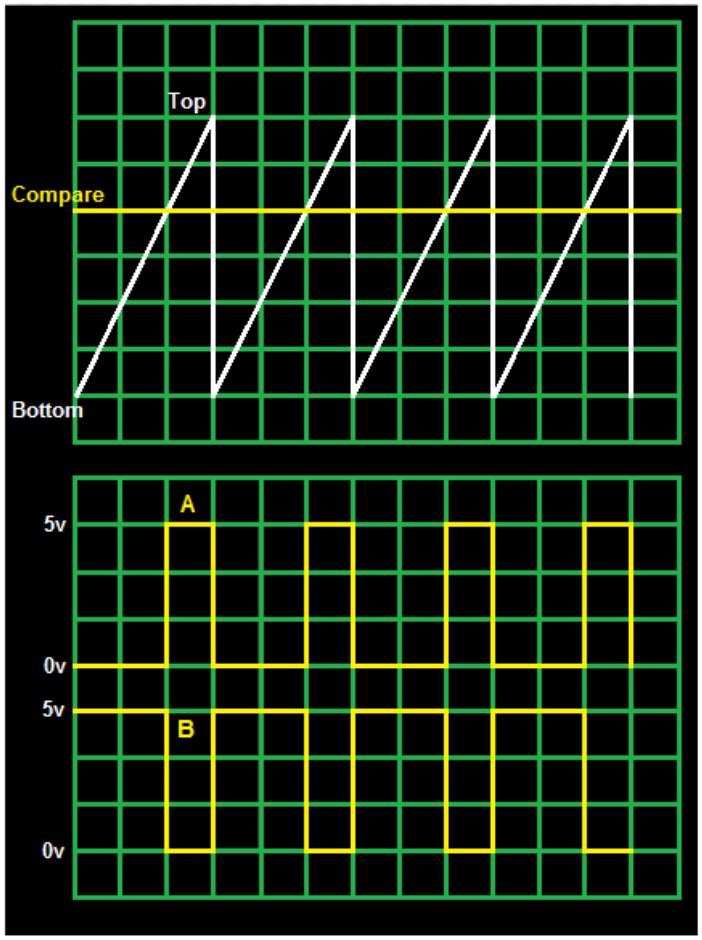
This is what we call **Pulse Width Modulation**, commonly known as **PWM**.

# AVR Pulse width modulation mode (PWM)

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

In the Figure shown, we have a predetermined waveform, sawtooth waveform. We compare this waveform with a fixed DC level. It has three compare output modes of operation:

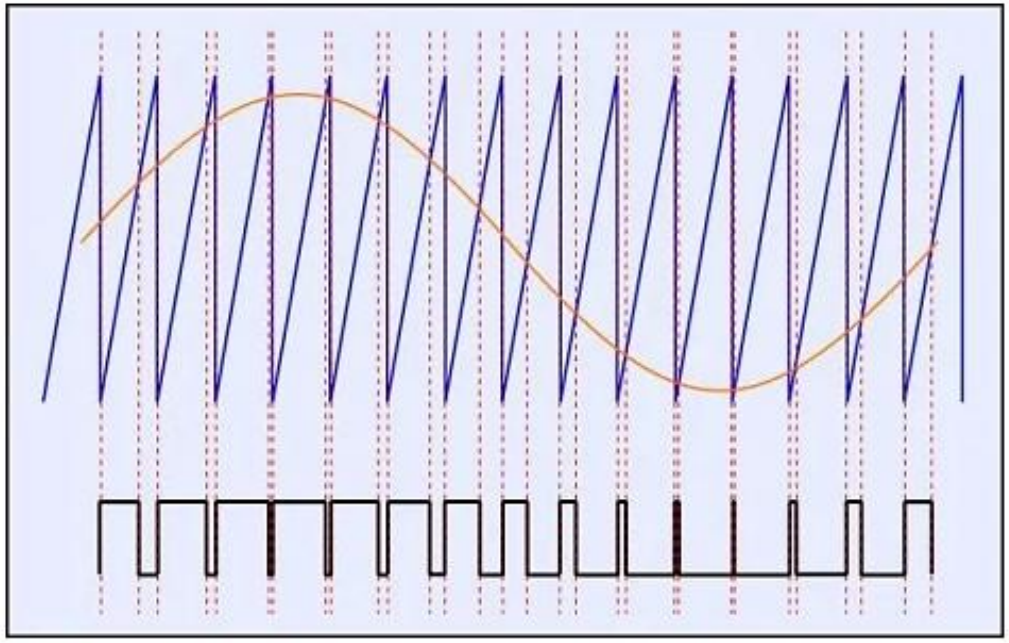
- **Inverted Mode**: In this mode, if the waveform value is greater than the compare level, then the output is set high, or else the output is low. This is represented in figure A.
- **Non-Inverted Mode**: In this mode, the output is high whenever the compare level is greater than the waveform level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



Compare PWM

# AVR Pulse width modulation mode (PWM)

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



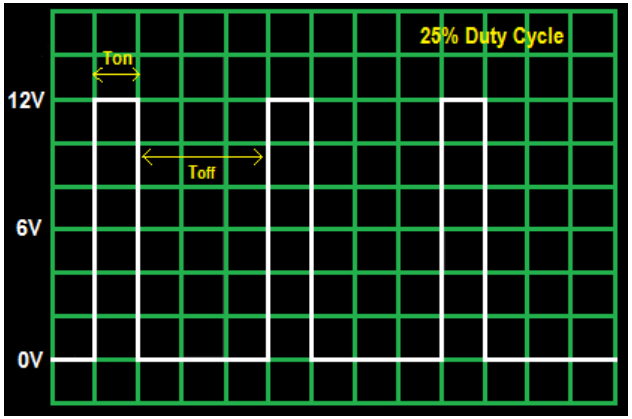
PWM Modulation

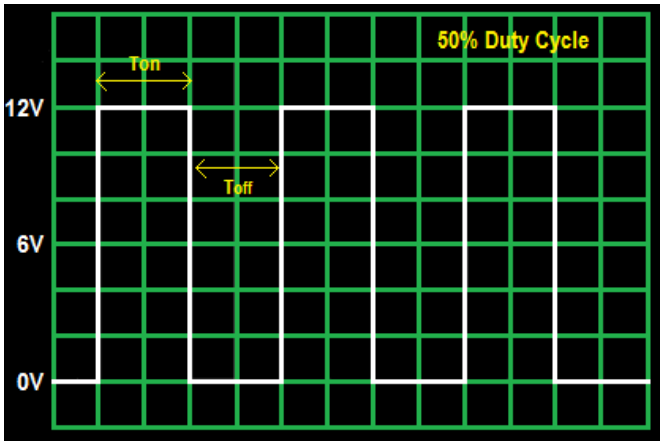
# AVR Pulse width modulation mode (PWM)

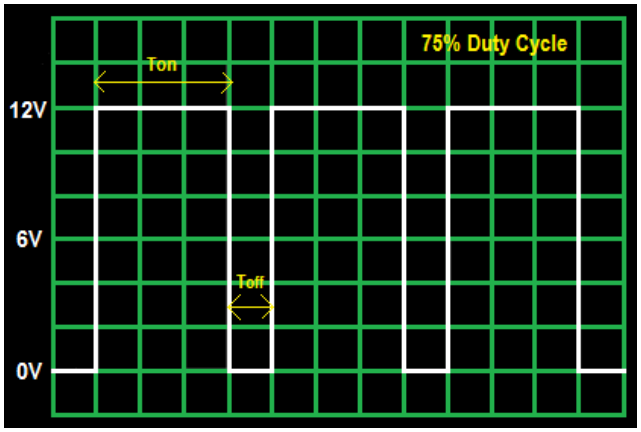
## Duty Cycle

The Duty Cycle of a PWM Waveform is given by

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

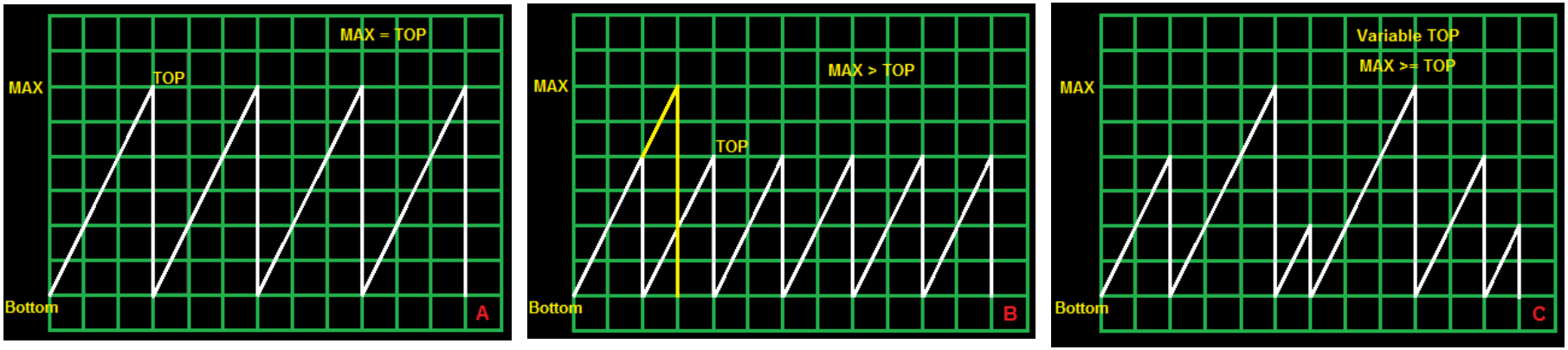


# AVR Pulse width modulation mode (PWM)

Consider the following timer diagram:




- We are very well aware that the AVR provides us with an option of 8 and 16 bit timers. 8bit timers count from 0 to 255, then back to zero and so on. 16bit timers count from 0 to 65535, then back to zero. Thus for a 8bit timer, MAX = 255 and for a 16bit timer, MAX = 65535.
- The timer *always* counts from 0 to TOP, then overflows back to zero. In figure A shown above, TOP = MAX.
- Due to this, the value of TOP can be reduced as shown in figure B. The yellow line shows how the timer would have gone in normal mode.
- Now, the CTC Mode can be extended to introduce variable TOP as shown in figure C

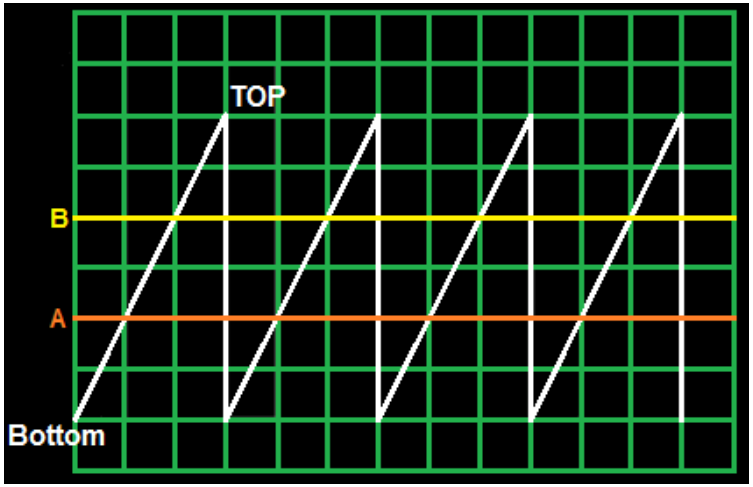
# AVR Pulse width modulation mode (PWM)

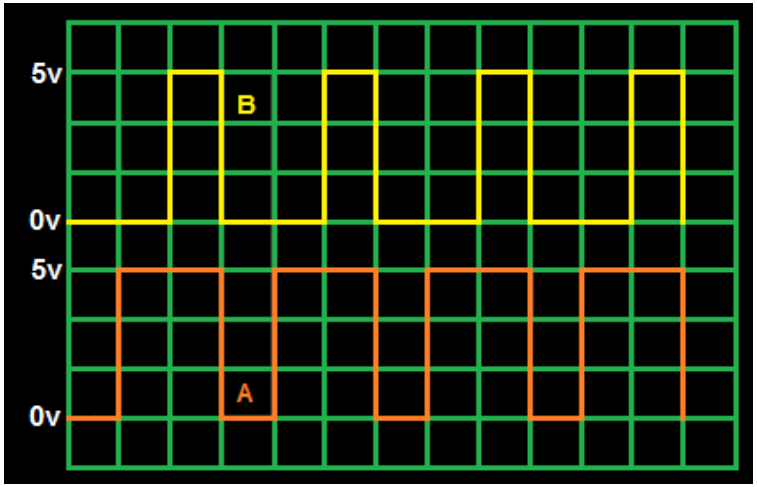
***PWM Modes of Operation:***

There are three modes of operation of PWM Timers:

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

*Fast PWM*

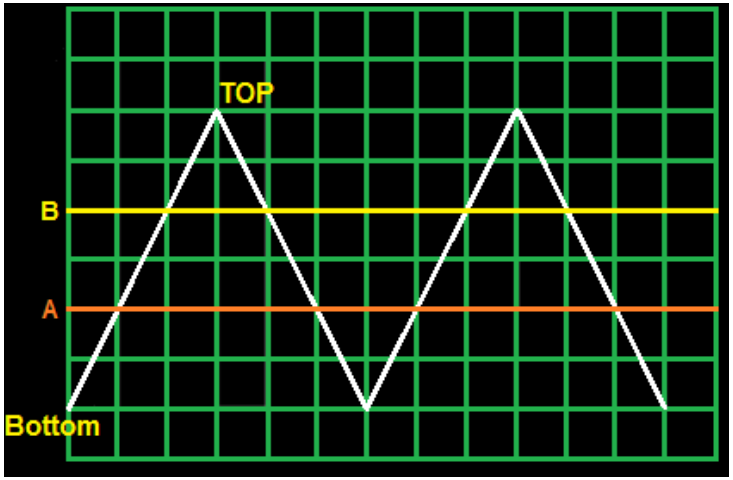


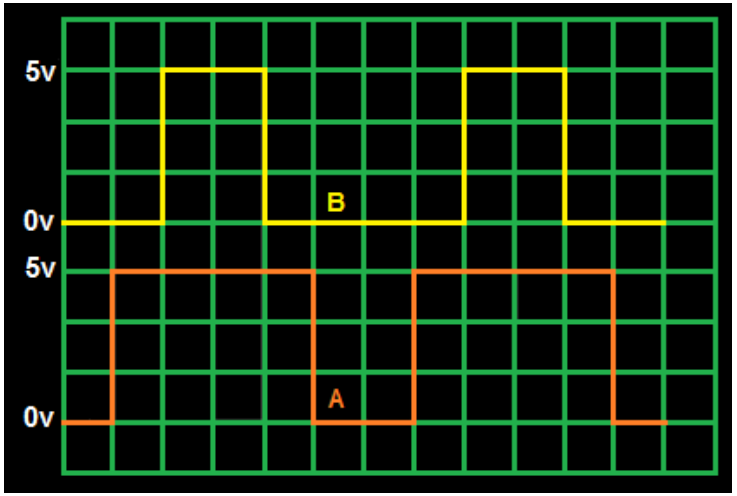




- We have a sawtooth waveform, and we compare it with a fixed voltage level (say A), and thus we get a PWM output as shown (in A). Now suppose we increase the compare voltage level (to, say B). In this case, as we can see, the pulse width has reduced, and hence the duty cycle. *But,* as you can see, both the pulses (A and B) end at the same time irrespective of their starting time.

# AVR Pulse width modulation mode (PWM)

*Phase Correct PWM*




- Here instead of a sawtooth waveform, we have used a triangular waveform. Even here, you can see how PWM is generated. We can see that upon increasing the compare voltage level, the duty cycle reduces. But unlike Fast PWM, in Phase correct PWM, both the pulses (A and B) end and start in different time.

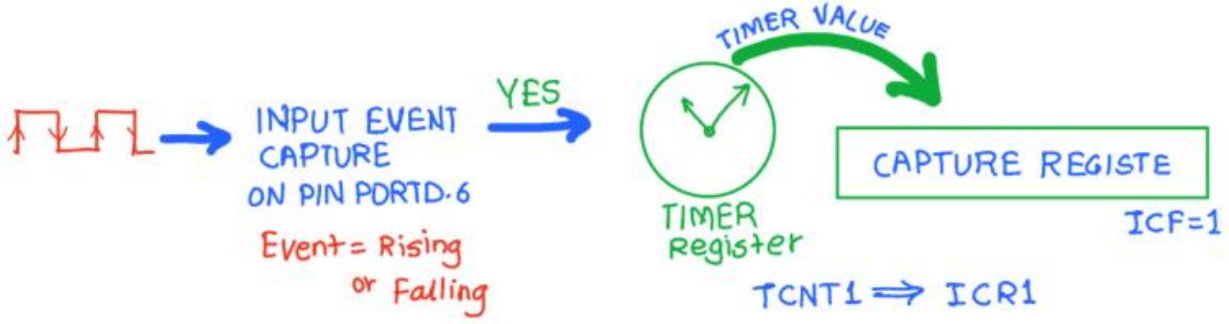
- By visual inspection, we can clearly see that the frequency of Fast PWM is twice that of Phase Correct PWM.

# AVR Timer Input Capture Mode (ICU)

An Input Capture unit is a peripheral that is able to listen from an input pin in MCU, if any transition change (HIGH to LOW or vice versa ) happens on this pin, a flag will be raised and time will be captured and saved into a register.

- Input capture function is used in many applications such as:
  - Pulse width measurement
  - Period measurement
  - Capturing the time of an event

In AVR ATmega32, Timer1 can be used as an input capture to detect and measure events happening outside the microcontroller. Upon detection of a defined event i.e. rising edge or falling edge on ICP1 pin (PORTB.0), TCNT1(Timer / Counter register) value is loaded into the ICR1 (input capture) register and the ICF1 (in TIFR1) flag will get set.



# AVR Timer Input Capture Mode (ICU)

- Programming:
  - **TCCR1B:** Timer Counter Control Register B

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ICNC1 | ICES1 | – | WGM13 | WGM12 | CS12 | CS11 | CS10 |

**Bit 7 - ICNC1:** Input Capture Noise canceller
Setting this bit activates noise canceller. It causes a delay of 4 clock cycles as it considers a change only if it persists for at least 4 successive system clocks.

**Bit 6 - ICES1:** Input Capture Edge select
Select edge detection for input capture function.
**0 =** Capture on falling edge
**1 =** Capture on rising edge

**Bit 4: 3 - WGM13 : WGM12:** Timer1 Mode select
These bits are used for mode selection like Normal mode, PWM mode, CTC mode etc. here we will select normal mode, so set these bits to zero.

# AVR Timer Input Capture Mode (ICU)

- Programming:
  - **TCCR1B:** Timer Counter Control Register B

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ICNC1 | ICES1 | – | WGM13 | WGM12 | CS12 | CS11 | CS10 |

**Bit 2: 0 - CS12: CS10:**Timer1 Clock Select

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer / Counter stopped) |
| 0 | 0 | 1 | clk (no pre-scaling) |
| 0 | 1 | 0 | clk / 8 |
| 0 | 1 | 1 | clk / 64 |
| 1 | 0 | 0 | clk / 256 |
| 1 | 0 | 1 | clk / 1024 |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

# AVR Timer Input Capture Mode (ICU)

- Programming:
  - ➢ Steps to Program
    - ✓ Initialize the TCCR1A and TCCR1B for proper timer mode (any mode other than 8, 10, 12, 14), to select the edge (Positive or Negative).
    - ✓ Monitor the ICF1 flag in TIFR register to see if edge is arrived. Upon the arrival of the edge, the TCNT1 value is loaded in to ICR1 register automatically by controller.

**Note:** The Input Capture Pin (ICP1) on PB0 also functions as the Analog Comparator Output (ACO). To use PB0 for Input Capture, disable the Analog Comparator by setting PB0 HIGH (PORTB |= (1 << PB0)). To use it as ACO, enable the comparator (ACSR |= (1 << ACO)).

# AVR Timer Input Capture Mode (ICU)

- Example1:
  - Assuming that the clock pulses are fed into the pin ICP1, following program will read TCNT1 value at every rising edge and place the result on PORTA and PORTB.

```
#include "avr/io.h"
int main ( )
{
    unsigned int t;
    DDRA = 0xFF;
    DDRB = 0xFF;
    PORTD = 0xFF;
    TCCR1A = 0;
    TIFR = (1<<ICF1);        /* clear input capture flag */
    TCCR1B = 0x41;           /* capture on rising edge */

    while ((TIFR&(1<<ICF1)) == 0);  /* monitor for capture*/
    t = ICR1;
    TIFR = (1<<ICF1);        /* clear capture flag */

    while ((TIFR&(1<<ICF1)) == 0);  /* monitor for next rising
                edge capture */

    t = ICR1 - t;            /* period= recent capture-
                previous capture */
    PORTA = t;               /* put period count on PORTA & PORTB */
    PORTB = t>>8;

    while (1);
    return 0;
}
```

# AVR Timer Input Capture Mode (ICU)

- Example2:
  - We want to measure the frequency and duty cycle and displaying it on PORTA and PORTB.

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
int main ( )
{
    unsigned int a,b,c,high,period;
    DDRA = 0xFF;
    DDRB = 0xFF;
    PORTD = 0xFF;           /* Turn ON pull-up resistor */
    while(1)
    {
        TCCR1A = 0;
        TCNT1=0;
        TIFR = (1<<ICF1);   /* Clear ICF (Input Capture flag) flag */

        TCCR1B = 0x41;      /* Rising edge, no prescaler */
        while ((TIFR&(1<<ICF1)) == 0);  /* wait for Rising edge to be detected*/
        a = ICR1;       /* Take value of capture register */
        TIFR = (1<<ICF1);   /* Clear ICF flag */

        TCCR1B = 0x01;      /* Falling edge, no prescaler */
        while ((TIFR&(1<<ICF1)) == 0); /* wait for Falling edge to be detected*/
        b = ICR1;       /* Take value of capture register */
        TIFR = (1<<ICF1);   /* Clear ICF flag */

        TCCR1B = 0x41;      /* Rising edge, no prescaler */
        while ((TIFR&(1<<ICF1)) == 0); /* wait for Rising edge to be detected*/
        c = ICR1;       /* Take value of capture register */
        TIFR = (1<<ICF1);   /* Clear ICF flag */
```

# AVR Timer Input Capture Mode (ICU)

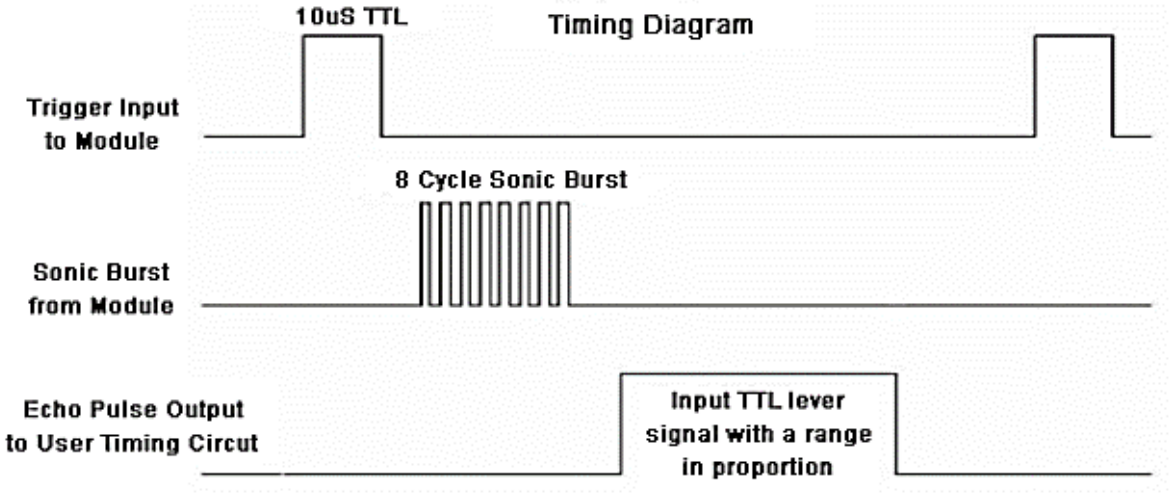
- Example2:
  - We want to measure the frequency and duty cycle and displaying it on PORTA and PORTB.

```
        TCCR1B = 0x41;      /* Rising edge, no prescaler */
        while ((TIFR&(1<<ICF1)) == 0); /* wait for Rising edge to be detected*/
        c = ICR1;       /* Take value of capture register */
        TIFR = (1<<ICF1);   /* Clear ICF flag */

        TCCR1B = 0;         /* Stop the timer */

        if(a<b && b<c)      /* Check for valid condition,
                    to avoid timer overflow reading */
        {
            high=b-a;
            period=c-a;
            long freq= F_CPU/period;/* Calculate frequency */
            /* Calculate duty cycle */
            float duty_cycle =((float) high /(float)period)*100;
            PORTA = freq;           /* put freq reading on PORTA & PORTB */
            PORTB = freq>>8;
        }

        else
        {
            PORTA = 0x00;           /* put period count on PORTA & PORTB */
            PORTB = 0x00;
        }
        _delay_ms(50);
    }

}
```

# AVR Timer Input Capture Mode (ICU)

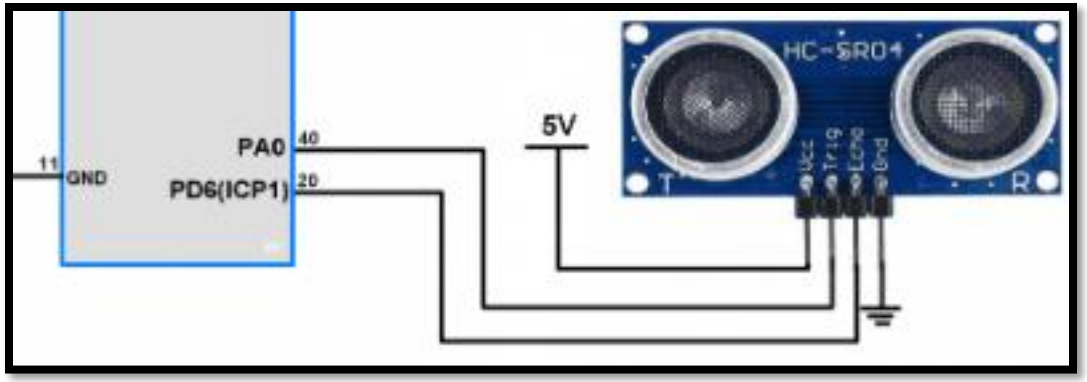
- Example3 (Unsolved):
  - We want to measure the distance using Ultrasonic Sensor.

  - Ultrasonic Sensor:
    - This sensor uses a technique called “ECHO” which is something you get when sound reflects back after striking with a surface.
    - **Ultrasonic sensor “HC-SR04”** provides an output signal proportional to distance based on the echo. The sensor here generates a sound vibration in ultrasonic range upon giving a trigger, after that it waits for the sound vibration to return. Now based on the parameters, sound speed (340m/s) and time taken for the echo to reach the source, it provides output pulse proportional to distance.



# AVR Timer Input Capture Mode (ICU)

- Example3 (Unsolved):
  - We want to measure the distance using Ultrasonic Sensor.
  - Ultrasonic Sensor Connection with MCU:
    It consists of 4 pins (VCC, Trig, Echo, GND)
    Trig pin: is connected with an output pin in MCU.
    Echo pin: is connected with ICP pin (PORTB0).

# AVR Timer Input Capture Mode (ICU)

- The idea:
  1- MUC has to send a pulse signal (>=10usec) to Trig pin by set the output pin to high for (>=10usec) then set it to low.
  2- Set capture in ICU configuration to rising edge.
  3- Clear flag of ICU (ICF flag).
  4- Wait until capture pin PORT0 (ICP) read high. for rising edge,
  5- Clear timer register (TCNT) and ICU flag (ICF flag).
  6- Set capture mode to falling edge.
  7- Wait until capture pin PORT0 (ICP) read low. for falling edge.
  8- Store the value in Timer counter register (OCR) in a variable e.x: (tcount).
  9- Calculate the distance by the following formula:

$$Distance = \frac{t_{count} * t_{Freq(tick)}}{2} * 340 = \cdots (m)$$