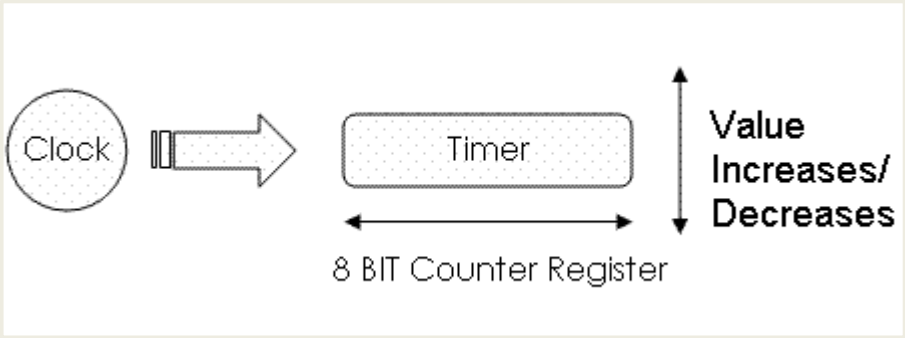
MICROCONTROLLER TUTORIAL II

TIMERS

WHAT IS A TIMER?

We use timers every day - the simplest one can be found on your wrist. A simple clock will time the seconds, minutes and hours elapsed in a given day - or in the case of a twelve hour clock, since the last half-day. AVR timers do a similar job, measuring a given time interval.

An AVR timer in simplest term is a register. Timers generally have a resolution of 8 or 16 bits. So an 8 bit timer is 8 bits wide, and is capable of holding value within 0-255. But this register has a magical property - its value increases/decreases automatically at a predefined rate (supplied by user). This is the timer clock. And this operation does not need CPU's intervention.



The AVR timers are very useful as they run asynchronous to the main AVR core. This is a fancy way of saying that the timers are separate circuits on the AVR chip which can run independent of the main program, interacting via the control and count registers, and something called timer interrupts.

WHAT IS AN INTERRUPT?

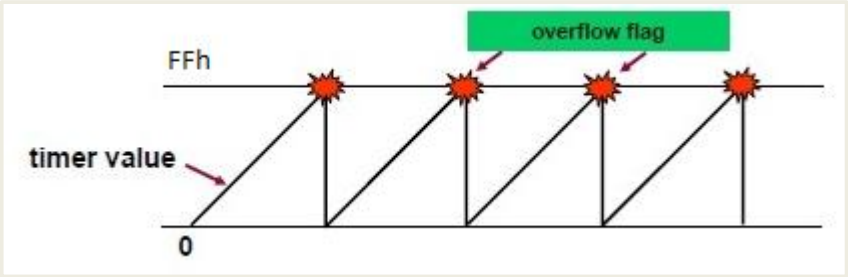
Prior to learning about timers you need to know the concept of interrupt because timers mainly interact with CPU through interrupts. The concept of an interrupt in reference to microcontrollers is similar to our daily life concept of interrupts: suppose you are reading this tutorial and suddenly your mobile rings - what you do is you stop reading for a while and attend the phone call, and then resume reading from where you left it. This is exactly what an interrupt does in MCU. While the MCU is executing a program, if there is something that needs immediate attention, an interrupt is generated by that task and the execution of the current program is left at that time and interrupt is handled. After that, the execution of program continues as usual from the point where it was stopped. The timers run parallel and independent of the CPU at a specific frequency, and interact with the CPU by issuing interrupts.

There are two types of interrupts:

- Overflow interrupt
- Compare match Interrupt

OVERFLOW INTERRUPT

Overflow interrupt is triggered whenever the timer register overflows, i.e. reaches its maximum value (in this case, 255, or, in hexadecimal, FFh).



In order to use overflow interrupt, you should first decide what your clock frequency will be (To learn how to do that, see "How to define clock frequency for a timer?" below). Then check the checkbox written "overflow interrupt" that appears in the Timers tab in the CodeWizard AVR (Note: The "Timer value" field can be used to set the initial value of the timer. By default, it is set to 0). Now when you generate your file, you will find that a function appears in the code:

```
.
.
.
.
#include <mega16.h>
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
// Place your code here
}
// Declare your global variables here
.
.
.
.
```

Now you can place a code here that is executed every time an overflow interrupt is generated.

This interrupt can be used to measure time intervals larger than 1 cycle. For example, let us suppose that an LED is connected to say pin 0 of Port A and you want to blink it at 0.5 Hz using overflow interrupts. Since the system frequency is 8 Mhz, you can set an appropriate clock speed say, $F_{CPU}/1024$ (see "Prescalar" below) and then do it as follows:

```
int count = 0;
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
   //Increment our variable
```

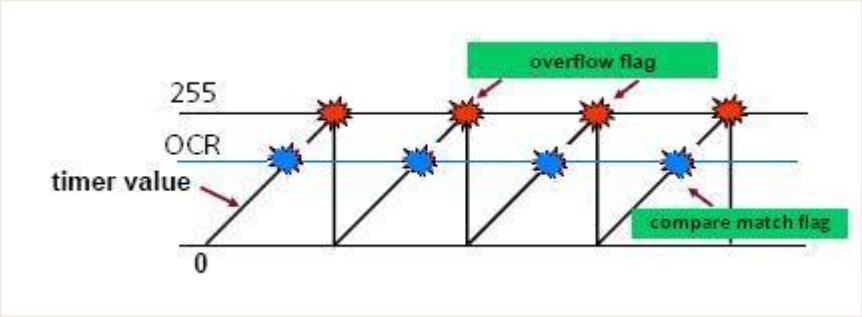
```
    count++;
    if(count==61)
    {
       PORTA.0=~PORTA.0; //Invert the Value of PORTA
       count=0;
    }
}
```

This function will increment the "count" variable every time the overflow interrupt is called and when the appropriate time has passed, will toggle the value of PORTA.0.

COMPARE MATCH INTERRUPT

A Compare Match Interrupt is issued by a timer whenever the value of the timer becomes equal to a certain predefined value. This predefined value is stored in a register known as the Output Compare Register.

Compare match Interrupts are required by the CTC, Fast-PWM, and Phase correct PWM modes of a timer (see below).



In order to use compare match interrupt, check the checkbox written "compare match interrupt" that appears in the Timers tab in the CodeVision Wizard. Set the value of the OCR in the Compare value in hexadecimal numbers. Now when you generate your file, you will find that a function appears in the code:

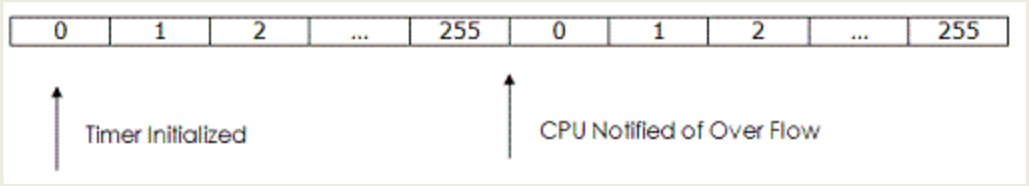
```
.
.
.
.
#include <mega16.h>
// Timer 0 output compare interrupt service routine
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
// Place your code here
}
// Declare your global variables here
.
.
.
.
```

Now you can place a code that is executed every time a compare match interrupt is generated, as shown:

```
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
   //Enter your handler code here
}
```

HOW TO USE TIMERS?

Since Timer works independently of CPU it can be used to measure time accurately. Timer upon certain conditions takes some action automatically or informs CPU. As we know a timer is an 8 bit register that keeps on increasing its value, so one of the basic conditions is the situation when timer register OVERFLOWS i.e. it has counted up to its maximum value (255 for 8 BIT timers) and rolled back to 0. In this situation timer can issue an interrupt and you must write an Interrupt Service Routine (ISR) to handle the event. There are three different timers available in Atmega16 and all the timers work in almost same way. They are TIMER0, TIMER1 and TIMER2.



PRESCALAR

The Prescalar is a mechanism for generating clock for timer by CPU clock. Every CPU has a clock source and the frequency of this source decides the rate at which instructions are executed by the processor. Atmega has clocks of several frequencies such as 1 MHz, 8 MHz, 12 MHz, 16 MHz (max). The Prescalar is used to divide this clock frequency and produce a clock for TIMER. The Prescalar can be set to produce the following types of clocks:

- No Clock(Timer stop)
- No prescaling (clock frequency = CPU frequency)
- $F_{CPU}/8$
- $F_{CPU}/64$
- $F_{CPU}/256$
- $F_{CPU}/1024$
- External clock, however, it will rarely be used.

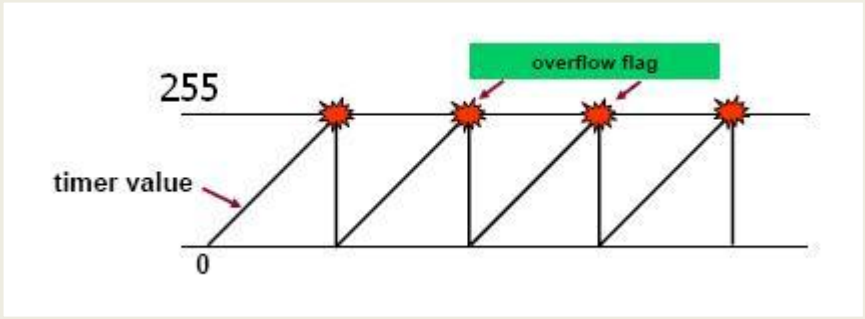
TIMER MODES

Timers are usually used in one of the following modes:

- Normal
- CTC
- Fast PWM
- Phase correct PWM

NORMAL MODE

A timer running in normal mode will count up to its maximum value. When it reaches this maximum value, it issues an Overflow interrupt and resets the value of the timer to its original value.



In the above case, you can see that the time period is 256 times the time period of the clock. 255 clock cycles are required to attain the maximum value and one clock cycle to clear the timer value.

So, $f_{timer} = f_{clock} / 256$

CTC MODE

Here we shall see how to use a timer in compare mode. In the normal mode, we set the clock of the timer using a prescalar and the let the timer run. When it overflowed, we set up an interrupt to handle the overflow. While simple, this mode has its limitations. We are confined to a very small set of values of frequency for the timer. This limitation is overcome by the compare mode.

Compare mode makes use of a register known as the Output Compare Register which stores a value of our choice. The timer continuously compares its current value with the value on the register and when the two values match, the following events can be configured to happen:

1. A related Output Compare pin can be made set (put to high), cleared (put to low) or toggled automatically. This mode is ideal for generating square waves of different frequency.
2. It can be used to generate PWM signals used to implement a DAC digital to analog converter which can be used to control the speed of DC motors.
3. Simply generate and interrupt and call a handler.

On a compare match, the timer resets itself to 0. This is called CTC – Clear Timer on Compare Match.

In this case, suppose we set our event to toggle the output pin. In that case, the output pin will remain high for one time period of the timer and will remain low for another time period.

So, $t_{out} = 2 \times t_{timer}$

From the normal case, we can draw an analogy to find out $t_{timer.}$

$t_{timer} = t_{clock} \times (OCR + 1)$

So, finally, we have the frequency as,

$f_{out} = f_{clock} / (2 \times (OCR + 1))$

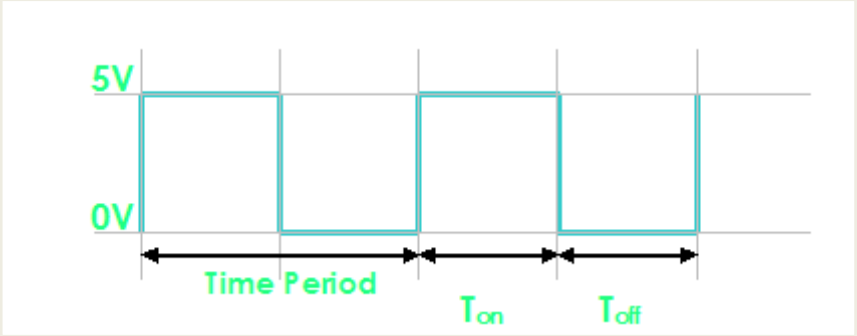
PULSE WIDTH MODULATION (PWM) MODE

A digital device like a microcontroller can easily work with inputs and outputs that have only two states - on or off. So you can easily use it to control a LED's state i.e. on or off. In the same way you can use it to turn “on or “off” any electrical device by using proper drivers (transistor, triac, relays etc). But sometimes you need more than just "on" & "off" control over the device. For example, if you want to control the brightness of an LED (or any lamp), or the speed of DC motor, then digital on/off signals will not suffice. This situation is very smartly handled by a technique called as **PWM or Pulse Width Modulation.**

PWM is the technique used to generate analog signals from a digital device like a MCU.

PWM: PULSE WIDTH MODULATION

A microcontroller can only generate two levels on its output lines, HIGH=5V and LOW=0V. But what if we want to generate 2.5V or 3.1V or any voltage between 0-5 volt as output? For these requirements, instead of generating a constant DC voltage output we generate a square wave, which has high = 5V and Low = 0V.

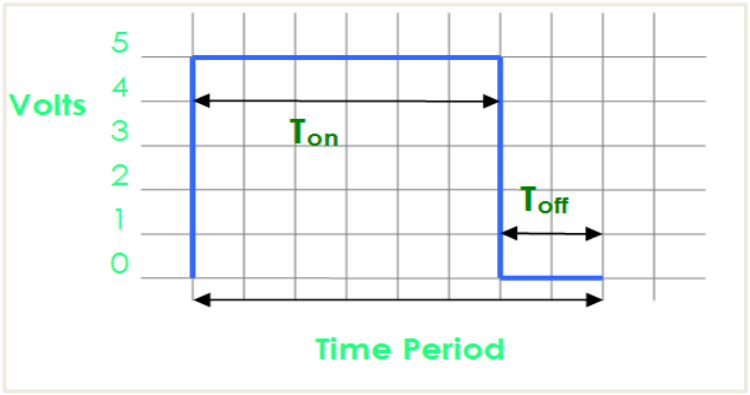


A term called as Duty Cycle is defined as

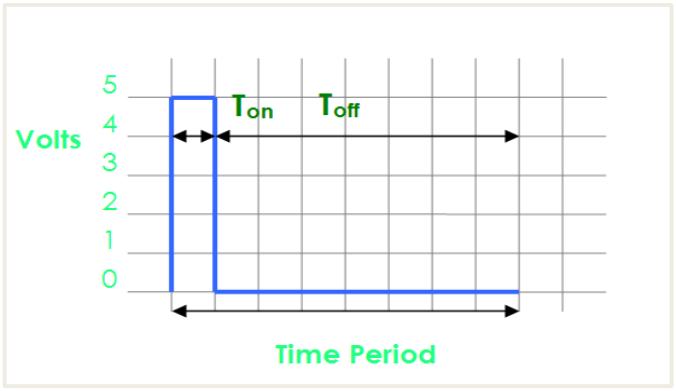
$d = t_{on} / t_{total} \times 100\%$

So you can see that the duty cycle in the above case is 50%. If the frequency of such a wave is sufficiently high (say 500 Hz) then the output you get is half of 5V i.e. 2.5 V. Thus if this output is connected to a motor (by means of suitable drivers) it will run at 50% of its full speed at 5V. The PWM technique utilizes this fact to generate any voltage between two extremes (for example between 0-12 volts). The trick is to vary the duty cycle between 0 to 100% and get same percentage of input voltage to output.

Consider the following examples:



Here the duty cycle is 75%. So the equivalent analog voltage output is 3.75V.



Here the duty cycle is 12.5%. So the analog voltage output is 0.625V.

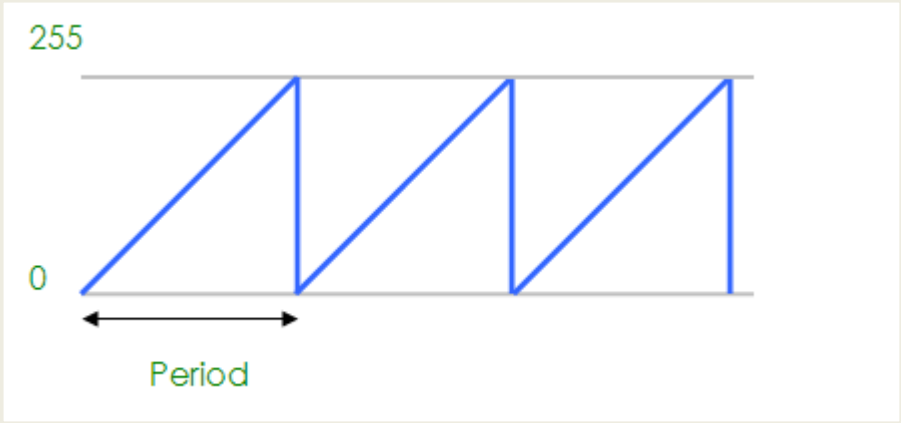
PWM SIGNAL GENERATION USING AVR TIMERS

In AVR microcontrollers, PWM signals are generated by timers. There are two methods by which you can generate PWM from timers:

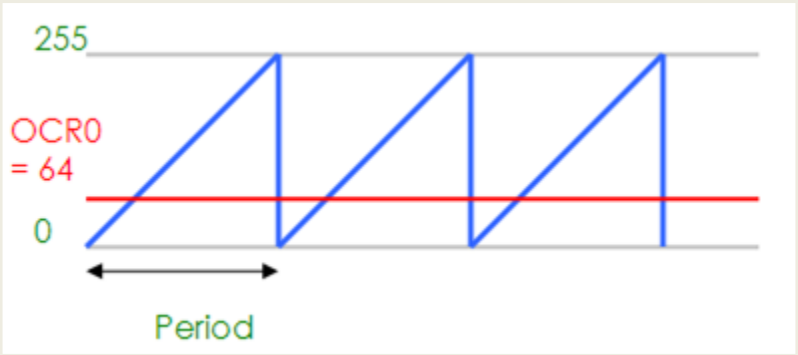
1. Fast PWM
2. Phase Correct PWM

These will be clear as we proceed.

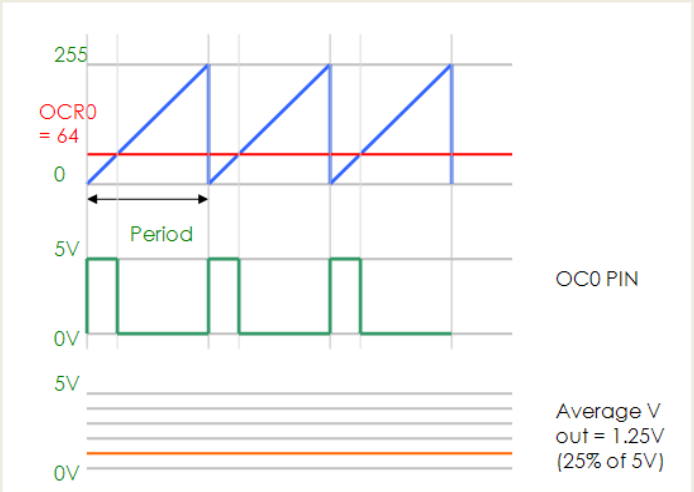
We will use the simplest timer, TIMER0 for PWM generation. So we have an 8 bit counter counting from 0 to 255 and then resetting to 0 and so on. This can be shown on graph as:



The period depends upon the PRESCALAR settings. Now for PWM generation from this count sequence OCR0 (Output Compare Register Zero) is used (Zero because it is for TIMER0 and there are more of these for TIMER1 & TIMER2). We can store any value between 0-255 in OCR0, say we store 64 in OCR0 then it would appear in the graph as follows (the RED line).



When the TIMER0 is configured for fast PWM mode, then, while the timer is counting up, whenever the value of TIMER0 counter matches the value in the OCR0 register, an output PIN is pulled low (0) and when counting sequence begin again from 0 it is SET again (pulled high=VCC). This is shown in the figure 3. This PIN is named OC0 and you can find it in the PIN configuration of ATmega32.



From the figure, you can see that a wave of duty cycle of 64/256 = 25% is produced by setting OCR0 to 64. You can set OCR0 to any value and get a PWM of duty cycle of (OCR0 / 256). When you set it to 0 you get a 0% duty cycle while setting it to 255 will give you a 100% duty cycle output. Thus by varying duty cycle you can get an analog voltage output from the OC0 PIN.

In the inverting mode the value of the OC0 pin is just the reverse of that in the above figure. So whenever the value of the TIMER0 counter is less than OCR0 value then the OC0 pin is LOW else it is HIGH. You can select the inverting or non-inverting mode in the "Output" field in the CodeWizard AVR. The inverting and non-inverting modes will have different duty cycles which are related as

$d_{inv} + d_{non\text{-}inv} = 100\%$

You can see that

$t_{out} = t_{timer} = 256 \times t_{clock}$

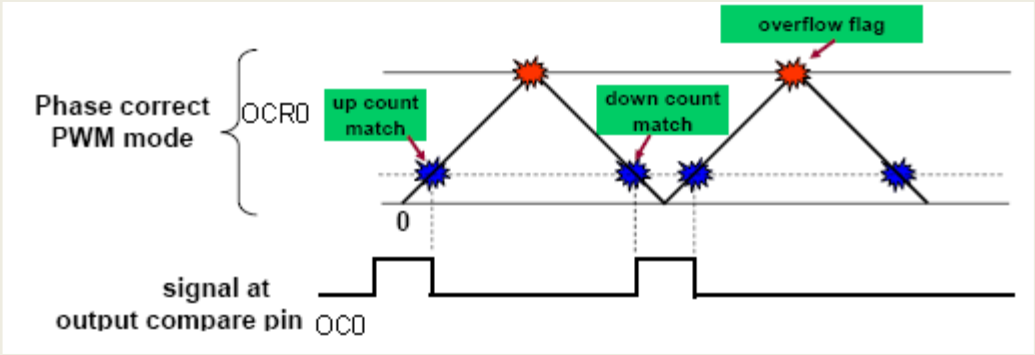
Hence,

$f_{out} = f_{clock} / 256$

PHASE CORRECT PWM MODE

This mode is very similar to the Fast PWM mode except that whenever the value of the timer reaches its maximum value then instead of clearing the value of the timer it simply starts counting down.

The value of the pin toggles only when the value of the OCR0 matches with the TIMER0 counter.



Here,

$t_{out} = t_{timer} = 2 \times t_{clock} \times OCR$

Hence,

$f_{out} = f_{clock} / (2 \times OCR)$

Overview of Timers in ATmega16

	Timer 0	Timer 1	Timer 2
Overall	- 8-bit counter - 10-bit prescaler	- 16-bit counter - 10-bit prescaler	- 8-bit counter - 10-bit prescaler
Functions	- PWM - Frequency generation - Event counter - Output compare	- PWM - Frequency generation - Event counter - Output compare 2 channels - Input capture	- PWM - Frequency generation - Event counter - Output compare
Operation modes	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM

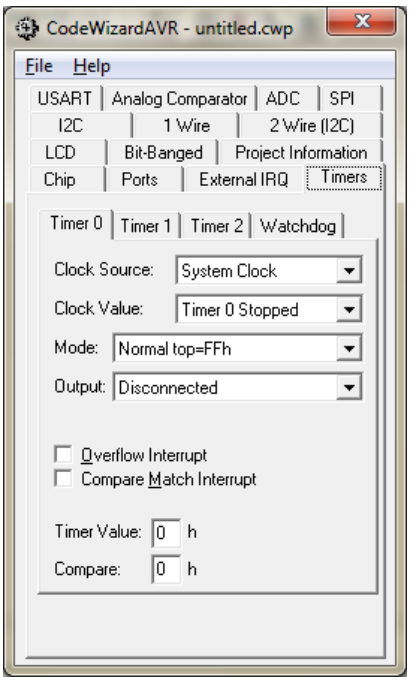
- **Timer 1 has the most capability among the three timers.**

SETTING UP TIMERS IN CODEVISION AVR

Timers can be configured in the CodeWizard AVR window while setting up a new project. To set up a timer, follow these instructions:

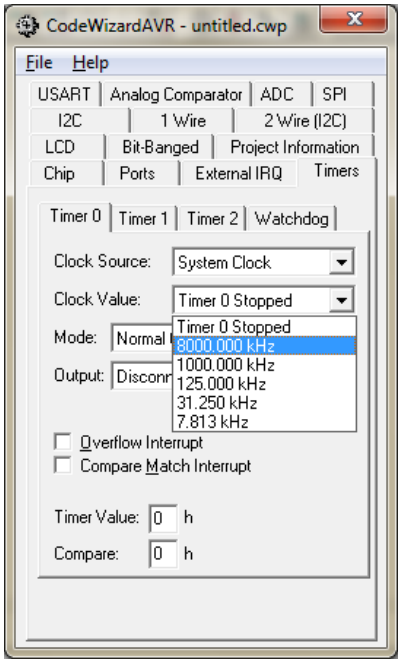
- Open CodeVision AVR and click on File -> New.

- The window shown in the above figure will appear. Click on “Yes”.
- On the CodeWizard AVR window, select your chip and frequency.
- Click on the “Timers” tab on the top.

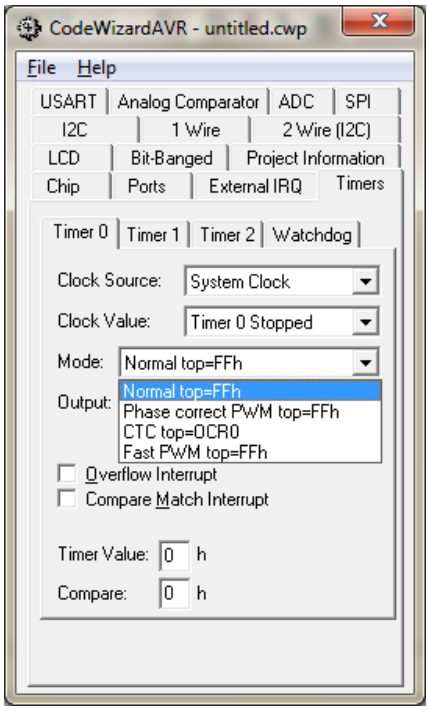


The window will allow you to configure TIMER0, TIMER1, TIMER2. For now, you can ignore the Watchdog timer.

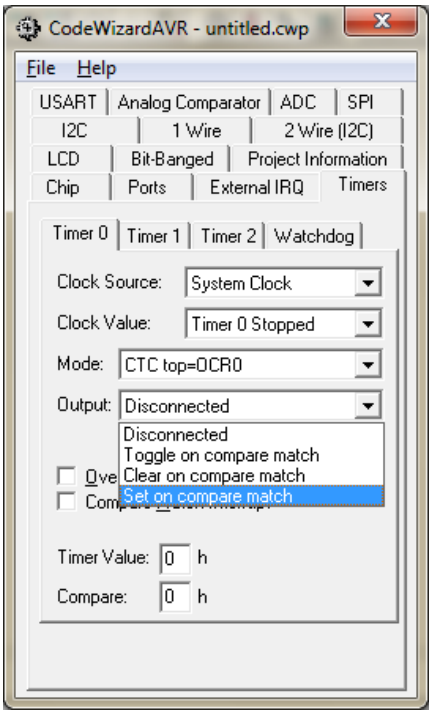
- The “Clock Source” menu will allow you to select the clock source for the current timer – you can either select the system clock or an external clock. For most purposes, you can select “System Clock”.
- The “Clock Value” menu will allow you to set the frequency of the clock of the timer. Select the appropriate value:



- Select the timer mode in the “Mode” menu:



- In the "Output" menu, select the appropriate option for your timer. This menu will have different options depending upon your timer mode, eg. Toggle, Set, Reset and Inverting, Non-Inverting, etc.:



- You can select to enable Overflow Interrupt and Compare Match Interrupt by selecting the appropriate checkboxes.
- The "Timer Value" option will allow you to select the starting value of the timer. The default value is 0.

- The “Compare Value” option will allow you to select the value for OCR. This value is required for CTC and PWM modes as well as Compare Match Interrupt. Enter the value in hexadecimal numbers.

Essentially, you’re done configuring your timer. You can now configure the remaining settings of the timer and click on File -> Generate, Save and Exit. After saving the files, you can configure your Interrupt handlers in the code window, if you activated any.