Uniform sampling and timer interrupt in Atmega 2560

In most of application we need to discretize continuous signals while interacting with real world. For measuring some real world quantity with a microcontroller we need to sample it with some required time period to discretise it for processing. Now in most of process we cannot assume that microcontroller general program will be able to sample it at same time interval as program execution flow may depend on conditional statements. So the solution for this kind of application is to use timer interrupt instead of sample the quantity within the normal execution of program with hope of same time interval within next sample.

How using timer interrupt the problem gets solved?

In Atmega 2560 there are dedicated hardware to count time duration or tics called as timers and counters. There are 6 timers in Atmega 2560 timer 0 to timer 1. In that timer 0 and 2 are 8 bit timer and all others are 16 bit timers. It just generates interrupt on over flow or at reaching some specified value. And it works in background and do not affect normal exaction of program. When above specified event happens it generates interrupt and main program completes on going execution and then gets diverted to run some specific function called ISR (interrupt service routine) and come back to continue main flow of program execution. Now if we write our ISR to sample something timer will autometically set specified time interval between consecutive samples and it also makes main processer free to do other work simultaneously.

Generally for configuring timer interrupt we need to configure the registers given below:

Register name	For what it is
TCCR1A	--For configuring the behaviour of timer
TCCR1B	--For configuring the behaviour of timer
OCR1AH	--High byte of compare match register
OCR1AL	--Low byte of compare match register
TIMSK1	--enabling timer interrupt on compare match register A

Table 1 : registers to be configured

For configuring these registers we need to read timer section in datasheet of Atmega 2560. Now let's dive in some deep with that registers to decide what we should write in them.

Register TCCR1A:

Bit	7	6	5	4	3	2	1	0	
(0x80)	**COM1A1**	**COM1A0**	**COM1B1**	**COM1B0**	**COM1C1**	**COM1C0**	**WGM11**	**WGM10**	**TCCR1A**
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	

In this register bit 2 to 7 is used to configure hardware pin behaviours so it's not useful for us right now so we will set it as 0. Bit1 and bit 0 combines with bit 3 and 4 of register TCCR1A will set the counting method which are given in table 2. So we will decide this bits later on.

Register TCCR1B:

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	

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn 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	OCRnA	Immediate	MAX

Table 1: Waveform Generation Mode Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

Table 2: Prescaler Description

In this register the bit 5, 6, 7 will not be useful for our task and we are not going to use their functionality so we will write them as 0. Bit 0, 1, 2, will configure by which factor the CPU clock will divide to feed clock to timer. It is called prescaler and according to table 3 we should set them with 101 as we want to generate long delays although we will see how to decide which precaler to be used.

Two common Modes of operation(WGMn0:3):

1. **Normal Mode:**

   The simplest mode of operation is the Normal mode (WGM02:0 = 0). In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00). In normal operation the Timer/Counter Overflow Flag (TOV0) will be set in the same timer clock cycle as the TCNT0 becomes zero. The TOV0 Flag in this case behaves like a ninth bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOV0 Flag, the

timer resolution can be increased by software. There are no special cases to consider in the Normal mode, a new counter value can be written anytime. The Output Compare Unit can be used to generate interrupts at some given time. Using the Output Compare to generate waveforms in Normal mode is not recommended, since this will occupy too much of the CPU time.

2. **Clear Timer on Compare Match (CTC) Mode**
   In Clear Timer on Compare or CTC mode (WGM02:0 = 2), the OCR0A Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT0) matches the OCR0A. The OCR0A defines the top value for the counter, hence also its resolution. This mode allows greater control of the Compare Match output frequency. It also simplifies the operation of counting external events. The timing diagram for the CTC mode is shown in figure 1. The counter value (TCNT0) increases until a Compare Match occurs between TCNT0 and OCR0A, and then counter (TCNT0) is cleared.
   An interrupt can be generated each time the counter value reaches the TOP value by using the OCF0A Flag. If the interrupt is enabled, the interrupt handler routine can be used for updating the TOP value. However, changing TOP to a value close to BOTTOM when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCR0A is lower than the current value of TCNT0, the counter will miss the Compare Match. The counter will then have to count to its maximum value (0xFF) and wrap around starting at 0x00 before the Compare Match can occur.
   For generating a waveform output in CTC mode, the OC0A output can be set to toggle its logical level on each Compare Match by setting the Compare Output mode bits to toggle mode $(COM0A1:0=1)$. The OC0A value will not be visible on the port pin unless the data direction for the pin is set to output. The waveform generated will have a maximum frequency of $fOC0 = fclk_I/O/2$ when OCR0A is set to zero (0x00). The waveform frequency is defined by the following equation:

$$f_{OCnx} = \frac{f_{clk\_I/O}}{2 * N * (1 + OCRnx)} \tag{1}$$

   The N variable represents the prescale factor (1, 8, 64, 256, or 1024).
   As for the Normal mode of operation, the TOV0 Flag is set in the same timer clock cycle that the counter counts from MAX to 0x00.

Figure 16-5. CTC Mode, Timing Diagram

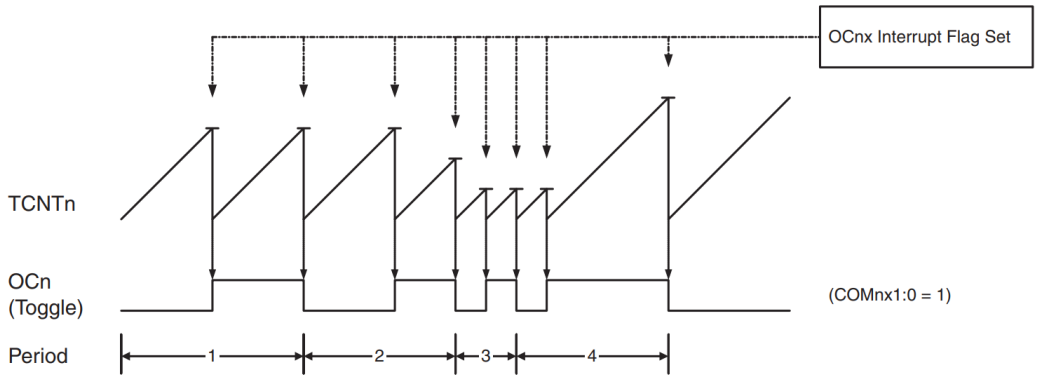

Figure 1: Timer compare match mode

According to upper discussion to use CTC mode we have to write WGM10:3 bits with B0100.

How to select prescaler:

We just need first to decide how much time we want to count for an example we want to generate delay of 1 sec and the value we can count max is 65535. And each step will take 1/16mhz = 0.06 micro second if we feed timer with CPU clock directly as our Arduino mega is having 16mhz Cpu clock. And the max time we can count is time delay between timer1 starting from 0X0000 to 0XFFFF. In our case it will be 0.06 * 65535 = 4.09 milliseconds only so we can extend this maximum range by just slowing timer clock. If we slow timer clock with some factor X our maximum delay will be increase by same factor X here we need to generate 1 sec so the minimum factor needed will be

$$\frac{1}{0.004} = 250.$$

Here according to table 3 we can use factor of 256 and 1024 both so we are choosing 1024 here.

Register OCR1AH and OCR1AL:

Bit	7	6	5	4	3	2	1	0	
(0x89)	OCR1A[15:8]								OCR1AH
(0x88)	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	

This register will togather store the value to be compared with register TCNT1A to generate interrupt. As we are using here Clear Timer on Compare Match. We can find out the values to be stored in this register using equation 1.

Register TIMSK1:

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	OCIE1C	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The register will be used to enable interrupt for any of the event. Here as we are using OSR1A

we have to enable the interrupt for it by making OCIEA bit 1.
So as the conclusion the values to be write to generate 500 ms delay for blinking is given as below

$$TCCR1A = B00000000; \quad \text{(clear whole register)}$$

$$TCCR1B = B00001101; \quad \text{(setting up prescaler to 1024)}$$

$$OCR1AH = 0x00; \quad \text{(high byte of compare match register)}$$

$$OCR1AL = 0xFE; \quad \text{(low byte of compare match register)}$$

$$TIMSK1 = B00000010; \quad \text{(enabling timer interrupt on compare match register A)}$$

In micro controller when interrupt occurs the CPU gets vectored to a specific location where the corresponding interrupt ISR is written. In Arduino we need to define our ISR using function ISR(vector address). Where to pass argument or vector address we need to refer vector table in data sheet in our case the vector address will be $\$0022$. We can also directly write $TIMER1\_COMPA\_vect$ and the compiler will replace it with its corresponding address automatically.