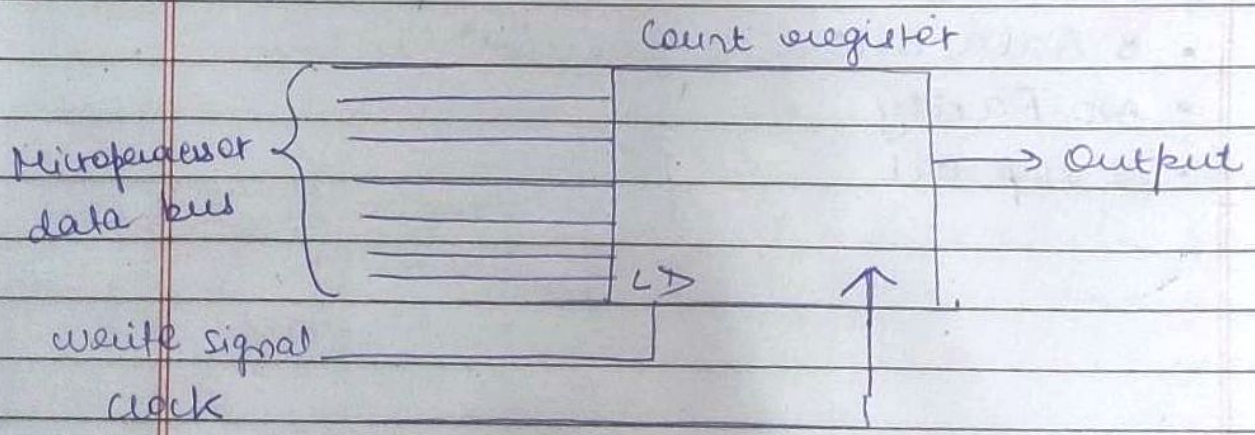
Unit-4

* Introduction to Timers / Counters &

	Timer	Counter
(i)	The register is incremented for every machine cycle.	The register is incremented considering 1 to 0 transition corresponding to an external pin (T_0, T_1)
(ii)	Max^m count rate 1/12 of the oscillator frequency.	Max^m count rate 1/24 of the oscillator frequency.
(iii)	Uses frequency of the internal clock and generates delay.	Uses external signal to count pulses.



Simple Counter / Timer

→ Above is a simple timer / counter similar to those included on-chip with a micro controller.

→ The timer consists of a loadable 8-bit count register, an input clock signal, and an output signal.

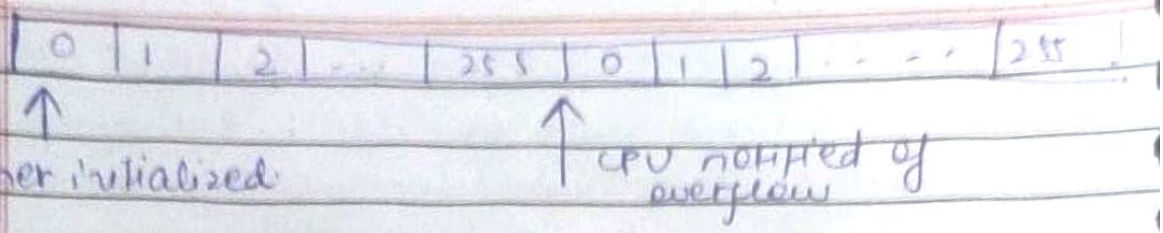
→ Software loads the count register with initial value bet^n 0x00 and 0XFF.

→ Each subsequent clock signal increments that value.
→ When a 8-bit count overflows, the output signal is active. The output signal may thereby trigger an interrupt at the processor.
→ To restart the timer, software reloads the count register with same or different initial value.
→ If a counter is an up counter, it counts up from the initial value toward 0xFF.
A down counter counts down, towards 0x00.
→ The ATmega 328P has a total 3 timer/counter.
Timer/counter0 (8-bits) Timer/counter1 (16-bits) Timer/counter2 (8-bits).

* Timer/counter :-
time delay generator / counts external events.

→ A Timer is a piece of hardware built in the Arduino controller and depending on the model it could have different amount of timers.
→ Arduino uno has 3-timers.
→ The timer can be programmed by some special registers so is like programming a clock

* Timer :- is an 8/16 bit register that keeps on ↑ its value, so one of its basic conditions is the situation where a ~~counter~~ timer register overflows i.e., it has counted up to its max^m value (255 for 8 bit) 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.



Timer0 can be loaded with $2^8 - 1 = 256 - 1 = 255$ FFH
Timer1 " " " " $2^{16} - 1 = 65536 - 1 = 65535$
Timer2 " " " " $2^8 - 1 = 256 - 1 = 255$ FFH

* Prescalar of CPU clock :-

→ Prescalar is a mechanism for generating clock for Timer by CPU clock.

→ 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 following types of clocks :

No clock (Timer stop)
No Prescaling (clock frequency = CPU frequency)
FCPU/8
FCPU/64
FCPU/256
FCPU/1024
External clock

* Timer Mode :-

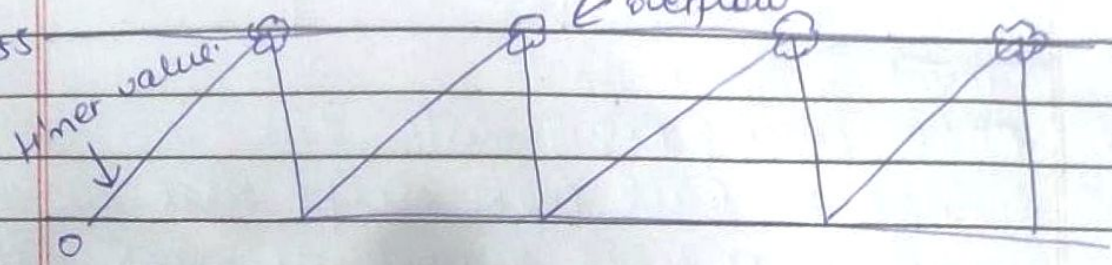
(I) Normal
(II) CTC
(III) Fast PWM
(IV) Phase correct PWM

① Normal Mode :-

→ A timer running in normal mode will count upto its max^m value. When it reaches its max^m value, it issues an overflow interrupt and reset the value of the timer to its original value.

→ $f_{timer} = f_{clock}/256$

→ Limitation :- We are confined to use a very small set of values of frequency for the timer. This is overcome by compare mode.

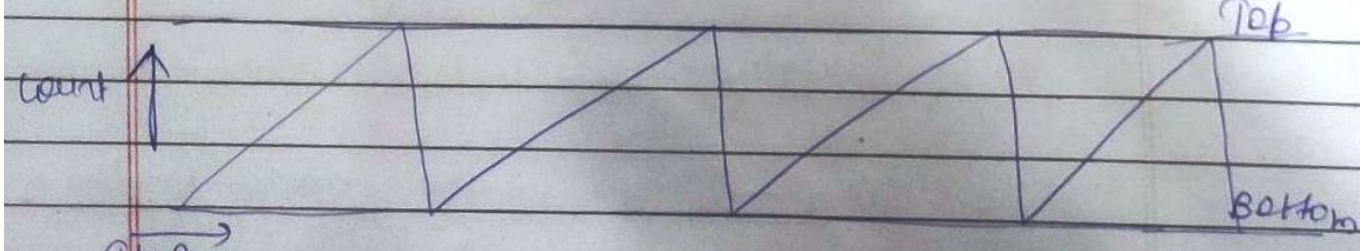


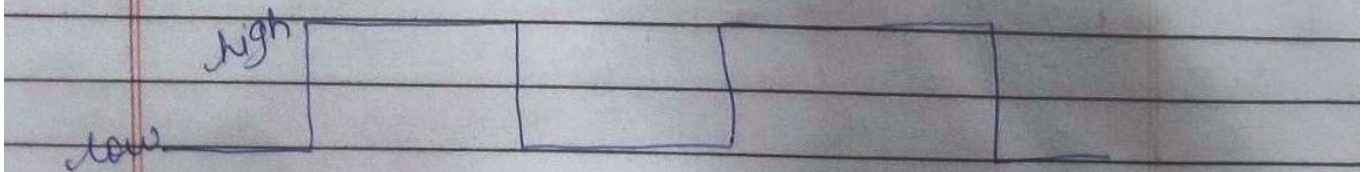
② CTC Mode [Clear Timer on Compare Match) :-

→ Compare mode makes use of a register known as Output Compare Register which stores a value of our choice.

→ The timer continously compares its current value with the value on the register and when two values match, the timer resets itself to 0.

→ The output pin will remain high for one time period and remain low for another time period.

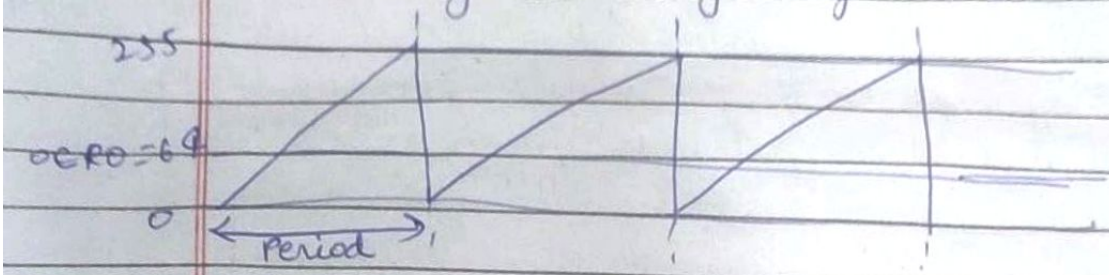




(iii) Fast PWM :-

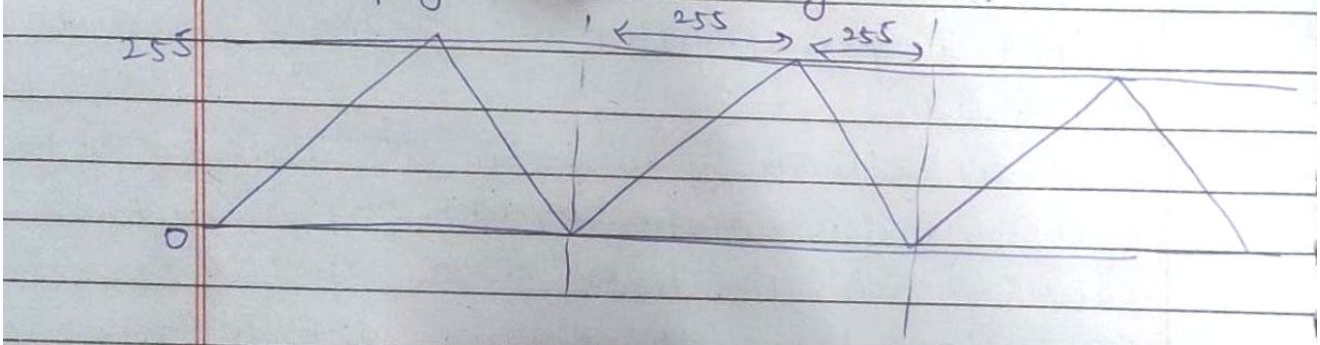
→ Based on single-slope operation.

→ In single slope operation, the register TCNTn counts from bottom value to max^m value and its value resets to zero. The counting starts again from zero.



(iv) Phase correct PWM mode :-

→ Similar to Fast PWM except that whenever the value of Times reaches its max^m value then instead of clearing the value of Time it simply starts counting down.



* Programming the Timers :-

(i) Timer 0 Programming :-

TCCR0A – Timer/Counter Control Register A

7	6	5	4	3	2	1	0
COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00
R/W	R/W	R/W	R/W	R	R	R/W	R/W
0	0	0	0	0	0	0	0

Bits 7:6 - COM0A1:0 : Compare Match Output A mode
→ These bits control the output compare Pin (OC0A) behaviour
→ If one or both of the COM0A1:0 bits are set, the OC0A output overrides the normal port functionality of I/O pins it is connected to.

Bits 3,2 - Reserved Bits :-
→ These bits are reserved and always reads as zero.

Bits 1:0 - WGM01:0 : Waveform Generation Mode :-
→ These bits control the counting sequence of the counter, the source of the max^m counter value, and what type of ~~gen~~ waveform generation is to be used.

* TCCR0B - Timer/Counter Register B :-

7	6	5	4	3	2	1	0
FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
W	W	R	R	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

① Bit 7 - FOC0A : Force Output compare A
→ This bit is only active when WGM bits specify a non-PWM mode.
→ However, for ensure compatibility with future devices, this bit must be set to 0 when TCCR0B is written when operating.
→ Doesn't generate any interrupt.

② Bit 6 - FOC0B : Force Output compare B
Same as Bit 7
Does not generate any interrupt
Always reads a zero.

Bit 5:4 : Reserved Bits
→ reserved & always reads as zero.
Bit 3 : Waveform Generation Mode
Bit 2:0 - CS02:0 : Clock Select
→ The three clock select bits select the clock source to be used by timer/counter.

* Timer/Counter Register

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

→ The timer/counter register gives direct access for both for read & write operations to the timer/counter unit 8-bit counter

* OCR0A - Output Compare Register A :
OCR0B - " " B

7	6	5	4	3	2	1	0
OCR0A[7:0] / OCR0B[7:0]							
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

→ contains 8-bit value that is continously compared with the counter value (TCNT0).
→ A match can be used to generate an output compare interrupt.

* TIMSK0 - Timer/Counter Interrupt Mask Register

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
R	R	R	R	R	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bits 7-3 : Reserved Bits

Bit 2 OCIEOB : Timer/counter Output Compare Match B Interrupt enable

→ When OCIEOB is written to 1, and the I-bit in the status register is set, the Timer/counter compare Match B interrupt is enabled.

→ ~~An~~ The corresponding interrupt is ~~generated~~ executed ~~Bit 1 OCIEOA when~~ if a compare match in Timer/counter occurs.

Bit 1 : OCIEOA : same as OCIEOB

Bit 0 : TOIEO : Timer counter0 Overflow Interrupt enable

→ When TOIEO is written to 1 the I-bit in the status register is set, the Timer/counter overflow interrupt is enabled.

* Timer/counter Interrupt Flag Register :-

7	6	5	4	3	2	1	0
-	-	-	-	-	OCFOB	OCFOA	TOVO
R	R	R	R	R	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bits 7-3 : Reserved Bits

Bit 2 - OCFOB - Timer/counter0 Output Compare B Match Flag

→ this bit is set when a compare match occurs bet^n Timer/counter and the data in OCR0B (Output compare Register0 B)

Bit 1 - OCFOA - Timer/counter0 Output compare A match Flag.

→ Same as OCFOB

Bit 0 : TOV0: Timer/Counter0 Overflow Flag

→ This bit is set when an overflow occurs in Timer/Counter0.

→ TOV is cleared by h/w when executing the corresponding interrupt handling vector.

* Timer 2 Programming :

~~Normal~~ like time 0, but no external clock, timer only

→ Timer 2 is preferred timer among programmers for short time delays because its prescalar has highest no. of options.

→ It can run in normal mode or CTC mode

→ In normal mode → TOV2 can generate overflow interrupt

→ In CTC mode → OCIF2 generates an interrupt when it detects a compare match.

* Registers are same that of Timer0.

* Timer1 Programming :

→ 16-bit counter/timer

→ 2 8-bit registers to control timer 1
- TCCR1L and TCCR1H

→ 2 registers in compare mode
- OCR1A and OCR1B

* Timer1 Control Registers :

TCNT1

D15 . . . . . D8 D7 . . . . . D0

TCNT1H TCNT1L

→ To control the timers we have two main registers, the TCCRA and TCCRB, each with the no. of bits
→ So for timer 1 we have TCCR1A & TCCR1B
→ TCCRA is used for controlling the PWM mode.
→ After TCCRA we need to change TCCRB register, here all we care are first 3-bits which are used to define the prescalar value.

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

TCCRA

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

TCCRB

Bit 7 :- ICNC1 : Input capture Noise canceller
→ Setting this bit to 1 activates the ICNC after which the input from the Input capture Pin is filtered.
(ICP1)

Bit 6 :- ICES1 : Input capture Edge Select
→ this bit selects which edge on the ICP1 is used to trigger a capture event.

Bit 5 :- Reserved Bits
Bit 4:3 :- WGM13:2 Waveform Generation Mode
Bit 2:0 :- CS12:0 Clock select

* TIFR1 (Timer/Counter1 Interrupt Flag Register)

7	6	5	4	3	2	1	0
-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
R	R	R/W	R	R	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit 7,6 :- Reserved Bits

Bit 5 :- ICF1 : Timer/Counter1, Input Capture Flag

→ this flag is set when a capture event occurs on the ICP1 pin (Input Capture Pin)

Bit 4,3 :- Reserved Bits

Bit 2 :- OCF1B : Timer/Counter1, Output Compare B Match Flag

→ this flag is set in the timer clock cycle after the counter (TCNT1) value matches the output compare register B (OCR1B)

Bit 1 :- OCF1A :

→ same as OCF1B

Bit 0 :- TOV1 : Timer/Counter1, Overflow Flag.

→ the setting of this flag is dependent of the WGM13:0 bits setting.

→ In normal & CTC modes, the TOV1 is set when the timer overflows.

* Timer1 Modes :- Normal mode, CTC mode

* Interrupts in ATmega 328p :-

* Methods of I/O operation :-

① Synchronous I/O operation :-

→ possible when I/O device operates at the same speed as that of MCU.

→ In this I/O operation is performed by MCU assuming that the device is always ready.

Asynchronous I/O operation :-

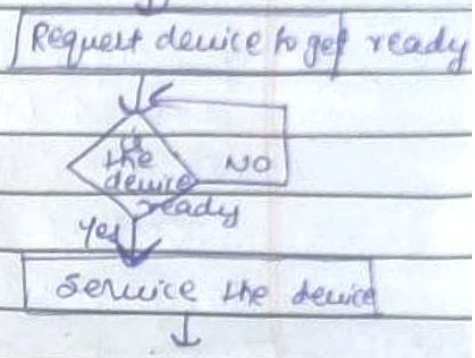
→ I/O device is very slow when compared to MCU.

→ It can perform I/O operation by following methods
- By polling
- By using interrupts

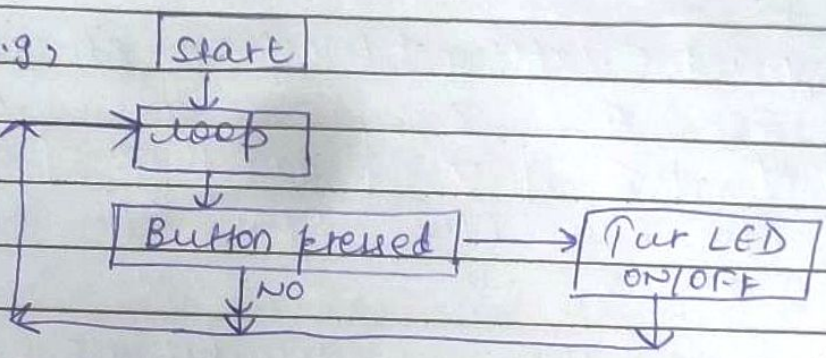
* I/O operation by Polling :-

→ Polling results in waste of MCU time and power

→ Does not permit masking of devices : all devices have to be checked.



e.g.,



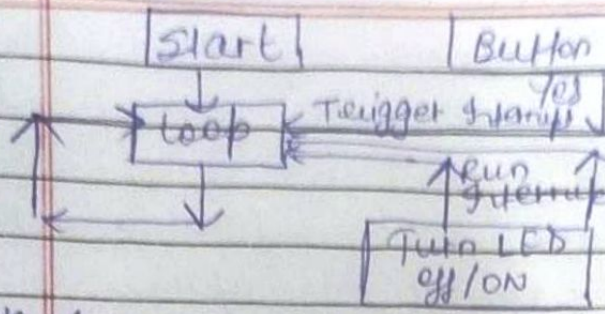
* I/O operation by Interrupts :

→ The MCU keeps on doing some useful operation, when I/O device is ready it interrupts the MCU.

→ The MCU completes execution of current instruction, saves the address on the stack, and jumps to ISR (Interrupt Service Routine) to serve the device.

→ After this MCU returns back to interrupted program by loading return address from stack to program counter.

→ Interrupts can be enabled or disabled dynamically



* Interrupt Vectors in Atmega 328p :-

→ Interrupt vector is a definite address in the program memory, where a JMP instruction should be written to jump to ISR corresponding to the interrupt.

→ Occupies two program memory words in order to provide space for JMP instruction

* Enabling and Disabling Interrupts in Atmega 328p

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

In SREG if I = 1, then all interrupts are enabled
In SREG if I = 0, " " " " disabled
Status Register

* Bits of (EICRA) External Interrupt control Register A :-

7	6	5	4	3	2	1	0
-	-	-	-	ISC11	ISC10	ISC01	ISC00

ISC11 → The rising edge of INT1 generates interrupt request
ISC10 → " falling " " " " " " "
ISC01 → The logical change of INT0/INT1 " " "
ISC00 → The low level of INT0 / INT1 generates interrupt request.
External interrupt 0

Masking → disabling
unmasking → enabling

* Bits of EIMSK (External Interrupt Mask Register)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INT1	INT0

When INT1 = 1 INT1 is enabled if I=1 in SREG
" INT1 = 0 " " disabled even if I = 1 " "
" INT0 = 1 INT0 " enabled if I=1 " "
" INT0 = 0 " " disabled even if I=1 " "

Activity on the pins of INT1/INT0 will cause an interrupt.
Bit 2-7 are unused and always be read as 0.

* Bits of (EIFR) External Interrupt Flag Register

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INTF1	INTF0

→ Bits 2-7 are unused and always be read as 0.
→ when INTF1 interrupt is triggered, INTF1 becomes 1. the flag is cleared when ISR is executed.
→ when INT0 interrupt is triggered, INTF0 becomes 1. the flag is cleared when ISR is executed.

* Interrupts :

Type 1	Type 2
(i) Event is remembered when interrupt is disabled	(i) Event is not remembered when interrupt is disabled
(ii) If interrupt is not enabled, flag is set & when interrupt is enabled again flag is reset.	(ii) Signal level causes interrupt.

* When will external Interrupt be Triggered
→ When the INT0 or INT1 interrupts are enabled and are configured as low level triggered (Type 2), the interrupts are Triggered as long as the pin is held low.
→ The low level is held until the completion of the currently executing instruction
→ Among other applications, low level interrupts may be used to implement handshake protocol

7	6	5	4	3	2	1	0
-	-	-	-	-	PCIF2	PCIF1	PCIF0

→ In addition to the 2 external interrupts, 23 pins can be programmed to trigger an interrupt if there pin changes state
→ These 23 pins are in turn divided into 3-interrupt groups.
→ Each of the groups are assigned to one pin change interrupt flag (PCIF) bit 2:0
→ A pin change interrupt flag (PCIF) will be set if the interrupt is enabled, and any pin assigned to the group changes state.

* Arduino Language Support for External Interrupts :-

Low → to trigger the interrupt whenever pin is low
Change → " " " " " " changes value
Rising → " " " " " " goes from low to high
Falling → " " " " " " goes from high to low

Example 9-39

Write a C program to toggle all the bits of PORTB continuously with some delay. Use Timer0, Normal mode, and no prescaler options to generate the delay.

Solution:

```
#include "avr/io.h"
void T0Delay ( );
int main ( )
{
      DDRB = 0xFF;       //PORTB output port

      while (1)
      {
            PORTB = 0x55;      //repeat forever
            T0Delay ( );       //delay size unknown
            PORTB = 0xAA;      //repeat forever
            T0Delay ( );
      }
}

void T0Delay ( )
{
      TCNT0 = 0x20;            //load TCNT0
      TCCR0 = 0x01;            //Timer0, Normal mode, no prescaler
      while ((TIFR&0x1)==0);   //wait for TF0 to roll over
      TCCR0 = 0;
      TIFR = 0x1;              //clear TF0
}
```

Example 9-40

Write a C program to toggle only the PORTB.4 bit continuously every 70 µs. Use Timer0, Normal mode, and 1:8 prescaler to create the delay. Assume XTAL = 8 MHz.

Solution:

XTAL = 8MHz → $T_{machine\ cycle}$ = 1/8 MHz
Prescaler = 1:8 → T_{clock} = 8 × 1/8 MHz = 1 µs
70 µs/1 µs = 70 clocks → 1 + 0xFF – 70 = 0x100 – 0x46 = 0xBA = **186**

```
#include "avr/io.h"

void T0Delay ( );

int main ( )
{
      DDRB = 0xFF;       //PORTB output port

      while (1)
      {
            T0Delay ( );             //Timer0, Normal mode
            PORTB = PORTB ^ 0x10;    //toggle PORTB.4
      }
}

void T0Delay ( )
{
      TCNT0 = 186;       //load TCNT0
      TCCR0 = 0x02;      //Timer0, Normal mode, 1:8 prescaler
      while ((TIFR&(1<<TOV0))==0);  //wait for TOV0 to roll over

      TCCR0 = 0;         //turn off Timer0
      TIFR = 0x1;        //clear TOV0
}
```


Example 9-7

Assuming that XTAL = 8 MHz, write a program to generate a square wave with a period of 12.5 µs on pin PORTB.3.

Solution:

For a square wave with T = 12.5 µs we must have a time delay of 6.25 µs. Because XTAL = 8 MHz, the counter counts up every 0.125 µs. This means that we need 6.25 µs / 0.125 µs = 50 clocks. 256 – 50 = 206 = 0xCE. Therefore, we have TCNT0 = 0xCE.

```
TCCR0=0x01    //normal mode, no prescaling
TCNT0=0xCE
```

```
#include <avr/io.h>
void T0Delay ();
int main ()
{
DDRB=0xFF;     // PORTB output port
while(1)
{            //repeat forever
PORTB = 0x55;
T0Delay();   //delay size unknown
```

```
PORTB=0xAA;
T0Delay();
}
}
void T0Delay()
{
TCNT0=0;  // timer starts from 00
OCR0=199; // Output Compare Register has value 199
TCCR0A=0x02;
TCCR0B=0x05; /*Run Timer0,(CTC)Clear Timer on Compare Match,// 1:1024
Prescaler */

while((TIFR & 0x02)==0);   // wait for OCF0 to roll over
TCCRB=0;// Stop Timer
TIFR=0x02;//clear TF0
}
```

```
// this code sets up timer2 for a 250us  @ 16Mhz Clock


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


int main(void)
{
    OCR2A = 62;

    TCCR2A |= (1 << WGM21);
    // Set to CTC Mode

    TIMSK2 |= (1 << OCIE2A);
    //Set interrupt on compare match

    TCCR2B |= (1 << CS21);
    // set prescaler to 64 and starts PWM

    sei();
    // enable interrupts


    while (1)
    {
        // Main loop
    }
}

ISR (TIMER2_COMPA_vect)
{
    // action to be done every 250 usec
}
```

ATmega168/328 Code:

```
// this code sets up timer1 for a 1s  @ 16Mhz Clock (mode 4)


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


int main(void)
{
    OCR1A = 0x3D08;

    TCCR1B |= (1 << WGM12);
    // Mode 4, CTC on OCR1A

    TIMSK1 |= (1 << OCIE1A);
    //Set interrupt on compare match

    TCCR1B |= (1 << CS12) | (1 << CS10);
    // set prescaler to 1024 and start the timer


    sei();
    // enable interrupts


    while (1)
    {
        // we have a working Timer
    }
}

ISR (TIMER1_COMPA_vect)
{
    // action to be done every 1 sec
}
```

Assuming that a 1 Hz clock pulse is fed into pin T0, use the TOV0 flag to extend Timer0 to a 16-bit counter and display the counter on PORTC and PORTD.

Solution:

```
#include "avr/io.h"

int main ( )
{
      PORTB = 0x01;            //activate pull-up of PB0
      DDRC = 0xFF;             //PORTC as output
      DDRD = 0xFF;             //PORTD as output

      TCCR0 = 0x06;            //output clock source
      TCNT0 = 0x00;

      while (1)
      {
            do
            {
                  PORTC = TCNT0;
            }while((TIFR&(0x1<<TOV0))==0);//wait for TOV0 to roll over

            TIFR = 0x1<<TOV0;        //clear TOV0
            PORTD ++;                //increment PORTD
      }
}
```

```
int pin = 2; //define interrupt pin to 2
 volatile int state = LOW; // To make sure variables shared between
an ISR //the main program are updated correctly, declare them as
volatile.
void setup()
 {
 pinMode(13, OUTPUT); //set pin 13 as output
attachInterrupt(digitalPinToInterrupt(pin), blink, CHANGE);
//interrupt at pin 2 blink ISR when pin to change the value
 }
 void loop()
{
digitalWrite(13, state); //pin 13 equal the state value
 }
 void blink()
 {
//ISR function
 state = !state; //toggle the state when the interrupt occurs
}
```

```
#include <avr/io.h>
#include <avr/interrupt.h>  //This is needed to use interrupts.

int main(void)
{
    DDRD &= ~(1 << DDD2);   //Make PD2 (PCINT0 pin) an input line.
    PORTD |= (1 << PORTD2); //Enable the pull-up resistor.
    EICRA |= (1 << ISC00);  //Set INT0 to trigger on ANY logic change.
    EIMSK |= (1 << INT0);   //Unmaks on INT0.
    sei();                  //Turn on interrupts.
    while(1)
    {
        /*main program loop here */
    }
}

ISR (INT0_vect)
{
    /* ISR for INT0 code here */
}
```