Embedded System on AVR Microcontroller (ATMEGA32)

Exp7: Timer and its Mode (Normal, Counter, CTC, PWM)

Submitted by
Ronit Dutta, MS in IOT and Signal Processing
Department of Electrical Engineering, IIT Kharagpur

Under the guidance of
Aurobinda Routray, Professor, Electrical Engineering

Department of Electrical Engineering

Indian Institute of Technology Kharagpur

March, 2025

Why are Timers required in Embedded System Applications?

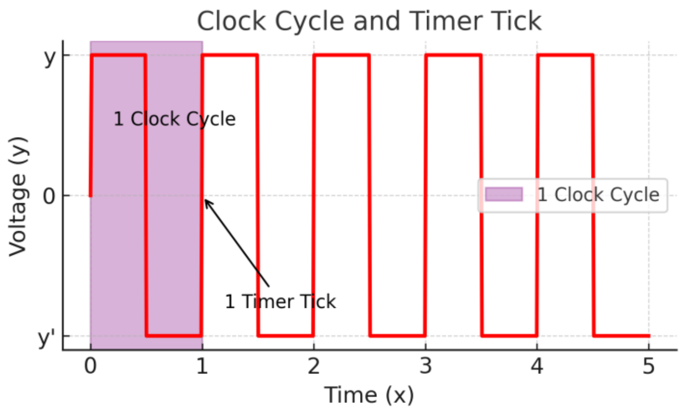
Timers are essential in embedded systems applications because they provide precise time control and event scheduling. Here are some key reasons why timers are required:

1. **Task Scheduling:** Timers help in scheduling periodic tasks, such as sensor sampling, communication, or data logging.
2. **Generating Delays:** Many embedded applications require precise time delays, which can be achieved using timers instead of software loops, reducing CPU load.
3. **Communication Protocols:** Timers help manage time-sensitive communication protocols like UART, SPI, I2C, and CAN by generating clock signals and handling timeouts.
4. **Pulse Width Modulation (PWM):** Timers generate PWM signals used for motor control, LED brightness control, and other applications requiring variable pulse width signals.
5. **Event Counting & Measurement:** Timers can count external events (such as counting the number of pulses from a sensor) or measure signal frequencies and pulse widths.
6. **Watchdog Timer (WDT):** A watchdog timer helps in system reliability by resetting the microcontroller if the system hangs or crashes.

Overall, timers improve the efficiency and reliability of embedded systems by handling time-dependent operations without burdening the processor.

What are the various terminologies associated with Timers?

1. **Clock Ticks:** Clock signals are square wave signals and 1 clock tick equals 1 clock cycle. Timers count time in terms of clock ticks.



2. **Oscillator:** It is the component that fundamentally generates the clock signal of a desired frequency for the proper operation of an embedded system application.
3. **Clock Frequency:** It is the no. of clock ticks generated in 1 second and is directly responsible for determining the timer's speed & resolution.
4. **Timer Resolution:** It is the minimum time duration that can be measured by the timer.

$$\text{Timer Resolution} = \frac{1}{\text{Clock Frequency}}$$

What are the various ways in which a timer can operate?

A Timer can operate in 3 ways:

1. **One-shot mode:** The timer turns on for a particular duration and then switches off by itself. It works like a stopwatch and is used when something needs to start at a specific time.
2. **Periodic Mode:** In this mode, after the set time duration, the timer resets to its initial value and starts counting again. This process repeats continuously until the timer is manually disabled using software or hardware.
3. **Interrupt Mode:** Timers are one of the best sources of interrupts in an MCU (Microcontroller Unit) or MPU (Microprocessor Unit). This feature is mainly used to execute multiple operations at the same time. Timers can generate interrupts for various reasons, such as timer overflow, compare match, or input capture etc.

Types of Timers in Embedded Systems

1. **General Purpose Timers:** These are the standard timers used to provide time delays, measure time between two events, and perform periodic operations. The Atmega32 has General Purpose Timers.
2. **SysTick Timers:** They are commonly found in ARM Cortex processors. These timers have only one mode of operation, which is decrementing and periodic. They are mainly used to provide time delays and periodic interrupts. SysTick timers are primarily used in RTOS (Real-Time Operating Systems) for multitasking.
3. **Real-Time Clock (RTC):** The RTC functions like a digital watch, providing real-world time with a resolution of 1 second. It is used in applications where timekeeping is essential.
4. **Watchdog Timer:** This is a special type of timer in an MCU (Microcontroller Unit) or MPU (Microprocessor Unit). It is used to prevent system hangs by resetting the system when a malfunction is detected. The Atmega32 has a Watchdog Timer.

Introduction to AVR Timers:

To create time delays, we connect an oscillator to the clock pin of a counter or we can use an internal oscillator. Each time the oscillator ticks, the counter increases by one. The number stored in the counter shows how many ticks have passed since it was last reset. Since we know the speed of the oscillator, we can calculate the time of each tick. By checking the counter value, we can determine how much time has elapsed.

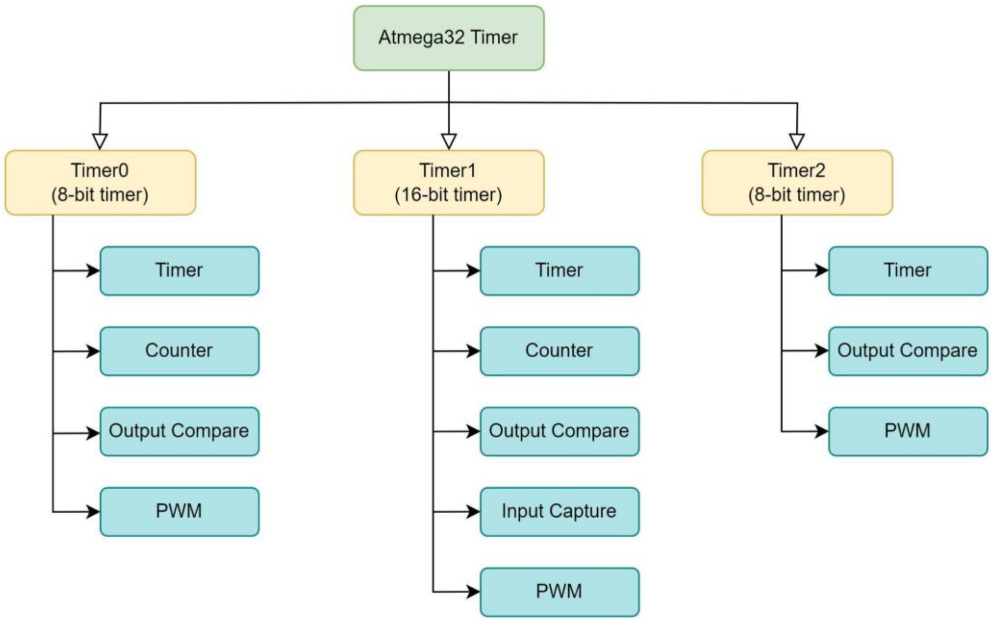
One way to generate a time delay is to clear the counter at the start and wait until it reaches a specific value. For example, we want to generate a 100us delay. Assume the microcontroller oscillator frequency is 1 MHz, the counter increases once per microsecond. So, to generate 100-microsecond delay, we clear the counter and wait until it reaches 100.

Another method is to use the counter's overflow flag. Each counter has a flag that turns ON when the counter overflows and resets to zero. The flag is then cleared by software. To generate a delay using this method, we load a starting value into the counter and wait for the overflow flag to turn ON. For instance, in a microcontroller with a 1 MHz frequency and an 8-bit counter, if we want a 3-microsecond delay, we set the counter to 0xFD. After three ticks, the counter overflows, resets to 0x00, and the flag is set.

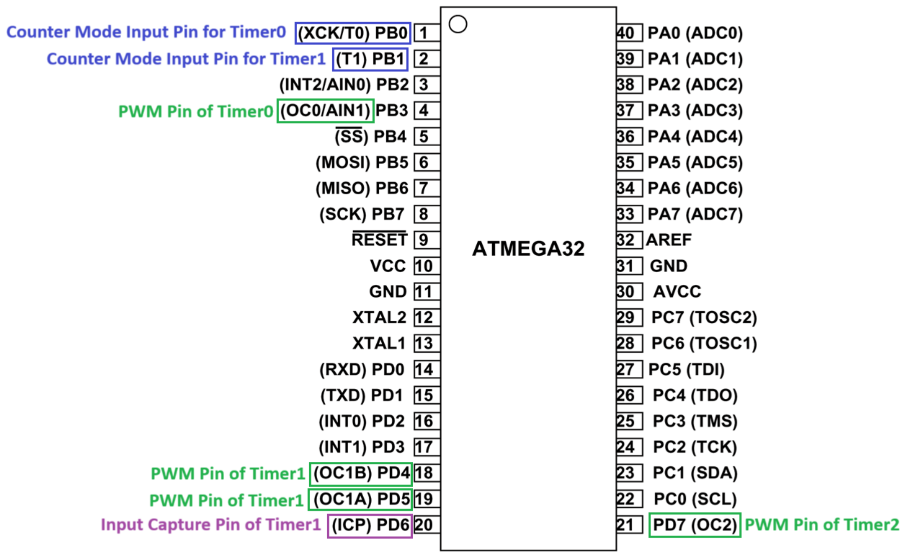
In AVR microcontrollers, there are one to six timers, named Timer0, Timer1, Timer2, etc. These timers can be used to generate delays or count external events. Some timers are 8-bit, while others are 16-bit. In the ATmega32, there are three timers:

- Timer0 (8-bit)
- Timer1 (16-bit)
- Timer2 (8-bit)

These Timers can be used in different ways:



Pins Associated with Timer Functionalities:



● Registers associated with Timer0 of Atmega32:

1. Timer/Counter Control Register – TCCR0

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	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 – FOC0: Force Output Compare

The FOC0 bit is only active when the WGM00 bit specifies a non-PWM mode. However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0 is written when operating in PWM mode. When writing a logical one to the FOC0 bit, an immediate compare match is forced on the Waveform Generation unit. The OC0 output is changed according to its COM01:0 bits setting. Note that the FOC0 bit is implemented as a strobe. Therefore it is the value present in the COM01:0 bits that determines the effect of the forced compare.

A FOC0 strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR0 as TOP.

The FOC0 bit is always read as zero.

• Bit 6, 3 – WGM01:0: Waveform Generation Mode

These bits control the counting sequence of the counter, the source for the maximum (TOP) counter value, and what type of Waveform Generation to be used. Modes of operation supported by the Timer/Counter unit are: Normal mode, Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes.

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of Operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

• Bit 5:4 – COM01:0: Compare Match Output Mode

These bits control the Output Compare pin (OC0) behavior. If one or both of the COM01:0 bits are set, the OC0 output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0 pin must be set in order to enable the output driver.

When OC0 is connected to the pin, the function of the COM01:0 bits depends on the WGM01:0 bit setting.

Below table shows the COM01:0 bit functionality when the WGM01:0 bits are set to a normal or CTC mode (non-PWM).

Table: Compare Output Mode, non-PWM Mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Below table shows the COM01:0 bit functionality when the WGM01:0 bits are set to fast PWM mode.

Table: Compare Output Mode, Fast PWM Mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Reserved
1	0	Clear OC0 on compare match, set OC0 at BOTTOM, (nin-inverting mode)
1	1	Set OC0 on compare match, clear OC0 at BOTTOM, (inverting mode)

Below table shows the COM01:0 bit functionality when the WGM01:0 bits are set to phase correct PWM mode.

Table: Compare Output Mode, Phase Correct PWM Mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Reserved
1	0	Clear OC0 on compare match when up-counting. Set OC0 on compare match when downcounting.
1	1	Set OC0 on compare match when up-counting. Clear OC0 on compare match when downcounting.

• **Bit 2:0 – CS02:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter.

Table: Clock Select Bit Description

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.

2. Timer/Counter Register – TCNT0

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	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the compare match on the following timer clock. Modifying the counter (TCNT0) while the counter is running, introduces a risk of missing a compare match between TCNT0 and the OCR0 Register.

3. Output Compare Register – OCR0

Bit	7	6	5	4	3	2	1	0	
	OCR0[7:0]								OCR0
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	

The Output Compare Register contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0 pin.

4. Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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 1 – OCIE0: Timer/Counter0 Output Compare Match Interrupt Enable

When the OCIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Compare Match interrupt is enabled. The corresponding interrupt is executed if a compare match in Timer/Counter0 occurs, that is, when the OCF0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

• Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, that is, when the TOV0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

5. Timer/Counter Interrupt Flag Register– TIFR

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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 1 – OCF0: Output Compare Flag 0

The OCF0 bit is set (one) when a compare match occurs between the Timer/Counter0 and the data in OCR0 – Output Compare Register0. OCF0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0 is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0 (Timer/Counter0 Compare Match Interrupt Enable), and OCF0 are set (one), the Timer/Counter0 Compare Match Interrupt is executed.

• Bit 0 – TOV0: Timer/Counter0 Overflow Flag
The bit TOV0 is set (one) when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE0 (Timer/Counter0 Overflow Interrupt Enable), and TOV0 are set (one), the Timer/Counter0 Overflow interrupt is executed. In phase correct PWM mode, this bit is set when Timer/Counter0 changes counting direction at $00.

• Timer0 Modes of Atmega32:

1. Normal Timer: Generate exact delay time

Timer0 registers value calculations to generate a specific delay:
Assume we want to generate **250ms** delay with **Timer0**. Assume the microcontroller oscillator frequency is **1MHz**. We want to use pre-scaler **1024**.
Therefore, the value of three clock select bits are **CS02=1, CS01=0, CS00=1**.

The applied clock frequency of the Timer0 will be

$$Clock\ Frequency\ of\ Timer0 = \frac{Oscillator\ Frequency\ of\ the\ Microcontroller}{Applied\ Prescaler}$$

Therefore, for this case the **Clock Frequency of Timer0**= 1000000/1024=**976.5625Hz**
Time Period= 1/976.5625= 0.001024sec= **1.024ms.** This Time Period is also called **Clock Tick**.
Hence, for this settings we cannot measure below 1.024ms.
To generate 250ms delay, the **Number of Timer Cycles** needed,

$$No.of\ Timer\ Cycles = \frac{Time\ Delay\ Required}{Clock\ Tick}$$

Therefore, for this case the Number of Timer Cycles= 250/1.024=244.14= **244 cycles**

Since, we are using Timer0 as Normal Timer mode so Overflow flag will be checked. To overflow the register TCNT0 after 244 cycles, the initial value of the TCNT0 will be

TCNT0= 256-244= 12

First we will complete this experiment through Polling Method then we will complete same experiment with Interrupt Method.

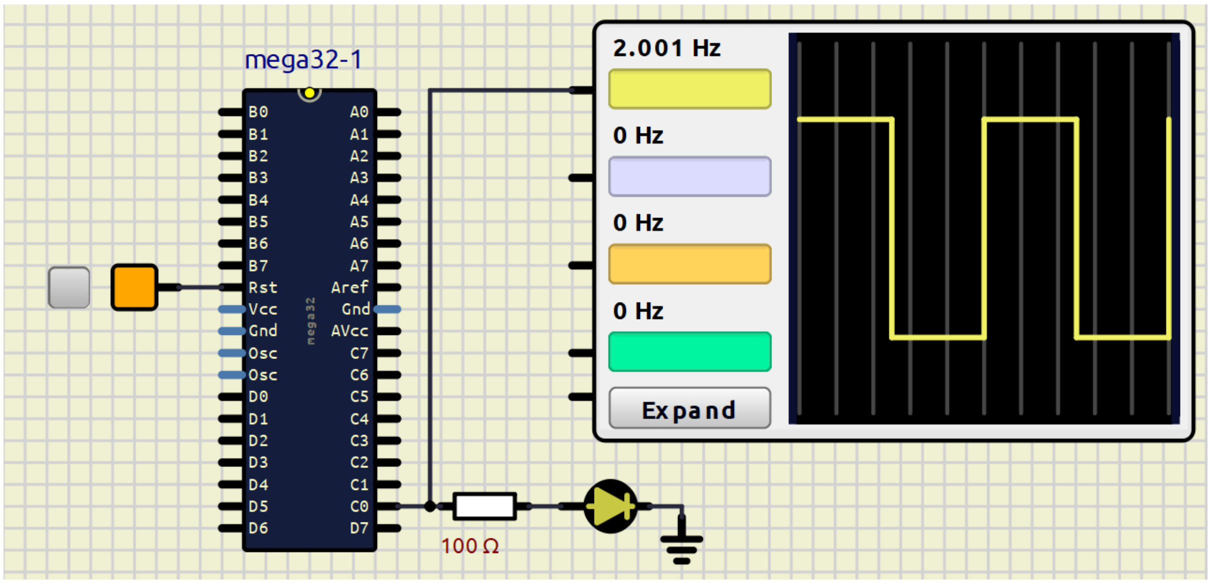
- **For Polling Method:** We will continuously check **TOV0** bit in **TIFR** register.
- **For Interrupt Method:** We set **TOIE0** bit in **TIMSK** and **I-bit** in **SREG**. The corresponding interrupt service routine will be executed.

Experiment 1: Create 250ms delay with Timer0 with polling method. MCU Oscillator Frequency=1MHz

Calculation is same as above.

```
/* Normal Mode Timer0 Programming for delay routine through polling method.
Create 250ms delay with Timer0. MCU Oscillator Frequency= 1MHz. */

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Data Direction Register for LED
SBI DDRC,PINC0
CBI PORTC,PINC0 // At the begining LED turn off

// Initialization Timer0
LDI R16,0x05  // Normal Mode and timer clock frequency= oscillator frequency/1024
OUT TCCR0,R16

Infinity_Loop:              SBI PORTC,PINC0
                            CALL Delay
                            CBI PORTC,PINC0
                            CALL Delay
                            JMP Infinity_Loop

Delay:         LDI R17,12
               OUT TCNT0,R17
               LDI R17,0x01
               OUT TIFR,R17
               LOOP:  IN R18,TIFR
                            AND R18,R17
                            BREQ LOOP
               RET
```

Simulation on SimulIDE:



Verify the calculated frequency with simulated frequency. Make the above circuit with hardware and verify the frequency with DSO.

Experiment 2: Create 500ms delay with Timer0 polling method. MCU Oscillator Frequency=8MHz.

We want to use pre-scaler **1024**.
Therefore, the value of three clock select bits are **CS02=1, CS01=0, CS00=1**.
Therefore, for this case the **Clock Frequency of Timer0**= 8000000/1024=**7812.5Hz**
Time Period= 1/7812.5= 0.000128sec= **0.128ms.** This Time Period is also called **Clock Tick**.
To generate 500ms delay, the **Number of Timer Cycles**= 500/0.128= 3906.25= 3906 cycles.

Since Timer0(8-bit timer) can count maximum upto 255. We have to make 3906 as multiplication of two integers which are less than equal to 255.

Therefore, we can write **3906= 217*18**

We will use Timer0 to count **217 clocks for 18times**.

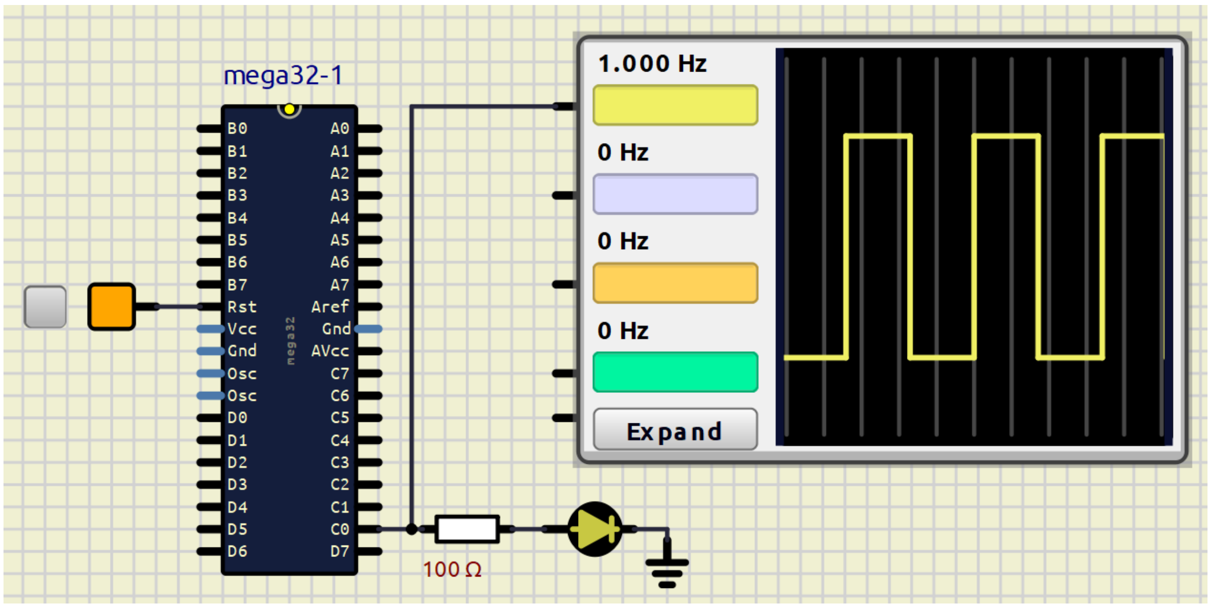
Since, we are using Timer0 as Normal Timer mode so Overflow flag will be checked. To overflow the register TCNT0 after 244 cycles, the initial value of the TCNT0 will be

TCNT0= 256-217= 39

```
/* Normal Mode Timer0 Programming for delay routine through polling method.
Create 500ms delay with timer0. MCU Oscillator Frequency=8MHz */

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Data Direction Register for LED
SBI DDRC,PINC0
CBI PORTC,PINC0 // At the begining LED turn off

// Initialization Timer0
LDI R16,0x05  // Normal Mode and timer clock= oscillator frequency/1024
OUT TCCR0,R16

Infinity_Loop:              SBI PORTC,PINC0
                            CALL Delay
                            CBI PORTC,PINC0
                            CALL Delay
                            JMP Infinity_Loop

Delay:        LDI R17,39
              OUT TCNT0,R17
              LDI R18,0x01
              OUT TIFR,R18

              LDI R16,18

              LOOP:         IN R19,TIFR
                            AND R19,R18
                            BREQ LOOP
                            DEC R16
```

```
                                          OUT TCNT0,R17
                                          OUT TIFR,R18
                                          BRNE LOOP
                                   RET
```

Simulation on SimulIDE:



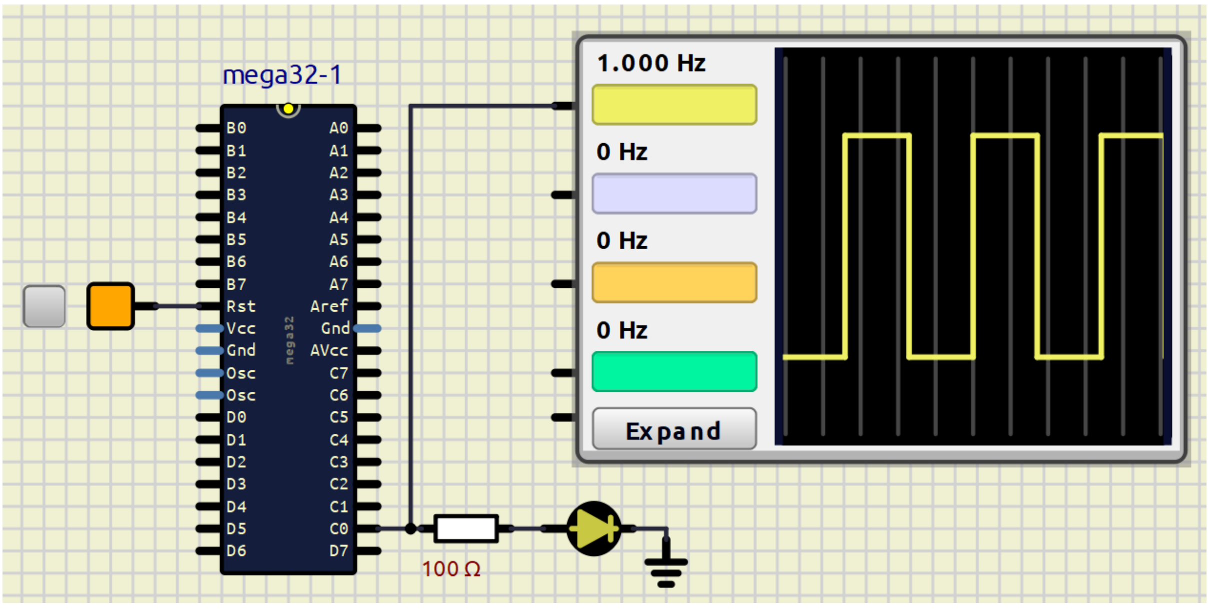
Verify the calculated frequency with simulated frequency. Make the above circuit with hardware and verify the frequency with DSO.

Experiment 3: Create 500ms delay with Timer0 Interrupt method. MCU Oscillator Frequency=8MHz.

```
// Timer0 using Interrupt

.INCLUDE "M32DEF.INC"
.ORG 0X0000
       JMP MAIN
.ORG 0x0016
       JMP Timer0_Overflow

MAIN:         LDI R16,HIGH(RAMEND)
              OUT SPH,R16
              LDI R16,LOW(RAMEND)
              OUT SPL,R16

// Data Direction Register
SBI DDRC,PINC0
LDI R19,0x00
OUT PORTC,R19

//Timer0 Initialization
LDI R16,0x01
OUT TIMSK,R16
LDI R16,0x05
OUT TCCR0,R16
LDI R17,39
OUT TCNT0,R17
SEI
LDI R16,18
Infinity_LOOP:              NOP
```

```
                            JMP Infinity_LOOP

Timer0_Overflow:             DEC R16
                             BRNE JUMP
                             COM R19
                             OUT PORTC,R19
                             LDI R16,18
                             JUMP: OUT TCNT0,R17
                             RETI
```

Simulation on SimulIDE:



Verify the calculated frequency with simulated frequency. Make the above circuit with hardware and verify the frequency with DSO.

Experiment 4: UART Bit-Banging (Software UART)

Data Frame of UART Communication is

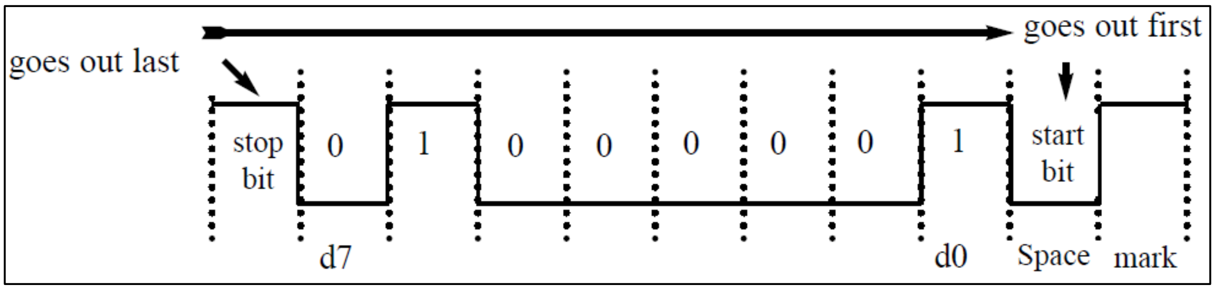


Fig. UART Transmission of ASCII 'A'

The ASCII character "A" (8-bit binary 0100 0001 and hex 0x41) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

When there is no transfer, the signal is 1 (high), which is referred to as mark. This 'mark' is called Idle state. The 0 (low) is referred to as space. This 'Space' is called start bit. Notice that the transmission begins with a start bit (space) followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

We want to generate this data frame using any GPIO pin of the ATmega32. This method of generating a UART data frame with GPIO is known as **UART bit-banging**. Since we are creating this data frame through programming, it is also referred to as **software UART**.

Note: To install the required software and check the corresponding hardware details, please refer to the lab manual Exp9A.

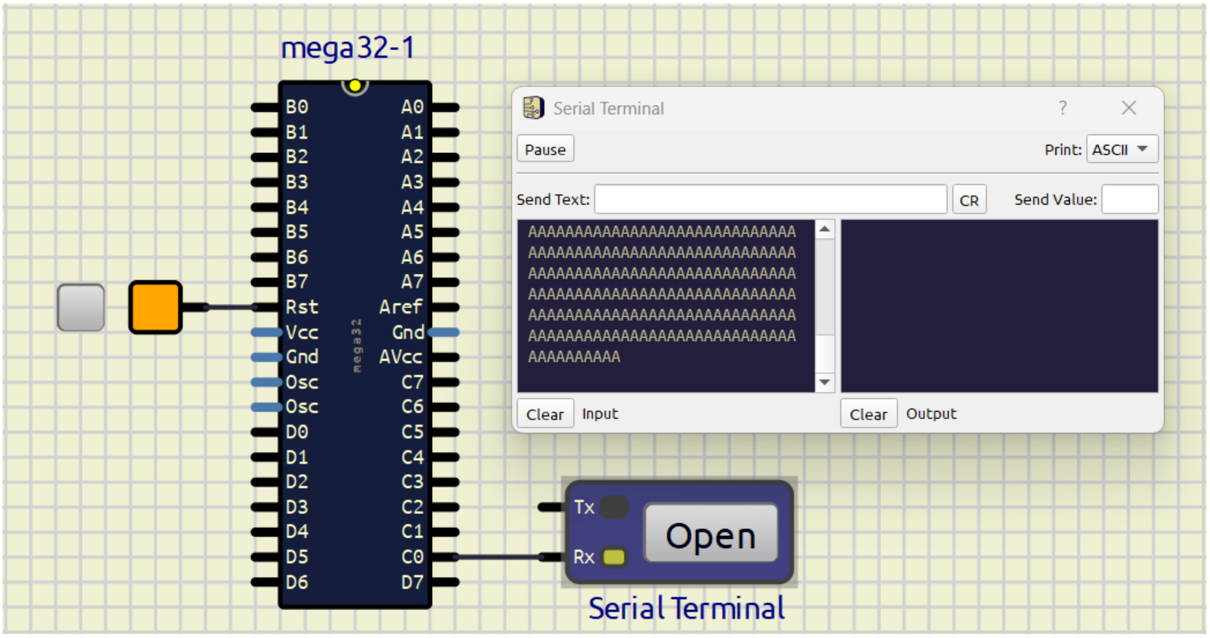
- **UART Transmitter Bit-Banging:**

We want to transfer data at a baud rate of 9600. Therefore, each data bit stays on the GPIO for 1000000/9600=104.167us= approximately 104 microseconds.

Implement a UART bit-banging transmitter on PORTC0, operating at a baud rate of 9600 with one stop bit and no parity bit. The ATmega32 microcontroller runs at an 8 MHz oscillator frequency. The transmitter will continuously send the character 'A' in an infinite loop.

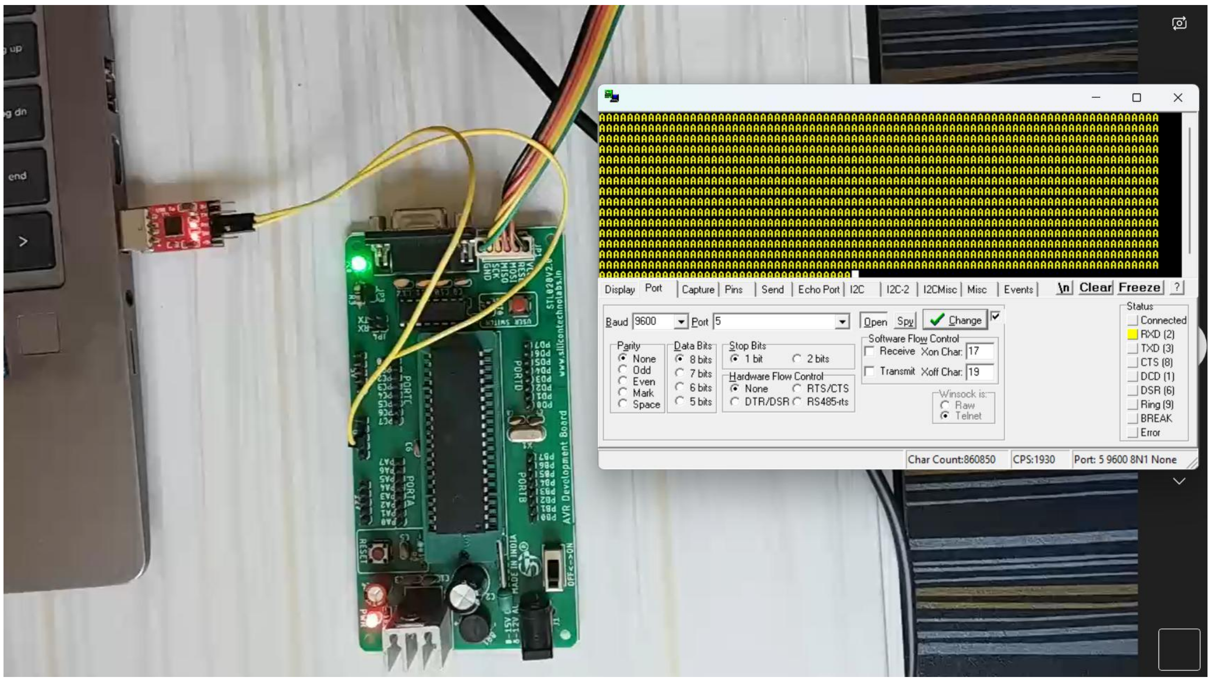
```
/* UART Bit-Banging Tx through PORTC0 with 9600 Baud. MCU Oscillator= 8MHz*/

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Tx is always output pin
SBI DDRC,PINC0
//Make it as Idle state
SBI PORTC,PINC0

// Set Timer0 registers to create delay with respect to 9600 baud rate
LDI R16,0x02
OUT TCCR0,R16

UART_LOOP:          LDI R16,'A' //Data
                    CBI PORTC,PINC0 //Start Bit
                    CALL UART_BAUD
                    LDI R19,8 // For 8-bit data shifting
                    ROTATE:             OUT PORTC,R16
                                        CALL UART_BAUD
                                        LSR R16
                                        DEC R19
                                        BRNE ROTATE
                    // Stop Bit
                    SBI PORTC,PINC0
                    CALL UART_BAUD
                    JMP UART_LOOP

//Register overflow & polling method for UART Baud rare
UART_BAUD:          LDI R17,152
                    OUT TCNT0,R17
                    LDI R17,0x01
                    OUT TIFR,R17
                    LOOP:         IN R18,TIFR
                                  AND R18,R17
                                  BREQ LOOP
                    RET
```

Simulation on SimulIDE:


mega32-1
B0 B1 B2 B3 B4 B5 B6 B7 Rst Vcc Gnd Osc Osc D0 D1 D2 D3 D4 D5 D6
A0 A1 A2 A3 A4 A5 A6 A7 Aref Gnd AVcc C7 C6 C5 C4 C3 C2 C1 C0 D7
Serial Terminal
Pause
Print: ASCII
Send Text:
CR
Send Value:
Clear
Input
Clear
Output
Tx
Rx
Open
Serial Terminal

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
```

Make the below hardware to verify the UART transmitted data on Realterm.


Display
Port
Capture
Pins
Send
Echo Port
I2C
I2C-2
I2CMisc
Misc
Events
\n
Clear
Freeze
?
Baud 9600
Port 5
Open
Spy
Change
Parity None Odd Even Mark Space
Data Bits 8 bits 7 bits 6 bits 5 bits
Stop Bits 1 bit 2 bits
Hardware Flow Control None RTS/CTS DTR/DSR RS485-rts
Software Flow Control Receive Xon Char: 17 Transmit Xoff Char: 19
Winsock is: Raw Telnet
Status Connected RXD (2) TXD (3) CTS (8) DCD (1) DSR (6) Ring (9) BREAK Error
Char Count:860850
CPS:1930
Port: 5 9600 8N1 None

Class Assignment 1: Implement a UART bit-banging transmitter on PORTC2, configured for a baud rate of 9600, with one stop bit and no parity bit. The ATmega32 microcontroller operates at an 8 MHz oscillator frequency. The transmitter will continuously send the character 'E' in an endless loop.

- **UART Receiver Bit-Banging:**

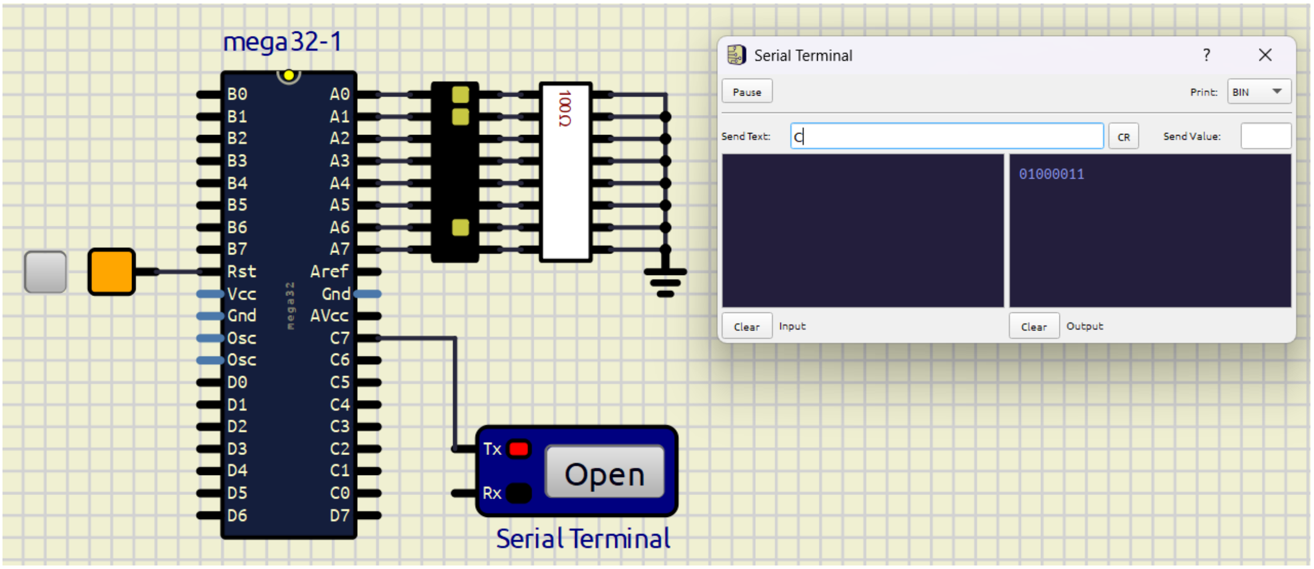
We want to receive data at a baud rate of 9600. Therefore, each data bit stays on the GPIO for 1000000/9600=104.167us= approximately 104 microseconds.

Implement a UART bit-banging receiver on PORTC7, configured for a baud rate of 9600, with one stop bit and no parity bit. The ATmega32 microcontroller operates at an 8 MHz oscillator frequency. The received data will be displayed on an LED bar graph connected to PORTA.

```
/* UART Bit-Banging Rx through PORTC7 with 9600 Baud. MCU Oscillator= 8MHz.
Display the received data on a LED Bar Graph at PORTA. */

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

//DDDR of PORTA
LDI R16,0xFF
OUT DDRA,R16

// Rx is always input pin with active internal pull up resistor
CBI DDRC,PINC7
SBI PORTC,PINC7 // No data means in idle state(Logic High)

// Set Timer0 registers to create delay with respect to 9600 baud rate
LDI R16,0x02
OUT TCCR0,R16

UART_LOOP:              IN R16,PINC  // Idle state until loop breaks
                        ANDI R16,0x80
                        BRNE UART_LOOP       //If loop breaks then start bit received
                        CALL UART_BAUD
                        LDI R20,0x00 // Register to store received data
                        LDI R19,7 // For 7-bit data receiving
                        ROTATE:              IN R16,PINC
                                             ANDI R16,0x80
                                             OR R20,R16
                                             LSR R20
                                             CALL UART_BAUD
                                             DEC R19
                                             BRNE ROTATE
                        IN R16,PINC // 8th bit receive data but no shift required
                        ANDI R16,0x80
                        OR R20,R16
                        CALL UART_BAUD
                        // Stop Bit
                        CALL UART_BAUD
                        OUT PORTA,R20
                        JMP UART_LOOP

//Register overflow & polling method for UART Baud rare
UART_BAUD:              LDI R17,152
                        OUT TCNT0,R17
                        LDI R17,0x01
                        OUT TIFR,R17
                        LOOP:         IN R18,TIFR
```

```
                          AND R18,R17
                          BREQ LOOP
                RET
```

Simulation on SimulIDE: Set the print type to BIN on the serial terminal to display the binary representation of the corresponding entered character. The LED bar graph can be used to verify this binary sequence.

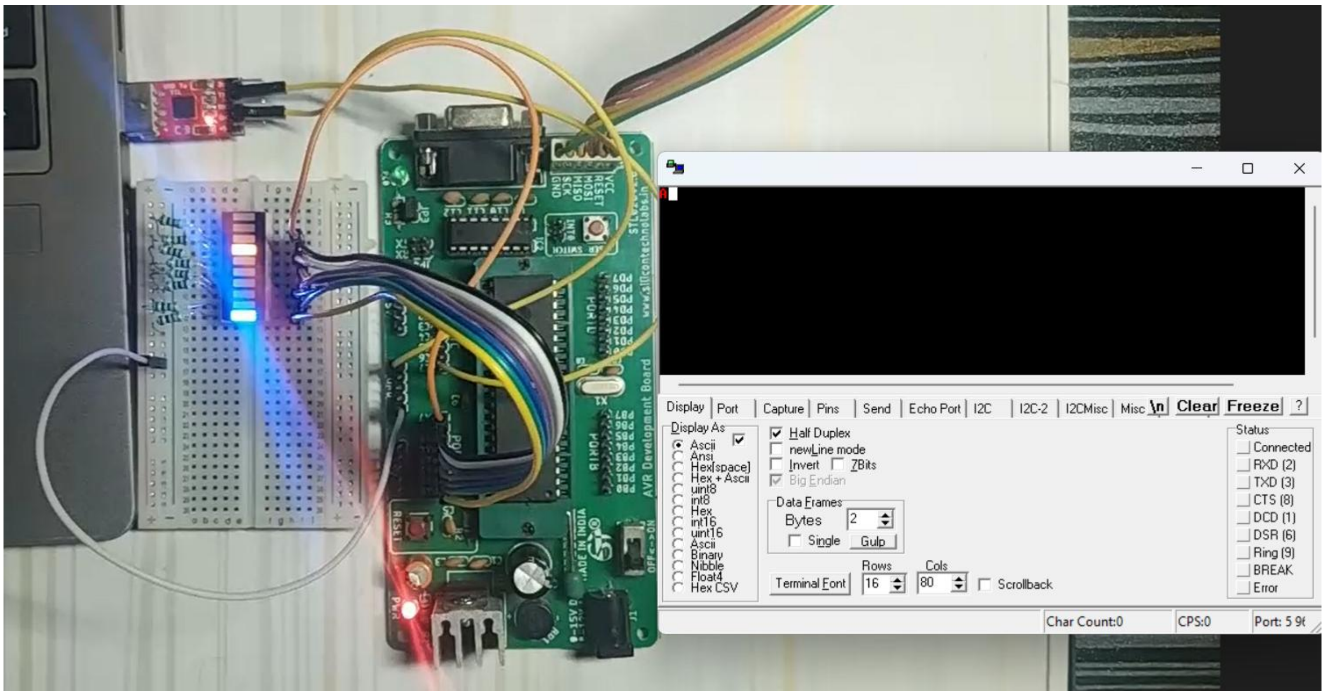


Send Text: c

```
01000011
```

Make the below hardware to verify the UART received data on LED bar graph.

Choose the correct COM port and set the baud rate with no parity and one stop bit. Navigate to the Display menu and select 'Half Duplex' to monitor the character entered from the keyboard. Finally, click 'Open' to start the receiver operation.



Class Assignment 2: Implement a UART bit-banging receiver on PORTC5, configured for a baud rate of 9600, with one stop bit and no parity bit. The ATmega32 microcontroller

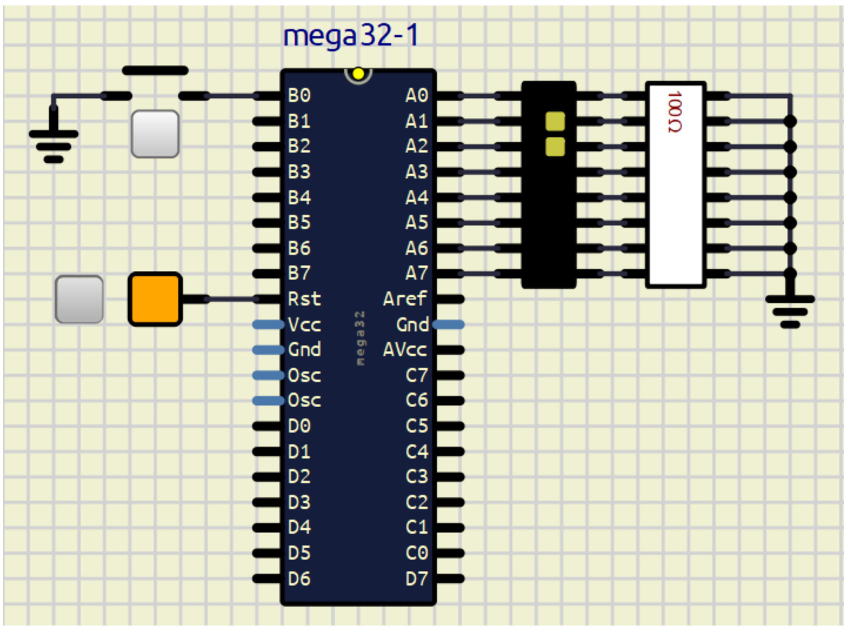
operates at an 8 MHz oscillator frequency. The received data will be displayed on an LED bar graph connected to PORTA.

2. Timer0 as Counter Mode:

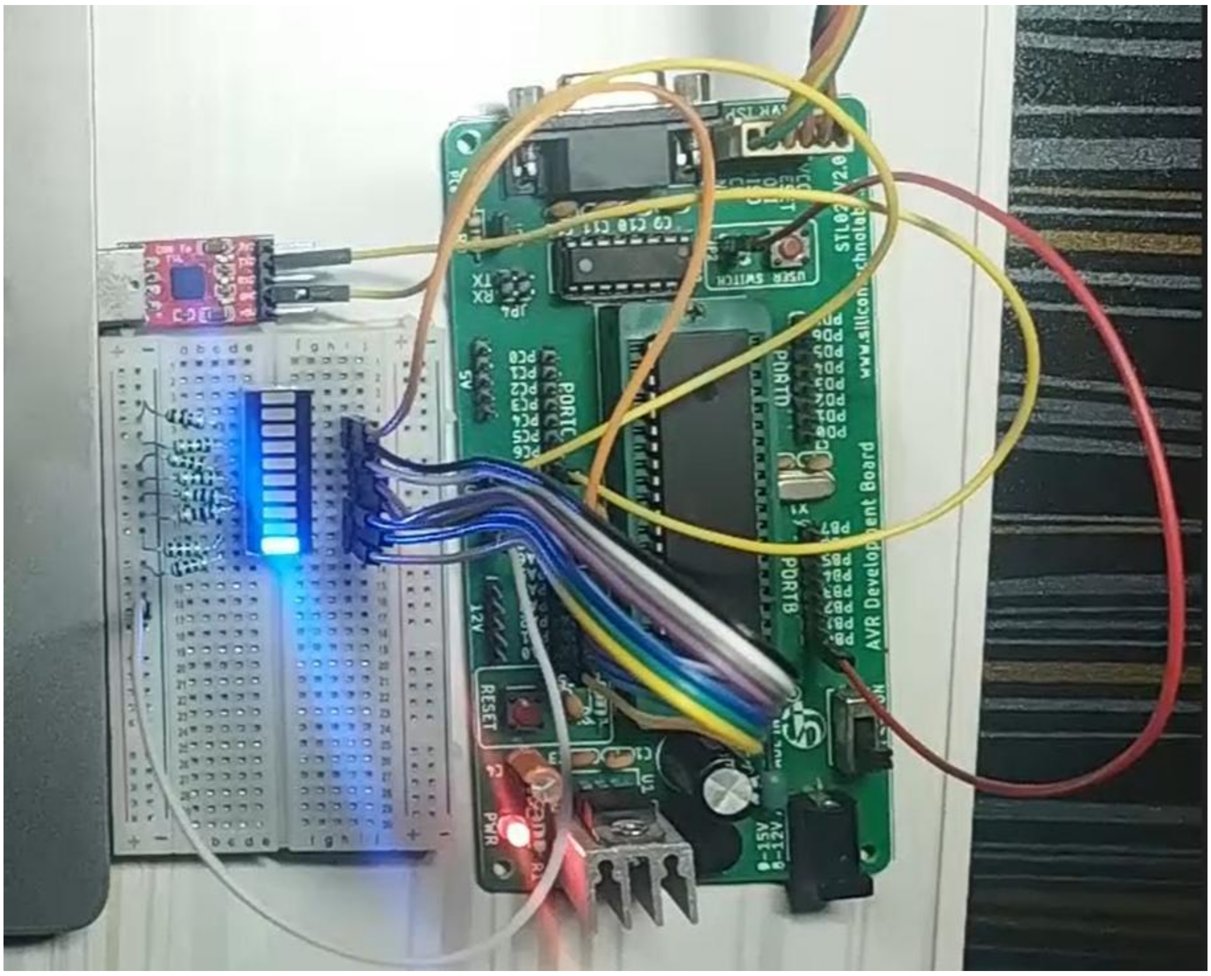
Experiment 5: In this experiment, Timer0 is configured in Counter Mode to count external pulses. The value stored in the TCNT0 register is then visualized on an LED bar graph connected to PORTA, allowing real-time monitoring of the counter's progress.

```
/* Timer0 as counter mode.
Display the counter value(TCNT0) on a LED Bar Graph at PORTA. */

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

//DDDR of PORTA
LDI R16,0xFF
OUT DDRA,R16

// Configure counter mode pin T0 as input pin with internal pull up
CBI DDRB,PINB0
SBI PORTB,PINB0

// Set Timer0 registers as counter mode to detect falling edge
LDI R16,0x06
OUT TCCR0,R16
// Reset TCNT0 register
LDI R16,0x00
OUT TCNT0,R16

LOOP:  IN R17,TCNT0
              OUT PORTA,R17
              JMP LOOP
```

Simulation on SimulIDE:



Make the below hardware to verify the counter through LED bar graph.
If an external tactile switch on the breadboard is not preferred, the built-in user switch on the development board can serve as an alternative for input control.



3. Timer0 as Output Compare (CTC: Clear Timer on Compare Match) Mode:

- CTC mode stands for "Clear Timer on Compare Match." This mode is specifically used for generating square waves of different frequencies and duty cycles.
- In all the previous programs for timers and counters, it was observed that the timer register resets only when an overflow condition occurs. However, in CTC mode, the timer can be reset even if an overflow doesn't occur.
- Here, the timer will be reset only when the count value exceeds the value present in the compare match register.
- The pins of the ATMEGA32 MCU used for observing CTC mode waveforms are OC0, OC1A, OC1B, and OC2.

Experiment 6: To toggle OC0 using CTC mode of Timer0, set the comparison value OCR0 = 99. This configuration allows the timer to reset when the count reaches 99, thereby generating a square wave with a specific frequency on the OC0 pin. Oscillation frequency MCU is 8MHz and pre scaler is 8. Calculate toggling frequency at OC0.

Here, the oscillation frequency=8MHz. Pre scaler used= 8.

Therefore, the clock frequency of the timer0= 8MHz/8= 1MHz.
The time period of the timer clock= 1us.

The comparison value OCR0=99. Therefore, to loop between 99 to 99, the 100 clock will be required.
Therefore, OC0 remains logic high for 100x1us=100us and logic low for 100us.
Hence, the time period of the generated square wave at OC0= 200us.
Therefore, the frequency of the generated square wave= 5KHz.

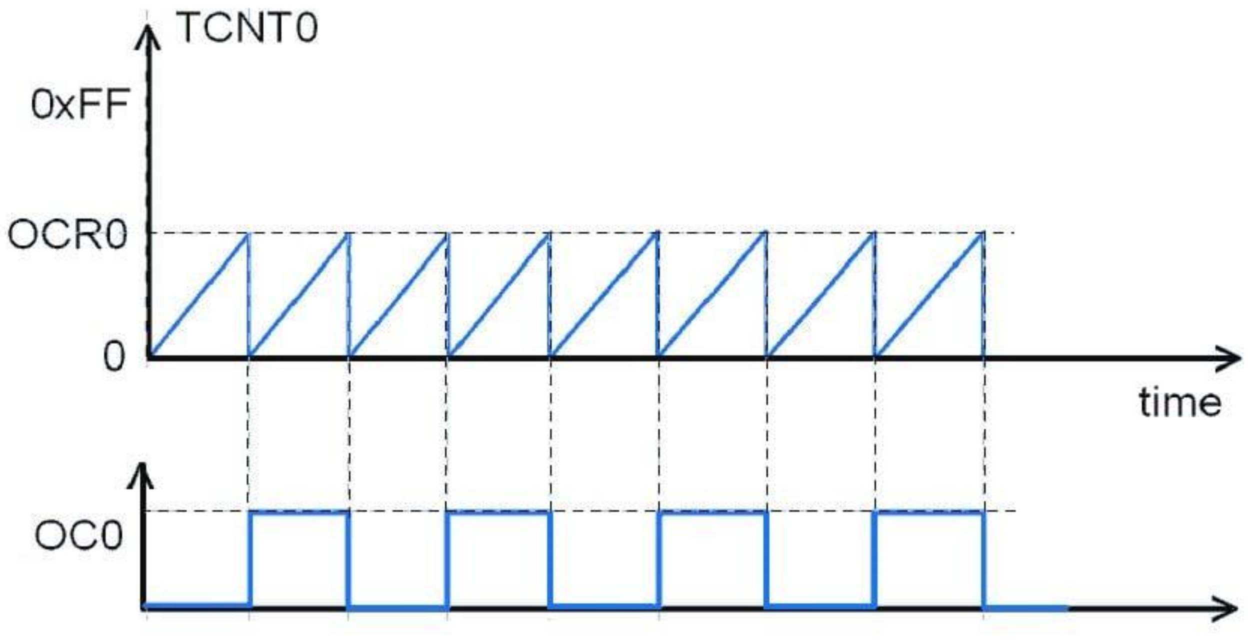
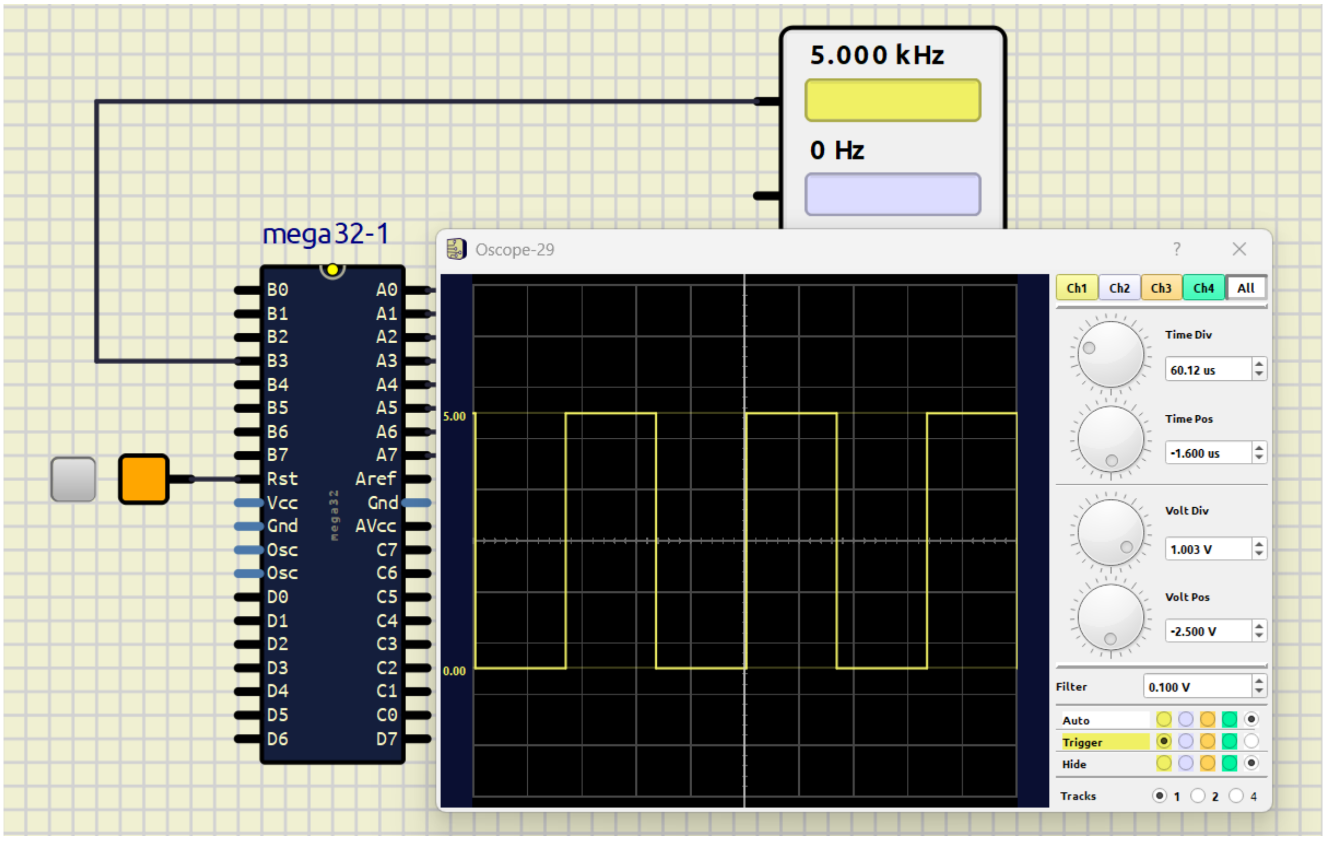


Fig: Waveform Generation Using CTC Mode

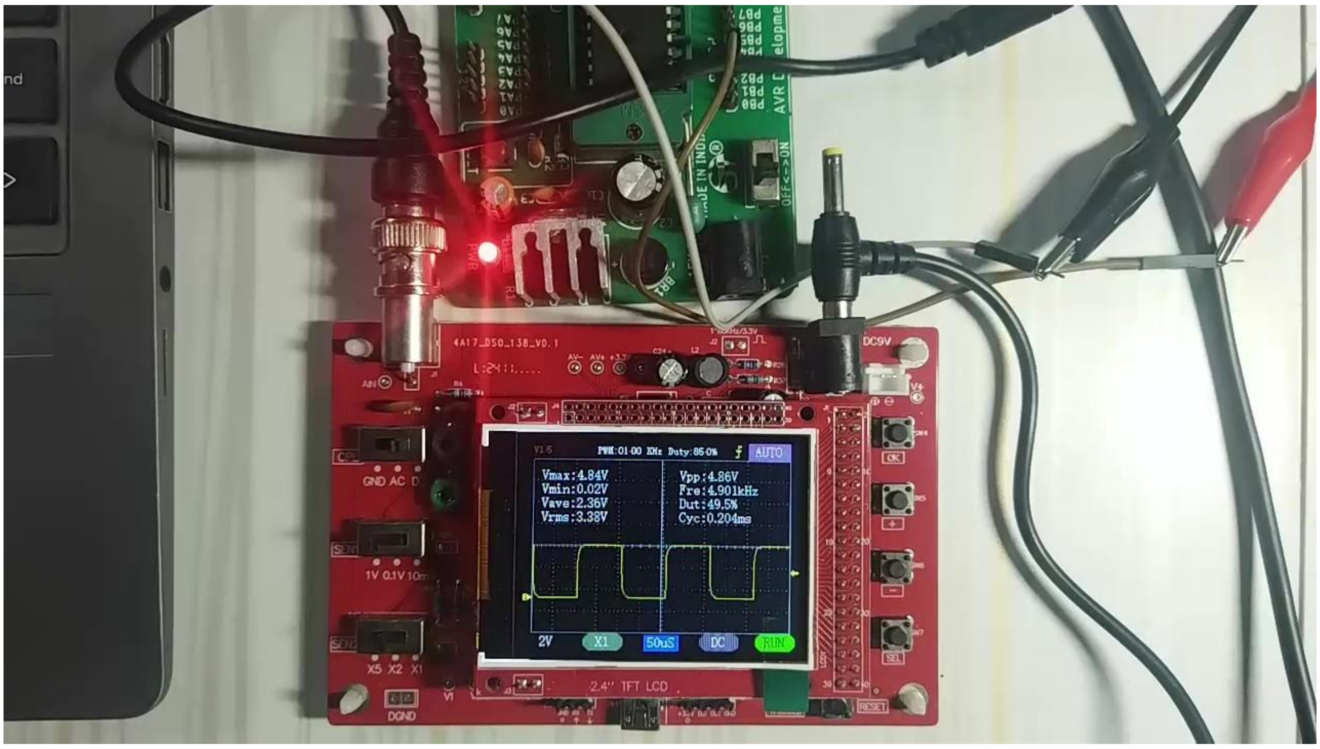
```
/* Clear Timer on Compare Match Mode */

.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Configure CTC Mode pin (OC0)
SBI DDRB,PINB3

// Set Timer0 registers as CTC mode and Toggle OC0 on compare match
LDI R16,0x1A
OUT TCCR0,R16
// Reset TCNT0 register
LDI R16,0x00
OUT TCNT0,R16
// Load OCR0 for compare match
LDI R16,99
OUT OCR0,R16

LOOP:           NOP
                JMP LOOP
```

Simulation on SimulIDE:


5.000 kHz
0 Hz
mega32-1
B0 B1 B2 B3 B4 B5 B6 B7 Rst Vcc Gnd Osc Osc D0 D1 D2 D3 D4 D5 D6
A0 A1 A2 A3 A4 A5 A6 A7 Aref Gnd AVcc C7 C6 C5 C4 C3 C2 C1 C0 D7
Oscope-29
5.00
0.00
Ch1 Ch2 Ch3 Ch4 All
Auto
Trigger
Hide
Tracks

Time Div 60.12 us
Time Pos -1.600 us
Volt Div 1.003 V
Volt Pos -2.500 V
Filter 0.100 V

Make the below hardware to verify the frequency with DSO.



Class Assignment 3: Configure the Atmega32 microcontroller to generate a 2 kHz square wave on OC0 using CTC mode. The oscillator frequency of the microcontroller is set to 4 MHz.

4. Timer0 as PWM (Pulse Width Modulation) Mode:

Concept of PWM:

- PWM stands for Pulse Width Modulation, and it's a very important concept in embedded systems.
- This technique allows us to change the duty cycle of a square wave without affecting its overall frequency.
- In this method, the width of the pulse is varied, i.e., the on-time of the square wave is varied, and the off-time is automatically changed in a proportionate way to keep the frequency constant.
- By changing the duty cycle of the generated PWM wave, the power (P) and voltage (V) of the wave are also changed. Hence, it is generally used to control the light intensity of bulbs or LEDs, control the speed of DC motors, and operate servo motors.
- Apart from this, PWM also finds applications in power electronic devices like inverters, choppers, etc.

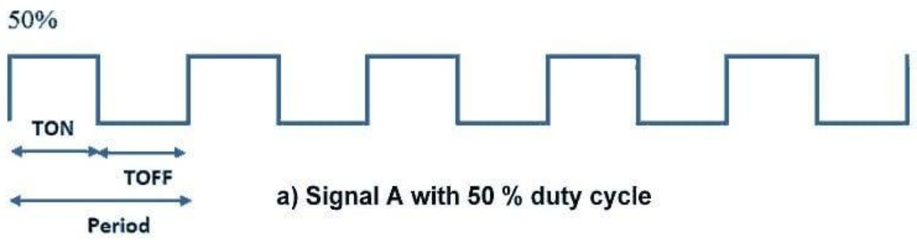
Duty Cycle:

A period of a pulse consists of an ON cycle (5V) and an OFF cycle (0V). The fraction for which the signal is ON over a period is known as the duty cycle.

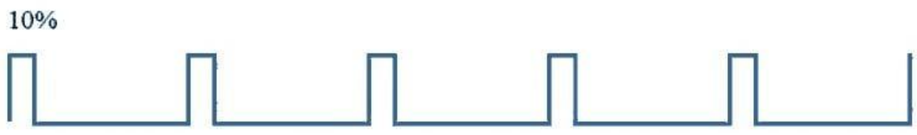
$$Duty\ Cycle(In\ \%) = \frac{T\_ON}{Total\ Period} \times 100$$

Example: Consider a pulse with a period of 10ms which remains ON (high) for 2ms.The duty cycle of this pulse will be Duty Cycle = 2ms / 10ms = 20%

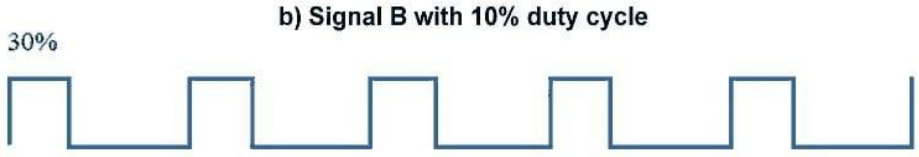
Pulse Width Modulated signals with different duty cycle are shown below:



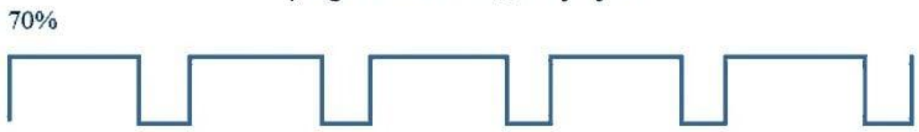
a) Signal A with 50 % duty cycle



b) Signal B with 10% duty cycle



c) Signal C with 30% duty cycle

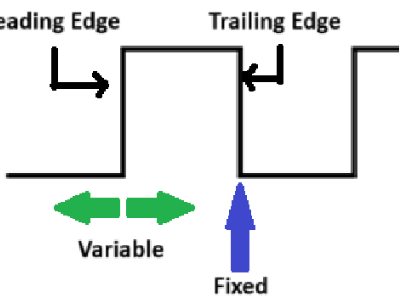


d) Signal D with 70% duty cycle

How a microcontroller generates PWM signal:

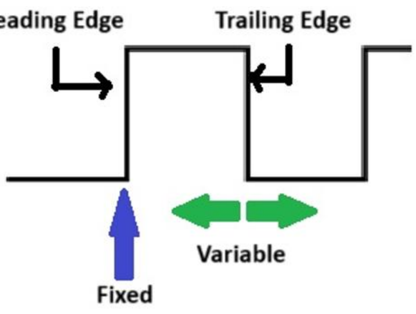
- In microcontroller-based circuits, Pulse Width Modulation (PWM) is generated using timers. The process involves a counter that increments periodically in sync with the timer's clock signal. When the counter reaches the end of the PWM period, it resets to its initial value.
- During the counting process, a reference value is set between the starting and final values of the counter. When the counter reaches this reference value, the PWM output state switches from logic 1 to logic 0.
- This technique of generating a PWM wave is known as time-proportioning. It is called so because a fixed portion of the cycle time is spent in the high state.
- The width of the pulse in PWM can be varied in three ways:

  i. The trailing edge is kept constant while the leading edge is moved.



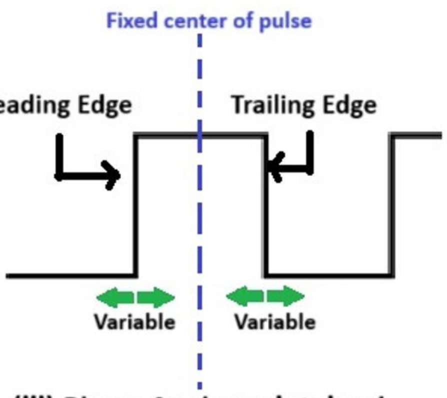
(i) Phase Angle not maintained

  ii. The leading edge is kept constant while the trailing edge is moved.



(ii) Phase Angle not maintained

  iii. Both edges are moved by equal amounts in opposite directions.



(iii) Phase Angle maintained

PWM in ATMEGA32 MCU:

- The timing proportion method is used to generate PWM waves.
- All three timers, i.e., timers 0, 1, and 2, are capable of generating PWM signals.
- The output-compare pins for each timer will be the ones where PWM output can be obtained.
- In ATMEGA32, two types of PWM are available:
  - a) Fast PWM ✓
  - b) Phase Correct PWM ✓
- In fast PWM, the pulse width is adjusted by keeping one edge fixed and the other edge moving, as discussed earlier in methods (i) and (ii).
- In phase-correct PWM, the pulse width is adjusted by moving both edges in opposite directions by the same amount, as discussed earlier in method (iii).
- In both PWM types, there is an option to choose between inverted and non-inverted output. ✓
- In this MCU, the frequency of fast PWM is twice the frequency of phase-correct PWM:

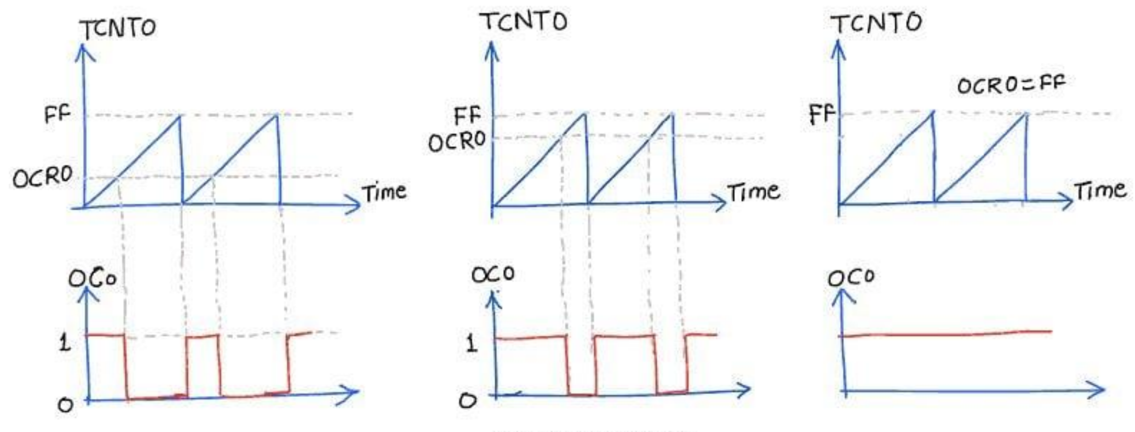
$$f_{fast-PWM} = 2 \times f_{phase-correctPWM}$$

Registers to program Timers in PWM mode:

- The WGM bits in the TCCR register are used to select the PWM type, whether fast or phase-correct.
- The COM bits in the TCCR register are used to choose between inverted and non-inverted PWM output.
- The CS02, CS01, CS00 bits in the TCCR register are responsible for selecting the pre-scaler for the timer clock signal.
- The TCNT register is responsible for maintaining the count sequence as usual.
- The OCR register holds the reference value, after which the PWM pin will change its state.

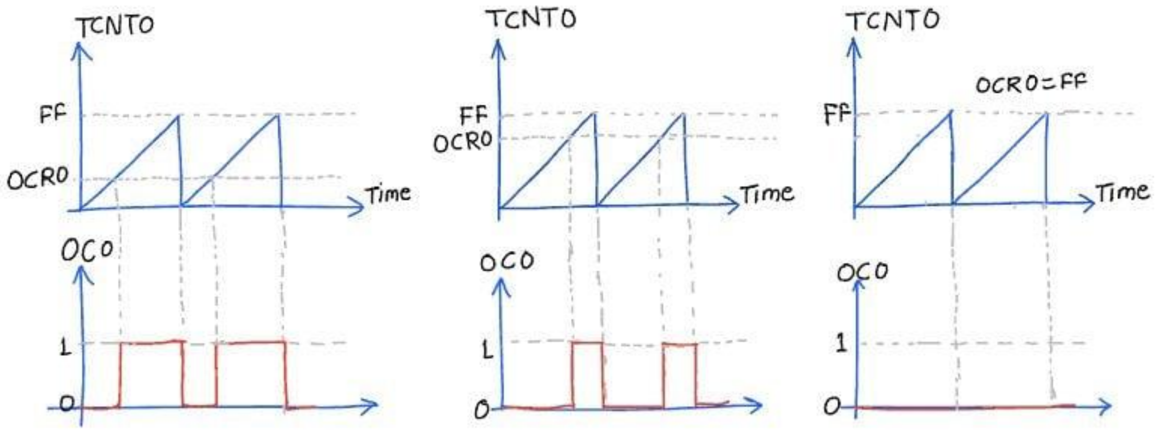
➢ Fast PWM:

- To set Fast PWM mode, we have to set WGM00: 01= 11. To generate a PWM waveform on the OC0 pin, we need to set COM01:00= 10 or 11.
- COM01:00= 10 will generate Noninverting PWM output waveform.



Non-Inverted Fast PWM

- COM01:00= 11 will generate Inverting PWM output waveform.



Inverted Fast PWM

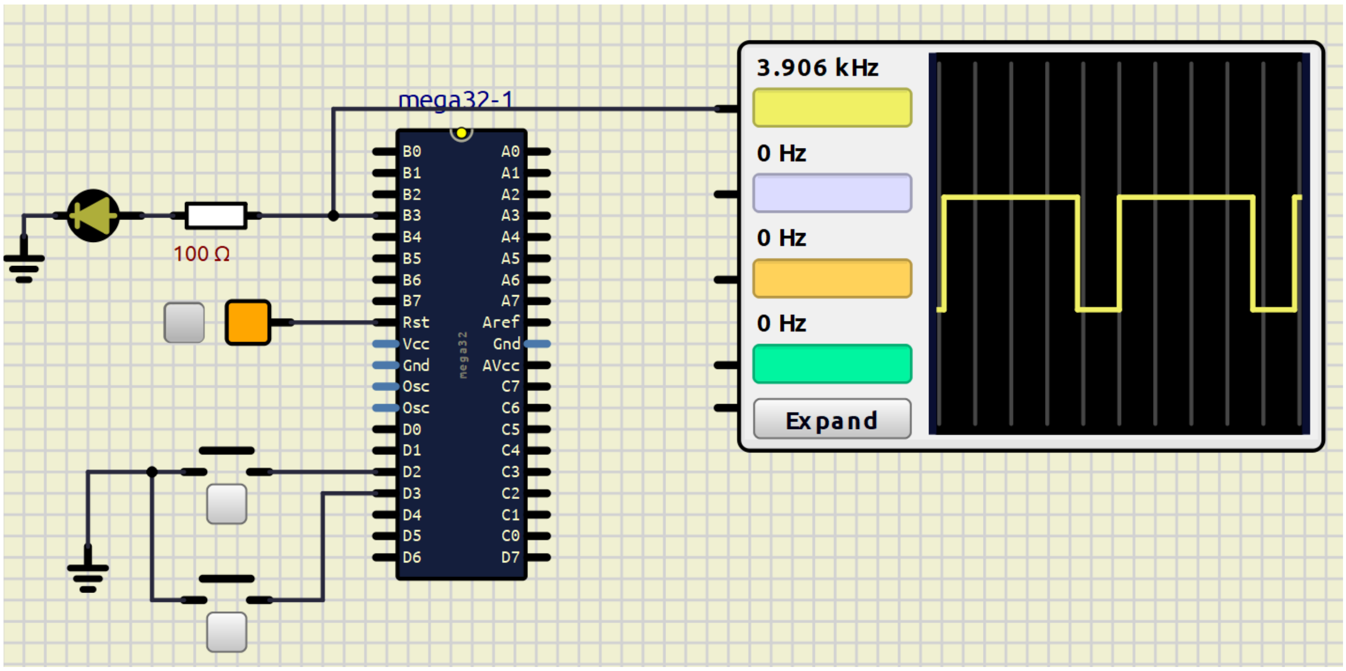
- The advantage of using PWM mode in AVR is that it is an inbuilt hardware unit for waveform generation and once we set the PWM mode and duty cycle, this unit starts generating PWM and the controller can do other work.

Experiment 7: Write an assembly program for Atmega32 to Generate Fast PWM using Timer0 and change the duty cycle with two push buttons (Increment & Decrement).

```
// Fast PWM using Timer0
.INCLUDE "M32DEF.INC"
.ORG 0x0000
       JMP MAIN
.ORG 0x0002
       JMP INT0_ISR
.ORG 0x0004
       JMP INT1_ISR

MAIN:          LDI R16,HIGH(RAMEND)
               OUT SPH,R16
               LDI R16,LOW(RAMEND)
               OUT SPL,R16

SBI DDRB,PINB3 // PWM pin data direction
CBI PORTB,PINB3

CBI DDRD,PIND2 //Input Pin
CBI DDRD,PIND3
SBI PORTD,PIND2 //Internal Pull-up
SBI PORTD,PIND3

// Interrupt Initialization
SEI
LDI R16,0xC0
OUT GICR,R16
LDI R16,0x0A
OUT MCUCR,R16

LDI R16,0x69   // PWM Initialization
OUT TCCR0,R16
LDI R16,64    // Duty cycle selection
OUT OCR0,R16
```

```
LDI R17,10 // For incrementing or decrementing step of duty cycle

Infinite_Loop:              NOP
                            JMP Infinite_Loop

INT0_ISR:            ADD R16,R17
                     OUT OCR0,R16
                     RETI
INT1_ISR:            SUB R16,R17
                     OUT OCR0,R16
                     RETI
```

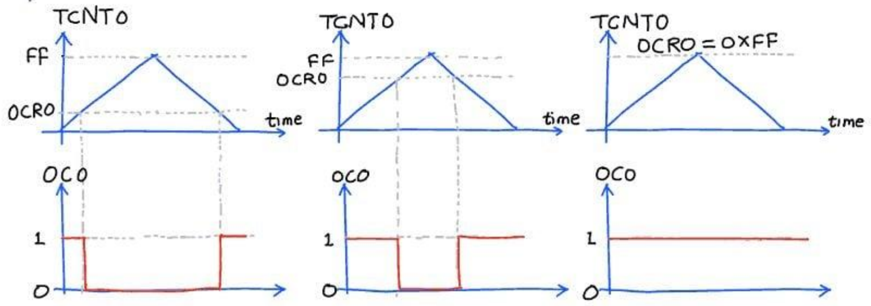
Simulation on SimulIDE:



Make the above circuit in hardware to verify the PWM & frequency with DSO. Connect one LED to check the change of intensity with PWM.

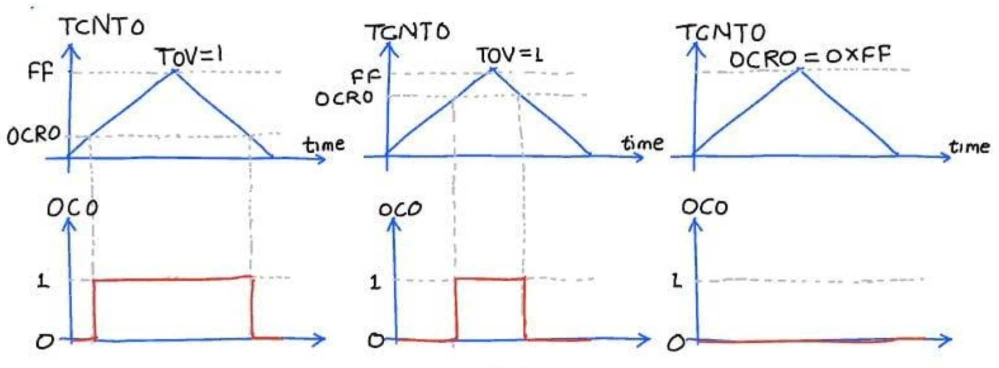
➢ Phase Correct PWM:

- To set Phase Correct PWM mode, we have to set WGM00: 01= 01. To generate a PWM waveform on the OC0 pin, we need to set COM01:00= 10 or 11.
- COM01:00= 10 will generate Noninverting PWM (Clear OC0 on compare match when up-counting. Clear OC0 on compare match when up-counting.) output waveform.



Non-Inverted Phase correct PWM

- COM01:00= 11 will generate Inverting PWM (Set OC0 on compare match when up-counting. Clear OC0 on compare match when down counting.) output waveform.



Inverted Phase correct PWM

Class Assignment 4: Write an assembly program for Atmega32 to Generate Phase Correct PWM using Timer0 and change the duty cycle with two push buttons (Increment & Decrement).