Module 2 Develop Programs for AVR Microcontrollers in C
M2.01 Examine C data types for the AVR microcontroller.
M2.02 Develop C programs for time delay and I/O operations.
M2.03 Develop C programs for logic and arithmetic operations.
M2.04 Develop C programs for data conversion and data serialisation.
M2.05 Describe the Normal and CTC mode of Timers
M2.06 Develop C Programs to generate time delays and count events using Timer/Counter.
M2.07 Explain Interrupts in AVR
M2.08 Illustrate Interrupt Programming in C

Contents:
AVR Programming in C – data types, C programs to generate time delay, I/O Programming, logic and arithmetic operations, Data Conversion, Data Serialisation, Memory Allocation.
Timer and Counter: Timers and their associated registers, Normal and CTC mode, Programming Timers in C, Counter Programming in C.
Interrupts : AVR Interrupts, ISR, Steps executing an Interrupt, Sources of interrupts, Enabling and disabling Interrupts, Interrupt priority, Interrupt Programming in C.

AVR programming in C

major reason for writing programs in c instead of assembly language is
1. it is easier and less time consuming
2. easier to modify and update
3. you can use function library codes.
4. Is portable.

C data type for the AVR C

Data type	Size in Bits	Data Range/Usage
Unsigned char	8-bit	0 to 255
char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65,535
int	16-bit	-32,768 to +32,767
unsigned long	32-bit	0 to 4,294,967,295
long	32-bit	-2,147,483,648 to +2,147,483,648
float	32-bit	+-1.175e-38 to +-3.402e38
double	32-bit	+-1.175e-38 to +-3.402e38

Unsigned char
- The unsigned char is an 8-bit data type the value range of 00-FFH(0-255).
- Most widely used data type.
- avoid the use of int if possible, because AVR microcontroller have limited registers and data RAM locations.

- By default C compliers use signed char.

Signed char

- Is an 8-bit data type that use most significant bit to represent - or + value.
- We have only 7 bits for the magnitude of the signed number values from -128 to +127.

Unsigned int

- Is 16-bit data type, value in the range of 0 to 65,535(0000-FFFFH).
- Used to represent 16-bit variables like memory addresses.
- Used to set counter values of more than 256.
- It takes 2 bytes of RAM.
- The misuse of int variables will result in larger hex file, slower execution of program.

Signed int

- Is a 16-bit data type that use most significant bit to represent + or - value.
- 15-bits for magnitude, range of -32,768 to +32,767.

Other data types

- AVR C compiler supports long data types, if want values greater than 16-bit.

Problem 1

write an AVR program to send values 00-FF to port B.

ans.

```
#include<avr/io.h>    //standard avr header
int main(void)
{
• unsigned char i;
• DDRB=0xFF;          //PORTB is output
• for(i=0;i<=255;i++)
•    PORTB=i;
• return 0;
• }
```

Time Delay

- There are 3 ways to create a time delay in AVR C.
  - Using a simple for loop
  - Using predefined C functions
  - Using AVR timers

1. Using a simple for loop

- In creating time delay using for loop, we must be mindful of two factors that can affect the accuracy of delay.
- The crystal frequency connected XTAL1-XTAL2 input pins is the most important factor in time delay calculation.
- Affects the time delay is the complier used to compile the C program.

2.Using predefined C functions

- Use predefined functions such as -delay-ms() and -delay-us() defined in delay.h in WinAVR.

- Drawback is probability problem, because different compliers do not use same name for delay functions.
- Overcome by using wrapper or macro function.
- Wrapper call the predefined delay function.

3.Using AVR timers

- One way to generate a time delay is to clear the counter at the start time and wait until the counter reaches a certain number.
  The content of the counter register represents how much time has elapsed.

Problem 2:

Write an AVR program to toggle all thebits of port b continuously with a 100ms delay. Assume that the system is ATmega32 with XTAL=8mhz.

Ans:-

```
#include<avr/io.h>    //standard avr header
void delay(void)
{
unsigned char i;
for(i=0;i<1000;i++);
}
int main(void)
{
DDRB=0xFF;          //PORTB is output
while(1)
{
 PORTB=0xAA;
delay();              // call delay program
PORTB=0x55;
 delay();
}
return 0;
}
```

I/O Programming in C

- All port registers of the AVR are both byte accessible and bit accessible.
- **Byte Programming**
  - To access a PORT register as a byte, we use PORTx label, where x indicate the name of register.
- Access the DDRx register , x indicate data direction of port.
- Access the PINx register, x indicate name of port.

- **Bit Programming**
  - The I/O ports of ATmega32 are bit-accessible.
  - We can access a single bit of I/O port registers.
  - For eg: PORTB.0=1; Here we set the 0th bit of PORTB is equal to 1.

    Also we can use the AND and OR bit wise operations to access a single bit of a given register.

Problem 3

Write an AVR program to get a byte of data from portb, and then send it to portc.

Ans:-

```
#include<avr/io.h>    //standard avr header
int main(void)
{
unsigned char i;
DDRB=0x00;          //PORTB is input
DDRC=0xFF;         //PORTC is output
while(1)
{
   i=PINB;              //take the value of port b to a variable
   PORTC=i;           // send the value to portc
}
return 0;
}
```

problem 4

Write an AVR program to get a byte of data from port c. if it is less than 100, send it to port b , otherwise send it to port d.

Ans:-

```
 #include<avr/io.h>    //standard avr header
int main(void)
{
unsigned char i;
DDRC=0x00;          //PORTB is input
DDRB=0xFF;         //PORTC is output
DDRD=0xFF;         //PORTD is output
while(1)
{
   i=PINC;              //take the value of port b to a variable
if(i<100)
{
   PORTB=i;           // send the value to portc
}
else
{
   PORTD=i;
}
}
return 0;

}
```

Logic Operations in C

- One of the most powerful feature of C language is its ability to perform bit manipulation.

The logic operators are

- AND(&&)
- OR(||)
- not(!)

- C also supports bit-wise operators. Those are
  - AND(&)
  - OR(|)
  - EX-OR(^)
  - Inverter(~)
  - Shift right(>>)
  - Shift left(<<)
- **eg:** 0x35&0x0f=0x05 /*ANDing*/

Bit-wise logic operators for C

AND			OR	EX-OR	Inverter
A	**B**	**A&B**	**A|B**	**A^B**	**Y=~B**
0	0	0	0	0	1
0	0	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

find the value of the following

- 0x35&0xFF= 0x05
- 0x04|0x68=0x6c
- 0x54^0x78=0x2c
- ~0x55=0xAA

Problem 5

write an AVR C program to toggle only bit 4 of port b continuously without disturbing the rest of the pins of port b.

Ans :-

```
#include<avr/io.h>    //standard avr header
int main(void)
{
unsigned char i;
DDRB=0xFF;          //PORTB is output
while(1)
{
   PORTB=PORTB|0b00010000;
PORTB=PORTB&0b11101111;
}
return 0;
}
```

problem 6

write an AVR program to monitor bit 5 of port c. If it is high, send 55h to port b,otherwise send Aah to port b.

Ans:-

```
#include<avr/io.h>    //standard avr header
int main(void)
{
unsigned char i;
DDRB=0xFF;          //PORTB is output
DDRC=0x00;
while(1)
{
   if(PINC&0b00100000)
   {
  PORTB=0x55;}
else
{
PORTB=0xAA;
}
}
return 0;
}
```

Compound assignment operators in C

- To reduce coding.

Compound assignment operator in C

Operation expression	**Abbreviated expression**	**Equal C**
And assignment	a &= b	a =a & b
OR assignment	a|=b	a =a | b

problem 7

write an AVR C program to toggle only bit 4 of port b continuously without disturbing the rest of the pins of port b.

Ans :-

```
#include<avr/io.h>    //standard avr header
int main(void)
{
unsigned char i;
DDRB=0xFF;          //PORTB is output
while(1)
{
   PORTB|=0b00010000;
PORTB&=0b11101111;
}
return 0;
}
```

Bit-wise shift operators in C

Bit-wise shift operators for C

Operation	Symbol	Format of shift operation
Shift right	>>	data>>number of bits to be shifted right
Shift left	<<	data<<number of bits to be shifted left

- eg: 0b00010000 >> 3 = 0b00000010 /* shifting right 3 times */
- to leave the generation of ones and zeros to the compiler and improve the code clarity, we use shift operations. ~(1<<5)=11101111
- write the code to generate the following numbers
  - a number that has only a one in position D7
  - a number that has only a one in position D2
  - a number that has only a one in position D4
  - a number that has only a zero in position D5
  - a number that has only a zero in position D3
  - a number that has only a zero in position D1

answers

- 1<<7
- 1<<2
- 1<<4
- ~(1<<5)
- ~(1<<3)
- ~(1<<1)

Problem 8

Write an AVR C program to toggle all the pins of Port B continuously.
(a) Use the inverting operator. (b) Use the EX-OR operator.

Solution:

(a)

```
#include <avr/io.h>                 //standard AVR header
int main(void)
{
      DDRB = 0xFF;                  //Port B is output
      PORTB = 0xAA;
      while (1)
            PORTB = ~ PORTB;        //toggle PORTB
      return 0;
}
```

(b)

```
#include <avr/io.h>                 //standard AVR header
int main(void)
{
      DDRB = 0xFF;                  //Port B is output
      PORTB = 0xAA;
      while (1)
            PORTB = PORTB ^ 0xFF;
      return 0;
}
```

Data Conversion Programs in C

- In newer microcontrollers have a real-time clock(RTC) where the time and date are kept even when power is off.
- Very often RTC provides data and time packed in BCD.
- To display them we must convert them to ASCII.
- **BCD to ASCII convertion:-** first convert it to unpacked BCD. Then the unpacked BCD is tagged with 30H(0110000).
  - e.g. packed BCD-0x29, unpacked BCD-0x02,0x09, ASCII-0x32,0x39
- **BINARY TO DECIMAL CONVERSION-**to display binary data, in ADC &RTC, we need to convert it to decimal and then to ASCII.
  - Hexadecimal format is a convenient way of representing binary data. Then convert into decimal. First divide the number by 10 and keep the remainder.

Problem

Write an AVR C program to convert ASCII digits of '4' and '7' to packed BCD and display them on PORTB.

Solution:

```
#include <avr/io.h>             //standard AVR header

int main(void)
{
   unsigned char bcdbyte;
   unsigned char w = '4';
   unsigned char z = '7';
   DDRB = 0xFF;                 //make Port B an output
   w = w & 0x0F;                //mask 3
   w = w << 4;                  //shift left to make upper BCD digit
   z = z & 0x0F;                //mask 3
   bcdbyte = w | z;             //combine to make packed BCD
   PORTB = bcdbyte;

   return 0;
}
int main(void)
{
   unsigned char x, y;
   unsigned char mybyte = 0x29;

   DDRB = DDRC = 0xFF;               //make Ports B and C output
   x = mybyte & 0x0F;                //mask upper 4 bits
   PORTB = x | 0x30;                 //make it ASCII
   y = mybyte & 0xF0;                //mask lower 4 bits
   y = y >> 4;                       //shift it to lower 4 bits
   PORTC = y | 0x30;                 //make it ASCII

   return 0;
}
```

Problem

Write an AVR C program to convert 11111101 (FD hex) to decimal and display the digits on PORTB, PORTC, and PORTD.

Solution:

```
#include <avr/io.h>                     //standard AVR header
int main(void)
{
     unsigned char x, binbyte, d1, d2, d3;
     DDRB = DDRC = DDRD =0xFF;          //Ports B, C, and D output
     binbyte = 0xFD;                    //binary (hex) byte
     x = binbyte / 10;                  //divide by 10
     d1 = binbyte % 10;                 //find remainder (LSD)
     d2 = x % 10;                       //middle digit
     d3 = x / 10;                       //most-significant digit (MSD)
     PORTB = d1;
     PORTC = d2;
     PORTD = d3;

     return 0;
}
```

Data Serialization in C

- Serializing data is a way of sending a byte of data one bit at a time through a single pin of microcontroller.
- There are 2 ways to transfer a data serially:
- Using a serial port. The programmer has very limited control over the sequence of data transfer.
- Transfer data one bit a time and control the sequence of data and spaces between them.

Problem

Write an AVR C program to send out the value 44H serially one bit at a time via PORTC, pin 3. The LSB should go out first.

Solution:

```
#include <avr/io.h>
#define serPin 3

int main(void)
{
     unsigned char conbyte = 0x44;
     unsigned char regALSB;
     unsigned char x;
     regALSB = conbyte;
     DDRC |= (1<<serPin);

    for(x=0;x<8;x++)
        {
          if(regALSB & 0x01)
              PORTC |= (1<<serPin);
         else
              PORTC &= ~(1<<serPin);
          regALSB = regALSB >> 1;
        }
        return 0;
}
```

memory allocation in c

- in AVR, we have 3 spaces to store data.
- 1. SRAm- 64kbytes space, address range is-0000-ffffH. We store temporary variable in SRAM.
- 2. Flash ROM- $mbytes space. Address range is 00000_1FFFFFH. It stores programs. It is directly under the control of Program counter.

• 3. EEPROM- it can save data when the power is off. When there is not enough code space,we can place permanent variables in EEPROM to save some code space.

	Flash	SRAM	EEPROM
ATmega 8	8K	256	256
ATmega 16	16K	1K	512
ATmega 32	32K	2K	1K
ATmega 64	64K	4K	2K
ATmega 128	128K	8K	4K

PROGRAMMING TIMERS

- The AVR has one to six timers depending on the family member.they are referred as Timers0,1,2,3,4 and 5.they can be used as timers or counter.
- Every timer needs a clock pulse to tick.
- The clock source can be internal or external.
- If we use the internal clock source, then the frequency of the crystal oscillator is fed into the timer.
- There for, it is used for time delay generation and consequently is called a *timer.*
- By choosing the external clock option, we feed pulses through one of the AVR's pins. This is called a *counter.*
- *In ATMega32 , there are 3 timers- Timer0, Timer1 and Timer3.in this Timer0 and Tmer2 are 8 bit,while Timer1 is 16bit.*

Basic registers of timers

• The timer registers are located in the I/O register memory.Therefore, you can read or write from timer registers using IN and OUT instructions, like the other I/O registers.Basic registers of timers/counters are:-

1. TCNTn (timer/Counter)
2. TOVn(Timer Overflow)
3. TCCRn (timer/counter Control register)
4. OCRn(Output Compare Register)
5. TIFR(Timer/counter Interrupt Flag Register) register

TCNTn (timer/Counter)

- InATmega32, we have TCNTO, TCNTI, and TCNT2 registers.
- The TCNTn register is a counter.
- Upon reset, the TCNTn contains zero.
- It counts up with each pulse. You can load a value into the TCNTn register or read its value.

TCNT0 and TCNT2

TCNT0	D7	D6	D5	D4	D3	D2	D1	D0

same as TCNT2

TCNT1H								TCNT1L							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

TOVn(Timer Overflow)

- It is a flag register.
- When a timer over flows, its TOVn flag will be set.

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
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

TOV0 D0 Timer0 overflow flag bit
0 = Timer0 did not overflow.
1 = Timer0 has overflowed (going from $FF to $00).

OCF0 D1 Timer0 output compare flag bit
0 = compare match did not occur.
1 = compare match occurred.

TOV1 D2 Timer1 overflow flag bit
OCF1B D3 Timer1 output compare B match flag
OCF1A D4 Timer1 output compare A match flag
ICF1 D5 Input Capture flag
TOV2 D6 Timer2 overflow flag
OCF2 D7 Timer2 output compare match flag

TCCRn (timer/counter C o n t r o l register)

- This register is used for setting modes of operation.
- Figure shows TCCRn register. It is 8 bit register.

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
W	RW	RW	RW	RW	RW	RW	RW
0	0	0	0	0	0	0	0

CS02:00 D2 D1 D0 Timer0 clock selector

D2	D1	D0	
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk (No Prescaling)
0	1	0	clk / 8
0	1	1	clk / 64
1	0	0	clk / 256
1	0	1	clk / 1024
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

WGM00, WGM01

D6	D3	Timer0 mode selector bits
0	0	Normal
0	1	CTC (Clear Timer on Compare Match)
1	0	PWM, phase correct
1	1	Fast PWM

FOC0 D7 Force compare match: This is a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.

COM01:00 D5 D4 Compare Output Mode:
These bits control the waveform generator

Bit	7	6	5	4	3	2	1	0	
	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	

ICNC1 D7 Input Capture Noise Canceler
0 = Input Capture is disabled.
1 = Input Capture is enabled.

ICES1 D6 Input Capture Edge Select
0 = Capture on the falling (negative) edge
1 = Capture on the rising (positive) edge

D5 Not used

WGM13:WGM12 D4 D3 Timer1 mode

TCCR1B

OCRn(Output Compare Register)

- The content of the OCRn is compared with the content of the TCNTn. When they are equal the OCFn (Output Compare Flag)flag will be set.

TIFR(Timer/counter Interrupt Flag Register) register

- The TIFR register contains the flags of different timers.
- Figure shows TIFR register. It is 8 bit register.

7	6	5	4	3	2	1	0
OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

TOV0 D0 Timer0 overflow flag bit
0 = Timer0 did not overflow.
1 = Timer0 has overflowed (going from $FF to $00).

OCF0 D1 Timer0 output compare flag bit
0 = compare match did not occur.
1 = compare match occurred.

TOV1 D2 Timer1 overflow flag bit

OCF1B D3 Timer1 output compare B match flag

OCF1A D4 Timer1 output compare A match flag

ICF1 D5 Input Capture flag

TOV2 D6 Timer2 overflow flag

OCF2 D7 Timer2 output compare match flag

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

COM1A1:COM1A0 D7 D6 Compare Output Mode for Channel A
(discussed in Section 9-3)

COM1B1:COM1B0 D5 D4 Compare Output Mode for Channel B
(discussed in Section 9-3)

FOC1A D3 Force Output Compare for Channel A
(discussed in Section 9-3)

FOC1B D2 Force Output Compare for Channel B
(discussed in Section 9-3)

WGM11:10 D1 D0 Timer1 mode (discussed in Figure 9-18)

TCCR1A

Mode of Operations

1.**Normal mode**

- In this mode, the content of the timer/counter increments with each clock.
- It counts up until it reaches its max (for 8 bit, it is 0xFF, for 16 bit it is 0xFFFF).
- When it rolls over from maximum value to 0, it sets high a flag bit called TOVn(Timer Over flow).
- This timer flag can be monitored.

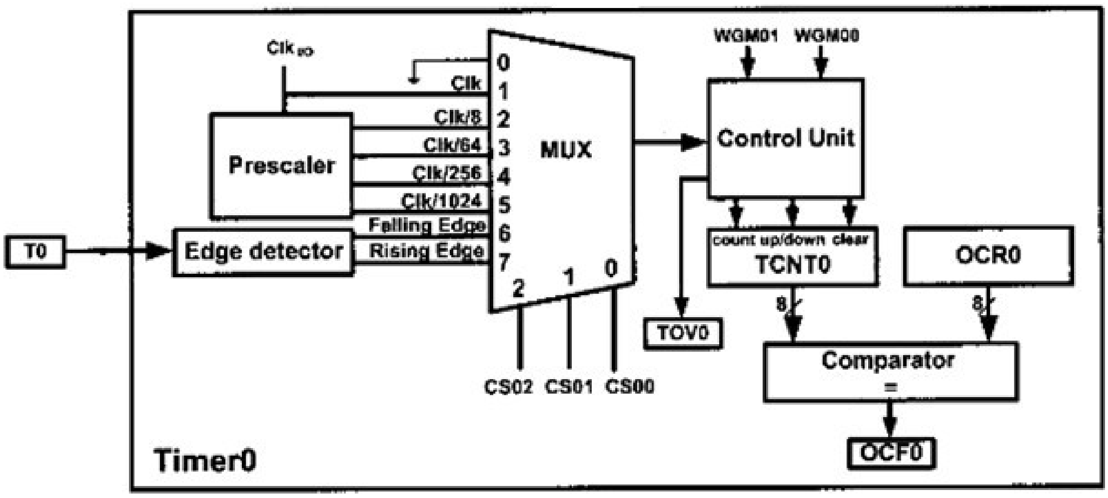
2. CTC mode (Clear Timer on Compare Match)

- The OCRn register is used with CTC mode.
- In the CTC mode, the timer is incremented with a clock.
- But it counts up until the content of the TCNTn register becomes equal to the content of OCRn (compare match occurs); then, the timer will be cleared and the OCFn flag will be set when the next clock occurs.
- The OCFn flag is located in theTIFR register.

Timer0 programming

- Timer0 is 8-bit in ATmega32.

Block diagram Timer0 in ATmega32



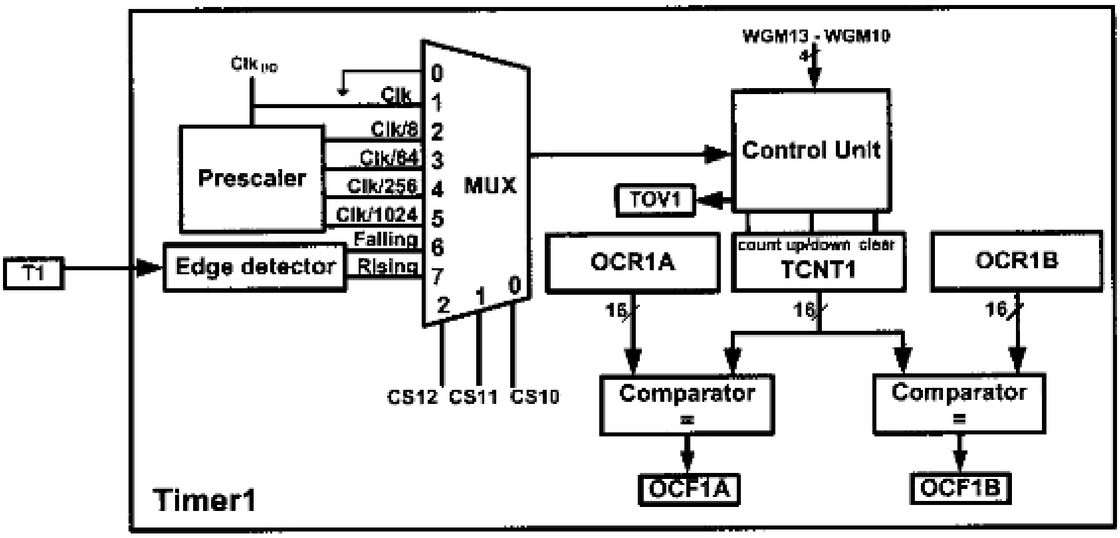
Steps to program Timer 0 in Normal mode

- To generate at time delay using Timer0 in Normal mode, the following steps are taken:
  1. Load the TCNTO register with the initial count value.
  2. Load the value into the TCCR0.
  3. Keep monitoring the timer over flow flag(TOV0) to see if it is raised.
  4. Stop the timer by disconnecting the clock source.
  5. Clear the TOVO flag for the next round.
  6. Go back to Step 1 to load TCNTO again.

Timer1 programming

- Timer1 is a16-bit timer.
- Its 16-bit register is split in to two bytes.
- The TCNT1 is referred as TCNT1L (Timer1 low byte) and TCNT1H (Timer1 high byte).
- Timer1 also has two control registers named TCCR1A(Timer/counter1 control register and TCCR1B.
- TheTOV1 (timer over flow)flag bit goes HIGH when over flow occurs.
- There are two OCR registers in Timer1:OCR1A andOCR1B.
- There are two separate flags for each of the OCR registers.
- Whenever TCNT1 equals OCR1A, the OCF1A flag will be set on the next clock.
- When TCNT1 equals OCR1B, the OCF1B flag will be set on the next clock.
- As Timer1 is a 16-bit timer, the OCR registers are16-bit registers as well and they are made of two 8-bit registers.
- For example, OCR1A is made of OCR1AH (OCR1A high byte) and OCR1AL (OCR1A low byte).

Block diagram Timer1 in ATmega32



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
}
```

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
        }
}
```

counters- timers can be used as counter if we provide pulses from outside. Example for counter as shown above

INTERRUPTS

- There are two methods by which devices receive service from the microcontroller: interrupts or polling.
- Whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal.
- Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device.
- The program associated with the interrupt is called the ***interrupt service routine*****(ISR) or *interrupt handler.***
- For every interrupt, there must be an interrupt service routine(ISR),or interrupt handler.
- When an interrupt is invoked, the microcontroller runs the interrupt service routine.
- For every interrupt there is a fixed location in memory that holds the address of its ISR.
- The group of memory locations set aside to hold the addresses of ISRs is called the *interrupt vector table***.**

Interrupts vs. polling

Interrupts	**Polling**

Whenever any device needs service, it send an interrupt signal.	The microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service.
can serve many devices not at the same time using priority.	Can not assign a priority.It checks all devices in a round-robin fashion.
It can ignore a device request.	It cannot ignore a device request.
It avoids the tying down the micro controller.	It wastes much of the time by polling devices that do not need service.
More efficient	Not efficient

Steps in executing an interrupt

1. It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.
2. It jumps to a fixed location in memory called the *interrupt vector table*. The Interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI(return from interrupt).
4. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted .First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

Sources of interrupts in the AVR

- Depending on which peripheral is incorporated in to the chip, there are many interrupts.
- The following are some of the most widely used sources of interrupts in the AYR:

1. There are at least two interrupts set aside for each of the timers, one for over flow and another for compare match.
2. Three interrupts are set aside for external hardware interrupts. Pins PD2 (PORTD.2), PD3(PORTD.3), and PB2(PORTB.2) are for the external hardware interrupts INT0 ,INT1, and INT2, respectively.
3. Serial communication's USART has three interrupts, one for receive and two interrupts for transmit.
4. The SPI interrupts.
5. The ADC (analog-to-digita lconverter).

Enabling and disabling an interrupt

- At starting, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated.
- The interrupts must be enabled (unmasked)by software in order for the microcontroller to respond to them.
- The D7 bit(I flag) of the SREG(Status Register)register is responsible for enabling and disabling the interrupts globally.
- Bits of status register is shown below:

Steps in enabling an interrupt

1. Bit D7(I) of the SREG register must be set to HIGH to allow the interrupts to happen. This is done with the "SEI" (Set Interrupt) instruction.
2. If I= I, each interrupt is enabled by setting to HIGH the interrupt enable (IE) flag bit for that interrupt. If I=0,no interrupt will be responded.

PROGRAMMING TIMER INTERRUPTS

- There are 2 timer interrupts.
  1. Overflow 2.Compare match

Rollover timer flag and interrupt

- The timer overflow flag is raised when the timer rolls over.
- When the timer overflows, the TOV0 flag will set.
- If the timer interrupt in the interrupt register is enabled, TOYO is raised whenever the timer rolls over and the microcontroller jumps to the interrupt vector table to service the ISR.

Compare match timer flag and interrupt

- We load the OCR register with the proper value and initialize the timer to the CTC mode.
- When the content of TCNT matches with OCR ,the OCF flag is set, which causes the compare match interrupt to occur.

INTERRUPT PRIORITY IN THE AVR

- If two interrupts are activated at the same time, the interrupt with the higher priority is served first.
- The priority of each interrupt is related to the address of that interrupt in the interrupt vector.
- The interrupt that has a lower address, has a higher priority.
- When the AVR begins to execute an ISR ,it disables the I bit of the SREG register ,causing all the interrupts to be disabled, and no other interrupt occurs while serving the interrupt.
- When the RETI instruction is executed, the AVR enables the I bit, causing the other interrupts to be served.

Context saving in task switching

- In multitasking systems, such as multitasking real-time operating systems (RTOS),the CPU serves one task(job or process)at a time and then moves to the next one.
- In these cases, we can save the contents of registers on the stack before execution of each task ,and reload the registers at the end of the task.
- This saving of the CPU contents before switching to a new task is called *context saving* (or *context switching).*

Interrupt latency

- The time from the moment an interrupt is activated to the moment the CPU starts to execute the task is called the *interrupt latency.*
- This latency is 4machine cycle times.
- During this time the PC register pushed on the stack and the I bit of SREG register clears. i.e all interrupts are disabled.

Interrupt Programming in C

- in C language , there is no instruction to manage the interrupts.
- In AVR programming the following have been added to manage the interrupts.
  - Interrupt include file:- include the interrupt header in our program. #include<avr/interrupt.h>
  - cli() and sei():- these macros do clear and set the I bit of SREG register.
  - Defining ISR:- to write an ISR for an interrupt we use the following structure.
    - ISR(interrupt vector name)
      {
          //our programmer}
      }
    - for the interrupt vector name we must use interrupt vector name in AVR.
    - e.g. ISR(TIMER0_COMP_vect)
      {
      }
- the c compiler automatically adds instruction to the beginning of the ISRs, which save the contents of all of the general purpose register and the SREG register on the stack.

Interrupt	Vector Name in WinAVR
External Interrupt request 0	INT0_vect
External Interrupt request 1	INT1_vect
External Interrupt request 2	INT2_vect
Time/Counter2 Compare Match	TIMER2_COMP_vect
Time/Counter2 Overflow	TIMER2_OVF_vect
Time/Counter1 Capture Event	TIMER1_CAPT_vect
Time/Counter1 Compare Match A	TIMER1_COMPA_vect
Time/Counter1 Compare Match B	TIMER1_COMPB_vect
Time/Counter1 Overflow	TIMER1_OVF_vect
Time/Counter0 Compare Match	TIMER0_COMP_vect
Time/Counter0 Overflow	TIMER0_OVF_vect
SPI Transfer complete	SPI_STC_vect
USART, Receive complete	USART0_RX_vect
USART, Data Register Empty	USART0_UDRE_vect
USART, Transmit Complete	USART0_TX_vect
ADC Conversion complete	ADC_vect
EEPROM ready	EE_RDY_vect
Analog Comparator	ANALOG_COMP_vect
Two-wire Serial Interface	TWI_vect
Store Program Memory Ready	SPM_RDY_vect

Using Timer0 generate a square wave on pin PORTB.5, while at the same time transferring data from PORTC to PORTD.

Solution:

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

int main ()
{
        DDRB  |= 0x20;              //DDRB.5 = output

        TCNT0 = -32;                //timer value for 4 µs
        TCCR0 = 0x01;               //Normal mode, int clk, no prescaler

        TIMSK = (1<<TOIE0);         //enable Timer0 overflow interrupt
        sei ();                     //enable interrupts

        DDRC = 0x00;                //make PORTC input
        DDRD = 0xFF;                //make PORTD output

        while (1)                   //wait here
              PORTD = PINC;
}

ISR (TIMER0_OVF_vect)               //ISR for Timer0 overflow
{
        TCNT0 = -32;
        PORTB ^= 0x20;              //toggle PORTB.5
}
```

Video links:

1.[***https://www.youtube.com/watch?v=U9B6j1I06rI***](https://www.youtube.com/watch?v=U9B6j1I06rI)
2.[***https://www.youtube.com/watch?v=EnC387Rtas0***](https://www.youtube.com/watch?v=EnC387Rtas0)
*3.**https://www.youtube.com/watch?v=i_CeYCd4A80***
4.*https://www.youtube.com/watch?v=MeRb6bY0rxs*
*5.**https://www.youtube.com/watch?v=ICIKWlUjYuw***
6.https://www.youtube.com/watch?v=gqDGeWKg62I

Important questions:

1. explain C data types for the AVR microcontroller.
2. Describe the Normal and CTC mode of Timers.
3.Explain Interrupts in AVR
4.Illustrate Interrupt Programming in C .
5.Explain memory allocation in C.
6. how to create a time delay in AVR C.
7. Explain the basic registers of Timer.
8. what is ISR(Interrupt service Routine).
9. Explain the methods to receive service from the microcontroller.
10.How to enable and disable an interrupt.
11. Which are the sources of interrupts.
12. What is the difference between PORTC=0x00 and DDRC=0x00?

One word questions:

1. give two factors that can affect the delay size.

2. to access the data direction register of port b, we use DDRB. True or false.
3. find the content of portc after the execution of the following code.

```
Portc=~(0<<3);
```

4. find the content of portc after the execution of the following code.

```
Portc=0;
portc=portc|0x99;
portc=~portc;
```

5. the AVR family has a maximum of ....of program ROM space.
6. The ATMega128 has ....of program ROM.
7. how many timers do we have in the ATMega32?
8.Timer1 is 16 bit timer. True or false.
9. in CTC mode , the counter rolls over when the counter reaches.......
10. for counter0, which pin is used for the input clock?
11. timer0 supports the highest prescaler value of .....
12. TCCR0 is a ....bit register.
13. what is the job of TCCR0 register.
14. which register holds the TOV0 and TOV1 bits.
15. state the difference between timer0 and timer1.
16. in the ATMega32 what memory area is assigned to the interrupt vector table.
17. for each of Timer0 and Timer1, there is a unique address in the interrupt vector table.

Problems

1. write a c program to toggle all bits of portb every 200ms.
2. write a c program to toggle only bit 3 of PORTC every 200ms.
3. write a c program to count up port b from 0-99 continously.
4.write a c program to convert packed BCD 0x34 to ASCII and display the bytes on PORTB and PORTC.
5. write a c program to convert ASCII digits of ‘7’ and ‘2’ to packed BCD and display them on PORTB.
6. write a c program to send out the value 23H serially one bit at a time via PORTB,pin 4.the MSB should go out first.