MODULE II
AVR Microcontroller

CO2	Make use of AVR Microcontrollers to develop embedded programs using embedded C		
M2.01	Familiarize AVR controllers family members and criteria to select a microcontroller	2	Understanding.
M2.02	Explain block diagram of Atmega32 and its blocks	2	Understanding
M2.03	Illustrate Registers, Memory organization, Status register, Program counter, I/O ports and its registers, Interrupts, priority of interrupts	3	Understanding.
M2.04	Illustrate Timers in AVR	3	Understanding.
M2.05	Develop embedded C programs for logic operations, data conversions and I/O operations.	2	Applying.
M2.06	Develop embedded C programs for time delay, time delay calculation. Timer programming and timer interrupts handling	2	Applying.
M2.07	Develop Programs to handle external hardware interrupts and programming based on priority of interrupts	2	Applying.
	Series Test - I	1	

Syllabus Content:
AVR Microcontroller Architecture *- Comparison of AVR family members and Selection of a microcontroller, ATMega32- Simplified Block diagram of ATmega32 microcontroller - Registers, - data memory - I/O memory -SFRs - internal data SRAM, Status register, Program Counter and Program ROM space, I/O ports, Registers associated with I/O ports, Timers-0,1,2 (Block level) associated registers, Interrupts.*
AVR programming using embedded C: *Data types, I/O programming, logic operations - data conversion programs - time delays- programming of timers 0 - timer 1 - timer2, AVR interrupts - programming of timer interrupts - programming external hardware interrupts - interrupt priority in AVR microcontroller.*

AVR(ADVANCED VIRTUAL RISC) Microcontrollers:

- AVR was developed in the year **1996** by **Atmel Corporation**. The architecture of AVR was developed by Alf-Egil Bogen and Vegard Wollan. AVR derives its name from its developers and stands for **Alf-Egil Bogen Vegard Wollan RISC microcontroller**, also known as **A**dvanced **V**irtual **R**ISC.
- AVR microcontrollers are used for high speed signal processing operations in an embedded system.
- It is generally 8 bit advanced RISC single chip microcontroller(except for AVR32 which is a 32-bit microcontroller) with Harvard architecture.
- On chip program ROM, data RAM, EEPROM & Flash memory.
- On chip timers & I/O ports.

- Variety of serial interfaces (I2C, USART/UART, SPI,TWI)
- Additional features like ADC, PWM.

Q) List any two features of Atmega32 microcontroller. (1 mark)

NOTE:

RISC(Reduced Instruction Set Computer)	**CISC(Complex Instruction Set Computer)**
• It uses a limited number of instruction that requires less time to execute the instructions.	• It uses a large number of instruction that requires more time to execute the instructions.
• The program written for RISC architecture needs to take more space in memory.	• Program written for CISC architecture tends to take less space in memory.
• Uses more number of registers.	• Uses less number of registers.
• RISC architecture can be used with high-end applications like telecommunication, image processing, video processing, etc.	• CISC architecture can be used with low-end applications like home automation, security system, etc.
• Example of RISC: PIC, AVR	• Examples of CISC: Intel 8051, ziglog Z8

AVR family: AVRs are generally classified into 4 broad groups:

- **Classic AVR (AT90SXXXX):**This is the original AVR chip which has been replaced by newer chips. Some members of this family are – AT90S2313, AT90S2323, AT90S4433.
- **Mega AVR (ATMegaXXXX)**: These are powerful micro controllers with more than 120 instructions and lots of different peripheral capabilities. Some members of this family are – ATMega8, ATMega16, ATMega32, ATMega64, ATMega128.
- **Tiny AVR (ATTiny XXXX):** Micro controller in this group have less instructions and smaller packages when compared to mega family. Some members of this family are – ATTiny13, ATTiny25, ATTiny44, ATTiny84.
- **Special Purpose AVR:** They are made with special capabilities for specific applications. Examples for special capabilities are USB controller, CAN controller, LCD controller, Zigbee, Ethernet controller, FPGA and advanced PWM. Some members of this family are – AT90CAN128, AT90USB128, AT90PWM128.

Q) List any two AVR family microcontrollers. (1 mark)
Q) Describe features of AVR family of microcontrollers. (7 marks)

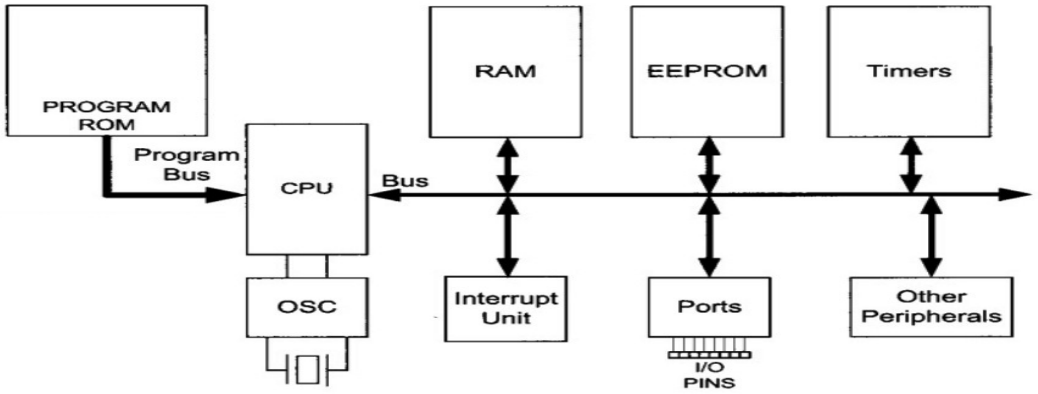
Choosing a microcontroller: There are five major 8-bit microcontrollers. They are Intel 8051, Motorola MC68HC11, Atmel AVR, ziglog Z8, Microchip PIC families. Each of the above mentioned microcontrollers has a unique set of instruction set & register set, therefore they are not compactable with each others.

Hence the different criteria for choosing a mocrocontroller are:

1. It must meet the task efficently & cost effective. It includes several factors like speed of the controllers, packaging, power consumption, memory capacity, number of I/O pins, timers etc.
2. Availability of hardware & software tools like compilers, assembles, debuggers, emulator etc.
3. Wide availability & reliable sources of microcontrollers in required quantities both now and in the future.

Q) List the criteria to a microcontroller for an embedded system. (3marks)

Simplified Block Diagram of ATMEGA32 Microcontroller:



I/O ports: AT Mega32 has four ports(Port A,Port B, port C and Port D) having 32 pins.

Oscillator: AT Mega32 has an internal oscillator for its clock. By default it is set to operate an internal caliberated oscillator of 1 MHz.

Timer/counter: It is used to count an event or to generate time delays between two operations. AT Mega32 consist of two 8 bit(Timer0 & Timer2) and one 16 bit timer/counter(Timer1).

Watchdog timer: A watchdog timer is a simple countdown timer which is used to reset a microprocessor after a specific interval of time. System reset is required for preventing failure of the system in a situation of a hardware fault or program error.

Interrupt unit: AVR ATmega32 consist of 21 interrupt sources out of which three are external. The remaining are internal interrupts which supports the peripherals like USART, ADC, timer etc. The three external hardware interrupts are on pins PD2, PD3, and PB2 which are referred to as INT0, INT1, and INT2 respectively.

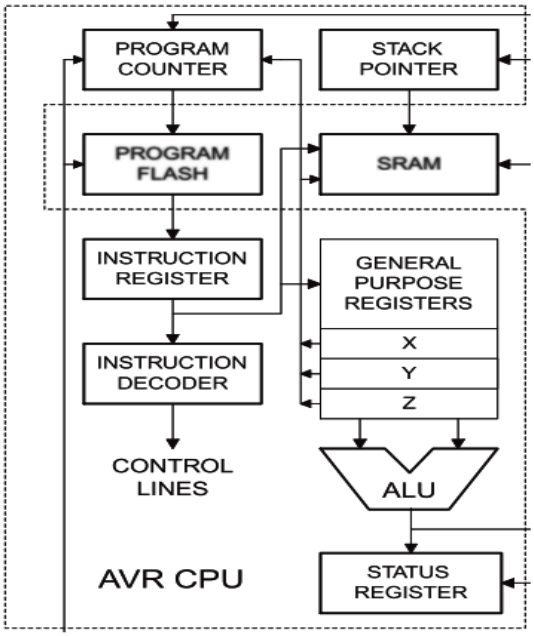
Memory: AT Mega32 consist of three different memory sections.

- **Program ROM:** It is used to store program or codes. AT Mega32 has 32KB as Program ROM
- **EEPROM:** It is also a non volatile memory used to store data. AT Mega32 has 1 KB of EEPROM.
- **Data RAM:** It is a volatile memory used to store data. AT Mega32 has 2 KB of Data RAM.

Other Pheripherals: It includes:

- **ADC interface:** AT Mega32 has an 8 channel ADC with resolution of 10 bits for Analog to Digital signal conversions.
- **USART(universal synchronous asynchronous receiver transmitter):** USART interface is available for interfacing with external device capable of communicating serially.
- **SPI(Serial Peripheral Interface):** It is used for serial communication between two devices on a common clock source. The data transmission rate of SPI is more than that of USART.
- **TWI(Two Wire Interface):** Can be used to attach low speed peripherals to a motherboard, embedded systems, cell phones or other electronic devices. It is a multi master serial single ended computer bus.

AVR CPU:



Program Counter: The PC stores a program memory address that contains the location of the next instruction.
Instruction Register: Instructions fetched from the memory are placed in this register.
Instruction Decoder: Decodes instruction placed in the instruction register.
Arithmetic Logic Unit(ALU): ALU executes the instruction or it performs the arithematic & logical operations.
ISP(In System Programmable): AVR have in system programmable flash memory which can be programmed without removing the IC from the circuit. It allows to reprogram the micro controller while it is in the application circuit.
Stack pointer (SP): It is a 16 bits register which stores return address of subroutine/interrupt calls. It may be used to store temporary data and local variables.

AVR Status Register or Flag register: The status register is an 8 bit register.

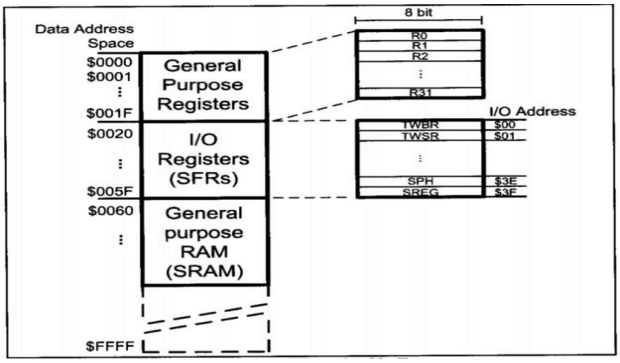
I	T	H	S	V	N	Z	C
D7	D6	D5	D4	D3	D2	D1	D0

- **Carry flag(C):** It is set whenever there is a carry out of D7. This flag is affected after an 8 bit addition or subtraction.
- **Zero flag(Z):** It reflects the result of ALU operation. If the result is zero then Z=1 and ifresult is non zero Z=0.
- **Negative flag(N):** It is the binary representation of signed number uses D7 as signed bit . It reflects the result of arithematic operation . if D7 bit of the result is 0, the N=0 and the result is positive. If D7 bit is 1,then N=1 and the result is negative.
- **Overflow flag(V):** This flag is set whenever the result of a signed number operation is too large , causing higher order bit to overflow into the sign bit.
- **Sign bit(S):** This flag is the result of XOR of N and V flags.
- **Half carry flag(H):** If there is a carry out from D3 to D4 during an ADD or SUB operation , this bit is set, otherwise it is cleared.
- **Temporary flag or Bit copy storage(T):** It is used when we want to copy a bit of data from one general purpose register to another.

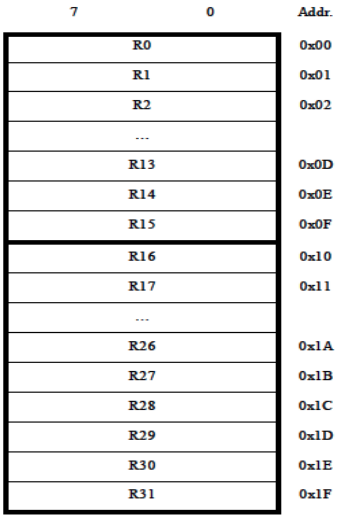
- **Global Interrupt Enable flag(I):** Setting this bit enables all interrupts. Resetting disables all interrupts.

Q) Describe about Atmega32 with the help of a simplified block diagram. (7 marks)

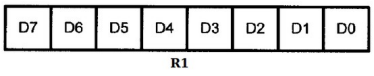
AVR data Memory: Data memory store data . this is composed of 3 parts: General Purpose Registers , I/O memory and internal data SRAM.



General Purpose Registers:



- In ATmega32 there are **32 general purpose registers (R0-R31)**.
- All registers are **8-bit**.
- Located in lowest location of memory from $00 to $1F.
- General purpose registers are used by all **arithmetic and logical operations, to store information temporarily**.
- Each registers can perform function of accumulator in other microcontrollers.
- **R26 + R27 => X - Register(16-Bits)**
- **R28 + R29 => Y - Register(16-Bits)**
- **R30 + R31 => Z - Register(16-Bits)**
- **X, Y, Z-Registers** are used as **address pointers**.



I/O Memory or Specific Function Registers(SFRs):

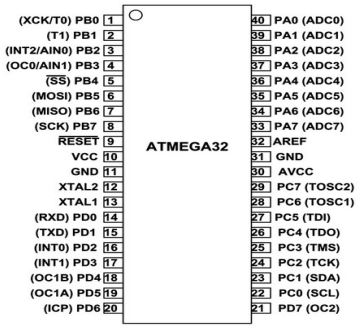
- It is dedicated to **special functions such as timer, serial communication, I/O port, ADC etc.**
- AVR I/O memory is made of **8-bit** registers.
- **The function of each memory location is fixed by CPU designer at time of design because it is used for control of micro controller or peripherals** .
- All of the AVR have at **least 64Bytes** of I/O memory . This section is called standard I/O memory.
- Address of standard I/O memory is $0020 to $005F.
- In AVRs with more than 32 pins there is also an extended I/O memory, which contains registers for controlling extra ports and peripherals.

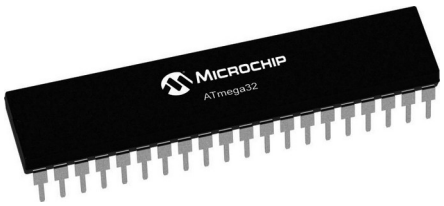
Internal data SRAM:

- It is used for **storing data and parameters** by AVR programmers.
- It is also called as **scratch pad**.
- Each location is **8 bit** and can be directly accessed by its address.
- Size of SRAM can vary from chip to chip. In ATmega32 SRAM is **2KB** size having address $0060 to $085F.

Q) Explain the Data memory architecture of ATmega 32 with necessary diagrams. Mention the purpose of each memory. (7 marks)

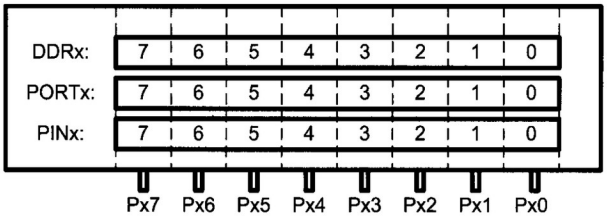
Pin Diagram of Atmega32:





I/O Ports & Registers: Atmega32 has **4 ports(PORTA, PORTB, PORTC, PORTD)** having **32** pins assigned to these ports. It is dedicated to special functions such as I/O operations, timer, serial communication, interrupt, ADC etc. To use any port as input or output, it must be programmed. . Each port has **three 8-bit I/O registers** associated with it. They are designated as **PORTx**, **DDRx** & **PINx**.

Port	Address	Usage
PORTA	$3B	output
DDRA	$3A	direction
PINA	$39	input
PORTB	$38	output
DDRB	$37	direction
PINB	$36	input
PORTC	$35	output
DDRC	$34	direction
PINC	$33	input
PORTD	$32	output
DDRD	$31	direction
PIND	$30	input



DDRx Register: It is used for the purpose of making a given port an input or output port.

- To make a output port write 1s to DDRx register.
- To make a input port write 0s to DDRx register.

PINx Register: To read the data present at a pin to CPU, we use the PINx register.

PORTx Register: To sent the data present in CPU to a pin, we use the PORTx register.

Q) Explain registers associated with AVR ports. (6 marks)

AVR programming in C:

NOTE: Why program the AVR in C?

A compiler is used to translate high level language program(like C-program) into machine code(hex file) & an assembler is used to translate an assembly language program into machine code(hex file). The hex file produced by assembler is smaller than hex file produced by compiler, C-programming is more popular than Assembly language programming because of the following reasons:

- It is easier & less time consuming to write a program in C than in Assembly.
- C codes are easier to modify & update.
- C code can be compactable with other microcontrollers with little or nomodification.
- C codes are readily available in function libraries compared to Assemble language codes.

Data Types: In C programming, data types are declarations for variables. This determines the type and size of data associated with variables.

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

1) Unsigned char: It is an **8-bit** data type that takes the value in the range of **0-255(00-FFH)**. It is one of the most widely used data types for AVR like setting counter values. The C-compiler by default use the signed char unless the keyword '**unsigned**' is specified.

2) Signed char: It is an **8-bit** data type that uses the Most Significant Bit(**MSB**) i.e **D7** bit to represent **+ve** or **-ve** value. As a result only 7 bits are available for use, giving the rang -128 to +127.

3) Unsigned int: It is an **16-bit** data type that takes the value in the range of **0-65,535(0000-FFFFH)**. It is used to define 16-bit variable such as memory address, set counter values more than 256. The C-compiler by default use the signed int unless the keyword '**unsigned**' is specified. Since it is a 2-byte data type, **int** is used only in situations that demands, because the use of int data type will result in the creation of larger hex files, slower execution and more memory usage.

4) Signed int: It is an **16-bit** data type that uses the Most Significant Bit(**MSB**) i.e **D15** bit to represent **+ve** or **-ve** value. As a result only 15 bits are available for use, giving the rang -**32,768** to +**32,767**.

5) Other data types:

- If we want values greater than **16-bit** use **long** data types.
- If we want deal with fractional numbers use **float** & **double** data types.

Q) Explain different data types in AVR C-program. (6 marks)

I/O programming in C:

Byte size I/O: To acces a port register as a byte, we use PORTx , where x indicate the port name. DDRx is used to indicate the data direction of port. To acess a a data as a byte, PINx is used.

Bit size I/O: The I/O port of Atmega32 are bit accessible. But some AVR C-compilers do not support this feature.

Write an AVR C program to get a byte of data from Port B, and then send it to Port C.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
   unsigned char temp;

   DDRB = 0x00;                        //Port B is input
   DDRC = 0xFF;                        //Port C is output

   while(1)
   {
      temp = PINB;
      PORTC = temp;
   }
   return 0;
}
```

Q) Write a C program to read 8 switches connected to PORTB and output the contents to LEDs connected to PORTC. (7 marks)

NOTE: The line **while(1)** in a C program creates an infinite loop- this is a loop that never stops executing. It executes over and over and over again, unless the program is intentionally stopped or there is some condition under this loop that gets met that takes the program out of this infinite loop.

Write an AVR C program to toggle all bits of Port B 50,000 times.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
   unsigned int z;
   DDRB = 0xFF;                        //PORTB is output

   for(z=0; z<50000; z++)
   {
      PORTB = 0x55;
      PORTB = 0xAA;
   }

   while(1);                           //stay here forever
   return 0;
}
```

Time delay:

There are three ways to create a time delay in AVR C

1. Using a simple for loop
2. Using predefined C functions
3. Using AVR timers

Using for loop: In creating the delay using for loop, the accuracy of delay may depends on the compiler used.

```
for (i=0; i<10000; i++)
{
}
```

Using predefined C functions: To generate time delays predefines functions such as ***_delay_ms()*** & ***_delay_us()*** defined in ***util/delay.h*** (header file) are used in AVR studio.

Write an AVR C program to toggle all the pins of Port B continuously with a 10 ms delay. Use a predefined delay function in Win AVR.

Solution:

```
#include <util/delay.h>                //delay loop functions
#include <avr/io.h>                    //standard AVR header

int main(void)
{
      DDRB = 0xFF;                     //PORTB is output
      while (1){
            PORTB = 0xAA;
            delay_ms(10);
            PORTB = 0x55;
            delay_ms(10);
               }
      return 0;
}
```

Q) Write a C program to blink LED alternatively connected to PORTB with a delay of 10ms. (7 marks)

Logical Operations(Bit-wise): It includes **AND**(&), **OR**(|), **EX-OR**(^), **inverter**(~), **shift right**(>>), **shift left**(<<).

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

Eg 1 AND portB

DDRB=0XFF;

PORTB=0x35 & 0x0F;

35H	00110101
&0FH	00001111
05H	00000101

Eg 2 OR portC

DDRC=0XFF;

PORTC=0x35 | 0x0F;

35H	00110101
| 0FH	00001111
3FH	00111111

Eg 3 EX-OR portD

DDRD=0XFF;

PORTD=0x35 ^ 0x0F;

35H	00110101
^ 0FH	00001111
3AH	00111010

Eg 4 Inverting portB

DDRB=0xFF;

PORTB= ~0x35

35H	00110101
~CAH	11001010

Q) Explain the different logical operations with examples? (7 marks)

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;
}
```

Write an AVR C program to monitor bit 5 of port C. If it is HIGH, send 55H to Port B; otherwise, send AAH to Port B.

Solution:

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

int main(void)
{
   DDRB = 0xFF;                  //PORTB is output
   DDRC = 0x00;                  //PORTC is input


   while(1)
   {
      if (PINC & 0b00100000)     //check bit 5 (6th bit) of PINC
        PORTB = 0x55;
      else
        PORTB = 0xAA;
   }

   return 0;
}
```

Bitwise shift operators in C:

operation	symbol	format
Shift right	>>	data>>number of bits to shift
Shift left	<<	data<<number of bits to shift

Eg

0b00010000>>4=0b00000001

0b00010000<<3=0b10000000

1<<2 =0b00000100

1<<6 = 01000000

1<<3 = 00001000

1<<5 = 00100000

~1<<3 = 11110111

~1<<1 = 11111101

Write an AVR C program to monitor bit 7 of Port B. If it is 1, make bit 4 of Port B input; else, change pin 4 of Port B to output.

Solution:

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

int main(void)
{
   DDRB = DDRB & ~(1<<7);              //bit 7 of Port B is input

   while (1)
   {
      if(PINB & (1<<7))
        DDRB = DDRB & ~(1<<4);         //bit 4 of Port B is input
      else
        DDRB = DDRB | (1<<4);          //bit 4 of Port B is output
   }

   return 0;
}
```

Q) Develop AVR embedded C program to display the status of a port pin into an LED connected to another pin. (7 marks)

SOL:

```
#include <avr/io.h>
int main(void)
   {
        DDRB = 0xFF;        //PORT B is output
        DDRC = 0x00;        //PORT C is input
        PORTB = 0x00;

         while(1)
             {
                 if (PINC & (1<<2))  //Read status of switch by reading value of PORTC Pin2 &
                                       Check it is high.
                       {
                             PORTB = 0x00;
                       }
                  else                 //Read status of switch by reading value of PORTC Pin2 &
                                       Check it is low.
                       {
                            PORTB = 0xFF;
                       }
              }

        return 0;
    }
```

Data conversion programs in C:

ASCII numbers

On ASCII keyboards, when the "0" key is activated, "0011 0000" (30H) is provided to the computer. Similarly, 31H (0011 0001) is provided for the "1" key, and so on, as shown in Table 7-5.

Table 7-5: ASCII Code for Digits 0–9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Packed BCD to ASCII conversion

To convert packed BCD to ASCII, you must first convert it to unpacked BCD. Then the unpacked BCD is tagged with 011 0000 (30H).

Packed BCD	Unpacked BCD	ASCII
0x29	0x02, 0x09	0x32, 0x39
00101001	00000010,00001001	00110010,00111001

Write an AVR C program to convert packed BCD 0x29 to ASCII and display the bytes on PORTB and PORTC.

Solution:

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

Q) Write a AVR C-program to convert packed BDC number 0x45 to corresponding ASCII codes & display on PORTC & PORTD? (7 marks)

ASCII to packed BCD conversion

To convert ASCII to packed BCD, you first convert it to unpacked BCD (to get rid of the 3), and then combine the numbers to make packed BCD. For example, 4 and 7 on the keyboard give 34H and 37H, respectively. The goal is to produce 47H or "0100 0111", which is packed BCD.

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

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;
}
```

Binary (hex) to decimal and ASCII conversion in C

One of the most widely used conversions is binary to decimal conversion. In devices such as ADCs (Analog-to-Digital Converters), the data is provided to the microcontroller in binary. In some RTCs, the time and dates are also provided in binary. In order to display binary data, we need to convert it to decimal and then to ASCII. Because the hexadecimal format is a convenient way of representing binary data, we refer to the binary data as hex. The binary data 00–FFH converted to decimal will give us 000 to 255. One way to do that is to divide it by 10 and keep the remainder. For example, 11111101 or FDH is 253 in decimal.
The following is one version of an algorithm for conversion of hex (binary) to decimal:

Hex	Quotient	Remainder
FD/0A	19	3 (low digit) LSD
19/0A	2	5 (middle digit)
		2 (high digit) (MSD)

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

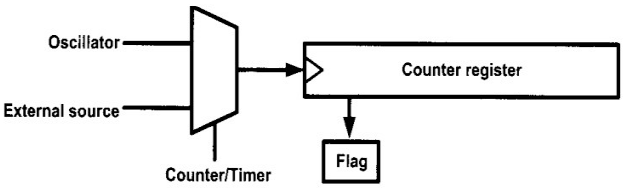
Solution:

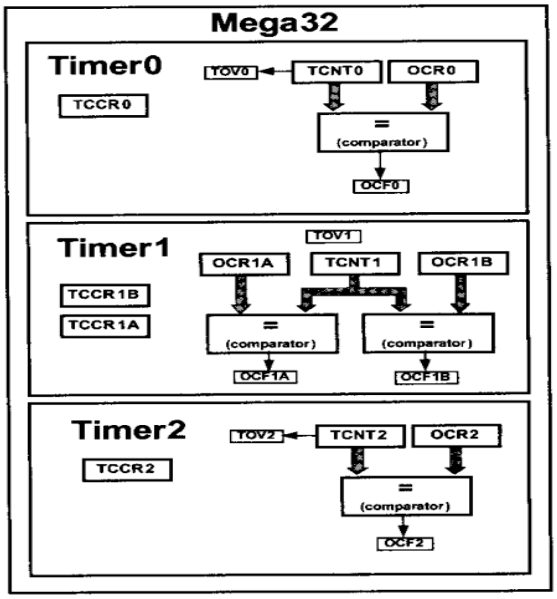
```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
     unsigned char x, binbyte, d1, d2, d3;
     DDRB = DDRC = DDRD =0xFF;         //Ports B, C, and D output
     binbyte = 0xFD;                   //binary (hex) byte
     x = binbyte / 10;                 //divide by 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;
}
```

Counter/Timer: It is used to count an event or to generate time delays between two operations.. AT Mega32 consist of **two 8 bit(Timer0 & Timer2)** and **one 16 bit timer/counter(Timer1)**.

NOTE: Every timer needs a clock pulse to function. The clock source can be internal or external. If we use an internal clock source, then the clock pulse of crystal oscillator is fed directly into it & is called a **timer**. If we choose an external source, we feed clock pulse through one of AVR pins & it is called a **counter.**

Q) Name any two timers available in Atmega32. (1 mark)





Basic Registers in Timers: The timer registers are located in I/O register memory. Hence we can read/write timer registers using I/O instructions. The following are the timer registers:

- **TCNTn(Timer/Counter) Registers:** Upon the reset, the TCNTn contains 0. It count up with each clock pulse. We can read or load a values into this register. It is a **8/16**-bit register.
- **TOVn(Timer Overflow) Flag:** When timer overflow, this flag is set.
- **TCCRn(Timer/Counter Control Register) Register:** This register is used to set the mode of operation. It is a **8/16**-bit register.
- **OCRn(Output Compare Register) Register:** The content of ORCn is compared with TCNTn. When they are equal **OCFn**(Output Compare Flag) flag is set. It is a **8/16**-bit register.
- **Input Capture Register (ICR1):** It is an auxillary 16-bit register used for capturing operation(to detect and measure events happening outside the microcontroller). Input Capture Flag 1(**ICF1**) bit is set denoting an input capture event occur, indicating that the Timer/Counter1 value has been transferred to the ICR1 register.
- **TIFR(Timer/Counter Interrupt Flag Register):** It is a **8**-bit register common to all the timers. The flag(**TOVn, OCFn, ICF1**) represent some events regarding timer.

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

- **TIMSK(Timer Interrupt Mask) Register:** It is a **8**-bit register & is **a control register used to mask or unmask the timer interrupts**.

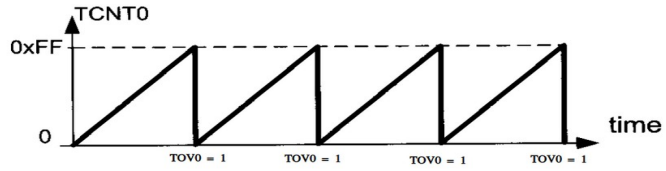
D7							D0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

TOIE0	Timer0 overflow interrupt enable
OCIE0	Timer0 output compare match interrupt enable
TOIE1	Timer1 overflow interrupt enable
OCIE1B	Timer1 output compare B match interrupt enable
OCIE1A	Timer1 output compare A match interrupt enable
TICIE1	Timer1 input capture interrupt enable
TOIE2	Timer2 overflow interrupt enable
OCIE2	Timer2 output compare match interrupt enable

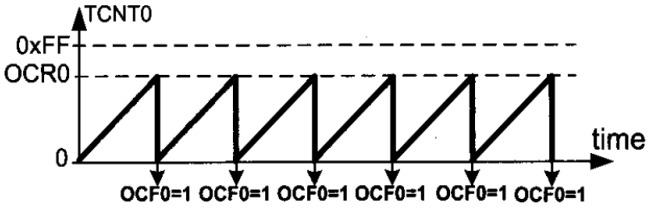
Q) What is TIFR register? Show the bit position and indicate the purpose of the bits.(4 marks)

Timer operation modes:

Normal mode: In this mode, the content of the timer/counter increments with each clock . It counts up until it reaches its max of 0xFF for 8-bit timer (0xFFFF for 16-bit timer). When it rolls over from 0xFF to 0x00 for 8-bit timer (0xFFFF to 0x0000 for 16-bit timer) it sets high TOV flag bit.



Clear Timer on Compare match(CTC) mode: In this mode, the content of the timer/counter increments with each clock, but count up until the content of TCNTn register becomes equal to OCRn, then the timer is cleared & the OCFn flag is set.



Steps to program Timer 0 in normal mode:

1) Load the **TCNT0** register with the initial count value.

TCNT0	D7	D6	D5	D4	D3	D2	D1	D0

Finding values to be loaded into the timer

Assuming that we know the amount of timer delay we need, the question is how to find the values needed for the TCNT0 register. To calculate the values to be loaded into the TCNT0 registers, we can use the following steps:

1. Calculate the period of the timer clock using the following formula:
   $T_{clock} = 1/F_{Timer}$
   where F_{Timer} is the frequency of the clock used for the timer. For example, in no prescaler mode, $F_{Timer} = F_{oscillator}$. T_{clock} gives the period at which the timer increments.
2. Divide the desired time delay by T_{clock}. This says how many clocks we need.
3. Perform 256 – *n*, where *n* is the decimal value we got in Step 2.
4. Convert the result of Step 3 to hex, where *xx* is the initial hex value to be loaded into the timer's register.
5. Set TCNT0 = *xx*.

2) Load the values into the **TCCR0** register (mode, prescalar, clock etc).

Bit	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Read/Write	W	RW	RW	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0

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.

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

COM01:00 D5 D4 Compare Output Mode: These bits control the waveform generator (see Chapter 15).

CS02:00

D2	D1	D0	Timer0 clock selector
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.

TCCR0 (Timer/Counter Control Register) Register

3) Keep monitoring the **timer overflow flag (TOV0)** and get out of the loop when it is set.
4) Stop the timer by disconnecting the clock source.
5) Clear the TOV0 flag for the next round.
6) Go back to step 1 to load TCNT0 again.

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
}
```

Q) Write embedded C program to toggle a port bit with a time delay, use timer0 operation. (7 marks)

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
}
```

NOTE: The clock unit of the AVR timers consists of a **prescaler** connected to a multiplexer. A prescaler can be considered as a clock divider.

Interrupts in Atmega32: When ever any device want to communicate, the device notifies it by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops the task and respond to the device.

Sources of Interrupts in the AVR: AVR ATmega32 consist of **21** interrupt sources out of which **three** are **external**. The remaining are internal interrupts which supports the peripherals like USART, ADC, timer etc.

There are many sources of interrupts in the AVR, depending on which peripheral is connected to the chip. Some of the important interrupt sources are,

- Two interrupts for each **timer**.
- Three interrupts for **external** hardware which are **INT0, INT1**, and **INT2** respectively. The three external hardware interrupts are on pins **PD2, PD3, and PB2**
- **Serial communication** has 3 interrupts.
- **ADC** interrupts

Interrupt Service Routine(ISR): The program associated with the interrupt is called **interrupt service routine**. Generally for every interrupt there is a fixed location in memory that holds the address of ISR. This group of memory location set aside for handling ISR is called **interrupt vector table**.

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

Enabling and Disabling an interrupt: Upon reset, all interrupts are disabled. The interrupts must be enabled by software for the microcontroller to respond to them. The **D7** bit of the **SREG** (Status Register) is used to enable or disable interrupts globally.

Steps in enabling an Interrupts:

- Bit **D7** (**I – flag**) of SREG register must be set to HIGH to enable all interrupts.
- This is done with the SEI instruction.
- If **I=1**, each interrupts is enabled by setting to HIGH the **IE**(Interrupt Enable) flag bit for that interrupts.

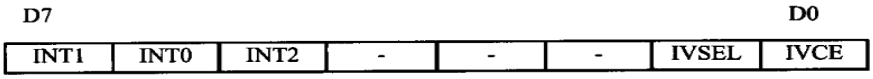
- The **TIMSK(Timer Interrupt Mask)** register is used to enable and disable different timer interrupts. These hardware interrupts are enabled by setting the 3 bits in the **GICR (General Interrupt Control Register).**

Enabling Timer Interrupts: The timer interrupts are handled using the **TOV**(Timer Overflow Flag) in the **TIFR** register and the **TOIE**(Timer Overflow Interrupt Enable) bit in the **TIMSK** register. We must enable both these interrupts flags to handle timer interrupts. Also we must enable or set the **global interrupt flag(I)** in the status register.

Timer Interrupt Flag Bits and Associated Registers

Interrupt	Overflow Flag Bit	Register	Enable Bit	Register
Timer0	TOV0	TIFR	TOIE0	TIMSK
Timer1	TOV1	TIFR	TOIE1	TIMSK
Timer2	TOV2	TIFR	TOIE2	TIMSK

Enabling External Hardware Interrupts: The ATMega32 has **3** external interrupts **INT0 (PORTD.2), INT1 (PORTD.3) and INT2 (PORTB.2)**. These hardware interrupts are enabled by setting the 3 bits in the **GICR (General Interrupt Control Register)**. These along with **global interrupt flag(I)** is set for an interrupt to be responded.



INT0 External Interrupt Request 0 Enable
INT1 External Interrupt Request 1 Enable
INT2 External Interrupt Request 2 Enable

Q) Explain how external interrupts are enabled & disabled in Atmega32? (4 marks)

Steps in executing an interrupt:

1. The peripheral device interrupts the processor.
2. Current instruction execution is completed.
3. The address of next instruction is stored on the stack from Program Counter(PC)
4. Address of ISR is loaded on the PC.
5. The processor executes the ISR.
6. The ISR execution completion is indicated by the **RETI(return from interrupt)** instruction
7. The processor loads the PC with the values stored on the stack and main program execution resumes.

Interrupt Priority: If 2 interrupt are activated at the same time, the interrupt with highest priority is served first. The priority of each interrupt is related to the address of that interrupt in the interrupt vector table**.** The interrupt having lower address, has a highest priority. Eg. INT0 has the highest priority & INT2 has least priority in case of external interrupts.

Interrupt Latency: It is the time that elapses between the occurrence of an interrupt and the execution of the first instruction of the ISR that handles the interrupt.

Q) Define interrupt priority. (1 mark)
Q) Define interrupt latency. (1 mark)
Q) Explain the hardware interrupt features in AVR. (3 marks)
Q) Explain diffbrent steps in executing an intemrpt in Almega32. (3 marks)

Interrupt C Programming:

Steps:

1) Interrupt header files(**#include <avr\interrupt.h>**) should be included to use interrupt in the program.

2) Instruction **cli( )** & **sei( )** are used to clear & set I bit of SREG register.

3) To write **ISR**(Interrupt Service Routine) for an interrupt, following structure is defined:

```
ISR(interrupt vector name)
{
        //our program
}
```

For example, the following ISR serves the Timer0 compare match interrupt:

```
ISR (TIMER0_COMP_vect)
{
}
```

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
}
```