CHAPTER 2

Fundamentals of AVR and Its Programming in C

Objectives ...

- To understand Classification of AVR Family.
- To study Architecture of AVR (ATmega 16/32).
- To learn Interfacing of various Devices to AVR and its Programming in 'C'.
- To learn Simple C Programs of Data Transfer Operation, Arithmetic Operation, Decision-making and Code Conversion.

INTRODUCTION

As discussed in previous chapter, the heart of the embedded system is microcontroller. There are various types of microcontrollers available in the market such as AVR, ARM, PIC etc. In this chapter, AVR microcontroller is discussed thoroughly including its architecture. The basic data types, operators, library files, functions of AVR are also discussed. The last part of this chapter consists of AVR programming in 'C' and its simple programs.

2.1 AVR ARCHITECTURE

2.1.1 Overview of AVR

- AVR is a family of microcontrollers developed by Atmel Corporation (now part of Microchip Technology) that is based on RISC (Reduced Instruction Set Computer) architecture.
- The basic architecture of AVR was designed by two students of Norwegian Institute of Technology, Alf-Egil Bogen and Vegard Wollan and then it was developed by Atmel in 1996.
- AVR can have different meanings. Atmel says, it is a product name but it might stand for Advanced Virtual RISC or Alf and Vegard RISC (the names of the AVR designers).
- AVR microcontrollers are widely used in embedded system design due to their simplicity, efficiency and ease of programming.
- ATmega16/32 is a low power CMOS 8-bit microcontroller based on RISC architecture. It can execute powerful instructions in single clock cycle.

2.1.2 Classification of AVR family

The AVR family is broadly classified based on features, memory size, processing speed and peripheral support. The classifications based on these parameters are given below:

1. **ATtiny Series (Tiny AVR):**
   It is small in size, has low pin count (6-20 pins), limited memory and peripherals. It is simple, cost effective and due to small size can be used for space constrained applications such as LED blinking, small sensors, simple control system.
   Examples of tiny AVR are ATtiny13, ATtiny85, ATtiny2313.
2. **ATmega Series (Mega AVR):**
   It has moderate to high memory (up to 256 KB flash), more I/O pins and rich peripheral set. It is mostly used by both beginners and professionals in applications like Arduino boards, robotics, embedded control systems.
   Examples of Mega AVR series are ATmega8, ATmega16, ATmega32, ATmega328P (used in Arduino Uno), ATmega2560.
3. **ATxmega Series (Extended AVR):**
   It is high performance series having advanced peripherals, larger memory (up to 384 KB), DMA, 16-bit timers. It is used in complex and high-speed embedded applications.
   Examples of extended AVR series are ATxmega128A1, ATxmega256A3.

The classification in tabular form is given below:

Series	Bit Width	Memory Range	Pin Count	Common Uses
ATtiny	8-bit	Low	6-20	Basic control, minimal systems
ATmega	8-bit	Moderate	28-100+	General-purpose, Arduino
ATxmega	8-bit	High	44-100+	Complex, high-speed systems

Classification of AVR family with chip names and important features is given is below table:

Series	Example Chips	Flash Size	Important Features
TinyAVR	ATtiny4, ATtiny85	0.5-8 KB	Small size, basic I/O
MegaAVR	ATmega16, ATmega328, 128	8-256 KB	Rich peripherals, widely used (Arduino)
AVR Dx	AVR64DA, AVR128DA	16-128 KB	Modern, high-performance, low-power
App-Specific	AT90CAN128, AT90USB1286	Varies	USB, CAN, Crypto support

2.2 AVR (ATMEGA16/32) ARCHITECTURE

The ATmega16 and ATmega32 are both 8-bit microcontrollers. They have almost the same architecture, pin configuration and peripherals. The only major difference is the size of the memory as mentioned in below table:

Feature	ATmega16	ATmega32
Flash Memory	16 KB	32 KB
SRAM	1 KB	2 KB
EEPROM	512 Bytes	1 KB
Program Memory Pages	128 Pages	256 Pages

2.2.1 Block Diagram of AVR

The block diagram of AVR (ATmega32) is given in below Fig. 2.1.

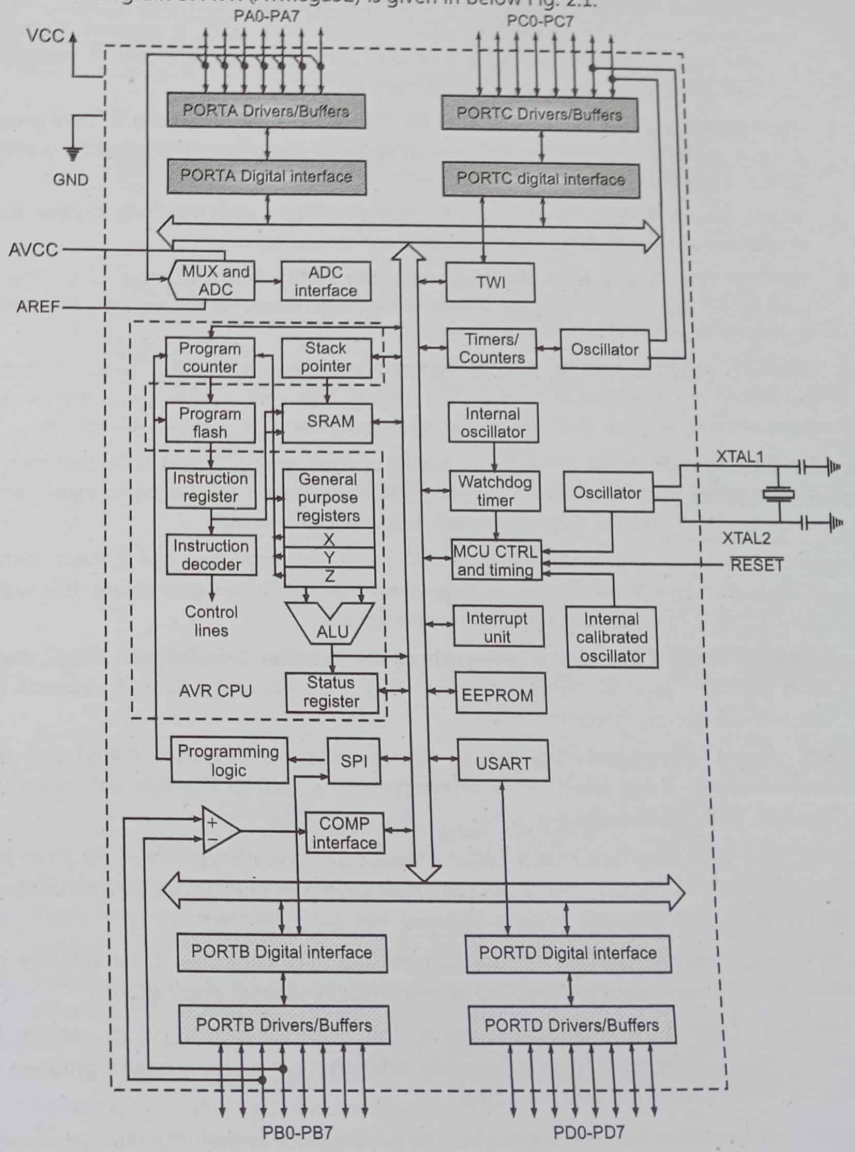


Fig. 2.1: ATmega 32 block diagram

Source: From datasheet of Atmel

The description of various blocks is given below:

1. **CPU (Central Processing Unit):** CPU is 8-bit RISC (Reduced Instruction Set Computer) core. It executes most of the instructions in one clock cycle. It contains 32 general-purpose working registers (R0-R31). It handles arithmetic, logic and control operations. It supports interrupt handling and efficient program control.
2. **Flash Memory:** Flash memory is of 32 KB (non-volatile memory) used to store program code. It is electrically erasable and reprogrammable. It supports In-System Programming (ISP) and In-Application Programming (IAP).
3. **SRAM (Static RAM):** SRAM is 2 KB volatile memory used for data storage during program execution. It stores stack, variables and temporary data.
4. **EEPROM (Electrically Erasable Programmable ROM):** It is 1 KB non-volatile memory used for permanent data storage. It retains data even when the power is off. It is used for storing calibration values, user settings, etc.
5. **I/O Ports (PORTA, PORTB, PORTC, PORTD):** It has four 8-bit bidirectional ports. It can be individually configured as input or output. Each port has its own Data Direction Register (DDR) and can control external peripherals like LEDs, switches, sensors etc.
6. **Timers/Counters:** It has Timer0 and Timer2 as 8-bit timers. Timer1 is 16-bit timer. It is used for delay generation, event counting, PWM generation and frequency measurement. It support input capture, output compare and overflow interrupt.
7. **ADC (Analog to Digital Converter):** ADC has 10-bit resolution and 8 input channels (PORTA). It converts analog signals (e.g., sensor output) into digital values. It is used in applications involving sensors and real-world signals.
8. **USART (Universal Synchronous/Asynchronous Receiver Transmitter):** USART enables serial communication with external devices (e.g., PCs, Bluetooth, GSM). It supports both synchronous and asynchronous modes.
9. **SPI (Serial Peripheral Interface):** SPI provides high-speed full-duplex serial communication. It has Master-Slave architecture. It is used to interface with devices like SD cards, EEPROM and displays.
10. **TWI (Two-Wire Interface / I2C):** TWI is a serial communication protocol using two lines: SDA (data) and SCL (clock). It enables communication with multiple peripherals using only two pins.
11. **Watchdog Timer:** It is safety feature to reset the microcontroller if the software gets stuck. It runs independently of the CPU. It helps in fault-tolerant applications.
12. **Interrupt Control:** It supports external and internal interrupts. It enables the microcontroller to respond quickly to events. Interrupts can pause the main program and execute an ISR (Interrupt Service Routine).
13. **Clock and Oscillator System:** It provides the system clock to the CPU and peripherals. It can use internal RC oscillator or external crystal/resonator. It affects the speed of program execution and timing accuracy.

14. **Power Management Unit:** It controls power-saving modes like Idle, Power-down and ADC noise reduction. It extends battery life in portable applications. It includes Brown-Out Detection (BOD) to prevent erratic behavior during voltage dips.

2.2.2 Architecture of ATmega32

The architecture of ATmega32 is given in below Fig. 2.2.

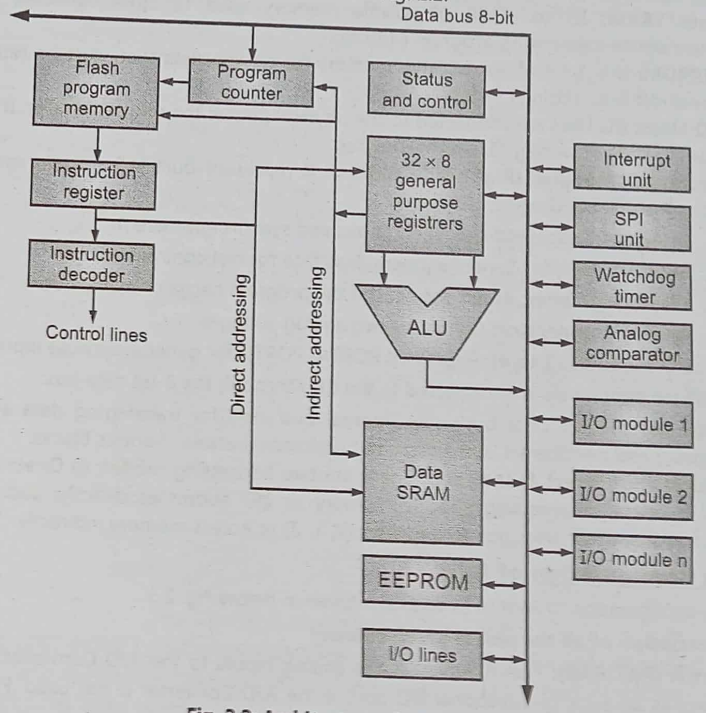


Fig. 2.2: Architecture of ATmega32

Source: From datasheet of Atmel

The functions of various blocks in the architecture are described below:

1. **Flash Program Memory:** It is non-volatile memory (32 KB in ATmega32). It stores the program code. It is directly accessed by the Program Counter.
2. **Program Counter (PC):** It holds the address of the next instruction to be executed. PC automatically increments after each instruction.
3. **Instruction Register and Decode:** Instruction Register holds the current instruction fetched from program memory. Instruction Decoder decodes the instruction and sends signals to control units and other blocks.
4. **32 × 8 General Purpose Registers:** These 32 registers (R0 to R31) are used for fast data manipulation and temporary storage. It is directly connected to the ALU for efficient processing.

5. **ALU (Arithmetic Logic Unit):** ALU performs all arithmetic and logical operations (addition, subtraction, AND, OR, shifts etc.). It is connected to the register file and contributes to decision-making operations.
6. **Status and Control:** It holds flags and status bits like zero, carry, overflow, etc., based on ALU results. It influences program control flow like branching.
7. **Data SRAM:** It has 2 KB of volatile memory used to store variables, stack and intermediate data during program execution.
8. **EEPROM:** It is 2 KB of non-volatile memory for storing data that must be retained after power-off (e.g., settings, logs).
9. **I/O Lines:** I/O lines are connected to the physical pins of the microcontroller. It is used for reading from or writing to external devices.
10. **Peripheral Modules (Right Side Blocks):** It represent built-in hardware modules that assist the main CPU which consists of:
    - **Interrupt Unit:** Handles asynchronous and synchronous events.
    - **SPI Unit:** Provides Serial Peripheral Interface for fast communication.
    - **Watchdog Timer:** Resets the MCU if the program hangs or malfunctions.
    - **Analog Comparator:** Compares two analog voltages.
    - **I/O Modules (1 to n):** Represent PORTA–PORTD for general-purpose input/output.

    All these peripherals are connected to the CPU through the 8-bit data bus.
11. **Data Bus (8-bit):** Data bus is an internal bus used for transferring data among CPU, memory and peripherals. It enables communication between various blocks.
12. **Addressing (Direct & Indirect):** There are two addressing modes: (i) Direct Addressing, which refers to accessing specific memory or I/O addresses directly; and (ii) Indirect Addressing, which uses pointer registers (X, Y, Z) to access memory indirectly.

2.2.3 Pin Configuration of AVR

The pin configuration of AVR (ATmega32) is given in below Fig. 2.3.

The description of all the pins is given below:

1. **Port A (PA7-PA0):** Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pin scan provide internal pull-up resistors (selected for each bit).
2. **Port B (PB7-PB0):** Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). Port B also serves the functions of various special features of the ATmega32 as listed below:
   (i) PB0 (XCK/T0): USART External Clock / Timer 0 Input
   (ii) PB1 (T1): Timer/Counter 1 Input
   (iii) PB2 (INT2/AIN0): External Interrupt 2 / Analog Comparator Input
   (iv) PB3 (OC0/AIN1): Output Compare for Timer 0 / Analog Comparator Input
   (v) PB4 (SS): SPI Slave Select
   (vi) PB5 (MOSI): SPI Master Out Slave In
   (vii) PB6 (MISO): SPI Master In Slave Out
   (viii) PB7 (SCK): SPI Clock

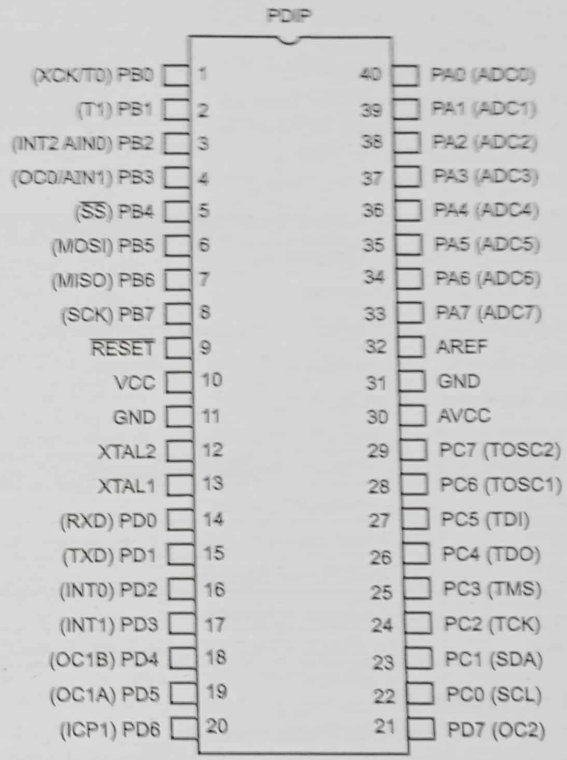


Fig. 2.3: Pin configuration of ATmega32

3. **Port C (PC7-PC0):** Port C is an 8-bit bi-directional I/O port with internal pull-up resistors. Port C also serves the functions of the JTAG interface and other special features of the ATmega32 as listed below:
   (i) PC0 (SCL): I2C Clock
   (ii) PC1 (SDA): I2C Data
   (iii) PC2 (TCK): JTAG Test Clock
   (iv) PC3 (TMS): JTAG Test Mode Select
   (v) PC4 (TDO): JTAG Test Data Out
   (vi) PC5 (TDI): JTAG Test Data In
   (vii) PC6 (TOSC1): Timer Oscillator Input
   (viii) PC7 (TOSC2): Timer Oscillator Output
4. **Port D (PD7-PD0):** Port D is an 8-bit bi-directional I/O port with internal pull-up resistors. Port D also serves the functions of various special features of the ATmega32 as listed below:
   (i) PD0 (RXD): USART Receive

(ii) PD1 (TXD): USART Transmit
(iii) PD2 (INT0): External Interrupt 0
(iv) PD3 (INT1): External Interrupt 1
(v) PD4 (OC1B): Output Compare for Timer 1B
(vi) PD5 (OC1A): Output Compare for Timer 1A
(vii) PD6 (ICP1): Input Capture for Timer 1
(viii) PD7 (OC2): Output Compare for Timer 2

5. **RESET (Pin 9):** Active-low input to reset the microcontroller.
6. **VCC (Pin 10):** Supply voltage (+5V typically).
7. **GND (Pin 11 & 32):** Ground reference.
8. **XTAL1, XTAL2 (Pins 12, 13):** External crystal oscillator connections.
9. **AVCC (Pin 30):** Power for ADC (Analog-to-Digital Converter).
10. **AREF (Pin 32):** Reference voltage for ADC.

The summary of all pins and their primary functions is given in below table.

Port	Primary Functions
PORT A	ADC inputs (ADC0-ADC7)
PORT B	SPI, Timer, Interrupt, Analog Comparator
PORT C	I2C, JTAG, Timer Oscillator
PORT D	USART, Timer, Interrupt
Control	RESET, XTAL1, XTAL2, VCC, GND, AREF, AVCC

2.2.4 Features of ATmega AVR

1. High-performance, Low-power, 8-bit Microcontroller.
2. Advanced RISC Architecture.
3. Executes most instructions in a single clock cycle.
4. Supports 131 powerful instructions.
5. 8-bit CPU with 32 general-purpose registers.
6. High-speed instruction execution: Up to 16 MIPS at 16 MHz.
7. Flash Non-volatile Memory (Program Memory) ,E.g., 16 KB (ATmega16), 32 KB (ATmega32).
8. SRAM, 1 KB (ATmega16), 2 KB (ATmega32).
9. Supports ISP (In-System Programming) via SPI interface.
10. Allows programming without removing the chip from the circuit.
11. Three timers: Two 8-bit and one 16-bit.
12. Supports PWM (Pulse Width Modulation) modes.
13. USART for serial communication; SPI and TWI (I2C) interfaces.
14. Analog Comparator; 10-bit ADC with multiple channels (8 channels in ATmega32).
15. Watchdog Timer for system reset on fault.

16. Low power consumption.
17. Multiple sleep modes (Idle, ADC Noise Reduction, Power-down, etc.)
18. Operates from 2.7 V to 5.5 V.
19. Supports external and internal interrupts: Each peripheral has its own interrupt vector.
20. Available in various packages: PDIP, TQFP, QFN.

Below table gives summary of the features:

Table 2.1

Feature	Description
Architecture	Modified Harvard (RISC)
CPU Speed	Up to 16 MIPS
Program Memory	16-32 KB Flash
Data Memory	1-2 KB SRAM, 512 B-1 KB EEPROM
Timers/Counters	2×8-bit, 1×16-bit
Communication Interfaces	USART, SPI, I2C
ADC	8-channel, 10-bit
Operating Voltage	2.7 V to 5.5 V
Power Management	Multiple sleep modes
Interrupts	21+ interrupt sources

2.3 AVR PROCESSOR MEMORY MAP

AVR microcontrollers have a Harvard architecture, meaning they have separate memory spaces for program code and data.

Thus, the AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the ATmega32 features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

2.3.1 AVR Program Memory

Program Memory Map is given in below Fig. 2.4.

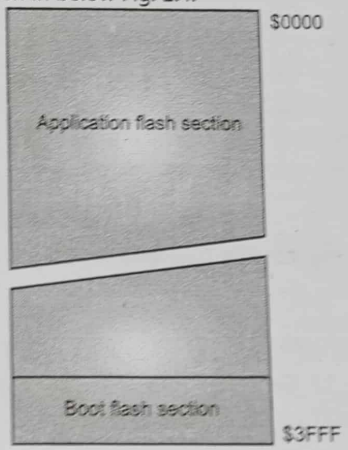


Fig. 2.4: AVR Program Memory Map

The size of Program Memory (Flash Memory) is 16 KB for ATmega16 and 32 KB for ATmega32. It stores the executable program code (machine instructions). It is Read-only memory during normal operation, written during programming. Its address Range is from 0x0000 to 0x3FFF for ATmega16 and from 0x0000 to 0x7FFF for ATmega32.

2.3.2 AVR Data Memory

AVR Data Memory is shown in Fig. 2.5.

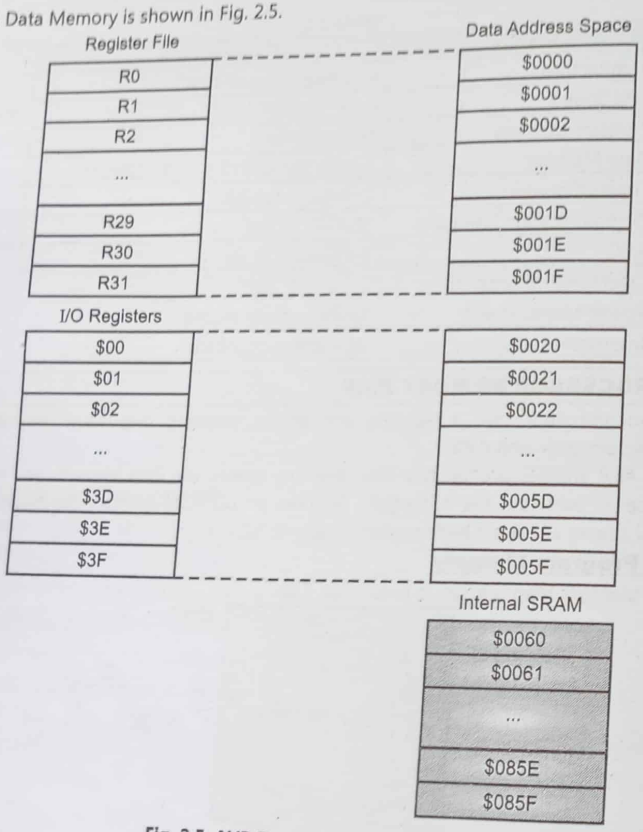


Fig. 2.5: AVR Data Memory Map

Source: From data sheet of Atmel.

The Data Memory is divided into several sections as given below:

(a) Register File: There are 32 General Purpose Registers (R0 to R31) in ATmega32. Its address range is from 0x00 to 0x1F. They are used by the ALU for arithmetic/logical operations.

(b) I/O Registers: There are 64 bytes for controlling peripherals like timers, UART, SPI, etc. Its address range from 0x20 to 0x5F. They are accessible using IN and OUT instructions.

(c) Extended I/O Registers: These are additional control registers having address range from 0x60 to 0xFF (depending on the device).

(d) Internal SRAM: It is used for runtime data storage. Its size is 1 KB in ATmega32 and its address range starts from 0x100 onwards.

EEPROM:

AVR has additional memory, EEPROM (Electrically Erasable Programmable ROM) for data storage. Its size is 512 bytes in ATmega16 and 1 KB in ATmega32. It stores non-volatile data (e.g., calibration values, settings). It can be accessed via special registers, not directly mapped into data space.

AVR also has Special Memory Sections such as Stack which is allocated in SRAM and it is used for function calls and interrupts. Interrupt Vectors are located at the beginning of program memory (0x0000 onwards).

Below summary table gives the details of AVR memory sections and their functions:

Table 2.2

Memory Section	Range/Size	Function
Program Memory	16-32 KB (Flash)	Stores program instructions.
Register File	0x00-0x1F	32 general purpose CPU registers.
I/O Registers	0x20-0x5F	Control of peripherals.
Extended I/O	0x60-0xFF	Extra control registers.
SRAM	From 0x100	Data storage, stack, runtime variables.
EEPROM	512 B-1 KB	Non-volatile data (via special access).

2.4 CPU REGISTERS

The AVR CPU registers are key components in AVR which support data processing, arithmetic and control operations. They are fast-access memory locations used during execution.

2.4.1 General Purpose Registers

There are 32 general purpose working registers in the AVR CPU as shown in Fig. 2.6.

As shown in Fig. 2.6, there are total 32 registers having width 8-bit each. They are used for data manipulation, arithmetic, logic operations. They are grouped into R0-R15 which are directly used for data and R16-R31 which are often used for immediate values in instructions.

Although these registers are not being physically implemented as SRAM locations, this memory organization provides great flexibility in access of the registers, as the X-, Y- and Z-pointer Registers can be set to index any register in the file.

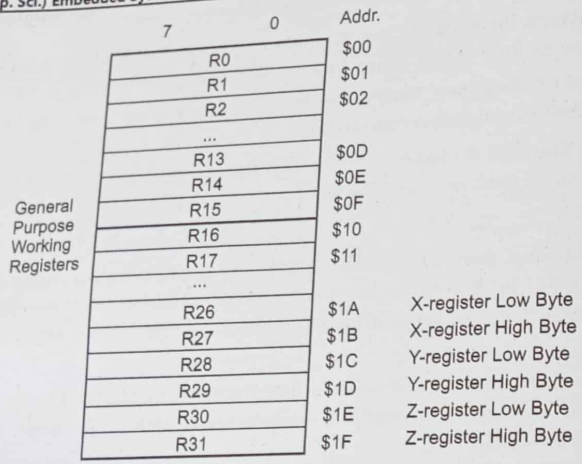


Fig. 2.6: 32 General purpose working registers in the CPU

Source: From datasheet of Atmel.

The registers R26 - R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the Data Space. The three indirect address registers X, Y and Z are defined as described in Fig. 2.7.

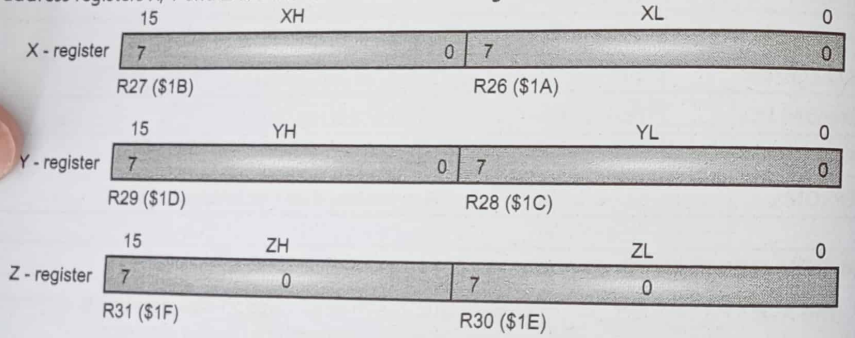


Fig. 2.7: The X-, Y- and Z- registers

Source: From datasheet of Atmel.

Register pairs R26:R27, R28:R29 and R30:R31 serve as pointer registers as below:

- X Register: R26 - R27
- Y Register: R28 - R29
- Z Register: R30 - R31

They are used for indirect addressing (important for accessing memory efficiently).

These address registers have functions as fixed displacement, automatic increment and automatic decrement in the different addressing modes.

2.4.2 Status Register (SREG)

The Status Registers are 8-bit register that holds flags set or cleared by the CPU during operations. It affects branching and decision-making instructions.

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations.

The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

AVR status register is shown in Fig. 2.8.

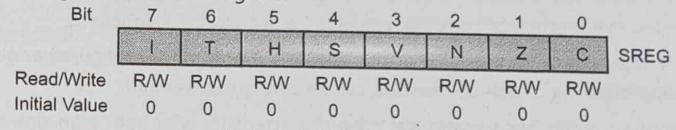


Fig. 2.8 : 8-bit status register

Source: From datasheet of Atmel.

The function of each bit along with its flag is given in below table:

Bit	Flag	Name	Description
7	I	Global Interrupt Enable	Enables/disables all interrupts.
6	T	Bit Copy Storage	Used with BST (Bit Store) and BLD (Bit Load) instructions.
5	H	Half Carry Flag	Carry from bit 3 to bit 4 (used in BCD arithmetic).
4	S	Sign bit	S = N ⊕ V, used to detect signed overflow.
3	V	Two's Complement Overflow Flag	Set on two's complement overflow.
2	N	Negative Flag	Set if result is negative (MSB = 1).
1	Z	Zero Flag	Set if result is zero.
0	C	Carry Flag	Set if result generated a carry out of MSB.

2.5 ALU

- The Arithmetic Logic Unit (ALU) is a core component of the CPU, responsible for performing arithmetic and logical operations. It plays a vital role in data processing and program execution.
- The main function of ALU is to execute all the mathematical and logical instructions that are part of the instruction set. These operations are essential for decision-making, looping, branching and general data manipulation.
- ALU can perform arithmetic operations like addition, subtraction, increment, decrement, multiplication (in higher-end AVR models). ALU can also performs logical operations like AND, OR, XOR, NOT, Bit shifting (left/right). ALU can compare two results and based on comparison results flags (Zero, Carry, Negative etc.) can be set or reset. ALU can also perform bit operations like set, clear, toggle and test individual bits.

- ALU in AVR uses general-purpose registers (R0-R31) to fetch operands. After performing operations, it updates the Status Register (SREG) with flag values. ALU affects Carry, Zero, Negative, Overflow, Sign and Half carry flags.
- The ALU is fully integrated within the CPU and executes most instructions in a single clock cycle.

2.6 I/O PORTS

- All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports, means the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the SBI and CBI instructions.
- This is applicable while changing drive value also. If port pin is configured as output, then enabling/disabling of pull-up resistors, can be configured as input.
- Each output buffer has symmetrical drive characteristics with both high sink and source capability.
- The pin driver of AVR is strong enough to drive LED displays directly.
- All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance as shown in Fig. 2.9.

All I/O pins have protection diodes to both V_{CC} and Ground as indicated in Fig. 2.9.

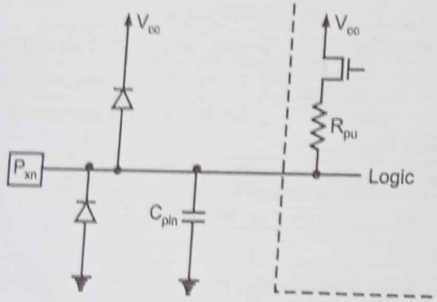


Fig. 2.9: Schematic of I/O pin equivalent

- ATmega16/32 has 4 I/O ports as PORTA, PORTB, PORTC, PORTD. Each port has 8 individual pins. So, there are total 32 I/O pins. They can be configured as input or output pins.
- Each port is controlled by three special registers: DDR (Data Direction Register), PORT (Output Register), PIN (Input Register).
- DDR sets direction of each pin (1 = Output, 0 = Input), PORT writes HIGH/LOW to output pins or enables pull-up on input pins, PIN reads logic level from input pins.
- These 3 registers for PORT A are shown in Fig. 2.10.

Port A Data Register - PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
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	

Port A Data Direction Register - DDR

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
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	

Part A Input Pins Address - PINA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

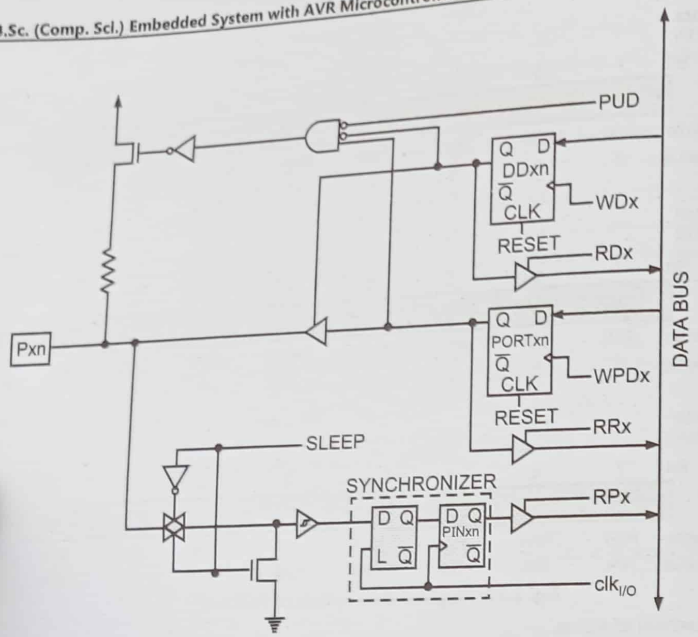
Fig. 2.10: Register description of PORT A

Source: From datasheet of Atmel.

As shown in Fig. 2.10, the structure of these registers is same for PORT B, C and D.

Pin Configuration:

- Using the I/O port as General Digital I/O is shown in Fig. 2.11.
- As shown in Fig. 2.11, each port pin (PORTxn) behavior depends on the combination of DDxn (Data Direction bit) and PORTxn (Output or Pull-up bit).
- If PORTxn = 1, the internal pull-up resistor is activated. If PORTxn = 0, the pin is set to high impedance (floating). To disable the pull-up, either, write PORTxn = 0, or set DDxn = 1 (configure as output).
- If PORTxn = 1, the pin is driven high (logic 1) and if PORTxn = 0, the pin is driven low (logic 0).
- When switching from tri-state ({DDxn=0, PORTxn=0}) to output high ({DDxn=1, PORTxn=1}): pass through an intermediate state, either, input with pull-up ({DDxn=0, PORTxn=1}), or output low ({DDxn=1, PORTxn=0})
- To disable all pull-up resistors globally: set the PUD (Pull-up Disable) bit in the SFIOR register.
- Switching directly between input with pull-up and output low (i.e., {DDxn=0, PORTxn=1} to {DDxn=1, PORTxn=0}) may cause unintended behavior. In such cases, use one of the following as an intermediate step: tri-state mode: {DDxn=0, PORTxn=0}, output high: {DDxn=1, PORTxn=1}



PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
$clk_{I/O}$: I/O CLOCK

WDx: WRITE DDRx
RDx: READ DDRx
WPx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN

Note: 1. WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

Fig. 2.11: General digital I/O

Source: From datasheet of Atmel.

- Port Pin modes are given in below table:

Mode	DDR Bit	PORT Bit	Behavior
Input	0	0	High Impedance
Input + Pull-up	0	1	Input with internal pull-up
Output	1	X	Drives logic HIGH or LOW

- Many I/O pins also serve alternate functions like ADC input, PWM output, Serial communication (USART, SPI, TWI), External interrupts. The alternate function is activated automatically when the corresponding peripheral is enabled.

- The summarized PORTs and their functions are given below table.

Port	Functions/Pins	Common Uses
PORTA	ADC, General I/O	Analog inputs, LCD, etc.
PORTB	SPI, Timer, I/O	SPI interface, PWM, switches
PORTC	JTAG, I2C, I/O	Sensor interfaces, data buses
PORTD	USART, External Int.	Serial comms, user inputs, LEDs

2.7 PERIPHERALS IN AVR

AVR microcontrollers have a range of on-chip peripherals that enable interfacing with external devices and performing various tasks without needing additional hardware. The on-chip peripherals are listed below:

1. **General Purpose I/O (GPIO):**
   - 32 I/O lines divided into PORTA, PORTB, PORTC and PORTD.
   - Pins can be individually configured as input or output.
   - Supports internal pull-up resistors.
2. **Timers/Counters:**
   - 3 timers; Timer0: 8-bit, Timer1: 16-bit, Timer2: 8-bit.
   - Normal Mode, CTC (Clear Timer on Compare), PWM and Fast PWM.
   - Used for delays, wave generation, event counting and timing applications.
3. **ADC (Analog-to-Digital Converter)**
   - 10-bit resolution.
   - 8 channels (in ATmega32) via PORTA.
   - Converts analog signals (0-5 V) to digital values (0-1023).
   - Used for reading sensors, analog inputs, etc.
4. **USART (Universal Synchronous/Asynchronous Receiver Transmitter):**
   - Enables serial communication (RS232).
   - Used for communication with PCs, GSM, GPS, Bluetooth, etc.
   - Supports both synchronous and asynchronous modes.
5. **SPI (Serial Peripheral Interface):**
   - Full-duplex synchronous communication.
   - Master-slave architecture.
   - Commonly used to interface with EEPROMs, SD cards, RTCs and other microcontrollers.
6. **TWI (Two Wire Interface – I2C Compatible):**
   - Used for serial communication with a limited number of wires.
   - Supports multi-master and slave communication.
   - Suitable for connecting sensors, RTCs, displays, etc.

7. **Watchdog Timer (WDT):**
   - A fail-safe timer that resets the system if it crashes or hangs.
   - Helps in building reliable, fault-tolerant systems.
8. **Analog Comparator:**
   - Compares two analog voltages and sets a flag based on comparison.
   - Useful in applications like zero-crossing detection, battery monitoring etc.
9. **EEPROM (Electrically Erasable Programmable Read-Only Memory):**
   - 512 bytes to 1 KB (varies by model).
   - Stores non-volatile data, such as configuration parameters.
10. **Interrupt System:**
    - 21+ interrupt sources.
    - Internal and external interrupt handling.
    - Prioritized vector addresses for each interrupt source.
11. **JTAG Interface (ATmega32):**
    - Used for on-chip debugging and boundary scan.
    - Enables real-time debugging through external debuggers.

2.8 PROGRAMMING OF AVR IN C

To write programs for AVR microcontrollers in C (using tools like AVR-GCC or Atmel Studio/Microchip Studio) allows low-level hardware control with readability and portability.

2.8.1 Basic Structure

Basic structure of AVR 'C' program is given below:

```
#include <avr/io.h>          // Include AVR device-specific definitions
#include <util/delay.h>      // Include delay functions (if needed)
int main(void)
{                            // 1. Initialization
   DDRB |= (1 << PB0);       // Set PB0 as output (e.g., LED pin)
                             // 2. Main Loop
   while (1)
   {
      PORTB |= (1 << PB0);   // Set PB0 HIGH (turn ON LED)
      _delay_ms(500);        // Wait for 500 milliseconds
      PORTB &= ~(1 << PB0);  // Set PB0 LOW (turn OFF LED)
      _delay_ms(500);        // Wait for 500 milliseconds
   }
   return 0;
}
```

- #include <avr/io.h> includes device-specific register definitions. It provides access to registers like DDRB, PORTB, etc.
- #include <util/delay.h>, enables use of delay functions such as _delay_ms() and _delay_us() for timing control.

- DDRx Registers (Data Direction Registers) used to configure each pin as input or output. Example: DDRB |= (1 << PB0); sets pin PB0 as output.
- PORTx Registers used to write logic levels (HIGH/LOW) to output pins. Also used to enable internal pull-up resistors on input pins.
- PINx Registers used to read digital input values from pins. Example: if (PINB & (1 << PB0)) checks if PB0 is HIGH.
- while(1) Loop represents an infinite loop. It keeps the program running continuously, executing the main tasks repeatedly. It is essential for most embedded programs.

2.8.2 Data Types

- When programming AVR microcontrollers in C (using AVR-GCC), it is important to use data types efficiently, especially due to limited memory and processing power.
- There are various data types available in 'C' such as int, char, float, double, void, derived types as arrays, pointers, functions and user defined data types as structure, union, enum.

Standard C Data Types used in AVR Programming are given in below table.

Table 2.3

Data Type	Size (bits)	Range	Format Specifier	Use Case
char	8	−128 to +127 (signed)	%c	Characters, small integers
unsigned char	8	0 to 255	%u	Bit-level operations, ports
int	16	−32,768 to +32,767	%d	General-purpose signed integers
unsigned int	16	0 to 65,535	%u	Positive-only counters, timers
long	32	−2,147,483,648 to +2,147,483,647	%ld	Large signed integers
unsigned long	32	0 to 4,294,967,295	%lu	Large positive-only numbers
float	32	Approx. ± 3.4E + 38	%f (in printf)	Real numbers (use sparingly)
double (same as float on AVR)	32	Same as float	%f	Real numbers

The data types used in AVR programming are listed below:

1. **uint8_t – Unsigned 8-bit Integer:** Its range is 0 to 255. It is used for port manipulation, small counters, flag values.

   Example:

```
uint8_t led = 1;
DDRB |= (1 << PB0);          // Set PB0 as output
PORTB |= (1 << PB0);         // Turn ON LED
```

2. **int8_t – Signed 8-bit Integer:** Its range is – 128 to 127. It is used for signed values like temperature offset, small negative numbers.

   Example:

   ```
   int8_t temp_offset = -5;
   ```

3. **uint16_t – Unsigned 16-bit Integer:** Its range is 0 to 65535. It is used for ADC readings, timer values, delay counters.

   Example:

   ```
   uint16_t adc_result = ADC;      // Read ADC value (10-bit resolution)
   ```

4. **int16_t – Signed 16-bit Integer:** Its range is – 32,768 to 32,767. It is used in programming the control systems, signed calculations.

   Example: `int16_t error = -100;      // PID error value`

5. **uint32_t – Unsigned 32-bit Integer:** Its range is 0 to 4,294,967,295. It is used in long counters, time tracking.

   Example: `uint32_t pulse_count = 0;`

6. **float – 32-bit Floating Point:** Its range is 3.40282×10^{38} (approx.), 6-7 decimal digits precision. It is used for real numbers, sensor values with decimals (use sparingly).

   Example:

   ```
   float voltage = 5.0;
   float current = 0.12;
   float power = voltage * current;
   ```

- As AVR is an 8-bit microcontroller, so 8-bit operations are faster and more efficient.
- For speed and memory efficiency, uint8_t, uint16_t are preferred.
- Usually, float data types are avoided, unless decimal precision is required, as they are slow and consumes more memory.
- For direct port manipulation, usually, unsigned char or uint8_t (from <stdint.h>) is used.

2.8.3 Operators

- A 'C' operator is the symbol that helps to perform some mathematical, bitwise, conditional or logical, relational computations on values and variables.
- The values and variables used with operators are known as operands.
- Operators are the symbols that perform operations on operands.

Different Types of Operators used in 'C' are listed below:

1. **Arithmetic Operators:** It is used to perform mathematical operations. The arithmetic operators are given in below table.

Table 2.4

Operator	Description	Example
+	Addition	a + b
–	Subtraction	a – b
*	Multiplication	a × b
/	Division	a / b
%	Modulus (remainder)	a % b

2. **Assignment Operators:** It is used to assign values to variables. They are listed in below table.

Table 2.5

Operator	Description	Example
=	Assign	a = 5
+=	Add and assign	a += 3
-=	Subtract and assign	a –= 2
*=	Multiply and assign	a *= 4
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 3
&=	AND and assign	a &= b
|=	OR and assign	a |= b
^=	XOR and assign	A ^= b
>>=	Right shift and assign	a >>= b
<<=	Left shift and assign	a <<= b

3. **Comparison (Relational) Operators:** They are used for conditional expressions. They are listed in below table.

Table 2.6

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

4. **Logical Operators:** Logical operators are used to combine conditions as listed in below table.

Table 2.7

Operator	Description	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	Logical NOT	!a

The result of the operation of a logical operator is a Boolean value either **true** or **false**.

5. **Bitwise Operators:** They are used for low-level or bit-level operations on operands. The operators are first converted to bit-level and then the calculations are performed on operands. They are important in embedded system. Bitwise operators are given in below table.

Table 2.8

Operator	Description	Example
&	Bitwise AND	a & b
|	Bit wise OR	a | b
^	Bitwise XOR	a ^ b
~	Bitwise NOT or complement	~a
<<	Bit wise Left shift	a << 2
>>	Bit wise Right shift	a >> 1

6. **Conditional (Ternary) operators:** These are conditional operators used in place of if...else.

Format	Example
condition ? x : y	a > b ? a : b

2.8.4 Library Files

Library files provide predefined functions, macros and definitions that make it easier to control hardware like I/O ports, timers, delays and interrupts.

Common library files used in AVR 'C' programming are listed below:

1. **#include <avr/io.h>**

   It includes definitions for I/O registers, port names (like PORTB, DDRB) and pin control. It is mandatory to use for accessing microcontroller-specific hardware features.

2. **#include <util/delay.h>**

   It provides delay functions like _delay_ms() and _delay_us(). It is used for generating time delays in code.

3. **#include <avr/interrupt.h>**

   It enables and manages interrupts in AVR. It is used for ISR (Interrupt Service Routines).

4. **#include <avr/sfr_defs.h>**

   It contains macros and bit manipulation tools for special function registers. It is used internally in other AVR headers for register and bit definitions.

5. **#include <avr/eeprom.h>**

   It allows reading/writing to EEPROM memory. It is used for saving data that must persist after power off.

6. **#include <avr/pgmspace.h>**

   It helps store constant data in flash memory instead of SRAM. It is useful for saving RAM by placing data like strings in program memory.

7. **#include <stdlib.h> and #include <stdio.h>**

   They are Standard C libraries. They are used for memory allocation, formatted I/O, string conversions, etc.

   Functions like printf() require special setup to work with UART in AVR.

2.8.5 Delay Functions

The delays are essential for timing operations like blinking LEDs, waiting between inputs, or generating pulses. The delay functions are provided through the <util/delay.h> library.

#include <util/delay.h> has to be used at the beginning of the program to include delay functions.

To create a time delay in AVR C, there are three ways as follows:

1. **Using a Simple 'for' Loop:**

   By using nested 'for' loop, delay can be created in 'C'. While creating delays using for loop, two things are important and have to be considered carefully.

   - In time delay calculations, crystal frequency XTAL 1 and XTAL 2 is important as the duration of the clock period for the instruction cycle is a function of this crystal frequency.
   - The second factor which affects the time delay is the compiler used to compile AVR program. In assembly language program, the exact instruction and their sequence used in delay subroutine can be controlled. However, in C program, it is the compiler which converts the C statements and functions to assembly language instructions. So different compilers can produce different codes, i.e. different compilers can produce different hex codes for the same C program.

2. **Using predefined C Functions:**

   The AVR library provides two key functions:

   1. **_delay_ms(double __ms)**

      It delays execution for a specified number of milliseconds. __ms is the time in milliseconds (e.g 100 for 100 ms)

      Example: _delay_ms(500); // Delay of 500 milliseconds

   2. **_delay_us(double __us)**

      It delays execution for a specified number of microseconds. __us is the time in microseconds.

      Example: _delay_us(100); // Delay of 100 microseconds

      These functions are blocking delays, the microcontroller does nothing else during this time. This is the disadvantage of this delay method.

3. **Using AVR Timers:**

   - To generate delays, AVR timers can also be used. It is known as non-blocking delays.
   - A non-blocking delay allows microcontroller to perform other tasks while waiting for a delay to expire, unlike _delay_ms() which halts everything. This can be done using hardware timers with interrupts.

Through the example, how non-blocking delay can be created is explained below:

Example: Toggle LED every 1 second using Timer 0.

Timer 0 in CTC mode (Clear Timer on Compare Match) with interrupts is used to create a delay without blocking the CPU.

Assume ATmega16/32 with clock 1 MHz.

Program for LED blinking is as follows:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
volatile uint16_t counter = 0;
int main(void)
{
   DDRB |= (1 << PB0);    // Set PB0 as output
                          // Configure Timer0
   TCCR0 = (1 << WGM01); // CTC mode
   OCR0 = 156;           // Compare value for ~10 ms at 1 MHz and 64 prescaler
   TIMSK = (1 << OCIE0);
                          // Enable compare match interrupt
   TCCR0 |= (1 << CS01) | (1 << CS00);
                          // Start timer with 64 prescaler
   sei();                 // Enable global interrupts
   while (1)
   {
      if (counter >= 100)
      {                   // 100 * 10ms = 1s
         PORTB ^= (1 << PB0);  // Toggle LED
         counter = 0;
      }
   }
}
ISR(TIMER0_COMP_vect)
{
   counter++;   // Increment every 10ms
}
```

- **TCCR0:** Timer0 Control Register is used to configure the timer's mode and clock source.
- **WGM01:** Enables CTC (Clear Timer on Compare Match) mode, which resets the timer when it matches the value in OCR0.

- **OCR0 = 156:** Sets the compare value. At 1 MHz clock with a 64 prescaler. This results in an interrupt of approximately every 10 milliseconds.
- **TIMSK:** Timer Interrupt Mask Register is used to enable the Timer 0 Compare Match Interrupt.
- **ISR():** Interrupt Service Routine. This function is executed automatically every time the timer reaches the compare value (every 10 ms in this example).
- **counter >= 100:** This condition triggers a 1-second event (100 × 10ms = 1000ms = 1s), used here to toggle an LED or perform another task.

Timer programming is discussed in AVR Timer programming section.

2.8.6 Bitwise Operators

Bitwise operators are given in below Table 2.9.

Table 2.9

Operator	Symbol	Meaning	Example Syntax	Description
AND	&	Bitwise AND	a = b & c;	Bits set to 1 only if both are 1
OR	|	|	Bitwise OR	`a = b
XOR	^	Bitwise XOR	a = b ^ c;	Bits set to 1 if only one is 1
NOT	~	Bitwise Complement	a = ~b;	Inverts all bits (1→0, 0→1)
LEFT SHIFT	<<	Shift bits left	a = b << 2;	Multiplies by 2^2 = 4
RIGHT SHIFT	>>	Shift bits right	a = b >> 1;	Divides by 2

Common Examples in AVR Microcontroller Programming:

1. **Set a Bit (Make Bit = 1)**

   PORTB |= (1 << PB0); // Set bit 0 of PORTB

2. **Clear a Bit (Make Bit = 0)**

   PORTB &= ~(1 << PB0); // Clear bit 0 of PORTB

3. **Toggle a Bit (Invert its value)**

   PORTB ^= (1 << PB0); // Toggle bit 0 of PORTB

4. **Check if a Bit is Set (Bit = 1)**

```
if (PINB & (1 << PB0))
{
   // Bit 0 is HIGH
}
```

5. **Shift a Byte Left (Multiply by 2)**

   uint8_t a = 5;

   uint8_t result = a << 1; // result = 10

6. **Shift a Byte Right (Divide by 2)**

```
uint8_t a = 8;
uint8_t result = a >> 1;          // result = 4
```

2.9 SIMPLE C PROGRAMS

As explained in 2.8.1, a simple C program can be written.

In this section, C programs of data transfer, arithmetic operation, decision making and code conversions are discussed.

2.9.1 C Program for Data Transfer Operation

A simple AVR C program for data transfer operation is the transferring a value from one port to another.

Example: Read input from PORTD and write it to PORTB (e.g., switches to LEDs)

```
#define F_CPU 1000000UL
#include <avr/io.h>
int main(void)
{
    DDRD = 0x00;        // Configure PORTD as input (all pins)
    PORTD = 0xFF;       // Enable internal pull-up resistors on PORTD
    DDRB = 0xFF;        // Configure PORTB as output (all pins)
    while (1)
    {
        PORTB = PIND;   // Transfer the data from PORTD to PORTB
    }
}
```

In this program, all PORTD (DDRD) pins are defined as input pins and all PORTB pins (DDRB) are defined as output pins. PIND reads the value from PORTD input pins and writes it to PORTB output pins.

In most of the programs, these data transfer operations are required. For e.g., reading switch states and turning on LEDs, passing sensor input to output devices, debugging GPIO configuration etc.

2.9.2 C Program for Arithmetic Operation

A simple AVR C program to perform basic arithmetic operations like addition, subtraction, multiplication and division using integer variables and output the result via an output port (e.g., PORTB) is given below:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
```



```
int main(void)
{
    // Variables for arithmetic
    uint8_t a = 20;
    uint8_t b = 10;
    uint8_t result;
    // ConFig. 2.PORTB as output to show result
    DDRB = 0xFF;
    // Example: Addition
    result = a + b;
    PORTB = result;     // Output result to PORTB
    _delay_ms(1000);
    // Example: Subtraction
    result = a - b;
    PORTB = result;
    _delay_ms(1000);
    // Example: Multiplication
    result = a * b;
    PORTB = result;
    _delay_ms(1000);
    // Example: Division (check for divide by zero)
    if (b != 0)
    {
        result = a / b;
        PORTB = result;
        _delay_ms(1000);
    }
    while (1)
    {
        // Keep displaying the last result
    }
}
```

In this program, a and b are the operands for the arithmetic operations. The result stores the output of each operation which is send to PORTB. The function _delay_ms() waits 1 second between operations to visualize each result clearly.

2.9.3 Decision Making AVR C Program

For decision making, if, if-else, and switch statements are used.

AVR C Program:

```
LED ON/OFF Based on Push Button Status (if-else decision)
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
   DDRC |= (1 << PC0);              // Set PC0 as output (LED)
   DDRD &= ~(1 << PD0);             // Set PD0 as input (Button)
   PORTD |= (1 << PD0);             // Enable internal pull-up on PD0
   while (1)
   {
      if (!(PIND & (1 << PD0)))     // If button is pressed (active low)
      {
         PORTC |= (1 << PC0);          // Turn ON LED
      }
      else
      {
         PORTC &= ~(1 << PC0);         // Turn OFF LED
      }
   }
}
```

Switch Case to Control LED Pattern Based on Command:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
void blink_LED(uint8_t times)
{
   for (uint8_t i = 0; i < times; i++)
   {
      PORTC ^= (1 << PC0);   // Toggle LED
      _delay_ms(300);
   }
}
```



```
int main(void)
{
   DDRC |= (1 << PC0);     // PC0 as output (LED)
   uint8_t command = 2;    // Simulated input, can be from UART or switch
   switch (command)
   {
      case 1:
         PORTC |= (1 << PC0);       // LED ON
         break;
      case 2:
         PORTC &= ~(1 << PC0);      // LED OFF
         break;
      case 3:
         blink_LED(5);              // Blink 5 times
         break;
      default:
         // Do nothing
         break;
   }
   while (1);                       // Infinite loop
}
```

2.9.4 Code Conversion AVR C Program

Decimal to binary and hexadecimal programs are discussed here.

AVR C Program for Decimal to Binary Conversion:

Example: Convert a decimal number (e.g., 13) into its binary equivalent and store it in a string.

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
void decimal_to_binary(uint8_t decimal, char *binaryStr)
{
   for (int i = 7; i >= 0; i--)
   {
      binaryStr[7 - i] = (decimal & (1 << i)) ? '1' : '0';
   }
   binaryStr[8] = '\0';      // Null-terminate the string
}
```

```
int main(void)
{
    char binary[9];              // 8 bits + null character
    uint8_t number = 13;
    decimal_to_binary(number, binary);
    // Now binary[] contains "00001101"
    // You can send it via USART or display on LCD
    while(1);
}
```

AVR C Program for Binary to Decimal Conversion:

Example: Convert "1101" (binary) to 13 (decimal).

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <string.h>
// Function to convert binary string to decimal number
int binaryToDecimal(const char *binStr)
{
    int decimalValue = 0;
    int length = strlen(binStr);
    for (int i = 0; i < length; i++)
    {
        if (binStr[i] == '1')
        {
            decimalValue = (decimalValue << 1) | 1;
        }
        else if (binStr[i] == '0')
        {
            decimalValue = (decimalValue << 1);
        }
        else
        {
            return -1;   // Invalid character
        }
    }
    return decimalValue;
}
int main(void)
{
    const char binaryInput[] = "1101";     // Binary string
    int decimalResult = binaryToDecimal(binaryInput);
    // Now decimalResult contains the converted value (13)
    while (1);
}
```


AVR C Program for Decimal to Hexadecimal Conversion:

Example: Convert a number to a hex string.

Here, sprintf() is used to convert a number to a hex string.

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <stdio.h>
int main(void)
{
    char hexStr[5];                    // Enough for "0xFF" + null
    uint8_t number = 255;
    sprintf(hexStr, "%02X", number); // Converts to hex (e.g., "FF")
    // hexStr now contains "FF"
    // You can transmit this via UART or display on LCD
    while(1);
}
```

AVR C Program for Hex to Decimal Conversion:

Example: Convert "1A" (hex) to 26 (decimal).

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <string.h>
#include <ctype.h>
// Function to convert a single hex character to decimal value
int hexCharToDecimal(char c)
{
    if (c >= '0' && c <= '9') return c - '0';
    else if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    else if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    else return -1;     // Invalid character
}
// Function to convert hex string to decimal number
int hexToDecimal(const char *hexStr)
{
    int length = strlen(hexStr);
    int base = 1;
    int decimalValue = 0;
```

```
    for (int i = length - 1; i >= 0; i--)
    {
        int digit = hexCharToDecimal(hexStr[i]);
        if (digit < 0) return -1; // Error
        decimalValue += digit * base;
        base *= 16;
    }
    return decimalValue;
}
int main(void)
{
    const char hexInput[] = "1A";  // You can change this input
    int decimalResult = hexToDecimal(hexInput);
    // Now decimalResult contains the converted decimal (26)
    while (1);
}
```

2.10 AVR PROGRAMMERS

There are various AVR programmers and IDEs are available in the market. The summary of it in the tabular form is presented at the end of this section.

The Step-wise Procedure of Microchip Studio:

If Microchip Studio is used to program AVR, then below is the procedure to program AVR:

1. Connect the AVR microcontroller to a programming tool.
2. Open Microchip Studio and navigate to Tools->Device Programming dialog box.
3. Select the programming tool, device and the programming interface.
4. Read the Device ID to verify the connections between the tool and the device.
5. Select the binary to be programmed (hex/elf/bin format) and the options (Erase/Verify).
6. Select the Program command to program the device.

The Procedure to Program an AVR Device in MPLAB X IDE:

1. Make sure the application project is set as the Main project
2. After the build of the application project. Right-click on the project and choose properties.
3. In the device option make sure the right target device is chosen.
4. In the connected hardware tool option choose the programming tool that is used.
5. The next step is to choose the required XC8 compilers and click Apply then OK.
6. Finally, choose to make and program the device's main project.

Programming tools such as PICkit4, PICkit5, Atmel ICE, MPLAB Snap, ICD 4/5 are supported in MPLAB X IDE.

Programming Interfaces:

There are various programming interfaces such as ISP, UPDI ,High-voltage, JTAG and PDI used by AVR which are described below:

1. **In-System Programming (ISP):** To download binary into the flash and EEPROM memory of the AVR, AVR internal SPI (Serial Peripheral Interface) of In-system programming (ISP) is used. ISP programming requires VCC, GND, RESET and 3 signal lines for programming. No high-voltage signals are required, the AVR can be programmed at the normal operating voltage. The ISP programmer can program the internal flash, EEPROM, fuses and lock-bits. The ISP frequency (SCK) must be less than 1/4 of the target clock. Microchip tools that support ISP programming are STK600, STK500, AVRISP mkII, JTAGICE mkII, AVR Dragon, JTAGICE3 and Atmel-ICE.
2. **UPDI (Unified Program and Debug Interface):** For external programming and on-chip debugging, UPDI which is a one-wire interface is used. VTG, GND and UPDI pins are used for programming. Some of the AVR devices has separate pins for UPDI and RESET. Tools that support the UPDI interface are Atmel ICE, PIckit 4/5, STK600, Power debugger etc. For High Voltage UPDI programming, PICKit4, PICkit5, STK600 and Power Debugger can be used.
3. **High Voltage Programming (Serial/Parallel):** For High-Voltage programming, 12V programming voltage is applied to the RESET pin of the AVR device. All AVR devices can be programmed with High-Voltage programming and the target device can be programmed while it is mounted in its socket. There are two different methods used for High-Voltage programming; 8-pin parts use a serial programming interface, while other parts use a parallel programming interface. The Microchip tools that support high-voltage programming are STK600, STK500 and the low-cost AVR Dragon.
4. **JTAG programming:** AVR which have the JTAG interface (Devices with 40 pins or more) can also be programmed using JTAG programming. The AVR Tools that support JTAG programming are STK600, JTAGICE mkII, AVR Dragon, JTAGICE3 and Atmel-ICE.
5. **PDI programming:** PDI is the new two-wire exclusive debug interface for ATxmega devices. It can download code into the flash application and boot memories, EEPROM memory, fuses, lock-bits and signature information. A minimum of four wires is needed to connect the AVR debugging Tool to the target board using the PDI interface. These signals are Vcc, GND, DATA and CLK. The CLK line is driven by the debugging tool and the DATA line carries half-duplex communications between the debugging tool and the target.

Summary of AVR Programmers and IDEs:

Below table gives the popular **IDEs** for AVR Programming:

Table 2.10

IDE	Platform	Use Case	Compiler Support	Programmer Support
Microchip Studio	Windows	Full-featured AVR development	AVR-GCC	AVRISP mkII, ICE
MPLAB X	Win/macOS/ Linux	Unified AVR + PIC development	XC8	Atmel ICE, Snap, PICkit
Arduino IDE	Cross-platform	Beginner-friendly, Arduino boards	Arduino core	Bootloader, USBasp
PlatformIO	Cross-platform	Advanced dev with VS Code	AVR-GCC	USBasp, Arduino ISP
AVR-Eclipse	Cross-platform	Open-source, flexible	AVR-GCC	External programmers
Code::Blocks AVR	Cross-platform	Lightweight and configurable	AVR-GCC	External programmers

List of popular **AVR Programmers** is given in below table:

Table 2.11

Programmer Name	Interface	Features	Supports
Atmel-ICE	ISP, JTAG, PDI, UPDI, SWD	Professional-grade debugger and programmer by Microchip.	AVR and SAM microcontrollers
MPLAB Snap	UPDI, JTAG	Budget-friendly programmer/Debugger from Microchip.	AVR and PIC (via MPLAB X IDE)
USBasp	ISP	Open-source, low-cost USB programmer. Needs avrdude for uploading.	Most ATmega and ATtiny devices
USBtinyISP	ISP	Simple USB programmer, often used with Arduino IDE.	Basic ATmega and ATtiny support
Pocket AVR Programmer (SparkFun)	ISP	Compact programmer; ideal for hobbyists.	Works with avrdude, limited debugging
PICkit 4	UPDI, JTAG, ISP	Advanced Microchip programmer, supports AVR via MPLAB X	Wide AVR/PIC/SAM support
STK500/ STK600	ISP, HVPP, JTAG	Official development/programming boards (older but robust).	Broad AVR support, legacy devices

Exercise

[I] Multiple Choice Questions:

1. Which architecture does AVR microcontrollers use?
   (a) Von Neumann (b) Harvard
   (c) RISC-V (d) CISC
2. What is the word size of the ALU in ATmega32?
   (a) 4-bit (b) 8-bit
   (c) 16-bit (d) 32-bit
3. How many general-purpose registers are there in AVR?
   (a) 8 (b) 16
   (c) 32 (d) 64
4. What type of memory stores the program code in AVR?
   (a) SRAM (b) EEPROM
   (c) Flash (d) ROM
5. Which pin on ATmega32 is used to reset the microcontroller?
   (a) VCC (b) GND
   (c) RESET (d) XTAL
6. Which header file includes register definitions in AVR C?
   (a) <stdio.h> (b) <math.h>
   (c) <avr/io.h> (d) <delay.h>
7. Which data type in AVR C is typically used for 8-bit unsigned operations?
   (a) char (b) unsigned char
   (c) int (d) float
8. Which operator is used to invert bits in AVR C?
   (a) & (b) |
   (c) ^ (d) ~
9. What is the purpose of _delay_ms() in AVR C?
   (a) To initialize ports (b) To create a software delay
   (c) To execute interrupts (d) To enable serial communication
10. Which register is used to make a port pin an output in AVR C?
    (a) PORTx (b) PINx
    (c) DDRx (d) SPDR
11. Which statement assigns a value to Port B in AVR C?
    (a) `PORTB = 0xFF;` (b) `DDRB = 0x00;`
    (c) `PINB = 0xFF;` (d) `TCCR0 = 0x00;`
12. Which stage comes first in AVR development?
    (a) Simulation (b) Programming
    (c) Designing (d) Testing

13. Which AVR memory type is non-volatile and used for permanent program storage?
    (a) SRAM (b) EEPROM
    (c) Flash (d) DRAM
14. _____ is/are the official AVR programmers by Microchip.
    (a) Atmel-ICE (b) MPLAB Snap
    (c) Both (a) and (b) (d) None of these
15. _____ are the popular IDEs used for AVR programming.
    (a) Microchip Studio (b) MPLAB X
    (c) Arduino IDE (d) All of above

Answers

1. (b)	2. (b)	3. (c)	4. (c)	5. (c)	6. (c)	7. (b)	8. (d)	9. (b)	10. (c)
11. (a)	12. (c)	13. (c)	14. (c)	15. (d)					

[II] State True or False:

1. AVR microcontrollers use the Harvard architecture.
2. ATmega16 and ATmega32 have different architectures.
3. AVR microcontrollers typically have 32 general-purpose registers.
4. The program memory in AVR microcontrollers is usually EEPROM.
5. All AVR microcontrollers have exactly 64 I/O pins.
6. The PORTx register configures the direction of port pins.
7. _delay_ms() function introduces a hardware-based delay.
8. Bitwise operators are useful for setting or clearing specific bits in ports.
9. The int data type in AVR is 32-bit by default.
10. Using floating-point operations in AVR is memory-efficient.
11. Simulation is typically done after writing and compiling the code.
12. AVR microcontrollers cannot be reprogrammed once flashed.
13. Microcontrollers need a special hardware programmer to upload code.
14. IDE is used only for simulation, not for writing or compiling code.
15. Arduino IDE is the official IDE programmer by Microchip.

Answers

1. True	2. False	3. True	4. False	5. False
6. False	7. False	8. True	9. False	10. False
11. True	12. False	13. True	14. False	15. False

[III] Fill in the Blanks:

1. The ATmega16 and ATmega32 differ mainly in their _____ size.
2. The AVR architecture contains _____ general-purpose registers.
3. The memory area used to store variable data during execution is called _____ memory.

4. To create delays in AVR C programming, we use the header file _____.
5. The _____ operator is used to invert all bits in a byte.
6. Programming an AVR involves stages like writing, compiling, burning and _____.
7. To perform a logical AND operation in C, we use the operator _____.
8. The _____ memory map provides the address space layout for all memory areas.
9. The _____ performs arithmetic and logical operations inside the AVR.
10. The #include <avr/io.h> directive allows access to _____ specific register definitions.

Answers

1. memory
2. 32
3. SRAM
4. <util/delay.h>
5. ~
6. testing or debugging
7. &&
8. AVR processor
9. ALU
10. device

[IV] Short Answer Questions:

1. Which architecture is used in AVR microcontrollers?
2. List any two differences between ATmega16 and ATmega32.
3. What is the function of the ALU in an AVR microcontroller?
4. How many general-purpose registers are available in AVR architecture?
5. What is the use of the AVR processor memory map?
6. Name any two peripherals available in AVR microcontrollers.
7. What is the purpose of the DDRx register in AVR micro-controllers?
8. What type of memory is used for program storage in AVR?
9. What is the role of the `#include <avr/io.h>` directive in AVR C programming?
10. Which header file is used for delay functions in AVR?
11. Differentiate between `PORTx` and `PINx` registers.
12. Give one example of a bitwise operator used in AVR programming.
13. What is the purpose of using `while(1)` in an AVR program?
14. Name any two standard data types used in AVR programming with their typical size.
15. Write a short use case where a data transfer operation is required in embedded C.
16. How can arithmetic operations be performed in AVR C? Give an example.
17. What is code conversion in embedded programming?
18. Give an example of a decision-making statement used in AVR C programming.
19. List the main stages involved in AVR microcontroller-based project development.
20. Why is debugging important in microcontroller programming?

[V] Long Answer Questions:

1. Draw the architecture of AVR microcontrollers (ATmega16/32). Explain memory organization, CPU registers, ALU and I/O ports. 3
2. Describe the classification of the AVR family of microcontrollers. Highlight the key features that differentiate TinyAVR, MegaAVR and XmegaAVR. 1
3. Explain the AVR processor memory map. What are the different memory segments and their purposes in program execution?
4. Explain the function and importance of CPU registers in the AVR architecture. Describe the role of general-purpose registers and special function registers. 6
5. List and explain any four commonly used peripherals in AVR microcontrollers. How do they enhance the functionality of an embedded system? 9
6. Describe the basic structure of an AVR C program. Include the purpose of header files, main function and how input/output operations are handled. 10
7. Explain the use of different data types in AVR C programming with examples. Include their size, range and suitable use cases. 10
8. Discuss the role of operators in AVR C programming. Categorize them into arithmetic, logical and bitwise operators with examples. 11
9. What is the purpose of including library files in AVR programming? Name at least three essential libraries and their typical use. 12
10. Differentiate between blocking and non-blocking delay functions in AVR C. Explain how delay is implemented using `_delay_ms()` and timers.
11. Write and explain a C program for performing a simple data transfer operation using an AVR microcontroller. 14
12. Write a C program to perform arithmetic operations (addition, subtraction, multiplication, division) on two variables in AVR. 14
13. What are decision-making statements in C? Write a program to turn ON a LED when a switch is pressed using if statement. 15
14. What are AVR development boards and programmers? List a few examples of both.
15. List various types of bitwise operators along with their syntax. 13

Syllabus ...

1. **Introduction to Embedded System and Microcontroller [06 Lectures]**
   - **Introduction to Embedded Systems:** Embedded Systems: Introduction, Characteristics, Elements and Applications. Design Metrics: NRE Cost, Unit Cost, Time to Market, Safety, Maintenance, Size, Cost and Power Dissipation. Software Development Tools: Editor, Assembler, Linker, Compiler, IDE, ICE, Programmer and Simulator.
   - **Microcontroller and Architectures:** History, Introduction, Classification, Applications. Differences between Microcontroller and Microprocessor, Criteria for Choosing a Microcontroller. Architectures - Harvard and Von-Neumann Architecture, RISC vs CISC. Concept of Pipelining.
2. **Fundamentals of AVR and Its Programming in C [06 Lectures]**
   - **AVR Architecture:** Overview of AVR, Classification of AVR Family, AVR (ATmega16/32) Architecture, AVR Processor Memory Map, CPU Registers, ALU, I/O Ports, Peripherals in AVR.
   - **Programming of AVR in C:** Basic Structure, Data Types, Operators, Library Files, Delay Functions and Bitwise Operation Syntax. Simple C Programs: Data Transfer Operation, Arithmetic Operation, Decision-making and Code Conversion.
3. **AVR Peripherals Programming in C [10 Lectures]**
   - **AVR Timer Programming:** Introduction, Difference between Timer and Counter operation, Basic SFR Registers used – Timer 0, 1 & 2, C Programs for Delay Generation, Counter Programming.
   - **AVR Serial Port Programming:** Basics of Serial Communication (Serial Vs Parallel, Simplex Vs Duplex), Difference between Asynchronous and Synchronous Communication, USART Operation, SFR used, C Programs for Data Transmission and Reception.
   - **I2C and SPI:** Introduction, Specifications, Bus Signals, Master-slave Configuration, Error Handling and Addressing. SFR used in AVR, C Programs to Transfer and Receive Information.
   - **On-chip ADC:** Features, Block Diagram, Operation, SFR used, C Programs to Convert the Analog Signal to Digital.
4. **Real World Interfacing with AVR and Case Studies [08 Lectures]**
   - **I/O Device Interfacing:** LED, Push Button, Buzzer, Seven Segment Display, Thumbwheel Switch, DC and Stepper Motor, Relay Interfacing, 16*2 LCD Interfacing, DAC Interfacing (Waveform Generation using DAC).
   - **Case Studies:** Traffic Light Controller using AVR, Single Digit Event Counter using Opto-interrupter and SSD, Real Time Clock using IC DS1307 Chip, Temperature Monitoring System using LM35 Sensor, Smart Phone Controlled Devices using Bluetooth Module HC05.

■■■