🧠 1. Introduction to Microprocessor

What is a microprocessor?

A **microprocessor** is an electronic chip that acts as the brain of a computer system. It processes instructions and makes decisions based on those instructions.

🎯 Real-world analogy:

Think of a **microprocessor as a manager in an office**:

- Employees (I/O devices, memory) come to it with tasks.
- The manager reads (fetches), understands (decodes), and performs (executes) those tasks.
- All office activities (calculations, decisions) go through the manager.

💻 CS perspective:

- The microprocessor is the **central unit (CPU)** that executes low-level machine code.
- It performs:
    - Arithmetic (add, subtract)
    - Logic (AND, OR, NOT)
    - Control (IF conditions, jumps)
- These are **assembly-level operations**, below what we write in C or Python.

🔄 2. How a Microprocessor Works: Fetch-Decode-Execute Cycle

🧩 Three main steps:

1. **Fetch:** Reads the instruction from memory (e.g., ADD A, B)
2. **Decode:** Understands the instruction (e.g., what is ADD, who is A & B)
3. **Execute:** Carries out the instruction (e.g., performs A + B)

🧠 Analogy:

Imagine a chef:

- 📜 Reads a recipe (Fetch)
- 🧠 Understands what it means (Decode)
- 🍳 Cooks the dish (Execute)

👨‍💻 Relevance to CS:

- This is how all modern programs run under the hood.
- Your C/Python program is eventually compiled into **machine code**, which the microprocessor fetches & executes.

🧮 3. Components of Microprocessor

Component	Role
ALU (Arithmetic Logic Unit)	Performs math and logic
Control Unit	Directs operations like traffic police
Register Array	Temporary data storage (like variables in RAM)

📦 Analogy:

- ALU = Calculator in your brain.
- Control Unit = Brain's planner.
- Registers = Your sticky notes or scratchpad while solving a math problem.

🛠️ Example in coding:

```
int a = 2, b = 2;
int c = a + b;
```

- Registers store `a` and `b`.
- ALU performs addition.
- Control Unit tells the system to store result in `c`.

📈 4. Evolution of Microprocessors

Timeline Highlights:

Year	Processor	Bits	Speed
1971	Intel 4004	4-bit	740 kHz
1976	Intel 8085	8-bit	3 MHz
1982	Intel 80286	16-bit	6 MHz
2000	Pentium 4	32-bit	GHz range
2017	Intel i9	64-bit	4+ GHz

🎯 Why it matters:

- More **bits** → better precision and memory addressing
- Higher **clock speed** → faster execution
- More **cores** → more tasks handled at once

5. Applications of Microprocessors

Real-world devices:

- Smart TVs
- Washing machines
- Mobile phones
- Cars (engine control, ABS, GPS)

Analogy:

Just like our brain processes decisions, **microprocessors control decisions** in devices—e.g., when to stop water in a washing machine, or how much to heat in a microwave.

Microprocessors vs Microcontrollers:

Microprocessor	Microcontroller
General-purpose	Task-specific
Needs external memory	Has inbuilt memory & I/O
Used in PCs, laptops	Used in appliances, robots

6. Features of Microprocessors

Feature	Meaning
Low cost	One chip replaces many components
High speed	Executes millions of instructions/sec

Portable	Small form factor, fits anywhere
Low power	Efficient for embedded devices
Reliable	Low failure rate

💡 CS Relevance:

Microprocessors are foundational for understanding:

- Operating systems
- Embedded systems
- Real-time systems
- Assembly & machine code

☑ 7. Advantages & Disadvantages

✔ Advantages:

- High processing speed
- Reprogrammable (software-based logic)
- Used in almost every modern device

✖ Disadvantages:

- Can overheat with continuous use
- Not suitable for floating-point math (without FPU)
- Less integrated than microcontrollers

🌐 8. Real-World Applications

Domain	Examples
Home	Fridge, washing machine, AC
Medical	Monitors, insulin pumps

Transport	GPS, traffic signals, EVs
Industrial	Robotics, automation
Entertainment	Game consoles, TVs
Office	Printers, phones, security systems

🎓 Summary for a CS Student:

- Microprocessors are the **execution engines** of the digital world.
- They **fetch, decode, and execute** instructions just like interpreters.
- Understanding microprocessors gives you deep insight into **how software meets hardware**.
- As a CS engineer, this forms the **base for embedded systems**, **IoT**, **RTOS**, and even **compiler design**.

🧠 What is a Microcontroller?

🔑 Definition:

A **microcontroller (MCU)** is a compact integrated circuit (IC) designed to **control specific operations** in embedded systems. It contains:

- A processor (CPU),
- Memory (RAM & ROM),
- Input/Output (I/O) peripherals

👉 All packed into **a single chip**.

🎯 Analogy:

A **microprocessor** is like a **CEO** — great at computing, but needs others to manage memory, I/O, etc.

A **microcontroller** is like a **self-contained robot** — has a brain (CPU), memory, and can talk to sensors/actuators directly.

👨‍💻 CS Context:

- Microcontrollers are used in **embedded systems**: smartwatches, microwaves, drones, etc.
- Unlike general-purpose microprocessors, microcontrollers run a **fixed, small program** repeatedly (like blinking an LED, controlling a fan, etc.).

⚙️ Available Microcontrollers

Family	Notes
8051	Classic 8-bit MCU, popular in academia
AVR	Used in Arduino Uno (ATmega328)
PIC18	Used in this course, from Microchip
ARM (Cortex-M)	32-bit, modern, used in industry

🔧 In this course, focus is on **PIC18F microcontroller**, which is 8-bit, and widely used in embedded projects.

🧩 Types of Microcontrollers

Microcontrollers can be categorized by:

1️⃣ Bit-width (Data Size Handled)

Type	Example	Application
8-bit	PIC18, 8051	Simple tasks like blinking LEDs, keypad input
16-bit	MSP430	Moderate control: ECG, digital thermometers
32-bit	ARM Cortex	Complex systems: smart TVs, robots, industrial control

🎯 **Bits** indicate how much data can be processed in one instruction. 8-bit = 1 byte at a time.

2️⃣ Memory Type

Type	Description
Embedded Memory (modern)	All memory (RAM, ROM), I/O are built-in
External Memory (older)	Needs separate memory chips

🧠 Like having all organs in one body (embedded) vs attaching organs separately (external).

👨‍💻 As CS students, you'd appreciate that embedded memory makes systems:

- Compact
- Fast
- Lower power

3️⃣ Instruction Set Architecture

Type	Full Form	Description
RISC	Reduced Instruction Set Computer	🧠 Simple, fast, one instruction per cycle
CISC	Complex Instruction Set Computer	📚 Rich set of instructions, but slower

🔁 **PIC18 uses RISC**, meaning it executes instructions quickly, often 1 per clock cycle. Great for time-sensitive tasks.

🎯 Analogy for RISC vs CISC:

Imagine building a chair:

- 🪛 **RISC** = lots of simple tools, fast use (screwdriver, drill)
- 🛠️ **CISC** = fewer tools but they do complex things (a machine that does screwing, cutting, painting — but slower to configure)

⚙️ Microcontroller vs Microprocessor

Feature	Microprocessor	Microcontroller
Focus	High computing tasks	Control + sensing tasks
RAM/ROM	External	Embedded
Application	PCs, Laptops	Washing machines, drones
Size & Cost	Larger, costlier	Compact, cheaper
Example	Intel i5/i7/i9	PIC, 8051, Arduino, STM32

🧩 Elements of a Microcontroller

Here's what's inside a microcontroller like **PIC18F**:

Main Components:

- **CPU** – brain of the MCU
- **RAM** – volatile memory for temporary data
- **ROM/Flash** – program memory (your code lives here)
- **Timers** – keep time, generate delays (e.g., blink LED every 500ms)
- **Interrupts** – respond to events (e.g., button pressed)
- **ADC (Analog to Digital Converter)** – reads analog signals (e.g., temperature sensors)
- **I/O Ports** – talk to outside world (LEDs, buzzers, motors)
- **Serial Communication (UART, SPI, I2C)** – for communication with sensors/modules

All of this is **in one chip**, no need to connect separate components like in a microprocessor.

🎯 Analogy:

Microcontroller is like a **Swiss Army Knife** – a single compact tool with:

- Knife (CPU)
- Screwdriver (Timers)
- Scissors (ADC)
- Tweezers (I/O)
- And more!

🔗 Microcontroller Connectivity

🔌 Input Devices:

- Push buttons, keypads
- Sensors (temperature, fire, motion)
- Switches

📢 Output Devices:

- LEDs
- Buzzers
- Displays (LCD, 7-seg, OLED)
- Relays (to control motors, lights)

🧠 CS Tip:

- Inputs are **digital (on/off)** or **analog (variable voltage)**.
- Output devices are controlled via **digital signals** or **PWM (Pulse Width Modulation)**.

🌐 Applications of Microcontrollers

📱 Household:

- Washing machine (controls water level, spin cycle)
- Microwave (timing, heating)
- Smart lighting
- Fire alarms

🏭 Industrial:

- Robots
- Bar code readers
- Smart meters
- CNC machines

🚗 Automotive:

- Airbags
- Cruise control
- Parking sensors
- GPS trackers

🏥 Medical:

- Thermometers
- CT scanners
- Ventilators
- ECG monitors

Microcontrollers are **everywhere** — low power, cheap, reliable, and smart.

🎓 Summary for a CS Student

Microprocessor	Microcontroller
Think “brain” of a **PC**	Think “brain” of an **appliance**
General-purpose computing	Specific task-oriented
Needs RAM, ROM, I/O externally	All included on one chip
High power, high cost	Low power, low cost
OS-dependent	Often no OS (bare-metal)
Example: Intel i7	Example: PIC18F, Arduino, STM32

☑ You Should Now Understand:

- What a microcontroller is and why it matters
- Key differences from microprocessors
- Types based on **bit size**, **memory**, and **ISA**
- Components and real-life **applications**
- Why **RISC-based** MCUs are fast and efficient

🧠 Lecture Notes: Microcontrollers (with Real-World Analogies)

1. What is a Microcontroller?

- **Definition:** A microcontroller is a small, integrated chip used to control a specific operation in an embedded system.
- **Analogy:** Think of a microcontroller like the **brain of a washing machine**. It controls when to take in water, when to spin, when to heat — all based on internal instructions.
- **Key Features:**
  - Small in size
  - Contains CPU, RAM, ROM, I/O ports, timers — all on one chip
  - Handles both computation and control

2. Microcontroller vs. Microprocessor

Feature	Microcontroller	Microprocessor
Usage	Embedded systems (e.g., appliances)	Computers and laptops
Components on chip	CPU + RAM + ROM + I/O + Timers	Only CPU
Real-world example	Fire alarm system	Gaming laptop processor (e.g., Intel i7)

3. Popular Microcontrollers:

- **AVR, ATmega, Arduino, 8051, PIC18F4550** (used in syllabus)
- PIC18 is an **8-bit** microcontroller popular in industry and academics.

4. Types of Microcontrollers:

A. Based on Data Width (Bits):

- **8-bit**: Handles 8-bit data at a time. (e.g., PIC18F4550)
    - 🧠 Example: Like a student who can carry 8 notebooks at a time.
- **16-bit**: Can carry 2x more data with more precision.
- **32-bit**: Used in advanced applications (Smart TVs, Washing Machines)

B. Based on Memory:

- **Embedded Memory** (modern, all-in-one): Memory + I/O built-in
- **External Memory** (older design): Requires external RAM/ROM

C. Based on Instruction Set:

- **RISC (Reduced Instruction Set Computer)**: Executes simple, fast instructions (used in PIC18)
- **CISC (Complex Instruction Set Computer)**: Executes complex instructions but slower.

5. Components of a Microcontroller:

Component	Description
CPU (Processor)	Executes instructions
RAM	Temporary data storage
ROM (Flash)	Stores program permanently
I/O Ports	Connects sensors, LEDs, etc.
Timers/Counters	Handle time-based events
ADC	Converts analog signals to digital
Watchdog Timer	Prevents system from hanging

💡 **Analogy**: Think of a microcontroller like a **mini desktop** inside a washing machine:

- CPU = brain
- RAM/ROM = short- and long-term memory
- I/O = mouse/keyboard (inputs), screen (output)
- Timer = alarm clock

6. Connectivity:

- **Input Devices:** Buttons, switches, sensors
- **Output Devices:** LEDs, buzzers, displays

7. Applications of Microcontrollers:

- Smart Lights
- Fire Alarms
- Microwave Oven
- Industrial Robots
- Barcode Scanners

- Medical Devices (e.g., Ventilator, X-ray)
- Traffic Light Systems
- GPS Trackers

🔧 Practical: Embedded C – Addition of Two Numbers (Using MPLAB IDE & PIC18F4550)

Software/Tools Used:

- **MPLAB IDE v8.89** – Development Environment
- **MPLAB C18 Compiler** – Compiler for PIC18
- **Target Device:** PIC18F4550 Microcontroller

💻 Step-by-Step Procedure:

🛠 Step 1: Setup Project

- Create a folder, e.g., `E:\AdditionDemo`
- Open MPLAB → Project → Project Wizard → Select PIC18F4550
- Choose MPLAB C18 Compiler → Save project as `Addition1.mcp` in the folder

🛠 Step 2: Write Code (add1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <p18f4550.h>  // Specific to our microcontroller

void main() {
    int sum = 0;
    sum = 0x0A + 0x02;  // Hexadecimal: 10 + 2 = 12 (0x0C)

    TRISD = 0x00;       // Set PORTD as Output
    PORTD = sum;        // Output result to PORTD
```

```
}
```

✅ `TRISD = 0x00` → Configures Port D as output

✅ `PORTD = sum` → Displays result on output pin

🧪 Step 3: Build & Simulate

- Save as `add1.c`
- Add this file to your MPLAB project under **Source Files**
- Build project → Make sure you see `Build Succeeded`

Run Simulation:

- Go to `Debugger → Select Tool → MPLAB SIM`
- View → Special Function Registers
- Press `F9` to run
- Press `F5` to stop
- Check `PORTD` value (should be `0x0C` = 12)

🧪 Test: Change Input

Change:

```
sum = 0x03 + 0x02;  // 3 + 2 = 5 → 0x05
```

Rebuild → Simulate → Check `PORTD` → Should show `0x05`

☑️ Summary of Practical Concepts:

Concept	Explanation

TRISx Register	Direction: `0 = output, 1 = input`
PORTx Register	Value sent to output pin
Hexadecimal format	`0x0A = 10, 0x0C = 12`, etc.
Header Files	`p18f4550.h` is MCU-specific
Embedded C vs C	Includes hardware control (registers, ports)

☑ Summary of What You've Learned So Far:

🧠 *1. Microcontroller Basics (PIC18F4550)*

- A microcontroller is the backbone of an embedded system.
- PIC18F4550 is an 8-bit microcontroller from Microchip, with inbuilt memory, I/O ports, timers, and more.

🔧 *2. MPLAB IDE & C18 Compiler*

- **MPLAB IDE v8.89** is used to write, compile, and simulate Embedded C code.
- **C18 Compiler** converts C code into machine code for the PIC18F4550.

🧪 Practical 1: Addition of Two Hexadecimal Numbers

- Wrote a C program to add two hex numbers (`0x0A + 0x02`).
- Stored the result in PORTD.
- Used `TRISD = 0x00` to configure PORTD as output.
- Used MPLAB SIM to simulate and observe output in **Special Function Registers (SFRs)**.

Output:

- Example: `0x0A + 0x02 = 0x0C` → Result displayed at PORTD.

Practical 2: Addition of Array of N Numbers

- Declared an array: `int number[10] = {1,2,...,10};`
- Used a `for` loop to sum all values → `sum = 55`
- Output (`sum`) stored to `PORTD`.
- Microcontroller showed `0x37` (hex for 55) at `PORTD`.

Key Concepts You Used:

Concept	Explanation
`TRISx` Register	Sets direction of port pins (`0 = Output`, `1 = Input`)
`PORTx` Register	Holds output values or reads input
Hexadecimal Format	Embedded systems use hex for compact memory representation
`#include <p18f4550.h>`	Imports all microcontroller-specific registers and definitions
Simulation in MPLAB	Allows you to run the code and check output without physical hardware

1. Serial Communication Protocols

Analogy: Imagine you're sending messages using a walkie-talkie – you speak one word at a time. That's how serial communication works: **one bit at a time**.

USART (Universal Synchronous Asynchronous Receiver Transmitter)

- **How it works**: Sends and receives one bit at a time over a **single wire** (TX/RX).
- **Example**: Like plugging a **printer or keyboard** into your computer using a USB cable – one wire handles both send and receive.
- **TX (Transmit)** pin sends data, **RX (Receive)** pin receives data.
- **Real-world devices**: Bluetooth, GSM, RFID, GPS connected to the microcontroller.

SPI (Serial Peripheral Interface)

- **Pins**: Used on pin numbers 18, 23, and 24.
- **How it works**: Communicates faster than USART using multiple wires.
- **Master–Slave architecture**: The microcontroller (master) controls other devices (slaves).
- **Example**: Like a teacher (master) talking to multiple students (slaves) using separate channels at the same time.
- **Real-world use**: Communicates with SD cards, sensors, and shift registers.

I^2C (Inter-Integrated Circuit)

- **Pins**: Also uses pin numbers 18 and 23.
- **How it works**: Uses only **2 wires** – one for data (SDA) and one for clock (SCL).
- **Master–Slave system**: Microcontroller as master, EEPROM/ADC as slave.
- **Example**: Like a computer (master) connected to printer, keyboard, mouse (slaves).
- **Used for**: Low-speed communication (EEPROM, temperature sensors).

☑ 2. Interrupts

Analogy: Like a fire alarm in a building – it **interrupts your routine** to tell you something important.

- **Purpose**: Stops the normal flow of code and **executes an urgent task**, then goes back to the original job.
- **Example**: When you press a button during execution, the microcontroller stops the task, handles the button press (interrupt), then resumes the task.
- **Used in**: Emergency stop buttons, alarms, sensors, etc.

☑ 3. Input/Output Ports

Analogy: Think of I/O ports as doors on a house — they can be used to either **take things in (input)** or **send things out (output)**.

- **Ports**: A, B, C, D, E
  - Port A = 7 pins
  - Ports B, C, D = 8 pins
  - Port E = 3 pins
- **Total usable I/O pins**: 34 out of 40
- **Registers involved**:
  - `TRIS` – Direction register (1 = Input, 0 = Output)
  - `PORT` – Reads the current value on the pin
  - `LAT` – Used to write a value to a pin

☑ 4. CCP Module (Capture Compare PWM)

CCP = Capture, Compare, PWM (Pulse Width Modulation)

- **Capture**: Records the time when a signal arrives (like a stopwatch).
- **Compare**: Compares a value with a timer to do something (like turn on a fan at 30°C).
- **PWM**: Generates a repeating signal with variable duty cycle (used for motor speed, LED brightness).
- **Example**: ECG machines – show the pulse rate using waveforms.

☑ 5. Timers

Analogy: Like a kitchen timer that helps you know when your food is ready.

- **Timers**: 0, 1, 2, 3
  - Timer 0: 8/16-bit
  - Timer 1: 16-bit
  - Timer 2: 8-bit

    - Timer 3: 16-bit
- **Prescaler**: Like fan speed levels (0–5), used to adjust timer speed.
- **Used for**: Delay generation, event counting, real-time clocks.

☑ 6. ADC & DAC (Analog to Digital / Digital to Analog Converter)

- **ADC**: Converts **real-world analog signals** (like temperature, voltage) to digital (0s and 1s).
    - Example: Thermometer connected to microcontroller → gives digital temperature.
- **DAC**: Converts **digital data into analog signals** (like sound or voltage).
    - Example: Microcontroller sending audio to speakers.
- **Voltage Range**: Typically 0V–5V
- **Example**: Telephone:
    - You speak (analog) → converted to digital → sent → converted back to analog on other side.

☑ Summary Table

Component	Function	Example Use
USART	Serial comm. via single wire (TX/RX)	GPS, GSM, Bluetooth
SPI	Fast serial comm. using multiple wires	SD card, sensors
I2C	Two-wire comm. with multiple slave devices	EEPROM, ADC
Interrupts	Temporarily stop main task to handle events	Button press, alarms
I/O Ports	Interface to external components (input/output)	LED, switches
CCP Module	Signal capturing, comparing, PWM generation	ECG, motor control
Timers	Count time or events	Delay, real-time clock
ADC/DAC	Convert analog to digital & vice versa	Thermometers, Audio output

Introduction to PIC18F4550 Architecture (Part 1)

Types of Microcontrollers

Microcontrollers can be categorized based on:

Criteria	Types / Example
Bits	4-bit, 8-bit, 16-bit, 32-bit
Memory	Embedded memory / External memory
Instruction Set	RISC (Reduced) / CISC (Complex)
Memory Architecture	Harvard / Von Neumann

- **Our focus**: PIC18F4550 – an **8-bit** microcontroller with **embedded memory**, **RISC** instruction set, and **Harvard architecture**.

About PIC Microcontroller

- **PIC = Peripheral Interface Controller**
- Developed by **Microchip Technology**
- **Advantages over 8085/8086**: Faster, easier for embedded C or Assembly
- Components:
  - RAM, ROM, CPU, ADC, DAC
  - Timers/Counters
  - Communication protocols: UART, SPI, I2C

PIC Families:

Family	Bit Width
PIC10, PIC12, PIC16, PIC18	8-bit
PIC24, dsPIC	16-bit
PIC32	32-bit

Features of PIC18F4550

Feature	Details
Bit Size	8-bit
Program Memory	32 KB Flash
Data Memory (RAM)	2 KB (General), 256 Bytes EEPROM
Operating Voltage	2.5V – 5.5V
Pin Count	40/44 pins
Timers	4 (Timer0–Timer3)
Interrupt Sources	18
ADC Resolution	10-bit
Max CPU Speed	40 MHz, 10 MIPS
Address Bus	Program: 21-bit, Data: 12-bit
Data Bus	Program: 16-bit, Data: 8-bit

Harvard Architecture in PIC

- **Separate buses** for program and data memory
- Enables **parallel access** → faster execution

Bus Type	Program Memory	Data Memory
Address	21-bit	12-bit
Data	16-bit	8-bit

Types of Buses

Bus Name	Direction	Purpose
Address Bus	Unidirectional	Carries memory locations (e.g., 1010H)
Data Bus	Bidirectional	Carries data (binary: 0101, etc.)
Control Bus	Control signal	Manages memory/IO read/write operations

Main Components in Architecture

CPU:

- Contains:
  - **ALU**: Arithmetic & Logic Unit
  - **Memory**: Stores & processes instructions
  - **Control Unit**: Manages all device interactions

Memory Organization Overview

1. **Program Memory**:
   a. Stores: `.asm`, `.c` files, library functions
   b. **21-bit address bus** = 2 MB access space
   c. **Program Counter (PC)**: Executes instructions sequentially
   d. **Stack Memory**: 31 levels; handles function calls, returns, nesting
2. **Data Memory**:
   a. **Size**: 4 KB RAM
   b. Divided into:
      i. **GPR (General Purpose Register)**:
         1. For temporary data (e.g., C = A + B)
         2. 15 banks × 256 bytes = 3840 bytes
      ii. **SFR (Special Function Register)**:
         1. Controls I/O, status, direction (TRIS), ports
         2. Stores hardware control settings (non-user-changeable)
3. **EEPROM (Data Memory)**:
   a. Electrically Erasable Programmable ROM
   b. Stores user-defined, re-writable data (non-volatile)
   c. Not directly mapped to RAM but accessed via special instructions

Important Memory Concepts

- **TRIS Register** (in SFR):
  - Controls pin direction (input/output)
  - TRIS = 1 → Input, TRIS = 0 → Output
- **STACK**:

- Used for subroutine calls/returns
- Behaves as LIFO (Last In First Out)

🔄 Summary:

- **Program Memory** → Stores code
- **Program Counter** → Tracks instruction
- **Stack** → Manages function calls
- **Data Memory (RAM)** → GPR (temporary) + SFR (control)
- **EEPROM** → Rewritable non-volatile storage

If you'd like, I can now prepare **Part 2 notes** (serial communication, interrupts, ports, CCP module, timers, ADC/DAC). Want to continue?

Here are clear and concise notes from **Part 2: Registers and Addressing Modes in PIC18F4550**, explained with real-world analogies where possible:

🧠 Registers and Addressing Modes in PIC18F4550

🧾 Registers in PIC18F4550

Registers are small memory units **inside the microcontroller**, used to:

- Store **temporary values**
- Control **hardware operations**
- Act as **pointers or accumulators**

🔧 Types of Registers

🔗 Register	📝 Purpose
SFR (Special Function Register)	Stores status/control info of hardware (like PORTC, TRIS, etc.)
GPR (General Purpose Register)	Scratchpad memory for temporary data like `A + B`, `C = D * E`
WREG (Working Register)	Acts as an **accumulator** for ALU – holds data for arithmetic/logic
STATUS Register	Stores **flags** that tell the result of last operation (like a scoreboard for ALU)
BSR (Bank Select Register)	Selects **RAM bank** – allows access to large memory
FSR (File Select Register)	Points to **file registers** (used in indirect addressing)
PC (Program Counter)	Points to **next instruction** in program memory
SP (Stack Pointer)	Used during **function calls/returns**, manages stack memory

🔋 Detailed: STATUS Register (Flag Register)

Think of it like a **results dashboard** showing what happened after an operation:

Flag	Name	When it's set (1)
C	Carry	If addition caused a carry / subtraction caused a borrow
DC	Digit Carry	Carry from lower nibble (BCD ops)
Z	Zero	If result = 0
OV	Overflow	Result exceeded bit limit (e.g. 8-bit + 8-bit > 8-bit)
N	Negative	If result is negative (MSB = 1)

📬 Addressing Modes in PIC18F4550

Addressing modes define *how the operand (data) is accessed* in instructions.

🎯 1. Immediate Addressing Mode

- The **value is directly given** in the instruction
- Example:

`MOVLW 0x35` → Move 0x35 directly into WREG

`MOVLW 01100010B` → Move binary data into WREG

✅ **Advantage**: Easy to read and quick

⚠️ **Limitation**: Small data only

🎯 2. Direct Addressing Mode (aka Absolute)

- Access a memory **location by its address**
- Example:

`ADDWF 0x30, W` → Add contents at address 0x30 to WREG

`MOVWF PORTC` → Move WREG value to PORTC register

✅ **Advantage**: Simple and efficient

⚠️ **Limitation**: Limited to fixed memory locations

🧠 **Analogy**: Like accessing a room using its exact room number

🔁 3. Register Indirect Addressing Mode

- The **address of data is stored in a register**, like a pointer
- Example:
  - `FSR = 0x20` → File Select Register points to 0x20
  - `MOVWF INDF` → Move WREG content to memory pointed by FSR

✅ **Advantage**: Reusable for loops and dynamic access

⚠️ **Disadvantage**: Needs more memory lookups

🧠 **Analogy**: Like using a key (FSR) to open a locker (actual address)

🧱 4. Indexed Addressing Mode

- Access data like in **arrays**
- Base address + offset = actual location
- Example:
  - `FSR = BASE_ADDR`, `MOVF POSTINC` → Access [base], then increment pointer

✅ **Advantage**: Best for **arrays and loops**

⚠️ **Disadvantage**: Slightly complex logic

🧠 **Analogy**: Like accessing items in an array: `arr[0], arr[1], ...`

📜 Summary: Comparison Table

Mode	Data Source	Example	Use Case
Immediate	In the instruction	`MOVLW 0x25`	Constants
Direct	Memory location	`MOVWF 0x30`	SFR/GPR access
Indirect	Address via FSR	`MOVWF INDF`	Dynamic memory/pointers
Indexed	Address + Offset	`MOVF POSTINC`	Arrays, loops, structured data

💡 Real-World Analogy (All Modes):

Mode	Analogy
Immediate	Telling someone a value directly: “Take 5 apples”
Direct	Giving an exact location: “Pick the item from Shelf #30”
Indirect	Giving a key to a box: “Use key A to open the drawer and find the item”
Indexed	Going through an array: “Check the 3rd student in the attendance list”

⚙️ Power Management & Reset Concepts – PIC18F4550

Topics Covered:

- Watchdog Timer (WDT)
- Power-down Modes (Run, Idle, Sleep)
- Brown-out Reset (BOR)
- Power-on Reset (POR)

⏰ 1. Watchdog Timer (WDT)

🔍 What is it?

A **hardware timer** that resets the microcontroller if the system stops working or "hangs".

🧠 Analogy:

Think of it like a **radar system** in the military. If the radar fails even for a few seconds, it may miss enemy aircraft. So a **watchdog** keeps checking if the radar is working and **resets it** if needed.

📌 How it works:

- WDT runs on its **own internal clock** (1 MHz).
- If the program fails to "reset" (or **kick**) the WDT within a time window, it **assumes a fault** and resets the MCU.
- Helps recover from **software crashes or infinite loops**.

🔧 Important Bits:

Bit	Meaning
`WDTEN = 0`	Watchdog Timer is **enabled**
`WDTEN = 1`	Watchdog Timer is **disabled**

SWDTEN	Software-controlled WDT enable

WDT in PIC18F4550:

- Pin 14 is related to the internal WDT circuit.
- Automatically resets CPU on failure.
- Used in **mission-critical applications** (e.g., aerospace, medical, military).

2. Power-Down Modes (Power Management)

Used to **conserve energy** by controlling the system's activity levels.

Three Modes:

Mode	CPU Status	Peripheral Status	Analogy
Run	Active	Active	Normal working state
Idle	OFF (halted)	ON	Laptop screen off but music plays
Sleep	OFF	OFF	Entire system is in sleep mode

Real-life Examples:

- **Run**: You are using a computer and watching videos.
- **Idle**: Laptop screen turns off after inactivity, but downloads continue.
- **Sleep**: Full system sleep – wakes up on button press.

3. Brown-out Reset (BOR)

What is it?

A feature that **resets the microcontroller when voltage drops** below a safe threshold.

🧠 Analogy:

At home, if voltage drops too low (lights dim), you switch off the fridge or TV to avoid damage. BOR does the same for the microcontroller.

📈 Voltage Levels:

Label	Meaning
V1	Normal Voltage (e.g. +5V)
V2	Safe but low (+3V)
V3	Danger Zone (< +3V)

If voltage < V3, **BOR triggers reset** to avoid malfunction.

🧩 BOR Types:

Type	Function
Hardware BOR	Waits until voltage is safe again before resuming
Software BOR	Monitors voltage and saves vital data if needed

🔁 4. Power-on Reset (POR)

🔍 What is it?

Automatic reset when microcontroller is **first powered on**.

🧠 Analogy:

Like pressing the **restart** or **reset** button on your laptop or mobile phone.

📌 MCLR Pin (Master Clear):

- **MCLR = 0** → System is **reset**
- **MCLR = 1** → System **starts/executes normally**

- Found on **Pin 1** of PIC18F4550

🔧 Working:

A **circuit with diode, capacitor, and resistors** is used to manage this power-on reset.

It ensures stable operation before letting the CPU start.

🔁 Summary Table: Reset and Power Control

Feature	Purpose	Trigger Condition	Real-world Analogy
WDT	Reset if system crashes	No response within time window	Watchdog barking to wake you up
Run Mode	Full operation	Normal working	Laptop ON
Idle Mode	CPU off, peripherals on	Sleep-like state	Laptop in hibernate
Sleep Mode	Full low power	User-controlled or auto	Laptop sleep mode
BOR	Reset on low voltage	Voltage < threshold	Dim lights = shut down appliances
POR	Reset on startup	Power ON or reset button pressed	Turning on a PC

☑️ Overview of PIC18F4550 Pin Diagram

- The **PIC18F4550** microcontroller has **40 pins**.
- Out of these, **36 pins** are used for **I/O (Input/Output)** functions.
- Remaining pins are used for **power supply and control**.

Power and Control Pins

Pin No.	Name	Function
1	MCLR	Master Clear Reset – Resets the microcontroller (like rebooting your PC).
11, 32	VDD	+5V Power Supply (input power like your laptop charger).
12, 31	VSS	Ground (0V) – the return path for current, like the neutral wire.

Understanding TRIS Registers

- Every I/O pin can work as **input or output**.
- **TRISx Register** decides the direction:
  - `TRISx = 1` → **Input**
  - `TRISx = 0` → **Output**

(Just like a road sign that says "One Way IN" or "One Way OUT")

Port A – 7 Pins (RA0 to RA6)

Features:

Each pin is **bi-directional** and can have **alternate functions** like analog inputs, timers, voltage references, etc.

Pin	Function(s)
RA0–RA4	Analog Input (AN0–AN4) – Used to read analog signals like temperature or light sensors.
RA2	Vref+ – Positive Voltage Reference input.
RA3	Vref− – Negative Voltage Reference input.
RA4	Timer0 Clock Input (T0CKI), Schmitt Trigger Input, Open Drain Output
RA5	SS (Slave Select), LVDIN (Low Voltage Detection Input)

RA6	Oscillator Input (helps in generating internal clock signals like a heart rhythm)

🔄 Real-Life Analogies for Port A Functions:

- **Vref+ and Vref−**: Like setting **maximum and minimum brightness levels** for a screen.
- **Schmitt Trigger**: Converts noisy analog signal into clean digital ON/OFF (like a sound noise filter).
- **Open Drain**: Like a tap draining into a bucket only when needed.
- **Oscillator**: Think of this like a metronome providing timing to music players.

🧠 Port B – 8 Pins (RB0 to RB7)

➕ Features:

- All 8 pins are **bi-directional** with many special features including interrupts and programming.

Pin	Function(s)
RB0 – RB2	INT0, INT1, INT2 → External Interrupts – When something urgent happens (e.g., emergency button), it stops current task to address it.
RB3	CCP1 – Capture Compare PWM (used in motor control, measuring time, or generating pulses).
RB4 – RB7	Interrupt-on-change – Detect any input state changes like a door sensor that alerts when opened.
RB6	PGC – Programming Clock – Provides clock for in-circuit programming.
RB7	PGD – Programming Data – Transfers program data during flashing.
RB5	PGM – Low Voltage Programming Enable – Checks if system has safe voltage to program.

Real-Life Analogies for Port B Functions:

- **Interrupts (INT0/1/2)**: Like a fire alarm interrupting class.
- **CCP (PWM)**: Like controlling fan speed by pulsing power.
- **PGC/PGD**: Like USB communication between PC and a phone.
- **PGM**: Like a safety lock that allows firmware update only at correct voltage.

Summary Table of Port A and B

Port	Pin Count	Direction	Special Features
A	7	I/O	Analog input, voltage reference, timer, oscillator
B	8	I/O	Interrupts, CCP, programming signals

Key Takeaways:

- **TRIS Registers** control whether a pin is input or output.
- Some I/O pins are **multiplexed**: they can serve multiple purposes.
- Ports A and B are critical for **analog-digital interfacing**, **power management**, and **interrupt handling**.
- Port A → More analog/power-related features.
- Port B → More digital/interrupt and programming-related features.

Overview: Ports in PIC18F4550 (Recap)

Port	Pins	TRIS Register	Nature
A	7	TRISA	Bi-directional
B	8	TRISB	Bi-directional
C	**8**	**TRISC**	**Bi-directional**

D	**8**	**TRISD**	**Bi-directional**
E	**3**	**TRISE**	**Bi-directional**

PORT C (Pins RC0 to RC7)

General Info:

- **8-bit** wide port
- Controlled using **TRISC** register
- Bi-directional: `TRISC = 0` → Output, `TRISC = 1` → Input

Pin Functions:

Pin	Alternate Functions	Purpose / Real-world Analogy
RC0	T1OSO, T1CKI	Timer1 oscillator and clock input – like stopwatch trigger
RC1	T1OSI, CCP2	Another Timer1 oscillator and Capture/Compare/PWM
RC2	CCP1	Capture/Compare/PWM – useful for motor control or waveform generation
RC3	SCK (SPI Clock), SCL (I^2C Clock)	Serial communication clock – like a metronome that syncs master/slave data
RC4	SDI (SPI Data In), SDA (I^2C Data Line)	Receiving data from other devices
RC5	SDO (SPI Data Out)	Sending data to SPI devices
RC6	TX (USART Transmit)	Serial data sending (e.g., from a PC to microcontroller)

RC7	RX (USART Receive)	Serial data receiving (e.g., microcontroller receiving command from PC)

Protocols Used:

- **USART** (Universal Synchronous/Asynchronous Receiver/Transmitter)
- **SPI** (Serial Peripheral Interface)
- **I^2C** (Inter-Integrated Circuit)

Analogy:

Imagine Port C like a **Post Office**:

- TX/RX = Sending/Receiving letters.
- SCK/SCL = Clock telling postman when to go.
- SDI/SDO = Reading/writing to/from customer inboxes.

PORT D (Pins RD0 to RD7)

General Info:

- **8-bit wide**
- Bi-directional: Controlled by **TRISD**
- Provides **PSP (Parallel Slave Port)** functionality.

PSP = Parallel Slave Port:

Used for parallel data transfer like how old printers used **parallel cables**.

Feature	Function
PSPMODE	Enables/disables PSP features
IBOV	Input Buffer Overflow – Write attempted before reading old data
OBF	Output Buffer Full – Previous output not yet cleared

IBF	Input Buffer Full – Input waiting to be read by CPU

🧠 Analogy:

- Like working on **multiple Word documents**.
- If you try typing (writing) in one before the last is saved (read), you get **overflow**.
- Buffer = A memory desk. You can write or read only when it's clear.

🧩 PORT E (Pins RE0 to RE2)

➕ General Info:

- **3-bit wide**
- Bi-directional: Controlled by **TRISE**
- Also supports **Analog** and **PSP** features.

Pin	Function	Purpose
RE0	AN5, RD (Read Enable)	Used for analog input or reading data from slave
RE1	AN6, WR (Write Enable)	Used for analog input or writing data to slave
RE2	AN7, CS (Chip Select)	Used for analog input or selecting the slave device

❗ Important Note:

- Cannot read and write at the **same time** using the same port. Just like you can't **edit and print** a document simultaneously without saving.

🔁 Summary of Special Functions Across Ports

Port	Features
C	SPI, I2C, USART, CCP, Timers
D	PSP mode for high-speed data exchange

E	Analog Inputs + Read/Write Control for PSP

🎯 Final Key Takeaways

- **TRISx Registers** define input/output direction.
- Ports **C, D, and E** handle **serial**, **parallel**, and **analog** communications.
- **Multiplexing**: Many pins perform **multiple functions**, depending on the application.
- Used in **real-time systems**, **data acquisition**, **motor control**, **communication**.

☑ Practical 3 – Menu-Driven Embedded C Program for 8-bit Multiplication and Division

(For PIC18F4550 using MPLAB IDE and C18 Compiler)

🛠 Objective:

Write and execute an Embedded C program for **8-bit multiplication and division** using **PIC18F4550** microcontroller.

🧱 Steps to Execute in MPLAB IDE:

◇ 1. Create Project Folder

- Example: `MultiplicationAndDivision` on Desktop
- All files (`.c`, `.mcp`, `.hex`) will be stored here.

◇ 2. Open MPLAB IDE

- Click on **Project → Project Wizard**
- Select **Device**: `PIC18F4550`
- Select **Tool Suite**: `Microchip C18 Toolsuite`
- Browse and select your folder.
- Name project file: `multiplication_division.mcp`

◇ 3. Create & Save `.c` Source File

- Click `File → New`
- Paste/write the Embedded C program
- Save as: `multiplication_division1.c` inside the same folder

◇ 4. Add Source File to Project

- Right-click `Source Files → Add Files`
- Select your `.c` file

◇ 5. Build and Debug

- Click **Build All** (Hammer icon) → Should display: `BUILD SUCCEEDED`
- Click **Debugger → Select Tool → MPLAB SIM**

- Click `View → Special Function Registers` (to see PORTC, PORTD outputs)

💻 Embedded C Code Breakdown

```
#include <p18f4550.h>
#include <stdio.h>
#include <stdlib.h>

void main() {
    int mul = 0;
    int div = 0;

    // Hexadecimal 8-bit inputs
    mul = 0x04 * 0x01;  // 4 × 1 = 4
    div = 0x06 / 0x02;  // 6 ÷ 2 = 3

    // Output to PORTD for multiplication result
    TRISD = 0x00;        // Set PORTD as output
    PORTD = mul;

    // Output to PORTC for division result
    TRISC = 0x00;        // Set PORTC as output
    PORTC = div;

    while(1); // Infinite loop to keep output visible
}
```

📊 Explanation:

Part	Description
`#include <p18f4550.h>`	Contains all registers and port definitions for PIC18F4550

`mul = 0x04 * 0x01;`	4 in hex × 1 = 4
`div = 0x06 / 0x02;`	6 in hex ÷ 2 = 3
`TRISD = 0;`	Set PORTD as **output** (to display multiplication result)
`PORTD = mul;`	Display multiplication output on PORTD
`TRISC = 0;`	Set PORTC as **output** (to display division result)
`PORTC = div;`	Display division output on PORTC
`while(1);`	Keep the program running (do nothing else)

🔍 Observation Using SFR (Special Function Registers):

Register	Expected Output
PORTD	`0x04` (Multiplication = 4)
PORTC	`0x03` (Division = 3)

📌 Key Concepts:

☑ TRISx Registers:

- **TRISD = 0** → PORTD as **output**
- **TRISC = 0** → PORTC as **output**
- TRIS stands for **TRI-State Control Register**

☑ PORTx Registers:

- Used to **display results** or **send data** out of microcontroller.

💡 Real-World Analogy:

- Think of TRIS register as a **light switch**:
  - `0` = output mode (light ON → data flows out)

    - 1 = input mode (light OFF → data received in)
- PORTC and PORTD are **LED displays** showing results of math calculations.

🧪 Experiment Suggestions:

- Try changing values in:

```
mul = 0x05 * 0x03;
div = 0x08 / 0x02;
```

- Observe changes in PORTC and PORTD.

☑ Practical 4 – Embedded C Program to Blink an LED using PIC18F4580 and PROTEUS

🎯 Objective:

Write and execute an **Embedded C program** to interface **PIC18F4580 microcontroller** with an **LED** and **blink it** with a specified delay using:

- **MPLAB IDE**
- **C18 Compiler**
- **Proteus 8 Professional**

🧱 Part 1: MPLAB IDE – Programming Phase

◇ Step 1: Create Project Folder

- Example: `LED_Blinking`
- All files (`.c`, `.hex`, `.mcp`) will be stored here.

◇ Step 2: Create New Project

- Open **MPLAB IDE → Project → Project Wizard**
- Select **Device**: `PIC18F4580`
- Select **Compiler**: `Microchip C18 Toolsuite`
- Set file name: `led_blink.mcp`
- Save it inside your project folder

◇ Step 3: Write the Embedded C Program

Save file as: `led1.c`

```
#include <p18f4580.h>      // PIC18 header

// LED connected to RB4
#define mybit PORTBbits.RB4

// Delay function
void to_delay() {
    T0CON = 0x01;              // Timer0 ON, 16-bit, no prescaler
    TMR0H = 0x0B;              // Load High byte for delay
    TMR0L = 0xDC;              // Load Low byte for delay
    T0CONbits.TMR0ON = 1;      // Turn ON timer
    while (INTCONbits.TMR0IF == 0); // Wait for overflow
    T0CONbits.TMR0ON = 0;      // Turn OFF timer
    INTCONbits.TMR0IF = 0;     // Clear flag
}
```

```
void main() {
    TRISBbits.TRISB4 = 0;  // Set RB4 as output

    while (1) {
        mybit = 1;      // Turn ON LED
        to_delay();     // Wait
        mybit = 0;      // Turn OFF LED
        to_delay();     // Wait
    }
}
```

◇ Step 4: Add `.c` File to Project

- Right-click on `Source Files → Add Files → led1.c`

◇ Step 5: Build Project

- Click **Build All**
- Status: ✅ `BUILD SUCCEEDED`
- A `.hex` file will be created (used later in Proteus)

🔌 Part 2: Proteus – Hardware Simulation Phase

🧰 Required Tools:

- Proteus 8 Professional
- `.hex` file from MPLAB project

◇ Step 1: Create New Project in Proteus

- Project Name: `LED_Structure`
- Location: Same `LED_Blinking` folder
- Follow wizard:
  - ✅ Schematic layout
  - ✅ PCB layout
  - ✅ Firmware project (select PIC18F4580)

◇ Step 2: Design the Circuit

🔌 *Components to Add from Library:*

Component	Use
PIC18F4580	Microcontroller
LED-Red	Output light
Resistor (330 ohm)	To protect LED
Ground (GND)	Circuit grounding

⚡ *Wiring:*

- **Connect Resistor** to **RB4 (Pin 37)** of microcontroller
- **Connect LED** in series with resistor
- **Connect cathode (–)** of LED to **Ground**
- **VDD** (power) is already present in microcontroller

◇ Step 3: Load Hex File to Microcontroller

- Double-click on **PIC18F4580**
- In **Program File** → Browse → Select `led1.hex` from `LED_Blinking` folder

◇ Step 4: Run Simulation

- Click **Run (Green triangle)**

✅ LED will blink ON and OFF based on the delay defined in `to_delay()` function.

💡 How It Works:

Step	What Happens
`TRISB4 = 0`	Sets RB4 as **Output** pin
`PORTBbits.RB4 = 1`	Sends HIGH signal → LED ON
`PORTBbits.RB4 = 0`	Sends LOW signal → LED OFF
`to_delay()`	Uses **Timer0** to create a delay before toggling LED state

🧠 Key Concepts Recap:

☑ TRISx Register:

- Controls input/output mode
- `0` = Output, 1 = Input

☑ PORTx Register:

- Controls HIGH/LOW state on the pin

☑ Timer0:

- Used for delay generation (no `delay_ms()` in Embedded C)

🔍 Real-World Analogy:

Imagine a **streetlight** (LED) turning ON/OFF at intervals.

The **microcontroller** is like a **traffic controller**, deciding when to send power ON/OFF using a **timer**.

☑ Summary Flow:

```
Write Code → Save .c File → Build in MPLAB → Get .hex File →
Load .hex in Proteus → Connect LED + Resistor to RB4 → Run → Watch LED
Blink
```

🧪 Try It Yourself:

- Try connecting LED to **RB0**, **RB5** instead of RB4
- Change Timer values to **increase/decrease blinking speed**

◇ 1. What is an Interrupt?

An **interrupt** is a signal that **pauses the current task of the microcontroller**, handles a special task (interrupt service), and **then resumes the original task**.

🪵 *Analogy:*

You're reading a book (main task), and your phone rings (interrupt). You pause reading, attend the call (interrupt service), and then go back to reading.

◇ 2. Types of Interrupts

Type	Description
Hardware	Triggered by external devices like keyboard, mouse, printer, ADC, UART, etc.
Software	Triggered by software errors or instructions like divide by zero, etc.

◇ 3. Examples of Interrupts

- **Paper jam** while printing → Hardware interrupt
- **Disk full** message → Software interrupt
- Pressing **Ctrl + Alt + Del** → Hardware/software combined

◇ 4. Interrupt Vector Table (IVT)

It stores the **addresses of all Interrupt Service Routines (ISRs)**.

Priority	Address in RAM
High Priority	`0008H`
Low Priority	`0018H`
Power-on Reset	`0000H`

- Priority is decided by a **priority bit**:
  - `1` → High Priority
  - `0` → Low Priority

◇ 5. Sources of Interrupts in PIC18F4550

◇ *Hardware Sources:*

- **External Pins**: `INT0`, `INT1`, `INT2` (on `RB0`, `RB1`, `RB2`)
- **Peripherals**: UART, ADC, CCP, etc.

◇ *Software Sources:*

- **Timers**: `TMR0`, `TMR1`, `TMR2`
- **Errors**: Division by zero, arithmetic overflow

◇ 6. Polling vs Interrupts

Feature	Polling	Interrupt
Definition	MCU **repeatedly checks** all devices	Devices **notify** MCU only when needed
CPU usage	**Wastes time** checking every device	**Efficient**, only reacts when needed
Analogy	You **keep checking the oven** for cake	Oven **rings a bell** when cake is done

☑ ***Interrupts are more efficient than polling.***

◇ 7. Steps of Interrupt Execution

1. An **interrupt occurs**
2. Current instruction finishes (if low priority)
3. MCU **saves** current status to **stack**
4. **Jumps** to ISR (based on IVT)
5. Executes ISR instructions
6. After `RETFIE` (Return from Interrupt):
    a. Restores stack
    b. Resumes main program

◇ 8. Enabling & Disabling Interrupts in PIC18

The interrupt system is controlled by a **register with 7 bits**:

Bit Name	Bit No	Function
GIE	7	Global Interrupt Enable (1=enable all, 0=disable all)
PEIE	6	Peripheral Interrupt Enable
TMR0IE	5	Timer0 Interrupt Enable
INT0IE	4	External INT0 Enable
RBIE	3	Port B Change Interrupt Enable

✅ These bits can be set/cleared to **control individual interrupt sources**.

☑ Summary

- Interrupts allow multitasking and responsiveness in microcontrollers.
- They are **more efficient than polling**.
- **Hardware** and **Software** can both trigger interrupts.
- Interrupt vector table tells **where to go** when interrupt occurs.
- **GIE and related bits** control what is enabled or disabled.

☑ Notes: LED Interfacing with Microcontroller (PIC18F4550)

(Based on Unit 2 of Processor Architecture – 2019 pattern, Pune University)

◇ Introduction to LEDs

- **LED (Light Emitting Diode)** is a semiconductor device that emits light when current passes through it.
- It's used in almost all **electronic applications** due to its:

- Low power consumption
- Small size
- High brightness
- Long life

◇ Real-World Applications of LEDs

- **Consumer Electronics**: Televisions, remote controls, digital clocks
- **Indicator Lights**: In computers, cars, routers
- **Segment Displays**: Digital meters, elevator displays
- **Industrial Equipment**: Panel indicators
- **Agriculture**: LED lighting systems for indoor farming

◇ Advantages of LEDs

Advantage	Description
🔋 Energy Efficient	Consumes less power
📦 Small Size	Compact and easy to mount
🧲 No Magnetic Interference	Electromagnetic-safe
🔁 Fast Switching	On/off in microseconds
🍼 Long Life	No filament to burn out
🌈 Multiple Colors	Available in red, green, blue, UV etc.

◇ LED Types Used in Displays

1. **Single LEDs** – Simple ON/OFF indicators
2. **Seven-Segment Displays** – Used to display numbers (0–9)
    a. **Common Cathode (CC)**: All cathodes connected to ground
    b. **Common Anode (CA)**: All anodes connected to Vcc

3. **Multi-digit 7-segment displays**
    a. 2-digit, 3-digit, 4-digit LED displays
    b. Used to display values from **0001 to 9999**

◇ LED Interfacing Concept

Power Supply Connections

- Each LED is connected through a **current-limiting resistor** to avoid burning.
- Current flows from **logic HIGH (1)** to **logic LOW (0)**.
- If using **Common Cathode**, connect cathode to GND.
- If using **Common Anode**, connect anode to Vcc.

Microcontroller Side

- Each LED pin is connected to a **microcontroller port pin** (e.g., PORTB).
- You control ON/OFF via **C code** in MPLAB (using Embedded C).

Logic Table (for 7-Segment Display)

Digit	Segments ON (a–g)
0	a b c d e f
1	b c
2	a b g e d
…	…
9	a b c d f g

◇ Example: Displaying Number "2012"

- Use a **4-digit 7-segment display**
- Each digit is controlled one by one (multiplexing)
- The number “2012” is split into:
    - Digit 1: 2
    - Digit 2: 0

    - Digit 3: 1
    - Digit 4: 2
- Appropriate binary logic is sent to segments to glow correct LEDs

◇ Programming LED using C in MPLAB

- Code initializes TRIS registers (to set pins as output)
- Sends 1s and 0s to PORTx to control LED segments
- Uses delay to blink or display the digits sequentially

◇ Tips and Precautions

- Always use a **resistor** (220–330Ω) in series with LED
- Avoid connecting LEDs directly to **positive power supply**
- Understand the **bit pattern** for each number in 7-segment
- Multiplexing is used when **more digits** need to be displayed with fewer pins

◇ Summary Table

Feature	Description
Interfacing Type	LED to Microcontroller (PIC18F4550)
Applications	Displays, Indicators, Digital meters
Programming Language	Embedded C
Hardware Requirements	LED, Resistor, Breadboard, PIC Microcontroller
Display Range	0001 to 9999
Control Technique	Bitwise control through PORTs
Power Supply	+5V (typically)
Display Type	7-Segment (Common Cathode or Anode)

🧠 Real-World Analogy

Think of a **traffic light system** where red, yellow, and green LEDs turn on and off based on time and control logic. A microcontroller is like the traffic cop controlling these lights using coded instructions.

◇ Topic: LCD Interfacing with Microcontroller

◇ 1. Introduction

- LCD stands for **Liquid Crystal Display**.
- Widely used in **real-life applications** like:
  - Digital clocks, calculators
  - Microwave ovens
  - Instrument panels
  - Flight panel displays
  - Electronic gadgets
- LCDs display **characters, numbers**, and even **graphics** (in graphical LCDs).

◇ 2. Advantages of LCD over LED

Feature	LCD	LED
Power Consumption	Low	Moderate
Information	Can display text, characters, graphics	Limited to ON/OFF states or numeric output
Cost	Slightly expensive	Cheaper
Flexibility	High (multi-line, multi-character)	Limited (e.g., 7-segment only)

◇ 3. LCD Types Used

- Most commonly used in microcontroller interfacing:
  - **16×2 LCD** (16 characters per line × 2 lines)
  - **20×4 LCD**
- Internally contains **HD44780 LCD controller**.

◇ 4. LCD Pin Description (for 16x2 LCD)

Pin No	Name	Function
1	VSS	Ground
2	VDD	+5V Power Supply
3	VEE	Contrast Adjustment
4	RS (Register Select)	Command (0) / Data (1) selection
5	RW (Read/Write)	Write = 0, Read = 1
6	E (Enable)	Starts the data read/write
7–14	D0–D7	Data lines (8-bit data bus)
15	LED+	Backlight +ve (optional)
16	LED−	Backlight −ve (optional)

Usually, 4-bit data lines (D4–D7) are used to save microcontroller pins.

◇ 5. Steps for Interfacing LCD with Microcontroller

Step 1: Power Supply

- Connect VDD to **+5V**, VSS to **GND**.
- VEE connected to **potentiometer** for **contrast control**.

Step 2: Control Pins

- RS, RW, E connected to microcontroller digital I/O pins.
- Set RS = 0 for command, RS = 1 for data.

- Set RW = 0 for write mode.
- Enable pulse (E) must be toggled to latch command/data.

Step 3: Data Pins

- Connect D4–D7 for **4-bit mode** (saves I/O lines).
- Send **nibble-wise data** (high nibble first, then low nibble).

Step 4: Initialization

- LCD needs initialization before use:

```
// Example Initialization (pseudo-C)
LCD_Command(0x28); // 4-bit mode, 2 line, 5x7 matrix
LCD_Command(0x0C); // Display ON, Cursor OFF
LCD_Command(0x06); // Increment cursor
LCD_Command(0x01); // Clear display
```

6. Functions Used in Code

Command Function

Sends control commands to LCD (e.g., clear screen, set cursor):

```
void LCD_Command(unsigned char cmd) {
  RS = 0;
  RW = 0;
  DATA = cmd; // 8-bit or high nibble
  EN = 1;
  delay();
  EN = 0;
}
```

Data Function

Displays characters on LCD:

```
void LCD_Data(unsigned char data) {
  RS = 1;
  RW = 0;
  DATA = data;
  EN = 1;
  delay();
  EN = 0;
}
```

☑ *Display String*

```
void LCD_Display(char *str) {
  while(*str) {
    LCD_Data(*str++);
  }
}
```

◇ 7. Real-World Applications

- **Digital display boards**
- **Weather monitoring systems**
- **Home automation panels**
- **Security systems**
- **Embedded projects** showing sensor readings

◇ 8. Common Commands for LCD

Command	Hex Code	Description
Clear Display	`0x01`	Clears screen
Return Home	`0x02`	Cursor to beginning
Entry Mode Set	`0x06`	Auto increment
Display ON	`0x0C`	Cursor OFF

Display ON + Cursor ON	`0x0E`	Cursor blinking
4-bit Mode	`0x28`	Use 4-bit interface

◇ 9. Key Takeaways

- LCD is versatile for **displaying user-friendly output** in embedded systems.
- **8051**, **PIC**, **AVR**, and **ARM microcontrollers** can interface easily with LCDs.
- Practical demonstration includes:
  - Displaying **"Welcome" message**
  - Displaying **sensor values**
  - Menu systems

🔌 RS-232 Protocol and Serial Communication (Unit 4 – PIC Interfacing 2)

📌 Topics Covered

1. Parallel vs Serial Data Transfer
2. Serial Communication Protocols
3. Introduction to RS-232
4. Applications of RS-232
5. RS-232 Pin Description (DB9 Connector)
6. Working of RS-232 (Request–Response Protocol)
7. RS-232 Specifications

1️⃣ Parallel vs Serial Data Transfer

Feature	Parallel Communication	Serial Communication

Data Transfer	Multiple bits (e.g., 8-bit) at once	One bit at a time
Speed	High data transfer rate	Slower compared to parallel
Distance	Short-distance only	Suitable for long-distance
Cabling Cost	High (multiple lines needed)	Low (single line for data)
Reliability	Less reliable (cross-talk, skew)	More reliable over long distances
Used Protocols	GPIB, PCI, ISA	RS-232, USB, I^2C, SPI, UART
Example Use	Inside computer buses	Telecommunication lines

2️⃣ Serial Communication Protocols in Syllabus

- RS-232 ✅
- I^2C
- SPI
- UART

Today's focus: **RS-232**

3️⃣ What is RS-232?

- **RS** stands for **Recommended Standard**
- Developed by **EIA** (Electronics Industries Association) and **TIA** (Telecommunications Industry Association)
- Enables **bit-by-bit (serial)** communication between devices
- Connects **DTE** (Data Terminal Equipment, e.g., computer) and **DCE** (Data Communication Equipment, e.g., modem)
- Uses **D-type connectors (DB9 or DB25)** for physical connections

🔁 It is a **2-way serial communication protocol** (Full Duplex)

4️⃣ Applications of RS-232

Using **D-type connectors**, RS-232 is used to connect:

- ✅ Modems
- ✅ Printers
- ✅ Projectors
- ✅ Serial mouse or keyboard
- ✅ Embedded microcontrollers to PCs (via serial cable)
- ✅ Debugging and console logs

5️⃣ DB9 Connector Pin Description (RS-232)

Pin	Abbreviation	Full Form	Function
2	TXD	Transmit Data	Data from **computer → device**
3	RXD	Receive Data	Data from **device → computer**
6	DSR	Data Set Ready	Device is ready to receive data from PC
4	DTR	Data Terminal Ready	PC is ready to send data
7	RTS	Request To Send	PC sends request to device before sending data
8	CTS	Clear To Send	Device says “yes, I’m ready to receive”
1	DCD	Data Carrier Detect	Verifies connection between DTE and DCE is active
9	RI	Ring Indicator	Indicates **incoming signal** from device (like a ringing modem call)
5	GND	Ground	Reference for signal voltages

6️⃣ Working of RS-232 Protocol – Step-by-Step (Request–Response Protocol)

Example: PC (DTE) sends data to Modem (DCE)

1. **RTS (Request to Send)**
    a. PC tells modem: “I want to send data.”
2. **CTS (Clear to Send)**
    a. Modem replies: “I’m ready, send it.”
3. **Data Transmission**
    a. PC sends data bit-by-bit via TXD
4. **Optional: Reverse Transfer**
    a. Modem can also send data back (bidirectional)

This communication uses a **serial cable** and operates on **full duplex mode**.

7 RS-232 Protocol Specifications

Feature	Description
Communication Type	Serial (bit-by-bit), Full Duplex
Topology	Point-to-point
Signaling	Unbalanced (asymmetric)
Max Distance	~50 feet (15 meters)
Speed	Up to 19.2 kbps (typically 1.492 kbps in simple systems)
Power Supply	±12V logic levels (true = negative voltage)
Cable Type	DB9 / DB25 Serial Cable

END Summary

- RS-232 is a **reliable, simple protocol** used for **long-distance serial communication**.
- It enables communication between a computer (DTE) and external devices (DCE) using **request-response signals** like RTS/CTS.
- Widely used in **modems, printers, embedded systems, and diagnostics**.

I^2C Protocol – Inter-Integrated Circuit Communication

Topics Covered:

1. About I^2C Protocol
2. I^2C Terminologies
3. Working Steps of I^2C
4. Communication Example
5. Advantages and Disadvantages

1 About I^2C Protocol

Feature	Description
Full Form	**Inter-Integrated Circuit**
Developed By	**Philips**
Communication Type	**Serial (bit-by-bit)**
Alternate Name	**Two-wire Communication Protocol**
Use Case	Communication **between components** on the **same circuit board**
Communication Format	Master ↔ Slave(s)
Master–Slave Types	- One Master, Multiple Slaves - Multiple Masters, Multiple Slaves

Why Two-Wire?

- Only **two lines** required for communication:
  - SDA (Serial Data Line)
  - SCL (Serial Clock Line)

Common Applications:

Connecting sensors, ADCs, EEPROMs, LCDs, etc., to microcontrollers.

2️⃣ I^2C Terminologies

Term	Full Form	Purpose
SDA	Serial Data Line	Transfers data **bidirectionally** between Master and Slave
SCL	Serial Clock Line	Provides **timing/synchronization** of data transmission (controlled by Master)

🔁 Both lines are **shared** among all connected devices.

3️⃣ Working Steps of I^2C Protocol

🔄 I^2C Communication Sequence (Master → Slave)

Step	Description
1. **Start Condition**	Communication begins when **SDA goes High to Low** while **SCL is High**
2. **Address Frame**	Master sends a **7-bit or 10-bit address** to identify the target slave
3. **R/W Bit**	Master sends a bit to specify if it wants to **Read (1)** or **Write (0)**
4. **ACK from Slave**	If address matches, slave replies with an **ACK (Acknowledge)** signal
5. **Data Frame(s)**	Master sends (or receives) **8-bit data frames**
6. **ACK After Data**	Receiver sends **ACK** after each 8-bit data frame
7. **Stop Condition**	Communication ends when **SDA goes Low to High** while **SCL is High**

I²C Communication Example (Master → Slave 3)

Scenario:

- 1 Master
- 3 Slaves (Printer 1, 2, 3)
- Master wants to send data to **Slave 3**

Step-by-Step:

Step	Description
1. **Master Initiates**	Start Condition triggered via SDA and SCL
2. **Send Address**	Master sends address bits matching **Slave 3**
3. **Address Check**	Only Slave 3 accepts (others ignore)
4. **Slave ACK**	Slave 3 sends ACK (1-bit low)
5. **Data Sent**	Master sends data frames (e.g. `01011010`)
6. **Slave ACKs**	Slave acknowledges after each data frame
7. **Stop Condition**	Master sends Stop (SDA: Low → High while SCL is High)

Advantages of I²C Protocol

1. **Multiple Masters Supported**

→ More flexible compared to SPI or UART

2. **Only Two Wires Required**

→ Reduces PCB complexity and cost

3. **Simple Addressing System**

→ Master knows each slave's address

4. **Easy Expansion**

→ Add/remove devices without much rework

5. **Debug-Friendly**

→ Easy to test, debug using logic analyzers

❌ Disadvantages of I^2C Protocol

1. **Hardware Complexity**

→ Difficult when multiple masters & many slaves are used

2. **Half-Duplex Communication**

→ Cannot send and receive simultaneously

3. **Software-Managed (Stack Priority)**

→ Master manages multiple tasks using software stack, which adds processing overhead

🧠 Summary – Key Takeaways

- I^2C is a **synchronous, serial, two-wire protocol** for communication between ICs
- **Master** controls clock; **Slaves** respond when addressed
- Very useful in **embedded systems** for short-distance, onboard communications
- **Steps**: Start → Address → R/W → ACK → Data Frames → Stop
- Widely used for **sensors, EEPROM, RTC, displays**, etc.

📡 UART Protocol – Notes

☑ Topics Covered:

1. Introduction
2. Applications
3. Working Steps
4. UART Data Packet Format
5. Example
6. Advantages & Disadvantages

1 What is UART?

Feature	Description
Full Form	Universal Asynchronous Receiver and Transmitter
Type	**Physical Circuit** (not a communication protocol like I2C or RS-232)
Use	Used for **serial communication** between a **microcontroller and a device**
Nature	**Asynchronous** – No clock line required
Wiring	Only **2 wires** needed – TX (Transmit), RX (Receive)
Common Baud Rate	9600 bps (can be configured)
Communication Format	**1 Master ↔ 1 Slave** (point-to-point only)

🔁 Unlike I^2C or SPI, UART does **not use a clock line**, hence "asynchronous".

2 Applications of UART

🧩 UART is commonly used in:

- GPS modules
- Bluetooth modules
- GSM/GPRS modems
- Arduino and other embedded systems

- Serial terminals (via USB-to-serial)

🔌 **USB has largely replaced UART**, but UART is still crucial for:

- **Debugging**
- **Low-cost projects**
- **Microcontroller peripheral communication**

3️⃣ Working Steps of UART Protocol

🎯 Task:

Microcontroller sends data to a **GPS module** using UART.

Step	Description
1️⃣	**Microcontroller sends data** (in parallel) to UART Transmitter (UART1) via data bus
2️⃣	**UART1 Transmitter** converts it into a **serial data packet**
3️⃣	**UART1 TX pin** transmits the packet to UART2 RX pin
4️⃣	**UART2 Receiver** reads serial data bit-by-bit
5️⃣	UART2 checks validity using **parity**, then converts to parallel data
6️⃣	Final data sent to **GPS module** through data bus

4️⃣ UART Data Packet Structure

```
[Start Bit] [Data Bits (5–9)] [Optional Parity Bit] [Stop Bit(s)]
```

Element	Description
Start Bit	Indicates beginning of transmission Value = **0** (High → Low)
Data Bits	Actual data (5 to 9 bits)
Parity Bit	Optional bit for **error checking** - Even parity → even 1s - Odd parity → odd 1s
Stop Bit	Signals end of packet Value = **1** (Low → High)

🧠 **Example**: If data = `01001101`

- Start Bit = 0
- Parity (even parity, 4 ones → parity = 0)
- Stop Bit = 1

→ Packet: `0 01001101 0 1`

5️⃣ Example – Step-by-Step

🔌 Microcontroller → GPS Module

Step	Description
1.	Microcontroller sends: `01001101` to UART1 via data bus
2.	UART1 wraps it as: `0` (Start), `01001101` (Data), `0` (Even parity), `1` (Stop)
3.	UART1 sends serially via TX pin
4.	UART2 receives via RX pin and checks parity
5.	UART2 strips Start/Stop/Parity and recovers `01001101`
6.	UART2 sends this in parallel to GPS via data bus

✅ UART ensures data integrity using **parity**, even without a clock!

☑️ Advantages of UART

Advantage	Explanation
🔌 **Only 2 wires**	TX and RX, very simple
⏱️ **No clock**	Asynchronous → saves wiring
🧠 **Parity checking**	Detects simple transmission errors
🧰 **Flexible packet size**	5 to 9 data bits per packet
🌍 **Well-supported**	Standard interface in many systems (Arduino, PIC, etc.)

❌ Disadvantages of UART

Disadvantage	Explanation
⚠️ **Only one-to-one communication**	Can't connect multiple slaves like I^2C or SPI
📦 **Limited frame size**	Max 9 bits per packet
💻 **No synchronization**	Needs same baud rate on both sides or communication fails

🧠 Summary – Key Points

- **UART is a hardware module** inside microcontrollers.
- Enables **serial data transmission** using TX & RX only.
- Uses **start bits, stop bits, and optional parity** for data framing and error checking.
- Does **not require a clock line** (unlike SPI or I^2C).
- Widely used for communication with GPS, Bluetooth, and debugging tools.

☑️ UART Protocol (Universal Asynchronous Receiver and Transmitter)

◇ Introduction

- UART is not a communication protocol like I2C/SPI — it is a **hardware circuit** (inbuilt in microcontrollers).
- It handles **serial communication** between **one master and one slave** (point-to-point).
- Communication is **asynchronous** → No clock line needed.
- Requires **only 2 wires**:
  - **TX (Transmit)**
  - **RX (Receive)**

◇ Key Features

- Speed: Common baud rate is **9600 bps**.
- **Serial data transfer** (bit-by-bit).
- Used in simple microcontroller communications (e.g., Arduino ↔ GPS/Bluetooth module).

◇ Applications

- Connecting **Bluetooth, GPS, GSM**, or **sensors** to microcontrollers.
- Arduino or PIC-based **embedded systems**.

◇ Working Steps of UART

Example: Microcontroller sends data to GPS module using UART

◇ *Step 1: Parallel to Serial Conversion*

- Microcontroller sends **parallel data** to UART hardware (e.g., 8 bits).

◇ *Step 2: UART prepares data packet*

- Packet Format:

```
| Start Bit | Data Bits (5–9) | Parity Bit (optional) | Stop Bit |
```

    - **Start Bit**: 1 bit, always `0` → Tells receiver that data is starting.
    - **Data Bits**: Actual data (e.g., `01001101`)
    - **Parity Bit**:
        - For error detection.
        - Adds a `0` or `1` to make the number of 1s either even (even parity) or odd (odd parity).
    - **Stop Bit**: 1 or 2 bits, always 1 → Tells receiver that data has ended.

◇ *Step 3: Transmit Packet Serially*

- The data packet is sent bit-by-bit through **TX** pin to the receiver's **RX** pin.

◇ *Step 4: Receiver checks packet*

- Receiver checks:
  - **Start Bit** is present.
  - **Parity Bit** matches the data.
  - **Stop Bit** confirms packet end.
- If correct, it removes Start/Parity/Stop bits and extracts the **actual data**.

◇ *Step 5: Serial to Parallel Conversion*

- Receiver (e.g., GPS module) converts serial bits to parallel format and uses the data.

◇ Advantages of UART

- Uses only **2 wires** (TX and RX).
- No need for clock signal.
- **Parity bit** provides basic **error detection**.
- **Well-documented** and easy to use.

◇ Disadvantages of UART

- **Only 1 master and 1 slave** supported.
- Max data size = **9 bits per frame**.
- **Lower speed** than SPI or I2C.

☑ SPI Protocol (Serial Peripheral Interface)

◇ Introduction

- Developed by **Motorola**, mid-1980s.
- Used for **short-distance**, **high-speed**, and **synchronous** communication.
- Commonly used in systems like:
  - Microcontroller ↔ LCD, EEPROM, ADC, Flash Memory, RTC
- Uses **1 Master and multiple Slaves**.
- **Full duplex**: Simultaneous data transmission and reception.

◇ SPI Interface Pins

Pin	Full Form	Role
MOSI	Master Out, Slave In	Master sends data to Slave
MISO	Master In, Slave Out	Slave sends data to Master
SCLK	Serial Clock	Master-generated clock
SS/CS	Slave Select / Chip Select	Selects which slave to talk to

🔁 Analogy: Imagine a teacher (master) giving and receiving assignments from multiple students (slaves), one at a time, using labeled folders (SS) and talking bit-by-bit in sync with a metronome (SCLK).

◇ Working Steps

Example: Microcontroller sends data to ADC, Sensor, LED (3 slaves)

◇ *Step 1: Send Clock*

- Master sends **clock pulses** to all slaves via SCLK.

◇ *Step 2: Select Slaves*

- Master activates specific slaves by pulling **SS/CS LOW**.

◇ *Step 3: Send Data (MOSI)*

- Master sends data bit-by-bit to the selected slave using **MOSI**.

◇ *Step 4: Receive Response (MISO)*

- The selected slave sends back data to the master using **MISO**.

◇ SPI Communication Types

◇ *1. Independent SPI (default)*

- **Each slave** is connected directly to master.
- Each slave has **its own SS line**.
- All slaves receive clock and MOSI lines.

◇ *2. Daisy Chain SPI*

- Slaves are **connected in a chain**.
- Master sends data to **Slave 1 → Slave 2 → Slave 3**.
- Data flows through each slave in a series.
- Only **one SS line** required.

⚙️ Use Independent SPI when all slaves need independent communication. Use Daisy Chain SPI when order matters and fewer SS lines are preferred.

◇ SPI Transmission Example

Master (microcontroller) ↔ Slave (ADC)

1. Master sends **clock signal** to slave.
2. Master pulls **SS LOW** to select the slave.
3. Master sends data (e.g., `11000`) via **MOSI**, MSB first.
4. Slave responds (e.g., `1101`) via **MISO**, LSB first.

◇ Advantages of SPI

- **Faster** than UART/I2C (up to 10 Mbps or more).
- **No start/stop/parity bits**, simple protocol.
- Supports **full-duplex** communication.
- Can connect **multiple slaves**.

◇ Disadvantages of SPI

- **4 wires** needed (vs. 2 in UART/I2C).
- No built-in **error checking** (like UART's parity bit).
- **No acknowledgement** system.
- **No addressing** like I2C → Needs one SS pin per slave.
- Only **one master** supported.

☑ Summary Table

Feature	UART	SPI
Type	Asynchronous	Synchronous
Wires Used	2 (TX, RX)	4 (MOSI, MISO, SCLK, SS)
Speed	Lower (e.g., 9600 bps)	Higher (up to 10 Mbps)
Error Checking	Parity Bit	❌ None
Master/Slave Support	1 Master, 1 Slave	1 Master, Multiple Slaves
Start/Stop Bits	✅ Yes	❌ No
Data Direction	Half Duplex	Full Duplex
Hardware Required	UART module	SPI module