ESP32 WiFi Connectivity

1. Overview: ESP32 WiFi Connectivity

- **Key Topics**:

1. **Connecting ESP32 to WiFi** (Access Point / Station mode)
2. **Creating server and client objects**
3. **Using the WiFi library (WiFi.h)**
4. **Using the WebServer library (WebServer.h)**
5. **Basic syntax and functions** to handle WiFi connections and web servers on the ESP32.

---

2. Connecting ESP32 to a WiFi Network

2.1 Station Mode (ESP32 connects to an existing network)

- **Station Mode** means the ESP32 will act like a normal “client” device connected to your router.
- **Typical Steps**:
  1. **Include the library**: #include <WiFi.h>
  2. **Set WiFi credentials**: e.g., const char* ssid = "Your_SSID"; const char* password = "Your_PASSWORD";
  3. **Begin connection**: WiFi.begin(ssid, password);
  4. **Check connection status** in a loop until connected:
     - Use WiFi.status() to check if it’s WL_CONNECTED.
  5. **Obtain IP address**: Use WiFi.localIP().

2.2 Access Point (AP) Mode (ESP32 creates its own network)

- **Access Point Mode** means the ESP32 itself creates a WiFi network that other devices can connect to.
- **Typical Steps**:
  1. **Include the library**: #include <WiFi.h>
  2. **Set AP credentials**: e.g., const char* ssid = "MyESP32AP"; const char* password = "MyPassword";
  3. **Begin AP**: WiFi.softAP(ssid, password);
  4. **Get AP IP**: WiFi.softAPIP() to retrieve the IP address of the AP.

---

3. WiFi.h Library: Important Classes & Functions

Below are the **basic functions from WiFi.h** that are typically used.

3.1 WiFi.begin(ssid, password)

- **Purpose**: Initiates the WiFi connection in **station mode**.
- **Arguments**:
  - ssid: A const char* containing the SSID (network name).
  - password: A const char* containing the WiFi password.
- **Return Value**: Usually wl_status_t (though not always used). Typically, you check WiFi.status() after calling WiFi.begin().

3.2 WiFi.begin(ssid, password, channel, bssid, connect)

- **Overloaded variant** that allows specifying:
  - WiFi channel,
  - BSSID (MAC address of AP),
  - whether to connect (bool).
- **Mostly used** if you need more control over the connection parameters.

3.3 WiFi.status()

- **Purpose**: Returns the current status of the WiFi connection.
- **Return Value**: An integer/enum of type wl_status_t. Common values:
  - WL_IDLE_STATUS (0): Idle status.
  - WL_NO_SSID_AVAIL (1): SSID cannot be reached.
  - WL_CONNECTED (3): Connected to the WiFi network.
  - WL_CONNECT_FAILED (4): Connection failed.
  - WL_DISCONNECTED (6): Disconnected from the network.

3.4 WiFi.localIP()

- **Purpose**: Returns the IP address assigned to the ESP32 in station mode.
- **Return Value**: An IPAddress object (often printed with .toString() in code).

3.5 WiFi.softAP(ssid, password)

- **Purpose**: Configures the ESP32 as an **Access Point**.
- **Arguments**:
  - ssid: Name of the AP to create. password: Password for the AP (can be left empty).
- **Return Value**: bool indicating success or failure (true on success).

3.6 WiFi.softAPIP()

- **Purpose**: Gets the IP address of the ESP32 when it's acting as an Access Point.
- **Return Value**: An IPAddress object.

3.7 WiFi.disconnect()

- **Purpose**: Disconnect from the currently connected WiFi network.
- **Return Value**: wl_status_t or void depending on implementation. Typically used to ensure a fresh reconnect or to switch modes.

3.8 WiFi.mode(wifi_mode_t mode)

- **Purpose**: Set the WiFi operating mode.
- **Arguments**:
    - mode: can be WIFI_OFF, WIFI_STA, WIFI_AP, or WIFI_AP_STA.
- **Return Value**: bool (in some versions) or void. Helps explicitly control the WiFi mode.

3.9 WiFi.setHostname(hostname)

- **Purpose**: Assign a custom hostname to your device in station or AP mode.
- **Arguments**:
    - hostname: a const char* for the device name.
- **Return Value**: bool success/failure in some implementations.

3.10 WiFi.getMode()

- **Purpose**: Get the current WiFi mode (station, AP, or both).
- **Return Value**: wifi_mode_t.

3.11 WiFi.scanNetworks()

- **Purpose**: Scans for nearby WiFi networks.
- **Return Value**: int representing the number of networks found.

---

4. Creating a Server and Client Objects

4.1 Server Object

- Typically done with classes like **WiFiServer** (for raw TCP) or **WebServer** (for HTTP).
- **WiFiServer server(port)**:
    - Creates a server that listens on the specified port (default HTTP is 80).
- **server.begin()**:
    - Starts listening for incoming client connections.

- **server.available()**:
  - Checks if a client has connected; returns a **WiFiClient** object if a client is available.

4.2 Client Object

- **WiFiClient client = server.available();**:
  - Retrieves an available client.
- **Common Methods**:
  - client.connected(): Checks if the client is still connected.
  - client.print(...) / client.println(...): Sends data to the client.
  - client.read() / client.readString(): Reads incoming data.
  - client.stop(): Disconnects the client.

These are **basic** WiFi server/client usage patterns. In practice, for a **web server**, you might use WebServer class for simplicity.

---

5. WebServer.h Library: Important Classes & Functions

When building an HTTP server (i.e., a web server) on ESP32, the **WebServer** class (or ESP32WebServer in older libraries) is often used. The below functions and methods are commonly mentioned in the tutorials like LastMinuteEngineers' "Creating ESP32 Web Server".

5.1 WebServer server(port)

- **Purpose**: Create an HTTP web server object that listens on a specified TCP port (default 80).
- **Arguments**:
  - port: The port number on which the server will listen (e.g., 80).
- **Return Value**: None (it's a constructor). You then use server to set up request handlers.

5.2 server.begin()

- **Purpose**: Start the HTTP server so it begins listening for incoming requests.
- **Arguments**: None.
- **Return Value**: void.

5.3 server.on(uri, handler_function)

- **Purpose**: Specify which function (handler) to call when a client requests a particular **URI** (endpoint).
- **Arguments**:
  - uri: A string (e.g., "/", "/LED").
  - handler_function: A function **without arguments** that defines how to handle that request.

Example Explanation:

```
server.on("/", handleRoot);
```

- Means: when the client requests "/", call handleRoot() function to send the response.

5.4 server.on(uri, HTTP_METHOD, handler_function)

- **Purpose**: Overloaded version that can specify which HTTP method (GET, POST, etc.) triggers the handler.
- **Arguments**:
  - uri: A string for the path.
  - HTTP_METHOD: e.g., HTTP_GET, HTTP_POST.
  - handler_function: The function to handle that request.
- **Return Value**: void.

5.5 server.handleClient()

- **Purpose**: Needs to be called **frequently** (often in loop()) to process incoming client requests.
- **Arguments**: None.
- **Return Value**: void.

This method handles all incoming HTTP requests and routes them to the appropriate handler function(s).

5.6 server.send(status_code, content_type, content)

- **Purpose**: Send a response back to the client.
- **Arguments**:
  - status_code: e.g., 200 (OK), 404 (Not found), etc.
  - content_type: e.g., "text/html", "text/plain".
  - content: The actual string or HTML you want to send as the response body.
- **Return Value**: void.

5.7 server.arg(name)

- **Purpose**: Retrieve the value of a parameter (argument) from the client's request, typically in GET or POST requests.
- **Arguments**:
  - name: The name/key of the parameter you want to get.
- **Return Value**: A String containing the parameter value.

5.8 server.hasArg(name)

- **Purpose**: Check if the client request included a parameter with a given name.

- **Arguments**:
    - name: The name/key of the parameter to check.
- **Return Value**: bool (true if the argument is found, false if not).

5.9 server.method()

- **Purpose**: Returns the HTTP method of the current request.
- **Return Value**: A type enumerated for GET, POST, etc.
- **Usage**: Used inside a handler function to see if the request was GET or POST.

5.10 server.uri()

- **Purpose**: Returns the **URI** (path) requested by the client.
- **Return Value**: String of the path, e.g. "/", "/data", etc.

---

6. Putting It All Together: Typical Flow

1. **Include Libraries**

```
#include <WiFi.h>
#include <WebServer.h>
```

2. **Define Credentials** (for Station mode)

```
const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";
```

3. **Connect to WiFi**

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
  // maybe print dots or a status message
}
```

4. **Create WebServer Object**

```
WebServer server(80);
```

5. **Handle Requests**

```
void handleRoot() {
  server.send(200, "text/html", "<h1>Hello from ESP32</h1>");
}
server.on("/", handleRoot);
```

6. **Start the Server**

```
server.begin();
```

7. **In loop(), call**

```
server.handleClient();
```

8. **(Optional) Access Point Setup**

```
WiFi.softAP("MyESP32AP", "MyPassword");
IPAddress IP = WiFi.softAPIP();
```

---

7. Common Pitfalls & Tips

- **Check WiFi Status** with WiFi.status() to ensure your device is connected before starting the server in station mode.
- Always call **server.handleClient()** inside the main loop() function for the web server. Otherwise, no requests will be processed.
- Use **WiFi.mode(WIFI_STA)** or **WiFi.mode(WIFI_AP_STA)** to explicitly set the mode you need.
- If you want the device to have a **static IP** in station mode, use WiFi.config() before WiFi.begin().
- For simple projects, you might not need advanced parameters, but for robust setups, be mindful of your **channel**, **BSSID**, and **hostname** if dealing with more advanced network configurations.

---

8. Key Terms & Definitions (Quick Reference)

- **SSID**: Service Set Identifier – the name of the WiFi network.
- **Password**: The WiFi network's passphrase (if WPA/WPA2 secured).
- **Station (STA) Mode**: ESP32 acts as a client device connecting to an existing WiFi router.
- **Access Point (AP) Mode**: ESP32 creates its own WiFi network.
- **Server**: A program that listens for incoming requests (e.g., HTTP requests) from clients.
- **Client**: A program or device that connects to a server to request or send data.
- **HTTP**: HyperText Transfer Protocol – the protocol used for serving web pages.
- **URI**: Uniform Resource Identifier – path portion of the URL, e.g., /, /temp.
- **Handler**: A function that handles an incoming request (defines how to respond).

Non blocking delays

1. Non-Blocking Delays: Why They Matter

- **Definition**: A **non-blocking delay** is a timing approach where the CPU is **not** halted or frozen by a function like delay(). Instead, the microcontroller keeps doing other tasks while periodically checking if a specified interval has passed.
- **Contrast with delay()**: The standard delay(ms) is **blocking**; it suspends code execution for ms milliseconds. During this time, no other operations occur.
  - **Problem**: Blocking can cause issues in time-sensitive or multitasking scenarios.
  - **Solution**: Use non-blocking approaches to allow code to continue running (polling sensors, responding to inputs, etc.) while "waiting" for a delay interval to complete.

---

2. Core Concept: millis() Based Timing

Most common non-blocking delay patterns in Arduino or ESP32 rely on **millis()**, which returns the number of milliseconds since the board was powered on (overflow occurs after ~49 days).

1. **Store a "previous time"**:
   - Example: unsigned long previousMillis = 0;
2. **Define an "interval"**:
   - Example: const unsigned long interval = 1000; // 1 second
3. **Check if the interval has passed**:

```
if (millis() - previousMillis >= interval) {
    previousMillis = millis();
    // do something here (e.g., toggle LED)
}
```

4. **Result**: The code **only** executes the "do something" block after the interval passes but never uses delay(), so the loop can keep running other code in the meantime.

---

3. PLC Techniques Adapted for Microcontrollers

- **PLC Scan Cycle**:
  - In PLCs, the program repeatedly runs a cycle:
    1. Read inputs
    2. Execute logic
    3. Update outputs
    4. Housekeeping tasks
  - The loop runs continuously and **never blocks**.
- **Applying to Microcontrollers**:

1. **Structure your loop()** so it mimics a PLC scan: read sensors, execute state machine logic, update outputs, etc.
2. **No blocking calls** (like delay()) so the "scan" keeps repeating quickly.

Key Points from PLC Approach

- **Deterministic**: Ensures tasks run at known intervals if coded carefully.
- **Event/State-based**: Use **states** (like "waiting", "running", "completed") to handle tasks.
- **Timers**: Instead of blocking delays, PLCs use "timer instructions." On microcontrollers, we replicate with millis() checks.
- **Easier debugging**: Because everything runs each cycle, it's straightforward to see logic flow in each iteration.

---

4. Common Patterns for Non-Blocking Delays

4.1 Blink Without Delay (Classic Example)

- **Goal**: Toggle an LED every second without stopping the program flow.
- **Steps**:
    1. Define previousMillis and interval.
    2. In loop(), compare millis() with previousMillis.
    3. When enough time has passed, toggle the LED and store the new previousMillis.
    4. The rest of the loop() can handle other tasks freely.

Example Code Snippet:

```
const int ledPin = 2; // Example LED pin
unsigned long previousMillis = 0;
const unsigned long interval = 1000; // 1 second

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  unsigned long currentMillis = millis();

  // Check if time interval has passed
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    // Toggle LED
    digitalWrite(ledPin, !digitalRead(ledPin));
  }

  // Other code can run here without delay
}
```

- **Outcome**: LED blinks every second, and the loop remains unblocked.

4.2 Multiple Timers with millis()

- **Goal**: Handle several timed tasks in parallel (e.g., blinking LED1 at 1 second interval, LED2 at 500 ms interval).
- **Method**: Use separate previousMillis variables (or arrays) for each timed task. Each task has its own interval. Check them all in loop().

---

5. State Machines for More Complex Tasks

- **When**: For more advanced timing sequences, you might have multiple steps or conditions (like a motor ON for 2s, OFF for 3s, repeat).
- **Non-Blocking State Machine**:
  1. **States**: e.g., STATE_START, STATE_DELAY_1, STATE_DELAY_2, ...
  2. **Transitions**: Use millis() checks to see when to move from one state to the next.
  3. **No delay()**: Each time through loop(), you check if it's time to move to the next state.

---

6. Handling millis() Overflow

- **Definition**: millis() eventually wraps around to 0 after about 49 days on Arduino/ESP32.
- **Typical Pattern**:

```
if ((currentMillis - previousMillis) >= interval) {
  // works fine even if currentMillis overflows
}
```

- **Reason**: Unsigned integer subtraction handles overflow gracefully. As long as you use **unsigned long** and do the standard comparison, it's safe.

---

7. Advantages of Non-Blocking Delays

- **Responsiveness**: System can react to sensor changes, incoming data, or user input at any time.
- **Parallel Tasks**: Multiple timed operations can run seemingly "in parallel" using separate checks.
- **Scalability**: Easy to add new timed tasks without messing up existing ones.
- **Energy Efficiency**: Allows you to incorporate low-power modes effectively, or at least better usage of CPU cycles.

---

8. Key Terms & Definitions (Quick Reference)

- **Blocking Delay**: A code routine (like delay()) that stops all other code execution until the delay is over.
- **Non-Blocking Delay**: A technique that checks if a certain time has elapsed, letting the rest of the code run freely in the meantime.
- **millis()**: A built-in function returning milliseconds since power-up or reset.
- **Previous Time (previousMillis)**: A stored value of millis() from the last event to calculate elapsed time.
- **Interval**: A fixed duration (in milliseconds) you want between operations.
- **State Machine**: A way of organizing code in discrete "states," transitioning to new states based on conditions or time checks.

---

9. Putting It All Together

1. **Initialize** variables (previousMillis, intervals, state flags).
2. **In setup()**, set up pins or states, ensure no calls to delay().
3. **In loop()**:
   - Continuously read **currentMillis = millis()**.
   - For each timed event:
     - Check **if (currentMillis - previousMillis >= interval)**:
       - Update previousMillis = currentMillis.
       - Perform the action (e.g., toggle LED, move to next state).
   - Handle other tasks in the same loop, no waiting required.
4. **For Complex Sequences**: Use state machines with conditional logic driven by millis() checks.

Interrupts in Arduino

1. Overview: What Are Interrupts?

- **Definition**: An interrupt is a signal that temporarily halts (interrupts) the normal program flow so that a special function (the **Interrupt Service Routine**, or ISR) can be executed.
- **Why Use Them?**:
  - **Responsiveness**: Quickly react to events (e.g., button press, sensor trigger, timer overflow).
  - **Efficiency**: No need to constantly **poll** an input in the main loop.
- **Key Rule**: Keep the ISR **short and fast**. Avoid using delay() or complex operations (Serial printing, dynamic memory allocation, etc.) inside ISRs.

---

2. External Interrupts with attachInterrupt()

2.1 Function Prototype

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

- **digitalPinToInterrupt(pin)**: Converts an Arduino digital pin number to the corresponding external interrupt number (e.g., on Arduino UNO, digital pins 2 and 3 map to interrupt 0 and 1, respectively).
- **ISR**: The name of the Interrupt Service Routine function that will run when the interrupt occurs.
- **mode**: Defines the condition that triggers the interrupt:
  - **RISING**: Trigger when the pin goes from LOW to HIGH.
  - **FALLING**: Trigger when the pin goes from HIGH to LOW.
  - **CHANGE**: Trigger when the pin changes value (LOW ↔ HIGH).
  - **LOW**: Trigger whenever the pin is LOW (level-triggered).

2.2 Example Usage

```
void setup() {
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(2), myISR, FALLING);
}

void loop() {
  // Main code runs here continuously
}

void myISR() {
  // Code in ISR (keep short!)
}
```

- **Explanation**:
    - This code attaches an external interrupt to pin 2.
    - The ISR named myISR will be called whenever **pin 2 transitions from HIGH to LOW** (FALLING).

2.3 Important Notes / Best Practices

- **ISR Restrictions**:
    - No delay(), no large computations, no Serial.print().
    - Use **volatile** variables if you modify them in ISR and read them in the main loop.
- **Detach Interrupt**: You can remove an interrupt using detachInterrupt(digitalPinToInterrupt(pin));.
- **Supported Pins**:
    - Arduino UNO, for example, only supports external interrupts on digital pins 2 and 3.
    - Other boards (Mega, Leonardo, etc.) have more.

---

3. Pin Change Interrupts

3.1 What Are Pin Change Interrupts?

- **Definition**: Pin Change Interrupts allow **any** digital pin (or a group of pins) to trigger an interrupt when the state of the pin(s) changes.
- **Difference from External Interrupts**:
    - External interrupts are limited to certain pins (like 2 & 3 on UNO).
    - Pin change interrupts can be used on a **wider range** of pins but are grouped into “ports” or “banks.”
    - The ISR is triggered if **any** pin in the port changes; code inside ISR must determine which pin actually caused the interrupt.

3.2 How It Works (Arduino UNO Example)

- **Hardware Groupings**:
    - Port B (digital pins 8–13),
    - Port C (analog pins 0–5 when used as digital),
    - Port D (digital pins 0–7).
- Each port has its own pin change interrupt vector (e.g., PCINT0_vect for Port B, PCINT1_vect for Port C, PCINT2_vect for Port D).
- You **enable** the pin change interrupt for a specific pin by:
    1. Setting the **Pin Change Interrupt Enable** for the port (PCIE).

2. Setting the **Pin Change Mask** for the specific pin(s) (PCMSK).

3.3 Example Configuration Steps (Simplified)

1. **Enable global interrupts** (using sei() or interrupts() in Arduino).
2. **Enable Pin Change Interrupt** for the port:
   - For Port B, set PCIE0 bit in PCICR.
3. **Enable the specific pin** in the mask register:
   - For pin 8 (which is PB0), set bit 0 in PCMSK0.
4. **Write the ISR**:

```
ISR(PCINT0_vect) {
  // Check which pin caused interrupt
  // e.g., read pin 8, do something
}
```

5. Alternatively, some Arduino libraries & macros simplify these steps for you.

3.4 Pros & Cons

- **Pros**:
  - More flexible than attachInterrupt (not limited to pins 2/3 on UNO).
  - Good for multiple input triggers.
- **Cons**:
  - More complex to set up.
  - Single ISR for all pins on a port – you must figure out which pin changed.

---

4. Timer and Timer Interrupts

4.1 Introduction to Arduino Timers

- **Definition**: Timers are hardware counters that increment at a specific rate (based on clock frequency and prescalers).
- **Why Use Timers?**
  - Generate precise periodic interrupts (e.g., for tasks that need to run at fixed intervals).
  - Measure pulse durations or frequencies.
  - PWM signal generation.

4.2 Types of Arduino Timers (UNO Example)

1. **Timer0**: 8-bit, used for millis()/micros() functions by default.
2. **Timer1**: 16-bit, often used for more precise timing or servo control.
3. **Timer2**: 8-bit, can also be used for audio frequency generation, etc.

4.3 Timer Modes

- **Normal Mode**: Timer counts up and triggers an overflow interrupt when it rolls over.
- **CTC (Clear Timer on Compare)**: Triggers an interrupt when the timer matches a compare value, then resets the timer to 0.

4.4 Configuring a Timer Interrupt (Example with Timer1)

1. **Select the Mode**: e.g., CTC mode.
2. **Set the Compare Value**: e.g., OCR1A = desired count.
3. **Set the Prescaler**: e.g., /1, /8, /64, etc., to adjust timer increment speed.
4. **Enable the Interrupt**:
    - TIMSK1 |= (1 << OCIE1A); for Compare A Match interrupt on Timer1.
5. **Write the ISR**:

```
ISR(TIMER1_COMPA_vect) {
  // Code to run on each compare match event
}
```

4.5 Example Calculation

- Suppose you want an interrupt every 1 ms on a 16 MHz Arduino with Timer1 in CTC mode:
    - Timer1 increments at 16 MHz / prescaler.
    - If prescaler = 8, then timer ticks at 2 MHz (16 MHz / 8).
    - For 1 ms intervals (1 kHz), we need 2000 ticks (because 2 MHz = 2,000,000 ticks/s).
    - So we set OCR1A = 1999 (because counting starts at 0, so 2000 counts total).
- This yields an interrupt at 1 kHz.

4.6 Common Interrupt Vectors (UNO)

- **Timer1**:
    - ISR(TIMER1_OVF_vect) – Timer1 overflow
    - ISR(TIMER1_COMPA_vect) – Compare match channel A
    - ISR(TIMER1_COMPB_vect) – Compare match channel B

Each timer has similar registers/interrupt vectors; specifics vary by microcontroller.

4.7 Best Practices for Timer ISRs

- Keep code short (just like any ISR).
- Avoid delay() or heavy computations.
- If you need to share data with the main loop, use **volatile** variables or **atomic** operations.

5. Key Differences and Usage Scenarios

1. **External Interrupts (attachInterrupt())**:
    - Triggers on specific pins (e.g., 2 & 3 on UNO).
    - Useful for immediate response to high-priority external events (button press, sensor, encoder).
2. **Pin Change Interrupts**:
    - Triggers on a change of state for a group of pins.
    - Suitable when you have multiple inputs that can all trigger an action, but you must decode which pin changed.
3. **Timer Interrupts**:
    - Trigger based on a hardware timer's count reaching a certain value (compare match) or overflowing.
    - Ideal for periodic tasks (e.g., generating waveforms, refreshing displays, sampling sensors at fixed intervals, etc.).

---

6. Practical Tips & Common Pitfalls

- **ISR Must Be Fast**:
    - Don't do anything that takes too long or depends on interrupts (like Serial).
    - Typically, just set a flag or increment a counter and exit.
- **Global Interrupt Enable**:
    - Make sure interrupts are globally enabled (interrupts() in Arduino or sei() in AVR C).
- **Debouncing**:
    - Mechanical switches can cause **bouncing**. If using external or pin change interrupts for a button, you might get multiple triggers. Either handle debouncing in software or via hardware (RC circuits, specialized libraries).
- **Shared Variables**:
    - If an ISR updates a variable used in the main loop, declare it **volatile**.
    - For multi-byte variables (e.g., long on an 8-bit AVR), consider reading/writing them **atomically** or disabling interrupts briefly around the access to avoid data corruption.
- **Timer Conflicts**:
    - If you reconfigure Timer0 (used for millis(), micros()), you'll lose or alter the default Arduino timekeeping.
    - Some libraries rely on certain timers, so changing them might cause conflicts.

---

7. Key Terms & Definitions (Quick Reference)

- **Interrupt**: A hardware or software mechanism that interrupts normal program flow.
- **ISR (Interrupt Service Routine)**: A special function called when an interrupt triggers.
- **RISING/FALLING/CHANGE/LOW**: Trigger modes for external interrupts.
- **Pin Change Interrupt**: An interrupt triggered when a pin's level changes (usable on multiple pins, grouped by ports).
- **Timer**: A hardware counter that increments at a known rate, can trigger interrupts upon overflow or compare match.
- **Prescaler**: A hardware divider that slows down (or speeds up) the timer tick frequency.
- **Overflow**: When the timer's count goes past its maximum (e.g., 255 for 8-bit, 65535 for 16-bit) and rolls over to zero.
- **Compare Match**: A condition where the timer count matches a specified register (OCRnA, OCRnB, etc.), triggering an interrupt in CTC mode.
- **volatile**: A keyword ensuring the compiler does not optimize away accesses to a variable that can change unexpectedly (e.g., in an ISR).

Naive Bayes Algorithm

1. Basic Probability Concepts

1.1 Random Experiment

- **Definition**: An action or process that leads to **one** of several possible outcomes.
- **Example**: Rolling a die (outcomes 1–6).

1.2 Sample Space

- **Definition**: The **set of all possible outcomes** of a random experiment.
- **Example**: For rolling a die, the sample space S={1,2,3,4,5,6}.

1.3 Event

- **Definition**: Any **subset** of the sample space.
- **Example**: Event A = “Roll is an even number” = {2,4,6}.

1.4 Probability of an Event AAA

- **Definition**: P(A) is the **likelihood** that event A occurs.
- **Formula** (classical definition if outcomes are equally likely):

$$P(A) = \frac{\text{Number of outcomes in } A}{\text{Total number of outcomes in } S}.$$

- **Example**: Probability of rolling an even number (event A) is 3/6=1/2.

1.5 Conditional Probability

- **Definition**: The probability of event AAA **given** event BBB has occurred.
- **Notation**: P(A|B).
- **Formula**:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)},$$

where P(A∩B) is the probability that both A and B occur.

2. Bayes' Theorem

2.1 Statement

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}.$$

Interpretation:

- Lets you **update** the probability of a hypothesis (event AAA) after seeing new evidence (event BBB).
- **Example** (Medical test scenario): Probability of having a disease (A) given a positive test result (B).

2.2 Basic Example

- Suppose:
  - P(A)=0.01 (1% chance of having a disease),
  - P(B|A)=0.99 (99% chance test is positive if you have the disease),
  - P(B)=0.05 (5% chance of test being positive overall, including false positives).
- By Bayes' Theorem:

$$P(A \mid B) = \frac{0.99 \times 0.01}{0.05} = 0.198 \approx 19.8\%.$$

- **Interpretation**: Even with a test that's 99% accurate, the probability you actually have the disease (given you tested positive) is about 19.8% — that's the power of Bayes' Theorem in re-evaluating probabilities with new evidence.

---

3. Naive Bayes Theorem (Classifier) Basics

3.1 What is Naive Bayes?

- **Definition**: A **classification algorithm** based on applying **Bayes' Theorem** with the "naive" assumption that **features are conditionally independent** given the class label.
- **Why "Naive"?**: Because it **assumes** each feature contributes independently to the outcome, ignoring any correlation between features.

3.2 Naive Bayes Classifier Formula

For a classification task with classes CkC_kCk (e.g., "play tennis" vs. "don't play tennis"), and features x1,x2,...,xn:

$$P(C_k \mid x_1, x_2, \ldots, x_n) \;\propto\; P(C_k) \times P(x_1 \mid C_k) \times P(x_2 \mid C_k) \times \cdots \times P(x_n \mid C_k).$$

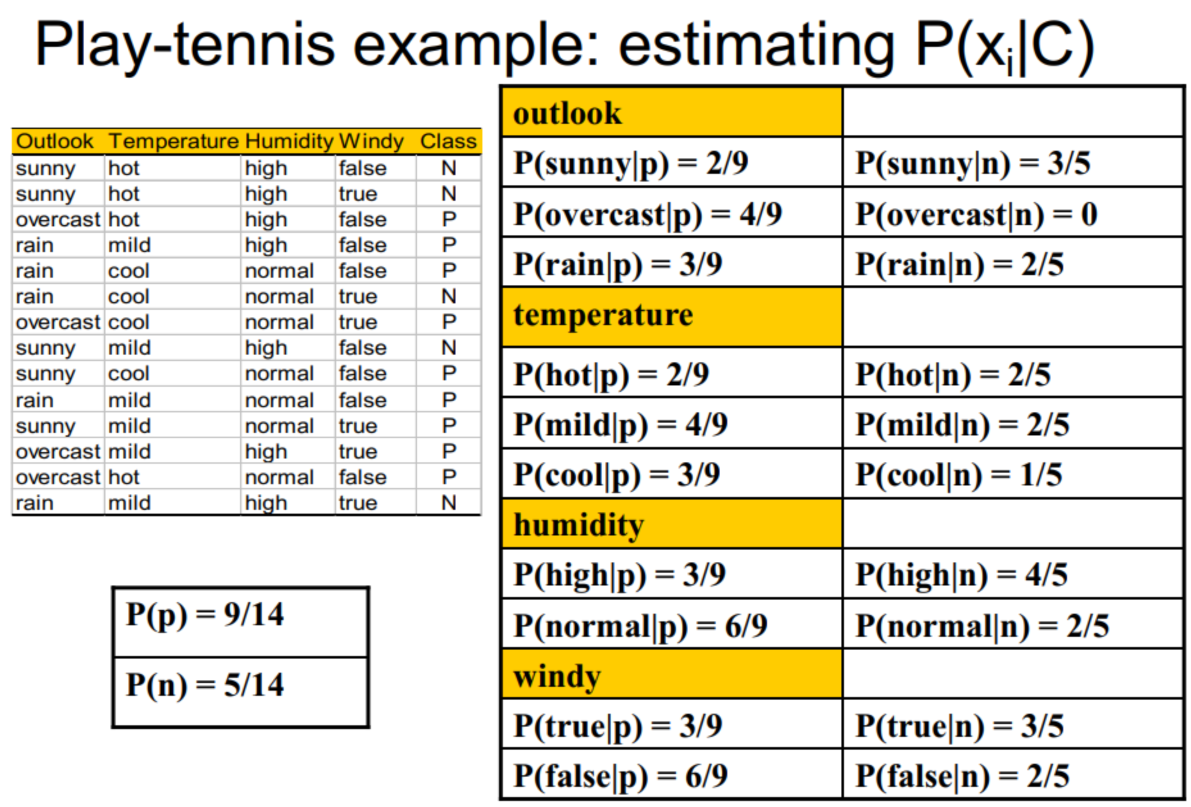
- **Explanation**:
  - P(Ck) is the **prior probability** of class Ck.
  - P(xi|Ck) is the **likelihood** of feature xi given class Ck.
  - The symbol “∝” means “proportional to,” because typically we **normalize** at the end to ensure all class probabilities sum to 1.

3.3 Classification Rule

- Compute the above product for **each** class Ck.
- **Choose** the class that gives the **highest** posterior probability.

3.4 Simple Example (Hypothetical)

- Suppose we want to classify an email as **Spam** (Cspam) or **Not Spam** (Cnot-spam).
- We have features (words) x1=“sale”,x2=“cheap”, etc.
- We estimate:
  - $P(C_{\text{spam}})$, $P(\text{“sale”} \mid C_{\text{spam}})$, etc.
- Then use:
  $P(C_{\text{spam}} \mid \text{“sale”}, \text{“cheap”}) \propto P(C_{\text{spam}}) \times P(\text{“sale”} \mid C_{\text{spam}}) \times P(\text{“cheap”} \mid C_{\text{spam}})$.
- Do the same for $C_{\text{not-spam}}$, then compare.



Play-tennis example: estimating $P(x_i|C)$

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	false	N
sunny	hot	high	true	N
overcast	hot	high	false	P
rain	mild	high	false	P
rain	cool	normal	false	P
rain	cool	normal	true	N
overcast	cool	normal	true	P
sunny	mild	high	false	N
sunny	cool	normal	false	P
rain	mild	normal	false	P
sunny	mild	normal	true	P
overcast	mild	high	true	P
overcast	hot	normal	false	P
rain	mild	high	true	N

P(p) = 9/14	
P(n) = 5/14	

outlook	
P(sunny|p) = 2/9	P(sunny|n) = 3/5
P(overcast|p) = 4/9	P(overcast|n) = 0
P(rain|p) = 3/9	P(rain|n) = 2/5
temperature	
P(hot|p) = 2/9	P(hot|n) = 2/5
P(mild|p) = 4/9	P(mild|n) = 2/5
P(cool|p) = 3/9	P(cool|n) = 1/5
humidity	
P(high|p) = 3/9	P(high|n) = 4/5
P(normal|p) = 6/9	P(normal|n) = 2/5
windy	
P(true|p) = 3/9	P(true|n) = 3/5
P(false|p) = 6/9	P(false|n) = 2/5

Given a training set, we can compute the probabilities

Outlook	P	N
sunny	2/9	3/5
overcast	4/9	0
rain	3/9	2/5
Tempreature		
hot	2/9	2/5
mild	4/9	2/5
cool	3/9	1/5

Humidity	P	N
high	3/9	4/5
normal	6/9	1/5
Windy		
true	3/9	3/5
false	6/9	2/5

Play-tennis example: classifying X

- An unseen sample X = <rain, hot, high, false>

- P(X|p)·P(p) =
  P(rain|p)·P(hot|p)·P(high|p)·P(false|p)·P(p) =
  3/9·2/9·3/9·6/9·9/14 = 0.010582
- P(X|n)·P(n) =
  P(rain|n)·P(hot|n)·P(high|n)·P(false|n)·P(n) =
  2/5·2/5·4/5·2/5·5/14 = 0.018286

- Sample X is classified in class n (don’t play)

Port Registers

1. Overview: Why Use Port Registers?

- **Definition**: Port registers give **direct, low-level access** to the microcontroller's I/O pins.
- **Efficiency**: Reading/writing with port registers is **faster** than using pinMode(), digitalWrite(), digitalRead().
- **Tradeoff**: Less readability and portability. Code is specific to the **AVR architecture**.

---

2. Arduino Port Mapping

On an **Arduino Uno** (ATmega328P), digital pins map to ports as follows:

Port	Register Names	Arduino Pins	Bits	Typical Usage
B	DDRB, PORTB, PINB	Digital pins **8** to **13**	Bits **0 to 5**	Pin 13 often used for built-in LED
C	DDRC, PORTC, PINC	**Analog** pins **A0** to **A5**	Bits **0 to 5**	Typically used for analog input pins
D	DDRD, PORTD, PIND	Digital pins **0** to **7**	Bits **0 to 7**	Pins 0/1 used for Serial RX/TX

Bit Values and Directions

- **DDRx**:
  - 1 in DDR bit → **Pin acts as OUTPUT**
  - 0 in DDR bit → **Pin acts as INPUT** (default)
- **PORTx**:
  - When pin is OUTPUT: 1 → **Pin goes HIGH**, 0 → **Pin goes LOW**
  - When pin is INPUT: 1 → **Enables Internal Pull-up**, 0 → **No pull-up**
- **PINx**:
  - **Read-Only** register. Reflects current pin states: 1 = HIGH, 0 = LOW.

---

3. pinMode() vs Direct DDR Manipulation

3.1 pinMode() Explanation

```
// Standard Arduino approach
pinMode(5, OUTPUT);
```

- Internally, this sets the bit in DDRD corresponding to pin 5 to 1.
- Pin 5 is on **Port D**, **bit 5**.

Equivalent Direct Register Approach:

```
// Using bitwise OR (safer, doesn't overwrite other bits)
DDRD |= B00100000; //Sets bit 5 of DDRD to 1, making pin 5 an OUTPUT

// Alternatively (but overwrites other bits!):
DDRD = B00100000;
```

Example 1: Simple LED Setup

Using Arduino Functions

```
void setup() {
  pinMode(5, OUTPUT);  // Set pin 5 as output
}

void loop() {
  digitalWrite(5, HIGH);
  delay(500);
  digitalWrite(5, LOW);
  delay(500);
}
```

Using Port Registers

```
void setup() {
  // Set pin 5 (bit 5 of DDRD) as output
  DDRD |= (1 << 5);  // same as DDRD |= B00100000;
}

void loop() {
  // Set pin 5 HIGH
  PORTD |= (1 << 5);  // same as PORTD |= B00100000;
  delay(500);

  // Set pin 5 LOW
  PORTD &= ~(1 << 5); // same as PORTD &= ~B00100000;
  delay(500);
}
```

Key Points:

- |= (bitwise OR) sets the specific bit without altering others.
- &= ~(...) (bitwise AND with the inverse) clears the specific bit.

4. digitalWrite() vs Direct PORT Manipulation

4.1 digitalWrite() Explanation

```
// Standard Arduino approach
digitalWrite(5, HIGH);
```

- Internally, this sets **bit 5 of PORTD** to 1 if pin 5 is configured as OUTPUT.

Equivalent Direct Register Approach:

```
// Setting pin 5 HIGH
PORTD |= B00100000;

// Setting pin 5 LOW
PORTD &= ~B00100000;
```

Example 2: Multiple Pins in One Go

Using Arduino Functions

```
// Turn pins 5 and 6 HIGH, pin 7 LOW
digitalWrite(5, HIGH);
digitalWrite(6, HIGH);
digitalWrite(7, LOW);
```

Using Port Registers

```
// Suppose pins 5,6,7 are all OUTPUT already
// Turn pins 5 & 6 HIGH in one operation
PORTD |= (1 << 5) | (1 << 6);
// Turn pin 7 LOW
PORTD &= ~(1 << 7);
```

Benefit: Fewer operations = faster code.

---

5. digitalRead() vs Direct PIN Reading

5.1 digitalRead() Explanation

```
// Standard Arduino approach
int val = digitalRead(5);
```

- Reads **bit 5 of PIND** and returns 1 if set, 0 otherwise.

Equivalent Direct Register Approach:

```
int val = (PIND >> 5) & 0x01;
```

- **Right-shift** PIND by 5 so bit 5 becomes bit 0.
- **Mask** with 0x01 (binary 00000001) to keep only that bit.

Example 3: Reading a Button

Using Arduino Functions

```
void setup() {
  pinMode(5, INPUT_PULLUP); // internal pull-up on
  Serial.begin(9600);
}

void loop() {
  int buttonState = digitalRead(5);
  Serial.println(buttonState);
  delay(200);
}
```

Using Port Registers

```
void setup() {
  Serial.begin(9600);

  // Pin 5 as INPUT
  DDRD &= ~(1 << 5); // bit 5 in DDRD = 0
  // Enable internal pull-up
  PORTD |= (1 << 5); // bit 5 in PORTD = 1
}

void loop() {
  // Read the pin directly
  int buttonState = (PIND >> 5) & 0x01;
  Serial.println(buttonState);
  delay(200);
}
```

Note: A 0 reading means the button is pressed to ground (since pull-up is active).

---

6. Interrupts with Port Registers (Pin Change Interrupts)

1. **PCICR (Pin Change Interrupt Control Register)**:
   - Bits **0,1,2** (PCIE0, PCIE1, PCIE2) enable pin change interrupts for **Port B, C, D**.
2. **PCMSK (Pin Change Mask Register)**:
   - **PCMSK0** for Port B, **PCMSK1** for Port C, **PCMSK2** for Port D.
   - Setting a bit in these registers **enables** pin-change interrupt on that specific pin.

Example 4: Pin Change Interrupt on Pin 5

Using Arduino approach (some libraries do this, or use direct references)

(No direct Arduino function for pin-change interrupts, so we typically set the registers manually.)

Direct Register Approach

```
void setup() {
  // Enable pin change interrupt for Port D => set PCIE2 (bit 2)
in PCICR
  PCICR |= (1 << PCIE2);

  // Enable pin change interrupt for pin 5 => set bit 5 in
PCMSK2
  PCMSK2 |= (1 << PCINT21);
  // (PCINT21 corresponds to PD5 on ATmega328)

  // Set Pin 5 as input
  DDRD &= ~(1 << 5);

  // Optionally enable pull-up
  PORTD |= (1 << 5);

  // Global interrupt enable
  sei();
}

ISR(PCINT2_vect) {
  // Runs whenever a change occurs on Port D’s enabled pins
  if (PIND & (1 << 5)) {
    // Pin 5 is HIGH
  } else {
    // Pin 5 is LOW
  }
}

void loop() {
  // Main program
}
```

Key Points:

- PCIE2 is bit 2 in PCICR, enabling pin-change interrupts for **Port D**.
- PCINT21 is the pin-change interrupt number for **PD5** specifically.
- We then use PIND & (1 << 5) to check the current level.

---

7. Efficiency & Cautions

- **Speed**: Accessing port registers is **much faster** than standard Arduino functions because it avoids the overhead of function calls.
- **Risk**: Direct writing (=) can **overwrite** other bits if not using OR (|=) or AND (&=).
  - Example: DDRD = 0x20; sets **only** bit 5, clearing all others. Use DDRD |= 0x20; if you want to preserve existing bits.
- **Portability**: Code is less portable. On different microcontrollers (like ARM-based boards), the registers differ.

---

8. Timers & Clock Cycles (Recap from Class Notes)

While **port registers** specifically affect **I/O** efficiency, **timers** are also controlled by hardware registers:

1. **Clock**: Arduino Uno runs at **16 MHz** → **16 million cycles/sec**.
2. **Time per clock cycle**: $\frac{1}{16\times10^6} \approx 62.5\,\mathrm{ns}.$
3. **Timers**:
   - **Timer0** & **Timer2** = 8-bit
   - **Timer1** = 16-bit
4. **Overflow**:
   - 8-bit: 0→255 = 256 counts → ~16µs to overflow at no prescaling
   - 16-bit: 0→65535 = 65536 counts → ~4.096ms to overflow at no prescaling
5. **Prescalers** (e.g., 8, 64, 256, 1024) slow down the timer increments.
   - Example: Prescaler 64 → Timer increments every (62.5ns × 64) = 4µs.

---

9. Extended Code Example: Faster Toggle via Port Registers

Objective: Toggle pin 13 (built-in LED on Port B, bit 5) at a high frequency.

Using Arduino Functions

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  digitalWrite(13, LOW);
}
```

- This will toggle the LED but **includes function call overhead** each time.

Using Port Registers

```
void setup() {
  // Pin 13 -> Port B, bit 5
  DDRB |= (1 << 5);   // Set bit 5 of DDRB => OUTPUT
}

void loop() {
  PORTB |= (1 << 5);  // Set bit 5 => HIGH
  PORTB &= ~(1 << 5); // Clear bit 5 => LOW
}
```

- This code toggles pin 13 **much faster** than the standard approach.
- In practice, you'd add some delay or do more meaningful tasks in between toggles.

10. Summary

1. **Direct Register Access**:
   - **DDRx** → pin direction (INPUT/OUTPUT).
   - **PORTx** → output high/low or pull-up enable.
   - **PINx** → read input state.
2. **Bitwise Ops**:
   - |= to **set** a bit.
   - &= ~() to **clear** a bit.
   - >> n to shift bits (for reading).
3. **Mapping Pins**: Know which port/bit corresponds to each Arduino pin.
4. **Interrupts**:
   - **PCICR** & **PCMSK** for **pin change interrupts**.
   - Examples show enabling **PCIE2** and setting the correct bit in **PCMSK2** for a given PD pin.
5. **Timers**:
   - Not strictly part of port manipulation, but related to **hardware register** usage.
   - Understand **overflow**, **prescalers**, **compare match** if it comes up.
6. **Advantages**:
   - **Speed** and **fine-grained control**.
   - Potential for advanced, efficient embedded code.
7. **Disadvantages**:
   - Less **code clarity**.
   - **Hardware-specific**, not portable across different microcontroller families.

Detailed Explanation of Pin-Change Interrupts, Timers, Timer interrupts, and Prescalars.

---

1. Pin Change Interrupt Control Register (PCICR) & Pin Change Mask (PCMSK)

What They Are & Why They Matter

- **Pin Change Interrupt** allows **any** (enabled) pin on a port to generate an interrupt when its logic level **changes** (from LOW to HIGH or HIGH to LOW).
- **PCICR** (Pin Change Interrupt Control Register) has **3 bits** (PCIE0, PCIE1, PCIE2) that enable pin-change interrupts **for an entire port**:
    - PCIE0 → Port B
    - PCIE1 → Port C
    - PCIE2 → Port D
- **PCMSK** (Pin Change Mask Register) decides **which specific pins** on that port will trigger the interrupt.
    - PCMSK0 → for Port B
    - PCMSK1 → for Port C
    - PCMSK2 → for Port D

Together, they let you say:

1. "I want to enable pin-change interrupts on this port" (via PCICR).
2. "I want these specific pins on that port to cause the interrupt" (via PCMSK).

Example: Configure Arduino pin-5 as an Interrupt Pin

- **Pin 5** on Arduino Uno is **PD5** → belongs to **Port D**.

1. **Enable pin change interrupt for Port D** by setting **PCIE2** in **PCICR**:

```
PCICR |= B00000100;
// This sets bit 2 (PCIE2) of PCICR to 1
```

2. **Enable pin-change for pin PD5** by setting the corresponding bit in **PCMSK2**:
    - Pin 5 is **bit 5** in Port D, but sometimes you'll see it referred to as PCINT21 internally.
    - Example (simplified):

```
PCMSK2 |= B00100000;
// This sets bit 5 in PCMSK2, enabling pin-change on PD5
```

3. **Set PD5 as input** (optionally enable pull-up):

```
pinMode(5, INPUT);
// or DDRD &= ~(1<<5); // direct register approach
// optionally PORTD |= (1<<5) for pull-up
```

4. **Write an ISR** for pin-change on Port D:

```
ISR(PCINT2_vect) {
  // This function is automatically called on ANY change on
enabled pins in Port D
  // Check if PD5 is now HIGH or LOW:
  if (PIND & (1 << PD5)) {
    // PD5 is HIGH
  } else {
    // PD5 is LOW
  }
}
```

Use Cases:

- React to a **button press** or a **sensor** that toggles a digital pin.
- Count pulses on a pin (though there are also dedicated external interrupts or input capture for that).

---

2. Timers, Clock, and Prescalers

What Is a Timer?

- A **timer** is basically a **hardware counter** in the microcontroller.
- It increments every clock cycle (or every few clock cycles if you use a **prescaler**).
- On Arduino Uno:
    - **Timer0**: 8-bit (counts 0 → 255)
    - **Timer1**: 16-bit (counts 0 → 65535)
    - **Timer2**: 8-bit (counts 0 → 255)

(The names can differ on other boards, but the idea is the same.)

Clock Frequency (16 MHz on Arduino Uno)

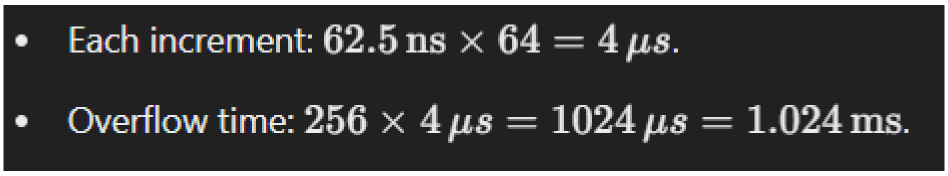
- The crystal runs at **16 million pulses per second**.
- One clock cycle = 1/(16×106)1 / (16 \times 10^6)1/(16×106) seconds = **62.5 ns**.
    - That means every **62.5 ns**, the hardware clock "ticks".

Prescaler

- A **prescaler** is a little divider. It slows down how fast the timer increments by dividing the incoming clock.
- Common prescale factors: **1, 8, 64, 256, 1024**.
- **Example**: If prescaler = 8,
    - Timer increments every 62.5 ns × 8 = 500 ns
    - This helps achieve longer delays without the timer rolling over too quickly.

Prescaler Numerical Example

- Suppose Timer0 is 8-bit, so it goes 0→255.
- With **no prescaler (1)**, each increment is 62.5 ns, so it overflows after 256 × 62.5 ns ≈ 16 µs
- If you need a slower overflow, use a bigger prescaler. For instance, **prescaler = 64**:



- Each increment: $62.5\,\text{ns} \times 64 = 4\,\mu s$.
- Overflow time: $256 \times 4\,\mu s = 1024\,\mu s = 1.024\,\text{ms}$.

---

3. Types of Timer Interrupts

Timers can generate **interrupts** based on different conditions:

1. **Compare Match Interrupt**
   - You set a **target value** (in the OCR – Output Compare Register).
   - When the timer counter = that value, **interrupt** triggers.
   - **Usage**: Precisely time events or waveforms. E.g., toggle an LED every time the timer hits the compare value, thus controlling frequency.
2. **Overflow Interrupt**
   - The interrupt occurs when the timer counts from its max back to 0 (for an 8-bit timer, from 255→0).
   - **Usage**: Good for periodic tasks, but the rate depends on your clock + prescaler. If you need a more flexible rate, use compare match.
3. **Input Capture Interrupt**
   - When an **external signal** changes on a specific pin, the timer's current count is "captured" into a register, and an interrupt is triggered.
   - **Usage**: Measuring frequency or time between pulses (e.g., measuring how many microseconds between two rising edges of a signal).

Real-World Examples

- **Compare Match**:
  - Precisely blink an LED at 1 kHz, or output a PWM signal for motor speed.
- **Overflow**:
  - Periodically update some variable every time the timer overflows.
  - E.g., a simple "ticker" that increments a millisecond counter.
- **Input Capture**:
  - Count how long between two pulses from a sensor to measure RPM or frequency.

---

4. Creating Custom Delays with Timers

Why Not Just Use delay(n)?

- delay(n) is a **blocking** function. The CPU sits idle until n milliseconds elapse.
- If you want **more precise** or **non-blocking** timing, or if you need to do other tasks while waiting, timer interrupts can help.

Basic Formula

$$\text{Timer Value} = \frac{(\text{CPU Frequency})}{\text{Prescaler}} \times (\text{Desired Delay})$$

- Then you place that **Timer Value** into the OCR (Output Compare Register), if you're using CTC mode.
- When the timer hits that value, the **Compare Match Interrupt** fires.

Formula Example

- **CPU freq** = 16 MHz
- **Prescaler** = 128
- **Desired delay** = 1 ms = 10−310^{-3}10−3 s

$$\text{Value} = \frac{16 \times 10^6}{128} \times 10^{-3} = 125$$

- So you'd set **OCR0A** = 125 (assuming Timer0 or Timer2 can be set to prescaler=128 – note that standard prescalers are often 1, 8, 64, 256, 1024, so 128 might be from a special mode or on a different microcontroller; the principle is the same).

Simple CTC Example (Timer0)

Below is a **bare-bones** demonstration of setting up **Timer0** in **CTC (Clear Timer on Compare)** mode for a custom delay.

```
// We'll blink LED at pin 13 precisely using Timer0 compare match.
#define LED_PIN 13

void setup() {
  pinMode(LED_PIN, OUTPUT);

  // 1) Set the timer mode to CTC (Clear Timer on Compare)
  //    For Timer0, WGM01 is bit 1 in TCCR0A.
  //    WGM0 bits: WGM02 in TCCR0B, WGM01:0 in TCCR0A
  //    010 = CTC mode with OCR0A
  TCCR0A = (1 << WGM01);   // WGM01=1 => CTC mode, WGM00=0 => skip

  // 2) Set the compare value (OCR0A)
  // Suppose delay ~1ms with prescaler 64 => OCR0A = 249 for ~1ms
  // because 16 MHz / 64 = 250kHz => 1 tick = 4us => 250 ticks =
     1ms => but we start from 0 => set 249)
  OCR0A = 249;

  // 3) Set prescaler
  // We set CS00,CS01,CS02 bits in TCCR0B for prescaler = 64
  // CS02:0 = 011 => 64
  TCCR0B = (1 << CS01) | (1 << CS00); // = 0b00000011 => prescale=64

  // 4) Enable Compare Match Interrupt on OCR0A
  TIMSK0 = (1 << OCIE0A);

  // 5) Enable global interrupts
  sei();
}

ISR(TIMER0_COMPA_vect) {
  // This ISR fires every ~1ms
  // Toggle the LED
  static bool ledState = false;
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState);
}
void loop() {
  // The LED toggling is handled entirely by the timer interrupt,
  so we can do anything else we want here without delay() blocking.
}
```

- **Explanation**:
  - We set **CTC mode** so the timer resets to zero whenever it hits OCR0A (249).
  - With **prescaler=64**, each tick is 4µs. 250 ticks = 1ms.
  - The interrupt routine toggles the LED. **Every 1ms** it flips the LED state, so the LED toggles 1000 times a second (i.e., 500 ON–OFF cycles per second = 500 Hz flash).

(You can adjust prescaler and OCR0A to get different intervals.)

5. Register Names & Bits

TCCR (Timer/Counter Control Register)

- **TCCR0A / TCCR0B**, **TCCR1A / TCCR1B**, **TCCR2A / TCCR2B**: They set the **mode** (e.g., normal, CTC, PWM) and the **prescaler**.
- **WGMx (Waveform Generation Mode)** bits:
  - WGM**0**1, WGM**0**0 for Timer0's mode, plus WGM**0**2 in TCCR0B.
  - Example:
    - WGM02=0, WGM01=0, WGM00=0 → **Normal mode**
    - WGM02=0, WGM01=1, WGM00=0 → **CTC mode**
- **CSx2, CSx1, CSx0** bits: Select prescaler. (CS stands for "Clock Select")

CSx2	CSx1	CSx0	Timer Clock
0	0	0	Stopped
0	0	1	No prescaling (1)
0	1	0	clk/8
0	1	1	clk/64
1	0	0	clk/256
1	0	1	clk/1024

TCNT (Timer Counter Register)

- The actual **count** is stored here and increments on each tick.

OCR (Output Compare Register)

- The **target** value for generating a compare match interrupt or controlling PWM duty cycle.

TIMSK (Timer Interrupt Mask)

- **TIMSK0**, **TIMSK1**, **TIMSK2**: Turn interrupts on/off for each timer.
  - For compare match A: set **OCIE0A**, **OCIE1A**, **OCIE2A**, etc.
  - For overflow: set **TOIE0**, **TOIE1**, etc.

6. Putting It All Together: Real-World Use Cases

1. **Pin Change Interrupt**:
   - A **button** on pin 5 triggers an interrupt whenever you press or release it.
   - A **hall sensor** on a wheel gives digital pulses you want to count precisely.
2. **Timer Compare Match**:
   - **Generate a stable frequency** for a digital waveform (like a precise beep).
   - **Blink an LED** on/off at exact intervals without using delay().
3. **Timer Overflow**:
   - **Keep a simple timebase** (like a millisecond counter) by incrementing a variable each time the timer overflows.
4. **Input Capture**:
   - Measure the **period** of an external signal (like the frequency of a rotating fan).
5. **Prescaler**:
   - Use it to match your desired timing range. If you need a 1-second interval, a prescaler helps you not overflow the counter too fast.

Key Takeaways

1. **PCICR & PCMSK**: Let you enable **pin change interrupts** on specific ports and pins.
2. **Timers** are **counters** that increment at a rate determined by **clock** and **prescaler**.
3. **Interrupt Types**:
   - **Compare Match**: Trigger precisely at a set count value.
   - **Overflow**: Trigger on maximum count wrap-around.
   - **Input Capture**: Latch the timer value on an external event.
4. **Prescaler** = hardware “divider,” used to slow the timer increment for longer intervals.
5. **No More Blocking**: Replacing delay() with timer interrupts offers **greater precision** and **non-blocking** operation.

Sample Exam (Short/Medium/Numerical/Long)

Short Questions:

1. **Explain the purpose of WiFi.begin(ssid, password) in ESP32 WiFi connectivity.**

Answer: WiFi.begin(ssid, password) is used to initiate the connection process to a WiFi network by providing the network's SSID and password. It starts the process of connecting the ESP32 to the specified WiFi network.

2. **What is the main difference between blocking and non-blocking delays in embedded systems?**

Answer: Blocking delays halt the execution of the program for a specified time, preventing the CPU from doing other tasks. Non-blocking delays allow the CPU to perform other tasks while waiting, improving the system's efficiency.

3. **What are port registers in Arduino, and why are they used?**

Answer: Port registers are special function registers in Arduino used to control the input/output (I/O) pins more efficiently. They allow direct manipulation of the I/O pins, which is faster than using standard digital read/write functions.

Medium-Length Questions:

4. **Describe how to set up a web server on ESP32 using the Arduino IDE.**

Answer: To set up a web server on ESP32:

- Include the necessary libraries like WiFi.h and WebServer.h.
- Initialize WiFi using WiFi.begin(ssid, password) and wait for the connection to be established.
- Create a WebServer object and define the routes and handler functions using server.on().
- Start the server using server.begin().
- In the loop function, call server.handleClient() to handle incoming client requests.

5. **Explain how attachInterrupt() is used in Arduino to handle interrupts.**

Answer: attachInterrupt() is used to attach a specific function to an external interrupt on a digital pin. It takes three parameters: the interrupt pin number, the ISR (Interrupt Service Routine) function to execute, and the mode (e.g., RISING, FALLING, CHANGE, or LOW). When the specified condition occurs, the ISR function is called, allowing the program to respond to external events asynchronously.

Long Technical Questions:

6. **Discuss the process of connecting the ESP32 to a WiFi network and creating a server-client architecture. Provide detailed explanations of the functions and objects involved.**

Answer:

- **Connecting to WiFi:**
  - Use WiFi.begin(ssid, password) to start the connection to the WiFi network.
  - Monitor the connection status using WiFi.status() in a loop until it returns WL_CONNECTED.
- **Server-Client Architecture:**
  - Create a server using the WebServer class: WebServer server(port).
  - Define routes using server.on(path, handler), where path is the URL endpoint and handler is the function to process requests.
  - Start the server with server.begin().
  - Continuously call server.handleClient() in the main loop to handle client requests.

7. **Analyze the use of non-blocking delays in embedded systems. Explain how they improve system performance, with an example using the millis() function in Arduino.**

Answer:

- **Non-Blocking Delays:** Allow the CPU to continue executing other tasks while waiting for a specific time to elapse, improving system responsiveness and multitasking.
- **Example:** Using millis() to create a non-blocking delay:

```
unsigned long previousMillis = 0;
const long interval = 1000; // 1 second

void setup() {
  // initialization code
}

void loop() {
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    // code to execute every interval
  }
  // other tasks can be performed here
}
```

In this example, the system can perform other tasks in the loop without being stuck in a delay function, enhancing performance.

8. **Illustrate how timer interrupts work in Arduino and describe a scenario where they are beneficial.**

Answer:

- **Timer Interrupts:** Timer interrupts are triggered when a timer overflows or reaches a specific value, allowing periodic execution of code without polling.
- **Setup:**
  - Configure the timer using registers like TCCR1A, TCCR1B, and OCR1A.
  - Enable the interrupt with TIMSK1.
  - Define an ISR using ISR(TIMER1_COMPA_vect).
- **Scenario:** In a real-time clock application, a timer interrupt can trigger every second to update the clock display, ensuring precise timekeeping without continuously checking the time in the main loop.

Numericals:

9. **Given an ESP32 WiFi setup where the connection takes an average of 5 seconds, calculate how many connection attempts can be made in 2 minutes.**

Answer:

- Total time available = 2 minutes = 120 seconds.
- Time per connection attempt = 5 seconds.
- Number of connection attempts = Total time / Time per attempt = 120 / 5 = 24 attempts.

10. **An Arduino program uses a non-blocking delay with an interval of 200 ms. Calculate how many times the delay loop will execute in 1 minute.**

Answer:

- Total time available = 1 minute = 60 seconds = 60,000 milliseconds.
- Time per iteration = 200 milliseconds.
- Number of iterations = Total time / Time per iteration = 60,000 / 200 = 300 iterations.