Name: Laiba Fatima

Registration# 23-ntu-cs-1257

Subject: IOT

BSAI-5TH

ASSIGNMENT# 1

Question # 1

1. Why is volatile used for variables shared with ISRs?

The **volatile** keyword is used for variables that are shared with **interrupt service routines (ISRs)** because their values can change at any time, not just during the normal program flow. It tells the compiler **not to optimize or store** the variable's value in a register, but instead always **read it directly from memory**. This makes sure the program always gets the **latest value** that might have been changed by the ISR, preventing wrong or outdated data from being used.

2. Compare hardware-timer ISR debouncing vs. delay()-based debouncing.

Hardware-timer ISR debouncing uses a timer interrupt to check the button state after a short, fixed time, which makes it **more accurate and doesn't block** the rest of the program. It lets the microcontroller do other tasks while waiting.
On the other hand, **delay()-based debouncing** simply pauses the program for a few milliseconds after a button press, which is **easier but blocks** everything else during the delay. So, timer-based debouncing is **better for multitasking**, while delay()-based is **simpler but less efficient**.

3. What does IRAM_ATTR do, and why is it needed?

IRAM_ATTR tells the compiler to **store a function in the internal RAM (IRAM)** instead of flash memory. It is needed because when an **interrupt happens**, the CPU must quickly run the ISR (Interrupt Service Routine). If the ISR is in flash, it might be **too slow or temporarily unavailable**, but if it's in IRAM, it runs **faster and safely** even during flash operations.

4. Define LEDC channels, timers, and duty cycle.

LEDC channels are like separate lines that control different LEDs or devices using PWM signals.
LEDC timers decide the **frequency and resolution** of those PWM signals (how fast and smooth they switch on and off).
The **duty cycle** means how long the signal stays ON compared to OFF — for example, a higher duty cycle makes the LED brighter because it stays ON for a longer time in each cycle.

5. Why should you avoid Serial prints or long code paths inside ISRs?

We should avoid **Serial prints or long code** inside ISRs because interrupts need to run **very quickly** and return control to the main program. Serial printing or long code takes time, which can **delay other interrupts**, **slow down the system**, or even **make it crash**. ISRs should be kept **short and fast**, only doing what's absolutely necessary.

6. What are the advantages of timer-based task scheduling?

Timer-based task scheduling lets tasks run automatically at **specific time intervals** without blocking the main program. This makes the system more **organized, efficient, and responsive**, since multiple tasks can run smoothly without using delay(). It helps in **multitasking**, **better timing control**, and keeps the program running **consistently and accurately**.

7. Describe I²C signals SDA and SCL.

In **I²C communication**, there are two main signals: **SDA** and **SCL**.
SDA (Serial Data Line) is used to **send and receive data** between devices, while **SCL (Serial Clock Line)** is used to **control the timing** of that data transfer. The **master device** controls the clock (SCL), and both master and slave use the **same two wires** to communicate, making I²C simple and efficient.

8. What is the difference between polling and interrupt-driven input?

Polling means the program keeps **checking repeatedly** if an input (like a button or sensor) has changed, which can **waste time and CPU power**.
Interrupt-driven input, on the other hand, lets the program **wait calmly**, and when something happens (like a button press), the **interrupt automatically alerts the CPU** to

handle it right away. So, polling is **continuous checking**, while interrupts are **event-based and more efficient**.

9. What is contact bounce, and why must it be handled?

Contact bounce happens when a button is pressed or released — instead of making one clean connection, it **quickly turns on and off many times** in a few milliseconds. This can make the microcontroller think the button was pressed **multiple times** instead of once. It must be handled (debounced) to make sure the system reads **only one clean and correct press**.

10. How does the LEDC peripheral improve PWM precision?

The **LEDC peripheral** improves PWM precision by using **hardware timers** and **high-resolution settings** to control the signal's frequency and duty cycle very accurately. It runs independently from the main CPU, so the PWM signal stays **smooth and stable**, even when the processor is busy doing other tasks.

11. How many hardware timers are available on the ESP32?

The **ESP32** has **four hardware timers** in total — **Timer 0, Timer 1, Timer 2, and Timer 3**. Each timer can be used independently for tasks like timing events, generating interrupts, or creating precise delays.

12. What is a timer prescaler, and why is it used?

A **timer prescaler** is a divider that **slows down the input clock** going to the timer. It's used to make the timer **count slower**, so you can measure **longer time intervals** without the timer overflowing too quickly. In simple words, it helps you adjust how **fast or slow** the timer runs.

13. Define duty cycle and frequency in PWM.

In **PWM (Pulse Width Modulation)**, the **duty cycle** means how long the signal stays **ON** compared to the total time of one cycle — a higher duty cycle gives more power or brightness.
The **frequency** means how **many times the signal repeats per second**, showing how fast the PWM turns ON and OFF.

14. How do you compute duty for a given brightness level?

To compute the **duty** for a given brightness level, we multiply the **maximum duty value** by the **brightness percentage**.

For example:
duty = max_duty × (brightness / 100)
So, if max_duty is 255 and brightness is 50%, then duty = 255 × 0.5 = **128**. This means the LED will glow at half brightness.

15. Contrast non-blocking vs. blocking timing.

Blocking timing (like using delay()) makes the program **pause completely** for a set time — nothing else can run during that wait.
Non-blocking timing uses methods like **millis() or timers** to keep track of time **without stopping** the rest of the program, allowing other tasks to run smoothly. So, non-blocking timing makes the system more **efficient and responsive**.

16. What resolution (bits) does LEDC support?

The **LEDC** on the ESP32 supports up to **20-bit resolution**.

17. Compare general-purpose hardware timers and LEDC (PWM) timers.

General-purpose hardware timers are flexible and can be used for many things like counting, delays, or triggering interrupts.
LEDC timers, on the other hand, are made specifically for **PWM (Pulse Width Modulation)** tasks — they control signal frequency and duty cycle for devices like LEDs or motors. So, general timers are **multi-use**, while LEDC timers are **optimized for smooth PWM control**.

18. What is the difference between Adafruit_SSD1306 and Adafruit_GFX?

Adafruit_SSD1306 is a library made to **control OLED displays** that use the SSD1306 driver chip.
Adafruit_GFX is a **graphics library** that provides basic drawing functions like lines, shapes, and text.
They work together to show visuals on the screen.

19. How can you optimize text rendering performance on an OLED?

We can optimize **text rendering on an OLED** by **reducing how often the display updates** — only refresh the parts that change instead of the whole screen. We can also use **smaller fonts**, **avoid clearing the screen repeatedly**, and **store text or images in memory (buffers)** to make drawing faster and smoother.

20. Give short specifications of your selected ESP32 board (NodeMCU-32S).

The **NodeMCU-32S (ESP32)** board has a **dual-core 32-bit processor** running up to **240 MHz**, with about **520 KB SRAM** and **4 MB Flash memory**. It supports **Wi-Fi and Bluetooth (BLE)**, has **30–36 GPIO pins**, and includes **ADC, DAC, PWM, I^2C, SPI, and UART** features, making it great for IoT and embedded projects.

Question 2 — Logical Questions

1. A 10 kHz signal has an ON time of 10 ms. What is the duty cycle? Justify with the formula.

Duty Cycle = (ON Time ÷ Total Time) × 100 = (10 ms ÷ 0.1 ms) × 100 = 10 000% → invalid (ON time too long).

The duty cycle is **10,000%**, which is **not possible** for a 10 kHz signal — this means the ON time value doesn't match the given frequency. For a valid 10 kHz signal, the ON time must be **less than or equal to 0.1 ms**.

2. How many hardware interrupts and timers can be used concurrently? Justify.

ESP32 can use **4 hardware timers** and up to **32 GPIO interrupts** at the same time because each works **independently** without conflict.

Each timer has its own registers and control, so they don't interfere with one another. Similarly, each GPIO pin with interrupt capability can trigger its **own ISR**, allowing multiple interrupts and timers to run **simultaneously without conflict**.

3. How many PWM-driven devices can run at distinct frequencies at the same time on ESP32? Explain constraints.

ESP32 can run **up to 8 PWM channels** at **4 different frequencies** at the same time because it has **4 LEDC timers**, and each timer controls the frequency for its group of channels. If two devices share the same timer, they must use the **same frequency**.

4. Compare a 30% duty cycle at 8-bit resolution and 1 kHz to a 30% duty cycle at 10-bit resolution (all else equal).

Both signals have the **same brightness level (30%)**, but the one with **10-bit resolution** gives **finer control** over the output because it has **1024 possible steps** instead of **256** in 8-bit. This means smoother and more accurate changes in brightness or speed, even though both run at **1 kHz** and **30% duty cycle**.

5. How many characters can be displayed on a 128×64 OLED at once with the minimum font size vs. the maximum font size? State assumptions.

Assuming the **minimum font size is 6×8 pixels** and the **maximum font size is 12×16 pixels**:

- For **6×8 font**:
  128 ÷ 6 = 21 characters per row
  64 ÷ 8 = 8 rows
  → **21 × 8 = 168 characters total**

- For **12×16 font**:
  128 ÷ 12 = 10 characters per row
  64 ÷ 16 = 4 rows
  → **10 × 4 = 40 characters total**

So, about **168 characters (small font)** vs **40 characters (large font)** can be shown at once.