ASSIGNMENT 01

Registration #	**23-NTU-CS-1011**
Name:	Ali Hassan
Course Name	Embedded iOT
Section	BSCS-5th-A
Department	Computer Science
Submit To	Sir Nasir
Date	19-10-2025

QUESTION N0 01

Q1

Q: Why is volatile used for variables shared with ISRs?

The 'volatile' keyword is used for variables that can be modified unexpectedly by an interrupt service routine (ISR). Without 'volatile', the compiler might optimize access to such variables, assuming their value doesn’t change. This can cause incorrect program behavior since ISR may update the variable outside the normal flow. Hence, 'volatile' ensures every read and write happens directly from memory.

Q2

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

Hardware-timer ISR debouncing uses interrupts and timers to filter out switch noise precisely without blocking program execution. It allows multitasking and accurate timing, making it ideal for real-time or complex embedded systems. Although more efficient, it’s harder to implement and requires understanding of interrupts and timers. On the other hand, delay()-based debouncing simply pauses the program for a set time after input detection. It’s much easier to code but inefficient, as it blocks other processes during the delay.

Q3

Q: What does IRAM_ATTR do, and why is it needed?

The **IRAM_ATTR** attribute tells the compiler to place a function in **Instruction RAM (IRAM)** instead of flash memory. This is needed because some functions, like **Interrupt Service Routines (ISRs)**, must run quickly and can't rely on slower flash memory access. During certain operations, flash memory might be temporarily unavailable, causing crashes if ISRs are stored there. By placing them in IRAM, they execute instantly and reliably. Hence, **IRAM_ATTR ensures fast, safe ISR execution.**

Q4

Q: Define LEDC channels, timers, and duty cycle.

LEDC channels are independent outputs that control different LEDs or PWM signals using the LED Control (LEDC) module.
LEDC timers define the frequency and resolution of the PWM signal shared by one or more channels.
The **duty cycle** represents how long the signal stays HIGH during each PWM cycle, controlling brightness or speed.
A higher duty cycle means more ON time and thus brighter LEDs or faster motors.
Together, channels, timers, and duty cycle define precise PWM control on microcontrollers like the ESP32.

Q5

Q: Why should you avoid Serial prints or long code paths inside ISRs?

You should avoid **Serial prints** or **long code paths** inside ISRs because they make the interrupt routine slow and inefficient.
ISRs should execute quickly to let the system handle other interrupts on time.
Serial communication is slow and can block execution, causing missed or delayed interrupts.
Long code paths increase latency and risk system instability or crashes.
In short, keep ISRs short, fast, and limited to essential tasks only.

Q6

Q: What are the advantages of timer-based task scheduling?

Timer-based task scheduling allows tasks to run at **precise, regular intervals** without blocking other code.
It improves **efficiency** by letting multiple tasks share CPU time smoothly.
Such scheduling enhances **timing accuracy** compared to using delays.
It enables **real-time responsiveness**, ideal for sensor reading, control loops, or communication tasks.
Overall, it leads to **cleaner, more reliable, and scalable** embedded system designs.

Q7

Q: Describe I²C signals SDA and SCL.

I²C uses two main signals: **SDA (Serial Data)** and **SCL (Serial Clock)**.
SDA carries the actual data between the master and slave devices.
SCL provides the clock pulses that synchronize data transmission.
Both lines are **open-drain** and require pull-up resistors to function correctly.
Together, they enable simple, two-wire communication between multiple devices on the same bus.

Q8

Q: What is the difference between polling and interrupt-driven input?

Polling continuously checks an input's state in a loop, wasting CPU time even when nothing changes.
It's simple to implement but **inefficient** for systems needing quick responses or multitasking.
Interrupt-driven input triggers a response only when an event occurs, freeing the CPU for other tasks.
It's **faster and more efficient**, especially for time-sensitive applications.
In short, polling waits actively, while interrupts react instantly when needed.

Q9

Q: What is contact bounce, and why must it be handled?

Contact bounce happens when mechanical switches or buttons rapidly make and break contact when pressed or released.
This causes multiple unwanted signals instead of a single clean transition.
If not handled, it can lead to **false triggers** or erratic behavior in digital circuits.
Debouncing ensures the system registers only one stable input per press.
Handling bounce improves **accuracy, reliability, and user experience** in electronic designs.

Q10

Q: How does the LEDC peripheral improve PWM precision?

The LEDC peripheral improves PWM precision by using high-resolution timers that divide the clock into fine steps.
It allows adjustable frequency and duty cycle resolution, giving smoother control over brightness or speed.
Multiple channels can share timers, ensuring consistent and synchronized output.
Hardware-based control reduces timing errors caused by software delays.
Overall, it delivers stable, accurate, and flicker-free PWM signals for precise applications.

Q11

Q: How many hardware timers are available on the ESP32?

The ESP32 has four hardware timers in total.
They're divided into two groups, with two timers per group (Timer0 and Timer1 in each).
Each timer is a 64-bit general-purpose counter that can run independently.
They support features like auto-reload, interrupts, and variable prescalers.
These timers are widely used for precise timing, task scheduling, and PWM generation.

Q12

Q: What is a timer prescaler, and why is it used?

A timer prescaler is a divider that reduces the input clock frequency before it reaches the timer.
It's used to slow down the timer's counting speed for longer timing intervals.
By adjusting the prescaler, you can fine-tune how fast the timer increments.
This helps match the timer's range to different timing needs without overflow.
In short, it provides flexibility and precision in timer-based operations.

Q13

Q: Define duty cycle and frequency in PWM.

In PWM (Pulse Width Modulation), the duty cycle is the percentage of time the signal stays HIGH during one complete cycle.
It determines the power delivered to a device — higher duty means more ON time.
The frequency is how many PWM cycles occur per second, measured in hertz (Hz).
It affects how smooth the output appears, like LED brightness or motor speed control.
Together, duty cycle and frequency define the behavior and precision of a PWM signal.

Q14

Q: How do you compute duty for a given brightness level?

To compute duty for a given brightness level, you map the brightness value to the PWM range.
The formula is: duty = (brightness / max_brightness) × max_duty.
For example, if brightness = 128, max_brightness = 255, and max_duty = 1023, then duty = (128/255)×1023 ≈ 514.
This ensures the LED brightness scales smoothly with the input level.
In short, it converts a desired brightness percentage into a PWM duty cycle value.

Q15

Q: Contrast non-blocking vs. blocking timing.

Blocking timing uses functions like delay() that stop all code execution until the time passes.
It's simple but prevents other tasks from running, reducing responsiveness.
Non-blocking timing uses techniques like millis() or timers to check elapsed time without halting the program.
This allows multitasking and smoother, real-time performance.
In short, blocking timing pauses everything, while non-blocking keeps the system active and efficient.

Q16

Q: What resolution (bits) does LEDC support?

The LEDC (LED Control) peripheral on the ESP32 supports resolutions from 1 to 20 bits.
This means the PWM duty cycle can have up to 2^{20} (1,048,576) discrete levels.
Higher resolution allows finer control over brightness or motor speed.
However, increasing resolution reduces the maximum possible PWM frequency.
So, developers balance resolution and frequency based on application needs.

Q17

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

General-purpose hardware timers are flexible counters used for various timing tasks like delays, interrupts, or event counting.
They can generate precise time intervals but require manual setup for PWM generation.
LEDC timers, on the other hand, are specialized for PWM control with built-in hardware for frequency and duty management.
They provide higher resolution and multiple synchronized channels for LED or motor control.
In short, general timers handle broad timing tasks, while LEDC timers focus on accurate, hardware-driven PWM output.

Q18

Q: What is the difference between Adafruit_SSD1306 and Adafruit_GFX?

Adafruit_SSD1306 is a display driver library specifically made for controlling SSD1306-based OLED screens.
It handles low-level communication like sending pixels and commands to the display hardware.
Adafruit_GFX is a graphics library that provides drawing functions like lines, shapes, and text.
It works as a base library that the SSD1306 driver builds upon.
In short, GFX handles graphics, while SSD1306 handles display hardware control.

Q19

Q: How can you optimize text rendering performance on an OLED?

To optimize text rendering performance on an OLED, **minimize screen updates** by redrawing only changed areas instead of the whole display.
Use **smaller fonts** or **bitmap fonts** to reduce data transfer.
Avoid frequent calls to display.display()—update only when necessary.
Store static text or graphics in **PROGMEM** to save RAM and speed up rendering.
Overall, focus on **partial updates and efficient data handling** for smoother OLED performance.

Q20

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

The NodeMCU-32S is based on the ESP32 dual-core 32-bit Xtensa LX6 processor running up to 240 MHz.
It includes 520 KB SRAM and 4 MB Flash memory.
Supports Wi-Fi (802.11 b/g/n) and Bluetooth 4.2 (BLE + Classic) connectivity.
Has 30 GPIO pins with support for ADC, DAC, PWM, UART, SPI, and I^2C.
Operates at 3.3 V logic, with micro-USB power and programming interface.

QUESTION NO 02

Q1

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

First, find the **period (T)** of the 10 kHz signal:

$$T = \frac{1}{f} = \frac{1}{10{,}000} = 0.0001\ \text{s} = 0.1\ \text{ms}$$

Given **ON time = 10 ms**, which is **100 times longer** than the full period.
Now calculate duty cycle:

$$\text{Duty Cycle} = \frac{\text{ON Time}}{\text{Period}} \times 100 = \frac{10\ \text{ms}}{0.1\ \text{ms}} \times 100 = 10{,}000\%$$

So, the **duty cycle = 10,000%**, which is invalid for a 10 kHz signal — the ON time exceeds the period, meaning the signal can't physically be 10 kHz.

Q2

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

The **ESP32** supports up to **32 hardware interrupts per core**, giving a total of **64 interrupts** across both cores.
These can be used **concurrently**, as each core manages its own interrupt controller.
It also has **4 general-purpose hardware timers** (two per group), all of which can run at the same time.
Each timer operates independently with its own counter and prescaler.
Thus, multiple interrupts and all 4 timers can function **simultaneously without conflict**, enabling true multitasking.

Q3

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

The **ESP32** has **4 LEDC timers**, and each timer can generate a unique **PWM frequency**.
Each timer can control up to **8 PWM channels**, totaling **16 channels** (8 per high- and low-speed mode).
However, all channels linked to the same timer **must share the same frequency** but can have **different duty cycles**.
So, you can run up to **4 distinct PWM frequencies** simultaneously on different timer groups.
The constraint is that the **number of unique frequencies is limited by the 4 timers**, not the total number of channels.

Q4

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

At **8-bit resolution**, the PWM range is **0–255**, so 30% duty ≈ **0.3 × 255 = 76** (ON count).
At **10-bit resolution**, the range is **0–1023**, so 30% duty ≈ **0.3 × 1023 = 307** (ON count).
Both signals have the **same average power output (30%)** and **same frequency (1 kHz)**.
However, the **10-bit PWM** provides **finer control** and smoother transitions between brightness or speed levels.
In short, higher resolution means **more precision**, even though the overall duty ratio is unchanged.

Q5

Q: 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.

Assume a 128×64 OLED and a common minimum bitmap font of **6×8 pixels** (5×7 glyph + 1px spacing): chars/line = ⌊128/6⌋ = 21, lines = ⌊64/8⌋ = 8 → **21×8 = 168 chars**.
For a large display font, assume **16×32 pixels** per character: chars/line = ⌊128/16⌋ = 8, lines = ⌊64/32⌋ = 2 → **8×2 = 16 chars**.
Formula used: chars = floor(width / char_width) × floor(height / char_height).
Results vary with actual font metrics (kern, spacing, proportional glyphs) — e.g., a 5×7 packed font could show more, a 32×48 font far fewer.
State your exact font (width×height) and I'll compute the precise counts.