National Textile University

Department of Computer Science

Name:	**M Talha Iftikhar**
Class:	**BSCS 5th B**
Registration No:	**23-NTU-CS-1075**
Lab Report:	**Assignment No: 1**
Course Name:	**IOT**
Submitted To:	**Sir Nasir Mahmood**
Submission Date:	**10/19/2025**

Embedded IOT Systems Assignment

Question 1 — Short Questions

Q1: Why is volatile used for variables shared with ISRs?

Because ISRs can change a variable anytime, marking it `volatile` tells the compiler not to optimize or cache its value. This way, the main code always reads the latest value updated by the ISR.

Q2: Compare hardware-timer ISR debouncing vs. delay()-based debouncing.

Timer ISR debouncing is more accurate since it doesn't block other code the timer runs in the background. `delay()`-based debouncing pauses the whole program, which can make it slower 2and unresponsive.

Q3: What does IRAM_ATTR do, and why is it needed?

`IRAM_ATTR` tells the compiler to store a function directly in internal RAM instead of flash. It's used for ISRs so they can run faster and still work even if flash memory is busy.

Q4: Define LEDC channels, timers, and duty cycle.

LEDC channels are outputs that send PWM signals. Timers control how fast the PWM runs (frequency). The duty cycle defines how long the signal stays ON during one cycle basically controls brightness or motor speed.

Q5: Why should you avoid Serial prints or long code paths inside ISRs?

Because ISRs should run quickly. Doing things like Serial prints or long code delays can make the system lag or miss other interrupts.

Q6: What are the advantages of timer-based task scheduling?

It runs tasks automatically at fixed intervals without blocking the main loop. This gives more accurate timing and better control over multiple tasks.

Q7: Describe I^2C signals SDA and SCL.

I^2C uses two lines: SDA for data and SCL for clock. They let the master and slave devices talk using just those two wires.

Q8: What is the difference between polling and interrupt-driven input?

Polling keeps checking a pin in a loop, wasting CPU time. Interrupt-driven input reacts instantly when something happens, without constantly checking.

Q9: What is contact bounce, and why must it be handled?

When a button is pressed, the contacts physically bounce and create many quick signals. Without debouncing, the microcontroller may think it's pressed multiple times.

Q10: How does the LEDC peripheral improve PWM precision?

LEDC uses dedicated hardware timers and high bit-resolution to produce very stable and smooth PWM signals, better than using software PWM.

Q11: How many hardware timers are available on the ESP32?

The ESP32 has four general-purpose hardware timers, split between its two timer groups.

Q12: What is a timer prescaler, and why is it used?

A prescaler divides the main clock speed before it goes into the timer. This helps create slower and more usable timer intervals.

Q13: Define duty cycle and frequency in PWM.

Frequency is how many times the signal repeats per second. Duty cycle is the percentage of time the signal stays high in each cycle.

Q14: How do you compute duty for a given brightness level?

You take the max duty value (like 255 or 1023) and multiply it by the brightness level (for example, 50% brightness = max × 0.5).

Q15: Contrast non-blocking vs. blocking timing.

Blocking timing (like using delay()) stops everything until time passes. Non-blocking timing checks elapsed time and keeps the program running in between.

Q16: What resolution (bits) does LEDC support?

LEDC supports resolutions up to 20 bits, though usually 8 to 16 bits are common depending on the frequency.

Q17: Compare general-purpose hardware timers and LEDC (PWM) timers.

General timers can be used for many purposes like counting or delays, while LEDC timers are made specifically for generating PWM signals with accurate control.

Q18: What is the difference between Adafruit_SSD1306 and Adafruit_GFX?

Adafruit_SSD1306 controls the OLED hardware itself, while Adafruit_GFX provides the drawing functions (like lines, text, shapes) used by many different displays.

Q19: How can you optimize text rendering performance on an OLED?

Use smaller fonts, avoid redrawing the full screen every time, and only update the parts that actually change.

Q20: Give short specifications of your selected ESP32 board (NodeMCU-32S).

- Dual-core 32-bit processor (up to 240 MHz)
- 520 KB SRAM, built-in WiFi & Bluetooth
- 30 GPIO pins
- Supports SPI, I^2C, UART, PWM, ADC, and DAC
- Micro-USB powered and programmable via Arduino IDE.

Question 2 — Long Questions

Q1: A 10 kHz signal has an ON time of 10 ms. What is the duty cycle? Justify with the formula.

formula.

The duty cycle basically shows how much time a signal stays ON in one

full cycle. The formula is:
Duty Cycle (%) = (ON Time / Total Period) × 100

For a 10 kHz signal, one period = 1 / 10,000 = 0.0001 seconds (which is 0.1 ms).
If the ON time is actually 10 ms, that would be way longer than one period, so it doesn't fit a 10 kHz signal — it would mean the signal is continuously ON.

If we take a realistic case, say ON time = 0.03 ms for that same 10 kHz frequency, then:
Duty Cycle = (0.03 / 0.1) × 100 = **30%**.
So, in general, the formula helps us understand how long the signal remains HIGH during one cycle compared to the total cycle time.

Q2: How many hardware interrupts and timers can be used concurrently? Justify.

On the ESP32, there are **4 general-purpose hardware timers** available, grouped into two timer groups (each having two). These timers can all run at the same time for different purposes, like creating delays, generating PWM signals, or triggering interrupts.

When it comes to interrupts, the ESP32 can handle **many at once**, because each GPIO pin can act as an interrupt source, and peripherals like UART, I^2C, and timers can also generate interrupts.
However, what really matters is how efficiently your ISR (Interrupt Service Routine) is written — if they're short and optimized, the ESP32 can handle many interrupts concurrently without timing issues.

Q3: How many PWM-driven devices can run at distinct frequencies at the same time on ESP32? Explain constraints.

The ESP32's LEDC module supports **16 PWM channels** and **4 independent timers**.
Each timer defines the base frequency and resolution of the PWM signal. Multiple channels can share one timer, but if you want each group of channels to have its own unique frequency, you can only have **up to 4 different frequencies at once**, because there are only 4 timers.

So, for example, you could control 16 LEDs or motors, but only 4 unique speed/brightness frequency groups. Each channel can still have its own duty cycle (so brightness or intensity can differ), even if the frequency is shared.

Q4: 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 cases mean the signal is ON for 30% of each cycle, but the **resolution** changes how finely we can adjust that ON time.

At 8-bit resolution, there are 256 total steps (0–255).
So a 30% duty cycle is around 0.3 × 255 = 77 steps ON.
At 10-bit resolution, there are 1024 steps (0–1023), giving about 0.3 × 1023 = 307 steps ON.

The higher the resolution, the smoother and more accurate the output. This matters for things like LED dimming — the 10-bit signal will

produce smoother brightness changes with less visible flicker, while 8-bit might look more “steppy.”

Q5: 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.

A 128×64 OLED screen has 128 pixels horizontally and 64 pixels vertically.
If we use the smallest Adafruit_GFX default font, each character is roughly **6×8 pixels** (5×7 pixels plus 1 pixel space).
That means 128 ÷ 6 ≈ 21 characters per line, and 64 ÷ 8 = 8 lines → around **21 × 8 = 168 characters** on screen at once.

If we switch to a much larger font, like 12×16 pixels per character, we get 128 ÷ 12 ≈ 10 characters per line, and 64 ÷ 16 = 4 lines → about **40 characters** total.

So the total number of visible characters depends on the font size. Smaller fonts display more text but are harder to read, while larger fonts reduce the amount of text but improve clarity.