Blinking an LED Using ESP32 Hardware Timer

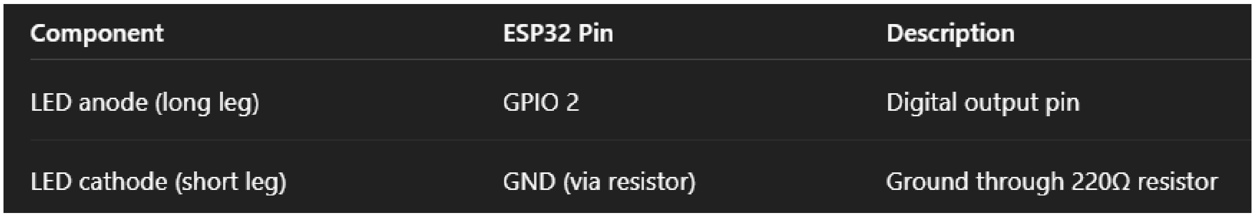
Objective:

Use ESP32's built-in hardware timer to toggle an LED on and off every 500 milliseconds (0.5 seconds) **without using `delay()` or the `loop()` function**.

Hardware Requirements:

- ESP32 Development Board
- 1x LED
- 1x 220Ω resistor
- Breadboard & jumper wires

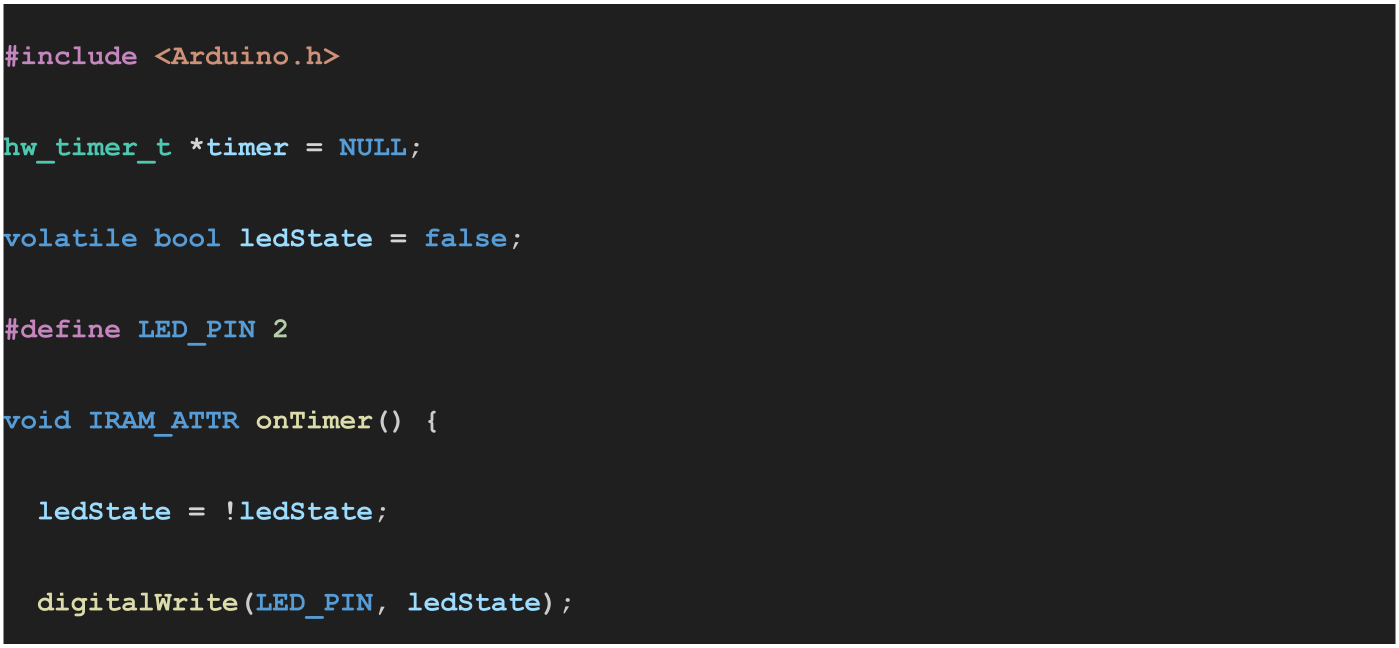
🔌 Pin Connections:



Component	ESP32 Pin	Description
LED anode (long leg)	GPIO 2	Digital output pin
LED cathode (short leg)	GND (via resistor)	Ground through 220Ω resistor

You can also use the **onboard LED** on many ESP32 boards, which is connected to **GPIO 2** by default.

🧾 Code Breakdown



```
#include <Arduino.h>

hw_timer_t *timer = NULL;

volatile bool ledState = false;

#define LED_PIN 2

void IRAM_ATTR onTimer() {

  ledState = !ledState;

  digitalWrite(LED_PIN, ledState);
}

void setup() {

  pinMode(LED_PIN, OUTPUT);

  timer = timerBegin(0, 80, true);

  timerAttachInterrupt(timer, &onTimer, true);

  timerAlarmWrite(timer, 500000, true);

  timerAlarmEnable(timer);

}

void loop() {

  // Nothing to do here

}
```

(Line-by-Line Explanation):

```
#include <Arduino.h>
```

- This includes the core Arduino functions. It's required for all sketches using the Arduino framework with ESP32.

```
hw_timer_t *timer = NULL;
```

- `hw_timer_t` is a hardware timer structure provided by ESP32.
- We declare a pointer `timer` to store the timer instance.

```
volatile bool ledState = false;
```

- `volatile` ensures the compiler knows `ledState` may change unexpectedly (due to interrupts).
- This variable tracks the **current LED state** (ON or OFF).

```
#define LED_PIN 2
```

- Define `LED_PIN` as GPIO 2.
- This is the pin connected to the LED.

```
void IRAM_ATTR onTimer() {
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState);
}
```

- This is the **interrupt service routine (ISR)**. It runs automatically every time the timer fires.
- `IRAM_ATTR` places this function in **internal RAM** (IRAM) for fast access.
- It toggles `ledState` from `true` to `false`, or vice versa.
- Then it writes the value to the `LED_PIN`, turning the LED **on or off**.

Setup Section:

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

- Set GPIO 2 as an **output pin** to control the LED.

```
  timer = timerBegin(0, 80, true);
```

- `timerBegin()` sets up **timer 0**.
  - First parameter `0`: Timer number (ESP32 has 4: 0–3).
  - `80` is the **prescaler**:
    - ESP32 runs at 80 MHz, so `80` prescaler makes it count at **1 MHz** (1 tick = 1 microsecond).
  - `true`: count up (instead of down).

```
  timerAttachInterrupt(timer, &onTimer, true);
```

- Link the interrupt service routine `onTimer()` with the timer.
- `true`: edge-triggered (fires on the timer edge).

```
  timerAlarmWrite(timer, 500000, true);
```

- Configure the timer to fire every **500,000 microseconds = 0.5 seconds**.
- `true`: repeat alarm (fires continuously every 0.5 seconds).

```
  timerAlarmEnable(timer);
```

- Enable the timer to start counting and triggering interrupts.

Loop Section:

```
void loop() {
  // Nothing to do here
}
```

- The LED is fully controlled by the **hardware timer and ISR**.
- No need to add logic here — the system runs **independently of `loop()`**.

Full Operation Summary:

1. The ESP32 initializes timer 0.
2. Every **0.5 seconds**, the timer triggers an **interrupt**.
3. The ISR toggles the LED by flipping `ledState` and writing it to GPIO 2.
4. LED blinks **ON and OFF every half-second**, controlled entirely by the timer.

Why Use Hardware Timers?

Unlike `delay()`, hardware timers:

- Don't block the CPU
- Can run **precisely and concurrently**
- Allow more responsive or real-time multitasking in complex projects

Advantages of This Method:

- **Accurate timing** (independent of code delays)
- **Non-blocking**, suitable for real-time applications
- Great for **motor control, PWM, sensor polling**, and blinking LEDs efficiently