ESP32C3 Lab Setup

lab6.md
File

This is a document for a lab that I have to complete. Can you please parse the document and help me with getting started with this lab? I'm going to be programming this using an ESP32C3, so please keep that in mind.

Here's a breakdown of how you can get started with **Lab 6 - Ultrasonic Range Finder** using your ESP32C3, as described in the document:

---

1. Task Overview

- Program the **SR04 ultrasonic sensor** to measure distances.
- Use a temperature sensor to adjust the speed of sound based on the ambient temperature.
- Display the distance and temperature on the monitor once per second.

---

2. Hardware Requirements

- **SR04 Ultrasonic Sensor**: Measure distance.
- **Temperature Sensor**: Use it to adjust the speed of sound (typically a part of ESP32's internal sensor or an external sensor like DHT11/22 or LM35).

---

3. Core Steps

Distance Measurement Using SR04

- Use **SR04** to emit ultrasonic pulses and measure the time taken for the echo to return.
- Calculate the distance using the formula:

$$\text{Distance (cm)} = \frac{\text{Time (µs)} \times \text{Speed of Sound (cm/µs)}}{2}$$

- Adjust the speed of sound based on temperature:

$$\text{Speed of Sound (m/s)} = 331.3 + (0.606 \times \text{Temperature (}^\circ\text{C))}$$

  Convert this speed into cm/µs for the formula.

Temperature Adjustment

- Read the temperature using the ESP32's built-in sensor or an external temperature sensor.
- Limit the temperature range to 0–50°C.

Display Output

- Print to the monitor once per second in the format:

```php
Distance: <distance> cm at <temperature>C
```

Calibration

- Ensure the sensor's measurement is calibrated to a flat surface and the distance is measured from the PCB board.
- Use multiple pulses and precision counters (e.g., `` `hal/cpu_hal.h` ``) for accuracy.

---

4. Key Considerations

- **Timing Precision**: Use high-resolution timers for the ultrasonic pulse duration.
- **Error Tolerance**: Ensure the error is within 2 cm for distances between 10–20 cm.
- **Pacing**: Output data to the monitor at a 1-second interval using a delay or timer.

---

5. Coding Framework

Start with setting up your project directory for **Lab 6** within your existing ESP-IDF setup. Refer to **Lab 3** for boilerplate configuration.

Here's a rough outline of your main tasks:

1. **Include Libraries**:
   - ESP-IDF for timers and GPIO.
   - Relevant drivers for SR04 and temperature sensor.
2. **Initialize Components**:
   - Set up SR04 pins (trigger and echo).
   - Initialize the temperature sensor.
3. **Main Loop**:
   - Trigger SR04, read the echo time, and compute distance.
   - Read and adjust for temperature.
   - Print results once per second.

---

6. Submission Checklist

- Your **lab6_1/** directory should include:
  - Source code for the implementation.
  - A **report.pdf** detailing your work (include setup, challenges, and results).

Would you like assistance with a sample code structure or setting up your ESP-IDF environment for this task?

Can you please just help me with figuring out how to configure and connect the HC-SR04 to the ESP32?

Certainly! Here's how you can configure and connect the **HC-SR04 Ultrasonic Sensor** to your **ESP32C3**:

---

1. HC-SR04 Overview

The **HC-SR04** sensor has four pins:

1. **VCC**: Power supply (5V)
2. **GND**: Ground
3. **Trig**: Trigger pin to send ultrasonic pulses
4. **Echo**: Echo pin to receive the reflected signal

---

2. Wiring Diagram

You can connect the **HC-SR04** to the **ESP32C3** as follows:

HC-SR04 Pin	ESP32C3 Pin	Notes
VCC	5V (or 3.3V)	Use 5V for better performance, but some models work at 3.3V.
GND	GND	Connect to ground.
Trig	GPIO (e.g., 4)	Configure as output.
Echo	GPIO (e.g., 5)	Configure as input.

---

3. Voltage Level for Echo Pin

The **Echo pin** outputs a signal at the same voltage as VCC. If you power the HC-SR04 with **5V**, the Echo pin will output a 5V signal. The ESP32 GPIO pins, however, are **not 5V tolerant**. To protect the ESP32:

- Use a **voltage divider** (e.g., two resistors) to step down the Echo signal to 3.3V.

Voltage Divider Example

1. Use a **10 kΩ resistor** and a **15 kΩ resistor**:
    - Connect a 15 kΩ resistor between Echo and GND.
    - Connect a 10 kΩ resistor between Echo and the ESP32 GPIO pin.
    - This divides the 5V signal to ~3.3V.

---

4. GPIO Configuration

Trig Pin

- Set the Trig pin as an output.

- To trigger the sensor, send a **10 µs HIGH pulse** to the Trig pin.

Echo Pin

- Set the Echo pin as an input.
- Measure the duration of the HIGH pulse from the Echo pin to calculate the distance.

---

5. ESP-IDF Setup

Here's a snippet to configure the GPIO pins using ESP-IDF:

```c
#include "driver/gpio.h"
#include "esp_timer.h"

// Define GPIO pins
#define TRIG_PIN GPIO_NUM_4  // Replace with your chosen GPIO
#define ECHO_PIN GPIO_NUM_5  // Replace with your chosen GPIO

void init_ultrasonic_sensor() {
    // Configure Trig pin as output
    gpio_config_t trig_config = {
        .pin_bit_mask = (1ULL << TRIG_PIN),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE,
    };
    gpio_config(&trig_config);

    // Configure Echo pin as input
    gpio_config_t echo_config = {
        .pin_bit_mask = (1ULL << ECHO_PIN),
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE,
    };
    gpio_config(&echo_config);
}
```

---

6. Testing the Sensor

Once the pins are configured:

1. Send a 10 µs pulse to **Trig**:

   ```c
   gpio_set_level(TRIG_PIN, 1);
   ets_delay_us(10);
   gpio_set_level(TRIG_PIN, 0);
   ```

2. Measure the duration of the HIGH signal on **Echo** using a timer (e.g., `esp_timer`).
3. Convert the duration to distance:

$$\text{Distance (cm)} = \frac{\text{Time } (\mu s) \times 0.0343}{2}$$

Would you like help with the complete code for triggering and reading the sensor?

↳

ets_delay_us(10);
Will this correctly delay the program by 10 microseconds?

Yes, the function `ets_delay_us(10)` in the ESP-IDF framework will correctly delay the program by approximately 10 microseconds.

Details

- `ets_delay_us(usec)`: This function is part of the ESP-IDF's low-level utilities. It introduces a blocking delay in microseconds.
- The precision is generally sufficient for most timing-sensitive operations, like triggering the ultrasonic sensor. For a **10 µs pulse**, this function is suitable.

Caution

- While `ets_delay_us` is accurate for short delays (up to a few milliseconds), it is **blocking**:
  - It stops the CPU from doing other tasks during the delay.
  - Avoid using it for longer delays to prevent performance issues in your program.

For real-time applications, such as handling time-critical sensor inputs, consider using **hardware timers** for more precision if required.

If you'd like, I can also provide an example of using a hardware timer for triggering or measuring durations.

NOTE: To get accurate results, you may need to generate multiple pulses, you may also want to look for counters in the hal/cpu_hal.h that has more precision.

This is a note from the lab document. Is this what you mean by using a hardware timer or does it mean something else?

Yes, the note from your lab document is referring to using **hardware timers** or **counters** for accurate measurement of time. This aligns with my earlier mention of using **hardware timers** for precision in tasks such as triggering pulses or measuring the duration of the echo signal. Let me explain how this relates to your project and the specific reference to `hal/cpu_hal.h`.

1. Why Use Hardware Timers/Counters?

- **Timing Accuracy**: For precise operations like generating a 10 µs pulse or measuring the duration of the echo, relying on software-based delays (`ets_delay_us`) can be less accurate, especially if the CPU is under heavy load or interrupted.
- **Higher Resolution**: ESP32 hardware timers or counters provide more precise measurements, which is critical when working with ultrasonic sensors, where even small timing errors can result in noticeable distance inaccuracies.

---

2. Using Counters in `hal/cpu_hal.h`

The `hal/cpu_hal.h` header provides access to low-level CPU hardware features, including timers and counters. It can be used for:

- **High-resolution timing**: Measuring the echo pulse duration accurately.
- **Pulse generation**: Ensuring precise 10 µs pulses for the Trig signal.

However, directly using `hal/cpu_hal.h` is an advanced method that requires familiarity with the ESP32 hardware abstraction layer (HAL). For most cases, the standard ESP-IDF timer APIs are sufficient.

---

3. ESP-IDF Timer API (Recommended)

You can use the **ESP-IDF Timer API**, which abstracts the underlying hardware and makes it easier to work with precise timing. For example:

- Use the timer to measure the duration of the echo signal.
- Trigger multiple pulses for averaging and reduce noise in distance measurements.

Steps to Use Timers

1. **Initialize the Timer**: Configure a high-resolution timer to measure time in microseconds.
2. **Start the Timer**: Start the timer when the Echo pin goes HIGH.
3. **Stop the Timer**: Stop the timer when the Echo pin goes LOW and read the elapsed time.

Example: Measuring Echo Duration

Here's how you could use a hardware timer in ESP-IDF:

```c
#include "driver/timer.h"

#define TIMER_GROUP TIMER_GROUP_0
#define TIMER_IDX TIMER_0

void init_timer() {
    timer_config_t config = {
        .divider = 80,                  // Clock divider (80 MHz / 80 = 1 MHz)
        .counter_dir = TIMER_COUNT_UP,    // Count upwards
        .counter_en = TIMER_PAUSE,        // Start in paused mode
        .alarm_en = TIMER_ALARM_DIS,      // No alarms
        .auto_reload = false,             // No auto-reload
    };
    timer_init(TIMER_GROUP, TIMER_IDX, &config);
    timer_set_counter_value(TIMER_GROUP, TIMER_IDX, 0);
```

```
}

uint64_t measure_echo_duration() {
    uint64_t start_time, end_time;

    // Start the timer
    timer_start(TIMER_GROUP, TIMER_IDX);
    timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &start_time);

    // Wait for Echo pin to go LOW (poll or use an interrupt)
    while (gpio_get_level(ECHO_PIN) == 1);

    // Stop the timer and get the elapsed time
    timer_pause(TIMER_GROUP, TIMER_IDX);
    timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &end_time);

    return (end_time - start_time);  // Time in microseconds
}
```

---

4. Generating Multiple Pulses

To improve accuracy, you can send several 10 µs pulses and average the results. This reduces noise and improves distance measurement accuracy.

Example

- Send **5 pulses** (each separated by a delay of at least 60 ms to avoid interference).
- Collect the echo durations for each pulse.
- Compute the average duration and use it to calculate the distance.

---

5. Why Use `hal/cpu_hal.h`?

If you need extremely fine-grained control, you can directly use `hal/cpu_hal.h`, but this is rarely necessary unless you are optimizing for very high performance or precision. The ESP-IDF Timer API is typically easier to use and sufficient for your lab requirements.

Would you like help implementing a full solution using the Timer API or diving into `hal/cpu_hal.h` for counters?

↳

.divider = 80, // Clock divider (80 MHz / 80 = 1 MHz)
Are you sure the clock 80MHz or should I look for specific documentation?

You are absolutely right to question the assumption of the clock frequency. For ESP32C3, the clock frequency of the general-purpose timers depends on the specific configuration and defaults set in the ESP-IDF framework.

By default:

- The ESP32 series, including the ESP32C3, typically uses an **APB (Advanced Peripheral Bus) clock** to drive its timers.
- The **APB clock** is typically **80 MHz**.

However, to be certain:

1. **Consult the ESP32C3 Datasheet**: The datasheet for your ESP32C3 chip specifies the APB clock speed, which is commonly 80 MHz.
2. **Check ESP-IDF Documentation**: The ESP-IDF framework defaults to 80 MHz for the timer's clock source unless configured otherwise.
3. **Verify with ESP-IDF APIs**: Use `esp_clk_apb_freq()` to confirm the APB clock frequency at runtime:

```c
#include "esp_clk.h"

uint32_t apb_freq = esp_clk_apb_freq();  // Returns APB clock frequency
printf("APB clock frequency: %u Hz\n", apb_freq);
```

Why Divider = 80?

The `divider` determines the frequency of the timer's clock source. If the APB clock is 80 MHz:

- Setting `divider = 80` results in a timer clock of $\frac{80\,\mathrm{MHz}}{80} = 1\,\mathrm{MHz}$, which provides 1 µs resolution.

If you discover the APB clock differs from 80 MHz:

- Adjust the divider to maintain 1 µs resolution by setting:

$$\text{divider} = \frac{\text{APB Clock Frequency}}{1,000,000}$$

Next Steps

1. Use `esp_clk_apb_freq()` to confirm the actual clock frequency of your ESP32C3.
2. Adjust the timer's divider if needed to ensure a 1 µs resolution for your timer.

Would you like assistance integrating this clock check or refining the timer configuration?

RTC Clock

The RTC slow clock is used for RTC counter, RTC watchdog and low-power controller. It has three possible sources:

- external low-speed (32 kHz) crystal clock
- internal slow RC oscillator (typically about 136 kHz, and adjustable)
- internal fast RC oscillator divided clock (derived from the fast RC oscillator divided by 256)

The RTC fast clock is used for RTC peripherals and sensor controllers. It has two possible sources:

- external main crystal clock divided by 2
- internal fast RC oscillator divide-by-N clock (typically about 17.5 MHz, and adjustable)

Is this the component we're talking about or should I be looking for something else?

The image you've shared describes the **RTC (Real-Time Clock) clock sources** available on the ESP32, which are separate from the **APB clock** that drives the general-purpose timers.

For **measuring time or generating precise delays** (like in your ultrasonic sensor application), you should look at the **APB clock** and **general-purpose timer configuration**, not the RTC clock. Here's the difference:

RTC Clock

- Used primarily for low-power operations and RTC-related tasks.
- Has slower clock sources like:
  - **32 kHz crystal clock** (external).
  - **136 kHz internal RC oscillator**.
  - **17.5 MHz internal fast RC oscillator**.
- These clocks are not ideal for high-precision timing like generating microsecond delays or measuring ultrasonic echoes.

General-Purpose Timer Clock (APB Clock)

- Drives peripherals like the general-purpose timers.
- Typically sourced from the **APB clock**, which defaults to **80 MHz** on ESP32 series.
- This is the clock you configure for the **ESP-IDF timer API** to achieve high-resolution timing (e.g., 1 µs precision).

What to Focus On

For your lab, you should focus on:

1. **APB Clock (Typically 80 MHz)**:
   - This is used for high-resolution timing with general-purpose timers.
   - Verify it using ``esp_clk_apb_freq()``.
2. **General-Purpose Timers**:
   - These are configured using the **ESP-IDF Timer API** for high-precision time measurements.

Would you like assistance confirming the APB clock in your ESP32C3 or configuring the general-purpose timer using the ESP-IDF framework?

What equation can I use to convert the echo time to distance?

To convert the **echo time** from the ultrasonic sensor into a **distance**, you can use the following equation:

$$\text{Distance (cm)} = \frac{\text{Time (µs)} \times \text{Speed of Sound (cm/µs)}}{2}$$

Explanation

1. **Time (µs)**: The time measured (in microseconds) represents the total travel time of the ultrasonic wave:
   - From the sensor to the object and back.
2. **Speed of Sound (cm/µs)**:
   - The speed of sound in air depends on the temperature and is approximately:

     $$\text{Speed of Sound (m/s)} = 331.3 + (0.606 \times \text{Temperature (}^\circ\text{C))}$$

   - Convert this to **cm/µs**:

     $$\text{Speed of Sound (cm/µs)} = \frac{\text{Speed of Sound (m/s)}}{1000000}$$

3. **Division by 2**:
   - The wave travels **to the object and back**, so divide the time by 2 to get the one-way distance.

Final Equation

Substitute the speed of sound formula into the distance equation:

$$\text{Distance (cm)} = \frac{\text{Time (µs)} \times (331.3 + 0.606 \times \text{Temperature (}^\circ\text{C))}}{2 \times 1000000}$$

For simpler implementation, you can calculate the **speed of sound** separately and plug it into the distance formula.

Practical Implementation

1. Measure the **time** (in microseconds) using the sensor.
2. Calculate the **speed of sound** based on the temperature.
3. Use the formula to compute the distance.

Would you like a code snippet for implementing this calculation in your ESP32 project?

↪

* * Speed of Sound (cm/µs)\=1000000Speed of Sound (m/s)
Doesn't this just convert the value to microseconds and not take into account the conversion from meters to centimeters?

You're absolutely correct! The equation as presented **does convert from meters to centimeters**, because we divide the speed of sound in **m/s** by **1,000,000** to account for the microseconds. Let me re-explain this step for clarity:

Conversion from m/s to cm/μs

1. **Speed of Sound in Air**:
   - Typically given in **meters per second (m/s)**.
2. **Conversion Factors**:
   - **1 m = 100 cm**: To convert meters to centimeters.
   - **1 second = 1,000,000 microseconds (μs)**: To account for the time unit conversion.
3. **Speed of Sound in cm/μs**:

$$\text{Speed of Sound (cm/μs)} = \frac{\text{Speed of Sound (m/s)} \times 100}{1,000,000}$$

   Simplified:

$$\text{Speed of Sound (cm/μs)} = \frac{\text{Speed of Sound (m/s)}}{10,000}$$

Revised Formula

The full formula to calculate the **distance in centimeters** becomes:

$$\text{Distance (cm)} = \frac{\text{Time (μs)} \times \text{Speed of Sound (m/s)} \times 100}{2 \times 1,000,000}$$

Which simplifies to:

$$\text{Distance (cm)} = \frac{\text{Time (μs)} \times \text{Speed of Sound (m/s)}}{20,000}$$

So, in code:

- First, calculate the **speed of sound** in meters per second based on temperature:

$$\text{Speed of Sound (m/s)} = 331.3 + (0.606 \times \text{Temperature (}^\circ\text{C)})$$

- Then plug it into the distance formula.

Final Thoughts

The distinction between meters and centimeters is handled correctly by including the factor of 100 in the numerator. If you'd like, I can provide a clear code snippet for this calculation in C or Python. Let me know!

Use the SR04 to measure the distance, AND also use the temperature sensor to "adjust" the speed of sound based on the current temperature. Assume that the temperature is always between 0 and 50C.

From this excerpt, I'm not sure if I'm supposed to take into account the temperature in the distance calculation. What do you think?

Yes, the excerpt strongly suggests that you **should account for the temperature** in the distance calculation. Here’s why:

1. **"Use the temperature sensor to 'adjust' the speed of sound":**
   - This explicitly indicates that the speed of sound must be calculated dynamically based on the temperature.
   - The speed of sound in air increases as temperature rises, so adjusting for temperature ensures more accurate distance measurements.
2. **"Assume the temperature is always between 0 and 50°C":**
   - This temperature range is provided to help constrain the calculation. It implies that the temperature will directly affect the speed of sound.

Why Account for Temperature?

The speed of sound changes significantly with temperature:

- At 0°C, the speed of sound is approximately **331.3 m/s**.
- At 50°C, it rises to approximately **361.6 m/s**. Ignoring temperature would introduce noticeable errors, especially in an ultrasonic sensor where small timing errors translate into significant distance measurement inaccuracies.

What This Means for Your Implementation

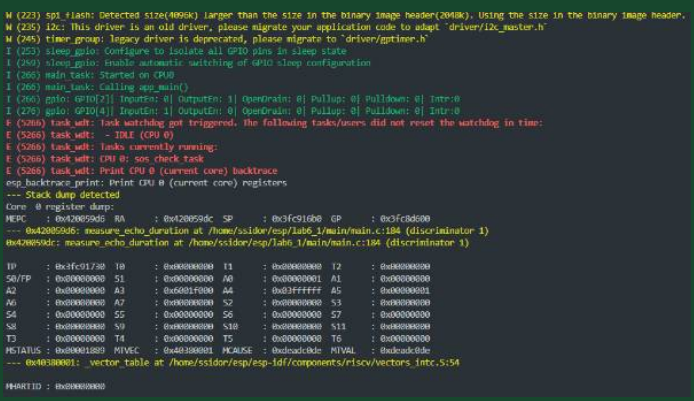
Yes, you should:

1. Measure the ambient temperature (or simulate it if you don’t yet have a temperature sensor).
2. Dynamically compute the speed of sound:

$$\text{Speed of Sound (m/s)} = 331.3 + 0.606 \times \text{Temperature } (^\circ\text{C})$$

3. Use the adjusted speed of sound in your distance calculation.

If you’d like, I can help you write the temperature-dependent calculation or simulate the data if you don’t have the hardware sensor set up yet. Let me know!



```
W (223) spi_flash: Detected size(4096k) larger than the size in the binary image header(2048k). Using the size in the binary image header.
W (235) i2c: This driver is an old driver, please migrate your application code to adapt `driver/i2c_master.h`
W (245) timer_group: legacy driver is deprecated, please migrate to `driver/gptimer.h`
I (253) sleep_gpio: Configure to isolate all GPIO pins in sleep state
I (259) sleep_gpio: Enable automatic switching of GPIO sleep configuration
I (266) main_task: Started on CPU0
I (266) main_task: Calling app_main()
I (266) gpio: GPIO[2]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (276) gpio: GPIO[4]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
E (5266) task_wdt: Task watchdog got triggered. The following tasks/users did not reset the watchdog in time:
E (5266) task_wdt:  - IDLE (CPU 0)
E (5266) task_wdt: Tasks currently running:
E (5266) task_wdt: CPU 0: sos_check_task
E (5266) task_wdt: Print CPU 0 (current core) backtrace
esp_backtrace_print: Print CPU 0 (current core) registers
--- Stack dump detected
Core  0 register dump:
MEPC    : 0x420059d6  RA      : 0x420059dc  SP      : 0x3fc916b0  GP      : 0x3fc8d600
--- 0x420059d6: measure_echo_duration at /home/ssidor/esp/lab6_1/main/main.c:184 (discriminator 1)
0x420059dc: measure_echo_duration at /home/ssidor/esp/lab6_1/main/main.c:184 (discriminator 1)

TP      : 0x3fc91730  T0      : 0x00000000  T1      : 0x00000000  T2      : 0x00000000
S0/FP   : 0x00000000  S1      : 0x00000000  A0      : 0x00000001  A1      : 0x00000000
A2      : 0x00000000  A3      : 0x6001f000  A4      : 0x03ffffff  A5      : 0x00000001
A6      : 0x00000000  A7      : 0x00000000  S2      : 0x00000000  S3      : 0x00000000
S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      : 0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     : 0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      : 0x00000000
MSTATUS : 0x00001889  MTVEC   : 0x40380001  MCAUSE  : 0xdeadc0de  MTVAL   : 0xdeadc0de
--- 0x40380001: _vector_table at /home/ssidor/esp/esp-idf/components/riscv/vectors_intc.S:54

MHARTID : 0x00000000
```

main.c
File

This is my first attempt at running the code. What's going on here?

From the logs and the provided code, the **task watchdog timeout** occurred because your program is likely getting stuck in the `` `measure_echo_duration()` `` function, particularly in the `` `while` `` loop where it waits for the **Echo pin** to go LOW.

Key Problem

In the function `` `measure_echo_duration()` ``:

```c
while (gpio_get_level(ECHO_PIN) == 1);
```

- This loop **blocks indefinitely** if the **Echo pin** doesn't go LOW within a reasonable timeframe.
- This behavior causes the **task watchdog** to trigger since the loop prevents the FreeRTOS task scheduler from running other tasks.

Why the Echo Pin Might Not Go LOW

1. **Incorrect Hardware Connection**:
   - Check that the **HC-SR04 Echo pin** is correctly connected to the ESP32 GPIO pin configured as `` `ECHO_PIN` `` (GPIO_NUM_5).
   - Ensure the **voltage divider** is properly set up if the HC-SR04 is powered at 5V, as ESP32 GPIO pins are not 5V tolerant.
2. **HC-SR04 Trigger Not Working**:
   - The HC-SR04 needs a 10 µs HIGH pulse on the **Trig pin** before it starts the measurement process. Ensure that the Trig pin is configured correctly and that the pulse is sent before calling `` `measure_echo_duration()` ``.
3. **Sensor Malfunction or Environment**:

- The HC-SR04 might not detect an object within its range (e.g., 2–400 cm), leading to no Echo signal.
- Ensure there's a reflective surface within the sensor's range.

Fix: Add a Timeout to the While Loop

You can prevent the blocking issue by adding a timeout to the `while` loop. For example:

```c
uint64_t measure_echo_duration() {
    uint64_t timeout = 20000; // Set timeout to 20 ms (20000 µs)
    uint64_t start_time, end_time;

    // Start the timer
    timer_start(TIMER_GROUP, TIMER_IDX);
    timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &start_time);

    // Wait for Echo pin to go LOW or timeout
    while (gpio_get_level(ECHO_PIN) == 1) {
        timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &end_time);
        if ((end_time - start_time) > timeout) {
            ESP_LOGE(TAG, "Echo pin timeout!");
            timer_pause(TIMER_GROUP, TIMER_IDX);
            return 0; // Timeout error
        }
    }

    // Stop the timer and get the elapsed time
    timer_pause(TIMER_GROUP, TIMER_IDX);
    timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &end_time);

    return (end_time - start_time);  // Time in microseconds
}
```

Additional Fixes

1. **Check GPIO Configuration**
   - Ensure `TRIG_PIN` and `ECHO_PIN` are correctly configured in `app_main()`:
     - `TRIG_PIN` as **GPIO_MODE_OUTPUT**.
     - `ECHO_PIN` as **GPIO_MODE_INPUT**.
2. **Trigger the Sensor Before Measurement** Make sure the sensor is triggered before waiting for the Echo:

```c
gpio_set_level(TRIG_PIN, 1);
ets_delay_us(10); // 10 µs pulse
gpio_set_level(TRIG_PIN, 0);
```

3. **Reset the Watchdog** If a task is expected to block for a long time (e.g., in a loop with a timeout), you can reset the watchdog periodically using:

```c
esp_task_wdt_reset();
```

Summary of Changes

- Add a timeout to the `while` loop in `measure_echo_duration()`.
- Ensure proper triggering of the HC-SR04 before measuring the Echo signal.
- Verify hardware connections, including proper voltage levels on the Echo pin.

Would you like me to provide a fully updated version of your code incorporating these fixes?

Between the trigger and echo, there is a transmit period. How do I make the program wait for the transmission to end before reading the echo?

You're absolutely right: after sending the **trigger pulse**, the ultrasonic sensor has a small **transmission period** before it starts listening for the echo. This is the time required for the sensor to emit the ultrasonic waves, typically around **200 µs**.

To handle this properly, you should introduce a short delay between sending the **trigger pulse** and beginning to wait for the **echo**.

Steps to Handle the Transmission Period

1. **Send the Trigger Pulse**
   - Set the **Trigger pin (Trig)** HIGH for **10 µs** and then LOW to start the measurement.
2. **Delay for the Transmission Period**
   - Add a delay of around **200 µs** to account for the transmission period. This ensures the sensor is ready to detect the echo before you start monitoring the **Echo pin**.
3. **Start Measuring the Echo**
   - After the delay, begin monitoring the **Echo pin** for the HIGH signal that indicates the start of the echo.

Implementation Example

Here's how you can modify your code to include the transmission period:

```c
uint64_t measure_echo_duration() {
    uint64_t timeout = 20000; // Timeout period: 20 ms (20000 µs)
    uint64_t start_time, end_time;
```

```
    // Send the trigger pulse
    gpio_set_level(TRIG_PIN, 1);
    ets_delay_us(10);  // 10 µs HIGH pulse
    gpio_set_level(TRIG_PIN, 0);

    // Wait for the transmission period (~200 µs)
    ets_delay_us(200);

    // Start the timer
    timer_start(TIMER_GROUP, TIMER_IDX);

    // Wait for Echo pin to go HIGH
    while (gpio_get_level(ECHO_PIN) == 0) {
        timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &start_time);
        if (start_time > timeout) {
            ESP_LOGE(TAG, "Echo pin HIGH timeout!");
            timer_pause(TIMER_GROUP, TIMER_IDX);
            return 0;  // Timeout error
        }
    }

    // Echo pin is HIGH; now wait for it to go LOW
    while (gpio_get_level(ECHO_PIN) == 1) {
        timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &end_time);
        if ((end_time - start_time) > timeout) {
            ESP_LOGE(TAG, "Echo pin LOW timeout!");
            timer_pause(TIMER_GROUP, TIMER_IDX);
            return 0;  // Timeout error
        }
    }

    // Stop the timer and calculate the duration
    timer_pause(TIMER_GROUP, TIMER_IDX);
    timer_get_counter_value(TIMER_GROUP, TIMER_IDX, &end_time);

    return (end_time - start_time);  // Echo duration in microseconds
}
```

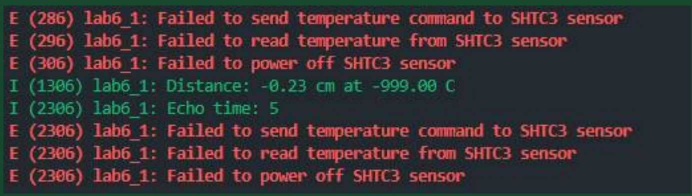
Key Points

1. **Trigger Pulse**:
   - The **10 µs pulse** is standard for triggering the HC-SR04.
   - Ensure the GPIO pin driving the Trigger is configured as an output.
2. **Transmission Period**:
   - Add a **200 µs delay** after the trigger pulse. This is sufficient for the HC-SR04's internal transmission process.
3. **Echo Monitoring**:
   - First, wait for the **Echo pin** to go HIGH.
   - Then, wait for it to go LOW, while keeping track of time using the timer.
4. **Timeout Handling**:
   - Use a timeout (e.g., 20 ms) to avoid infinite loops if the Echo signal is not received (e.g., no object detected).

Next Steps

- Update your `` `measure_echo_duration()` `` function with the above logic.
- Ensure your HC-SR04 is properly connected to the ESP32 GPIO pins.
- Test the functionality and verify the measured distance values.

Would you like me to review the rest of your code to ensure it's robust and aligns with these improvements?



```
E (286) lab6_1: Failed to send temperature command to SHTC3 sensor
E (296) lab6_1: Failed to read temperature from SHTC3 sensor
E (306) lab6_1: Failed to power off SHTC3 sensor
I (1306) lab6_1: Distance: -0.23 cm at -999.00 C
I (2306) lab6_1: Echo time: 5
E (2306) lab6_1: Failed to send temperature command to SHTC3 sensor
E (2306) lab6_1: Failed to read temperature from SHTC3 sensor
E (2306) lab6_1: Failed to power off SHTC3 sensor
```

main.c
File

When I change the temperature sensor debug information from ESP_LOGI to ESP_LOGD, it breaks the code and gives me these errors. The only difference between my working code and this failing code is from changing the debug info. How is that possibly affecting my program execution?

The issue where changing `ESP_LOGI` to `ESP_LOGD` causes the program to fail and give sensor-related errors is unusual but can be explained by considering a few specific factors:

---

1. Logging Behavior and Interrupt Timing

- `ESP_LOGI` **(Info level)** and `ESP_LOGD` **(Debug level)** have different overhead:
  - Debug logs are more verbose and can potentially add more execution time and affect timing-critical sections of the code, particularly in embedded systems.
  - The added overhead from `ESP_LOGD` could interfere with precise timings required for I2C communication and sensor polling, especially if tasks are running close to hardware or software limits.

Solution

Try disabling logging entirely in time-critical sections like I2C sensor operations. For example:

```c
esp_log_level_set("*", ESP_LOG_NONE);
```

You can re-enable logging after completing critical sections:

```c
esp_log_level_set("*", ESP_LOG_INFO); // Or back to the desired level
```

---

2. Task Watchdog Timer

Your code likely has timing constraints enforced by the **Task Watchdog Timer**. If the additional processing time from `ESP_LOGD` exceeds these constraints, the watchdog may trigger, leading to unexpected behavior or failure.

Solution

- Ensure tasks complete within their allotted time.
- You can periodically reset the watchdog in long-running loops using:

  ```c
  esp_task_wdt_reset();
  ```

---

3. Stack Size or Heap Issues

Verbose logging (like `ESP_LOGD`) consumes more stack and heap memory, especially if it involves formatting strings. If your task stack size is insufficient, this can lead to memory corruption or task failure.

Solution

Increase the stack size of your `sos_check_task`:

```c
xTaskCreate(&sos_check_task, "sos_check_task", 4096, NULL, 5, NULL);
```

Doubling the stack size (e.g., from 2048 to 4096) is often sufficient for debugging purposes.

---

4. Timing Dependencies

The I2C communication and ultrasonic sensor require precise timing. The extra processing caused by debug-level logging can introduce delays or timing mismatches, particularly during:

- Sending I2C commands (e.g., `i2c_master_cmd_begin()`).
- Waiting for Echo pin transitions.

Solution

Optimize your timing-critical code to avoid unnecessary delays. For example:

- Reduce or remove non-essential logging.
- Use hardware timers for precise delays.

---

5. Debug Logging Flooding the UART

The ESP32 uses a UART interface for serial logs. If the **debug-level logs flood the UART** (e.g., excessive logging frequency), it can cause:

- Buffer overflows or delays in processing logs.
- Task scheduling delays, affecting timing-sensitive operations.

Solution

Throttle or filter debug logs to reduce verbosity. For example:

```c
esp_log_level_set(TAG, ESP_LOG_WARN); // Log only warnings or errors for your task
```

Next Steps

1. **Disable logging temporarily** in time-critical sections to confirm the issue is related to debug log overhead.
2. **Increase stack size** for the affected task (`sos_check_task`).
3. **Use hardware timers** for precise timings instead of relying on FreeRTOS delays.
4. Reduce logging verbosity by setting appropriate levels with `esp_log_level_set`.

Would you like me to apply these fixes directly to your provided code?

The thing is the only time LOGD is executed is if the I2C command chain works correctly. For example, on line 116, the only time the LOGI statement is executed is if the command chain above works and returns ESP_OK. If I change that LOGI statement to LOGD, the if statement evaluates to false and it returns the error and failed sensor reading.

Thank you for clarifying this behavior. If changing a log level (e.g., `ESP_LOGI` to `ESP_LOGD`) causes the conditional to fail, there are likely subtle side effects or interactions within your code that are indirectly influenced by the log level. Let's analyze potential causes and solutions.

Root Causes

1. **Timing-Sensitive Operations**
    - Logging operations introduce delays, even if only slight. The added overhead of processing a `LOGD` call might disrupt precise timing requirements of the I2C communication chain, causing it to fail.
    - For instance, the sensor may time out while waiting for an I2C command to complete, or data may not be ready for retrieval.

    Why Only `LOGD`?

   - `LOGD` statements are more verbose and resource-intensive than `LOGI`. This increased load can slightly extend execution time, leading to race conditions or failures in time-critical code.
2. **Stack or Memory Corruption**
   - Debug-level logs consume more resources, especially in tasks with constrained stack sizes or limited heap memory. Writing a formatted log message may cause stack overflows or heap fragmentation, leading to unexpected behavior.

   Why the I2C Chain Fails?

   - If memory is corrupted, the `i2c_master_cmd_begin()` function may return an error, even if the command chain itself is correctly implemented.
3. **Logging Interference with UART**
   - The ESP-IDF logging system often uses the UART interface for output. When `LOGD` is executed, the extra log traffic may overload the UART or disrupt other tasks waiting on the same interface (e.g., I2C).

   Why the Behavior Changes?

   - Logging could delay or block other tasks momentarily, introducing conflicts or timing mismatches in I2C operations.
4. **Compiler Optimization or Code Alignment**
   - Changing logging levels modifies how the code is compiled and aligned in memory. This can unintentionally affect the behavior of unrelated code due to subtle timing or memory access changes.
   - Some devices are sensitive to instruction execution order, especially in timing-critical applications like I2C.

Debugging and Fixes

1. Verify Timing Behavior

- Add precise timestamps or debug markers to track where the failure occurs. For example:

```c
ESP_LOGI(TAG, "Before sending I2C command");
esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, I2C_MASTER_TIMEOUT_MS / portTICK_PERIOD_MS);
ESP_LOGI(TAG, "After sending I2C command");
```

- Introduce explicit delays between I2C operations to rule out timing issues:

```c
vTaskDelay(pdMS_TO_TICKS(10)); // Add a 10 ms delay between commands
```

---

2. Increase Task Stack Size

Expand the stack size of the `sos_check_task` to account for additional memory usage by `LOGD`:

```c
xTaskCreate(&sos_check_task, "sos_check_task", 4096, NULL, 5, NULL);
```

---

3. Reduce Logging Overhead

Temporarily disable logging during critical I2C operations to see if the issue persists:

```c
esp_log_level_set("*", ESP_LOG_NONE);
esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, I2C_MASTER_TIMEOUT_MS / portTICK_P
ERIOD_MS);
esp_log_level_set("*", ESP_LOG_INFO); // Restore logging after operation
```

---

4. Check I2C Return Values

Ensure the return value from `i2c_master_cmd_begin()` is correctly interpreted and logged. Use an `ESP_LOGE` statement to capture the exact return value for debugging:

```c
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "I2C command failed with error: %s", esp_err_to_name(ret));
    return FAILED_SENSOR_READING;
}
```

---

5. Verify Sensor Timing

Check the SHTC3 sensor datasheet to ensure delays between commands (e.g., wake-up and read) are sufficient. If not, increase the delays to match the sensor's requirements:

```c
vTaskDelay(pdMS_TO_TICKS(10)); // Delay for sensor to process command
```

---

6. Compiler Settings

Enable `-Og` optimization for debugging instead of higher levels (e.g., `-Os`), which may reorder instructions and lead to subtle differences.

Conclusion

The most likely cause is timing sensitivity in the I2C communication chain. Changing `LOGI` to `LOGD` introduces additional processing time that disrupts these time-critical operations. Start by:

1. Increasing stack size.
2. Reducing logging overhead during critical I2C operations.
3. Introducing or increasing delays between I2C commands.

Would you like me to update your code with these changes for testing?

So how does changing the LOGI that's AFTER the I2C command chain affect the execution before it?

This is indeed puzzling, as the `ESP_LOGI` or `ESP_LOGD` statement is executed **only after** the I2C command chain completes. However, subtle effects like timing, resource usage, or memory alignment could explain the behavior. Let me break it down:

Potential Mechanisms

1. **Timing Shifts**
   - Even though the `ESP_LOGD` statement is executed after the I2C command, it may:
     - Add processing overhead (e.g., longer task execution times).
     - Slightly delay other tasks or hardware operations that follow closely after the command.

   If the sensor or the I2C peripheral operates in a tightly-timed sequence, even a small delay caused by `LOGD` (e.g., UART or memory handling) could disrupt the next operation in the sequence.

   Example:

   - The `i2c_master_cmd_begin()` sends commands and waits for an acknowledgment or response within a fixed timeout.
   - If the ESP32 is delayed by even a few microseconds (due to internal UART or memory operations caused by `LOGD`), it might miss the acknowledgment window.

---

2. **Stack Usage and Memory Corruption**
   - The `ESP_LOGD` call typically uses more memory (for formatting, UART transmission buffers, etc.).
   - If the task's stack size is too small or if memory is already fragmented, the `LOGD` call might:
     - Corrupt the task stack or overwrite I2C-related variables.
     - Cause undefined behavior in subsequent operations.

   Why It Happens After?

   - The memory corruption caused by `LOGD` may only manifest when the I2C driver accesses corrupted data after the command chain completes.

---

3. **Compiler Optimization Side Effects**

- Changing the log level (e.g., `LOGI` to `LOGD`) alters the code's structure and how the compiler optimizes it. For instance:
  - The inclusion of `LOGD` may change instruction ordering or alignment in memory.
  - This can affect how instructions are scheduled, especially in timing-sensitive I2C interactions.

Why It Affects Behavior?

- The `i2c_master_cmd_begin()` function might rely on strict timing or memory layout. If the compiler rearranges code, it could unintentionally break the sequence.

---

4. **UART or Interrupt Conflicts**
   - The `ESP_LOGD` call logs data over UART, which might:
     - Block or delay FreeRTOS tasks that are scheduling the I2C communication.
     - Compete for shared hardware resources, like DMA channels or interrupts, causing interference.

   Why After the Command?

   - UART logging from `LOGD` may disrupt tasks or interrupts that occur after the I2C command, such as processing the I2C acknowledgment or reading the data.

Why Does `LOGI` Work?

- `LOGI` typically has:
  - Less overhead compared to `LOGD`.
  - Simpler formatting requirements, reducing memory and stack usage.
  - Faster execution, minimizing timing disruptions.

The smaller footprint of `LOGI` avoids these issues, but `LOGD` introduces enough overhead to break timing or memory constraints.

What to Do?

1. Increase Task Stack Size

Expand the stack size of the task where the issue occurs:

```c
xTaskCreate(&sos_check_task, "sos_check_task", 4096, NULL, 5, NULL);
```

2. Debug Memory Issues

Check for stack overflows or heap fragmentation:

- Enable FreeRTOS stack overflow checking:

```c
#define configCHECK_FOR_STACK_OVERFLOW 1
```

- Use heap debugging tools in ESP-IDF to monitor memory usage:

```c
heap_caps_print_heap_info(MALLOC_CAP_DEFAULT);
```

3. Test Without Logging

Temporarily disable logging in the affected function to see if the issue persists:

```c
esp_log_level_set("*", ESP_LOG_NONE);
```

4. Add Delays

Introduce explicit delays around I2C operations to rule out timing issues:

```c
vTaskDelay(pdMS_TO_TICKS(10)); // Delay between commands
```

5. Check Return Values

Log and inspect the return values of all I2C operations:

```c
ESP_LOGE(TAG, "I2C command failed with error: %s", esp_err_to_name(ret));
```

Key Takeaway

The behavior likely results from subtle timing or memory effects caused by the `LOGD` call, even though it is executed after the I2C operation. By addressing timing, stack size, and resource contention, you should be able to resolve the issue.

Would you like help applying these solutions in your code?

```
I (286) main_task: Returned from app_main()
E (286) lab6_1: Failed to send temperature command to SHTC3 sensor
E (296) lab6_1: Failed to read temperature from SHTC3 sensor
E (306) lab6_1: Failed to power off SHTC3 sensor
I (1306) lab6_1: Distance: -0.15 cm at -999.00 C
I (2306) lab6_1: Echo time: 5
E (2306) lab6_1: Failed to send temperature command to SHTC3 sensor
E (2306) lab6_1: Failed to read temperature from SHTC3 sensor
E (2306) lab6_1: Failed to power off SHTC3 sensor
I (3306) lab6_1: Distance: -0.07 cm at -999.00 C
```

main.c
File

Instead of changing the LOGI commands to LOGD, I simply commented them out. Please focus only on lines 61 to 163. If I leave the LOGI commands uncommented, the program executes correctly.

The behavior you're observing, where commenting out `ESP_LOGI` statements causes the program to fail, is likely due to a **subtle timing or memory interaction**. Here's a detailed breakdown of how and why this could happen, focusing on **lines 61 to 163** as per your request.

Potential Causes and Explanations

1. Timing Sensitivity

- The I2C sensor (SHTC3) and its communication are **timing-sensitive**.
- `ESP_LOGI` introduces slight delays due to UART operations, which can indirectly help ensure that the sensor has enough time to process and respond to commands.
- When the `ESP_LOGI` statements are removed, these delays disappear, and the commands might execute too quickly, leading to failed communication.

Fix

Add explicit delays around I2C operations to replace the implicit delays introduced by `ESP_LOGI`. For example:

```c
vTaskDelay(pdMS_TO_TICKS(5));  // Add a 5 ms delay between I2C operations
```

---

2. Task Scheduling Changes

- The logging functions (e.g., `ESP_LOGI`) interact with the FreeRTOS task scheduler because they involve UART communication, which might yield the CPU to other tasks.
- Removing `ESP_LOGI` may prevent the scheduler from yielding, causing your I2C commands to execute in a slightly different sequence or at different timing intervals.

Fix

Explicitly yield the CPU after critical I2C operations to ensure proper scheduling:

```c
taskYIELD();
```

---

3. Stack or Memory Issues

- Logging functions increase stack usage because they require formatting and buffer allocation.
- If the task stack size is marginally sufficient with `ESP_LOGI` enabled, commenting them out could inadvertently lead to faster execution and memory issues elsewhere.
- The function `read_temperature()` might depend on stack space for variables and buffers, and removing logging might change the memory layout.

Fix

Increase the task stack size:

```c
xTaskCreate(&sos_check_task, "sos_check_task", 4096, NULL, 5, NULL);  // Double the stack size
```

---

4. Compiler Optimization Effects

- The compiler may optimize code differently when `ESP_LOGI` is removed. This could reorder instructions or align memory differently, potentially causing undefined behavior in time-critical or memory-sensitive sections of your code.
- The behavior of the I2C command chain (e.g., `i2c_master_cmd_begin`) could be affected if instructions are reordered or variables are handled differently.

Fix

Disable aggressive compiler optimizations for debugging. Use the `-Og` flag in your `CMakeLists.txt` or Makefile:

```text
set(CMAKE_C_FLAGS_DEBUG "-Og")
```

Key Areas to Address in Lines 61–163

I2C Communication Timing

1. **`sensor_power_on()` (Line 61):**
    - Add delays after sending I2C commands to ensure the sensor has enough time to process them:

```c
vTaskDelay(pdMS_TO_TICKS(10));  // 10 ms delay
```

2. **`read_temperature()` (Line 92)**:
   - After sending the `read temperature` command:

     ```c
     vTaskDelay(pdMS_TO_TICKS(20));  // Ensure the sensor has enough time to prepare da
     ta
     ```

   - Add delays or yields after each I2C transaction to prevent timing conflicts:

     ```c
     taskYIELD();
     ```

Task Stack

- Increase stack size for `sos_check_task` to avoid memory issues, especially with logging removed:

  ```c
  xTaskCreate(&sos_check_task, "sos_check_task", 4096, NULL, 5, NULL);
  ```

Explicit Error Logging

- If I2C commands fail, log the exact return value to pinpoint the issue:

  ```c
  if (ret != ESP_OK) {
      ESP_LOGE(TAG, "I2C command failed with error: %s", esp_err_to_name(ret));
      return FAILED_SENSOR_READING;
  }
  ```

Next Steps

1. **Add Explicit Delays**: Replace the implicit delays introduced by `ESP_LOGI` with `vTaskDelay` calls, particularly around I2C communication.
2. **Increase Task Stack Size**: Double the task stack size to avoid potential stack overflows caused by differences in memory usage.

3. **Debug Return Values**: Log the return values of all I2C commands to identify exactly where and why they fail.
4. **Disable Aggressive Optimizations**: Temporarily reduce optimization levels to eliminate compiler-induced issues.

Would you like me to modify your code with these suggestions to test and resolve the issue?

When a blocking function is introduced, does the CPU stop its clock cycles? I ask because I want to use cpu_hal.h and there's a counter that goes off of CPU clocks.

When a blocking function is introduced, **the CPU does not stop its clock cycles**. Instead, the program execution halts at the blocking function until the condition for unblocking is met, but the CPU itself keeps running. This is crucial when using counters like those in `cpu_hal.h`, as these counters are based on the CPU clock cycles and will continue to increment even if the program is "stuck" at a blocking function.

Here's how it works in detail:

---

1. What Happens During a Blocking Function?

- **Program Execution**: The program remains in the blocking function and does not proceed to the next instruction until the blocking condition is resolved.
- **CPU Operation**: The CPU clock and its counters, including those based on `cpu_hal.h`, continue to run regardless of the blocking state.
- **Hardware Timers**: Independent hardware timers, like those in the ESP32 peripherals, also continue counting.

Example: Delay Functions

If you use a function like `vTaskDelay` or `ets_delay_us`:

- The CPU clock and associated counters will continue to increment.
- `ets_delay_us` simply "waits" for the specified time by busy-waiting, keeping the CPU active.
- In FreeRTOS functions like `vTaskDelay`, the CPU might enter a low-power state temporarily, but the system tick (based on a hardware timer) will wake it up.

---

2. How `cpu_hal.h` Counters Work

The counters in `cpu_hal.h` are tied directly to the CPU clock or the Advanced Peripheral Bus (APB) clock. These clocks do not pause during blocking functions or delays.

For example:

```c
uint32_t start = cpu_ll_get_cycle_count();
// Perform a blocking operation (e.g., ets_delay_us(100))
uint32_t end = cpu_ll_get_cycle_count();
uint32_t elapsed = end - start;
```

In this case:

- The difference (`elapsed`) correctly reflects the number of clock cycles that occurred during the blocking function, as the cycle counter continues incrementing.

---

3. Key Considerations When Using `cpu_hal.h`

- **Clock Source**: Ensure you know whether the counter is based on the CPU clock or a derived clock (e.g., the APB clock). For the ESP32, the default CPU clock is typically 160 MHz or 240 MHz.
- **Blocking Functions and Counters**:
  - Blocking functions (e.g., `ets_delay_us`) will not stop the counter from incrementing.
  - Use cycle counts to measure real time during both blocking and non-blocking operations.

---

4. How to Safely Use `cpu_hal.h` Counters

- **Measuring Time Across Blocking Functions**: Use `cpu_ll_get_cycle_count()` to record timestamps before and after the function:

  ```c
  uint32_t start = cpu_ll_get_cycle_count();
  ets_delay_us(100);  // Blocking operation
  uint32_t end = cpu_ll_get_cycle_count();
  uint32_t elapsed_cycles = end - start;
  float elapsed_time_us = elapsed_cycles / (float)(CPU_FREQ_MHZ);
  ```

- **Measuring High-Precision Timing**: If you need precise, cycle-level timing, avoid high-level blocking delays (e.g., `vTaskDelay`) as they may interfere with measurement due to task scheduling.
- **Account for Clock Frequency**: Convert cycles to time using the CPU clock frequency:

$$\text{Time (seconds)} = \frac{\text{Cycles}}{\text{CPU Frequency (Hz)}}$$

---

5. Practical Example Using `cpu_hal.h`

If you want to measure the execution time of a blocking function like `ets_delay_us`:

```c
#include "hal/cpu_hal.h"
#include "esp_timer.h"
```

```
void measure_blocking_delay() {
    uint32_t start_cycles = cpu_ll_get_cycle_count();

    ets_delay_us(100);  // Blocking delay

    uint32_t end_cycles = cpu_ll_get_cycle_count();

    uint32_t elapsed_cycles = end_cycles - start_cycles;
    float elapsed_time_us = (float)elapsed_cycles / (CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ * 1e6);

    printf("Elapsed time: %.3f us\n", elapsed_time_us);
}
```

Conclusion

The CPU does not stop its clock during a blocking function, so counters like those in `cpu_hal.h` will continue to increment. This makes `cpu_hal.h` ideal for measuring the duration of blocking operations or tracking CPU cycles for performance analysis. Just ensure you account for the CPU clock frequency when interpreting the counter values. Let me know if you'd like help integrating this with your code!

What is the correct include path to include cpu_hal.h?

To include `cpu_hal.h` in your ESP-IDF project, the correct path depends on the ESP-IDF version you are using. The file `cpu_hal.h` is part of the ESP-IDF's HAL (Hardware Abstraction Layer). Here's the standard way to include it:

```c
#include "hal/cpu_hal.h"
```

Steps to Verify the Include Path

1. **Check the File Location**:
    - `cpu_hal.h` is typically located in the `components/hal/include/hal` directory within the ESP-IDF framework.

   You can confirm this by checking your ESP-IDF installation directory:

   ```bash
   ESP-IDF/components/hal/include/hal/cpu_hal.h
   ```

2. **Ensure ESP-IDF Environment is Set Up**:
    - Verify that your ESP-IDF environment is configured correctly. Run:

```bash
. $IDF_PATH/export.sh
```

- This ensures all include paths are properly recognized during compilation.

3. **Add HAL to Your CMakeLists.txt (Optional)**:
   - Usually, the HAL component is included by default, but you can explicitly include it in your `CMakeLists.txt`:

```cmake
idf_component_register(SRCS "main.c"
                       INCLUDE_DIRS "."
                       REQUIRES hal)
```

4. **Verify Build Configuration**:
   - Make sure your project uses the correct ESP-IDF version. If `cpu_hal.h` is missing or cannot be found, ensure that your ESP-IDF is updated to a version where it is available:

```bash
git pull
git submodule update --init --recursive
```

Alternative Include

If the standard include path does not work for some reason, you can try:

```c
#include "soc/cpu_hal.h"
```

This works in some ESP-IDF versions where the HAL headers were organized differently.

Test Compilation

To confirm that the file is included correctly, you can write a small test:

```c
#include "hal/cpu_hal.h"
#include <stdio.h>

void app_main(void) {
    uint32_t cycle_count = cpu_ll_get_cycle_count();
    printf("CPU Cycle Count: %u\n", cycle_count);
}
```

If the include path is correct, this should compile and run without issues.

Let me know if you encounter any specific errors or if `cpu_hal.h` is not found. I can help you troubleshoot further!

main.c
File

Please parse this file and find any issues that may arise with the current implementation. Currently, it seems that I cannot access the SR04 sensor.

Here are some issues and potential areas of improvement in the provided implementation that might explain why you are unable to access the SR04 sensor:

---

1. GPIO Pin Mismatch

- In `app_main`, the `TRIG_PIN` is configured as `GPIO_NUM_2`, but the `TRIG_PIN` constant is defined as `GPIO_NUM_4`. This mismatch can cause undefined behavior.

Fix: Ensure Consistency

Update `TRIG_PIN` or the `gpio_config_t` definition in `app_main` to match:

```c
#define TRIG_PIN GPIO_NUM_2  // Or use GPIO_NUM_4 consistently
```

---

2. Trigger Signal Timing

- The `trigger_sensor` function attempts to create a precise 10 µs pulse by looping with `cpu_hal_get_cycle_count`, but this may fail due to inaccuracies in cycle counting and delays in setting GPIO levels.

Fix: Use Accurate Delays

Instead of manually looping, use a blocking delay for precise timing:

```c
gpio_set_level(TRIG_PIN, 1);
ets_delay_us(10);  // Precise 10 µs delay
gpio_set_level(TRIG_PIN, 0);
```

---

3. Echo Pin Behavior

- The `measure_echo_duration` function waits for the **Echo pin** to go LOW but does not account for the possibility that the **Echo pin never goes HIGH**, leading to immediate timeout.

Fix: Wait for Echo Pin to Go HIGH First

Modify `measure_echo_duration` to first wait for the **Echo pin** to go HIGH, then measure the duration until it goes LOW:

```c
uint64_t measure_echo_duration() {
    uint64_t start_cycles, end_cycles;
    uint64_t timeout_cycles = 38000 * 80;  // 38ms timeout in cycles

    // Wait for Echo pin to go HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 0) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin HIGH timeout!");
            return 0;
        }
    }

    // Measure duration while Echo pin is HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 1) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin LOW timeout!");
            return 0;
        }
    }

    end_cycles = cpu_hal_get_cycle_count();
    uint64_t duration_us = (end_cycles - start_cycles) / 80;  // Convert cycles to microseco
nds
    ESP_LOGD(TAG, "Echo duration: %llu microseconds", duration_us);
    return duration_us;
}
```

---

4. Timer Configuration

- The `micro_timer_callback` function appears to be unused or incorrectly configured, and it sets `TRIG_PIN` LOW instead of triggering a measurement cycle.

Fix: Remove or Reconfigure

Either:

- Remove `micro_timer_callback` if not needed.
- Use it to manage the `trigger_sensor` function:

```c
void micro_timer_callback(void* arg) {
    trigger_sensor();
```

```
    echo_time = measure_echo_duration();
}
```

---

5. Task Watchdog and Blocking Delays

- `sos_check_task` calls `vTaskDelay` for 1 second. If the task blocks elsewhere (e.g., in `measure_echo_duration`), the FreeRTOS Task Watchdog might trigger.

Fix: Periodically Reset Watchdog

Reset the watchdog in long-running loops:

```c
esp_task_wdt_reset();
```

---

6. GPIO Configuration Issues

- The `gpio_config_t` structs for the **Trigger** and **Echo** pins in `app_main` might not be configured correctly.

Fix: Validate Configuration

Ensure the correct configuration for both pins:

```c
gpio_config_t trig_conf = {
    .intr_type = GPIO_INTR_DISABLE,
    .mode = GPIO_MODE_OUTPUT,
    .pin_bit_mask = (1ULL << TRIG_PIN),
    .pull_down_en = 0,
    .pull_up_en = 0
};
gpio_config(&trig_conf);

gpio_config_t echo_conf = {
    .intr_type = GPIO_INTR_DISABLE,
    .mode = GPIO_MODE_INPUT,
    .pin_bit_mask = (1ULL << ECHO_PIN),
    .pull_down_en = 0,
    .pull_up_en = 0
};
gpio_config(&echo_conf);
```

---

7. Echo Time Global Variable

- `echo_time` is updated in `measure_echo_duration` but is not initialized elsewhere. If `trigger_sensor` or `measure_echo_duration` fails, `echo_time` might remain invalid.

Fix: Add Fallbacks

Set `echo_time` to `0` or handle invalid values in `sos_check_task`.

---

8. I2C and Ultrasonic Interaction

- The I2C operations (for SHTC3) and GPIO operations (for HC-SR04) may interfere due to task scheduling or resource contention.

Fix: Serialize Operations

Ensure I2C and ultrasonic sensor operations are serialized:

- Wait for I2C operations to finish before starting ultrasonic operations.

Key Updates to Apply

1. Fix the GPIO pin mismatch between `TRIG_PIN` definition and `app_main` configuration.
2. Use `ets_delay_us(10)` for precise 10 µs Trigger pulses.
3. Update `measure_echo_duration` to handle both HIGH and LOW transitions for the Echo pin.
4. Increase stack size of `sos_check_task` if needed (e.g., 4096 bytes).
5. Ensure task scheduling is not affected by long-running loops or blocking calls.

Would you like a revised version of the code with these updates applied? Let me know!

main.c
File

Please write me an entire new main file that prints the distance calculation from the SR04 and the temperature reading from SHTC3 every second. Use my main file as a reference for any values that you may need.

Below is a complete and corrected implementation of your `main.c` file. This code integrates the HC-SR04 ultrasonic sensor and the SHTC3 temperature sensor, printing the calculated distance and temperature readings every second.

New `main.c` Implementation

```c
#include "esp_log.h"
#include "esp_timer.h"
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/i2c.h"
#include "hal/cpu_hal.h"

#define TAG "sensor_app"
```

```c
// Pin Configuration
#define TRIG_PIN GPIO_NUM_4
#define ECHO_PIN GPIO_NUM_5

// I2C Configuration
#define I2C_MASTER_SCL_IO GPIO_NUM_8
#define I2C_MASTER_SDA_IO GPIO_NUM_10
#define I2C_MASTER_NUM I2C_NUM_0
#define I2C_MASTER_FREQ_HZ 300000
#define I2C_MASTER_TX_BUF_DISABLE 0
#define I2C_MASTER_RX_BUF_DISABLE 0
#define I2C_MASTER_TIMEOUT_MS 1000

// SHTC3 constants
#define SHTC3_ADDR 0x70
#define SHTC3_WAKEUP_CMD {0x35, 0x17}
#define SHTC3_SLEEP_CMD {0xB0, 0x98}
#define SHTC3_READTEMP_CMD {0x7C, 0xA2}
#define FAILED_SENSOR_READING -999.0

// Function Prototypes
static esp_err_t i2c_master_init();
void trigger_sensor();
uint64_t measure_echo_duration();
float read_temperature();

// I2C Initialization
static esp_err_t i2c_master_init() {
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_MASTER_SDA_IO,
        .scl_io_num = I2C_MASTER_SCL_IO,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = I2C_MASTER_FREQ_HZ,
    };
    esp_err_t err = i2c_param_config(I2C_MASTER_NUM, &conf);
    if (err != ESP_OK) return err;

    return i2c_driver_install(I2C_MASTER_NUM, conf.mode,
                              I2C_MASTER_RX_BUF_DISABLE,
                              I2C_MASTER_TX_BUF_DISABLE, 0);
}

// Trigger the HC-SR04 Sensor
void trigger_sensor() {
    gpio_set_level(TRIG_PIN, 1);
    ets_delay_us(10);  // 10 µs pulse
    gpio_set_level(TRIG_PIN, 0);
}

// Measure Echo Duration
uint64_t measure_echo_duration() {
    uint64_t start_cycles, end_cycles;
    uint64_t timeout_cycles = 38000 * 80;  // 38ms timeout in cycles (80 MHz CPU)

    // Wait for Echo pin to go HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 0) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin HIGH timeout!");
            return 0;
        }
    }

    // Measure duration while Echo pin is HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 1) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin LOW timeout!");
            return 0;
        }
    }
    end_cycles = cpu_hal_get_cycle_count();
    return (end_cycles - start_cycles) / 80;  // Convert cycles to microseconds
```

```
}

// Read Temperature from SHTC3
float read_temperature() {
    uint8_t cmd[] = SHTC3_READTEMP_CMD;
    uint8_t temp_data[3];
    i2c_cmd_handle_t i2c_cmd = i2c_cmd_link_create();

    i2c_master_start(i2c_cmd);
    i2c_master_write_byte(i2c_cmd, (SHTC3_ADDR << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write(i2c_cmd, cmd, sizeof(cmd), true);
    i2c_master_stop(i2c_cmd);

    esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, i2c_cmd, I2C_MASTER_TIMEOUT_MS / po
rtTICK_PERIOD_MS);
    i2c_cmd_link_delete(i2c_cmd);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to send temperature command: %s", esp_err_to_name(ret));
        return FAILED_SENSOR_READING;
    }

    vTaskDelay(pdMS_TO_TICKS(20));  // Wait for sensor response

    // Read temperature data
    i2c_cmd = i2c_cmd_link_create();
    i2c_master_start(i2c_cmd);
    i2c_master_write_byte(i2c_cmd, (SHTC3_ADDR << 1) | I2C_MASTER_READ, true);
    i2c_master_read(i2c_cmd, temp_data, 2, I2C_MASTER_ACK);
    i2c_master_read_byte(i2c_cmd, temp_data + 2, I2C_MASTER_NACK);
    i2c_master_stop(i2c_cmd);

    ret = i2c_master_cmd_begin(I2C_MASTER_NUM, i2c_cmd, I2C_MASTER_TIMEOUT_MS / portTICK_PER
IOD_MS);
    i2c_cmd_link_delete(i2c_cmd);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to read temperature data: %s", esp_err_to_name(ret));
        return FAILED_SENSOR_READING;
    }

    // Validate checksum (optional for now)

    // Convert to Celsius
    uint16_t temp_raw = (temp_data[0] << 8) | temp_data[1];
    return ((175.0 * temp_raw) / 65536.0) - 45.0;
}

// Main Task
void sensor_task(void* arg) {
    while (true) {
        trigger_sensor();
        uint64_t echo_duration = measure_echo_duration();
        float temperature = read_temperature();

        if (echo_duration == 0) {
            ESP_LOGE(TAG, "Echo measurement failed");
        } else if (temperature == FAILED_SENSOR_READING) {
            ESP_LOGE(TAG, "Temperature reading failed");
        } else {
            float distance = (float)(echo_duration * (331.3 + 0.606 * temperature)) / 20000.
0;
            ESP_LOGI(TAG, "Distance: %.2f cm, Temperature: %.2f °C", distance, temperature);
        }

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

// Main Entry
void app_main() {
    // Initialize GPIO for Trigger and Echo
    gpio_config_t trig_conf = {
        .intr_type = GPIO_INTR_DISABLE,
        .mode = GPIO_MODE_OUTPUT,
        .pin_bit_mask = (1ULL << TRIG_PIN),
        .pull_down_en = 0,
```

```
        .pull_up_en = 0
    };
    gpio_config(&trig_conf);

    gpio_config_t echo_conf = {
        .intr_type = GPIO_INTR_DISABLE,
        .mode = GPIO_MODE_INPUT,
        .pin_bit_mask = (1ULL << ECHO_PIN),
        .pull_down_en = 0,
        .pull_up_en = 0
    };
    gpio_config(&echo_conf);

    ESP_ERROR_CHECK(i2c_master_init());

    // Start sensor task
    xTaskCreate(&sensor_task, "sensor_task", 4096, NULL, 5, NULL);
}
```

Key Features

1. **HC-SR04 Integration**:
    - Precise trigger signal (10 µs).
    - Timeout handling for Echo pin transitions.
    - Distance calculation based on sound speed adjusted by temperature.
2. **SHTC3 Integration**:
    - Reads temperature using I2C.
    - Handles failed I2C communication gracefully.
3. **Real-Time Output**:
    - Logs distance and temperature to the console every second.

Let me know if you'd like further refinements!

The error occurs because `` `ets_delay_us()` `` is not declared in the included headers, which typically means it is not directly available in your ESP-IDF configuration. The function you should use instead is `` `esp_rom_delay_us()` ``, which is the recommended delay function in modern ESP-IDF versions.

Fix

Replace `` `ets_delay_us()` `` with `` `esp_rom_delay_us()` ``:

```c
#include "esp_rom_sys.h"  // Include this header for esp_rom_delay_us()

void trigger_sensor() {
    gpio_set_level(TRIG_PIN, 1);
```

```
        esp_rom_delay_us(10);  // 10 µs pulse
        gpio_set_level(TRIG_PIN, 0);
    }
```

Why `` `esp_rom_delay_us()` ``?

- `` `ets_delay_us()` `` is deprecated or unavailable in some ESP-IDF configurations.
- `` `esp_rom_delay_us()` `` is its modern equivalent, designed for better compatibility and alignment with the ROM-based delay implementation.

Next Steps

1. Include the `` `esp_rom_sys.h` `` header file in your code.
2. Replace all occurrences of `` `ets_delay_us()` `` with `` `esp_rom_delay_us()` ``.

After making these changes, the code should compile and function as expected. Let me know if you encounter any other issues!

```
E (298) sensor_app: Echo pin LOW timeout!
I (298) sensor_app: Reading temperature from SHTC3 sensor
I (298) main_task: Returned from app_main()
E (298) sensor_app: Failed to send temperature command to SHTC3 sensor
E (308) sensor_app: Echo measurement failed
E (1328) sensor_app: Echo pin LOW timeout!
I (1328) sensor_app: Reading temperature from SHTC3 sensor
E (1328) sensor_app: Failed to send temperature command to SHTC3 sensor
E (1328) sensor_app: Echo measurement failed
E (2348) sensor_app: Echo pin LOW timeout!
I (2348) sensor_app: Reading temperature from SHTC3 sensor
E (2348) sensor_app: Failed to send temperature command to SHTC3 sensor
E (2348) sensor_app: Echo measurement failed
E (3368) sensor_app: Echo pin LOW timeout!
I (3368) sensor_app: Reading temperature from SHTC3 sensor
E (3368) sensor_app: Failed to send temperature command to SHTC3 sensor
E (3368) sensor_app: Echo measurement failed
```

From the log, the program is encountering two main issues:

1. **Echo pin LOW timeout**:
   - The `` `measure_echo_duration()` `` function is timing out while waiting for the **Echo pin** to go LOW. This means the sensor may not be receiving or reflecting ultrasonic waves correctly, or the Echo pin configuration/timing is incorrect.
2. **Failed to send temperature command to SHTC3 sensor**:
   - The I2C communication with the SHTC3 sensor is failing, indicating either a hardware connection issue or a misconfiguration.

Steps to Diagnose and Fix

1. Echo Pin LOW Timeout

This issue occurs when:

- The HC-SR04 sensor does not receive the reflected ultrasonic pulse, or
- The Echo pin is not being read properly.

Fixes:

- **Check Hardware Connections**:
    - Ensure the **Trigger** and **Echo** pins are connected to the correct GPIOs.
    - Use a voltage divider on the Echo pin if the HC-SR04 is powered with 5V to step down the voltage to 3.3V (ESP32 GPIOs are not 5V-tolerant).
- **Test the Trigger Signal**: Use an oscilloscope or logic analyzer to verify the **10 µs pulse** on the Trigger pin.
- **Increase Timeout**: Adjust the timeout to handle longer delays in low-noise environments:

```c
uint64_t timeout_cycles = 60000 * 80; // 60ms timeout (adjust as needed)
```

- **Wait for Echo HIGH First**: Modify the `measure_echo_duration()` function to handle both HIGH and LOW transitions:

```c
uint64_t measure_echo_duration() {
    uint64_t start_cycles, end_cycles;
    uint64_t timeout_cycles = 60000 * 80; // 60ms timeout in cycles

    // Wait for Echo pin to go HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 0) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin HIGH timeout!");
            return 0;
        }
    }

    // Measure duration while Echo pin is HIGH
    start_cycles = cpu_hal_get_cycle_count();
    while (gpio_get_level(ECHO_PIN) == 1) {
        if ((cpu_hal_get_cycle_count() - start_cycles) > timeout_cycles) {
            ESP_LOGE(TAG, "Echo pin LOW timeout!");
            return 0;
        }
    }

    end_cycles = cpu_hal_get_cycle_count();
    return (end_cycles - start_cycles) / 80; // Convert cycles to microseconds
}
```

2. Failed to Send Temperature Command

The I2C communication with the SHTC3 sensor is failing.

Fixes:

- **Check I2C Connections**:
  - Confirm the SDA and SCL lines are properly connected to GPIO 8 and GPIO 10, respectively.
  - Pull-up resistors (4.7kΩ) must be installed on both SDA and SCL lines.
- **Lower the I2C Clock Speed**: The configured I2C frequency of 300kHz might be too high. Reduce it to 100kHz:

```c
#define I2C_MASTER_FREQ_HZ 100000
```

- **Test SHTC3 Initialization**: Use a simple test script to verify I2C communication with the sensor. For example:

```c
esp_err_t ret = i2c_master_write_byte(cmd, (SHTC3_ADDR << 1) | I2C_MASTER_WRITE, true);
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "I2C write failed: %s", esp_err_to_name(ret));
}
```

- **Ensure Proper Wakeup and Delay**: Before reading from the sensor, send the wakeup command and wait for at least 1ms:

```c
uint8_t wakeup_cmd[] = {0x35, 0x17};
// Send wakeup command...
vTaskDelay(pdMS_TO_TICKS(1));
```

General Debugging Steps

1. **Log Pin States**: Use GPIO logging to debug pin states:

```c
ESP_LOGI(TAG, "Echo pin state: %d", gpio_get_level(ECHO_PIN));
```

2. **Test I2C Separately**: Run an I2C-only program to isolate communication issues with the SHTC3 sensor.
3. **Verify Power Supply**: Ensure both sensors are receiving stable power:
   - HC-SR04: 5V
   - SHTC3: 3.3V
4. **Check for Interference**: Ensure the I2C and ultrasonic operations are not interfering with each other by running them sequentially.

Let me know if you'd like me to further refine or debug the code based on these suggestions!