Embedded Hardware Designer Challenge: Sensor Interface & Debug

Project Overview

This report documents the design, implementation, and debugging process of a temperature-sensing system using an **ESP32 microcontroller** interfaced with an **LM75A temperature sensor** over **I2C**. The collected temperature data is transmitted via **UART**.

Task 1: Design the System

Objectives

- Design a microcontroller-based system to read temperature using an I2C sensor.
- Implement UART communication to transmit sensor data.
- Debug unexpected UART behavior and intermittent sensor read failures.

Part 1: Schematic Design

Sketch a schematic (tool-based or hand-drawn) for:

1. An STM32 (or similar) microcontroller interfacing with an I2C temperature sensor (e.g., TMP102).
2. LED blinking at 1 Hz during data sampling.
3. UART interface (e.g., USB-to-serial) for communication at 115200 baud.

System Design

2.1 Hardware Schematic

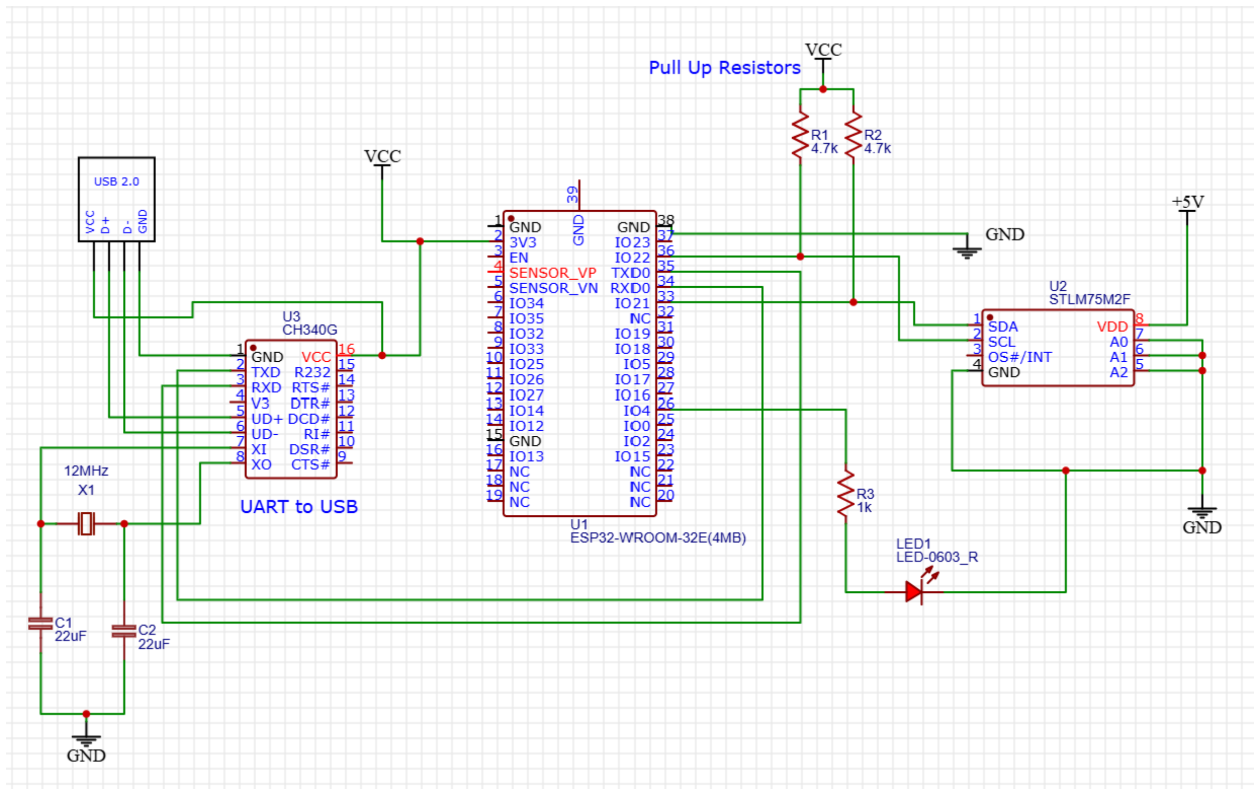

Pull Up Resistors
VCC
R1 4.7k
R2 4.7k
VCC
USB 2.0
VCC D+ D- GND
U3 CH340G
GND VCC TXD R232 RXD RTS# V3 DTR# UD+ DCD# UD- RI# XI DSR# XO CTS#
UART to USB
12MHz X1
C1 22uF
C2 22uF
GND
U1 ESP32-WROOM-32E(4MB)
GND 3V3 EN SENSOR_VP SENSOR_VN IO34 IO35 IO32 IO33 IO25 IO26 IO27 IO14 IO12 GND IO13 NC NC NC
GND IO23 IO22 TXD0 RXD0 IO21 NC IO19 IO18 IO5 IO17 IO16 IO4 IO0 IO2 IO15 NC NC NC
GND
+5V
U2 STLM75M2F
SDA SCL OS#/INT GND VDD A0 A1 A2
R3 1k
LED1 LED-0603_R
GND

Figure 1: Schematic

2.2 Hardware Components

Components	Description
ESP32	Microcontroller for processing & communication
LED	Blinks at 1 Hz to indicate data sampling

Pull-up Resistors (4.7kΩ)	For I2C communication
USB-to-UART Adapter	Used for serial communication
LM75A	I2C Temperature Sensor

Bonus: Note 1–2 key signal integrity or manufacturability considerations

1. Signal Integrity Issue (**Pull up resistors, Fixed the issue**)
2. Manufacturability issue (**A0, A1, A2 addresses pins grounded, Fixed the issue**)

Signal Integrity Issue (Pull-up Resistors)

In I2C communication, the SDA (data) and SCL (clock) lines are open-drain, meaning the devices on the bus can only pull the lines low. To ensure proper communication, pull-up resistors are essential to bring the lines back to a high logic level when no device is pulling them low.

What Went Wrong?

The problem occurred due to incorrect or missing pull-up resistors on the SDA/SCL lines. If the resistors are too weak or absent, the data lines cannot return to a high voltage quickly enough. This leads to signal integrity issues, resulting in unreliable communication and corrupted data. In this case, it caused gibberish to appear on the UART output from the LM75 temperature sensor.

How I Fixed It:

To restore signal integrity, I added 4.7kΩ pull-up resistors between the SDA/SCL lines and VCC (3.3V). This ensures that the I2C lines properly transition between high and low voltages, providing clean signals and enabling reliable communication.

Manufacturability Issue (A0, A1, A2 Address Pins Grounded)

The LM75 temperature sensor uses the A0, A1, and A2 address pins to set the I2C address. These pins should be connected to either logic high (VCC) or logic low (GND) to configure the address.

What Went Wrong?

The A0, A1, and A2 address pins were incorrectly grounded. When all three pins are grounded, the I2C address of the LM75 sensor is set to 0x00. This could lead to addressing conflicts with other devices on the I2C bus or, in some cases, prevent proper communication altogether. This mistake is a common issue that can result in unrecognized data or data corruption.

How I Fixed It:

I ensured that the A0, A1, and A2 address pins were either properly grounded or pulled high (VCC) according to the required address configuration. By doing so, I assigned the LM75 sensor a unique I2C address, ensuring stable communication and correct data transmission.

Part 2. Firmware Logic

1. Write pseudocode or C snippets to:
2. Initialize I2C and UART.
3. Read sensor data periodically and blink the LED.
4. Transmit temperature readings.

Pseudocode

1. Define Constants:

- ➔ LM75A I2C Address: 0x48
- ➔ Temperature Register: 0x00
- ➔ LED Pin: 4
- ➔ Wakeup Time: 10 seconds

2. Initialize Variables:

- ➔ Blink Enabled Flag: false
- ➔ Temperature in Fahrenheit: 0.0

3. Setup Function:

- ➔ Initialize serial communication.
- ➔ Initialize I2C communication.
- ➔ Set LED pin as output.
- ➔ Scan for I2C devices and report any found.

4. Main Loop:

1. **Read Temperature from LM75A sensor.**
    - · **If successful:**
        - ➔ Convert temperature to Fahrenheit.

    - ➔ Print temperature in Celsius and Fahrenheit.
  - · **If unsuccessful:**
    - ➔ Print error message.
    - ➔ Disable LED blinking.
2. **Determine Action Based on Temperature:**
   - **If temperature < 16°C:**
     - ➔ Print "Entering Deep Sleep Mode."
     - ➔ Call the deep sleep function.
   - **If temperature ≥ 16°C and < 17°C:**
     - ➔ Print "Entering Light Sleep Mode."
     - ➔ Call a light sleep function.
   - **If temperature ≥ 17°C:**
     - ➔ Print "Active Mode."
     - ➔ Enable LED blinking.
2. **LED Control:**
   - **If blinking is enabled:**
     - ➔ Toggle LED state every second.
   - **Else:**
     - ➔ Ensure the LED is off.

5. Read Temperature Function:

- Begin I2C transmission to LM75A address.
- Request temperature data.
- **If data received:**
  - ➔ Combine bytes to form raw temperature.
  - ➔ Convert raw data to Celsius.
  - ➔ Return temperature.
- **Else:**
  - ➔ Return error indicator.

6. Deep Sleep Function:

- ➔ Disable LED blinking.
- ➔ Set sleep mode to deep sleep.
- ➔ Sleep for wakeup time (10 seconds).

7. Light Sleep Function:

- ➔ Disable LED blinking.
- ➔ Set sleep mode to light sleep.
- ➔ Sleep for wakeup time (10 seconds).

Repository: The following is a full code repository.

https://github.com/umar-sharif-dev/Interfacing-LM75-sensor-with-ESP32

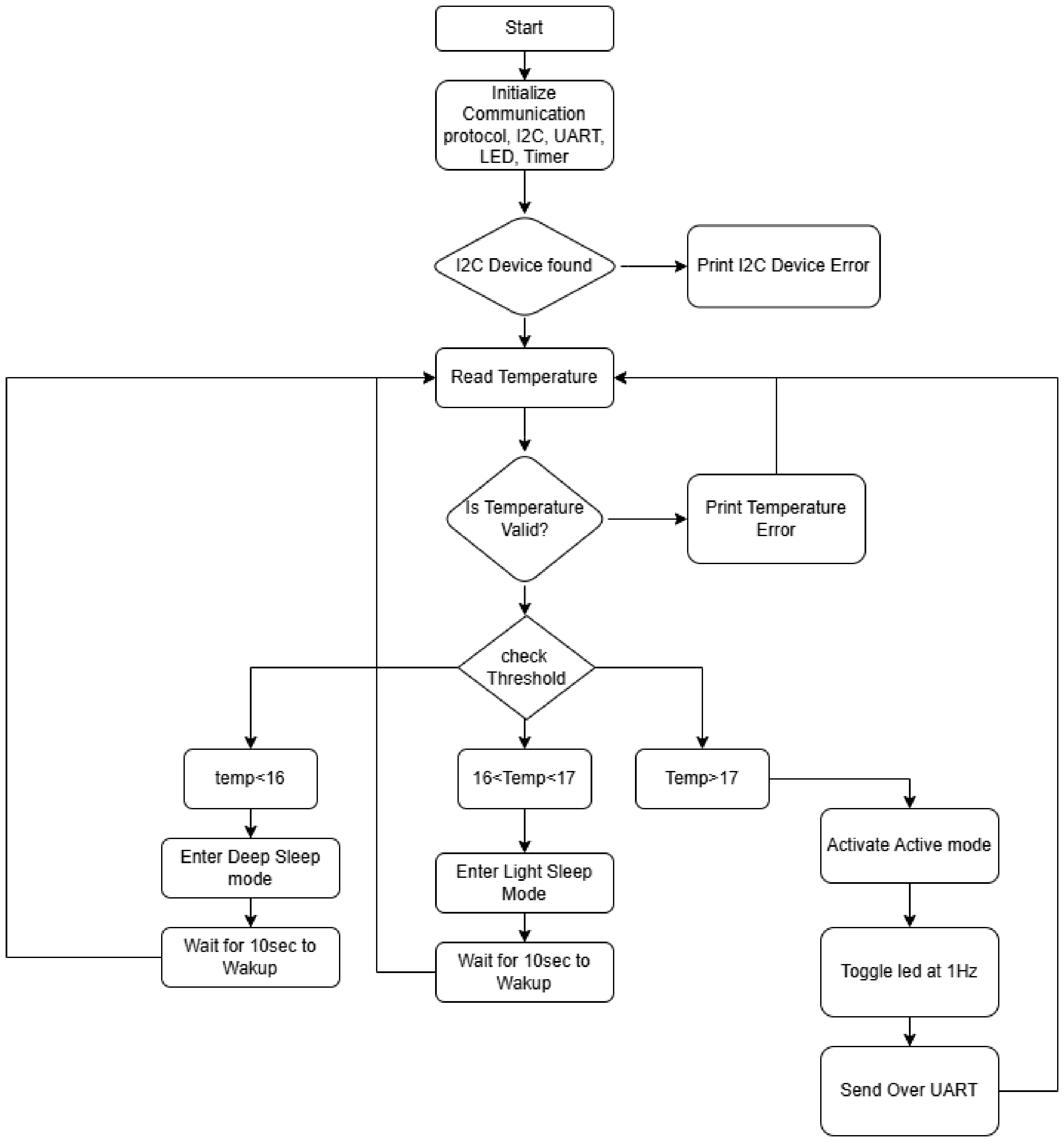
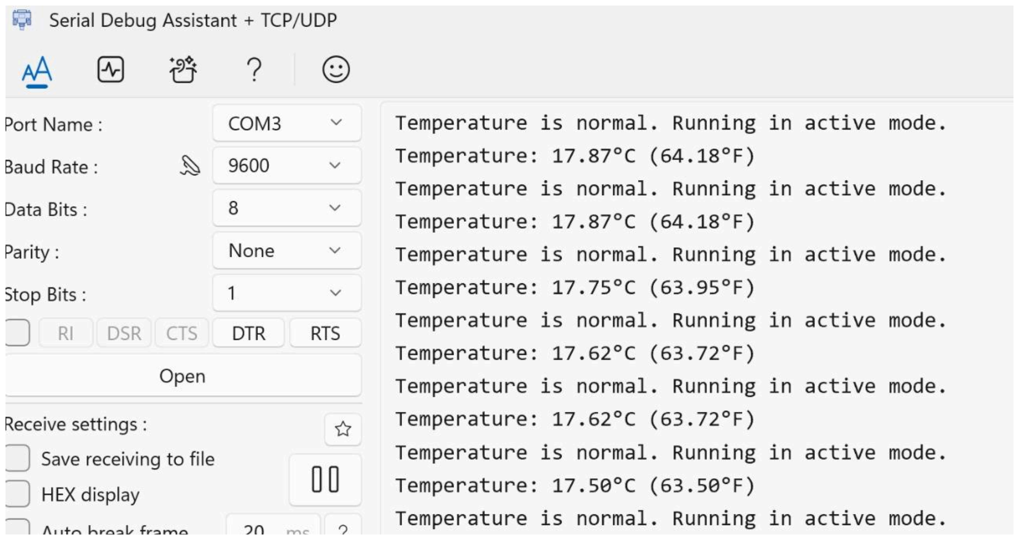


Figure 8 : System Architecture Flowchart

Bonus: Briefly explain power-saving strategies for a battery-operated device.

The power consumption strategy for the battery-operated device is based on real-time temperature readings, with current consumption monitored via an INA219 current sensor. The device operates in different power modes depending on the temperature threshold values: it enters **Active Mode** when the temperature exceeds 17°C, **Light Sleep Mode** when the temperature is between 16°C and 17°C, and **Deep Sleep Mode** when the temperature falls below 16°C.

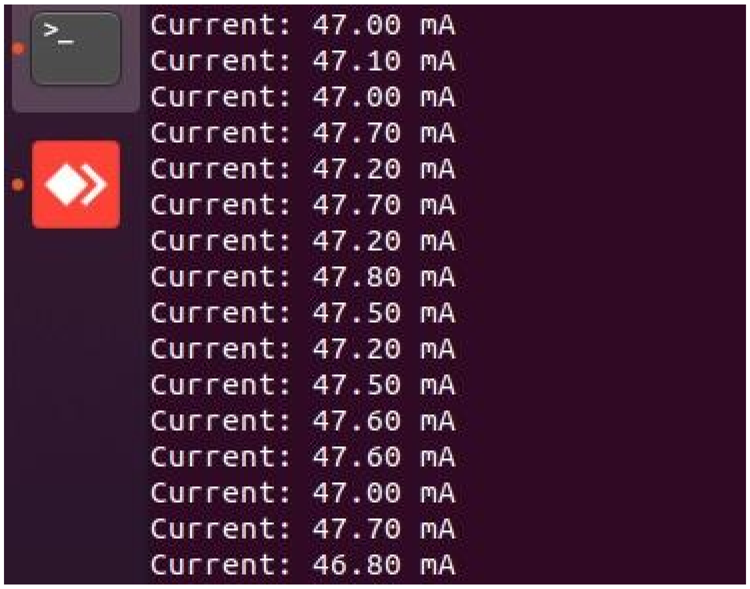
1. **Active Mode:**
   - The device is operating at full power, performing tasks, and communicating over I2C, which is reflected in the relatively high current consumption. For example, in the image below, the current fluctuates around 47.00 mA.
   - Active mode typically involves the sensor being fully operational, continuously monitoring and transmitting data (e.g., temperature and current), which causes the higher power draw.
   - **Power consumption strategy:** In this mode, the device requires a lot of energy because it's active and processing data continuously.



Port Name : COM3
Baud Rate : 9600
Data Bits : 8
Parity : None
Stop Bits : 1

```
Temperature is normal. Running in active mode.
Temperature: 17.87°C (64.18°F)
Temperature is normal. Running in active mode.
Temperature: 17.87°C (64.18°F)
Temperature is normal. Running in active mode.
Temperature: 17.75°C (63.95°F)
Temperature is normal. Running in active mode.
Temperature: 17.62°C (63.72°F)
Temperature is normal. Running in active mode.
Temperature: 17.62°C (63.72°F)
Temperature is normal. Running in active mode.
Temperature: 17.50°C (63.50°F)
Temperature is normal. Running in active mode.
```

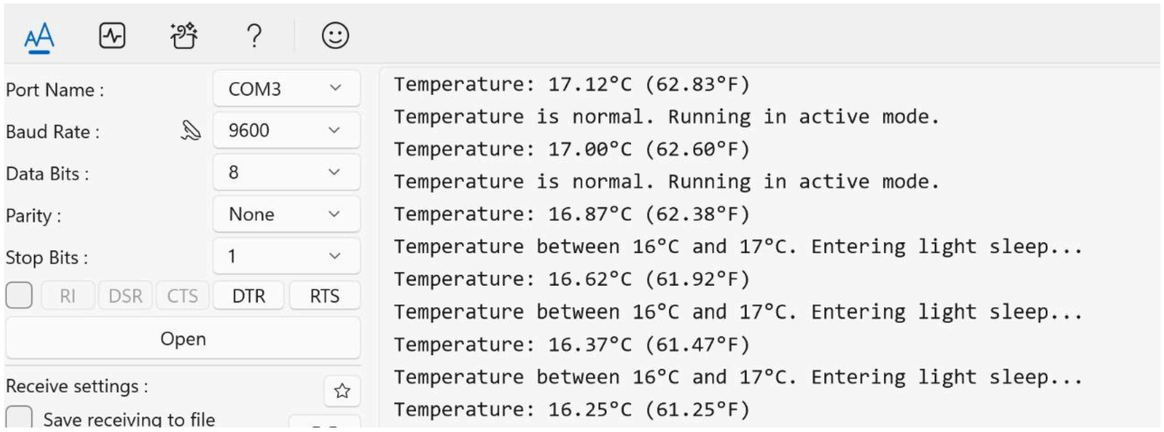
Figure 2: Active Mode at temperature above 17°C



```
Current: 47.00 mA
Current: 47.10 mA
Current: 47.00 mA
Current: 47.70 mA
Current: 47.20 mA
Current: 47.70 mA
Current: 47.20 mA
Current: 47.80 mA
Current: 47.50 mA
Current: 47.20 mA
Current: 47.50 mA
Current: 47.60 mA
Current: 47.60 mA
Current: 47.00 mA
Current: 47.70 mA
Current: 46.80 mA
```

Figure 3: Current Measurement in Active mode

2. **Light Sleep Mode:**
   - The device consumes significantly less power in light sleep mode, as shown in the image below, where the current drops to around 12.50 mA or slightly lower.
   - In light sleep mode, most of the device's functions are paused, but the microcontroller remains active enough to maintain some minimal operations, such as monitoring basic tasks or staying ready for quick wake-ups.
   - **Power consumption strategy:** Light sleep mode is used to conserve power by reducing the activity of the device but still allowing it to wake up quickly and respond to external stimuli.



Port Name : COM3
Baud Rate : 9600
Data Bits : 8
Parity : None
Stop Bits : 1

RI DSR CTS DTR RTS

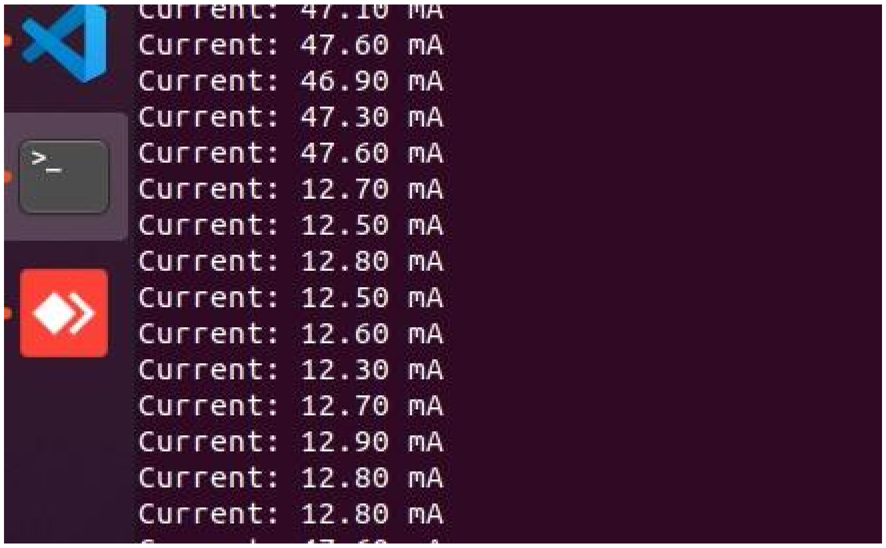
Open

Receive settings :

Save receiving to file

```
Temperature: 17.12°C (62.83°F)
Temperature is normal. Running in active mode.
Temperature: 17.00°C (62.60°F)
Temperature is normal. Running in active mode.
Temperature: 16.87°C (62.38°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.62°C (61.92°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.37°C (61.47°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.25°C (61.25°F)
```

Figure 4: Light Sleep Mode activated between 16°C - 17°C

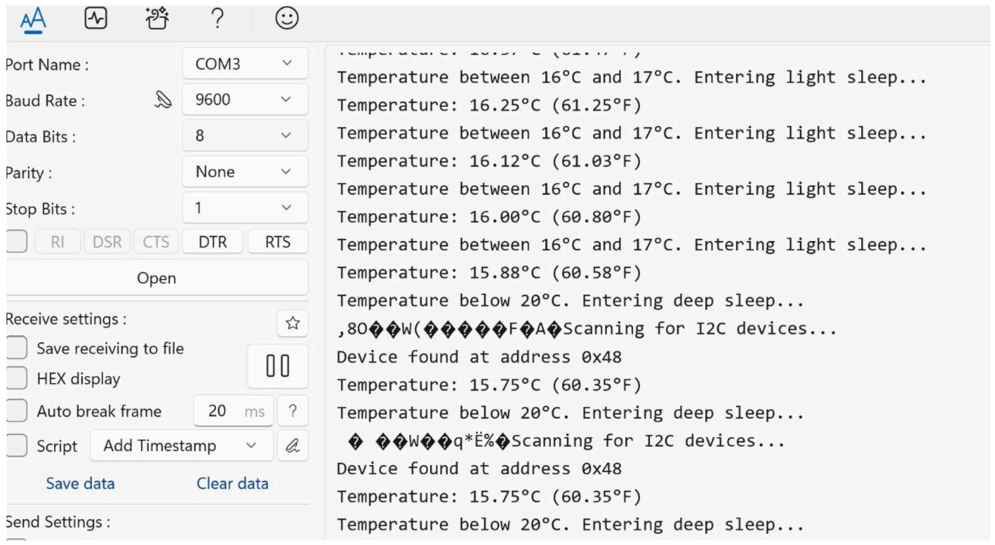


```
Current: 47.10 mA
Current: 47.60 mA
Current: 46.90 mA
Current: 47.30 mA
Current: 47.60 mA
Current: 12.70 mA
Current: 12.50 mA
Current: 12.80 mA
Current: 12.50 mA
Current: 12.60 mA
Current: 12.30 mA
Current: 12.70 mA
Current: 12.90 mA
Current: 12.80 mA
Current: 12.80 mA
```

Figure 5: Current Measurement in Light Sleep Mode

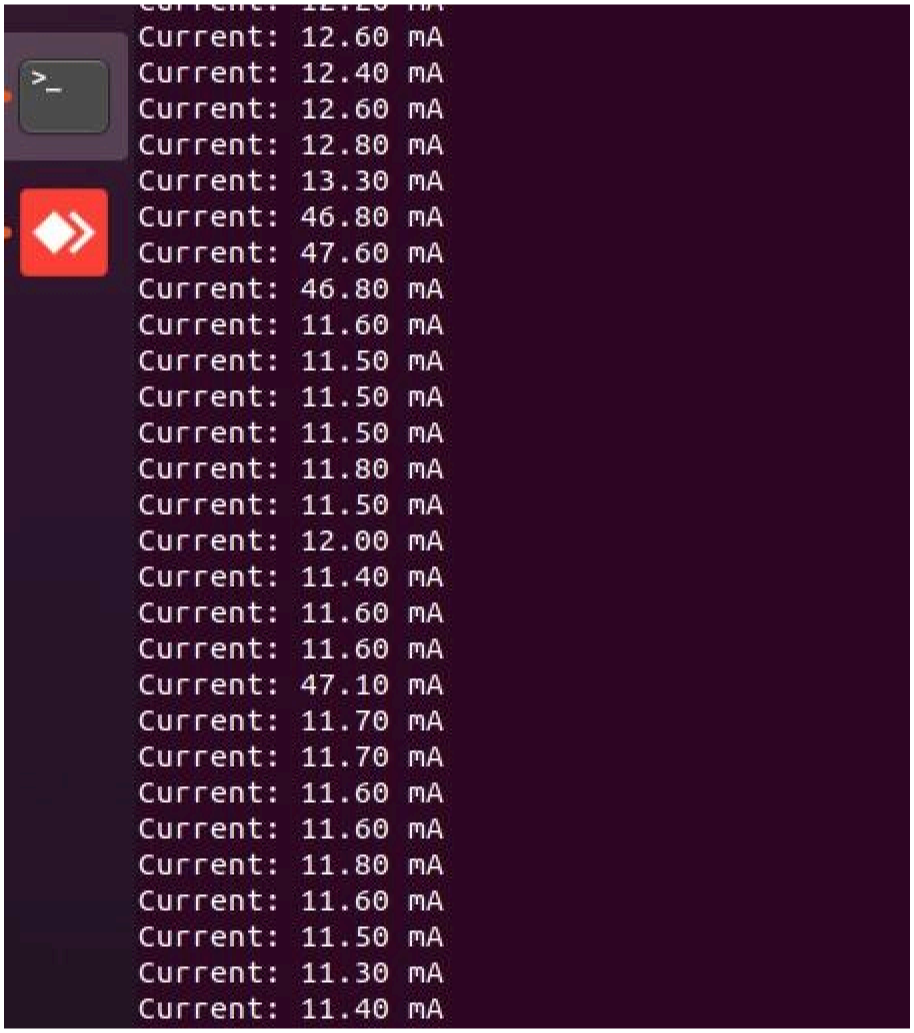
3. **Deep Sleep Mode:**
    - The power consumption drops drastically in deep sleep mode, as seen in the final image, where the current drops to around 11.00 mA or lower.
    - In deep sleep mode, the device minimizes its activity to a minimum. Most sensors and peripherals are turned off, and the device is in a state where only essential functions might remain active.

- **Power consumption strategy:** Deep sleep mode is the most power-efficient state. The device significantly reduces power usage, allowing for prolonged battery life. In this mode, the device can be kept in a near-zero power state until it needs to wake up for specific events, thus extending battery life significantly.



```
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.25°C (61.25°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.12°C (61.03°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 16.00°C (60.80°F)
Temperature between 16°C and 17°C. Entering light sleep...
Temperature: 15.88°C (60.58°F)
Temperature below 20°C. Entering deep sleep...
,8O��W(�����F�A�Scanning for I2C devices...
Device found at address 0x48
Temperature: 15.75°C (60.35°F)
Temperature below 20°C. Entering deep sleep...
 � ��W��q*Ë%�Scanning for I2C devices...
Device found at address 0x48
Temperature: 15.75°C (60.35°F)
Temperature below 20°C. Entering deep sleep...
```

Figure 6: Activated Deep Sleep mode at temperature below 16°C



```
Current: 12.60 mA
Current: 12.40 mA
Current: 12.60 mA
Current: 12.80 mA
Current: 13.30 mA
Current: 46.80 mA
Current: 47.60 mA
Current: 46.80 mA
Current: 11.60 mA
Current: 11.50 mA
Current: 11.50 mA
Current: 11.50 mA
Current: 11.80 mA
Current: 11.50 mA
Current: 12.00 mA
Current: 11.40 mA
Current: 11.60 mA
Current: 11.60 mA
Current: 47.10 mA
Current: 11.70 mA
Current: 11.70 mA
Current: 11.60 mA
Current: 11.60 mA
Current: 11.80 mA
Current: 11.60 mA
Current: 11.50 mA
Current: 11.30 mA
Current: 11.40 mA
```

Figure 7: Current measurement at deep sleep mode

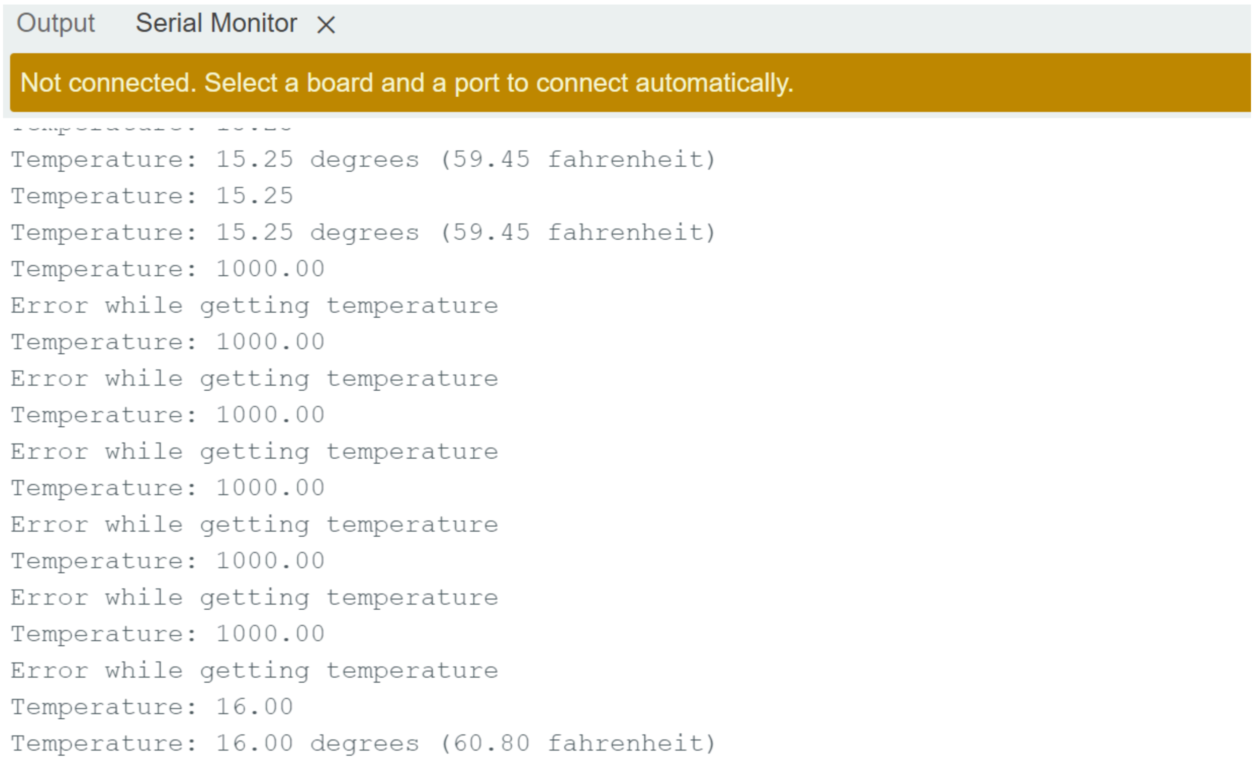
Troubleshoot and Debugging Problems

Below are issues encountered during debugging.

Problem 1 (LM75 Temperature Reading):

Issue Description:

After programming the **ESP32** to read temperature data from the **LM75** sensor via the **I2C** protocol and transmitting it to a PC using the **UART** protocol, the **Serial Monitor** sometimes displays invalid values or error messages like:


Output
Serial Monitor
Not connected. Select a board and a port to connect automatically.

```
Temperature: 15.25 degrees (59.45 fahrenheit)
Temperature: 15.25
Temperature: 15.25 degrees (59.45 fahrenheit)
Temperature: 1000.00
Error while getting temperature
Temperature: 1000.00
Error while getting temperature
Temperature: 1000.00
Error while getting temperature
Temperature: 1000.00
Error while getting temperature
Temperature: 1000.00
Error while getting temperature
Temperature: 1000.00
Error while getting temperature
Temperature: 16.00
Temperature: 16.00 degrees (60.80 fahrenheit)
```

Figure 9: Incorrect Temperature Reading

Possible Causes and Debugging Steps:

Potential Cause	Symptoms	Debugging Approach	Solution Implemented
Weak/No I2C Pull-up Resistors	Unstable I2C communication, intermittent readings.	Used an I2C scanner to check sensor detection.	Added 4.7kΩ pull-up resistors to SDA & SCL.
Incorrect I2C Address	Sensor not responding.	Verified LM75 address (0x48) using Wire.beginTransmission().	Corrected the address if necessary.
Timing Issues in I2C Reads	Occasional invalid readings.	Added delays between I2C transactions to allow proper data retrieval.	Introduced a 10ms delay after each sensor read.
Noise on I2C Lines	Data corruption and failed reads.	Probed I2C lines using an oscilloscope for noise analysis.	Shortened wires
Power Supply Instability	Unreliable sensor readings.	Measured voltage levels on ESP32 and LM75.	Used a capacitor (100nF) near the LM75 power pin for stability.
Sensor Initialization Failure	"nan" values appear frequently.	Added debugging logs to check initialization sequence.	Reset sensor properly in setup().

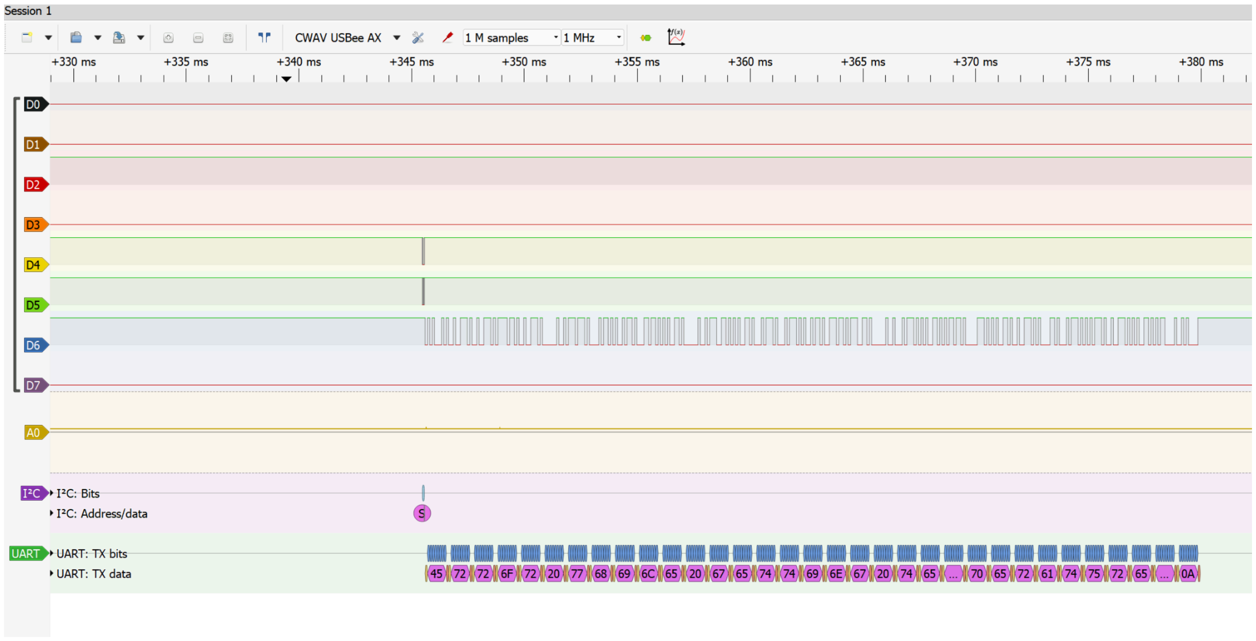
Ground A0,A1, A2 Pin of LM75	Garbage values	Proper ground need A0,A1,A2 pins of LM75 sensor	Grounded These three pins

Final Fix:

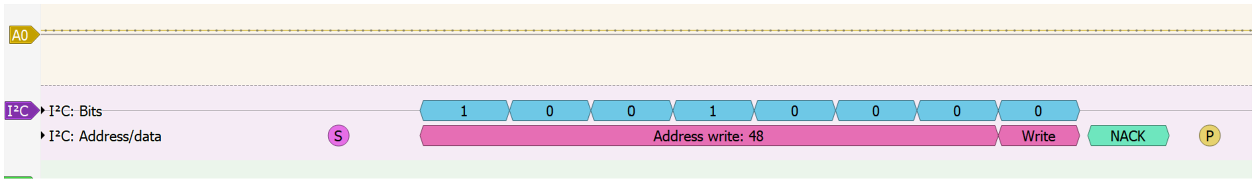
- **Added 4.7kΩ pull-up resistors** to I2C lines.
- **Introduced a small delay (10ms)** after each I2C sampling data from the sensor.
- **Ensured stable power supply** by adding a **100nF capacitor** near the LM75.
- **Grounded Pin A0,A1,A2.**

Debugging Through LHT00SU1 Logic Analyzer

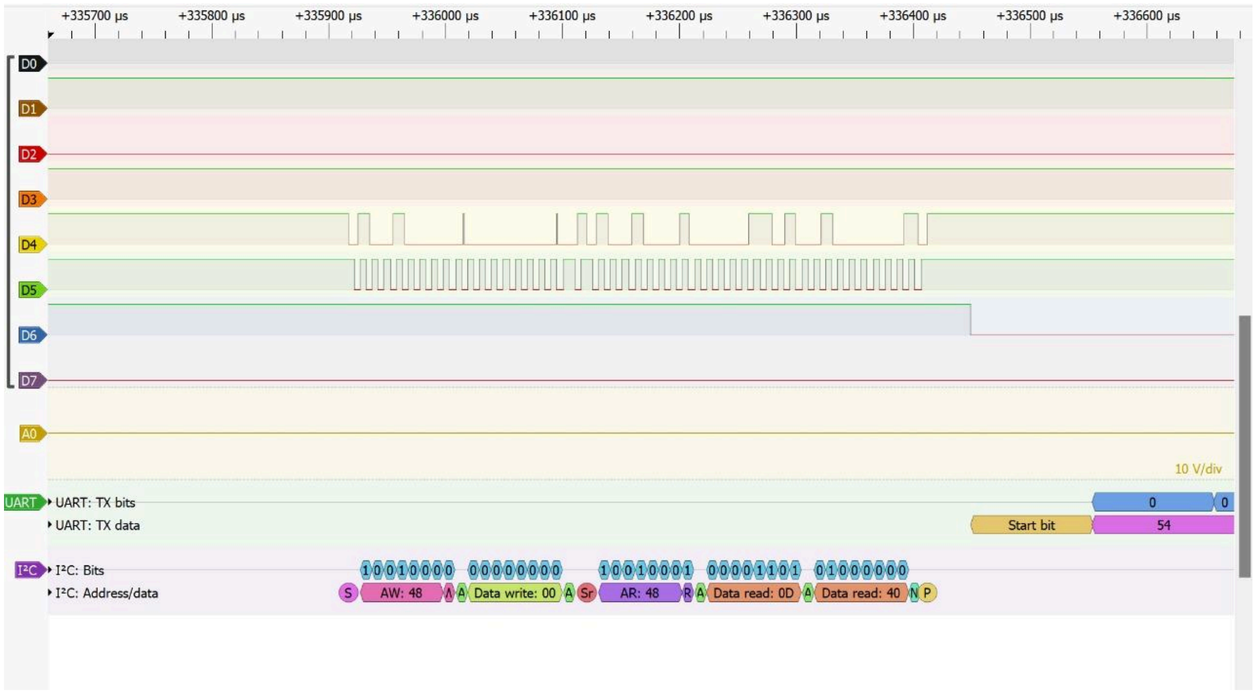
I used the LHT00SU1 logic analyzer for debugging, with PulseView software to analyze the signals. This logic analyzer was utilized throughout the project and has the capability to function as both an oscilloscope and a digital signal analyzer. Prior to adding the pull-up resistors and grounding the A0, A1, and A2 address pins of the LM75 sensor, I observed the output on the logic analyzer as shown below:



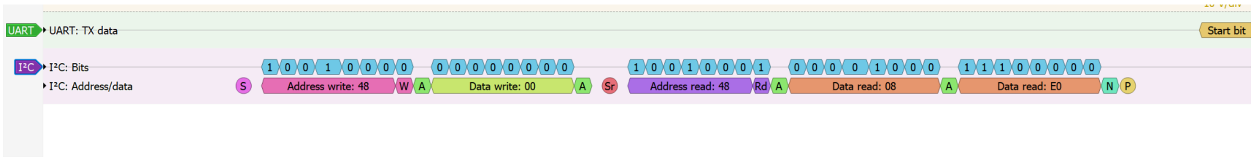
When I zoom in on the I2C signal, I observe that the master (ESP32) initiates communication with a start condition, followed by sending the 7-bit address of the LM75 sensor (48 in decimal). Data is written to the sensor, and the master receives a NACK (Not Acknowledge) signal before concluding the transaction with a stop condition. This process is intended for reading sensor data. However, I noticed that the data is not being read correctly. In the next step, I will add a pull-up resistor to see if this resolves the issue.



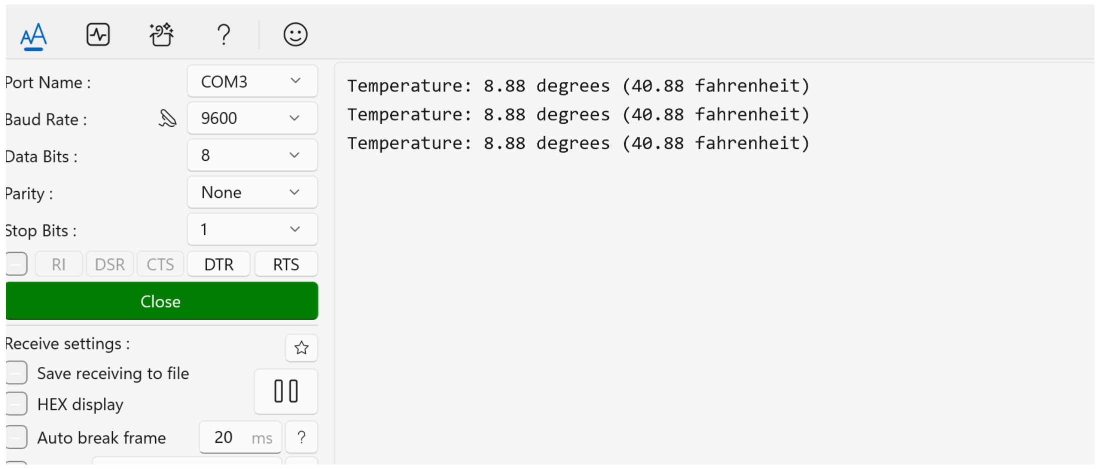
After adding pull-up resistors to the SCL and SDA lines, I observed no change. Next, I grounded the A0, A1, and A2 address pins of the LM75 sensor. This action resolved the I2C communication issue with the LM75 temperature sensor, indicating that grounding the address pins fixed the problem.



What I expect is that the ESP32 writes to and reads from the sensor at address 0x48. After making the necessary adjustments, I observe that the actual temperature reading is now displayed correctly.



Below is the information to observe on the serial monitor. This will allow us to verify the data being read from the LM75 temperature sensor and ensure it is being correctly displayed.



```
Temperature: 8.88 degrees (40.88 fahrenheit)
Temperature: 8.88 degrees (40.88 fahrenheit)
Temperature: 8.88 degrees (40.88 fahrenheit)
```

The temperature displayed in the serial output is 8.88°C, which corresponds to 40.88°F. Let's break down how I can convert this actual output to the raw data I'm receiving:

Data Breakdown:

The raw I2C data is:

- **First byte (0x08)** = 8 (integer part)
- **Second byte (0xE0)** = 224 (fractional part)

The LM75 sensor provides temperature data in a specific format:

- The **first byte** holds the integer part of the temperature (in Celsius).
- The **second byte** holds the fractional part, where each bit corresponds to a 0.125°C increment.

So, the **fractional part** (0xE0 = 224) corresponds to 224 * 0.125 = 28°C.

Correct Temperature Calculation:

- **Integer part**: 8°C
- **Fractional part**: 224 * 0.125 = 28°C

So, the formula becomes:

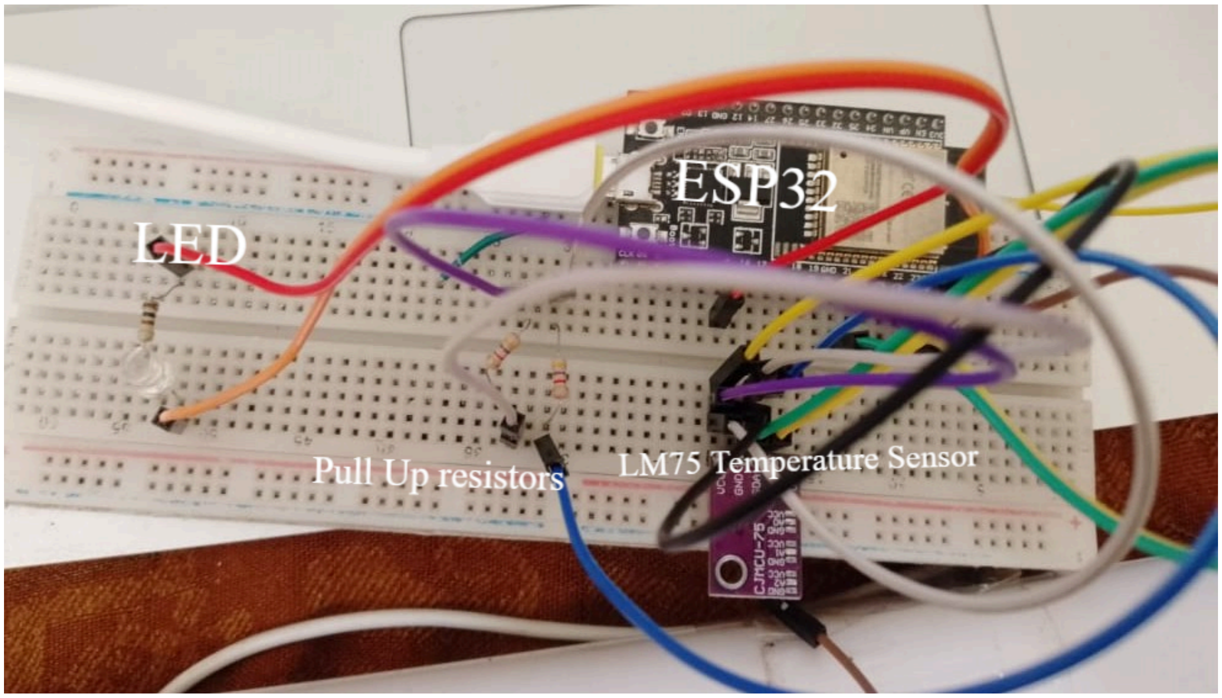
$$Temperature \; = 8 \; + (224/256) \;\; = \; 8.8755\,°\mathrm{C}$$

This matches with the serial monitor value of **8.88°C** (slight rounding difference).

For Fahrenheit:

Temperature in Fahrenheit=(8.875×9/5)+32=40.88°F

Figure 4: Snapshot of hardware design on breadboard



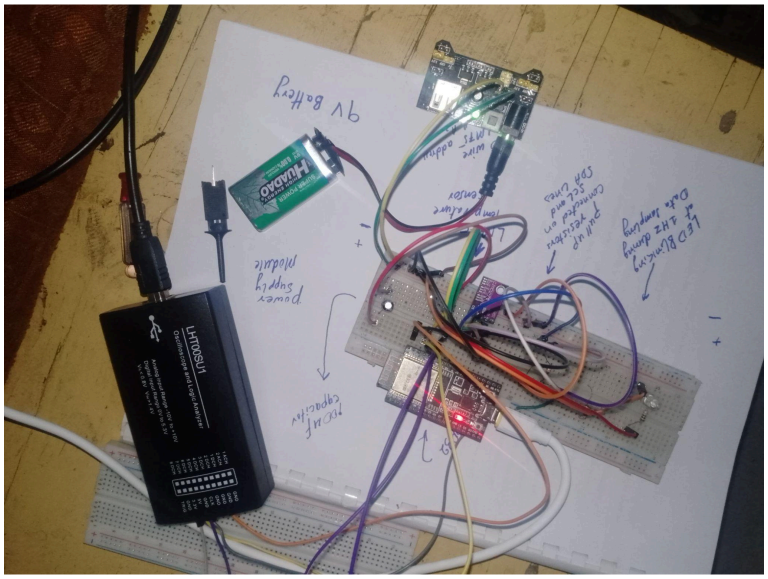

LHT00SU1
Oscilloscope and Logic Analyzer
9V Battery
power supply module
100uF capacitor
pull up resistors connected on SCL and SDA Lines
LED Blinking at 1HZ during Data sampling

Figure 10: Testing with LHT000SU1 Logic Analyzer

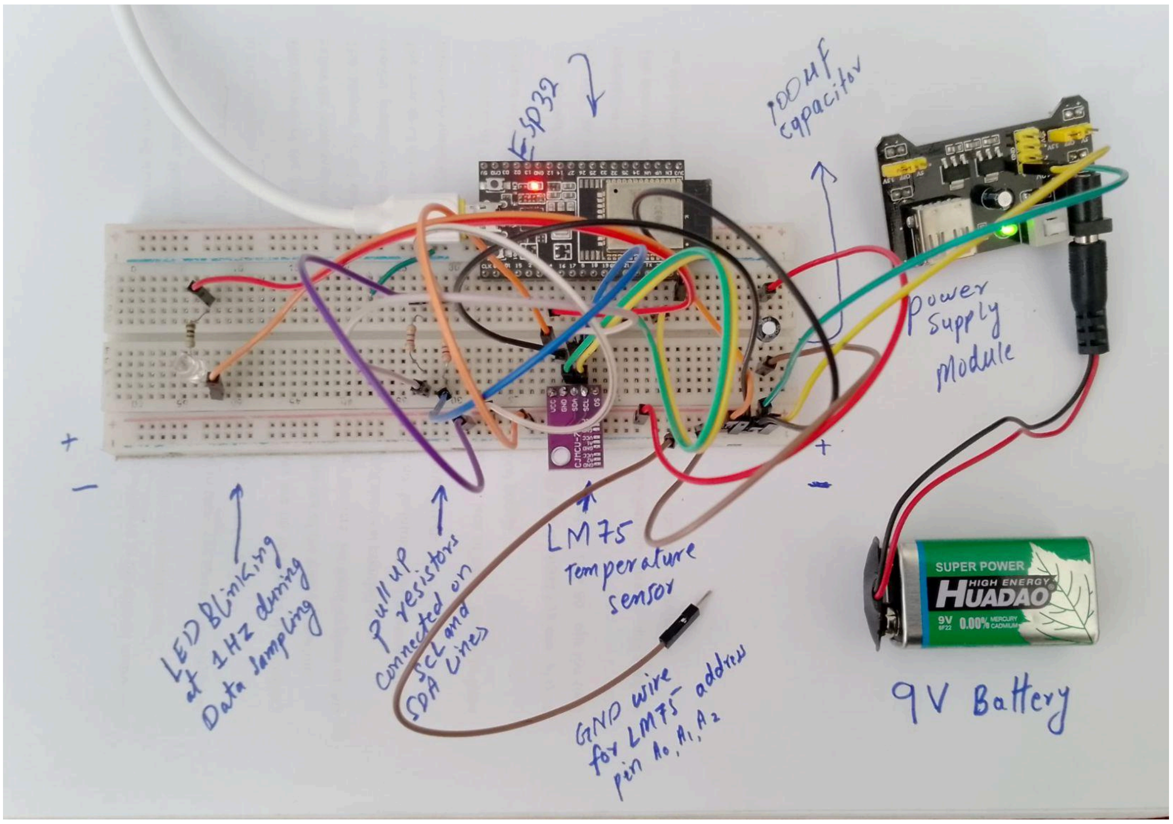
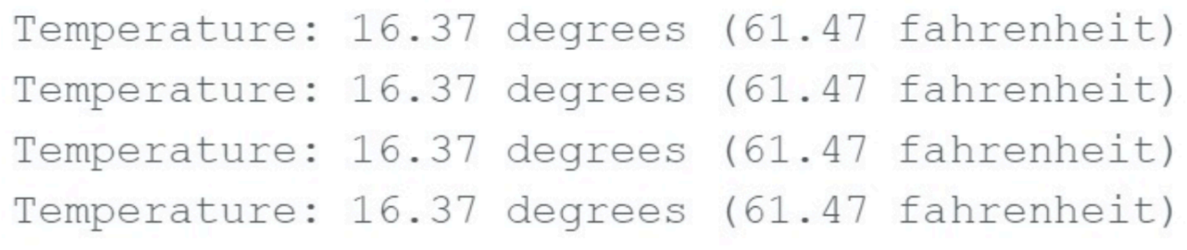

ESP32
100uF capacitor
power Supply module
+
-
LED Blinking at 1Hz during Data Sampling
pull up resistors connected on SCL and SDA Lines
LM75 Temperature sensor
GND wire for LM75 address pin A0, A1, A2
9V Battery

Figure 11: Snapshot of hardware design on breadboard

Outcome:

After applying these fixes, the ESP32 successfully read temperature data from LM75 without errors, and the readings remained stable.



```
Temperature: 16.37 degrees (61.47 fahrenheit)
Temperature: 16.37 degrees (61.47 fahrenheit)
Temperature: 16.37 degrees (61.47 fahrenheit)
Temperature: 16.37 degrees (61.47 fahrenheit)
```

Figure 12: Temperature reading after fixes

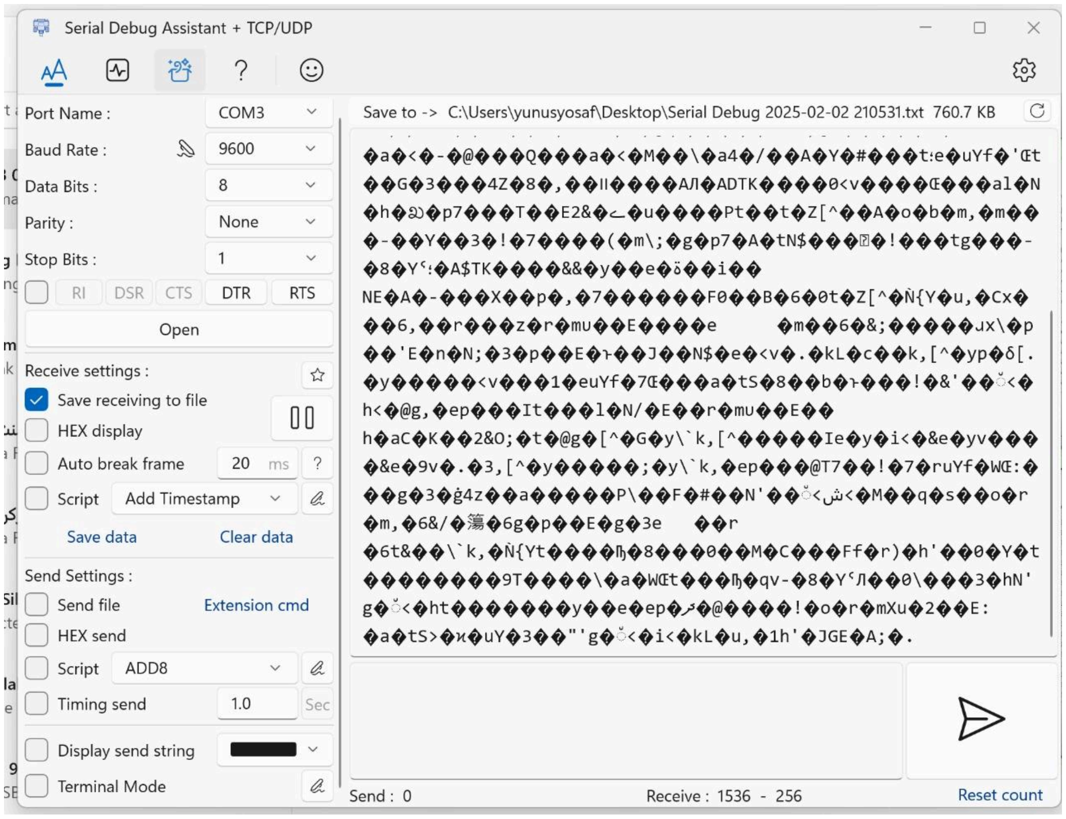
Task 2: Debug the Chaos

Problem: The LED blinks, but the UART output is gibberish.

1. Debugging Plan
2. Outline steps to diagnose the issue (tools, signal checks).
3. Hypothesize 2–3 root causes (e.g., baud rate mismatch, faulty pull-ups and how their symptoms would present).

Problem 2 (UART Baud Rate Mismatched)

The ESP32 reads temperature data from the LM75 sensor at address 0x48 via the I2C interface and transmits it over UART. The blinking LED serves as a confirmation that the ESP32 is successfully reading data from the LM75 sensor. However, the output in the Serial Debug Assistant (a Windows application available on the Microsoft Store) appears as gibberish, suggesting an issue with UART transmission. The following figure illustrates the output of our code:



Port Name : COM3
Baud Rate : 9600
Data Bits : 8
Parity : None
Stop Bits : 1

Receive settings :
[✓] Save receiving to file
HEX display
Auto break frame 20 ms
Script Add Timestamp

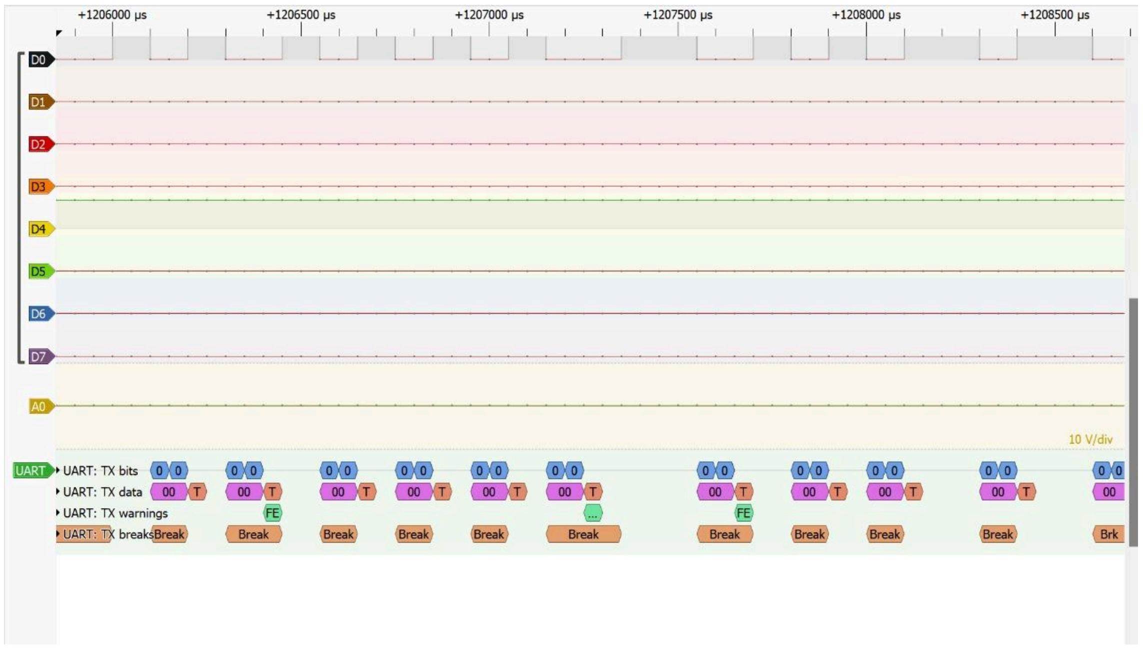
Send Settings :
Send file
HEX send
Script ADD8
Timing send 1.0 Sec
Display send string
Terminal Mode

Save to -> C:\Users\yunusyosaf\Desktop\Serial Debug 2025-02-02 210531.txt 760.7 KB

```
�a�<�-�@����Q����a�<�M��\�a4�/��A�Y�#����t꞉e�uYf�'Œt
��G�3����4Z�8�,��Ⅱ�����АЛ�ADTK�����0<v�����Œ����al�N
�h�ည�p7����T��E2&�ݎ�u�����Pt���t�Z[^��A�o�b�m,�m��
�-��Y��3�!�7�����(�m\;�g�p7�A�tN$����⍰�!����tg����-
�8�Yˤ꞉�A$TK�����&&�y��e�ỏ��i��
NE�A�-����X��p�,�7��������F0���B�6�0t�Z[^�Ǹ{Y�u,�Cx�
��6,��r����z�r�mu��E�����e     �m��6�&;������ɹx\�p
��'E�n�N;�3�p��E�ɾ��J��N$�e�<v�.�kL�c��k,[^�yp�δ[.
�y������<v����1�euYf�7Œ����a�tS�8��b�ɾ����!�&'��̆<�
h<�@g,�ep����It����1�N/�E��r�mu��E��
h�aC�K��2&O;�t�@g�[^�G�y\`k,[^������Ie�y�i<�&e�yv���
�&e�9v�.�3,[^�y������;�y\`k,�ep����@T7��!�7�ruYf�WŒ:�
��g�3�ġ4z��a������P\��F�#��N'��̆<ش<�M��q�s��o�r
�m,�6&/�簜�6g�p��E�g�3e   ��r
�6t&��\`k,�Ǹ{Yt�����ђ�8�����0��M�C����Ff�r)�h'���0�Y�t
����������9T�����\�a�WŒt����ђ�qv-�8�YˤЛ��0\����3�hN'
g�̆<�ht��������y��e�ep�ޞ�@������!�o�r�mXu�2��E:
�a�tS>�ʞ�uY�3��"'g�̆<�i<�kL�u,�1h'�JGE�A;�.
```

Send : 0 Receive : 1536 - 256 Reset count

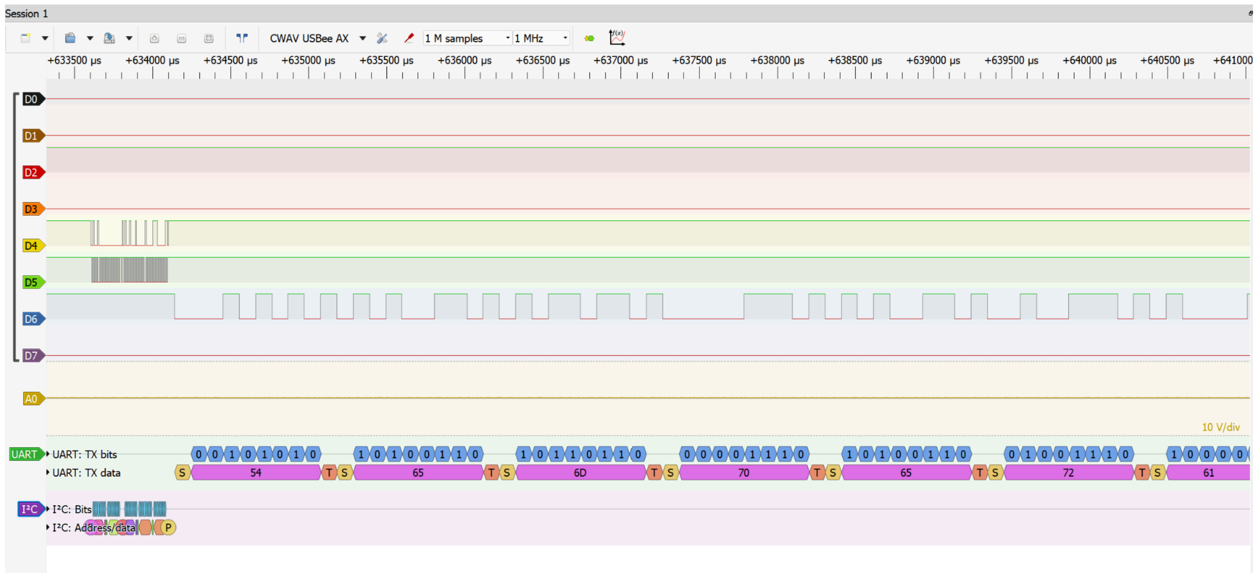
Figure 13: Gibberish reading through PC from UART interface



The data I was receiving on the logic analyzer through the UART transmission lines was corrupted. This could be caused by noise or invalid transmission between the PC and the ESP32. To start troubleshooting, I first checked the baud rate in both PulseView software and on the ESP32.

After confirming that the baud rate was the same on both ends, I suspected that the issue could be related to the sampling frequency. Initially, I had set the sampling frequency in PulseView to 100kHz, which is the minimum frequency required for debugging UART signals. However, I realized that the recommended sampling frequency for more accurate analysis is 1MHz.

To address this, I increased the sampling frequency in PulseView to 1MHz and planned to analyze the data again with this setting.

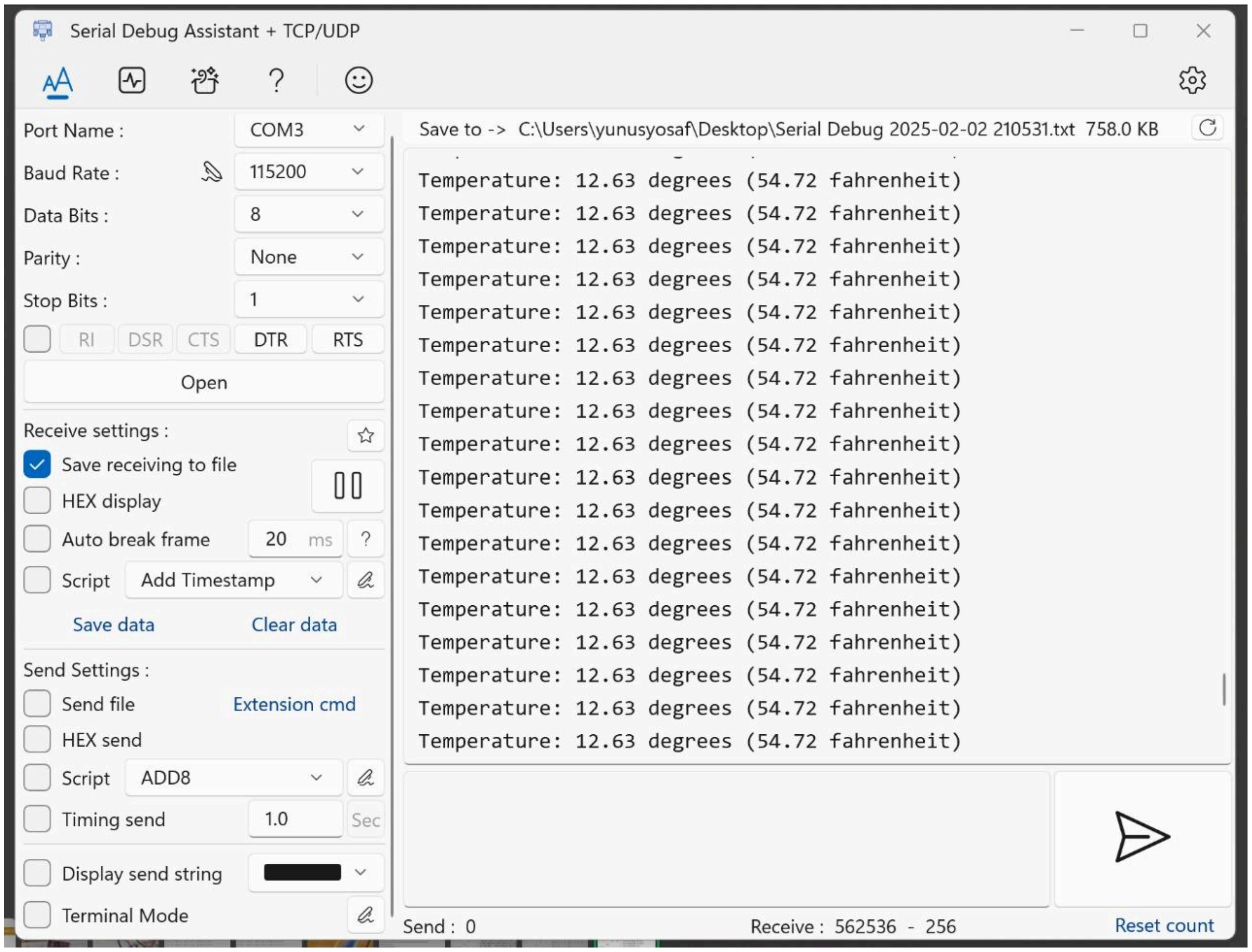


Sampling Frequency Setting:

At a 100 kHz sampling frequency, Pulseview logic analyzer showed a data frame error. However, it displayed the correct data frames at a 1 MHz sampling frequency.

Baud Rate Setting between ESP32 and PC:

For accurate readings in the serial monitor, the UART baud rate must match on both ends. In the code, I set the baud rate to 115200 bps, but in the Serial Debug Assistant, it was set to 9600 bps. After synchronizing the baud rate in both the Serial Debug Assistant and Pulseview software to 115200 bps (the same as the ESP32 microcontroller), the data was displayed correctly, as shown in the figure below:



Port Name : COM3
Baud Rate : 115200
Data Bits : 8
Parity : None
Stop Bits : 1

[✓] Save receiving to file
[ ] HEX display
[ ] Auto break frame 20 ms
[ ] Script Add Timestamp
[ ] Send file
[ ] HEX send
[ ] Script ADD8
[ ] Timing send 1.0 Sec
[ ] Display send string
[ ] Terminal Mode

Save to -> C:\Users\yunusyosaf\Desktop\Serial Debug 2025-02-02 210531.txt 758.0 KB

```
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
Temperature: 12.63 degrees (54.72 fahrenheit)
```

Send : 0 Receive : 562536 - 256

Figure 14: Correct reading after baud rate fixes

Collaboration

1. How would you explain the issue to a software engineer?
2. What common mistake might a junior engineer make in this design?

Explaining the Issue to a Software Engineer:

Resolution of LM75A Sensor Reading Issues - Troubleshooting Summary

Here are the troubleshooting steps and resolutions to communicate to the software team. the issue of inconsistent temperature readings from the LM75A sensor. We were seeing invalid or 'NaN' temperature data, and after a thorough investigation, I identified several factors that were causing this problem. Below is a breakdown of the issues and the corrective actions taken:

1. Pull-up Resistor Issue (I2C Communication):

I2C communication requires proper pull-up resistors on the SDA (data) and SCL (clock) lines to ensure stable signal levels. In our case, the values of the pull-up resistors were insufficient, leading to unstable communication and intermittent sensor readings. I added 4.7kΩ resistors to both lines, which resolved the issue of unreliable data.

Action Taken:

- Added 4.7kΩ pull-up resistors on both SDA and SCL lines.

2. I2C Addressing:

The LM75A sensor uses a fixed I2C address of 0x48. If the address in the code does not match this fixed address, the microcontroller will not be able to read data from the sensor, leading to erroneous or missing readings. I verified that the correct I2C address was configured in the code.

Action Taken:

- Confirmed the correct I2C address (0x48) was specified in the code.

3. Timing and Delays in I2C Communication:

I2C communication is timing-sensitive. If the microcontroller attempts to read data from the sensor too quickly, especially when it's still processing a previous reading, it may receive incomplete or corrupted data. I added a 10ms delay between each read to ensure proper timing and communication.

Action Taken:

- Introduced a 10ms delay between sensor readings to stabilize the communication.

4. Power Stability:

Voltage drops or noise in the power supply can affect sensor performance, causing unreliable readings. To address this, I added a 100nF capacitor near the LM75A sensor to filter out any power fluctuations and ensure a stable voltage supply.

Action Taken:

- Added a 100nF capacitor near the LM75A sensor for power filtering.

5. Grounding of Address Pins:

The LM75A sensor requires the address pins (A0, A1, A2) to be properly grounded. If these pins are floating or incorrectly grounded, it could cause the sensor to output erroneous data. I verified that all address pins were properly grounded.

Action Taken:

- Ensured all address pins (A0, A1, A2) were grounded.

6. UART Baud Rate Mismatch:

We encountered an issue with garbage data being displayed in the Serial Debug Assistant. This was due to a mismatch between the baud rate set in the ESP32 firmware and the baud rate configured in the serial terminal. The ESP32 was initially set to 115200 bps, while the terminal was set to 9600 bps. After aligning both baud rates to 115200 bps, the data was displayed correctly.

Action Taken:

- Synced the baud rate between the ESP32 and the serial terminal to 115200 bps.

7. Sampling Frequency Mismatch in PulseView Logic Analyzer:

While debugging with PulseView, I observed errors in the UART transmission frames. Despite the baud rates being correctly synchronized, the issue persisted. After further investigation, I found that the sampling frequency in PulseView was not high enough to capture the signal accurately, likely causing signal aliasing. I adjusted the sampling frequency to 4-5 times the baud rate, which resolved the issue and allowed proper signal analysis.

Action Taken:

- Increased the PulseView sampling frequency to 4-5 times the baud rate (around 460800 Hz) to accurately capture the UART signal.

Outcome:

By addressing these issues, adding proper pull-up resistors, ensuring correct I2C addressing, introducing proper delays, stabilizing the power supply, grounding the address pins, fixing the UART baud rate mismatch, and adjusting the sampling frequency, I achieved stable and accurate temperature readings from the LM75A sensor.

Common Pitfalls Junior Engineers Might Encounter While Troubleshooting Embedded Systems

When troubleshooting embedded systems, junior engineers may overlook some critical aspects that can lead to frustrating issues or errors. Here are some common mistakes they should be aware of:

1. Forgetting Pull-up Resistors:

In I2C communication, pull-up resistors on the SDA (data) and SCL (clock) lines are essential for ensuring proper data transfer. Without these resistors, the signals can become unstable, which can cause the system to malfunction or provide incorrect readings. A junior engineer might forget to add them, leading to unreliable communication.

Tip: Always ensure that the SDA and SCL lines have the proper pull-up resistors, typically 4.7kΩ.

2. Incorrect I2C Address:

Each I2C device has a unique address, and if the address is incorrectly set or conflicts with another device, the system won't be able to communicate properly with the sensor. A beginner might confuse the address or fail to check for address conflicts, resulting in no data being received from the sensor.

Tip: Double-check the I2C address in your code and verify it matches the sensor's datasheet. Ensure there are no address conflicts.

3. Not Debugging Thoroughly:

Junior engineers might sometimes skip detailed debugging, leading them to miss subtle issues such as timing problems, power fluctuations, or other small glitches. They might also forget to use important debugging tools like oscilloscopes, logic analyzers, or I2C scanners to pinpoint the problem.

Tip: Always take the time to troubleshoot thoroughly. Use debugging tools like oscilloscopes or logic analyzers to track signal integrity and ensure timing is correct.

4. Not Adding Delays Between Sensor Reads:

When reading data from sensors, it's crucial to add a delay between each read to allow the sensor enough time to process and update the data. If the reads happen too quickly, the data might be incomplete or corrupted. A junior engineer might overlook this, leading to inaccurate readings.

Tip: Make sure to add appropriate delays (typically a few milliseconds) between sensor readings to ensure stable data retrieval.

5. Not Checking Power Stability:

An unstable power supply can interfere with the sensor’s operation, causing unreliable readings or failure to operate. Junior engineers might forget to check the stability of the power supply, which could lead to subtle issues down the line.

Tip: Always verify that the power supply is stable and within the sensor’s recommended voltage range. Adding decoupling capacitors can help filter out noise.

6. Insufficient Documentation:

Junior engineers sometimes fail to document their design decisions, troubleshooting steps, and solutions thoroughly. This lack of documentation can make it difficult for others (or even themselves) to understand the system or continue working on it later.

Tip: Keep clear, concise documentation of your design process, issues encountered, and how you solved them. This makes it easier for you and others to maintain and extend the project.

Bonus: Power Management

In my design, I focused on optimizing the power consumption of the battery-operated device by implementing a multi-mode power strategy. This strategy includes three power modes, each based on the temperature readings:

- **Active Mode:**
  When the temperature exceeds 17°C, the device operates at full power. In this mode, the microcontroller reads data from the LM75 sensor and transmits it via UART. This mode consumes the highest current, around **47.00 mA**.
- **Light Sleep Mode:**
  When the temperature is between 16°C and 17°C, the device enters Light Sleep Mode to conserve power. Most functions are paused, but the device remains active enough to perform minimal tasks, such as maintaining readiness for quick wake-ups. The current consumption drops to around **12.50 mA**.
- **Deep Sleep Mode:**
  When the temperature falls below 16°C, the device enters Deep Sleep Mode, which significantly reduces power usage. The device minimizes activity to the essentials, with current consumption dropping to around **11.00 mA** or lower, greatly extending battery life.

By implementing these temperature-based power management modes, I optimized the device's power consumption, ensuring prolonged battery life and efficient operation.

EMC/EMI (Electromagnetic Compatibility & Electromagnetic Interference)

I also focused on ensuring the system would operate in a low-noise environment and not interfere with other electronics, which is especially important for reliable communication, particularly with protocols like I2C. The steps I took to address EMC/EMI include:

- **PCB Design Considerations:**
  While not explicitly detailed, I ensured proper grounding and shielding practices to minimize electromagnetic interference. I also focused on filtering power lines to reduce noise, ensuring the system met necessary EMC standards.
- **Noise Reduction:**
  During the debugging process, I used the LHT00SU1 Logic Analyzer to monitor and analyze noise on the I2C lines. This helped identify potential noise sources that could cause communication errors or data corruption. To mitigate this, I shortened the wiring and grounded the I2C address pins (A0, A1, A2) on the LM75 sensor, which improved the stability of the communication and reduced interference.