ESP32C3 SHTC3 Setup

lab2.md
File

This is the lab2.md file that we talked about in a previous conversation "Lab 2 Guide". I need help with the same project, but this time specifically for section 2.2.
I need to configure my ESP32C3 to be able to read temperature and humidity, then to print the temperature in degrees C and F and a percentage for humidity once every 2 seconds. An example output is included in the lab2.md document.
Special Requirements:
- power up and power down between reads. Power up should be called at most once every 2 seconds.
- separate functions for temperature and humidity. Each should read at most 3 bytes.
- use the checksum value to check read results.
I also have the SHTC3 sensor datasheet on hand. In order to avoid overwhelming you with too much information, I will give you the datasheet information later. Right now, I'd like you to formulate a game plan for me to complete section 2.2 of the project.

To complete Section 2.2 of the project, here's a structured game plan:

Step 1: Understand the Requirements

The requirements for Section 2.2 are:

1. Configure ESP32C3 to read temperature and humidity using the SHTC3 sensor.
2. Print temperature in degrees Celsius and Fahrenheit, and humidity percentage once every 2 seconds.
3. Implement the following special requirements:
    - Power up and power down the sensor between reads.
    - Use separate functions for reading temperature and humidity, each reading at most 3 bytes.
    - Use the checksum to verify read values.

Step 2: Set Up the Development Environment

1. **Platform Setup**: Make sure you have the ESP-IDF environment installed and configured.
2. **Library Dependencies**: Install necessary libraries for I2C communication to interface with the SHTC3 sensor.
3. **Include Header Files**: Include libraries for ESP32, FreeRTOS, and I2C communication in the code.

Step 3: Implement Core Logic

1. Initialize I2C Communication

- **Configure I2C**: Set up the I2C master interface with ESP32 to communicate with the SHTC3 sensor.
- **Set Pins**: Configure the appropriate GPIO pins for SCL and SDA according to your hardware configuration.

2. Sensor Power Management

- **Power Up/Power Down Functions**:
  - Create a `sensor_power_up()` function that powers up the SHTC3 sensor.
  - Create a `sensor_power_down()` function that powers down the sensor.
  - Ensure these functions are called every 2 seconds, but only once for each read cycle.

3. Reading Temperature and Humidity

- **Temperature Reading Function** (`read_temperature()`):
  - Send the appropriate command to the SHTC3 to trigger a temperature measurement.
  - Read **3 bytes**: 2 for temperature data and 1 for the checksum.
  - **Validate Checksum** to ensure the data integrity.
  - Convert the data to **degrees Celsius**, then to **degrees Fahrenheit**.
- **Humidity Reading Function** (`read_humidity()`):
  - Send the appropriate command to trigger a humidity measurement.
  - Read **3 bytes**: 2 for humidity data and 1 for the checksum.
  - **Validate Checksum** to verify accuracy.

- Convert the data to a **percentage** value.

4. Main Loop for Readings

- **Loop with Delay**:
  - Create a main loop that calls `sensor_power_up()`, waits for initialization, reads temperature and humidity, then calls `sensor_power_down()`.
  - Use a delay of 2 seconds (`vTaskDelay()` or similar function) to ensure proper timing.

5. Print Data

- Print temperature in both Celsius and Fahrenheit, and humidity in percentage format as shown in the example:

```csharp
Temperature is 20C (or 68F) with a 40% humidity
```

Step 4: Pseudocode and Breakdown

1. Initialization

```c
void app_main() {
    // I2C Configuration
    i2c_master_init();
    while (1) {
        sensor_power_up();
        vTaskDelay(pdMS_TO_TICKS(100));  // Allow time for sensor to power up

        // Read temperature and humidity
        float temperature_c = read_temperature();
        float temperature_f = convert_to_fahrenheit(temperature_c);
        float humidity = read_humidity();

        // Print the results
        printf("Temperature is %.0fC (or %.0fF) with a %.0f%% humidity\n",
                temperature_c, temperature_f, humidity);

        sensor_power_down();
        vTaskDelay(pdMS_TO_TICKS(2000));  // Wait for 2 seconds before the next read
```

```
    }
}
```

2. Sensor Power Functions

```c
void sensor_power_up() {
    // Send command to power up SHTC3 (Refer to datasheet for the command)
}

void sensor_power_down() {
    // Send command to power down SHTC3 (Refer to datasheet for the command)
}
```

3. Temperature and Humidity Read Functions

```c
float read_temperature() {
    // Send temperature read command via I2C
    // Read 3 bytes: data (2 bytes), checksum (1 byte)
    // Validate checksum
    // Convert data to Celsius and return
}

float read_humidity() {
    // Send humidity read command via I2C
    // Read 3 bytes: data (2 bytes), checksum (1 byte)
    // Validate checksum
    // Convert data to percentage and return
}

float convert_to_fahrenheit(float celsius) {
    return (celsius * 9.0 / 5.0) + 32.0;
}
```

Step 5: Verify with Datasheet

- Use the SHTC3 sensor datasheet to get:
    - **I2C Commands**: Understand the specific I2C commands required for power up, power down, and reading temperature/humidity.

- **Checksum Calculation**: Check the algorithm for verifying the checksum.

Step 6: Testing and Debugging

1. **Check Sensor Output**: Test each function independently to verify that the sensor returns valid values.
2. **Debugging**: Use `ESP_LOGI()` for additional debugging messages during development.
3. **Validation**: Ensure that the sensor powers up and powers down correctly, and that the readings are printed every 2 seconds without any issues.

Next Steps

- You mentioned you have the SHTC3 datasheet. If you could provide the relevant sections on the power-up/power-down commands, reading temperature and humidity, and checksum calculation, I can help refine the specific implementation details.

I have step 1 done and a directory specifically for 2.2 made. I don't have any files in the directory yet. Where do I start?

Lab2 Sensor Setup

I've set up an initial `main.c` file for your Lab 2.2. This includes the basics for I2C initialization, sensor power management, and placeholder functions for reading temperature and humidity. You can start by filling in the specific I2C commands using the SHTC3 datasheet.

Next, let me know when you're ready to add the specific I2C commands, and I can help guide you through implementing those details.

Canvas is fine. So, I just put main.c in the project directory? Don't I need to set up the project with a main directory, CMakeLists, and the like?

You're correct; setting up the project structure is essential to properly compile and build the program with ESP-IDF. Here is how you should set up the directory for your Lab 2.2 project:

1. Project Directory Structure

Create the following directory structure in your `lab2_2` folder:

```css
```

```
lab2_2/
├── CMakeLists.txt
├── sdkconfig
├── main/
│   ├── main.c
│   └── CMakeLists.txt
└── README.md
```

2. CMakeLists.txt in the Root Directory

Create a `CMakeLists.txt` file in the root directory (`lab2_2/CMakeLists.txt`) with the following content:

```cmake
cmake_minimum_required(VERSION 3.16)

include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(lab2_2)
```

This file defines your project and tells CMake to use the ESP-IDF build system.

3. CMakeLists.txt in the `main` Directory

Create another `CMakeLists.txt` file inside the `main` directory (`lab2_2/main/CMakeLists.txt`):

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

This file tells CMake which source files to include and compiles the `main.c` file.

4. sdkconfig

The `sdkconfig` file contains project-specific configuration settings. If you have already generated it during your setup, copy it to the `lab2_2/` directory. If not, run the `idf.py menuconfig` command to configure your project settings, which will automatically create this file.

5. Compiling and Flashing

After setting up the files, you can compile, flash, and monitor your project using the following commands:

```bash
idf.py set-target esp32c3  # Set the target to ESP32C3
idf.py build               # Build the project
idf.py flash               # Flash the project to the ESP32 board
idf.py monitor             # Monitor the serial output
```

With this setup, you will be able to compile and run the code on your ESP32C3 board. Let me know if you need more help configuring or writing the CMake files.

↳ Asked ChatGPT

How do you know the values to define these macros?

The values for these macros are usually determined based on the hardware setup of the ESP32C3 board and the connected sensor. Here's an explanation of each macro:

1. `I2C_MASTER_SCL_IO` **(22)** and `I2C_MASTER_SDA_IO` **(21)**:
    - These values represent the GPIO pins used for the I2C SCL (clock) and SDA (data) lines.
    - The GPIO numbers (22 for SCL and 21 for SDA) are typically chosen based on the board's hardware design or availability of these pins. You need to ensure that these values match the physical wiring of the sensor on your ESP32C3 board.
2. `I2C_MASTER_NUM` **(I2C_NUM_0)**:
    - This defines the I2C peripheral number. ESP32C3 has multiple I2C hardware peripherals, usually numbered `I2C_NUM_0` and `I2C_NUM_1`.
    - Here, `I2C_NUM_0` is used, meaning you're using the first I2C hardware peripheral.
3. `I2C_MASTER_FREQ_HZ` **(100000)**:
    - This is the frequency of the I2C clock, set to `100000` Hz, which is the standard speed for I2C communication (100 kHz).
    - You may choose higher (e.g., 400 kHz) or lower frequencies depending on the sensor capabilities and bus stability.
4. `I2C_MASTER_TX_BUF_DISABLE` **(0)** and `I2C_MASTER_RX_BUF_DISABLE` **(0)**:

- These settings disable the TX and RX buffers for the I2C driver, meaning that the driver doesn't need to allocate separate buffers for transmitting or receiving data.
- Setting them to `0` is common in simple sensor setups where direct communication is sufficient.

To determine these values for your project, you should:

- Refer to your ESP32C3 board's documentation or schematic to see which GPIO pins are available for I2C communication.
- Check the sensor datasheet to verify the maximum clock speed supported.
- Ensure that the chosen GPIO pins are correctly wired to the sensor's SCL and SDA pins.

How do I determine the i2c_master_num?

The value for `I2C_MASTER_NUM` is determined by the hardware peripherals available on your ESP32C3 chip. The ESP32C3 has multiple I2C peripherals, typically labeled as `I2C_NUM_0` and `I2C_NUM_1`. Here's how you determine which one to use:

Determining `i2c_master_num`:

1. **Hardware Availability**:
   - The ESP32C3 typically has two I2C ports (hardware modules) available: `I2C_NUM_0` and `I2C_NUM_1`.
   - You can use either of these ports depending on your application and availability of other peripherals.
2. **Choosing Between I2C_NUM_0 and I2C_NUM_1**:
   - `I2C_NUM_0`: This is the default and most common choice. It is often used because it might have default pin mappings that make it simpler to set up.
   - `I2C_NUM_1`: Use this if `I2C_NUM_0` is already in use by another peripheral or if your project requires multiple I2C interfaces (e.g., two different I2C buses connected to different sensors).
3. **Pin Multiplexing**:
   - The ESP32's GPIOs are highly flexible, and different I2C peripherals can be mapped to different GPIO pins.
   - For example, if you are already using GPIO pins 21 and 22 for `I2C_NUM_0`, but you need another I2C bus, you could use other available GPIO pins and assign them to

`I2C_NUM_1`.

4. **Practical Considerations**:
    - If your board's documentation specifies which I2C bus is mapped to which GPIO pins, it is often easiest to follow those default mappings.
    - If you're unsure, starting with `I2C_NUM_0` is a good approach, as it is commonly used and well-supported.

In your current setup:

```c
#define I2C_MASTER_NUM I2C_NUM_0
```

This means you are using the first I2C hardware peripheral, which is usually fine for most projects unless you have a specific reason to use `I2C_NUM_1`.

How to Verify Pin Compatibility

- Refer to the **ESP32C3 datasheet** or **technical reference manual** to verify which pins can be used for I2C communication.
- You can also check your development board's **schematics** to see if it has predefined pin mappings for I2C.

If you're not restricted by other peripherals or board-specific constraints, using `I2C_NUM_0` is generally the simplest and most straightforward approach.

↳ Asked ChatGPT

If these are unnecessary, why are they explicitly disabled?

The `I2C_MASTER_TX_BUF_DISABLE` and `I2C_MASTER_RX_BUF_DISABLE` macros are explicitly set to `0` to **disable the I2C driver's internal buffers** for transmit (TX) and receive (RX). This is done because, for most simple sensor applications like yours, there is no need to allocate extra memory for buffering I2C data.

Why Disable Buffers?

- **Efficiency**: Disabling the internal buffers helps to save RAM, which can be critical in embedded systems where resources are limited.

- **Direct Transactions**: If the I2C transactions are simple (e.g., sending a small command and reading sensor data directly), it is more efficient to communicate without using an internal buffer.
- **No Complex I2C Communication**: Since you're simply sending commands and receiving short data (temperature or humidity readings), the default settings (`0` for disabled buffers) are appropriate for this project.

If you were dealing with **more complex I2C communication**, such as buffering large chunks of data to send or receive, enabling these buffers by setting them to a non-zero size would be more useful. In your case, explicitly disabling the buffers clarifies that the driver will not use any additional memory for buffering data, and it directly communicates with the sensor in a straightforward manner.

↳ Asked ChatGPT

Where does this value come from?

The value `GPIO_PULLUP_ENABLE` is a predefined macro in the ESP-IDF framework that enables the **internal pull-up resistor** for the specified GPIO pin. It comes from the ESP-IDF library, which is a development framework provided by Espressif for ESP32 development.

Why is `GPIO_PULLUP_ENABLE` Used?

- **I2C Protocol Requirements**: The I2C bus requires pull-up resistors on both the SCL (clock) and SDA (data) lines to properly function. The pull-up ensures that the lines are pulled to a high logic level when not being actively driven low by a device.
- **Internal vs. External Pull-up**: By setting `sda_pullup_en` and `scl_pullup_en` to `GPIO_PULLUP_ENABLE`, you are using the **internal pull-up resistors** available on the ESP32 chip. This avoids the need for external resistors if the internal ones are sufficient for your configuration.

If the hardware setup uses external pull-up resistors instead, then setting `GPIO_PULLUP_DISABLE` may be more appropriate, as enabling both internal and external pull-ups can cause improper resistance values on the line.

Datasheet_SHTC3.pdf
PDF

I have included the SHTC3 datasheet. In order to avoid overwhelming you with too much data, let's take the functions one step at a time. I'm going to give you a function that I want to implement, and I want you to parse the datasheet and return me all relevant information required to implement the given function. When returning the information, concisely tell me exactly why it needs to be included in the function.

Great, let's implement one function at a time. Which function would you like to start with?

Let's start with sensor_power_up().

For the `sensor_power_up()` function, here is the relevant information extracted from the SHTC3 datasheet that will be necessary for implementing the power-up functionality:

SHTC3 Power-Up Details

1. **Power-Up Command**:
    - The datasheet mentions that upon VDD reaching the power-up voltage level, the SHTC3 enters the **idle state**.
    - However, to ensure proper power-up, a **wake-up command** should be issued if the sensor was in sleep mode.
    - **Wake-up Command**:
        - **Hex Code**: `0x3517`
        - **Binary Code**: `0011’0101’0001’0111`
    - This command is sent over the I2C bus to wake the sensor from sleep mode.
2. **I2C Address**:
    - The **I2C device address** for the SHTC3 is `0x70`. This address needs to be used for communication.

Steps to Implement `sensor_power_up()`

1. **Send the Wake-Up Command** using the I2C address (`0x70`).
2. **Use I2C Write Operation** to transmit the wake-up command (`0x3517`).

Here's an updated implementation of `sensor_power_up()`:

```c
void sensor_power_up() {
    ESP_LOGI(TAG, "Powering up the SHTC3 sensor");

    // SHTC3 I2C address
    uint8_t shtc3_addr = 0x70;

    // Wake-up command bytes
    uint8_t wakeup_cmd[2] = {0x35, 0x17};

    // Send the wake-up command to the sensor
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (shtc3_addr << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write(cmd, wakeup_cmd, sizeof(wakeup_cmd), true);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, pdMS_TO_TICKS(1000));
    i2c_cmd_link_delete(cmd);

    if (ret == ESP_OK) {
        ESP_LOGI(TAG, "Sensor wake-up command sent successfully");
    } else {
        ESP_LOGE(TAG, "Failed to send sensor wake-up command");
    }
}
```

Explanation:

- **I2C Address (`0x70`)**: The sensor uses this address to communicate with the I2C master.
- **Wake-Up Command (`0x3517`)**: This command is sent in two bytes: `{0x35, 0x17}`.
- **I2C Command Sequence**:
    - Create a new I2C command link using `i2c_cmd_link_create()`.
    - Send a **start condition** with `i2c_master_start(cmd)`.
    - Write the sensor's I2C address and specify it as a **write** operation with `i2c_master_write_byte()`.
    - Send the **wake-up command** with `i2c_master_write()`.
    - Complete the command by sending a **stop condition** using `i2c_master_stop(cmd)`.

- Execute the command with `i2c_master_cmd_begin()`, which takes in the I2C master number (`I2C_MASTER_NUM`), the command link (`cmd`), and a timeout.
- Finally, delete the command link with `i2c_cmd_link_delete()`.

This implementation wakes up the sensor, ensuring it transitions from sleep to idle state, making it ready for measurements.

Would you like me to assist with any further steps, like implementing the `sensor_power_down()` function next?

When I ask you to implement and explain a function, please implement the function in the canvas file and leave the explanation in the chat. Please try again with these added instructions.

I've implemented the `sensor_power_up()` function in the code. It now sends the appropriate wake-up command to the sensor using I2C, as specified in the datasheet.

Let me know if you'd like me to move on to the next function or need further clarification on anything.

Please do it again. My browser crashed so your changes were not saved

I've re-implemented the `sensor_power_up()` function in the code. It now includes the appropriate wake-up command for the SHTC3 sensor.

Let me know if you'd like me to work on the next function or need further help with anything else.

Now, please do the same for the sensor_power_down() function. Please remember I'd like the code implementation in canvas while the explanation for the code remains in chat. Once again, only parse the datasheet for relevant information for that function specifically and explain your actions in chat.

The `sensor_power_down()` function has been updated in the code. Here's a summary of what was added and why:

Relevant Information for `sensor_power_down()`

1. **Sleep Command**:

- The SHTC3 requires a **sleep command** to enter a low-power state.
- **Sleep Command Code**: `0xB098`
- This command helps reduce power consumption when the sensor is not in use.

Function Implementation

- **I2C Address (`0x70`)**: The sensor's I2C address is used again.
- **Sleep Command (`0xB098`)**: This command is sent in two bytes: `{0xB0, 0x98}`.
- **I2C Command Sequence**:
  - Create a new I2C command link using `i2c_cmd_link_create()`.
  - Send a **start condition**, followed by writing the I2C address, and then the **sleep command**.
  - Use **stop condition** to end the transaction, and then delete the command link.
  - Check the result of the command and log success or failure.

Let me know if you need help with the next function or if you need more information!

Please follow the same instructions for read_temperature()

I've implemented the `read_temperature()` function in the code. Here's what was done:

Relevant Information for `read_temperature()`

1. **Measurement Command**:
   - The SHTC3 requires a command to initiate a temperature measurement.
   - For **Normal Mode with "Read Temperature First"**:
     - **Hex Code**: `0x7CA2` (Sent as two bytes: `{0x7C, 0xA2}`)
2. **Data Reading**:
   - The sensor returns **2 bytes** of temperature data and **1 byte** of CRC checksum.
   - The I2C read sequence includes sending the sensor's address for reading, followed by reading the data and checksum.
3. **Checksum Validation**:
   - The **CRC-8 checksum** is calculated using polynomial `0x31`. The calculated checksum is compared to the received checksum for data validation.

4. **Temperature Conversion**:

   - The raw data from the sensor is converted to temperature in Celsius using the formula:

     ```scss
     Temperature (°C) = ((175.0 * Raw_Temp) / 65536.0) - 45.0
     ```

The function now reads temperature data, validates it using the checksum, and converts it to Celsius.

Let me know if you need to proceed with implementing `read_humidity()` or anything else!

↳ Asked ChatGPT

Please explain how you validated the checksum

The checksum validation in the `read_temperature()` function is performed using the **CRC-8 algorithm** with polynomial `0x31` to ensure the data received from the sensor is correct.

Here's how it works:

1. **Initialize the CRC Value**:

   - `uint8_t crc = 0xFF;` initializes the CRC value to `0xFF`. This is a common starting value for CRC-8 calculations.

2. **Loop Through Data Bytes**:

   - The loop iterates through each byte of the received data (`data[i]`).
   - For each byte, it **XORs** the current CRC value with the byte, which helps in incorporating the new byte into the checksum calculation.

3. **Bitwise Processing**:

   - The inner loop iterates **8 times** (once for each bit of the byte). During each iteration, it checks whether the **most significant bit** (`crc & 0x80`) is set.
   - If set, the CRC value is **shifted left by one** and then XORed with the polynomial `0x31`.
   - If not, it simply shifts the CRC value left by one bit.
   - This step processes each bit and modifies the CRC value accordingly.

4. **Compare the Calculated CRC with the Received CRC**:

- Finally, it compares the calculated CRC (`crc`) with the **third byte** (`data[2]`) received from the sensor.
- If the CRCs do not match, the data is considered **invalid**, and an error is logged.

This validation ensures that the two received data bytes are correct and have not been corrupted during I2C transmission. If the CRC matches, the data is used for conversion to temperature; otherwise, an error is logged, and a default value is returned.

↳ Asked ChatGPT

Can you explain how the temperature reading process works here?

The temperature reading process involves the following steps:

1. **Create an I2C Command Handle**:
   - A new I2C command link is created using `i2c_cmd_link_create()` to form a series of commands that will be executed.
2. **Initiate Communication (Start Condition)**:
   - `i2c_master_start(cmd)` starts an I2C communication session.
3. **Specify the Read Operation**:
   - `i2c_master_write_byte(cmd, (shtc3_addr << 1) | I2C_MASTER_READ, true)` writes the sensor address (`shtc3_addr` left-shifted by 1 bit) with the **read bit** set (`I2C_MASTER_READ`). This tells the sensor that data is to be read.
4. **Read Data**:
   - `i2c_master_read(cmd, data, 2, I2C_MASTER_ACK)` reads **2 bytes** of data from the sensor and sends an **ACK** after each byte to indicate successful receipt.
   - `i2c_master_read_byte(cmd, data + 2, I2C_MASTER_NACK)` reads **1 byte** of checksum data and sends a **NACK** to indicate that no more data is needed.
5. **End Communication (Stop Condition)**:
   - `i2c_master_stop(cmd)` signals the end of the I2C transaction.
6. **Send the I2C Commands**:
   - `i2c_master_cmd_begin()` sends the series of commands to the I2C bus, and the result is checked to confirm if the transaction was successful.
7. **Delete the Command Link**:

- Finally, `i2c_cmd_link_delete(cmd)` deletes the command link to free the resources used.

This sequence is crucial for reading the sensor data correctly, ensuring that the appropriate bytes are fetched in a controlled manner, and the necessary communication protocols are adhered to.

↳ `i2c_master_read_byte(cmd, data + 2, I2C_MASTER_NACK)` reads **1 byte** of checksum data and sends a **NACK** to indicate that no more data is needed.

How does this read 1 byte? I don't see how that happens from data + 2

The line `i2c_master_read_byte(cmd, data + 2, I2C_MASTER_NACK)` is used to read the third byte of data, which is the **checksum**. Here's how it works:

- **Pointer Arithmetic**: The expression `data + 2` points to the **third element** in the `data` array. In C, arrays are zero-indexed, so:
  - `data[0]` is the first byte (first data byte).
  - `data[1]` is the second byte (second data byte).
  - `data[2]` is the third byte, which stores the checksum.
- **Reading the Byte**: The `i2c_master_read_byte()` function reads **1 byte** from the I2C bus, starting from the address provided, which in this case is `data + 2`. This allows us to store the read byte in the third element of the `data` array.
- **NACK**: After reading this byte, the **NACK** (Not Acknowledge) is used to indicate that no further data will be requested from the sensor. This is the standard procedure to end an I2C read operation when no more bytes are expected.

In the context of the SHTC3 sensor:

- The sensor sends **2 bytes** of raw temperature data, followed by **1 byte** of checksum.
- The command reads the first 2 bytes into `data[0]` and `data[1]` using `i2c_master_read()`.
- The next read (`i2c_master_read_byte()`) fetches the checksum byte into `data[2]`.

This structure is used to ensure that the **entire response** (data + checksum) from the sensor is collected properly, allowing you to validate the received data with the provided checksum.

↳ Asked ChatGPT

How do you know how to convert the raw temperature?

The conversion from the raw data to temperature in Celsius uses the formula:

```c
float temperature_c = ((175.0 * raw_temperature) / 65536.0) - 45.0;
```

This formula comes from the **SHTC3 datasheet**, which provides the equation for converting the raw 16-bit temperature output from the sensor into an actual temperature value in Celsius. Here's the breakdown:

1. **Raw Temperature Value**:
   - The `raw_temperature` is a 16-bit value, composed of `data[0]` and `data[1]` (combined using `(data[0] << 8) | data[1]`).
2. **Conversion Formula**:
   - The datasheet specifies that the temperature can be calculated using:
   $$T(^\circ C) = \left(\frac{175 \times \text{Raw\_Temp}}{65536}\right) - 45$$
   - `175.0` and `65536.0` are constants provided by the manufacturer to properly scale the sensor's raw output.
   - `-45.0` is the offset to align the sensor reading with the correct temperature range.

This formula effectively scales the raw 16-bit data (which ranges from 0 to 65535) into the temperature range of **-40°C to 125°C**, which is the operating range of the sensor.

Please follow the same earlier instructions to implement read_humidity()

The `read_humidity()` function has been implemented in the code. Here's a breakdown of what was done:

Relevant Information for `read_humidity()`

1. **Measurement Command**:
   - The SHTC3 requires a command to initiate a humidity measurement.
   - For **Normal Mode with "Read Humidity First"**:

- **Hex Code**: `0x5C24` (Sent as two bytes: `{0x5C, 0x24}`).

2. **Data Reading**:
   - The sensor returns **2 bytes** of humidity data and **1 byte** of CRC checksum.
   - The I2C read sequence reads the data bytes and the checksum byte.
3. **Checksum Validation**:
   - The **CRC-8 checksum** is calculated using polynomial `0x31` to ensure data integrity.
4. **Humidity Conversion**:
   - The raw humidity data is converted to relative humidity percentage using:

$$RH(\%) = \left(\frac{100.0 \times \mathrm{Raw\_Humidity}}{65536.0}\right)$$

This ensures accurate reading and conversion of the humidity data from the sensor.

Let me know if there's anything else you need!

I see that the document was truncated due to it being too large. Fortunately, we are done implementing helper functions. Can you please implement the main() function such that the following special requirements are met?
 - power up and power down between reads. Power up should be called at most every 2 seconds.
 - sample output:
Temperature is 20C (or 68F) with a 40% humidity
Temperature is 22C (or 72F) with a 42% humidity
Temperature is 20C (or 68F) with a 44% humidity

I've updated the `app_main()` function to meet your requirements:

- The sensor is powered up and powered down between each reading.
- The power-up interval is maintained at 2 seconds to ensure it's called at most every 2 seconds.
- The output format is consistent with your requested sample.

Let me know if there's anything else you'd like to adjust or if you need further help!

According to the datasheet, it takes about 10-15ms for the measurements to complete. Do the reading functions already account for that, considering we're using the clock stretching enabled commands, or do I need to manually slow down the program to give the sensor time?

The current `read_temperature()` and `read_humidity()` functions do **not** include an explicit delay to account for the sensor measurement time (10-15 ms as specified in the datasheet). While the I2C protocol and the commands you use (which support clock stretching) **should** handle this by pausing until the sensor is ready, it's often recommended to manually introduce a small delay to ensure reliability. This is because some implementations or configurations of I2C might not properly account for clock stretching.

How to Address the Measurement Time

To ensure that your program provides enough time for the sensor to complete its measurements, you can add a delay of around 15 ms after sending the measurement command, but before attempting to read the data:

1. **Add a Delay After Sending the Measurement Command**:
    - After sending the temperature or humidity command (i.e., after `i2c_master_cmd_begin()`), add a delay using `vTaskDelay(pdMS_TO_TICKS(15))` to allow enough time for the sensor to finish its measurement.
2. **Updated `read_temperature()` Function** (example for temperature):

```c
// Send the temperature measurement command
// ... (same code as before)

esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, pdMS_TO_TICKS(1000));
i2c_cmd_link_delete(cmd);

if (ret != ESP_OK) {
    ESP_LOGE(TAG, "Failed to send temperature measurement command");
    return 0.0;
}

// Wait for the measurement to complete (10-15 ms delay)
vTaskDelay(pdMS_TO_TICKS(15));

// Read 3 bytes: 2 data bytes + 1 checksum byte
// ... (same code as before)
```

This modification ensures that the sensor has enough time to produce a stable reading before attempting to collect the data.

Uploaded image

I'm running into an issue with the import declaration. It's worth mentioning the environment I'm working on is WSL2

The error you're encountering, `"fatal error: driver/i2c.h: No such file or directory"`, means that the compiler is unable to find the header file `driver/i2c.h`, which is part of the **ESP-IDF framework**.

Here are a few possible reasons and solutions for this issue:

1. ESP-IDF Path Configuration

Ensure that the **ESP-IDF** environment variables are properly set up. The `driver/i2c.h` file is part of ESP-IDF's drivers, so the compiler needs to know where to find these files.

- Run the command:

  ```bash
  source $HOME/esp/esp-idf/export.sh
  ```

  Replace `$HOME/esp/esp-idf` with the path where your ESP-IDF installation is located. This script sets up the required environment variables for the compiler to locate ESP-IDF components.

2. Include Paths in CMakeLists.txt

Verify that your `CMakeLists.txt` is correctly configured to include the ESP-IDF components. Your `CMakeLists.txt` file should look like:

In the root directory (`lab2_2/CMakeLists.txt`):

```cmake
cmake_minimum_required(VERSION 3.16)

include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(lab2_2)
```

In the `main` directory (`lab2_2/main/CMakeLists.txt`):

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

Ensure that the ESP-IDF path is included so the build system can locate the necessary driver headers.

3. Install ESP-IDF Tools and Dependencies

If you're running this in a new WSL2 environment, you might need to install the ESP-IDF tools and dependencies properly.

- Navigate to the ESP-IDF directory and run:

  ```bash
  ./install.sh
  ```

  Then set up the environment:

  ```bash
  source export.sh
  ```

4. CMake Configuration

Make sure to configure the project using `idf.py` before building:

```bash
idf.py set-target esp32c3
idf.py menuconfig
```

These commands will help ensure your project is correctly configured for the ESP32C3 target.

5. Missing Driver Component

The error might also be caused if the **I2C driver component** is not included during the build. To fix this, ensure that the required component is enabled in `menuconfig`:

- Run:

  ```bash
  ```

```
idf.py menuconfig
```

- Navigate to **Component Config** > **ESP32-specific** > **Driver** and make sure that the **I2C driver** is enabled.

Summary:

- Make sure the ESP-IDF environment is correctly sourced using `source export.sh`.
- Verify that `CMakeLists.txt` includes the correct paths for ESP-IDF.
- Use `idf.py` to configure and build your project, making sure all components are included.

Try these solutions, and let me know if you're still facing issues. I'm here to help further if needed!