ESP32C3 LCD Library Adaptation

lab3.md
File

This is the file that describes a lab I need to do. I already did part 3.1. Please only focus on part 3.2 for now. Please go through the lab document, determine the goals and requirements to complete 3.2, and let me know how I should go about completing the lab. If you have any questions about hardware, software, or any other details that should be clarified for you to give me a thorough, complete answer, please ask.

Based on the lab document, the goal of **part 3.2** is to create a C++ library for the ESP32C3 that mimics the functionality of the DFRobot RGB LCD library but uses ESP32-specific calls. The task is to adapt an existing library that works for Arduino and reimplement it for ESP32C3, respecting the same API structure but with updated internals.

Goals and Requirements of Part 3.2:

1. **Adapt the DFRobot_RGBLCD1602 Library:**
   - Clone the library from DFRobot GitHub.
   - The original library is for Arduino, but you need to adapt it for ESP32C3 by:
     - Keeping the same API (`DFRobot_LCD.h`).
     - Completely reworking the internals to use ESP32-specific calls.
2. **Make Necessary Changes:**
   - **Print Removal:** Remove references to `Print` .
   - **Wire Replacement:** Replace the `Wire` library with the ESP32 equivalent (I2C communication for ESP32).
   - Respect the C++ language requirements since this library should be compatible with the ESP32 C++ environment.
3. **Display Content Requirement:**
   - Your final library should be able to print specific strings on the LCD:
     - **First Line:** Display "Hello CSE121!"

    - **Second Line:** Display your last name.

4. **App Main Example:**
    - Implement the logic provided in the sample code in the main loop. It should be similar but with your modifications, including your last name.

Steps to Complete Part 3.2:

1. **Clone the Library:**
    - Start by cloning the library from the GitHub repository: `https://github.com/DFRobot/DFRobot_RGBLCD1602`.
    - Download it to your development environment.
2. **Convert to ESP32-Compatible Library:**
    - Convert the library code to be compatible with ESP32C3.
    - **Remove `Print` Dependency:** Search for any references to `Print` in the library and remove or replace them, as per the lab requirements.
    - **Replace `Wire` with ESP32 I2C Functions:**
        - The Arduino `Wire` library is commonly used for I2C communication. You need to replace it with the equivalent I2C communication calls used in the ESP32 SDK.
        - Refer to the ESP-IDF documentation for I2C functions. Commonly, you'll need to include the header `<driver/i2c.h>` and use the `i2c_master_*` functions to initiate communication.
3. **Adhere to the Original API:**
    - Retain the function signatures and interface as described in `DFRobot_LCD.h`. The goal is to make the external interface of your library identical to the original so that other parts of your code that depend on this API can use your adapted library seamlessly.
    - Pay attention to functions like `init()`, `setRGB()`, `setCursor()`, and `printstr()`, ensuring they work with the new ESP32-compatible implementation.
4. **Test the Implementation:**
    - Make sure your new library can perform the following when implemented:
        - **Initialize the LCD:** Call `lcd.init()`.
        - **Set RGB and Print Text:**

- Use `lcd.setRGB("Hello CSE121!")` to print the first line.
- Use `lcd.setCursor(0, 1)` and `lcd.printstr("<your last name>")` to print your last name on the second line.

5. **Create App Main Loop:**
   - Your main loop should include initialization and calls to your functions.
   - Follow the given pseudocode:

```c
while (true) {
  lcd.init();
  lcd.setRGB("Hello CSE121!");
  lcd.setCursor(0, 1);
  lcd.printstr("Sidor"); // Replace "Renau" with your last name
}
```

   - Ensure the display works properly when powered and connected to the ESP32.

Questions for Clarification:

1. **Hardware Setup:**
   - How is your ESP32C3 set up for I2C communication with the LCD? I2C usually requires two pins for data (SDA) and clock (SCL). It would be helpful to know the pins used.
2. **Development Environment:**
   - Are you using the ESP-IDF for development, or are you working in another environment like the Arduino IDE? This affects which libraries and functions you'll have available.
3. **Existing Code or Changes:**
   - Do you have any portions of the code already implemented that I should be aware of? For example, are there any specific ESP32 functions you've used for I2C that I should align with?

Let me know if you need further assistance with any specific part of the lab or if you have questions about the steps provided.

I'm currently developing on WSL2 and using ESP-IDF. I have only set up the project directory for lab3 so I haven't started anything yet. The only code I have to go off of is what I have from the arduino library.

Thanks for the details, Sebastian! Here's a more specific breakdown of how you can proceed using WSL2 and ESP-IDF:

Steps for ESP32 Development with ESP-IDF on WSL2:

1. **Set Up ESP-IDF and Project Directory:**
    - Since you're using WSL2 and already have ESP-IDF, make sure your environment variables are set correctly and the ESP-IDF environment is sourced.
    - Your project directory for Lab 3 should have a structure like this:

```css
lab3/
├── components/
│   └── DFRobot_RGBLCD1602/
│       ├── DFRobot_LCD.cpp
│       ├── DFRobot_LCD.h
└── main/
    ├── main.cpp
    └── CMakeLists.txt
```

    - Place the cloned Arduino library files inside the `components` directory. You'll be modifying `DFRobot_LCD.cpp` and `DFRobot_LCD.h`.
2. **Porting the Arduino Library to ESP32:**
    - **Remove Print Dependencies:**
        - In the Arduino library, `Print` is a common base class used for printing text. Since ESP32 does not require `Print`, you can safely remove references to it.
        - Identify the use of `Print` in `DFRobot_LCD.cpp` and remove or modify the code to call native C++ methods for handling strings.
    - **Replace Wire with ESP-IDF I2C Functions:**
        - Instead of using the Arduino `Wire` library, you'll need to implement I2C communication using ESP-IDF functions.

- Include the header file for I2C:

  ```cpp
  #include "driver/i2c.h"
  ```

- Replace `Wire.begin()`, `Wire.write()`, and other Arduino I2C calls with ESP-IDF functions:

  - **Initialize I2C:** Create a function to initialize the I2C bus, which includes specifying SDA and SCL pins, clock speed, etc.

    ```cpp
    void i2c_master_init() {
      i2c_config_t conf;
      conf.mode = I2C_MODE_MASTER;
      conf.sda_io_num = GPIO_NUM_21;  // Use appropriate GPIO pins for SDA
      conf.scl_io_num = GPIO_NUM_22;  // and SCL based on your setup
      conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
      conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
      conf.master.clk_speed = 100000;  // Standard 100kHz clock speed
      i2c_param_config(I2C_NUM_0, &conf);
      i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
    }
    ```

  - **Writing Data:** For writing data, use `i2c_master_write()` or similar functions from ESP-IDF. You will need to handle start/stop conditions explicitly, unlike in Arduino.

    ```cpp
    void i2c_write(uint8_t address, uint8_t *data, size_t len) {
      i2c_cmd_handle_t cmd = i2c_cmd_link_create();
      i2c_master_start(cmd);
      i2c_master_write_byte(cmd, (address << 1) | I2C_MASTER_WRITE, true);
      i2c_master_write(cmd, data, len, true);
      i2c_master_stop(cmd);
      i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000 / portTICK_RATE_MS);
    ```

```
  i2c_cmd_link_delete(cmd);
}
```

    - Replace every instance where the Arduino `Wire` library is used with a call to this I2C function in `DFRobot_LCD.cpp`.

3. **Maintain the Original API:**
    - The idea is to keep the API unchanged so that functions like `lcd.init()`, `lcd.setRGB()`, `lcd.setCursor()`, and `lcd.printstr()` work seamlessly.
    - You may need to create ESP32-compatible equivalents for these methods in `DFRobot_LCD.cpp`.

4. **Testing and Debugging:**
    - **Compile and Flash the Code:**
        - Make sure your `CMakeLists.txt` in the `main` directory properly includes the component. You need to make sure ESP-IDF knows where to find the LCD library.
        - Run `idf.py build` and `idf.py flash` to compile and flash your project to the ESP32.
    - **Debugging:**
        - Use `idf.py monitor` to see serial output and verify that the messages are being printed correctly on the LCD.
        - If the LCD isn't displaying correctly, verify the I2C wiring (SDA/SCL pins) and make sure there are no issues with the initialization process.

5. **Main Loop Implementation:**
    - Follow the provided example in the lab document:

```cpp
extern "C" void app_main() {
  // Assuming lcd is already properly instantiated
  lcd.init();
  while (true) {
    lcd.setRGB("Hello CSE121!");
    lcd.setCursor(0, 1);
    lcd.printstr("Sidor");
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Add a delay to prevent
overwhelming the display
  }
}
```

- Modify the `lcd` class methods in your C++ library file so that the `init()`, `setRGB()`, `setCursor()`, and `printstr()` methods all call the ESP32 equivalents you've created.

By following these steps, you should be able to adapt the Arduino library to work with the ESP32C3, maintain compatibility with the original API, and achieve the functionality required for the lab. Let me know if you need more help on a specific step, or if you encounter any issues while coding!

I'm currently trying to wire up the LCD display to the ESP32C3. The LCD is a DFRobot LCD1602. What pins on the ESP32 should I wire up to the LCD's GND and VCC pins?

For wiring the DFRobot LCD1602 to your ESP32C3, you need to make the following connections for the **GND** and **VCC** pins:

- **GND (Ground) Pin**: Connect this to one of the **GND pins** on your ESP32C3. The ESP32 has multiple GND pins, and you can use any of them.
- **VCC (Power Supply) Pin**: Connect this to the **3.3V pin** on the ESP32C3. The ESP32 typically operates at 3.3V, so this will safely power the LCD.

Specifically, you can use the following pins on the ESP32C3:

1. **VCC to 3.3V**:
    - Use the **3V3 pin** (usually labeled as 3.3V or 3V3) on the ESP32C3 to supply power to the VCC pin on the LCD. Ensure you don't connect it to the 5V pin, as the ESP32C3's GPIO operates at 3.3V logic levels.
2. **GND to GND**:
    - Connect the GND pin on the LCD to any of the **GND pins** on the ESP32C3.

This setup ensures that the LCD is powered correctly, and the ESP32 will be able to communicate with the LCD effectively without any risk of damaging the components.

Let me know if you need any help with the I2C connections (SDA/SCL pins) or anything else related to your setup!

Uploaded image

The error message in the output suggests there is a failure in setting certain I2C registers on the LCD. The error code `ESP_FAIL` indicates that the ESP32 is not able to successfully write data to the LCD module over the I2C bus. Here are some possible causes and how you can troubleshoot them:

Troubleshooting Steps:

1. **Check I2C Wiring:**
    - Ensure that the **SDA (Data Line)** and **SCL (Clock Line)** are properly connected between the ESP32 and the LCD.
    - Common pins for the ESP32C3 are **GPIO21** (SDA) and **GPIO22** (SCL). Make sure they are securely connected and double-check if these are correctly assigned in your code.
    - Verify that **pull-up resistors** are in place (typically 4.7kΩ or 10kΩ) between the SDA and SCL lines and the 3.3V power. Many I2C modules have these resistors already integrated, but if not, you will need to add them.
2. **Correct I2C Address:**
    - Make sure you are using the **correct I2C address** for your LCD. The default address for DFRobot LCD1602 is usually **0x27** or **0x3F**. You can scan the I2C bus using an I2C scanner sketch to confirm the address of your device.
    - If you're unsure, you can try using an I2C scanner to discover the address of your LCD.
3. **Power Supply Issues:**
    - The **power supply** might be insufficient for the LCD. Ensure you are providing a stable **3.3V** from the ESP32.
    - If possible, try powering the LCD from an external 5V source while ensuring that the I2C lines are correctly level-shifted to 3.3V to match the ESP32 logic levels.
4. **Software Configuration:**
    - Make sure you initialized the I2C bus properly in your ESP-IDF code.

   - Here's an example of properly configuring I2C:

     ```c
     i2c_config_t conf;
     conf.mode = I2C_MODE_MASTER;
     conf.sda_io_num = GPIO_NUM_21;   // Your SDA pin
     conf.scl_io_num = GPIO_NUM_22;   // Your SCL pin
     conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
     conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
     conf.master.clk_speed = 100000;   // 100kHz
     i2c_param_config(I2C_NUM_0, &conf);
     i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
     ```

   - Ensure that the `i2c_master_cmd_begin` function, which actually performs the communication, is used with an appropriate **timeout value**. If the timeout is too short, communication might fail.

5. **Check the LCD's I2C Module:**
   - Some versions of the LCD1602 have an **I2C backpack** that might not be properly seated or might have cold solder joints. Check the solder joints between the LCD and its I2C backpack module.
   - It might also help to reflow the solder joints if they appear inconsistent or if there are signs of weak connections.

6. **Verify Permissions and Addressing in Your Code:**
   - Make sure you are passing the correct arguments when using `i2c_master_write_byte()` or `i2c_master_write()`. It should match the LCD’s I2C requirements.
   - Sometimes `ESP_FAIL` might indicate that the **target device is not acknowledging** the communication. This could mean the LCD is not responding on the address you're using.

7. **Debugging with Logging:**
   - Add additional logging in your code to understand which part of the initialization or write process is failing.
   - It can be helpful to separate the initialization step and the actual data writing step to isolate whether the issue lies with setting up I2C or with communicating specific data.

Suggested Code Snippet for I2C Write:

If you are trying to write a value to a specific register, you should do something like:

```c
i2c_cmd_handle_t cmd = i2c_cmd_link_create();
i2c_master_start(cmd);
i2c_master_write_byte(cmd, (LCD_ADDRESS << 1) | I2C_MASTER_WRITE, true); // Replace LCD_ADDRESS with the actual address of your LCD
i2c_master_write_byte(cmd, register_addr, true); // Write the register address
i2c_master_write_byte(cmd, value, true);         // Write the data value
i2c_master_stop(cmd);
esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000 / portTICK_RATE_MS);
i2c_cmd_link_delete(cmd);

if (ret != ESP_OK) {
    ESP_LOGE(TAG, "Error setting register - addr: 0x%02x, data: 0x%02x, error: %d", register_addr, value, ret);
}
```

Replace `LCD_ADDRESS`, `register_addr`, and `value` as needed for your application. This should help you narrow down which specific I2C transaction is failing.

Let me know if any of these suggestions help, or if you need more specific guidance on implementing one of these fixes.