Group 3: Mong Phan, Sami Barbaglia, Xuan Dang

Greenhouse CO2 controller

ARM Processors and Embedded Operating Systems Project Report

Third-year Hardware Project

School of ICT

Metropolia University of Applied Sciences

13 October 2024

Contents

1 Introduction

This project focuses on developing a CO2 fertilization controller for greenhouse environments, designed to maintain optimal CO2 levels to promote healthy plant growth. By controlling CO2 levels, temperature, humidity, and air pressure, the system helps optimize the conditions within the greenhouse to maximize plant productivity.

Additionally, a Modbus-controlled fan adjusts ventilation based on the current CO2 levels, while a solenoid valve controls CO2 injection. All settings and parameters are stored in EEPROM to ensure persistence across reboots.

Moreover, the project will feature a local user interface for setting parameters, and advanced features will include cloud reporting through ThingSpeak, as well as remote control capabilities via the ThingSpeak TalkBack queue. Data such as CO2 levels and fan speed are sent to the cloud, where they can be viewed and analyzed.

This document aims to provide a comprehensive project overview by delving into various key aspects. Chapter 2 explores the theoretical background, methods, and materials employed in the project, followed by a thorough examination of the implementation process in Chapter 3. Finally, Chapter 4 concludes the project's outcomes.

Figure 1. below displays the miniature test system, which we mostly worked on for the ventilation project. Figure 2. shows the full-scale test system used in the final demonstration.

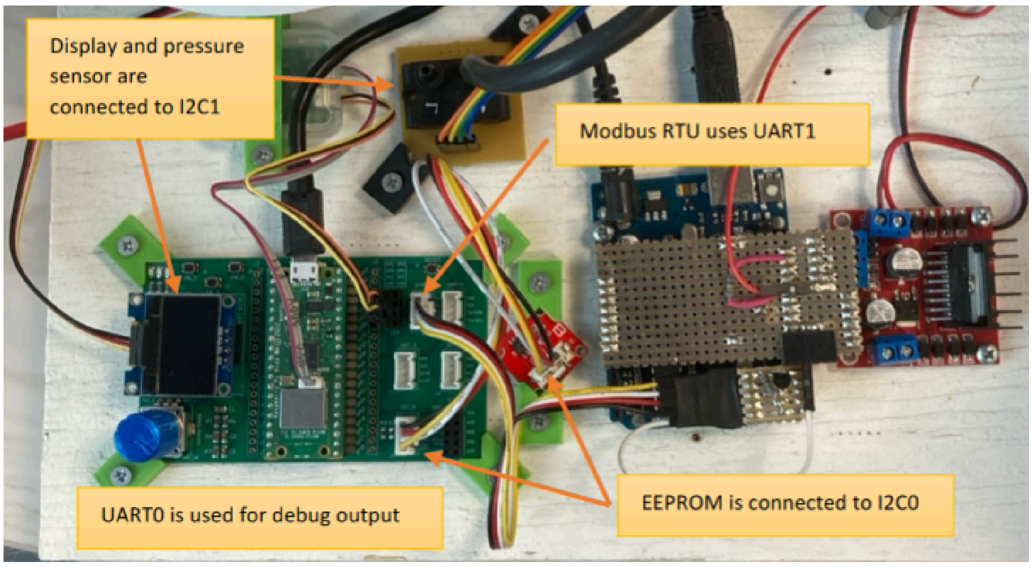
Display and pressure sensor are connected to I2C1
Modbus RTU uses UART1
UART0 is used for debug output
EEPROM is connected to I2C0

Figure 1: Miniature test system [1.]

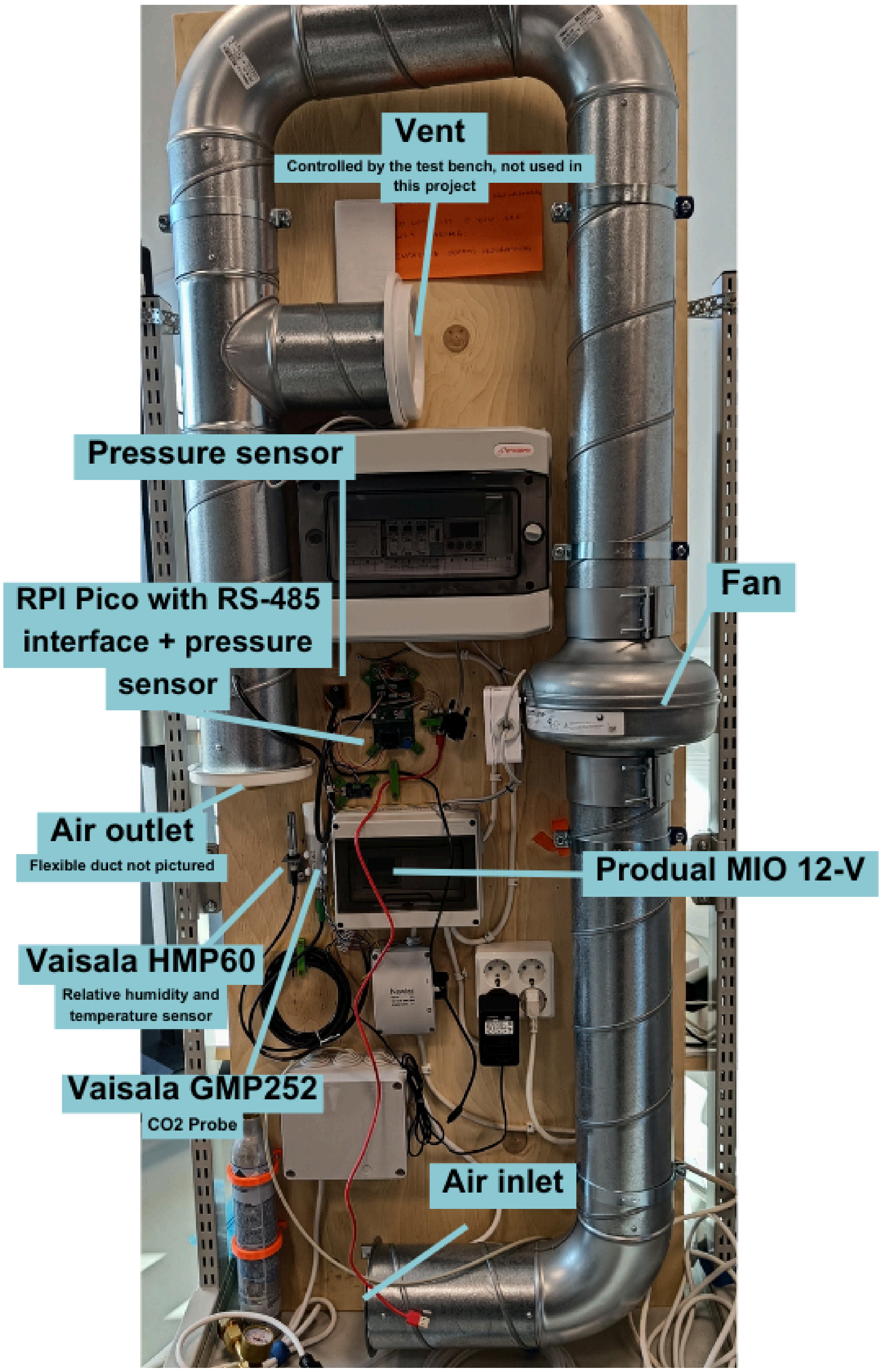
Vent
Controlled by the test bench, not used in this project
Pressure sensor
RPI Pico with RS-485 interface + pressure sensor
Fan
Air outlet
Flexible duct not pictured
Produal MIO 12-V
Vaisala HMP60
Relative humidity and temperature sensor
Vaisala GMP252
CO2 Probe
Air inlet

Figure 2: Full-scale test system.

2 Methods and Materials

This chapter details the hardware, software, and protocols used in the project. The hardware section lists all the physical components of both the miniature and full-size systems. The software and protocols section covers communication protocols like Modbus RTU and I2C, as well as cloud and API integration.

Figure 3. below illustrates the system diagram of the ventilation project.

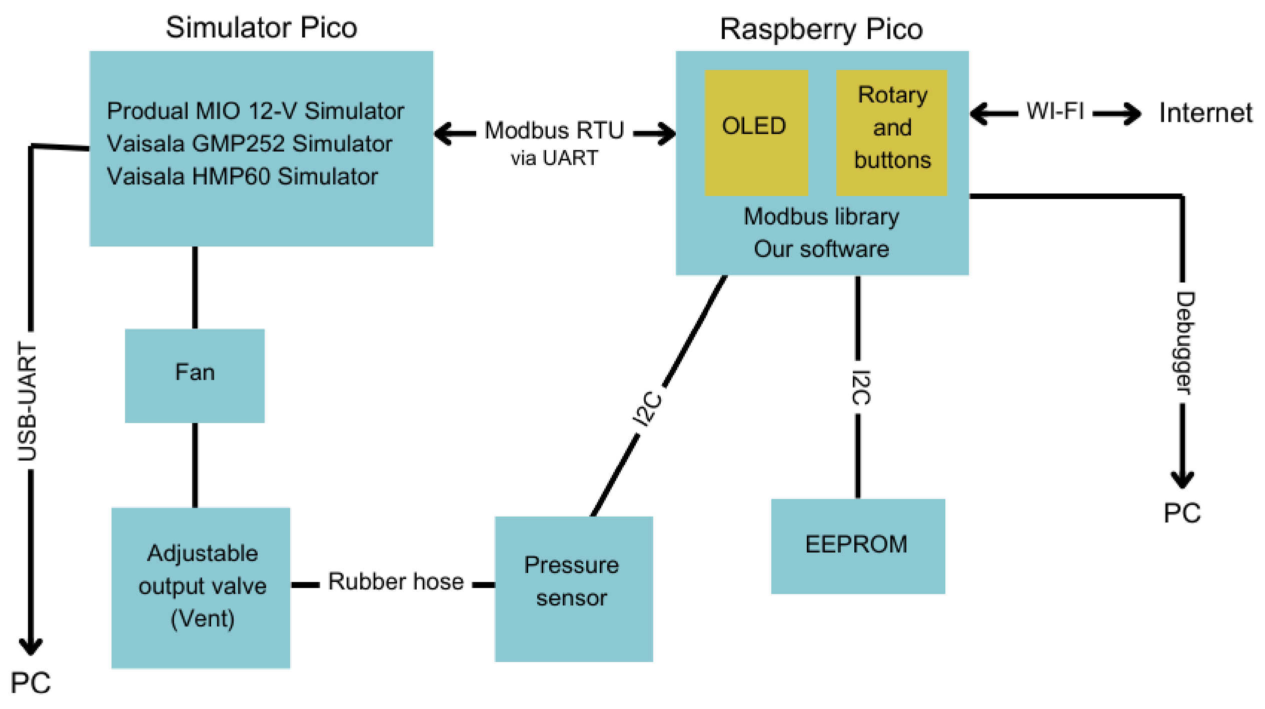

Figure 3: Miniature test system diagram.

2.1 Hardware components

This section goes over the hardware components, including the Produal MIO 12-V, OLED display, microcontroller, sensors, and EEPROM. It also covers the Raspberry Pi Pico W's GPIOs used in the project.

Table 1. lists the hardware components used in both the full-scale and miniature test systems, followed by a short description. Table 2. shows the Pico W's GPIO assignments, also followed by a short description.

Component	Description
Produal MIO 12-V	Modbus controlled IO-device. The voltage output is connected to the fan speed input, controlling the speed of the fan. Implemented a miniature test system by Pico W.
Vaisala GMP252	Carbon dioxide (CO2) Probe. Simulated by Pico W in the miniature test system.
Vaisala HMP60	Relative humidity and temperature sensor. Simulated by Pico W in the miniature test system.
SSD1306 OLED Display	0.96” with 128x64 pixels monochrome OLED based on SSD1306 drivers. Controlled over the I2C protocol. Enables users to view the status of the system and provides a local UI for manual interaction.
Raspberry Pi Pico W	The core microcontroller (heart of the system) that runs FreeRTOS and simulates the Modbus client in the miniature test system.
RPI Pico with RS-485 Interface	Controls the interface system in the full-size test bench.
Raspberry Pi Debug Probe	All-in-one USB-to-debug kit (Arm Serial Wire Debug), works with OpenOCD. Makes debugging possible without soldering etc.
EEPROM	Crowtail I2C EEPROM module used for storing Wi-Fi credentials. The CO2 setpoint is recalled from the EEPROM when booting the system.
Fan	A 12V-DC fan controlled through the Produal MIO 12-V.
Injection Valve	A DC valve that enables users to inject more CO2 into the Greenhouse from a CO2 container.

Table 1. Hardware materials used in the project.

Name	GPIO pins	Description/Feature
back_btn	**7**	BACK button used for return the SELECTION screen on local UI interaction
ok_btn	**9**	OK button used for confirming a selected option on local UI

valve	**27**	Controls the CO2 container valve to inject more CO2 into the Greenhouse
rotA	**10**	Pin A of a rotary encoder for local UI navigation. It is used together with rotB to determine the rotation direction: clockwise/counter-clockwise
rotB	**11**	Pin B of a rotary encoder for local UI navigation. It is used together with rotB to determine the rotation direction: clockwise/counter-clockwise

Table 2. GPIO assignments.

2.2 Software and Protocols

This section goes over the software and protocols used in the project. It covers the programming language C++, FreeRTOS, as well as the protocols used in the software, such as I2C, UART, Modbus RTU, and RESTful API.

Table 3. below lists the key software components and features, followed by a short description.

Component	Description
C++ / CLion	Programming language / IDE, specialized for Embedded programming.
Modbus RTU	Modbus Remote Terminal Unit is a serial communication protocol designed for industrial automation systems. Facilitates data exchange between various devices, such as sensors, actuators, and controllers, connected to a shared communication channel.
FreeRTOS	FreeRTOS is a real-time operating system kernel for embedded devices that has been ported to 40 microcontroller platforms.
I2C	I2C is a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL). The protocol supports multiple target devices on a communication bus and can also support multiple controllers that send and receive commands and data.
RESTful API	An architectural style for an application programming interface (API) that uses HTTP requests to access and use

	data. That data can be used to GET, PUT, POST and DELETE data types, which refers to reading, updating, creating and deleting operations related to resources.
ThingSpeak Cloud Service	An IoT analytics platform service from MathWorks, that allows users to aggregate, visualize, and analyze live data streams in the cloud.
UART	Universal Asynchronous Receiver-Transmitter (UART) is a communication protocol used for debug output and device-to-device communication.

Table 3. Software materials used in the project.

2.2.1 Protocols

The Modbus RTU is used to communicate with Modbus-enabled Produal MIO 12-V and Vaisala sensors (GMP252, HMP60), enabling real-time communication between the devices. I2C is used to communicate with the Sensirion SDP610 pressure sensor, OLED screen, and EEPROM. It allows these devices to share data with the microcontroller.

2.2.2 Cloud and API

The CO2 ventilation system connects to the cloud using a RESTful API. This architecture transmits the environmental data (CO2 levels, humidity, etc.) to the ThinkSpeak Cloud Service, enabling remote monitoring and control. Through ThinkSpeak, users can store and visualize the data, and also remotely control the CO2 setpoint.

3 Implementation

The project is designed to control a ventilation system by managing parameters such as CO2 levels, temperature, humidity, and fan speed. The software is built using FreeRTOS, which provides a multitasking environment with tasks for managing CO2 setpoints, Wi-Fi configuration, data exchange with a cloud service (ThingSpeak), interaction with a rotary encoder and buttons for user input, and displaying the local UI through an LCD screen.

Key Features:

- **CO2 Monitor and Control**: Adjusts fan speed and/or CO2 levels based on measured CO2 levels and the CO2 setpoint.
- **Modbus Communication**: Reads environmental data (CO2, temperature, humidity) using Vaisala's sensors.
- **Wi-Fi Configuration**: Allows users to configure Wi-Fi credentials via a local UI.
- **Cloud Communication**: Sends and receives data from ThingSpeak at regular intervals using a RESTful API.
- **Rotary Encoder & Button Inputs**: Enables user interaction for navigating the system's UI.
- **Local UI**: Allows users to monitor system data and set up parameters, such as CO2 levels and Wi-Fi credentials.

Task-to-Task Communication:

- The `WIFI_CHANGE_BIT_EEPROM` event bit is set in the **GPIO Task** when new Wi-Fi credentials are confirmed. The **EEPROM Task** will store the new credentials in the EEPROM if this bit is set.
- The `CO2_CHANGE_BIT_EEPROM` event bit is set when a new CO2 setpoint is confirmed through the local UI (via the **GPIO Task**) or ThingSpeak cloud (via the **TLS Task**). The **EEPROM Task** will store the new CO2 setpoint in the EEPROM if this bit is set.

- The `WIFI_CHANGE_BIT_TLS` event bit is set in the **EEPROM task** to notify the **TLS Task** to make a new Wi-Fi connection using the new credentials confirmed through the local UI.
- The `CO2_CHANGE_BIT_LCD` event bit is used to update the new setpoint data on the LCD screen in the **Display Task** whenever a new CO2 setpoint is confirmed through the **TLS Task**, or to update new CO2 level data on the LCD whenever a new CO2 level is read from the CO2 sensor in the **MODBUS Task**.
- The **GPIO Task** communicates with the **Display Task** using two queues: `wifiCharPos_q` and `gpio_data_q`. The `wifiCharPos_q` queue stores the current character and its position to be displayed on the LCD screen when updating Wi-Fi credentials. The `gpio_data_q` queue stores data related to GPIO actions, such as the direction of rotary movement (clockwise or counterclockwise) or button presses, as well as the screen where the action occurred. The **GPIO Task** sends data to the `gpio_data_q` queue whenever a GPIO action takes place, allowing the **Display Task** to update the LCD screen with the correct information.

3.1 EEPROM task

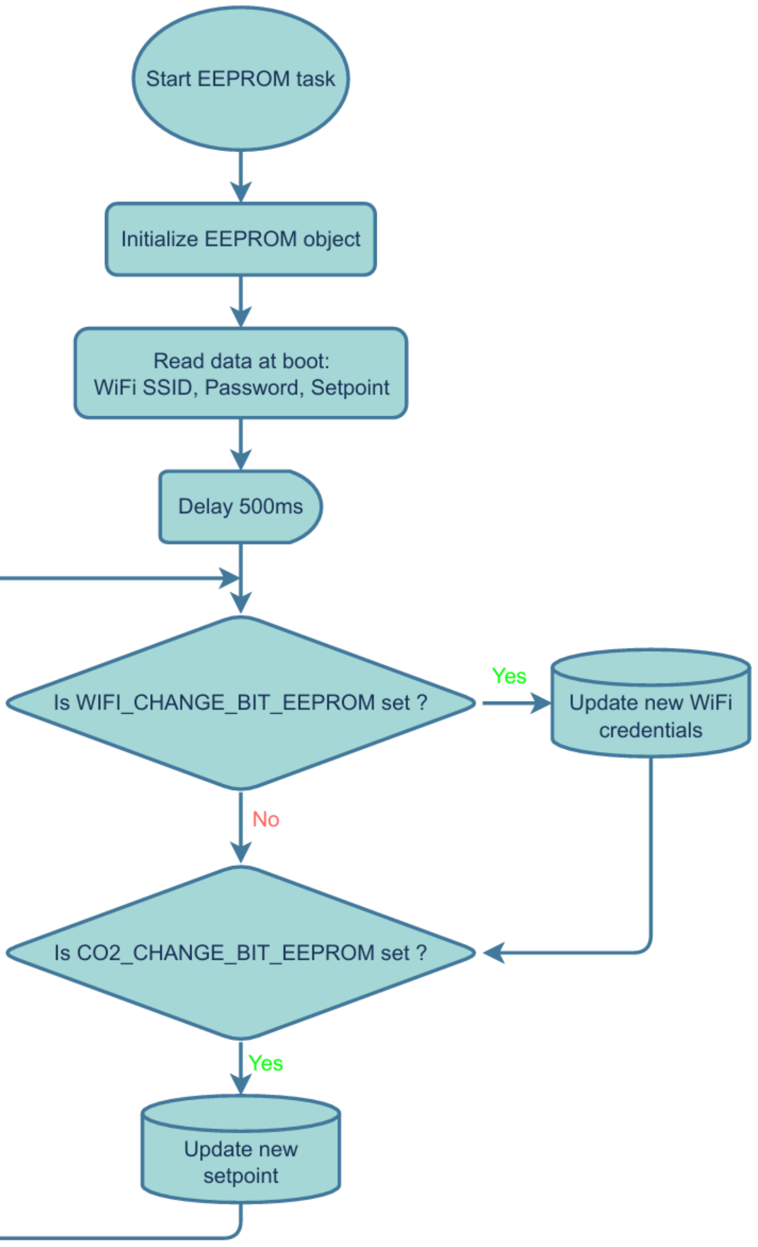
Start EEPROM task
Initialize EEPROM object
Read data at boot: WiFi SSID, Password, Setpoint
Delay 500ms
Is WIFI_CHANGE_BIT_EEPROM set ?
Yes
Update new WiFi credentials
No
Is CO2_CHANGE_BIT_EEPROM set ?
No
Yes
Update new setpoint

Figure 4: EEPROM task flow chart.

The `eeprom_task` is responsible for managing the non-volatile storage of key configuration parameters, such as Wi-Fi credentials and CO2 setpoints, in an EEPROM memory module. This task plays a crucial role in ensuring that these settings persist across system reboots, allowing the ventilation system to operate with minimal user intervention after power cycles or resets. The EEPROM task continuously monitors for changes in system parameters, writing updated data to the EEPROM when necessary and notifying other tasks to act on these changes.

Task Initialization

At the beginning of the task, a shared pointer to a `PicoI2C` object is created to facilitate communication with the EEPROM over the I2C bus. This object handles the low-level operations required to read from and write to the EEPROM. Immediately after initialization, the function `readAtBoot()` is called to retrieve any previously stored settings, ensuring that the system boots with the correct Wi-Fi credentials and CO2 setpoints. Following this, a brief delay (500ms) is introduced to allow the rest of the system to stabilize before the main task loop begins.

Main Task Functionality

The `eeprom_task` operates in a loop, where it checks for updates to two key system parameters: Wi-Fi credentials and CO2 setpoints. The task uses FreeRTOS event groups to monitor for changes, ensuring that it only responds when necessary, thereby optimizing system resource usage.

Handling Wi-Fi Credential Updates

One of the primary responsibilities of this task is to write updated Wi-Fi credentials to the EEPROM. This is triggered when the **WIFI_CHANGE_BIT_EEPROM** event bit is set, indicating that the Wi-Fi credentials have been changed through user input. The task responds by writing the new SSID and password to predefined EEPROM addresses using the `writeEEPROM()` function. Once the credentials have been successfully written, the task sets the **WIFI_CHANGE_BIT_TLS** event bit to signal the **TLS Task** to reconnect to the Wi-Fi network using the new credentials. This event-driven approach minimizes unnecessary polling and ensures timely updates to system configurations.

Handling CO2 Setpoint Updates

The `eeprom_task` also monitors for changes in the CO2 setpoint via the **CO2_CHANGE_BIT_EEPROM** event bit. When this bit is set, it indicates that the CO2 setpoint has been updated either locally, through the user interface, or remotely, via the cloud communication task. Upon detecting this event, the task writes the new CO2 setpoint to the EEPROM. Since the CO2 setpoint is a 16-bit value, it is split into two bytes (high and low) before being written to the EEPROM. This ensures that the system retains the correct setpoint across reboots, contributing to the proper control of the ventilation system.

3.2 TLS task

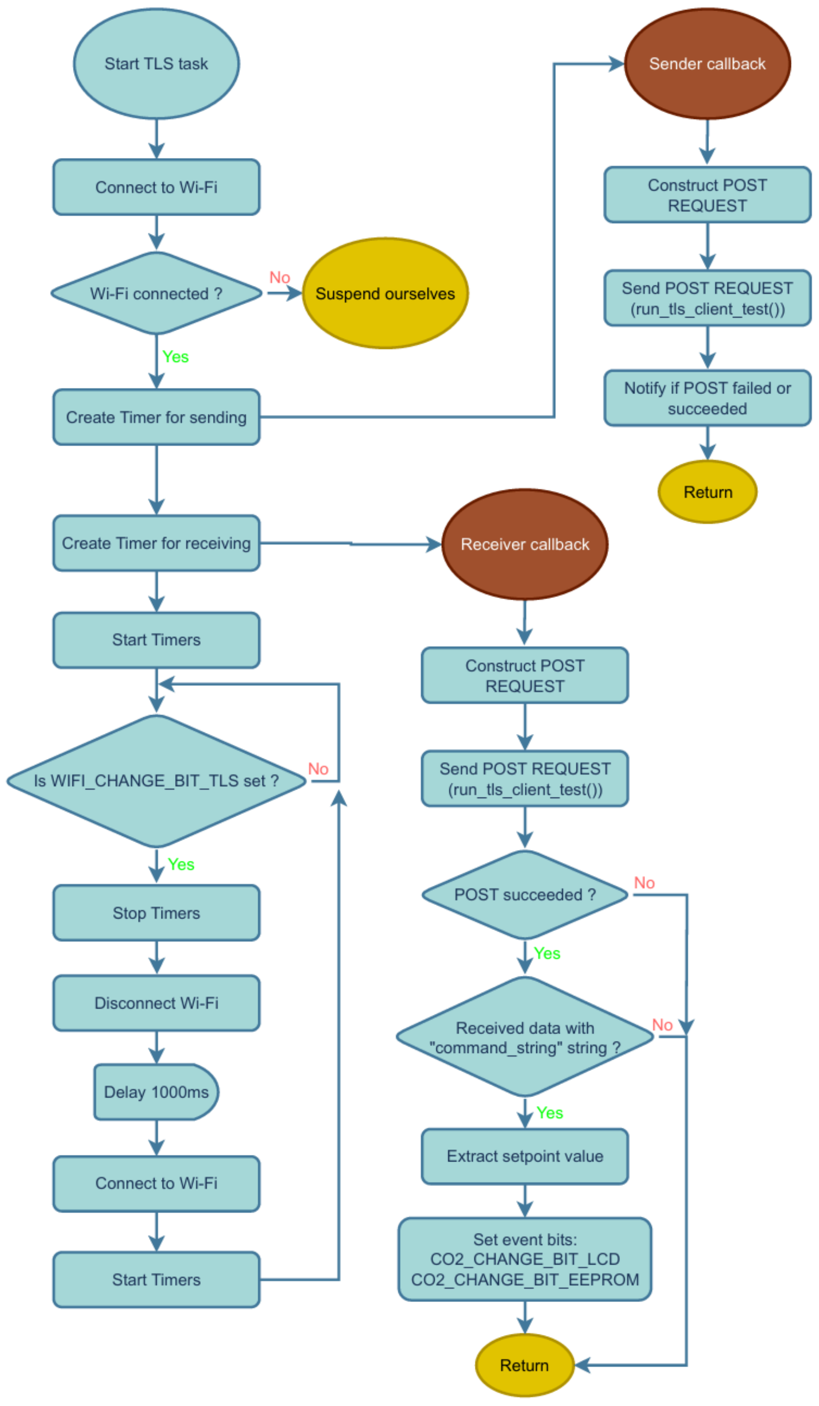

Figure 5: TLS task flow chart.

The `tls_task` is responsible for managing the Raspberry Pi Pico W's connection to a cloud service (ThingSpeak) via Wi-Fi, enabling data exchange between the local system and the cloud. This task establishes and maintains a Wi-Fi connection, periodically sends sensor data to the cloud (every 60 seconds), and checks for updates from the cloud (every 10 seconds). The task also handles reconnection when new Wi-Fi credentials are provided.

Task Initialization and Wi-Fi Connection

The task begins by attempting to establish a Wi-Fi connection, which handles the low-level operations required to initialize the Wi-Fi module and connect to the network. If the connection fails, the task suspends itself to process further operations.

Upon a successful connection, two FreeRTOS software timers, `cloudSender_T` and `cloudReceiver_T`, are created. These timers manage the periodic sending of system data to the cloud and the receiving of any commands from the cloud through their callback functions. The timers are then started to initiate regular communication with the cloud.

Sending Data to the Cloud

The function `sendToCloud()` is used to send sensor data (CO2 level, temperature, humidity, fan speed, and CO2 setpoint) to ThingSpeak via a POST request. This function is triggered by the `cloudSender_T` timer's callback function, `cloudSenderCB()`. The callback ensures that the data is sent at intervals defined by `SEND_INTERVAL` (60 seconds in this case).

The data is sent using the `run_tls_client_test()` function, which manages the secure communication with the ThingSpeak server. If the transmission is successful, a confirmation is printed; otherwise, an error message is logged.

Receiving Data from the Cloud

The task also periodically requests commands from the ThingSpeak TalkBack API using the `receiveFromCloud()` function, which sends a POST request to the API. This request is managed by the `cloudReceiver_T` timer's callback function, `cloudReceiverCB()`. If a new setpoint is received from the cloud, it is placed into the `setpoint_q` queue.

Once a new setpoint is successfully received, the event bits `CO2_CHANGE_BIT_LCD` and `CO2_CHANGE_BIT_EEPROM` are set in the `co2EventGroup`, signaling other tasks (e.g., display and EEPROM) to update the CO2 setpoint on the LCD screen and store the new value in EEPROM, respectively.

Handling Wi-Fi Reconnection

In the main loop, the task listens for any updates in Wi-Fi credentials using the `WIFI_CHANGE_BIT_TLS` event bit in the `wifiEventGroup`. When this event bit is set (usually triggered by the EEPROM task after saving new credentials), the task disconnects from the current Wi-Fi network, stops the cloud communication timers, and then reconnects to the network with the new credentials.

After reconnecting, the timers `cloudSender_T` and `cloudReceiver_T` are restarted, allowing the task to resume its periodic cloud communication.

3.3 Display task

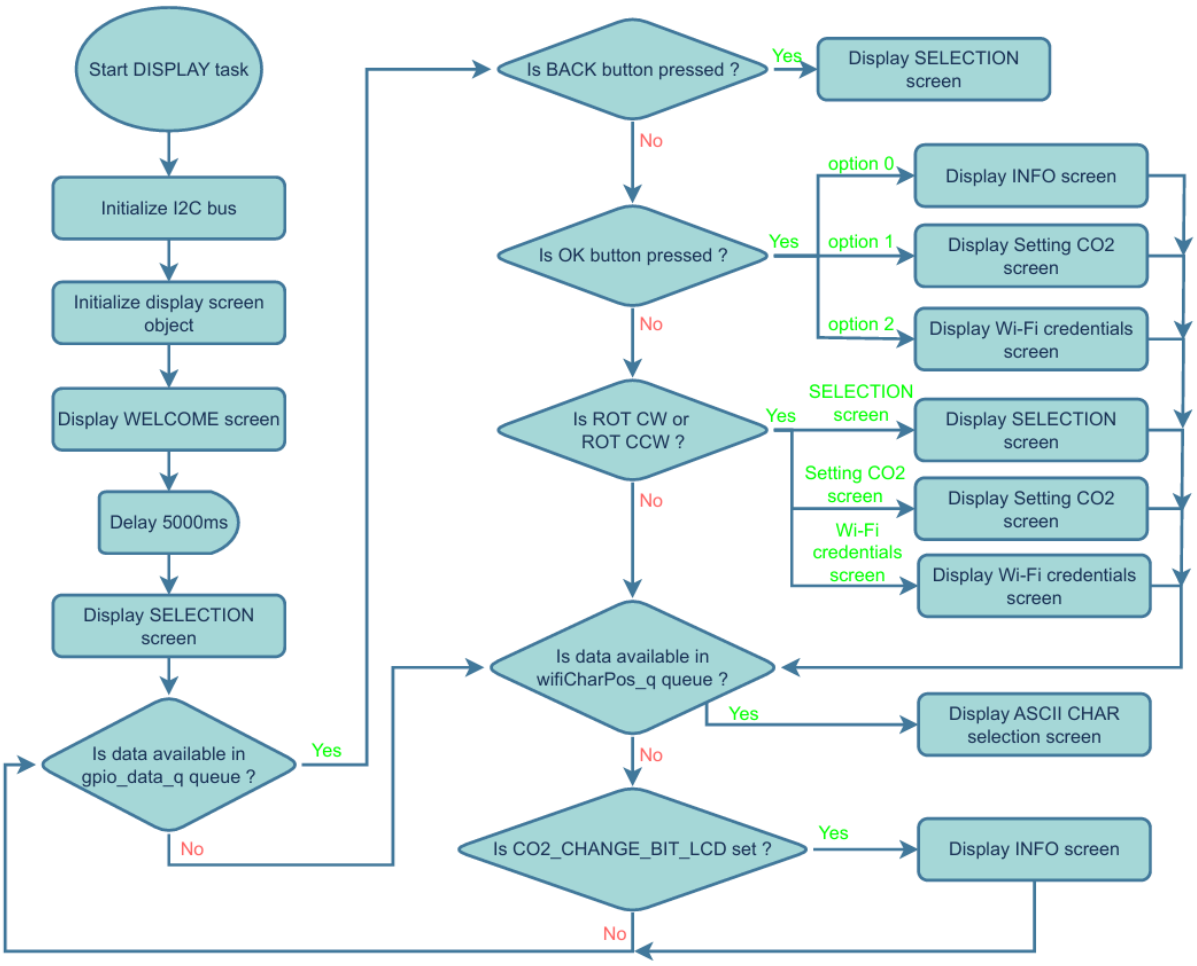

Figure 6: Display task flow chart

The `display_task` is responsible for managing the content shown on the OLED screen and interacts with various components of the system, such as GPIO inputs and event bits, to update the display accordingly.

At the start, the task initializes the I2C bus and display, using the `ssd1306os` driver for the OLED screen. It then calls the `welcome()` function to show a welcome message for 5 seconds, followed by the `screenSelection()` function to allow the user to choose from available options.

The task enters an infinite loop where it listens for updates through two queues, `gpio_data_q` and `wifiCharPos_q`, and also monitors the `co2EventGroup` for any changes in CO2 setpoints that need to be updated on the display. The key operations include:

- Handling GPIO Actions: When data is received from the `gpio_data_q` queue, the task processes various GPIO actions, such as:
  - Back Press: It calls the `screenSelection()` function to return to the main selection screen.
  - OK Press: Depending on the user's current selection (`selection_screen_option`), it either shows system information, the CO2 setpoint adjustment screen, or the Wi-Fi configuration screen.
  - Rotary Actions: When the rotary encoder is turned (clockwise or counterclockwise), the screen is updated to reflect the action based on the current screen type. For example, the CO2 setpoint screen (`SET_CO2_SCR`) or Wi-Fi configuration screen (`WIFI_CONF_SCR`) will update accordingly.
- Wi-Fi Credentials Input: The task listens to the `wifiCharPos_q` queue for characters and positions that need to be displayed when updating Wi-Fi credentials, using the `asciiCharSelection()` function.
- CO2 Setpoint Changes: The task checks for the `CO2_CHANGE_BIT_LCD` event bit in the `co2EventGroup` and updates the display with the latest CO2, temperature, humidity, fan speed, and setpoint values when the bit is set and the user is on the information screen (`INFO_SCR`).

3.4 GPIO task

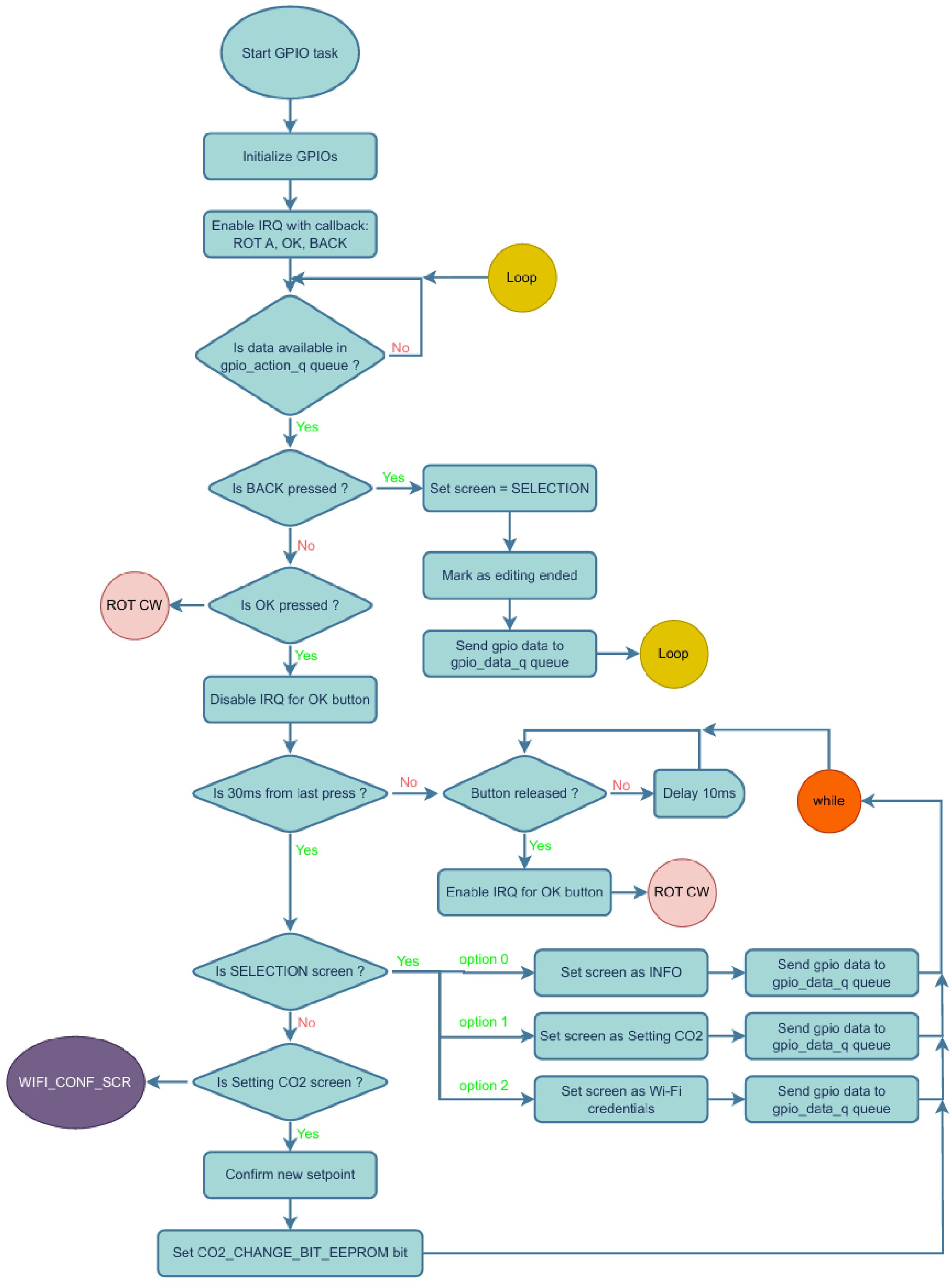
Start GPIO task
Initialize GPIOs
Enable IRQ with callback: ROT A, OK, BACK
Loop
Is data available in gpio_action_q queue ?
No
Yes
Is BACK pressed ?
Yes
Set screen = SELECTION
Mark as editing ended
Send gpio data to gpio_data_q queue
Loop
No
Is OK pressed ?
ROT CW
Yes
Disable IRQ for OK button
Is 30ms from last press ?
No
Button released ?
No
Delay 10ms
while
Yes
Enable IRQ for OK button
ROT CW
Yes
Is SELECTION screen ?
Yes
option 0
Set screen as INFO
Send gpio data to gpio_data_q queue
option 1
Set screen as Setting CO2
Send gpio data to gpio_data_q queue
option 2
Set screen as Wi-Fi credentials
Send gpio data to gpio_data_q queue
No
Is Setting CO2 screen ?
WIFI_CONF_SCR
Yes
Confirm new setpoint
Set CO2_CHANGE_BIT_EEPROM bit

Figure 7a: GPIO task flow chart

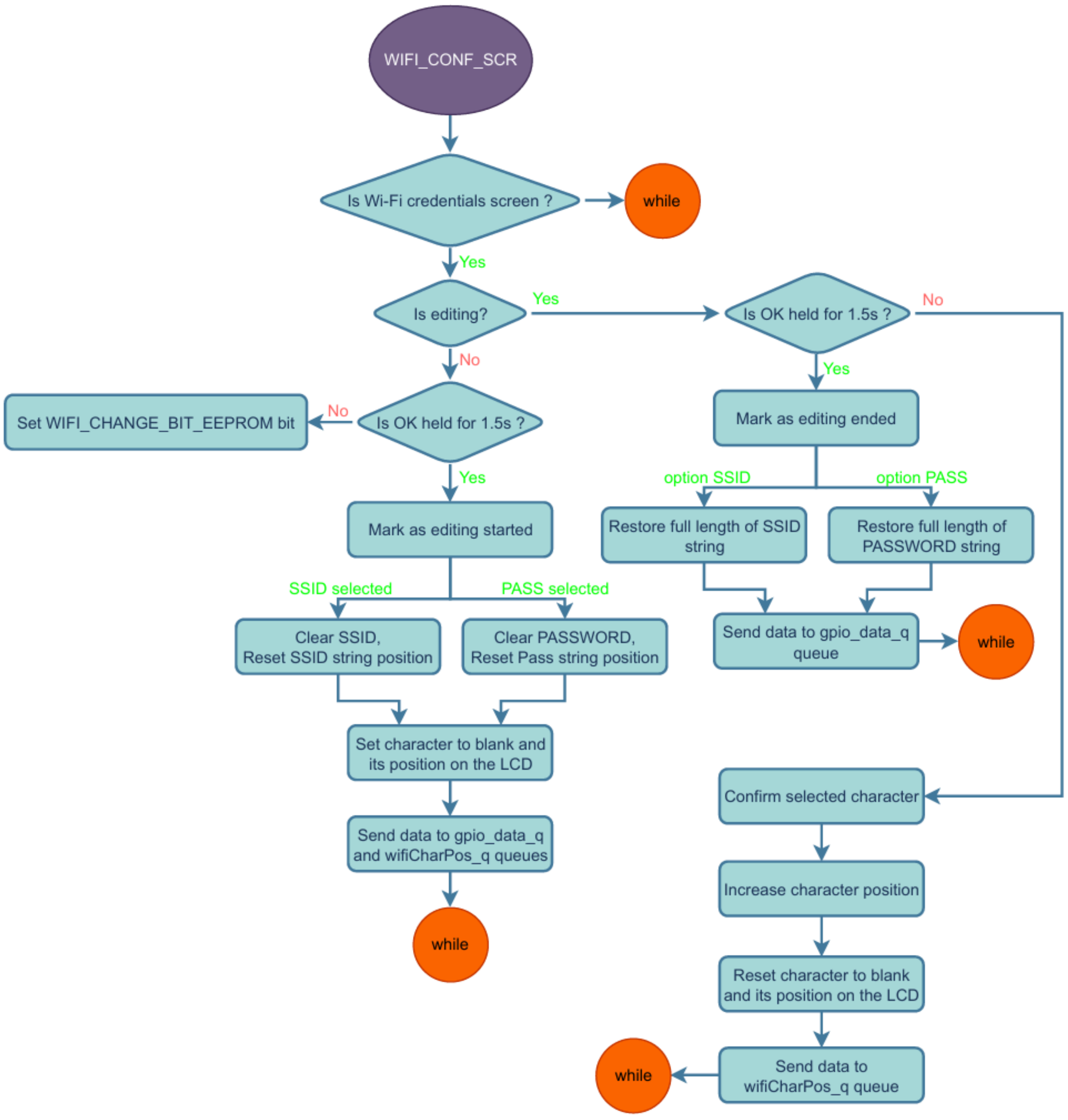
WIFI_CONF_SCR
Is Wi-Fi credentials screen ?
while
Yes
Is editing?
Yes
Is OK held for 1.5s ?
No
No
Yes
Set WIFI_CHANGE_BIT_EEPROM bit
No
Is OK held for 1.5s ?
Mark as editing ended
Yes
option SSID
option PASS
Mark as editing started
Restore full length of SSID string
Restore full length of PASSWORD string
SSID selected
PASS selected
Send data to gpio_data_q queue
while
Clear SSID, Reset SSID string position
Clear PASSWORD, Reset Pass string position
Set character to blank and its position on the LCD
Confirm selected character
Send data to gpio_data_q and wifiCharPos_q queues
Increase character position
while
Reset character to blank and its position on the LCD
while
Send data to wifiCharPos_q queue

Figure 7b: GPIO task flow chart

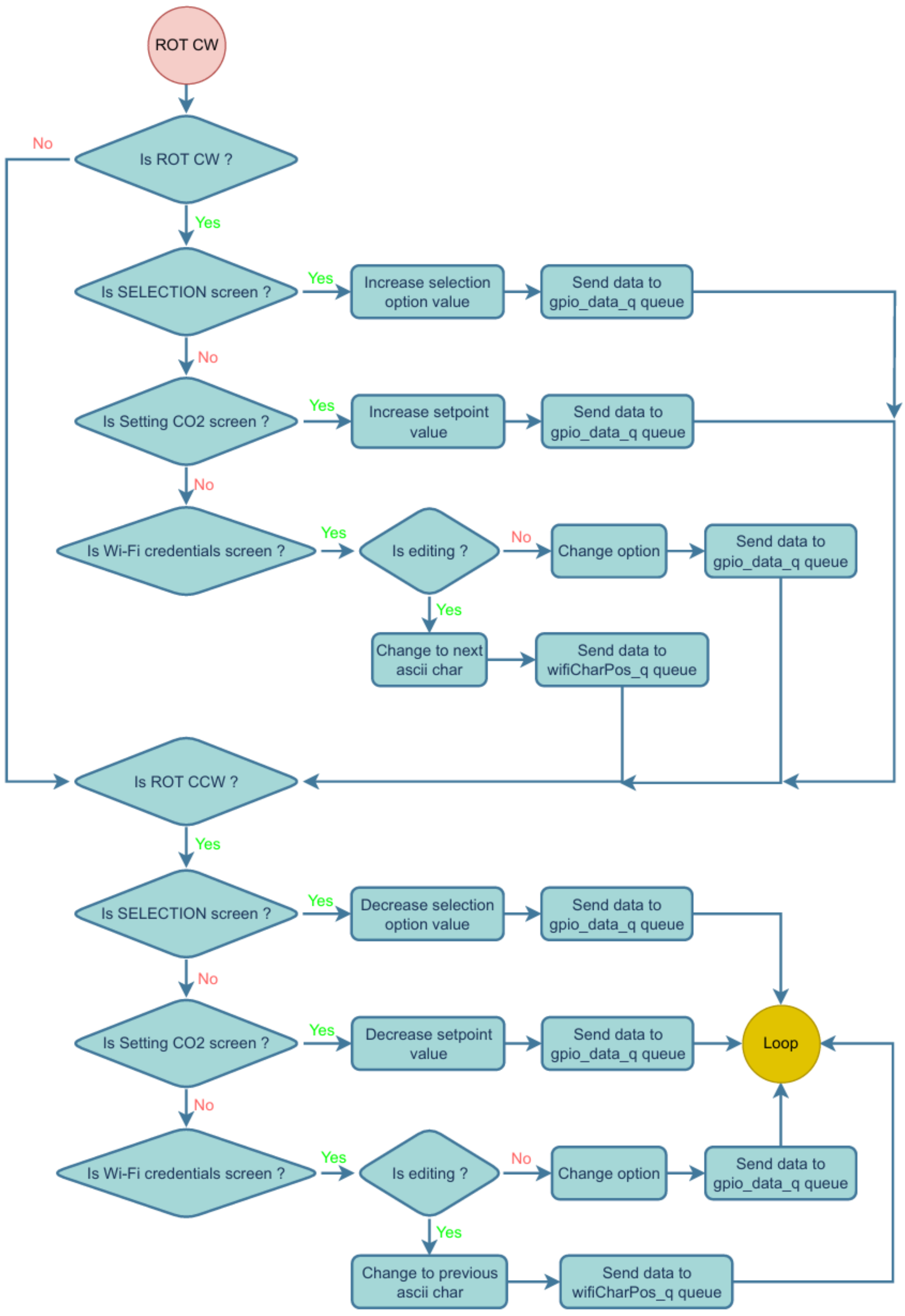
ROT CW
Is ROT CW ?
No
Yes
Is SELECTION screen ?
Yes
Increase selection option value
Send data to gpio_data_q queue
No
Is Setting CO2 screen ?
Yes
Increase setpoint value
Send data to gpio_data_q queue
No
Is Wi-Fi credentials screen ?
Yes
Is editing ?
No
Change option
Send data to gpio_data_q queue
Yes
Change to next ascii char
Send data to wifiCharPos_q queue
Is ROT CCW ?
Yes
Is SELECTION screen ?
Yes
Decrease selection option value
Send data to gpio_data_q queue
No
Is Setting CO2 screen ?
Yes
Decrease setpoint value
Send data to gpio_data_q queue
Loop
No
Is Wi-Fi credentials screen ?
Yes
Is editing ?
No
Change option
Send data to gpio_data_q queue
Yes
Change to previous ascii char
Send data to wifiCharPos_q queue

Figure 7c: GPIO task flow chart

The `gpio_task` is the core task responsible for reacting to events like button presses and rotary encoder movements. It handles user input, processes the state of the system, and updates the display or configuration parameters accordingly. The task processes these events, modifies the internal state of the system (like changing screens, adjusting the setpoint, or editing Wi-Fi credentials), and updates the display accordingly. The entire system operates in a loop, continuously responding to user input while managing screen transitions and system configurations.

GPIO Initialization:

- At the beginning, the task initializes the necessary GPIO pins for:
  - **Rotary encoder (rotA and rotB)**: These will detect rotation signals.
  - **Buttons (ok_btn, back_btn)**: Detect user button presses.
  - **Output devices (LEDs and valves)**: Set as output GPIOs.
- After initialization, the task sets up **interrupts** on these GPIOs by enabling edge detection using `gpio_set_irq_enabled_with_callback()`. This binds the ISR (`hw_ISR_CB()`) to these pins.
- The task waits indefinitely for an action to arrive on the `gpio_action_q` queue using `xQueueReceive()`. The actions will be sent from the ISR whenever a button is pressed or the rotary encoder is turned.
- When a new action is received, the task checks which action occurred (e.g., `OK_PRESS`, `BACK_PRESS`, `ROT_CW`, or `ROT_CCW`).

Process Each Action Based on Screen Type:

Screen Handling Logic:
The task's behavior depends on which screen the user is currently interacting with, which can be one of the following:

- **Selection Screen** (`SELECTION_SCR`): Where the user can navigate between Info, CO2 Setpoint, and Wi-Fi configuration.
- **Set CO2 Screen** (`SET_CO2_SCR`): Where the user adjusts the CO2 setpoint.
- **Wi-Fi Configuration Screen** (`WIFI_CONF_SCR`): Where the user edits Wi-Fi SSID and password.

Action: Back Button Press (`BACK_PRESS`):

If the Back button is pressed:

- The task resets the screen to the selection screen.

- It sends an update to `gpio_data_q` to refresh the display.
- CO2 setpoint is reset, and editing mode is disabled.

Action: OK Button Press (`OK_PRESS`):

When the OK button is pressed:

- The OK button interrupt is temporarily disabled to avoid triggering multiple events.
- A simple debounce mechanism is applied to ignore spurious or unintended rapid button presses (ignores button processing if two consecutive presses within 30ms).

Decision Block of OK Button Press (Screen-Specific Actions):

- ○ **Selection Screen** (`SELECTION_SCR`): If the user is on the Selection Screen, pressing OK will navigate to a submenu (Info, CO2 Setpoint, or Wi-Fi Config) based on the current option. After that data is sent to gpio_data_q queue to update the screen accordingly.
- ○ **Set CO2 Screen** (`SET_CO2_SCR`): Pressing OK confirms the new setpoint value and saves it, triggering an event bit (`CO2_CHANGE_BIT_EEPROM`) to write the setpoint to EEPROM.
- ○ **Wi-Fi Configuration Screen** (`WIFI_CONF_SCR`): The user can configure the Wi-Fi SSID or password by navigating to the **Wi-Fi Configuration Screen** and using both **short press** and **long press** interactions with the **OK button**. There are two main modes of operation:
    - ■ **Non-editing mode**: The user is navigating but not editing the SSID or password.
    - ■ **Editing mode**: The user is modifying the characters of the SSID or password.

    1. **Checking if the User is Editing** (`isEditing flag`)**:** The system starts by checking if the user is in **editing mode** or not. If the user is not currently editing, the behavior of the OK button changes based on whether it is a short press or a long press.

    2. **Long Press to Enter Editing Mode:** The code first checks for a **long press** on the OK button to enter **editing mode**:

        Step 1: Detect Long Press:

        The program initializes a `count` variable and enters a `while` loop to detect how long the OK button is pressed. This process is done by repeatedly checking the OK button's state. If the button remains pressed, the loop

increments `count` until it reaches 150 iterations or the button is released. Each loop iteration waits for 10 milliseconds, so 150 iterations correspond to a 1.5-second press. If `count >= 150`:
This indicates a **long press** (greater than or equal to 1.5 seconds), and the user enters **editing mode**.

3. **Entering Editing Mode:** Once a long press is detected, the system enters **editing mode**:

   Step 2: Initialize for Editing:

- Based on the current option (`wifi_screen_option`), the system decides whether the user is editing the SSID or the password:
  - **If `wifi_screen_option == 0`:**
    The system clears the `SSID_WIFI` and `backup_ssid` strings and sets `posX` to `SSID_POS_X` (position for displaying the SSID on the screen).
  - **If `wifi_screen_option == 1`:**
    The system clears the `PASS_WIFI` and `backup_pass` strings and sets `posX` to `PASS_POS_X` (position for displaying the password).
- The ASCII character `asciiChar` is initialized to `BLANK` character, which is just before the starting ASCII character to be selected.
- The current position and character are updated and sent to the appropriate queue (`gpio_data_q`, `wifiCharPos_q`) to display this information on the screen.

4. **Short Press to Save Data:** The press is less than 150 counts (1.5s)

   Step 3: Save Data:

   The system treats the short press as a command to save the current configuration (either SSID or password). The event group is set using `xEventGroupSetBits(wifiEventGroup, WIFI_CHANGE_BIT_EEPROM)`, signaling the system to store the Wi-Fi changes in EEPROM.

5. **While in Editing Mode** `(isEditing = true)`

   If the user is already in **editing mode** (`isEditing = true`), the system allows the user to either confirm characters or exit editing mode with short or long presses:

6. **Long Press to Exit Editing Mode**

While in editing mode, another **long press** (detected similarly to the previous method) will **exit the editing mode**:

Step 4: Exit Editing Mode:

- The user holds the OK button for 1.5 seconds again to exit editing.
- Upon exiting:
  - **If `wifi_screen_option == 0` (SSID editing)**: The system restores the original SSID from `backup_ssid` to restore a complete SSID string.
  - **If `wifi_screen_option == 1` (Password editing)**: The system restores the original password from `backup_pass` to restore a complete PASS string.
- After restoring the original values, the system sends an update to the queue to refresh the display.

7. Short Press to Confirm a Character

In **editing mode**, a **short press** of the OK button confirms the selected character:

Step 5: Confirm Character:

- The currently selected `asciiChar` (a character in the ASCII table) is converted into a string and appended to either the SSID or password:
  - **If `wifi_screen_option == 0` (SSID editing)**: The character is added to both `SSID_WIFI` and `backup_ssid`.
  - **If `wifi_screen_option == 1` (Password editing)**: The character is added to both `PASS_WIFI` and `backup_pass`.

Step 6: Adjust Cursor Position:

- After appending the character, the position `posX` is updated by adding the width of a character (`CHAR_WIDTH`) to display the next character.
- **"Scrolling" mechanism** (is not shown clearly on the flowchart because it is quite complicated to draw in detail):
  If `posX` exceeds `MAX_POS_X` (the maximum position for the screen), the system performs a "scrolling" mechanism: It shifts the SSID or password by one character to the left (removing the first character), simulating a scrolling text field.
- Finally, the new position and the next character (reset to `BLANK char`) are sent to the queues (`gpio_data_q`, `wifiCharPos_q`) for the screen to reflect the updated state.

Action: Rotary Encoder Clockwise (`ROT_CW`): If the Rotary Encoder is turned clockwise:

- **Selection Screen**: The selection option increases, allowing the user to navigate the menu options.
- **CO2 Setpoint Screen**: The setpoint increases, updating the display.
- **Wi-Fi Config Screen**: If the user is editing, the selected character is incremented.

Action: Rotary Encoder Counterclockwise (`ROT_CCW`): If the Rotary Encoder is turned counterclockwise:

- **Selection Screen**: The selection option decreases.
- **CO2 Setpoint Screen**: The setpoint decreases.
- **Wi-Fi Config Screen**: If editing, the selected character is decremented.

Repeat Loop:

After processing an action, the task loops back to the beginning and waits for another event from the `gpio_action_q` queue. This loop continues indefinitely, ensuring that all user inputs are handled in real time.

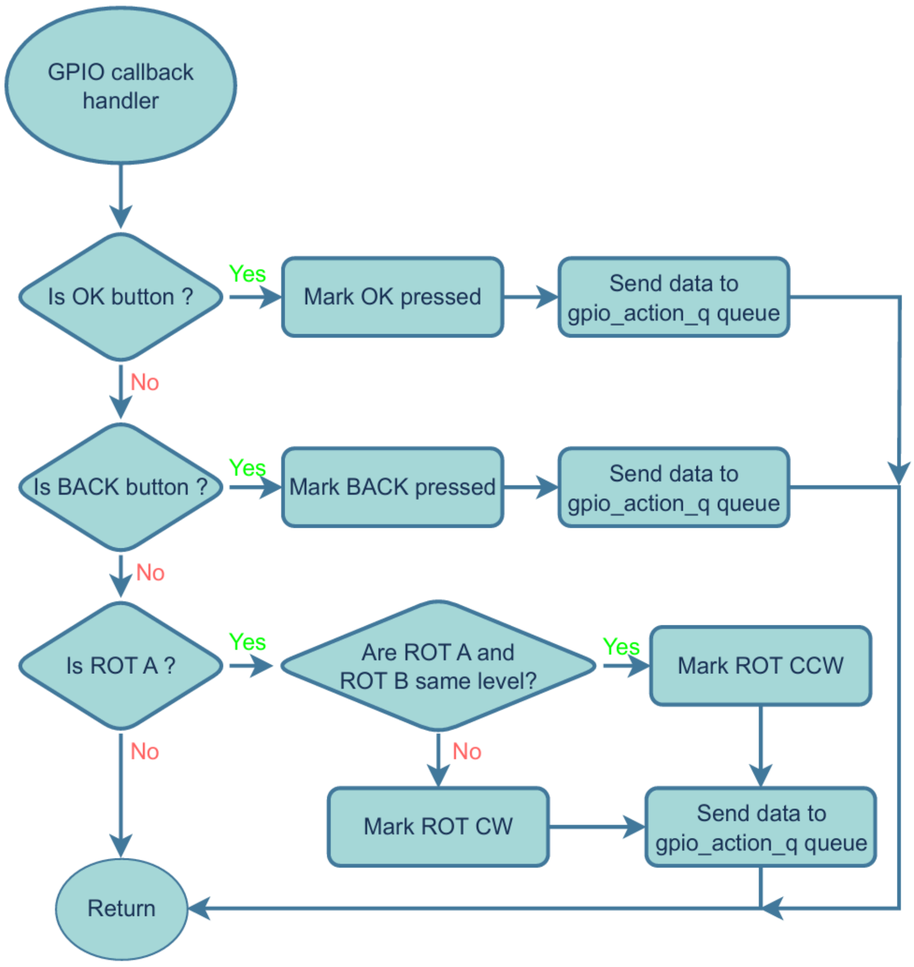

Figure 7d: GPIO task flow chart (ISR Callback Handler)

The `hw_ISR_CB()` function serves as the interrupt handler. It is responsible for detecting hardware events like button presses or rotary encoder movements and immediately notifying the system of the event.

1. **Interrupt Triggered:** When an external event (such as a button press or a rotary encoder signal) occurs on a GPIO pin, the ISR is triggered.
2. **Identify Which GPIO Pin Triggered the Interrupt:** The ISR checks which GPIO pin caused the interrupt using the `gpio` parameter.

Decision Block:

- **If gpio is OK button:**
    - The ISR assigns the action `OK_PRESS` to the `rot_btn_data.action_type`.
    - The action is immediately sent to the `gpio_action_q` queue using `xQueueSendFromISR()`.
- **If gpio is BACK button:**
    - The ISR assigns the action `BACK_PRESS` to the `rot_btn_data.action_type`.
    - The action is sent to `gpio_action_q` just like the OK button.
- **If gpio is ROT A pin:**
    - The ISR checks the state of both `rotA` and `rotB` to determine the direction of the rotary encoder rotation.
    - **Decision Block:**
        - If the state of `rotA` and `rotB` differs (`gpio_get(rotA) != gpio_get(rotB)`), the action `ROT_CW` (clockwise rotation) is assigned to `rot_btn_data.action_type`.
        - Otherwise, the action is assigned `ROT_CCW` (counterclockwise rotation).
    - This action is then sent to the `gpio_action_q` queue.

3.5 MODBUS task

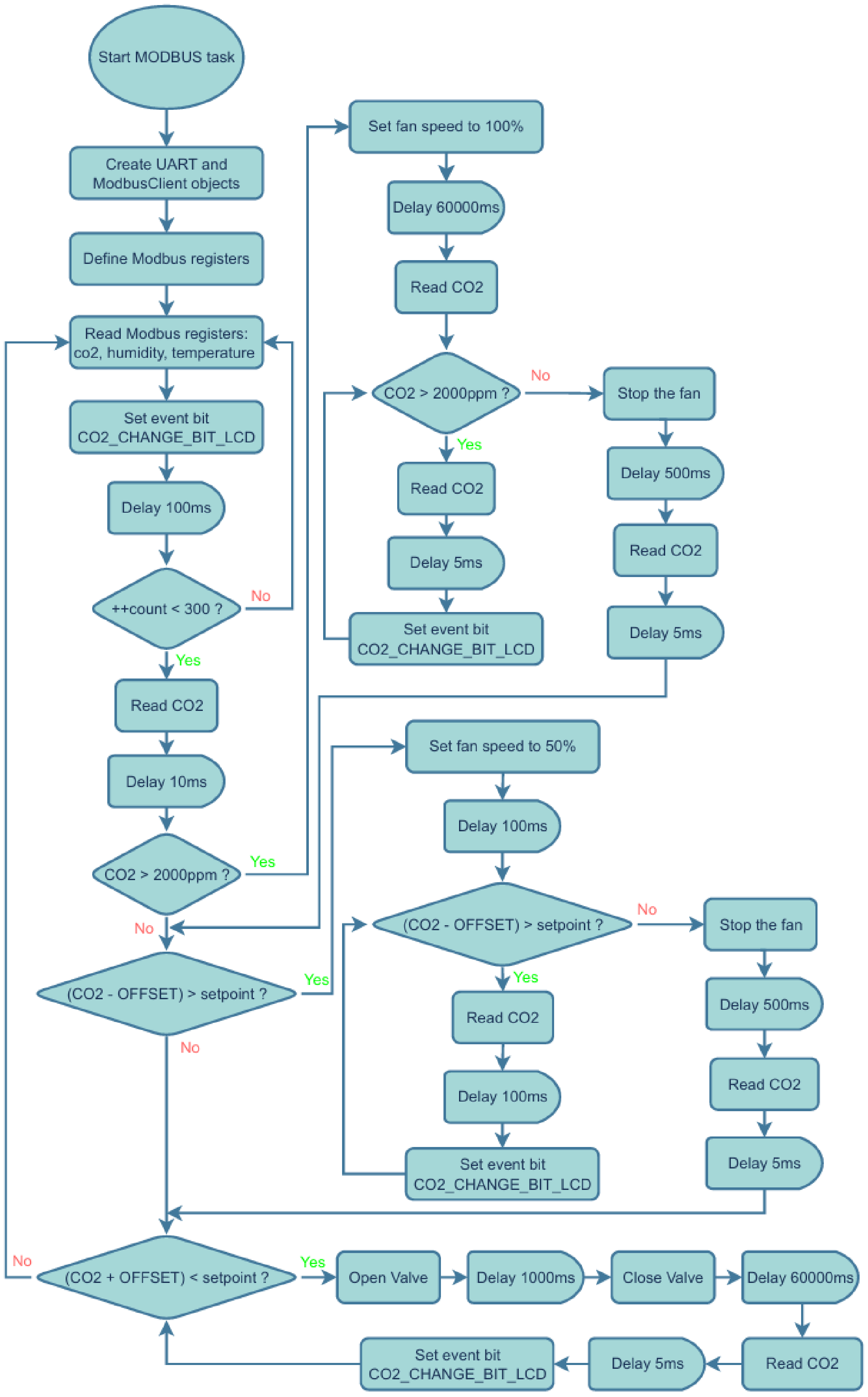

Figure 8: MODBUS task flow chart

The `modbus_task` manages the Modbus communication between the microcontroller and the peripherals like sensors and fan. This includes reading of environmental data (CO2, humidity, temperature) and controls the fan and CO2 valve based on predefined conditions. Here's a breakdown of the process in a flowchart style:

Initialization Phase

1. **Create UART and ModbusClient objects**:
    - A shared UART object is initialized using the `PicoOsUart` class with specific parameters like UART number, TX/RX pins, baud rate, and stop bits.
    - The `ModbusClient` object is created using the UART object, enabling communication with Modbus devices.
2. **Define Modbus registers**:
    - `mb_co2`: CO2 sensor register (Modbus address: 240, register 256).
    - `mb_rh`: Relative humidity sensor register (Modbus address: 241, register 256).
    - `mb_temp`: Temperature sensor register (Modbus address: 241, register 257).
    - `mb_fanSpeed`: Fan speed control register (Modbus address: 1, register 0).

Main Loop

The task enters an infinite loop that continually monitors CO2 levels and controls the fan and CO2 valve at every 3 seconds (300 iterations with 10 ms delay per iteration). During each iteration, the system checks the CO2 level and decides appropriate actions. The task includes checking whether the CO2 level exceeds limit (2000 ppm) in case of emergency and relative CO2 level (with an OFFSET) exceeds or falls below the setpoint.

1. Initial Data Acquisition:

- **CO2 Reading**: The task reads CO2 levels using `mb_co2.read()` and stores the value in `co2`.
- **Humidity (RH) and Temperature Reading**: The task reads the relative humidity and temperature from their respective registers. The humidity and temperature values are divided by 10 to adjust the scaling.
- **Delay**: A short delay of 5 ms is introduced after each reading.
- **Notify Change**: After gathering the data, the task sets the `CO2_CHANGE_BIT_LCD` bit in the `co2EventGroup`, signaling that CO2 data is available to refresh the LCD screen.
- **Wait for 100 ms**: Introduces a delay before proceeding to the next phase.

2. Loop to Check CO2 Levels and Control Fan:

The task enters a secondary loop that runs for approximately 3 seconds (300 iterations with 10 ms delay per iteration).

- **CO2 Level Reading**: CO2 levels are read every 10 ms during this loop.
- **Check if CO2 > CO2_LIMIT (2000 ppm in case of Emergency)**:

  Action (if true):

  - Starts and increases the fan speed to 100%.
  - The system waits for 1 minute (60,000 ms), continuously reading CO2 levels during this time.
  - If CO2 remains above the limit, the fan continues running, and the system keeps reading CO2 and signaling the `CO2_CHANGE_BIT_LCD` flag until the levels drop below the limit.
  - Once CO2 is below the limit, the fan is stopped followed by a delay of 500ms then read the CO2 sensor back.
- **Check if (CO2 - OFFSET) > setpoint**:
  - If CO2 exceeds the setpoint by more than the defined `OFFSET` (100 ppm), the fan is started at 50% speed.
  - The system reads CO2 levels every 100 ms and adjusts the fan speed as needed.
  - Once CO2 levels fall below the setpoint, the fan is stopped followed by a delay of 500ms then read the CO2 sensor back to input for next processing.
- **Check if (CO2 + OFFSET) < setpoint**:
  - If CO2 is below the setpoint by more than the `OFFSET`, the CO2 valve is opened by setting the corresponding GPIO pin high (valve) to inject more CO2 into the greenhouse.
  - The valve remains open for 1 second, after which it is closed.
  - The system waits for 1 minute to allow CO2 levels to stabilize before reading the CO2 sensor again and updating the display with the new values by setting the `CO2_CHANGE_BIT_LCD` bit.
- **Repeat:**

  After handling CO2 levels, fan speed, and valve control, the loop repeats from the beginning. This ensures continuous monitoring and control of the ventilation system based on the environmental conditions.

3.6 MAIN function

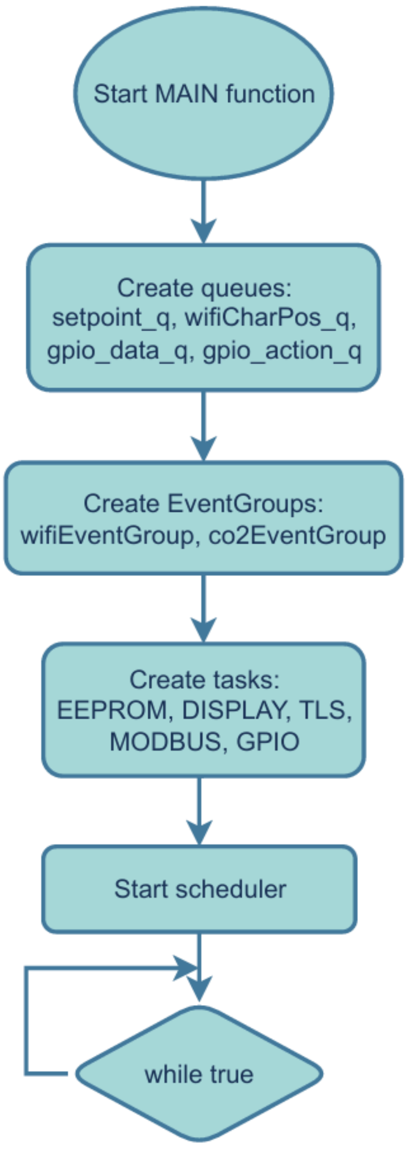
Start MAIN function
Create queues: setpoint_q, wifiCharPos_q, gpio_data_q, gpio_action_q
Create EventGroups: wifiEventGroup, co2EventGroup
Create tasks: EEPROM, DISPLAY, TLS, MODBUS, GPIO
Start scheduler
while true

Figure 9: Main function flow chart

The `main()` function is the entry point of the FreeRTOS-based embedded application. It initializes several system components, such as queues and event groups, and then creates multiple tasks for handling different functions like GPIO, Modbus communication, Wi-Fi, EEPROM, and display operations. After starting the FreeRTOS scheduler, the program begins running all created tasks based on their priority. Once the scheduler is started, the system is fully operational, and task switching is managed by FreeRTOS.

4 Conclusion

In conclusion, this project successfully implements a CO2 fertilization controller system for greenhouses, providing an efficient and automated solution for maintaining optimal growing conditions. By utilizing sensors to monitor CO2 levels, temperature, humidity, and pressure, and by controlling the ventilation fan and CO2 injection valve, the system ensures that plants receive the correct CO2 concentration to maximize growth and productivity.

With its cloud connectivity, the system allows for remote monitoring and control, giving greenhouse operators the ability to manage environmental conditions from anywhere. The use of EEPROM for storing settings ensures that critical parameters persist across system restarts, while safety mechanisms, such as automatic ventilation adjustments when CO2 exceeds safe levels, add an important layer of reliability.

Overall, this project delivers a practical and scalable solution that enhances greenhouse management through automation, improves energy efficiency, and ensures consistent environmental conditions for optimal plant health. The system's flexibility, combined with thorough documentation for both local and remote operations, makes it a valuable tool for modern greenhouse management.

5 References

1. Länsikunnas, Keijo. *Greenhouse CO2 controller specification.* 22.09.2024 [Unpublished lecture slides]. (Accessed: 13 October 2024)