```
/*
 * SPDX-FileCopyrightText: 2022-2023 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
// IMPORTANT!!!!!!!!!!! YOU HAVE TO USE GPIO32 for this to work

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"
#include "freertos/task.h"
#include "soc/soc_caps.h"
#include "esp_log.h"
#include "esp_adc/adc_oneshot.h"
#include "esp_adc/adc_cali.h"
#include "esp_adc/adc_cali_scheme.h"
#include "nvs_flash.h"
// Include your custom WiFi header
#include "hl_wifi.h"

const static char *TAG = "main"; // Changed TAG for main application

/*---------------------------------------------------------------
        ADC General Macros & Constants (from your temperature code)
---------------------------------------------------------------*/
//ADC1 Channels
#if CONFIG_IDF_TARGET_ESP32
#define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_4 // Make sure this matches
your sensor's GPIO, default is GPIO32
#else
#define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_2
#endif

#define EXAMPLE_ADC_ATTEN           ADC_ATTEN_DB_12
#define TX_BUFFER_SIZE              10

// LMT86 Temperature Sensor Constants
// From LMT86 datasheet Section 8.3.1
#define LMT86_SLOPE_MV_PER_C        -10.9f  // mV/°C (negative slope)
#define LMT86_INTERCEPT_MV          2103.0f // mV at 0°C
#define ADC_BITWIDTH_12             4096    // 2^12 for 12-bit ADC
#define ADC_VREF_MV                 3300    // Changed to 3.3V reference in
mV (typical for ESP32), was 5000

// Global queue handle for temperature values
QueueHandle_t tx_queue;

// Global ADC handles for sharing between tasks
static adc_oneshot_unit_handle_t adc1_handle;
static adc_cali_handle_t adc1_cali_chan0_handle = NULL;
static bool do_calibration1_chan0 = false;

// Function prototypes (from your temperature code)
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t
channel, adc_atten_t atten, adc_cali_handle_t *out_handle);
static void example_adc_calibration_deinit(adc_cali_handle_t handle);

// sample_adc function (from your temperature code)
static esp_err_t sample_adc(adc_oneshot_unit_handle_t adc_handle,
    adc_channel_t channel,
    int *result,
    adc_cali_handle_t cali_handle,
    int *voltage)
{
    esp_err_t ret;
    ret = adc_oneshot_read(adc_handle, channel, result);
    if (ret != ESP_OK) {
        return ret;
    }
    if (cali_handle != NULL && voltage != NULL) {
        ret = adc_cali_raw_to_voltage(cali_handle, *result, voltage);
    }
    return ret;
}

// convert raw ADC value to temperature (from your temperature code)
int8_t convert(int raw_adc_value)
{
    float voltage_mv = ((float)raw_adc_value * ADC_VREF_MV) /
ADC_BITWIDTH_12;
    float temperature_c = (voltage_mv - LMT86_INTERCEPT_MV) /
LMT86_SLOPE_MV_PER_C;
    int8_t result = (int8_t)(temperature_c >= 0 ? temperature_c + 0.5f :
temperature_c - 0.5f);

    ESP_LOGI(TAG, "ADC Raw: %d, Voltage: %.1f mV, Temperature: %.1f°C ->
%d°C",
             raw_adc_value, voltage_mv, temperature_c, result);

    if (voltage_mv < 1000) {
        ESP_LOGW(TAG, "⚠ Voltage too low (%.1f mV) - Check sensor
connections!", voltage_mv);
    }
    if (temperature_c < -40 || temperature_c > 125) {
        ESP_LOGW(TAG, "⚠ Temperature out of expected range (%.1f°C)",
temperature_c);
    }
    return result;
}

// convert from calibrated voltage to temperature (from your temperature
code)
int8_t convert_from_voltage(int voltage_mv)
{
    float temperature_c = ((float)voltage_mv - LMT86_INTERCEPT_MV) /
LMT86_SLOPE_MV_PER_C;
    int8_t result = (int8_t)(temperature_c >= 0 ? temperature_c + 0.5f :
temperature_c - 0.5f);

    ESP_LOGI(TAG, "Voltage: %d mV, Temperature: %.1f°C -> %d°C",
             voltage_mv, temperature_c, result);

    if (voltage_mv < 1000) {
        ESP_LOGW(TAG, "⚠ Calibrated voltage too low (%d mV) - Check sensor
connections!", voltage_mv);
    }
    return result;
}

/**
 * TaskSample - Continuously samples the ADC and puts values into the queue
 * (Modified from your temperature code to use the global tx_queue)
 */
void TaskSample(void* pvParameters)
{
    int adc_raw;
    int voltage_mv;

    ESP_LOGI(TAG, "Sample task started");

    while (1) {
        // Perform ADC conversion
        ESP_ERROR_CHECK(sample_adc(adc1_handle, EXAMPLE_ADC1_CHAN0,
&adc_raw,
                                   do_calibration1_chan0 ?
adc1_cali_chan0_handle : NULL,
                                   &voltage_mv));

        int8_t temperature_raw = convert(adc_raw);
        int8_t temperature_cal = 0;

        if (do_calibration1_chan0) {
            temperature_cal = convert_from_voltage(voltage_mv);
        }

        // Use calibrated temperature if available, otherwise use raw
conversion
        int value = do_calibration1_chan0 ? (int)temperature_cal :
(int)temperature_raw;

        ESP_LOGI(TAG, "ADC Raw: %d, Voltage: %d mV, Temp(raw): %d°C,
Temp(cal): %d°C -> Sending: %d°C",
                 adc_raw, voltage_mv, temperature_raw, temperature_cal,
value);

        // Send the temperature value to the global tx_queue
        // Use portMAX_DELAY to wait indefinitely if queue is full
        if (xQueueSendToBack(tx_queue, &value, portMAX_DELAY) != pdTRUE) {
            ESP_LOGE(TAG, "Failed to send value to queue (should not happen
with portMAX_DELAY)");
        }

        // Sample every 2 seconds
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}



/**
 * TaskCount - Connects to TCP server and transmits temperature values from
queue
 * (Modified from your Quest 3 TaskCount)
 */
void TaskCount(void *pvParameters)
{
    // pvParameters is not used here, as tx_queue is global
    // The previous connected_callback created this task.
    printf("TCP connection task started!\n");

    // This part runs once to establish the TCP connection
    sockaddr_in_t addr = hl_wifi_make_addr("10.42.0.1", 8000); // Your
laptop's hotspot IP
    int sock = hl_wifi_tcp_connect(addr);
    if (sock == -1) {
        ESP_LOGE(TAG, "Failed to connect to TCP server, deleting task.");
        vTaskDelete(NULL);
        return;
    }

    int temperature_value; // To store received temperature
    char buffer[10];       // Buffer to format the temperature string

    while (true) {
        // Receive temperature value from the global tx_queue
        // Use portMAX_DELAY to wait indefinitely for data to become
available
        if (xQueueReceive(tx_queue, &temperature_value, portMAX_DELAY) ==
pdPASS) {
            // Format the temperature into a string with a newline
            int len = sprintf(buffer, "%d\n", temperature_value);

            // Transmit the temperature string over TCP
            hl_wifi_tcp_tx(sock, buffer, len);

            ESP_LOGI(TAG, "Sent temperature: %d°C", temperature_value);
        } else {
            ESP_LOGE(TAG, "Failed to receive temperature from queue (should
not happen with portMAX_DELAY)");
        }
        // No vTaskDelay here, as TaskSample already delays and dictates
the rate.
    }
    // These lines should ideally not be reached if the while loop is
infinite
    // close(sock);
    // vTaskDelete(NULL);
}


// Your connected_callback - remains mostly the same
void connected_callback(void) {
    xTaskCreate(TaskCount,
                "TaskCount",
                4096, // Increased stack size for sprintf and loop
                NULL, // No parameter passed, tx_queue is global
                5,
                NULL);
}

// app_main - Entry point of your application
void app_main(void)
{
    // Initialize NVS (Non-Volatile Storage) for WiFi
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
      ESP_ERROR_CHECK(nvs_flash_erase());
      ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    //-------------ADC1 Init---------------//
    adc_oneshot_unit_init_cfg_t init_config1 = {
        .unit_id = ADC_UNIT_1,
    };
    ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));

    //-------------ADC1 Config---------------//
    adc_oneshot_chan_cfg_t config = {
        .atten = EXAMPLE_ADC_ATTEN,
        .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    // Ensure GPIO32 is connected to ADC1_CHANNEL_4
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle,
EXAMPLE_ADC1_CHAN0, &config));

    //-------------ADC1 Calibration Init---------------//
    do_calibration1_chan0 = example_adc_calibration_init(ADC_UNIT_1,
EXAMPLE_ADC1_CHAN0, EXAMPLE_ADC_ATTEN, &adc1_cali_chan0_handle);

    //-------------Queue Creation---------------//
    tx_queue = xQueueCreate(TX_BUFFER_SIZE, sizeof(int));
    if (tx_queue == NULL) {
        ESP_LOGE(TAG, "Failed to create queue");
        return;
    }
    //-------------Task Creation---------------//
    // Create Sample Task to read temperature and put into queue
    xTaskCreate(TaskSample, "Sample", 4096, NULL, 1, NULL); // No parameter
needed for TaskSample as tx_queue is global

    // Initialize WiFi and register the callback for TCP connection
    ESP_LOGI(TAG, "ESP_WIFI_MODE_STA");
    hl_wifi_init(connected_callback);

    ESP_LOGI(TAG, "Temperature pipeline and WiFi initialization
complete.");
    ESP_LOGI(TAG, "Touch the LMT86 sensor on GPIO32 to see temperature
changes!");
}

/*---------------------------------------------------------------
        ADC Calibration Functions (from your temperature code)
---------------------------------------------------------------*/
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t
channel, adc_atten_t atten, adc_cali_handle_t *out_handle)
{
    adc_cali_handle_t handle = NULL;
    esp_err_t ret = ESP_FAIL;
    bool calibrated = false;

#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
    if (!calibrated) {
        ESP_LOGI(TAG, "calibration scheme version is %s", "Curve Fitting");
        adc_cali_curve_fitting_config_t cali_config = {
            .unit_id = unit,
            .chan = channel,
            .atten = atten,
            .bitwidth = ADC_BITWIDTH_DEFAULT,
        };
        ret = adc_cali_create_scheme_curve_fitting(&cali_config, &handle);
        if (ret == ESP_OK) {
            calibrated = true;
        }
    }
#endif

#if ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
    if (!calibrated) {
        ESP_LOGI(TAG, "calibration scheme version is %s", "Line Fitting");
        adc_cali_line_fitting_config_t cali_config = {
            .unit_id = unit,
            .atten = atten,
            .bitwidth = ADC_BITWIDTH_DEFAULT,
        };
        ret = adc_cali_create_scheme_line_fitting(&cali_config, &handle);
        if (ret == ESP_OK) {
            calibrated = true;
        }
    }
#endif

    *out_handle = handle;
    if (ret == ESP_OK) {
        ESP_LOGI(TAG, "Calibration Success");
    } else if (ret == ESP_ERR_NOT_SUPPORTED || !calibrated) {
        ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
    } else {
        ESP_LOGE(TAG, "Invalid arg or no memory");
    }

    return calibrated;
}

static void example_adc_calibration_deinit(adc_cali_handle_t handle)
{
#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
    ESP_LOGI(TAG, "deregister %s calibration scheme", "Curve Fitting");
    ESP_ERROR_CHECK(adc_cali_delete_scheme_curve_fitting(handle));

#elif ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
    ESP_LOGI(TAG, "deregister %s calibration scheme", "Line Fitting");
    ESP_ERROR_CHECK(adc_cali_delete_scheme_line_fitting(handle));
#endif
}
```

Absolutely! Here's the explanation of the `main.c` file, with all code examples removed.

Explanation of the `main.c` File

This `main.c` file creates a comprehensive Internet of Things (IoT) application for an **ESP32 microcontroller**. It focuses on reading analog sensor data, processing it, and then transmitting it over a network using the TCP/IP protocol. The application integrates several key components of the ESP-IDF framework, along with your custom `hl_wifi` module for simplified networking.

---

1. Copyright and Essential Setup

The file begins with standard copyright and licensing information. A crucial comment at the start highlights a critical hardware requirement: the analog temperature sensor **must be connected to GPIO32** on the ESP32. This is because GPIO32 is specifically mapped to `ADC1_CHANNEL_4`, which the program uses to read the sensor's voltage.

It then includes various header files. These bring in functionalities for:

- **Basic C operations** like input/output, memory management, and string manipulation.
- **FreeRTOS**, the real-time operating system that powers the ESP32, enabling the creation and management of tasks and queues.

- **ESP-IDF's core features**, including logging, the Analog-to-Digital Converter (ADC) driver for reading analog sensors, and Non-Volatile Storage (NVS) for saving Wi-Fi credentials persistently.
- Your custom `hl_wifi.h` header, which provides higher-level functions for Wi-Fi connectivity and TCP communication.

---

2. ADC and LMT86 Sensor Configuration

This section defines essential parameters for interacting with the LMT86 analog temperature sensor via the ESP32's ADC:

- It specifies `ADC1_CHANNEL_4` (GPIO32) as the input channel for the sensor.
- An ADC **input attenuation** of `DB_12` is set, allowing the ADC to accurately measure a wider voltage range, compatible with the ESP32's typical 3.3V power supply.
- Constants specific to the **LMT86 sensor's electrical characteristics** are defined. These include the sensor's voltage **slope** and **intercept** at 0°C, which are vital for converting the measured voltage into a temperature value.
- ADC resolution (`12-bit`, meaning 4096 possible values) and the ESP32's internal **reference voltage** (3300mV or 3.3V) are also defined for accurate voltage calculations.
- Global variables are declared to manage the ADC unit and its calibration, and a **FreeRTOS Queue handle** (`tx_queue`) is set up to allow different parts of the application to pass temperature data between them.

---

3. ADC and Temperature Conversion Functions

Several helper functions are present to facilitate sensor reading and temperature calculation:

- A `sample_adc` function handles the direct interaction with the ESP32's ADC. It reads the raw analog value from the sensor and can also apply **ADC calibration** to convert this raw value into a more accurate voltage in millivolts (mV).
- The `convert` function takes a **raw ADC value** and uses the LMT86 sensor's defined slope, intercept, ADC bit-width, and reference voltage to calculate the corresponding temperature in Celsius. It includes logging for debugging and warnings to flag potential issues like voltages or temperatures outside the sensor's expected operating range.
- A `convert_from_voltage` function performs a similar temperature calculation but is designed to work with **calibrated voltage values** obtained from the ADC.

---

4. FreeRTOS Tasks

The application uses two main FreeRTOS tasks to manage its operations concurrently:

`TaskSample` - ADC Sampling Task

This task is the **data producer**. It runs continuously, periodically performing the following actions:

- It uses the `sample_adc` function to read the latest value from the LMT86 temperature sensor.
- It converts both the raw and, if available, the calibrated sensor readings into temperature values.

- It logs detailed information about the ADC raw value, voltage, and calculated temperatures.
- The most accurate temperature value (calibrated, if successful, otherwise raw) is then sent to the **`tx_queue`**. The task will wait indefinitely if the queue is full, ensuring no data is lost.
- It introduces a 2-second delay between samples, controlling the rate at which temperature data is generated and pushed to the queue.

`TaskCount` - TCP Transmission Task

This task acts as the **data consumer and transmitter**. It is responsible for sending the collected temperature data over the network:

- Upon its creation, it first uses functions from your `hl_wifi` module to establish a **TCP client connection** to a remote server. This server is expected to be running on your laptop at a specified IP address (e.g., "10.42.0.1") and port (e.g., 8000). If the connection fails, the task will shut itself down.
- It then enters an infinite loop, constantly waiting for new temperature values to become available in the `tx_queue`. It will block until `TaskSample` provides data.
- Once a temperature value is received from the queue, it's formatted into a simple string (e.g., "25\n") and immediately transmitted over the established TCP socket using another function from your `hl_wifi` module.
- This task's execution rate is effectively controlled by `TaskSample`'s sampling rate, as it processes data as soon as it's available.

---

5. `connected_callback`

This function serves as a crucial link. It's designed to be called by your `hl_wifi` module once the ESP32 successfully connects to a Wi-Fi network and obtains an IP address. Its primary role is to then **create and start the `TaskCount` TCP transmission task**. This design ensures that the application only attempts to establish network connections *after* Wi-Fi connectivity is confirmed, promoting robust behavior.

---

6. `app_main` - Application Entry Point

This is the very first function that runs when the ESP32 powers on:

- It initializes **Non-Volatile Storage (NVS)**, which is essential for the Wi-Fi driver and for storing persistent data like network credentials.
- It sets up and configures the **ADC unit and channel** for reading the LMT86 sensor, including attempting to initialize **ADC calibration** for improved accuracy.
- It creates the **FreeRTOS queue** (`tx_queue`) that will be used for inter-task communication.
- It then creates and starts **`TaskSample`**, which immediately begins collecting temperature data.
- Finally, it initiates the **Wi-Fi connection process** by calling `hl_wifi_init` and registers the `connected_callback` function. This means that once Wi-Fi is ready, your TCP communication task will automatically start.

In Summary

This program creates a functional and robust IoT solution on the ESP32. It demonstrates how to:

- Interface with an **analog sensor** (LMT86 temperature sensor) using the ESP32's ADC.
- Utilize **FreeRTOS tasks and queues** for efficient and concurrent data processing.
- Connect to a **Wi-Fi network** using an abstracted module (`hl_wifi`).
- Establish and maintain a **TCP connection** to send sensor data to a remote server.

This entire setup forms a pipeline for reliably sending environmental data from the ESP32 to another networked device.