this is the code for laying pipe

```
/*
 * 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"

const static char *TAG = "TEMPERATURE_SENSOR";

/*---------------------------------------------------------------
        ADC General Macros
---------------------------------------------------------------*/
//ADC1 Channels
#if CONFIG_IDF_TARGET_ESP32
#define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_4
#define EXAMPLE_ADC1_CHAN1          ADC_CHANNEL_5
#else
#define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_2
#define EXAMPLE_ADC1_CHAN1          ADC_CHANNEL_3
#endif

#if (SOC_ADC_PERIPH_NUM >= 2) && !CONFIG_IDF_TARGET_ESP32C3
/**
 * On ESP32C3, ADC2 is no longer supported, due to its HW limitation.
 * Search for errata on espressif website for more details.
 */
#define EXAMPLE_USE_ADC2            1
#endif

#if EXAMPLE_USE_ADC2
//ADC2 Channels
#if CONFIG_IDF_TARGET_ESP32
#define EXAMPLE_ADC2_CHAN0          ADC_CHANNEL_0
#else
#define EXAMPLE_ADC2_CHAN0          ADC_CHANNEL_0
#endif
#endif  //#if EXAMPLE_USE_ADC2
```

```
#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                 5000    // 3.3V reference in mV (typical
for 12dB attenuation)

// Expected voltage ranges for LMT86:
// At 25°C: ~1830 mV
// At 0°C:  ~2103 mV
// At 50°C: ~1830 mV

// Global queue handle
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;

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);

/**
 * Perform a single ADC sample
 *
 * @param adc_handle The ADC handle to use
 * @param channel The ADC channel to sample
 * @param result Pointer to store the raw result
 * @param cali_handle Calibration handle (NULL if no calibration)
 * @param voltage Pointer to store calibrated voltage (NULL if not needed)
 *
 * @return ESP_OK on success, error code otherwise
 */
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;

    // Read the raw value
    ret = adc_oneshot_read(adc_handle, channel, result);
    if (ret != ESP_OK) {
        return ret;
    }

    // Convert to voltage if calibration handle is provided and voltage
```

```
pointer is not NULL
    if (cali_handle != NULL && voltage != NULL) {
        ret = adc_cali_raw_to_voltage(cali_handle, *result, voltage);
    }

    return ret;
}

/**
 * Convert raw ADC value to temperature in Celsius
 *
 * This function implements the LMT86 temperature conversion formula:
 * - Raw ADC -> Voltage: voltage_mV = (raw_value * VREF_mV) / ADC_MAX_VALUE
 * - Voltage -> Temperature: temp_C = (voltage_mV - INTERCEPT_mV) /
SLOPE_mV_per_C
 *
 * Combined formula for maximum precision:
 * temp_C = ((raw_value * VREF_mV / ADC_MAX_VALUE) - INTERCEPT_mV) /
SLOPE_mV_per_C
 *
 * @param raw_adc_value Raw ADC reading (0 to 4095 for 12-bit)
 * @return Temperature in degrees Celsius as int8_t
 */
int8_t convert(int raw_adc_value)
{
    // Convert raw ADC to voltage in mV
    // Using float for intermediate calculation to maintain precision
    float voltage_mv = ((float)raw_adc_value * ADC_VREF_MV) /
ADC_BITWIDTH_12;

    // Convert voltage to temperature using LMT86 formula
    // T(°C) = (V_out - V_0°C) / Slope
    // Where: V_0°C = 2103mV, Slope = -10.9mV/°C
    float temperature_c = (voltage_mv - LMT86_INTERCEPT_MV) /
LMT86_SLOPE_MV_PER_C;

    // Round to nearest integer and cast to int8_t
    int8_t result = (int8_t)(temperature_c >= 0 ? temperature_c + 0.5f :
temperature_c - 0.5f);

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

    // Diagnostic warnings
    if (voltage_mv < 1000) {
        ESP_LOGW(TAG, "⚠ Voltage too low (%.1f mV) - Check sensor
connections!", voltage_mv);
        ESP_LOGW(TAG, "Expected ~1830mV at 25°C, ~2103mV at 0°C");
    }
    if (temperature_c < -40 || temperature_c > 125) {
        ESP_LOGW(TAG, "⚠ Temperature out of expected range (%.1f°C)",
temperature_c);
```

```
    }

    return result;
}

/**
 * Alternative convert function using calibrated voltage input
 * Use this version if you have calibrated voltage values from the ADC
calibration
 *
 * @param voltage_mv Calibrated voltage in millivolts
 * @return Temperature in degrees Celsius as int8_t
 */
int8_t convert_from_voltage(int voltage_mv)
{
    // Convert voltage to temperature using LMT86 formula
    float temperature_c = ((float)voltage_mv - LMT86_INTERCEPT_MV) /
LMT86_SLOPE_MV_PER_C;

    // Round to nearest integer and cast to int8_t
    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);

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

    return result;
}

/**
 * Sample Task - Continuously samples the ADC and puts values into the
queue
 */
void TaskSample(void* pvParameters)
{
    QueueHandle_t output_queue = (QueueHandle_t)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));
```

```
        // Convert to temperature using both methods for comparison
        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;

        // Debug output showing the conversion process
        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 queue
        while (xQueueSendToBack(output_queue, &value, 10) != pdTRUE) {
            ESP_LOGW(TAG, "Queue full, retrying...");
        }

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

/**
 * Transmit Task - Receives values from the queue and transmits them
 */
void TaskTransmit(void* pvParameters)
{
    QueueHandle_t input_queue = (QueueHandle_t)pvParameters;
    int value;

    ESP_LOGI(TAG, "Transmit task started");

    while (1) {
        // Receive value from queue
        while (xQueueReceive(input_queue, &value, 10) != pdPASS) {
            // Continue waiting for data
        }

        // Transmit the value
        printf("Temperature: %d°C\n", value);
        ESP_LOGI(TAG, "=== TEMPERATURE READING ===");
        ESP_LOGI(TAG, "Temperature: %d°C", value);
        ESP_LOGI(TAG, "============================");
    }
}
```

```
void app_main(void)
{
    //-------------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,
    };
    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
    xTaskCreate(TaskSample, "Sample", 4096, tx_queue, 1, NULL);

    // Create Transmit Task
    xTaskCreate(TaskTransmit, "Transmit", 4096, tx_queue, 1, NULL);

    ESP_LOGI(TAG, "Pipeline started - Sample -> Queue -> Transmit");
    ESP_LOGI(TAG, "Touch the LMT86 sensor to see temperature changes!");
}

/*---------------------------------------------------------------
        ADC Calibration
---------------------------------------------------------------*/
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
}
```

This C code, designed for **ESP32 microcontrollers** using the **ESP-IDF framework**, sets up a robust system for reading temperature from an **LMT86 analog temperature sensor**. Unlike simpler examples, this one utilizes **FreeRTOS tasks and queues** to create a multi-threaded pipeline: one task samples the sensor, and

another task transmits the processed temperature data. It also includes **ADC calibration** for improved accuracy.

Core Functionality

The program establishes a **producer-consumer model** for temperature data:

1. **Sensor Sampling Task (`TaskSample`):**
   - Continuously reads raw analog voltage from the **LMT86 sensor** via the ESP32's **Analog-to-Digital Converter (ADC)**.
   - Converts the ADC reading into a **temperature in Celsius** using the LMT86 sensor's specific formula.
   - **Pushes** the calculated temperature value into a **FreeRTOS queue**.
2. **Data Transmission Task (`TaskTransmit`):**
   - **Waits for and receives** temperature values from the FreeRTOS queue.
   - **Prints** the received temperature to the serial monitor.
3. **ADC Initialization and Calibration:** Sets up the ADC unit and attempts to calibrate it for more accurate voltage measurements from the sensor.
4. **Queue Management:** A FreeRTOS queue acts as a buffer, enabling the two tasks to communicate data efficiently without direct coupling.

Code Breakdown

1. Includes and Configuration Macros

This section defines necessary libraries and parameters:

- **Standard C Libraries:** `stdio.h`, `stdlib.h`, `string.h` for basic operations.
- **FreeRTOS Headers:** Crucial for multi-tasking: `freertos/FreeRTOS.h` (core FreeRTOS), `freertos/queue.h` (for inter-task communication), and `freertos/task.h` (for task management).
- **ESP-IDF Specific Headers:** `soc/soc_caps.h` (system-on-chip capabilities), `esp_log.h` (logging), and `esp_adc/*` for ADC functionalities (one-shot readings and calibration).
- **`TAG`**: Used for identifying log messages.
- **ADC Channels (`EXAMPLE_ADC1_CHAN0`, etc.):** Define which GPIO pins are configured as analog inputs. The comment **"IMPORTANT!!!!!!!!!! YOU HAVE TO USE GPIO32 for this to work"** highlights that `ADC_CHANNEL_4` (which maps to GPIO32 on ESP32) is the intended input for the sensor.
- **ADC Attenuation (`EXAMPLE_ADC_ATTEN`):** Set to `ADC_ATTEN_DB_12`, allowing the ADC to measure voltages up to approximately 3.9V.
- **`TX_BUFFER_SIZE`**: Defines the size of the FreeRTOS queue (10 elements), meaning it can buffer up to 10 temperature readings before the `TaskSample` might block if the `TaskTransmit` is slow.
- **LMT86 Temperature Sensor Constants:** These are vital for converting sensor voltage into temperature, derived from the LMT86 datasheet:
  - `LMT86_SLOPE_MV_PER_C`: The sensor's output change per degree Celsius (-10.9 mV/°C).
  - `LMT86_INTERCEPT_MV`: The sensor's output voltage at 0°C (2103.0 mV).
  - `ADC_BITWIDTH_12`: The maximum raw value for a 12-bit ADC (4096).

- `ADC_VREF_MV`: The assumed analog reference voltage for ADC calculations (set to 5000 mV, or 5.0V, which might be a typo and should likely be 3300mV for typical ESP32 usage, especially with 12dB attenuation, but depends on external circuitry). This is a critical value for accurate temperature conversion.

2. Global Variables and Function Prototypes

- `tx_queue`: A **global** `QueueHandle_t` (FreeRTOS queue handle) that will be used by both tasks for communication.
- `adc1_handle` **and** `adc1_cali_chan0_handle`: Global ADC handles to allow both tasks (or any part of the application) to access the initialized ADC without re-initializing.
- `do_calibration1_chan0`: A global boolean flag to indicate if ADC calibration was successful.
- Prototypes for `example_adc_calibration_init` and `example_adc_calibration_deinit` are declared.

3. `sample_adc` Function (ADC Reading Helper)

This helper function performs a single ADC reading.

- It uses `adc_oneshot_read()` to get a raw ADC value.
- If a calibration handle is provided and valid, it uses `adc_cali_raw_to_voltage()` to convert the raw reading into a more accurate **calibrated voltage in millivolts**. This accounts for potential manufacturing variations in the ESP32's ADC.

4. `convert` Function (Raw ADC to Temperature)

This function translates the raw ADC reading into a temperature in Celsius.

- **Raw ADC to Voltage Conversion:** It first calculates the voltage (in mV) from the raw ADC value using the `ADC_VREF_MV` and `ADC_BITWIDTH_12` constants.
- **Voltage to Temperature Conversion:** It then applies the linear formula from the LMT86 datasheet, utilizing `LMT86_INTERCEPT_MV` and `LMT86_SLOPE_MV_PER_C`.
- **Rounding and Diagnostics:** It rounds the floating-point temperature to the nearest integer and provides helpful log messages, including raw values, calculated voltage, and temperature. It also issues **warning messages** if the voltage or temperature falls outside expected ranges, which is useful for debugging.

5. `convert_from_voltage` Function (Calibrated Voltage to Temperature)

This is an alternative conversion function. If the ADC calibration was successful, you can directly pass the **calibrated voltage** (in mV) to this function, which then applies the LMT86 formula to derive the temperature. This generally provides more accurate temperature readings.

6. `TaskSample` Function (Producer Task)

This FreeRTOS task is responsible for periodically sampling the LMT86 sensor.

- It's passed the `output_queue` (which is `tx_queue` from `app_main`) as a parameter.
- In an infinite loop, it calls `sample_adc` to get raw and calibrated voltage readings.

- It then converts these readings into temperature using both `convert` (for raw data) and `convert_from_voltage` (for calibrated data).
- It selects the **calibrated temperature** if calibration is available; otherwise, it uses the raw conversion.
- It attempts to **send (`xQueueSendToBack`)** this temperature value to the `output_queue`. If the queue is full, it logs a warning and retries.
- It introduces a **2-second delay** (`vTaskDelay`) between samples, controlling the sampling rate.

7. `TaskTransmit` Function (Consumer Task)

This FreeRTOS task is responsible for displaying the temperature data.

- It's passed the `input_queue` (which is `tx_queue`) as a parameter.
- In an infinite loop, it **receives (`xQueueReceive`)** temperature values from the `input_queue`. It will block (wait) until a value is available in the queue.
- Once a value is received, it prints the temperature to the **serial console** using `printf` and also logs it using `ESP_LOGI`.

8. `app_main` Function (Main Application Entry Point)

This is the entry point of the ESP32 application.

- **ADC Initialization:** It initializes `ADC_UNIT_1` and configures `EXAMPLE_ADC1_CHAN0` (GPIO32) with the defined attenuation and bit width.
- **ADC Calibration:** It attempts to perform ADC calibration for the selected channel, setting the `do_calibration1_chan0` flag based on success.
- **Queue Creation:** It creates the `tx_queue` (a FreeRTOS queue) with a capacity for `TX_BUFFER_SIZE` (10) integer values. It checks if the queue creation was successful.
- **Task Creation:**
  - It creates the `TaskSample` and `TaskTransmit` tasks using `xTaskCreate`.
  - Both tasks are given a stack size of 4096 bytes and a priority of 1.
  - Crucially, `tx_queue` is passed as a parameter to both tasks, enabling them to share the queue.
- **Initial Logs:** It prints messages indicating that the "pipeline" has started and instructs the user to interact with the sensor.

9. ADC Calibration Functions (`example_adc_calibration_init` and `example_adc_calibration_deinit`)

These functions are standard ESP-IDF helpers for ADC calibration:

- `example_adc_calibration_init`: Attempts to create an ADC calibration handle. It tries to use the more accurate **Curve Fitting** scheme first (if supported and eFuse data is present). If not, it falls back to **Line Fitting**. It logs the success or failure of calibration.
- `example_adc_calibration_deinit`: Properly releases the resources associated with an ADC calibration handle. (Note: In this specific code, the `app_main` function's infinite loop means the de-initialization part of the code is never reached during normal operation.)

How to Use This Code

1. **Hardware Connection:**
    - Connect an **LMT86 analog temperature sensor** to **GPIO32** on your ESP32 development board.
    - Ensure the LMT86 sensor is correctly wired for power, ground, and its analog output to GPIO32.
2. **Review `ADC_VREF_MV`**: Double-check the `ADC_VREF_MV` macro. While 5000 mV is present in the code, for most ESP32 boards with 12dB attenuation, the ADC's reference voltage is typically 3300 mV (3.3V). An incorrect `ADC_VREF_MV` will lead to inaccurate temperature readings. Adjust it if necessary based on your hardware setup.
3. **Build and Flash:** Compile this code using the ESP-IDF toolchain and flash it onto your ESP32.
4. **Serial Monitor:** Open a serial monitor (at 115200 baud) connected to your ESP32.

You'll see continuous temperature readings printed to your serial monitor every two seconds. The system's robustness comes from FreeRTOS, allowing concurrent sampling and transmission without one blocking the other. Touching the LMT86 sensor should cause the temperature readings to change accordingly.