This is the code for a oneshot reading of the temperature sensor

```
/*
 * SPDX-FileCopyrightText: 2022-2023 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.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 = "EXAMPLE";

/*---------------------------------------------------------------
        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

static int adc_raw[2][10];
```

```
static int voltage[2][10];
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_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;
}


void app_main(void)
{
    //-------------ADC1 Init---------------//
    adc_oneshot_unit_handle_t adc1_handle;
    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));
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle,
EXAMPLE_ADC1_CHAN1, &config));

    //-------------ADC1 Calibration Init---------------//
    adc_cali_handle_t adc1_cali_chan0_handle = NULL;
    adc_cali_handle_t adc1_cali_chan1_handle = NULL;
    bool do_calibration1_chan0 = example_adc_calibration_init(ADC_UNIT_1,
EXAMPLE_ADC1_CHAN0, EXAMPLE_ADC_ATTEN, &adc1_cali_chan0_handle);
    bool do_calibration1_chan1 = example_adc_calibration_init(ADC_UNIT_1,
EXAMPLE_ADC1_CHAN1, EXAMPLE_ADC_ATTEN, &adc1_cali_chan1_handle);

#if EXAMPLE_USE_ADC2
    //-------------ADC2 Init---------------//
    adc_oneshot_unit_handle_t adc2_handle;
    adc_oneshot_unit_init_cfg_t init_config2 = {
        .unit_id = ADC_UNIT_2,
        .ulp_mode = ADC_ULP_MODE_DISABLE,
    };
    ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config2, &adc2_handle));

    //-------------ADC2 Calibration Init---------------//
    adc_cali_handle_t adc2_cali_handle = NULL;
    bool do_calibration2 = example_adc_calibration_init(ADC_UNIT_2,
EXAMPLE_ADC2_CHAN0, EXAMPLE_ADC_ATTEN, &adc2_cali_handle);

    //-------------ADC2 Config---------------//
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc2_handle,
EXAMPLE_ADC2_CHAN0, &config));
#endif  //#if EXAMPLE_USE_ADC2

     while (1) {
        // Sample ADC1 Channel 0
        ESP_ERROR_CHECK(sample(adc1_handle, EXAMPLE_ADC1_CHAN0, &adc_raw[0]
[0],
                               do_calibration1_chan0 ?
adc1_cali_chan0_handle : NULL,
                               &voltage[0][0]));
        ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1,
EXAMPLE_ADC1_CHAN0, adc_raw[0][0]);
        if (do_calibration1_chan0) {
            ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV",
ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN0, voltage[0][0]);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));

        // Sample ADC1 Channel 1
        ESP_ERROR_CHECK(sample(adc1_handle, EXAMPLE_ADC1_CHAN1, &adc_raw[0]
[1],
                               do_calibration1_chan1 ?
adc1_cali_chan1_handle : NULL,
                               &voltage[0][1]));
```

```
        ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1,
EXAMPLE_ADC1_CHAN1, adc_raw[0][1]);
        if (do_calibration1_chan1) {
            ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV",
ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN1, voltage[0][1]);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));

#if EXAMPLE_USE_ADC2
        // Sample ADC2 Channel 0
        ESP_ERROR_CHECK(sample(adc2_handle, EXAMPLE_ADC2_CHAN0, &adc_raw[1]
[0],
                               do_calibration2 ? adc2_cali_handle : NULL,
                               &voltage[1][0]));
        ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_2 + 1,
EXAMPLE_ADC2_CHAN0, adc_raw[1][0]);
        if (do_calibration2) {
            ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV",
ADC_UNIT_2 + 1, EXAMPLE_ADC2_CHAN0, voltage[1][0]);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
#endif  //#if EXAMPLE_USE_ADC2
    }

    //Tear Down
    ESP_ERROR_CHECK(adc_oneshot_del_unit(adc1_handle));
    if (do_calibration1_chan0) {
        example_adc_calibration_deinit(adc1_cali_chan0_handle);
    }
    if (do_calibration1_chan1) {
        example_adc_calibration_deinit(adc1_cali_chan1_handle);
    }

#if EXAMPLE_USE_ADC2
    ESP_ERROR_CHECK(adc_oneshot_del_unit(adc2_handle));
    if (do_calibration2) {
        example_adc_calibration_deinit(adc2_cali_handle);
    }
#endif //#if EXAMPLE_USE_ADC2
}

/*---------------------------------------------------------------
        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
}
```

Yes, I can definitely do that! Here's the explanation of the ESP32 ADC example code, with all the code blocks removed:

ESP32 Analog-to-Digital Converter (ADC) Example Explained

This C code is designed for **ESP32 microcontrollers** using the **ESP-IDF framework**. Its main purpose is to demonstrate how to perform **Analog-to-Digital Conversion (ADC) readings**, including the crucial aspect of **ADC calibration** for accurate voltage measurements.

Essentially, the program initializes one or two ADC units, configures specific channels, attempts to calibrate them for better accuracy, and then continuously takes samples from these channels. It then prints both the raw ADC readings and, if calibration is successful, the more precise calibrated voltage values to your serial monitor.

Core Functionality

The code is built to achieve the following:

1. **Initialize ADC Units:** It sets up ADC1 and, if the target chip supports it and it's enabled in the configuration, ADC2.
2. **Configure ADC Channels:** It defines which specific analog input pins will be used for measurement and sets their **attenuation**, which determines the maximum voltage range the ADC can accurately measure (e.g., 0-3.9V).
3. **Perform ADC Calibration:** A vital step for accuracy. The code attempts to use Espressif's built-in ADC calibration schemes (like Curve Fitting or Line Fitting) to convert the raw, uncalibrated ADC values into much more precise voltage readings. This is important because the raw readings can vary a bit from one chip to another.
4. **Continuously Sample:** In an endless loop, the program repeatedly reads the analog values from the configured ADC channels at regular intervals.
5. **Report Readings:** It sends the raw ADC values to your serial output. If the calibration was successful, it also sends the corresponding calibrated voltage in millivolts (mV).
6. **De-initialize (Cleanup):** Although in an infinite loop, the code includes the proper steps to release ADC resources and calibration handles, which would be executed if the main loop were to exit.

Detailed Explanation of Code Sections

1. Includes and Configuration Macros: The code starts by including necessary libraries from standard C and the ESP-IDF framework. These include headers for FreeRTOS (for task management), system-on-chip capabilities, logging, and specifically for ADC one-shot readings and calibration. Macros are used to define the specific **GPIO pins** that serve as ADC input channels (these vary by ESP32 chip type) and the ADC attenuation level, which dictates the measurement voltage range. There's also a macro to conditionally enable code for ADC2, as not all ESP32 variants support it or have limitations.

2. `sample` **Function (Helper for ADC Reading):** This is a utility function designed to perform a single ADC reading. It first gets the **raw ADC value**, which is an integer representing the analog measurement. If you've successfully set up ADC calibration, this function then uses the calibration data to convert that raw reading

into a more accurate voltage measurement in millivolts. The function returns a status code to indicate if the operation was successful.

3. `app_main` **Function (Main Application Logic):** This is where your ESP32 application begins.

- **ADC1 Setup:** It initializes **ADC Unit 1** and configures its designated channels (analog input pins) with the chosen attenuation and bit width (e.g., 12-bit resolution).
- **ADC1 Calibration:** It then calls a helper function to attempt to set up calibration for each of the configured ADC1 channels. If successful, it gets a calibration handle that allows for accurate voltage conversion.
- **ADC2 Setup (Conditional):** If the `EXAMPLE_USE_ADC2` macro is active (meaning your ESP32 supports ADC2), it performs similar initialization, configuration, and calibration steps for **ADC Unit 2**.
- **Main Sampling Loop:** The program enters an infinite loop. Inside this loop, it repeatedly calls the `sample` helper function for each configured ADC channel. For each sample, it prints the raw ADC data to the serial monitor. If calibration was successfully set up for that channel, it also prints the calibrated voltage value in millivolts. A one-second delay is introduced between each reading to make the output readable.
- **Resource Teardown:** Although the infinite loop means this part is rarely reached during normal operation, the code includes commands to properly de-initialize the ADC units and calibration handles. This is important for clean resource management if the application were to terminate or switch modes.

4. `example_adc_calibration_init` **Function (Calibration Setup):** This function is responsible for trying to set up ADC calibration for a specific ADC unit, channel, and attenuation. It attempts to use one of two calibration schemes: * **Curve Fitting:** This is generally more accurate and is used on newer ESP32 chips that have pre-programmed calibration data (eFuses) from the factory. * **Line Fitting:** A simpler, linear calibration method that might be used as a fallback if curve fitting isn't supported or fails. The function returns `true` if a calibration handle was successfully created, indicating that accurate voltage conversion is possible. It also provides helpful log messages if calibration succeeds, fails, or is skipped (e.g., if eFuse data isn't present).

5. `example_adc_calibration_deinit` **Function (Calibration Teardown):** This simple function is used to properly release the resources associated with an ADC calibration handle when it's no longer needed, preventing memory leaks.

How to Use This Code

To use this code, you would typically:

1. **Connect Hardware:** Attach analog sensors or voltage sources to the GPIO pins designated as ADC channels in the code (e.g., GPIO4 and GPIO5 for ADC1 channels on a standard ESP32). Ensure your input voltage stays within the ADC's configured range (e.g., 0-3.9V for `ADC_ATTEN_DB_12`).
2. **Compile and Flash:** Use the ESP-IDF tools to compile the code and flash it onto your ESP32 development board.
3. **Open Serial Monitor:** Connect to your ESP32's serial port (usually UART0 via USB) using a serial terminal program (like PuTTY or `idf.py monitor`) and set the baud rate to 115200.

You'll then see the raw ADC readings and, if calibration is active, the corresponding voltage values printed to your serial monitor, updating every second.

This example is a fundamental building block for any ESP32 application that needs to measure analog voltages accurately, whether it's for reading sensor data (like temperature or light) or monitoring power levels.

Do you have any more questions about specific parts of this ADC functionality or how it might be used in a project?