Федеральное государственное бюджетное образовательное учреждение
высшего образования «Воронежский государственный университет»

Отчет по лабораторной работе № 3
По курсу «Введение в интернет вещей»

«Аналого-цифровой преобразователь»

Выполнила:
Меркутова Кристина Денисовна
3 группа

Воронеж 2025

Цель работы:

Изучить работу с Аналого-цифрового преобразователя (АЦП) на ESP32, научиться получать данные с АЦП в двух режимах oneshot_read и continuous_read.

Ход выполнения работы:

1. Ответьте на вопросы в теоретической части максимально подробно.

2. Соберите схему практической части и напишите программу для микроконтроллера. Убедитесь, что код выполняется верно, используйте симулятор wokwi или лабораторный стенд. Скриншоты или фото устройства и листинг программы микроконтроллера обязательны.

1. Теоретическая часть. Ответьте на вопросы:

1. Каково максимальное разрешение АЦП ESP32, и как можно изменить его программно при использовании различных сред разработки; **Ответ:** максимальное разрешение АЦП ESP32 составляет 12 бит. Это означает, что АЦП может выдать $2^{12} = 4096$ дискретных значений, которые в идеале соответствуют диапазону входных напряжений от 0 В до опорного напряжения.
2. Как калибровка АЦП (например, с помощью функции adc1_calibrate() в ESP-IDF) повышает точность измерений, и какие источники погрешности существуют у встроенного АЦП ESP32; **Ответ:** adc_calibrate()**:** Функция считывает эти коэффициенты из eFuse и инициализирует структуру калибровки. Затем, при чтении АЦП, raw-значение пропускается через функцию компенсации (например, adc1_get_voltage()), которая использует эти коэффициенты для линейной коррекции результата. Это позволяет компенсировать ошибку усиления и смещения АЦП, приводя raw-значения к более точным значениям в милливольтах, калибровка компенсирует только систематические ошибки смещения и усиления, но не устраняет шум и нелинейность. Для высокоточных измерений требуются внешние меры: фильтрация питания,

использование внешнего Vref, усреднение множества отсчетов и экранирование.

3. Какие ограничения накладывает использование Wi-Fi или Bluetooth на работу АЦП2 в ESP32, и почему рекомендуется использовать ADC1 для приложений с активной беспроводной связью; **Ответ:** ограничения ADC2: прямой конфликт доступа: АЦП2 (каналы GPIO0, 2, 4, 12-15, 25–27) разделяет ресурсы с беспроводным модулем (Wi-Fi и Bluetooth).
   Блокировка: когда драйвер Wi-Fi или Bluetooth активен, он монопольно захватывает доступ к ADC2. Любая попытка чтения с канала ADC2 в это время завершится ошибкой (ESP_ERR_TIMEOUT). ADC1 рекомендуется использовать: ADC1 (каналы GPIO32-39) имеет отдельный контроллер и не имеет прямого конфликта доступа с беспроводным модулем, это позволяет надежно читать аналоговые сигналы в фоновом режиме, пока активно Wi-Fi или Bluetooth соединение.

4. Опишите основные характеристики АЦП, преобразователь какого типа (архитектуры) используется в esp32. Ответ:
   Основные характеристики:

- **Тип АЦП: Последовательного приближения (SAR - Successive Approximation Register)**.
- **Разрешение:** Программно-настраиваемое: 9, 10, 11 или 12 бит.
- **Количество каналов:** 2 независительных АЦП (ADC1 и ADC2), каждый с 8/10 мультиплексируемыми каналами.
  - ADC1: 8 каналов (GPIO32 - GPIO39)
  - ADC2: 10 каналов (GPIO0, 2, 4, 12-15, 25-27)
- **Опорное напряжение (Vref):** Внутреннее, номинально ~1.1 В. Имеет разброс от чипа к чипу.
- **Входной диапазон напряжений:**
  - Без ослабления (0 дБ): ~0 - ~1.1 В (Vref).
  - С программируемым аттенюатором: до ~0 - ~3.3 В (при аттенюации 11 дБ).
- **Аттенюация:** Программно-настраиваемая для каждого канала: 0 дБ, 2.5 дБ, 6 дБ, 11 дБ.
- **Скорость преобразования:** Максимальная частота дискретизации ~200 кВыборок/с (при 12 битах). Скорость увеличивается при уменьшении разрешения.

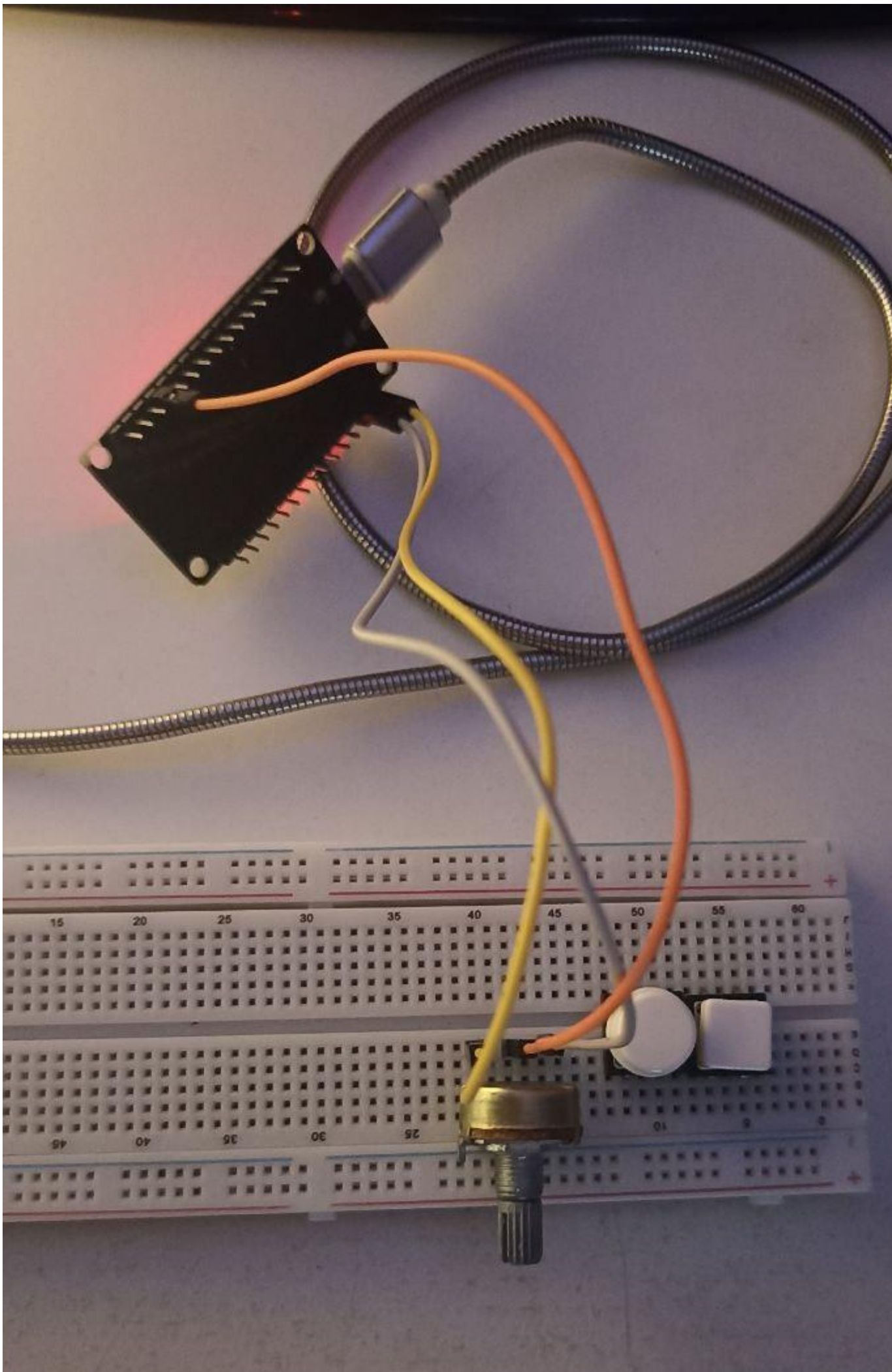
Архитектура АЦП:

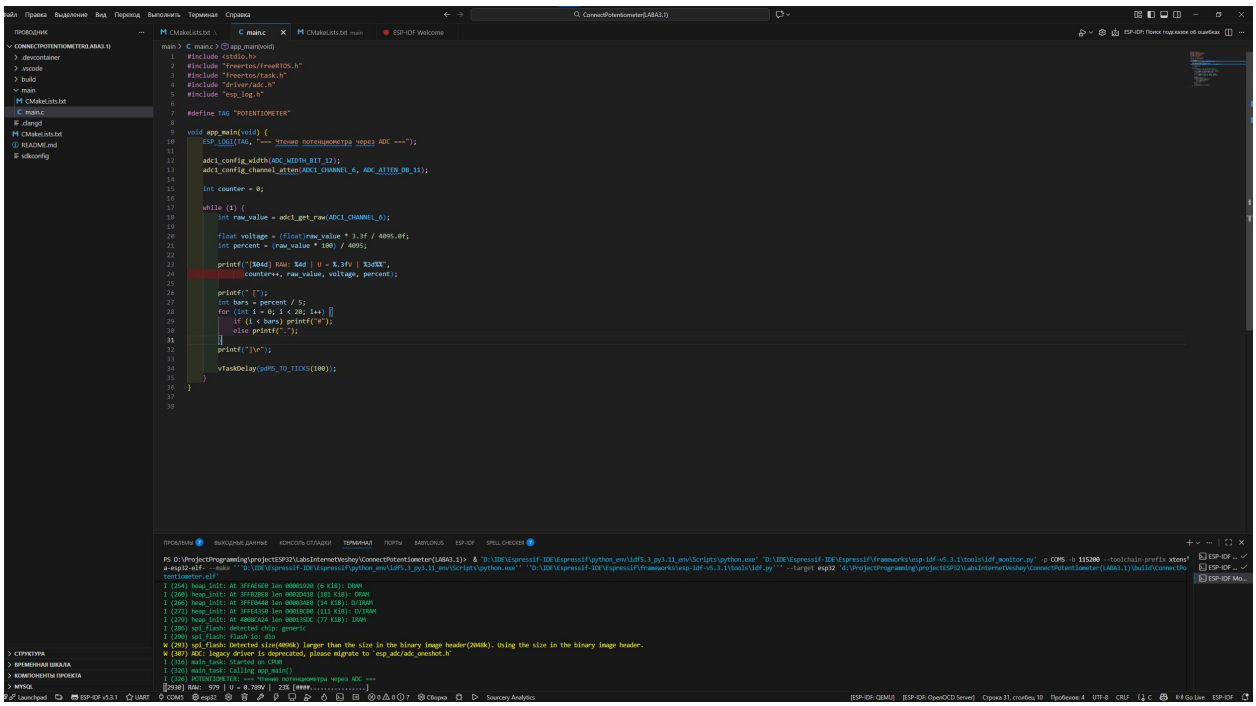
В ESP32 используется классический **SAR ADC**. Принцип его работы:

1. **Выборка и хранение:** Входной сигнал запоминается на внутреннем конденсаторе (Sample-and-Hold circuit).
2. **Бинарный поиск:** Цифро-аналоговый преобразователь (ЦАП) внутри SAR АЦП генерирует напряжение, соответствующее середине текущего диапазона поиска (начинается со старшего бита).
3. **Сравнение:** Сгенерированное ЦАП напряжение сравнивается с запомненным входным напряжением с помощью компаратора.
4. **Принятие решения:** Результат сравнения (1 если V_ЦАП < V_вх, иначе 0) устанавливает текущий бит в регистре последовательного приближения (SAR).
5. **Итерация:** Процесс повторяется для следующего по старшинству бита, и так до младшего бита. После N итераций (где N - разрядность) в регистре SAR оказывается готовое цифровое значение.

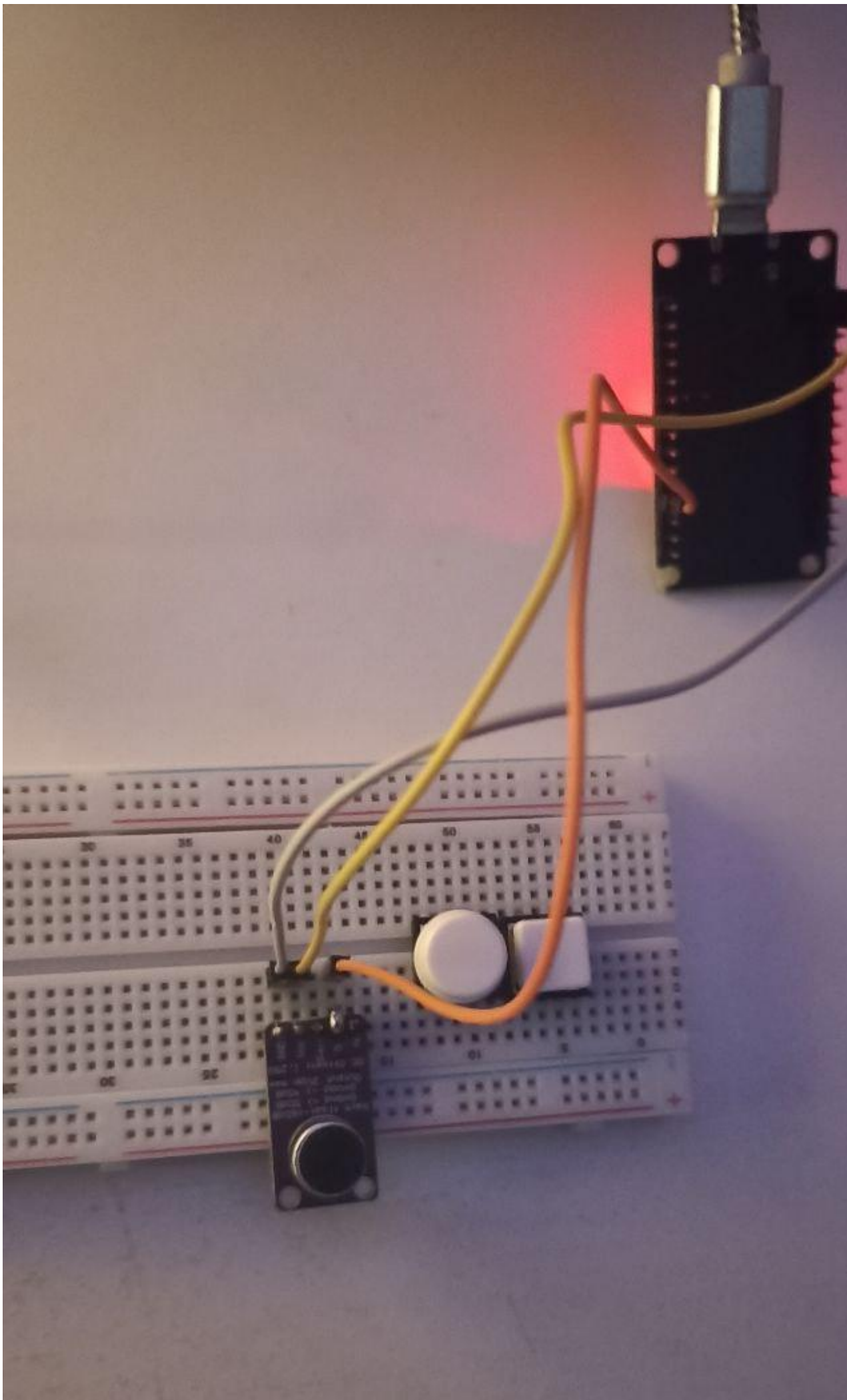
2. Практическая часть:

1. Соберите схему, использующую потенциометр на тестовом стенде или в wokwi. Подключите выход потенциометра к одному из выводов контроллера, работающего на ADC_1. Напишите код, отправляющий данные канала АЦП в консоль используя oneshot_read .

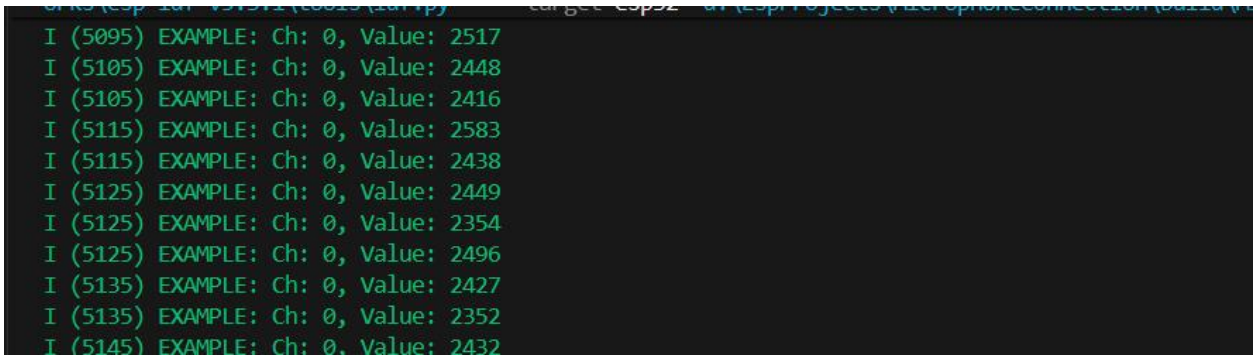




2. Соберите схему, использующую микрофон на модуле max9814используя тестовый стенд (данный пример на работает в wokwi). Попробуйте получать данные с микрофона с частотой боле 10кГц. Данные выводите в консоль.



```c

#include <string.h>
#include <stdio.h>
#include "sdkconfig.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "esp_adc/adc_continuous.h"

#define EXAMPLE_ADC_UNIT                    ADC_UNIT_1
#define _EXAMPLE_ADC_UNIT_STR(unit)         #unit
#define EXAMPLE_ADC_UNIT_STR(unit)          _EXAMPLE_ADC_UNIT_STR(unit)
#define EXAMPLE_ADC_CONV_MODE               ADC_CONV_SINGLE_UNIT_1
#define EXAMPLE_ADC_ATTEN                   ADC_ATTEN_DB_0
#define EXAMPLE_ADC_BIT_WIDTH               SOC_ADC_DIGI_MAX_BITWIDTH

#if CONFIG_IDF_TARGET_ESP32 || CONFIG_IDF_TARGET_ESP32S2
#define EXAMPLE_ADC_OUTPUT_TYPE             ADC_DIGI_OUTPUT_FORMAT_TYPE1
#define EXAMPLE_ADC_GET_CHANNEL(p_data)     ((p_data)->type1.channel)
#define EXAMPLE_ADC_GET_DATA(p_data)        ((p_data)->type1.data)
#else
#define EXAMPLE_ADC_OUTPUT_TYPE             ADC_DIGI_OUTPUT_FORMAT_TYPE2
#define EXAMPLE_ADC_GET_CHANNEL(p_data)     ((p_data)->type2.channel)
#define EXAMPLE_ADC_GET_DATA(p_data)        ((p_data)->type2.data)
#endif

#define EXAMPLE_READ_LEN                    256

#if CONFIG_IDF_TARGET_ESP32
static adc_channel_t channel[1] = {ADC_CHANNEL_0};
#else
static adc_channel_t channel[2] = {ADC_CHANNEL_0, ADC_CHANNEL_3};
#endif

static TaskHandle_t s_task_handle;
static const char *TAG = "EXAMPLE";

static bool IRAM_ATTR s_conv_done_cb(adc_continuous_handle_t handle, const adc_continuous_evt_data_t *edata, void *user_data)
{
    BaseType_t mustYield = pdFALSE;
    vTaskNotifyGiveFromISR(s_task_handle, &mustYield);

    return (mustYield == pdTRUE);
}

static void continuous_adc_init(adc_channel_t *channel, uint8_t channel_num, adc_continuous_handle_t *out_handle)
{
    adc_continuous_handle_t handle = NULL;

    adc_continuous_handle_cfg_t adc_config = {
        .max_store_buf_size = 1024,
        .conv_frame_size = EXAMPLE_READ_LEN,
    };
    ESP_ERROR_CHECK(adc_continuous_new_handle(&adc_config, &handle));

    adc_continuous_config_t dig_cfg = {
        .sample_freq_hz = 50 * 1000,
        .conv_mode = EXAMPLE_ADC_CONV_MODE,
        .format = EXAMPLE_ADC_OUTPUT_TYPE,
    };

    adc_digi_pattern_config_t adc_pattern[SOC_ADC_PATT_LEN_MAX] = {0};
    dig_cfg.pattern_num = channel_num;
    for (int i = 0; i < channel_num; i++) {
        adc_pattern[i].atten = EXAMPLE_ADC_ATTEN;
        adc_pattern[i].channel = channel[i] & 0x7;
        adc_pattern[i].unit = EXAMPLE_ADC_UNIT;
        adc_pattern[i].bit_width = EXAMPLE_ADC_BIT_WIDTH;

        ESP_LOGI(TAG, "adc_pattern[%d].atten is :%"PRIx8, i, adc_pattern[i].atten);
        ESP_LOGI(TAG, "adc_pattern[%d].channel is :%"PRIx8, i, adc_pattern[i].channel);
        ESP_LOGI(TAG, "adc_pattern[%d].unit is :%"PRIx8, i, adc_pattern[i].unit);
    }
    dig_cfg.adc_pattern = adc_pattern;
    ESP_ERROR_CHECK(adc_continuous_config(handle, &dig_cfg));

    *out_handle = handle;
}

void app_main(void)
{
    esp_err_t ret;
    uint32_t ret_num = 0;
    uint8_t result[EXAMPLE_READ_LEN] = {0};
    memset(result, 0xcc, EXAMPLE_READ_LEN);

    s_task_handle = xTaskGetCurrentTaskHandle();

    adc_continuous_handle_t handle = NULL;
    continuous_adc_init(channel, sizeof(channel) / sizeof(adc_channel_t), &handle);

    adc_continuous_evt_cbs_t cbs = {
        .on_conv_done = s_conv_done_cb,
    };
    ESP_ERROR_CHECK(adc_continuous_register_event_callbacks(handle, &cbs, NULL));
    ESP_ERROR_CHECK(adc_continuous_start(handle));

    while (1) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        char unit[] = EXAMPLE_ADC_UNIT_STR(EXAMPLE_ADC_UNIT);

        while (1) {
            ret = adc_continuous_read(handle, result, EXAMPLE_READ_LEN, &ret_num, 0);
            if (ret == ESP_OK) {
                ESP_LOGI("TASK", "ret is %x, ret_num is %"PRIu32" bytes", ret, ret_num);
                for (int i = 0; i < ret_num; i += SOC_ADC_DIGI_RESULT_BYTES) {
                    adc_digi_output_data_t *p = (adc_digi_output_data_t*)&result[i];
                    uint32_t chan_num = EXAMPLE_ADC_GET_CHANNEL(p);
                    uint32_t data = EXAMPLE_ADC_GET_DATA(p);

                    if (chan_num < SOC_ADC_CHANNEL_NUM(EXAMPLE_ADC_UNIT)) {
                        ESP_LOGI(TAG, "Ch: %"PRIu32", Value: %4"PRIu32, chan_num, data);
                    }
                }
                vTaskDelay(1);
            } else if (ret == ESP_ERR_TIMEOUT) {
                break;
            }
        }
    }

    ESP_ERROR_CHECK(adc_continuous_stop(handle));
    ESP_ERROR_CHECK(adc_continuous_deinit(handle));
}

```


I (5095) EXAMPLE: Ch: 0, Value: 2517
I (5105) EXAMPLE: Ch: 0, Value: 2448
I (5105) EXAMPLE: Ch: 0, Value: 2416
I (5115) EXAMPLE: Ch: 0, Value: 2583
I (5115) EXAMPLE: Ch: 0, Value: 2438
I (5125) EXAMPLE: Ch: 0, Value: 2449
I (5125) EXAMPLE: Ch: 0, Value: 2354
I (5125) EXAMPLE: Ch: 0, Value: 2496
I (5135) EXAMPLE: Ch: 0, Value: 2427
I (5135) EXAMPLE: Ch: 0, Value: 2352
I (5145) EXAMPLE: Ch: 0, Value: 2432