VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

ELEKTRONIKOS FAKULTETAS

ELEKTRONIKOS INŽINERIJOS KATEDRA

REALAUS LAIKO OPERACINĖS SISTEMOS

Laboratorinis darbas 2

Atliko: EKSFM-24 gr. studentas Ignas Malinauskas

Tikrino: dr. Saulius Sakavičius

VILNIUS 2025

1. ADC keitiklis

Darbas atliktas Visual Studio Code aplinkoje per ESP-IDF karkasą.

Programos kodas:

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/adc.h"
#include "esp_adc_cal.h"
#include "driver/uart.h"
#include <inttypes.h>

#define SAMPLE_RATE_MS 100   // 10 Hz sampling
#define ADC_CHANNEL ADC1_CHANNEL_5  // GPIO6
#define ATTEN ADC_ATTEN_DB_11 // 0-3V
#define ADC_UNIT ADC_UNIT_1 //ADC1
#define DEFAULT_VREF 1100 //reference voltage for calibration

static const char *TAG = "ADC_SAMPLE"; //for ESP_LOG messages

QueueHandle_t adc_queue; // global handle for FreeRTOS queue,
                         // transports data between tasks
// Calibration
esp_adc_cal_characteristics_t adc_chars;

// Sampling Task
void adc_sampling_task(void *arg) {
    uint32_t adc_reading = 0;

    while (1) {
        adc_reading = adc1_get_raw(ADC_CHANNEL); //read ADC values
        xQueueSend(adc_queue, &adc_reading, portMAX_DELAY); //store in queue
        vTaskDelay(pdMS_TO_TICKS(SAMPLE_RATE_MS)); //delay so sample rate is as
described
    }
}
// Data Sending Task (UART)
void uart_send_task(void *arg) {
    uint32_t sample;

    while (1) {
        if (xQueueReceive(adc_queue, &sample, portMAX_DELAY)) { //blocks until
new sample arrives
            uint32_t voltage = esp_adc_cal_raw_to_voltage(sample, &adc_chars);
//converting to volts for readability
            printf("ADC Raw: %" PRIu32 ", Voltage: %" PRIu32 "mV\n", sample,
voltage); //sends data to serial monitor via UART
        }
    }
}
```

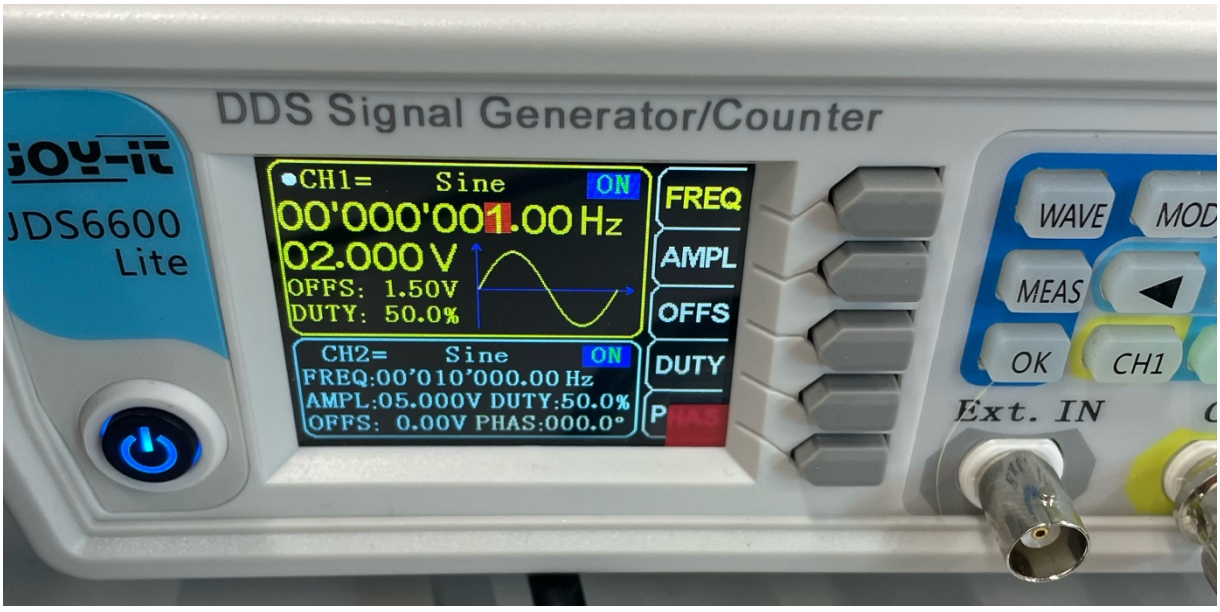
```
void app_main(void) {
    // Configure ADC
    adc1_config_width(ADC_WIDTH_BIT_12); //12bit resolution
    adc1_config_channel_atten(ADC_CHANNEL, ATTEN); //11db to channel 5
    esp_adc_cal_characterize(ADC_UNIT, ATTEN, ADC_WIDTH_BIT_12,
DEFAULT_VREF, &adc_chars); //set ADC characteristics

    adc_queue = xQueueCreate(10, sizeof(uint32_t)); //create queue for 10
samples

    // Create tasks
    xTaskCreate(adc_sampling_task, "adc_sampling_task", 2048, NULL, 5,
NULL);
    xTaskCreate(uart_send_task, "uart_send_task", 2048, NULL, 5, NULL);
}
```

Python kodas, kuris nuskaito duomenų srautą per UART *portą*, atlieka FFT ir atvaizduoja grafikus:

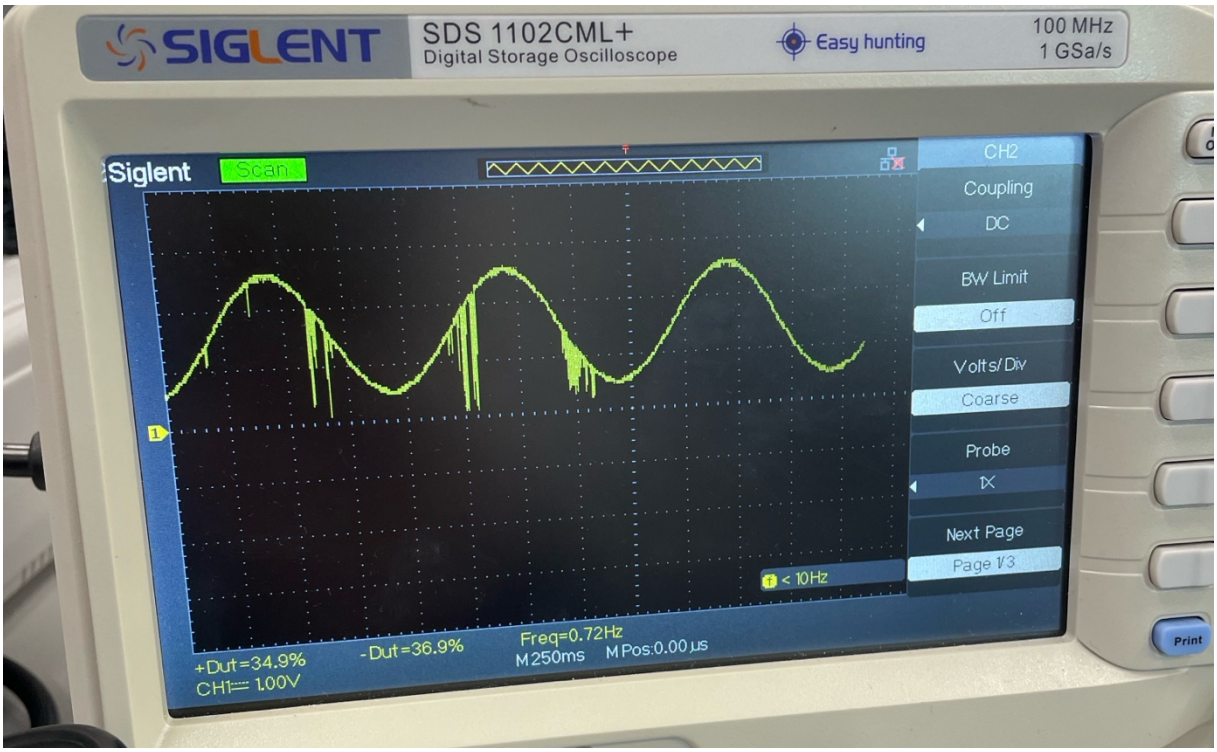
```
import serial
import numpy as np
import matplotlib.pyplot as plt

# --- Parameters ---
PORT = '/dev/tty.usbmodem1101'
BAUD = 115200
SAMPLES = 1024              # Number of samples to collect
SAMPLING_RATE = 10          # In Hz (must match ESP32's sampling rate)

# --- Read Data from Serial ---
adc_values = []
with serial.Serial(PORT, BAUD, timeout=1) as ser:
    print("Collecting data...")
    while len(adc_values) < SAMPLES:
        line = ser.readline().decode('utf-8').strip()
        print(line)
        try:
            if line:
                parts = line.split(',')
                if len(parts) >= 2:
                    raw = int(parts[0].split(':')[1].strip())
                    adc_values.append(raw)
        except:
            continue

adc_array = np.array(adc_values)
adc_centered = adc_array - np.mean(adc_array)  # remove DC offset
```

```python
# -- Time Vector for Time-Domain Plot ---
time_vector = np.arange(SAMPLES) / SAMPLING_RATE

# --- Perform FFT ---
fft_vals = np.fft.fft(adc_centered)
fft_freqs = np.fft.fftfreq(SAMPLES, d=1/SAMPLING_RATE)
amplitudes = np.abs(fft_vals)[:SAMPLES // 2]
frequencies = fft_freqs[:SAMPLES // 2]

# --- Plot Time-Domain and Frequency-Domain ---
plt.figure(figsize=(12, 6))

# Time-domain plot
plt.subplot(2, 1, 1)
plt.plot(time_vector, adc_array)
plt.title("ADC Signal - Time Domain")
plt.xlabel("Time (s)")
plt.ylabel("ADC Value")
plt.grid(True)

# Frequency-domain plot
plt.subplot(2, 1, 2)
plt.plot(frequencies, amplitudes)
plt.title("Amplitude Spectrum - Frequency Domain")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.grid(True)

plt.tight_layout()
plt.show()
```

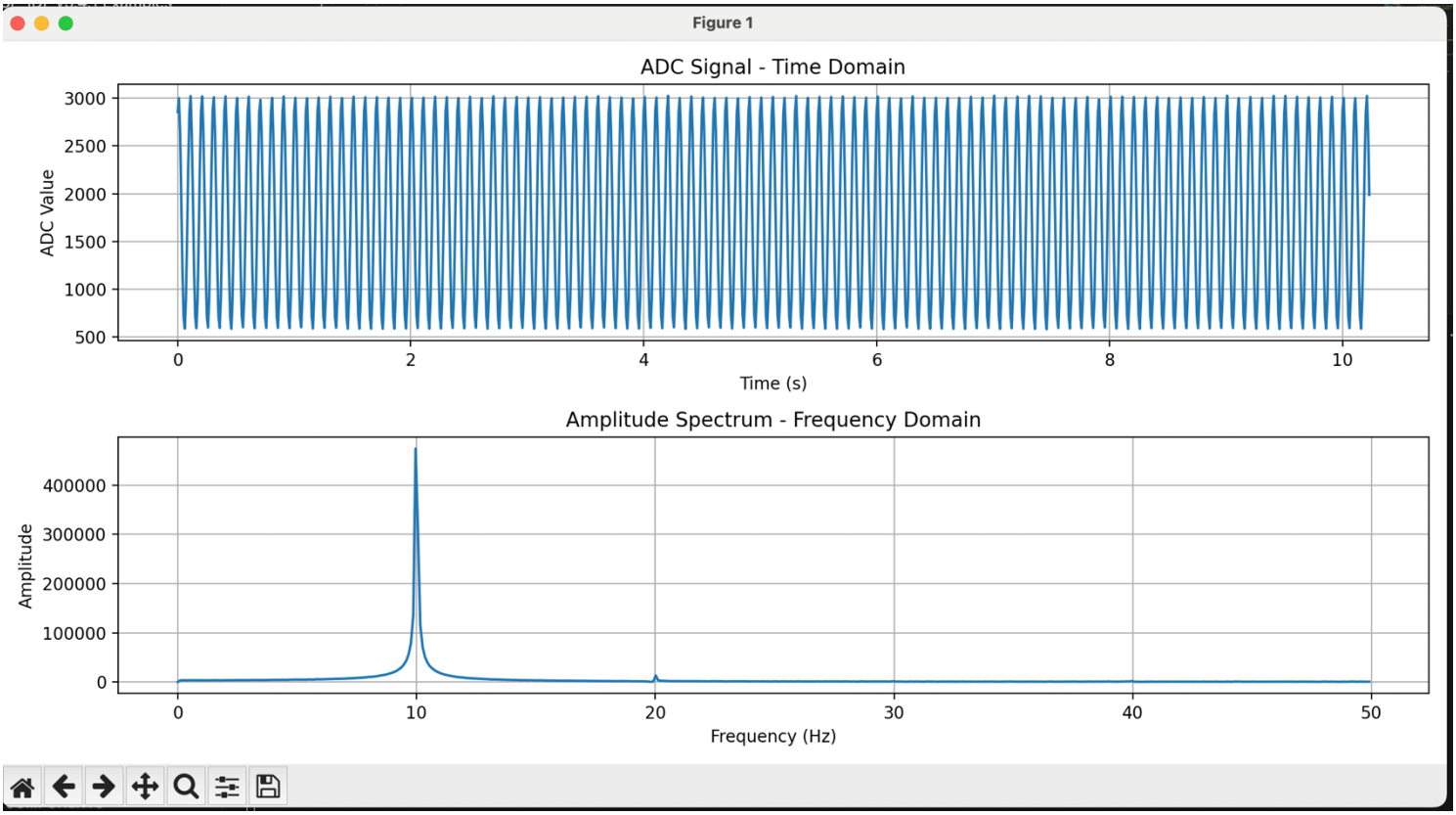
Funkcinio generatoriaus nustatymai:



Osciloskopo rodmenys:



Gautų ADC reikšmių laiko srity ir spektro grafikai:



Išvada

Darbą atlikti pavyko sėkmingai, gautos signalo reikšmės per įgyvendintą ADC keitiklį gana tiksliai atitinka osciloskopo rodmenis.

2. DAC

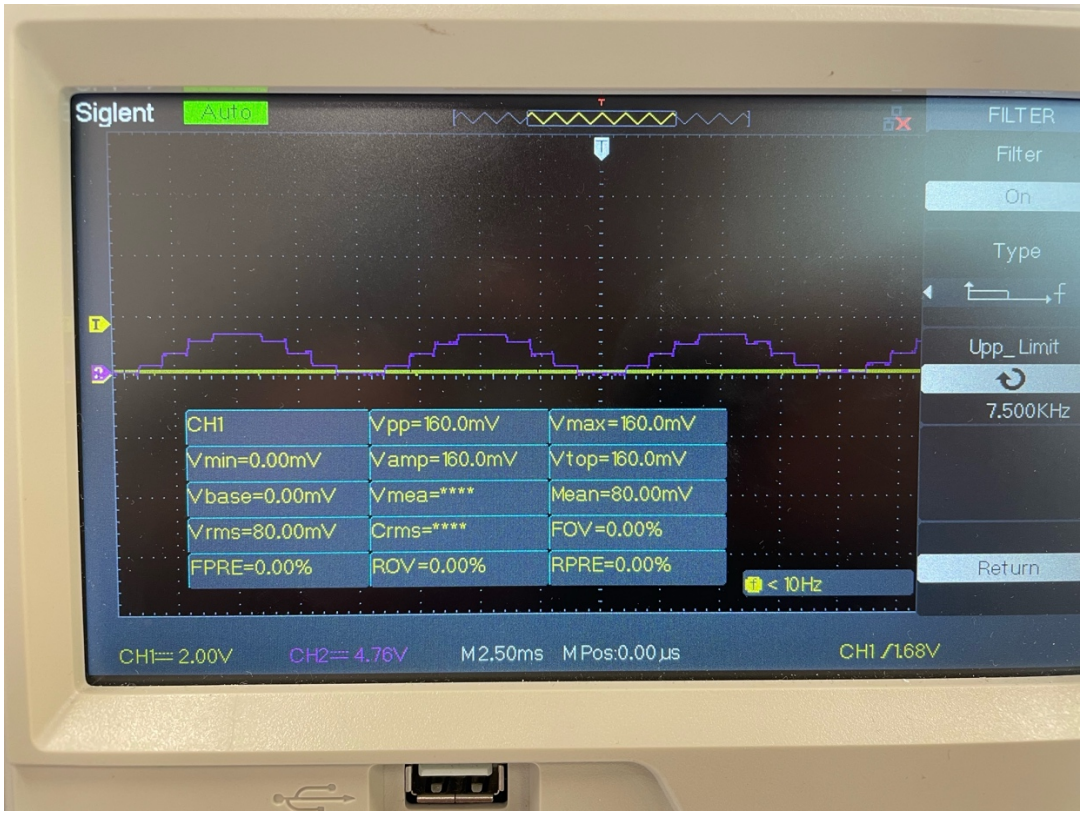
Programos kodas:

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "driver/ledc.h"

#define LED_GPIO            6          // output pin
#define PWM_TIMER_HZ        20000      // PWM carrier
#define PWM_RES_BITS        11         // 0 … 2047
#define SAMPLE_RATE_HZ      1000       // 1 kSa/s
#define OUTPUT_FREQ_HZ      100        // breathing frequency
#define LUT_BITS            8
#define LUT_SIZE            (1U << LUT_BITS)
#define PWM_DUTY_MAX        ((1U << PWM_RES_BITS) - 1)

static const char *TAG = "LED_BREATH";

/* ---------- full-cycle sine lookup table (256 samples, 0 … 2047) ------
---- */
static const uint16_t sin_full[LUT_SIZE] = {
     1024, 1049, 1074, 1099, 1124, 1149, 1174, 1198,
     1223, 1248, 1272, 1296, 1321, 1345, 1368, 1392,
     1415, 1438, 1461, 1484, 1506, 1528, 1550, 1571,
     1592, 1613, 1633, 1653, 1673, 1692, 1711, 1729,
     1747, 1765, 1782, 1799, 1815, 1830, 1846, 1860,
     1875, 1888, 1901, 1914, 1926, 1938, 1949, 1959,
     1969, 1978, 1987, 1995, 2003, 2010, 2016, 2022,
     2027, 2032, 2036, 2039, 2042, 2044, 2046, 2047,
     2047, 2047, 2046, 2044, 2042, 2039, 2036, 2032,
     2027, 2022, 2016, 2010, 2003, 1995, 1987, 1978,
     1969, 1959, 1949, 1938, 1926, 1914, 1901, 1888,
     1875, 1860, 1846, 1830, 1815, 1799, 1782, 1765,
     1747, 1729, 1711, 1692, 1673, 1653, 1633, 1613,
     1592, 1571, 1550, 1528, 1506, 1484, 1461, 1438,
     1415, 1392, 1368, 1345, 1321, 1296, 1272, 1248,
     1223, 1198, 1174, 1149, 1124, 1099, 1074, 1049,
     1024,  998,  973,  948,  923,  898,  873,  849,
      824,  799,  775,  751,  726,  702,  679,  655,
      632,  609,  586,  563,  541,  519,  497,  476,
      455,  434,  414,  394,  374,  355,  336,  318,
      300,  282,  265,  248,  232,  217,  201,  187,
      172,  159,  146,  133,  121,  109,   98,   88,
       78,   69,   60,   52,   44,   37,   31,   25,
       20,   15,   11,    8,    5,    3,    1,    0,
        0,    0,    1,    3,    5,    8,   11,   15,
       20,   25,   31,   37,   44,   52,   60,   69,
       78,   88,   98,  109,  121,  133,  146,  159,
      172,  187,  201,  217,  232,  248,  265,  282,
      300,  318,  336,  355,  374,  394,  414,  434,
      455,  476,  497,  519,  541,  563,  586,  609,
      632,  655,  679,  702,  726,  751,  775,  799,
      824,  849,  873,  898,  923,  948,  973,  998
};
```

```c
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "driver/ledc.h"

#define LED_GPIO            6          // connect LED (+
resistor) here
#define PWM_TIMER_HZ        20000      // PWM carrier
#define PWM_RES_BITS        11         // 0 … 2047

/* ---------- single-lookup DDS sample ------------------------
-- */
static inline uint16_t IRAM_ATTR dds_sample(uint32_t phase)
{
    uint32_t idx = phase >> (32 - LUT_BITS);      // top-8 bits →
0 … 255
    return sin_full[idx];
}

/* ---------- LEDC PWM initialisation -------------------------
-- */
static void ledc_init(void)
{
    ledc_timer_config_t t = {
        .speed_mode      = LEDC_LOW_SPEED_MODE,
        .timer_num       = LEDC_TIMER_0,
        .duty_resolution = PWM_RES_BITS,
        .freq_hz         = PWM_TIMER_HZ,
        .clk_cfg         = LEDC_AUTO_CLK,
    };
    ledc_timer_config(&t);

    ledc_channel_config_t c = {
        .speed_mode = LEDC_LOW_SPEED_MODE,
        .channel    = LEDC_CHANNEL_0,
        .timer_sel  = LEDC_TIMER_0,
        .gpio_num   = LED_GPIO,
        .duty       = 0,
        .hpoint     = 0,
    };
    ledc_channel_config(&c);
}

/* ---------- FreeRTOS task: 1 kSa/s DDS ----------------------
-- */
static void IRAM_ATTR dds_task(void *arg)
{
    const uint32_t phase_inc =
        (uint32_t)(((double)(1ULL << 32) * OUTPUT_FREQ_HZ) /
SAMPLE_RATE_HZ);

    uint32_t   phase        = 0;
    TickType_t tick_period  = pdMS_TO_TICKS(1000 /
SAMPLE_RATE_HZ);
    TickType_t next_wakeup  = xTaskGetTickCount();
```

```
    while (true) {
        uint16_t duty = dds_sample(phase);

        ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, duty);
        ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0);

        phase += phase_inc;
        vTaskDelayUntil(&next_wakeup, tick_period);
    }
}

/* ---------- app entry ----------------------------------------
- */
void app_main(void)
{
    //ESP_LOGI(TAG, "Starting LED breathing demo (%.1f Hz)…",
OUTPUT_FREQ_HZ);
    ledc_init();

    xTaskCreatePinnedToCore(dds_task, "dds_led",
                            2048, NULL, configMAX_PRIORITIES - 1,
                            NULL, 0);
}
```

Osciloskopo rodemenys:



Išvada

Atliekant DDS buvo susidurta su problema kai sinusinės bangos LUT buvo sudarytas iš ketvirčio bangos reikšmių, rezultate išvedamas signalas buvo netolygi sinusinė – tam tikrom

vietom banga nusikirsdavo – iš didžiausios reikšmės nukrisdavo į mažiausią, pakeitus LUT į sinusinės bangos 256 dydžio reikšmių lentelė problema susitvarkė ir uždėjus skaitmeninį filtrą per osciloskopa gaunama sinusinė banga. Galutinis signalas laiptuotas, nes fazės akumuliatoriaus žingsnis per didelis, kode jis apibrėžiamas kaip 2^{32} dydžio, pakeitus į 8 bitų dydį problema turėtų išsispręst.