Homework: Generate an Audio Signal in Python, Bias with RC Network, and Acquire on ESP32

Objective

You will:

a) Generate an audio signal in Python using `numpy` and save it as WAV using `soundfile`.

b) Install and use `sox` (`play`) to listen to the WAV from the command line.

c) Build a simple RC bias circuit (series capacitor + resistor bias) to add a DC offset and feed the signal into an ESP32 ADC input.

d) On the ESP32 firmware, experiment with `EXAMPLE_READ_LEN`, `SAMPLE_FREQ_HZ`, and `PRINT_RATE_HZ`.

e) Research how to change the UART `printf` baud rate in ESP-IDF, apply it, and match the baud in the provided Python plotting script.

Prerequisites & Install

1. Python packages (inside your virtual environment):

```
python -m pip install --upgrade pip
python -m pip install numpy soundfile pyserial dash plotly
# On some Linux distros you may need libsndfile:
sudo apt-get install -y libsndfile1
```

2. Command-line audio player (`sox` provides `play`):

```
sudo apt-get update
sudo apt-get install -y sox
```

3. ESP-IDF installed and working (`idf.py`, `menuconfig`, `flash`, `monitor`).

Part A — Generate and Save Audio (Python)

Create a short sine or arbitrary waveform, save it as WAV with `soundfile`.

Listing 1: Minimal NumPy + soundfile WAV writer

```python
import numpy as np
import soundfile as sf

fs = 48_000 # sample rate (Hz)
f = 440.0 # sine frequency (Hz)
dur = 2.0 # seconds

```

```python
t = np.arange(int(fs*dur)) / fs
x = 0.2*np.sin(2*np.pi*f*t).astype(np.float32) # float32 in [-1, 1]

sf.write("tone.wav", x, fs, subtype="PCM_16")
print("Wrote tone.wav @", fs, "Hz")
```

Part B — Play the WAV with `play` (sox)

Listing 2: Listen to your file

```
play tone.wav
```

Part C — Hardware: Raise DC Offset with RC Network

Most audio signals are centered at 0 V (AC). The ESP32 ADC expects a voltage between ≈ 0 and ≈ 3.3 V. We will AC-couple the audio and bias it around mid-supply.

Schematic (textual)

```
PC/USB audio out  -- C (1 µF to 10 µF, series) --o----> ESP32 ADC1_CH6 (GPIO34)
                                           |     |
                                           |     +-- Rb (100 k) --> GND
                                           |
                              3.3V -- R1 (100 k) --+
                                                    |---- Vbias node (1.65 V)
                               GND -- R2 (100 k) --+
                                            |
                                           Cb (0.1 µF) to GND (bypass)
```

Notes:

- Choose $R1 = R2 = 100\,\text{k}\Omega$ to make $V_{\text{bias}} \approx 1.65\,\text{V}$.
- Use $Rb \approx 100\,\text{k}\Omega$ from the ADC node to V_{bias} (or tie the node directly to V_{bias} through $100\,\text{k}\Omega$).
- Series capacitor C with Rb sets a high-pass cutoff: $f_c \approx \frac{1}{2\pi R_{\text{eq}} C}$. With $R_{\text{eq}} \approx 100\,\text{k}\Omega$ and $C = 1\,\mu\text{F}$, $f_c \approx 1.6\,\text{Hz}$.
- Keep the *AC amplitude* small enough so that $V_{\text{bias}} \pm \frac{V_{pp}}{2}$ stays within the ADC range (avoid clipping).
- Share ground between the audio source and the ESP32.

Part D — ESP32 Firmware Setup and Parameters

Wire the biased signal to **ADC1_CH6** (GPIO34) and optionally a second channel to **ADC1_CH7** (GPIO35). Build, flash, and monitor your app.

Parameters to experiment with

- `EXAMPLE_READ_LEN` → ADC driver frame size (bytes).
- `SAMPLE_FREQ_HZ` → total ADC sampling frequency.
- `PRINT_RATE_HZ` → approximately how many printed lines per second (per channel after decimation).

Change UART `printf` Baud Rate (research task)

Find how to change the console baud in ESP-IDF (`menuconfig`). Hint: look under *Component config* → *ESP System Settings* for *UART console baud rate*. After changing it, ensure:

1. Your serial monitor/terminal matches this baud.
2. You update the `BAUD` variable in the Python plotter script (Section **??**).

Build/Flash/Monitor

Listing 3: Typical ESP-IDF workflow

```
idf.py set-target esp32
idf.py menuconfig
idf.py build
idf.py -p /dev/ttyUSB0 flash
idf.py -p /dev/ttyUSB0 monitor --baud 115200
```

Part E — Python Serial Plotter

Set the correct serial port and baud to match your firmware. `FS_PRINT_HZ` should reflect your effective print rate (after decimation).

Listing 4: Dash serial plotter (time + spectrum)

```
import threading, queue, time, sys
from collections import deque
import numpy as np
import serial
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

from dash import Dash, dcc, html, Input, Output
import plotly.graph_objs as go

# -------- Match these to your firmware --------
PORT = '/dev/ttyUSB0'
BAUD = 115200
N = 1024 # samples in scrolling window
N_CHANNELS = 2 # values per line: "v0 v1"
FS_PRINT_HZ = 200.0 # ~ lines/sec PER CHANNEL after decimation
FFT_UPDATE_EVERY = 0.25 # s
# --------------------------------------------

buffers = [deque([0.0]*N, maxlen=N) for _ in range(N_CHANNELS)]
ser = serial.Serial(PORT, BAUD, timeout=0.1)

```

```python
def serial_reader():
    while True:
        try:
            raw = ser.readline().decode(errors='ignore').strip()
            if not raw:
                continue
            parts = raw.split()
            if len(parts) < N_CHANNELS: # skip malformed lines
                continue
            vals = [float(x) for x in parts[:N_CHANNELS]]
            for b, v in zip(buffers, vals):
                b.append(v)
        except Exception:
            pass

t = threading.Thread(target=serial_reader, daemon=True)
t.start()

def compute_fft(y, fs):
    y = np.asarray(y, dtype=float)
    y = y - y.mean()
    w = np.hanning(len(y))
    yw = y * w
    spec = np.fft.rfft(yw)
    cg = w.sum()/len(w) # coherent gain (Hann)
    mag = np.abs(spec)/(len(y)*cg)
    mag_db = 20*np.log10(mag + 1e-12)
    f = np.fft.rfftfreq(len(y), d=1.0/fs)
    return f, mag_db

app = Dash(__name__)
app.layout = html.Div([
    html.H3("ESP32 Serial Plotter (Time + Frequency)"),
    html.Div([
        html.Label("Channel for spectrum:"),
        dcc.Dropdown(
            id='ch-select',
            options=[{'label': f'Channel {i}', 'value': i} for i in range(N_CHANNELS)],
            value=0, clearable=False, style={'width':'200px'}
        )
    ], style={'marginBottom':'10px'}),

    dcc.Graph(id='time-graph'),
    dcc.Graph(id='freq-graph'),

    dcc.Interval(id='timer', interval=1000/30, n_intervals=0), # ~30 FPS
    dcc.Interval(id='fft-timer', interval=int(FFT_UPDATE_EVERY*1000), n_intervals=0)
])

@app.callback(Output('time-graph','figure'), Input('timer','n_intervals'))
def update_time(_):
    x = np.arange(N)/FS_PRINT_HZ
    data = []
    for i, b in enumerate(buffers):
        y = np.array(b, dtype=float)
        data.append(go.Scatter(x=x, y=y, mode='lines', name=f'Ch {i}'))
    fig = go.Figure(data=data)
    fig.update_layout(margin=dict(l=40,r=10,t=30,b=40))
```

```
    fig.update_xaxes(title="Time (s)")
    fig.update_yaxes(title="ADC Codes")
    return fig

@app.callback(Output('freq-graph','figure'),
              [Input('fft-timer','n_intervals'), Input('ch-select','value')])
def update_fft(_, ch):
    y = np.array(buffers[ch], dtype=float)
    f, mag_db = compute_fft(y, FS_PRINT_HZ)
    peak_idx = int(np.argmax(mag_db))
    peak_f = float(f[peak_idx]); peak_db = float(mag_db[peak_idx])

    fig = go.Figure(data=[go.Scatter(x=f, y=mag_db, mode='lines', name=f'Ch {ch}')])
    fig.update_layout(
        margin=dict(l=40,r=10,t=30,b=40),
        title=f"Spectrum (Hann)  peak  {peak_f:.1f} Hz ({peak_db:.1f} dB rel)"
    )
    fig.update_xaxes(title="Frequency (Hz)", range=[0, FS_PRINT_HZ/2])
    fig.update_yaxes(title="Magnitude (dB rel)", range=[-120, 0])
    return fig

if __name__ == "__main__":
    HOST = "127.0.0.1"
    PORT_WEB = 8050
    app.run(debug=False, host=HOST, port=PORT_WEB)
```

Part F — Optional Signal Shapes (Audio Files)

You may also render these two signals and listen with `play`.

Bézier Envelope (AM tone)

Listing 5: Bézier envelope + sine (writes WAV)

```
import numpy as np
import soundfile as sf

def vel_bezier_env(
    dur=5.0, fs=48_000, KF=24.0,
    r=(252, 1050, 1800, 1575, 700, 126),
    shift=0.0, tajuste=0.5
):
    r1, r2, r3, r4, r5, r6 = r
    T1b = 0.1 + shift; T2b = 0.2 + shift
    T3b = 1.0 + shift + tajuste; T4b = 1.7 + shift + tajuste
    T5b = 2.7 + shift + tajuste; T6b = 2.8 + shift + tajuste
    base_end = T6b
    s = dur / base_end
    T1, T2, T3, T4, T5, T6 = [t*s for t in (T1b,T2b,T3b,T4b,T5b,T6b)]
    t = np.arange(int(fs*dur)) / fs

    def Bezier(K1):
        K1 = np.clip(K1, 0.0, 1.0)
        return (K1**5) * (r1 - (r2*K1) + (r3*K1**2) - (r4*K1**3) + (r5*K1**4) - (r6*K1**5))

```

```python
    y = np.full_like(t, KF*(1-0.25), dtype=np.float32)
    m = (t <= T1); y[m] = 0.0
    m = (t > T1) & (t <= T2); K1 = (t[m]-T1)/(T2-T1); y[m] = KF*Bezier(K1)
    m = (t > T2) & (t <= T3); y[m] = KF
    m = (t > T3) & (t <= T4); K1 = (t[m]-T3)/(T4-T3); y[m] = KF - KF*0.5*Bezier(K1)
    m = (t > T4) & (t <= T5); y[m] = KF*0.5
    m = (t > T5) & (t <= T6); K1 = (t[m]-T5)/(T6-T5); y[m] = (KF*0.5) + KF*0.25*Bezier
        (K1)
    return y.astype(np.float32), fs

def render_bezier_tone(filename, dur=5.0, fs=48_000, f0=440.0, amp=0.9, **env_kwargs):
    env, fs = vel_bezier_env(dur=dur, fs=fs, **env_kwargs)
    env = env - env.min(); peak = env.max()
    if peak > 0: env = env/peak
    t = np.arange(int(fs*dur)) / fs
    car = np.sin(2*np.pi*f0*t).astype(np.float32)
    y = (amp*env*car).astype(np.float32)
    sf.write(filename, y, fs, subtype="PCM_16")
    return y, fs

# Examples
render_bezier_tone("bezier_5s.wav", dur=5.0, f0=440.0)
render_bezier_tone("bezier_1s.wav", dur=1.0, f0=660.0)
```

Overdamped Impulse

Listing 6: Overdamped impulse (writes WAV)

```python
import numpy as np
import soundfile as sf

def overdamped_impulse(fs=48_000, dur=3.0, wn=2*np.pi*4, zeta=1.6, peak=0.95):
    assert zeta > 1.0
    t = np.arange(int(fs*dur)) / fs
    d = np.sqrt(zeta**2 - 1.0)
    s1 = -wn*(zeta - d)
    s2 = -wn*(zeta + d)
    h = (wn/(2*d)) * (np.exp(s1*t) - np.exp(s2*t))
    y = h - h.mean()
    y /= (np.max(np.abs(y)) + 1e-12)
    y = (peak * y).astype(np.float32)
    return y, fs

y, fs = overdamped_impulse()
sf.write("overdamped_impulse.wav", y, fs, subtype="PCM_16")
```

Firmware Listing (ESP32 ADC Continuous)

Use this as your starting point. Modify `EXAMPLE_READ_LEN`, `SAMPLE_FREQ_HZ`, and `PRINT_RATE_HZ`. Remember to adjust your UART baud in both firmware (via `menuconfig`) and Python plotter.

Listing 7: ESP32 Firmware (ADC continuous, two channels)

```c
/*
 * SPDX-FileCopyrightText: 2021-2022 Espressif Systems (Shanghai) CO LTD
 *
```

```c
 * SPDX-License-Identifier: Apache-2.0
 */

#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

#define SAMPLE_FREQ_HZ (48 * 1000) // your .sample_freq_hz
#define PATTERN_NUM 2 // you scan 2 channels
#define FS_PER_CH (SAMPLE_FREQ_HZ / PATTERN_NUM)
#define PRINT_RATE_HZ 200 // ~200 lines/s is smooth
#define PRINT_DECIM (FS_PER_CH / PRINT_RATE_HZ)

static uint32_t last6=0, last7=0;
static bool have6=false, have7=false;
static uint32_t print_cnt=0;

#if CONFIG_IDF_TARGET_ESP32
static adc_channel_t channel[2] = {ADC_CHANNEL_6, ADC_CHANNEL_7};
#else
static adc_channel_t channel[2] = {ADC_CHANNEL_2, 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);
}

```

```c
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 = 48 * 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) {
```

```c
            ret = adc_continuous_read(handle, result, EXAMPLE_READ_LEN, &ret_num, 0);
            if (ret == ESP_OK) {
                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)) {
                        if (chan_num == 6) { last6 = data; have6 = true; }
                        else if (chan_num == 7) { last7 = data; have7 = true; }

                        if (have6 && have7) {
                            if ((print_cnt++ % PRINT_DECIM) == 0) {
                                printf("%" PRIu32 " %" PRIu32 "\n", last6, last7);
                            }
                            have6 = have7 = false;
                        }
                    } else {
                    }
                }
                vTaskDelay(1);
            } else if (ret == ESP_ERR_TIMEOUT) {
                break;
            }
        }
    }

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

Deliverables

1. `tone.wav` (and any other WAVs you generated).
2. A photo or schematic of your RC bias network showing component values.
3. Serial capture or screen recording showing live plots and correct frequency.
4. Short write-up: values you tried for `EXAMPLE_READ_LEN`, `SAMPLE_FREQ_HZ`, `PRINT_RATE_HZ`; the console baud you selected; and how you changed it in both firmware and Python.

Tips

- The print rate is approximately `FS_PER_CH`/`PRINT_DECIM`. Ensure `PRINT_DECIM` is an integer ≥ 1.
- If you increase the UART baud, update both `idf.py monitor` and the plotter's `BAUD`.
- Keep the input within the ADC range; start with small amplitudes and verify with a DMM or scope.