VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

ELEKTRONIKOS FAKULTETAS

ELEKTRONINIŲ SISTEMŲ KATEDRA

Vaizdo ir garso perdavimas per Wi-Fi naudojant XIAO ESP32S3 SENSE

Realaus laiko operacinės sistemos

Kursinis darbas

Atliko: EKSfm-24 gr. Ignas Malinauskas

Tikrino: dr. Saulius Sakavičius

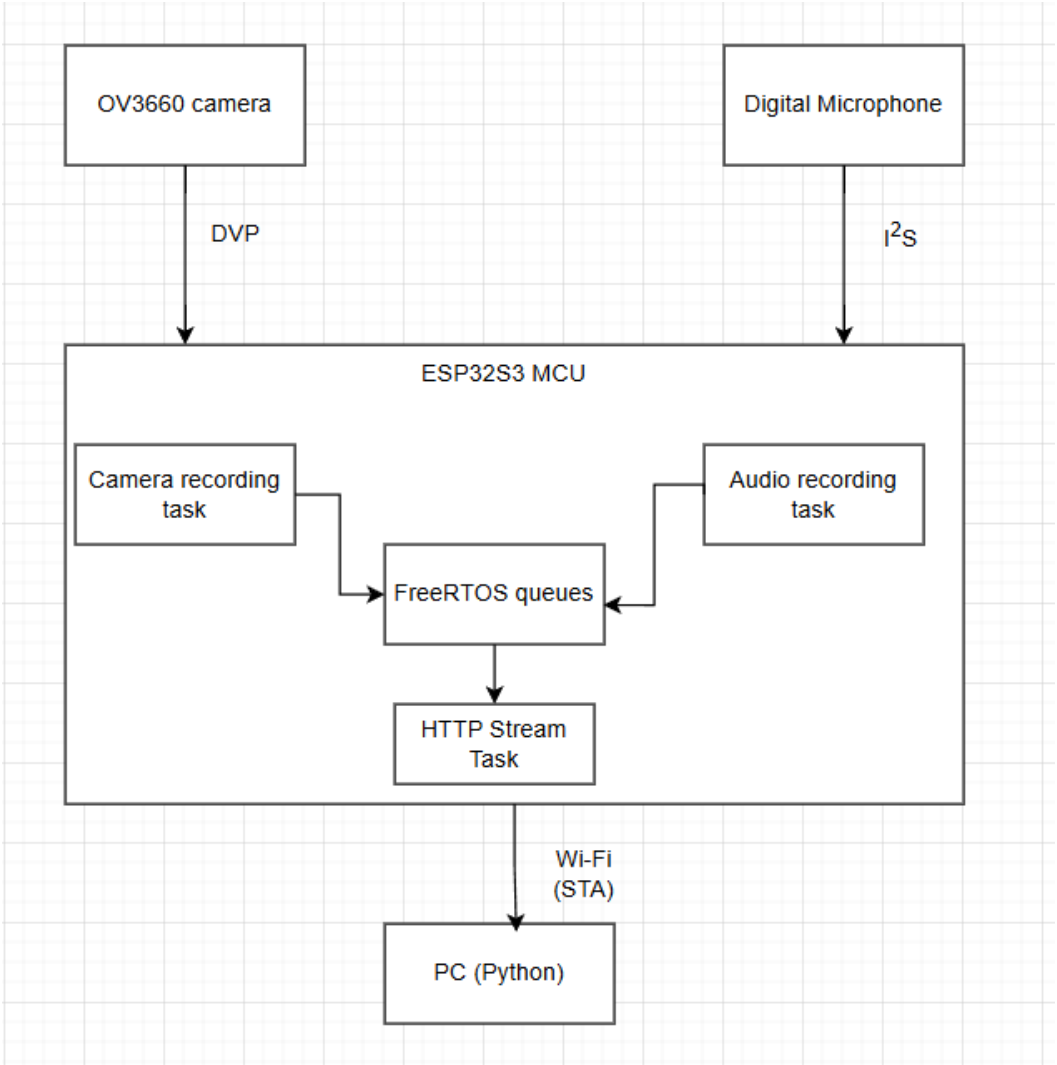
VILNIUS 2025

Turinys

ĮVADAS

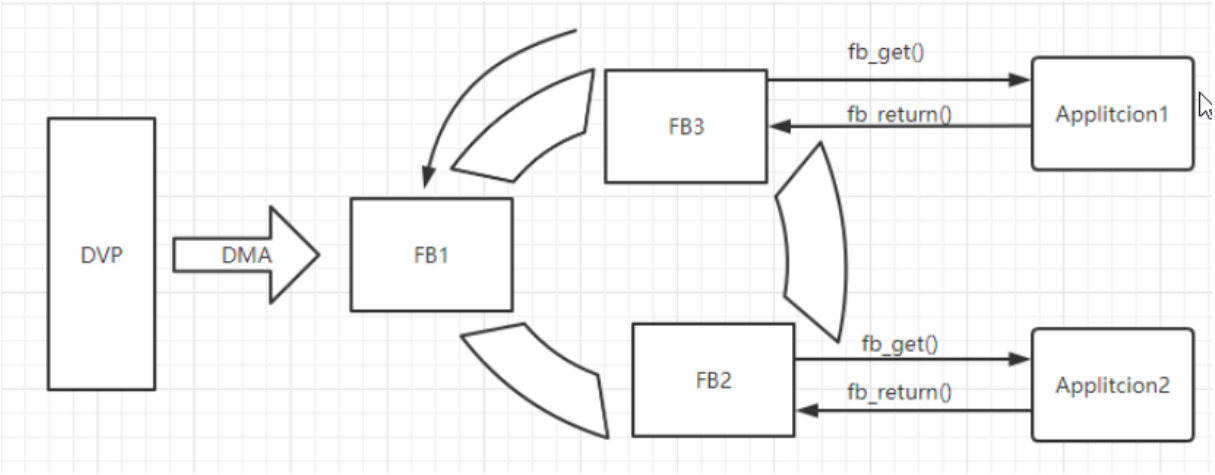
Šiame kursiniame darbe, naudojant ESP32S3 mikrovaldiklį ,buvo kuriama sistema, kuri realiu laiku užfiksuoja vaizdo ir garso rodmenis iš integruotų jutiklių, perduoda duomenis per Wi-Fi HTTP protokolu ir duomenis atvaizduojami Python programa. Programiniu kodui rašyti buvo naudojamas ESP-IDF karkasas per VS Code programavimo aplinką. Projekte naudojama kamera vaizdo įrašymui - OV3660, duomenų perdavimui – DVP (*digital video port*), garso įrašymui – skaitmeninis MEMS PDM mikrofonas, perdavimui – I^2S protokolas. Vaizdo ir garso įrašymui, duomenų perdavimui buvo naudojama FreeRTOS realaus laiko operacinė sistema.

Architektūra



Vaizdo užfiksavimo užduotis

Architektūra



1 pav. ESP32S3 kameros bendras veikimo principas (Espressif, 20225)

Kameros jutiklis perduoda duomenis į ESP32 įrenginį per DVP (lygiagretus skaitmeninis vaizdo prievadas) sąsają. Inicijuojant kamerą, dalis *frame_buffer* bus skirta kameros jutiklio perduodamiems duomenims saugoti. Kai kamera inicijuojama, ji iš karto pradeda veikti, o programa gali iškviesti *esp_camera_fb_get(),* kad gautų vaizdo duomenis. Kai programa pritaiko vaizdo duomenis, *frame_buffer* galima pakartotinai panaudoti iškviečiant *esp_camera_fb_return()* (Espressif, 2025).

Įgyvendinimas

Toliau aprašomas kameros vaizdo fiksavimo užduoties įgyvendinimas – programinis kodas bei parametrai. Pateikiamos ištraukos iš sistemos programinio kodo – kameros parametrų nustatymas ir inicijavimas bei vaizdo užfiksavimas.

Kameros parametrų nustatymas

```
static esp_err_t init_camera(uint32_t xclk_freq_hz, pixformat_t pixel_format,
framesize_t frame_size, uint8_t fb_count)
{
    camera_config_t camera_config = {
        .pin_pwdn = -1,
        .pin_reset = -1,
        .pin_xclk = 10,
        .pin_sscb_sda = 40,
        .pin_sscb_scl = 39,

        .pin_d7 = 48,
```

```
        .pin_d6 = 11,
        .pin_d5 = 12,
        .pin_d4 = 14,
        .pin_d3 = 16,
        .pin_d2 = 18,
        .pin_d1 = 17,
        .pin_d0 = 15,
        .pin_vsync = 38,
        .pin_href = 47,
        .pin_pclk = 13,

        .xclk_freq_hz = xclk_freq_hz,
        .ledc_timer = LEDC_TIMER_0,
        .ledc_channel = LEDC_CHANNEL_0,

        .pixel_format = pixel_format, //JPEG
        .frame_size = frame_size,    //QQVGA-UXGA, sizes above QVGA are not
been recommended when not JPEG format.

        .jpeg_quality = 10, //0-63
        .fb_count = fb_count,       //if more than one, i2s runs in continuous
mode. Use only with JPEG.
        .grab_mode = CAMERA_GRAB_LATEST,
        .fb_location = CAMERA_FB_IN_PSRAM
    };
```

Pirmi du išvadai nurodo, kad išjungimo ir perkrovimo išvadai nėra prijungti.

Išorinio laikrodžio dažnis (10MHz) bei sekantys 3 parametrai paduodami į iniciacijos funkciją kaip argumentai pagrindinio kodo dalyje.

MCLK yra pagrindinis kameros sistemos laikrodis, valdantis sistemos sinchronizavimą ir dažnį. Kameros luste MCLK naudojamas įvairių modulių, tokių kaip išankstiniai procesoriai, skaitmeniniai signalų procesoriai, pikselių matricos ir duomenų išvesties sąsajos, laikui valdyti. Paprastai MCLK dažnį kartu nustato pagrindinio valdymo lusto sistemos laikrodis ir kameros viduje esantis daliklis. Įprasti dažniai yra 6 MHz, 12 MHz, 24 MHz, 48 MHz ir kt. (Espressif, 2025). Šiuo atveju 10 MHz pasirinkta pagal gamintojo rekomendacijas.

ledc_timer / ledc_channel – kurį LEDC periferinį laikmatį/kanalą naudoti laikrodžio XCLK generavimui. LEDC yra lankstus PWM/laikmačio periferinis įrenginys; čia jis perdirbamas į stačiakampės bangos laikrodžio šaltinį.

Pikselių formatas – JPEG, dėl mažesnio kadro dydžio, kad vaizdas būtų persiunčiamas greitai ir be trukdžių. JPEG konvertavimas įvyksta pačiame kameros sensoriuje.

Rėmo dydis – QVGA 320 x 240 dėl anksčiau išvardintų priežasčių.

Kadrų buferių kiekis – 2 dėl nepertraukiamos vaizdo transliacijos – vienas buferis renka vaizdo duomenis, kitas siunčia atvaizdavimui.

Nurodyti sscb_sda ir sscb_cl išvadai atitinkamai reiškia SCCB (Serial Camera Control Bus) duomenų ir laikrodžio linijas, kas iš esmės yra tas pats kaip I^2C.

pin_vsync – vertikalus sinchronizavimas, pulsuoja kadro pradžioje.

pin_href – horizontalus atskaitos taškas: pulsuoja, kai perduodama linija.

pin_pclk – pikselių laikrodis: fiksuoja kiekvieną pikselio bitą.

pin_d0 - pin_d7 – aštuonių bitų lygiagrečių pikselių duomenys gaunami iš jutiklio.

grab_mode CAMERA_GRAB_LATEST – jei vaizdo apdorojimas vėluoja, visada grąžina naujausią galimą kadrą, praleidžiant senesnius.

fb_location CAMERA_FB_IN_PSRAM – įdeda kadrų buferius į išorinę pseudostatinę RAM. Atlaisvina vidinį SRAM kodui/steko atminčiai.

Kameros iniciacija ir sensoriaus pertvarkymas

```
esp_err_t ret = esp_camera_init(&camera_config);
```

Šis iškvietimas nustato I2S periferinį įrenginį, sukonfigūruoja kontaktus, inicijuoja SCCB (I^2C) sąsają ir susisiekia su jutikliu.

```
sensor_t *s = esp_camera_sensor_get();
s->set_vflip(s, 1);//flip it back
//initial sensors are flipped vertically and colors are a bit saturated
if (s->id.PID == OV3660_PID) {
    s->set_saturation(s, -2);//lower the saturation
}

if (s->id.PID == OV3660_PID || s->id.PID == OV2640_PID) {
    s->set_vflip(s, 1); //flip it back
} else if (s->id.PID == GC0308_PID) {
    s->set_hmirror(s, 0);
} else if (s->id.PID == GC032A_PID) {
    s->set_vflip(s, 1);
}
```

Tikslaus kameros modelio nustatyti nepavyko kadangi ESP32S3 makete buvo naudojami keli modeliai, šioje kodo ištraukoje pagal viduje esančią kamerą pakoreguojama kameros pozicija ir spalvos. Pagal numatytuosius nustatymus daugelis OV serijos modulių pradeda veikti „aukštyn kojomis “, todėl šis pirmasis iškvietimas tiesiog įjungia vertikalaus apvertimo bitą jutiklio registrų žemėlapyje.

```
camera_sensor_info_t *s_info = esp_camera_sensor_get_info(&(s->id));
```

```
    if (ESP_OK == ret && PIXFORMAT_JPEG == pixel_format && s_info-
>support_jpeg == true) {
        auto_jpeg_support = true;
    }

    return ret;
}
```

Šis paskutinis fragmentas vykdymo metu tiesiog patikrina, ar prijungtas kameros jutiklis iš tikrųjų palaiko JPEG fiksavimą, ir jei taip, įjungia automatinį JPEG režimą.

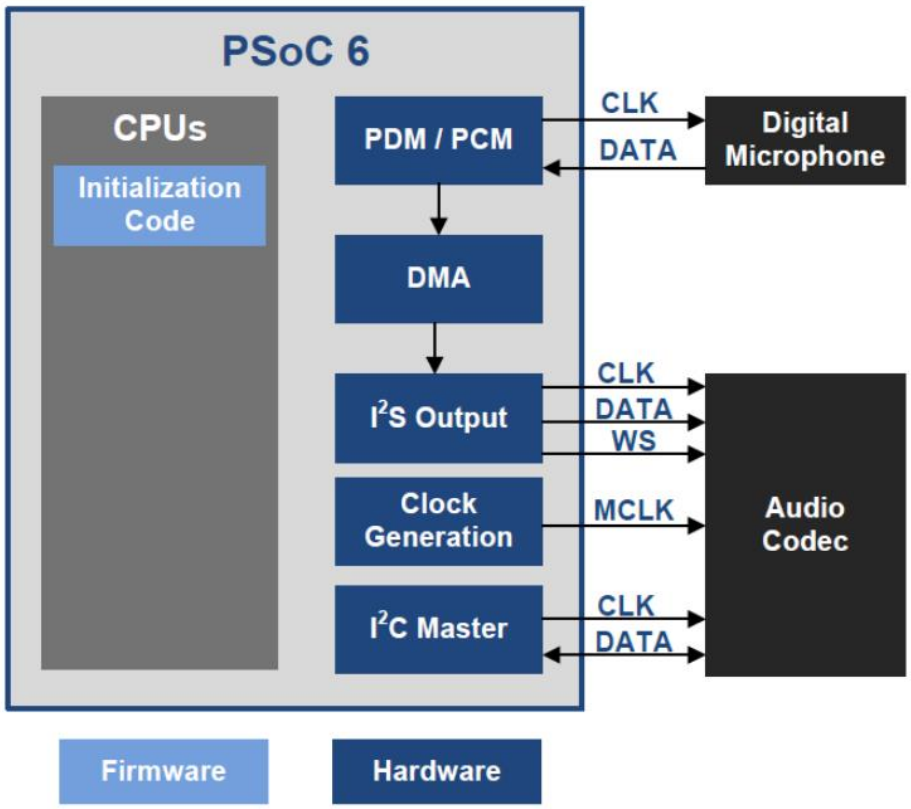
Vaizdo užfiksavimas

```
static void video_capture_task(void *arg) {
    camera_fb_t *frame;

    ESP_LOGI(TAG, "Starting video capture task");

        while (true) {
        frame = esp_camera_fb_get();
        if (frame) {
            xQueueSend(xQueueIFrame, &frame, portMAX_DELAY);
        }
    }
}
```

Šioje FreeRTOS užduoties ištraukoje užfiksuojamas kameros jutiklio duomenys ir siunčiami per FreeRTOS Queue vaizdo siuntimo per Wi-Fi užduočiai.

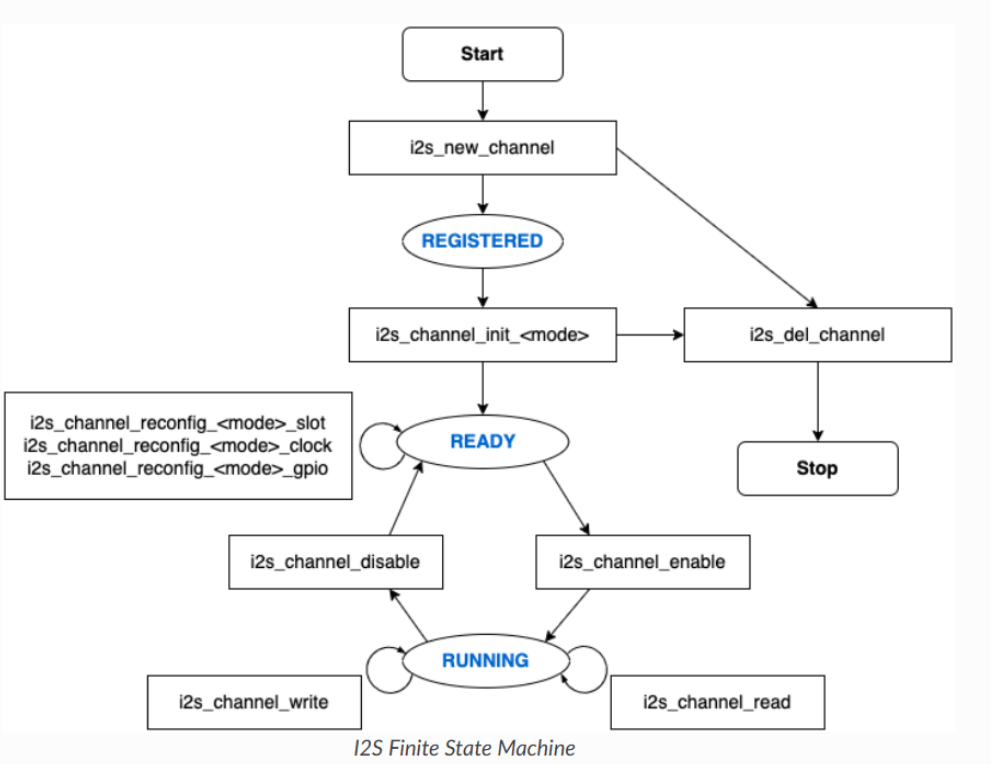
Garso įrašymo užduotis

Architektūra



2 pav. Įprasta PDM mikrofono per I²S schema

PDM (impulsų tankio moduliacijos) mikrofonas yra skaitmeninio mikrofono tipas, kuris išveda vieno bito duomenų srautą, kur vienetų tankis atitinka garso signalo amplitude, vietoj tiesioginės analoginės įtampos, kuri naudojama tradiciniuose mikrofonuose. PDM mikrofonai naudoja aukšto dažnio laikrodį ir sigma-delta moduliatorių, kad užkoduotų garso signalą į impulsų seriją. Šį skaitmeninį signalą, konvertavus į standartinį PCM (impulsų kodo moduliacijos) formatą, gali apdoroti kiti komponentai, pvz., kodekai arba DSP. Toliau DMA (*Direct Memory Access)* perkelia PCM paketus į atmintį neapkraunant pagrindinio procesoriaus. I²S magistralė, kuri bendrauja PDM režimu, susideda iš CLK: PDM laikrodžio ir DIN/DOUT: duomenų įvesties/išvesties linijų. I²S nuskaito PCM duomenis iš atminties ir perduoda standartinius I²S signalus (bitų takto CLK, žodžių pasirinkimo WS ir serijinius DATA) į išorinį garso kodeką ar kitą pasirinktą išvestį.



3 pav. ESP-IDF I²S gyvavimo ciklas (Espressif, 2025)

I2S periferinio įrenginio duomenų perdavimas, įskaitant siuntimą ir gavimą, vykdomas naudojant DMA. Prieš perduodant duomenis, iškviečiamas i2s_channel_enable(), kad įjungtų konkretų kanalą. Kai siunčiami arba gaunami duomenys pasiekia vieno DMA buferio dydį, bus suaktyvintas I2S_OUT_EOF arba I2S_IN_SUC_EOF pertraukimas. Vienas kadras čia reiškia visus atrinktus duomenis viename WS apskritime. Todėl dma_buffer_size = dma_frame_num * slot_num * slot_bit_width / 8. Duomenų perdavimui vartotojai gali įvesti duomenis iškviesdami i2s_channel_write(). Ši funkcija padeda vartotojams nukopijuoti duomenis iš šaltinio buferio į DMA TX buferį ir laukti, kol bus baigtas perdavimas. Tada tai kartosis tol, kol išsiųsti baitai pasieks nurodytą dydį. Duomenų priėmimui funkcija i2s_channel_read() laukia, kol bus gauta pranešimų eilė, kurioje yra DMA buferio adresas. Ji padeda vartotojams nukopijuoti duomenis iš DMA RX buferio į paskirties buferį. Tiek i2s_channel_write(), tiek i2s_channel_read() yra blokuojančios funkcijos. Jos laukia, kol bus išsiųstas visas šaltinio buferis arba įkeltas visas paskirties buferis, nebent jos viršija maksimalų blokavimo laiką, kai grąžinamas klaidos kodas ESP_ERR_TIMEOUT. Norint siųsti arba gauti duomenis asinchroniškai, grįžtamuosius iškvietimus galima registruoti naudojant i2s_channel_register_event_callback(). Vartotojai gali pasiekti DMA buferį tiesiogiai grįžtamojo iškvietimo funkcijoje, užuot siuntę arba gaudami duomenis dviem blokavimo funkcijomis (Espressif, 2025).

Įgyvendinimas

Pradiniai nustatymai ir iniciacija

```
#define PDM_CLK_IO          42
#define PDM_DATA_IO         41
#define PDM_SAMPLE_RATE     16000
#define PDM_BITS_PER_SAMPLE 16
#define PDM_CHANNELS_NUM    1
#define AUDIO_BUF_SAMPLES   (PDM_SAMPLE_RATE/10)
#define AUDIO_BUF_BYTES     (AUDIO_BUF_SAMPLES * sizeof(int16_t))
```

PDM_SAMPLE_RATE = 16 kHz diskretizavimo dažnis, išlaiko vidutinę duomenų perdavimo spartą ir procesoriaus/DMA apkrovą.

PDM_BITS_PER_SAMPLE = kiekvienas PCM paketos yra 16 bitų raiškos – tai įprastas pasirinkimas, suteikiantis ~96 dB dinaminį diapazoną neištuštindamas RAM.

PDM_CHANNELS_NUM = 1 (mono) dauguma mažų skaitmeninių mikrofonų yra vieno kanalo; mono taip pat perpus sumažina duomenų kiekį, palyginti su stereo.

AUDIO_BUF_SAMPLES = 1 600 padalijus iš 10, buferis sutalpina 100 ms garso (1,6 tūkst. paketų).100 ms pakankamai mažas, kad Wi-Fi srauto delsos laikas būtų mažesnis nei 200 ms, tačiau pakankamai didelis, kad amortizuotų RTOS eilės iškvietimus ir tinklo apkrovą.

AUDIO_BUF_BYTES = 1 600 pavyzdžių × 2 baitai = 3 200 baitų. Tiek atminties reikia kiekvienam 100 ms 16 bitų PCM fragmentui saugoti.

```
void init_microphone(void)
{
    i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO,
I2S_ROLE_MASTER);
    ESP_ERROR_CHECK(i2s_new_channel(&chan_cfg, NULL, &rx_handle));
```

Sukuria I²S kanalą pagrindiniu (*master)* režimu (jis generuoja mikrofono bitų laikrodį).

```
    i2s_pdm_rx_config_t pdm_rx_cfg = {
        .clk_cfg = I2S_PDM_RX_CLK_DEFAULT_CONFIG(PDM_SAMPLE_RATE),
        /* The default mono slot is the left slot (whose 'select pin' of the
PDM microphone is pulled down) */
        .slot_cfg = I2S_PDM_RX_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT,
I2S_SLOT_MODE_MONO),
        .gpio_cfg = {
            .clk = PDM_CLK_IO,
            .din = PDM_DATA_IO,
            .invert_flags = {
                .clk_inv = false,
            },
        },
    };
    ESP_ERROR_CHECK(i2s_channel_init_pdm_rx_mode(rx_handle, &pdm_rx_cfg));
    ESP_ERROR_CHECK(i2s_channel_enable(rx_handle));}
```

CLK konfigūracija: valdiklis pasirenka perteklinio diskretizavimo santykį (pvz., 64×) ir nustato PDM laikrodį taip, kad po decimacijos gautumėte 16 kHz PCM.

Lizdo konfigūracija: nurodo PDM blokui išvesti 16 bitų žodžius viename (mono) lizde.

GPIO konfigūracija: priskiria jūsų pasirinktus kontaktus PDM laikrodžio laikrodžiui ir duomenų linijai.

i2s_channel_init_pdm_rx_mode() programuoja aparatinę įrangą, o i2s_channel_enable() ją įjungia.

PDM režimas RX kanalui gali priimti PDM formato duomenis ir konvertuoti juos į PCM formatą. PDM RX palaikomas tik I2S0 ir tik 16 bitų pločio pavyzdžių duomenis. PDM RX reikia CLK kaiščio laikrodžio signalui ir DIN kaiščio duomenų signalui.

```
void audio_capture_task(void *arg) {
    audio_frame_t *audio_frame = (audio_frame_t
*)malloc(sizeof(audio_frame_t));
    if (!audio_frame) {
        ESP_LOGE(TAG, "Failed to allocate memory for audio frame");
        vTaskDelete(NULL);
    }

    audio_frame->data = (uint8_t *)malloc(AUDIO_BUF_BYTES);
    if (!audio_frame->data) {
        ESP_LOGE(TAG, "Failed to allocate memory for audio data");
        free(audio_frame);
        vTaskDelete(NULL);
    }
    audio_frame->len = AUDIO_BUF_BYTES;

    while (true) {
        size_t bytes_read;
        esp_err_t ret = i2s_channel_read(rx_handle, audio_frame->data,
AUDIO_BUF_BYTES, &bytes_read, portMAX_DELAY);
        if (ret == ESP_OK && bytes_read > 0) {
            xQueueSend(xQueueAudio, &audio_frame, portMAX_DELAY);
        }
    }
}
```

Šiame kodo fragmente iš pradžių paskiriama atminties dalis pagal audio_frame_t struktūrą:

```
typedef struct {
    uint8_t *data;
    size_t   len;
} audio_frame_t;
```

Toliau priskiriamas buferio dydis ir pradedamas įrašymas: i2s_channel_read nustato I^2S RX priėmimo kanalo valdiklį, nurodo rodyklę į buferį, kiek baitų duomenų nuskaityti, kur juos

išsaugoti ir užblokuoja PDM ir DMA kol buferis neprisipildė. Duomenis išsiunčiami tolesniam apdorojimui ir siuntimui per Wi-FI su FreeRTOS Queue.

ESP32-S3 Wi-Fi valdiklis

Wi-Fi gyvavimo ciklas

Iniciacija - Wi-Fi ciklo pradžia prasideda ties funkcija esp_netif_init(), kuri sukuria TCP/IP giją, pagrindines duomenų struktūras (ARP lenteles, TCP/UDP blokus, laikmačius) ir priskiria registrus pagal naudojamą Wi-Fi rėžimą.

Konfigūracija – iškviečiama funkcija esp_wifi_set_mode() ir nurodamas Wi-Fi rėžimas (NUKK, STA, AP, APSTA).

Pradžia – funkcija esp_wifi_start() iškviečia Wi-Fi valdiklį ir įjungią būsenų mašiną, kuri ieško Wi-Fi stočių, atlieka autentifikaciją bei keičia įvykių būsenas. Kitaip tariant, esp_wifi_start() iš tikrųjų suaktyvina seką:

valdiklio inicijavimas →

WIFI_EVENT_STA_START →

valdiklis iškviečia esp_wifi_connect() →

autentifikavimas/DHCP →

IP_EVENT_STA_GOT_IP arba, gedimo atveju, WIFI_EVENT_STA_DISCONNECTED.

Toliau vyksta prisijungimas naudojant funkciją esp_wifi_connect(). Šiame darbe ESP32S3 Wi-Fi valdiklis naudojamas STA režimu, t.y. maketas veikia kaip klientas, prisijungia prie Wi-Fi tinklo, kur ranka programiniame kode įrašytas SSID ir slaptažodis.

```
#define ESP_WIFI_SSID      "TP-Link_8403"
#define ESP_WIFI_PASS      "15586971"
static void wifi_event_handler(void* arg, esp_event_base_t event_base,
                               int32_t event_id, void* event_data
    /* Sta mode */
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id ==
WIFI_EVENT_STA_DISCONNECTED) {
        if (s_retry_num < ESP_MAXIMUM_RETRY) {
            esp_wifi_connect();
            s_retry_num++;
            ESP_LOGI(TAG, "retry to connect to the AP");
        } else {
            xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
        }
        ESP_LOGI(TAG, "connect to the AP fail");
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
```

```
        ESP_LOGI(TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));
        s_retry_num = 0;
        xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
    }

    return;
}
```

Ši architektūra yra pagrįsta įvykių logika, tad bendrai vienas wifi_event_handler apdoroja tris pagrindinius įvykius:

- STA_START – pradeda prisijungimą
- STA_DISCONNECT – bando dar kartą arba deklaruoja nesėkmę
- GOT_IP – signalizuoja sėkmingą prisijungimą ir gautą IP adresą

Pagrindinė sistemos Wi-Fi logika:

```
void app_wifi_main()
{
    // Initialize NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);
    // Determine mode
    wifi_mode_t mode = WIFI_MODE_NULL;
    if (strlen(ESP_WIFI_SSID)) {
        mode = WIFI_MODE_STA;
    }
    if (mode == WIFI_MODE_NULL) {
        ESP_LOGW(TAG, "STA has not been configured. WiFi will be off.");
        return;
    }
    // Initialize TCP/IP and event loop
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    s_wifi_event_group = xEventGroupCreate();
    // Initialize WiFi with default config
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    // Register handlers
    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        WIFI_EVENT, ESP_EVENT_ANY_ID, &wifi_event_handler, NULL, NULL));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        IP_EVENT, IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL, NULL));
    // Set WiFi mode
    ESP_ERROR_CHECK(esp_wifi_set_mode(mode));
```

```
    // Create the default STA netif and configure STA parameters
    esp_netif_create_default_wifi_sta();
    wifi_init_sta();  // your helper sets SSID/pass
    // Optional: disable power save
    ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_PS_NONE));
    // Start WiFi driver
    ESP_ERROR_CHECK(esp_wifi_start());
    ESP_LOGI(TAG, "WiFi started in STA mode. Connecting to SSID:%s",
ESP_WIFI_SSID);
    // Wait for connection or failure
    EventBits_t bits = xEventGroupWaitBits(
        s_wifi_event_group,
        WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
        pdFALSE,
        pdFALSE,
        portMAX_DELAY);
    if (bits & WIFI_CONNECTED_BIT) {
        esp_netif_ip_info_t ip;
        esp_netif_t* sta_if = esp_netif_get_handle_from_ifkey("WIFI_STA_DEF");
        esp_netif_get_ip_info(sta_if, &ip);
        ESP_LOGI(TAG, "Device IP: " IPSTR, IP2STR(&ip.ip));
    } else {
        ESP_LOGE(TAG, "Failed to connect to SSID:%s after %d attempts",
                 ESP_WIFI_SSID, ESP_MAXIMUM_RETRY);
    }
```

Duomenų perdavimas HTTP protokolu

HTTP serverio komponentas suteikia galimybę paleisti lengvą *web* serverį ESP32-S3 sistemoje. Toliau pateikiami išsamūs veiksmai, kaip naudoti HTTP serverio pateiktą API:

httpd_start(): Sukuria HTTP serverio egzempliorių, paskirsto jam atmintį / išteklius pagal nurodytą konfigūraciją ir išveda tvarkyklė serverio egzemplioriui. Serveris turi ir klausymo lizdą (TCP) HTTP srautui, ir valdymo lizdą (UDP) valdymo signalams, kurie parenkami ratu serverio užduočių cikle. Užduoties prioritetą ir steko dydį galima konfigūruoti serverio egzemplioriaus kūrimo metu, perduodant httpd_config_t struktūrą httpd_start(). TCP srautas analizuojamas kaip HTTP užklausos ir, priklausomai nuo prašomo URI, iškviečiami vartotojo registruoti tvarkyklės, kurios turėtų siųsti atgal HTTP atsakymo paketus.

httpd_register_uri_handler(): URI tvarkyklė registruojama perduodant httpd_uri_t tipo struktūros objektą, turintį narius, įskaitant URI pavadinimą, metodo tipą (pvz., HTTPD_GET/HTTPD_POST/HTTPD_PUT ir kt.), funkcijos rodyklę, kurios tipas yra esp_err_t *handler (httpd_req_t *req), ir user_ctx rodyklę į vartotojo konteksto duomenis.

Serverio sukūrimas

Šiame darbe HTTP valdiklis iš kameros kadrų sukuria vientisą vaizdo medžiagą ir pasiunčia ją per HTTP. Kad naršyklė arba vaizdo grotuvas suprastų kaip traktuoti atkeliaujančią medžiagą naudojamos HTTP antraštės, kurios iš esmės sukuria MJPEG formatą – srautą JPEG nuotraukų.

```
static const char *_STREAM_CONTENT_TYPE = "multipart/x-mixed-
replace;boundary=" PART_BOUNDARY
static const char *_STREAM_BOUNDARY = "\r\n--" PART_BOUNDARY "\r\n";
static const char *_STREAM_PART = "Content-Type: image/jpeg\r\nContent-Length:
%u\r\nX-Timestamp: %d.%06d\r\n\r\n";
```

Antraštės nurodo tikėtis visos serijos dalių, kuri nauja keičia praėjusią bei aprašomos JPEG nuotraukų ribos, dydis.

```
    res = httpd_resp_set_type(req, _STREAM_CONTENT_TYPE);
    if (res != ESP_OK) {
        return res;
    }
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    httpd_resp_set_hdr(req, "X-Framerate", "60");
```

Content-Type - įgalina kelių dalių srautinį perdavimą.

CORS antraštė - leidžia bet kokiam įrenginiui gauti srautą (naudinga tinklalapiams skirtinguose pagrindiniuose kompiuteriuose).

X-Framerate - pasirinktinė antraštė, kurioje aprašomas tikslinis kadrų dažnis.

Kai šie duomenys išsiunčiami, ryšys lieka atviras neribotą laiką – nėra jokio galutinio </body> ar atsakymo uždarymo.

```
esp_err_t start_stream_server(const QueueHandle_t frame_i,
const QueueHandle_t audio_q,
const bool return_fb)     httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    config.stack_size = 5120;
    httpd_uri_t stream_uri = {
        .uri = "/stream",
        .method = HTTP_GET,
        .handler = stream_handler,
        .user_ctx = NULL
    };
```

Prieš pradedant siųsti duomenis – sukuriamas HTTP serveri siųsti vaizdui (naudojamas 80 prievadas:

```
        httpd_uri_t audio_uri  = {
        .uri      = "/audio",
        .method   = HTTP_GET,
        .handler  = audio_stream_handler,
        .user_ctx = NULL
    };
```

```
    httpd_config_t acfg = config;
    acfg.server_port    = 81;
    acfg.ctrl_port      = 32769;

    esp_err_t err = httpd_start(&video_httpd, &config);
    if (err == ESP_OK) {
        err = httpd_register_uri_handler(video_httpd, &stream_uri);
   }
    err = httpd_start(&audio_httpd, &acfg);
    if (err == ESP_OK) {
        err = httpd_register_uri_handler(audio_httpd, &audio_uri);
    }
        ESP_LOGI(TAG,
                 "Video : http://<IP>:80/stream   |   Audio :
http://<IP>:81/audio");
   }
```

Garsui siųsti pasitelkiamas 81 HTTP prievadas ir įjungiami serveriai.

Duomenų siuntimas

```
        if (xQueueReceive(xQueueFrameI, &frame, portMAX_DELAY)) {
            _timestamp.tv_sec = frame->timestamp.tv_sec;
            _timestamp.tv_usec = frame->timestamp.tv_usec;

            if (frame->format == PIXFORMAT_JPEG) {
                _jpg_buf = frame->buf;
                _jpg_buf_len = frame->len;
            } else if (!frame2jpg(frame, 60, &_jpg_buf, &_jpg_buf_len)) {
                ESP_LOGE(TAG, "JPEG compression failed");
                res = ESP_FAIL; }}
```

Duomenų siuntimas prasideda sulaukus naujo vaizdo kadro iš FreeRTOS Queue, uždedami laiko antspaudai ir paruošiamas JPEG buferis.

```
if (res == ESP_OK) {
            res = httpd_resp_send_chunk(req, _STREAM_BOUNDARY,
strlen(_STREAM_BOUNDARY));
```

Jeigu iki tol nebuvo klaidų, pradedamas duomenų perdavimas ir siunčiamos ribų linijos.

```
            if (res == ESP_OK) {
                size_t hlen = snprintf((char *)part_buf, 128, _STREAM_PART,
_jpg_buf_len, _timestamp.tv_sec, _timestamp.tv_usec);
              res = httpd_resp_send_chunk(req, (const char *)part_buf, hlen);}
```

Nusiunčiamos antraštės ir laiko antspaudai.

```
            if (res == ESP_OK) {
                res = httpd_resp_send_chunk(req, (const char *)_jpg_buf,
_jpg_buf_len);
            }
```

Ir galų gale nusiunčiami JPEG vaizdo kadro baitai.

```
        if (gReturnFB) {
            esp_camera_fb_return(frame);
```

Paskutinis siuntimo žingsnis – atlaisvinti vaizdo buferį.

```
    res = httpd_resp_set_type(req, "application/octet-stream");
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    httpd_resp_set_hdr(req, "X-Sample-Rate", "16000");
    httpd_resp_set_hdr(req, "X-Bits-Per-Sample", "16");
```

Garso duomenų siuntimas veikia iš esmės taip pat kaip ir vaizdo, tik pagrindinis skirtumas – antraštės.

Octet–stream nurodo klientui, kad siunčiami duomenys, yra neapdoroti dvejetainiai („oktetiniai“) duomenys be aukštesnio lygio įrėminimo, tolesnės antraštės nurodo audio metadata – diskretizavimo dažnį ir raišką. Tai grotuvui ar naršyklei duomenis, kokiu dažniu groti arba analizuoti medžiagą bei kokio dydžio kintamasis yra vienas paketas. Diskretizavimo dažnis ir raiška kartu pilnai nusako PCM formatą.

Pagrindis programos kodas ir FreeRTOS

Sistema įgyvendinama išskaidant vaizdo ir audio medžiagos surinkimą į atskiras FreeRTOS užduotis ir informacija tarp duomenų surinkimo ir perdavimo siunčiama pasitelkiant FreeRTOS Queue.

```
static QueueHandle_t xQueueIFrame = NULL;
static QueueHandle_t xQueueAudio = NULL;
```

Sukuriamos duomenų perdavimo eilės.

```
esp_err_t start_stream_server(const QueueHandle_t frame_i,
                              const QueueHandle_t audio_q,
                              const bool return_fb);
```

Duomenų siuntimo serveriai perduodami vaizdo ir garso valdikliai bei kameros buferio išvalymo argumentas.

```
void audio_capture_task(void *arg) {....
            xQueueSend(xQueueAudio, &audio_frame, portMAX_DELAY);  }
static void video_capture_task(void *arg) {.....
            xQueueSend(xQueueIFrame, &frame, portMAX_DELAY);}
```

Sukuriamos vaizdo ir garso surinkimo užduotys ir persiunčiami buferiai serveriui.

Sistemos veikimo eiliškumas:

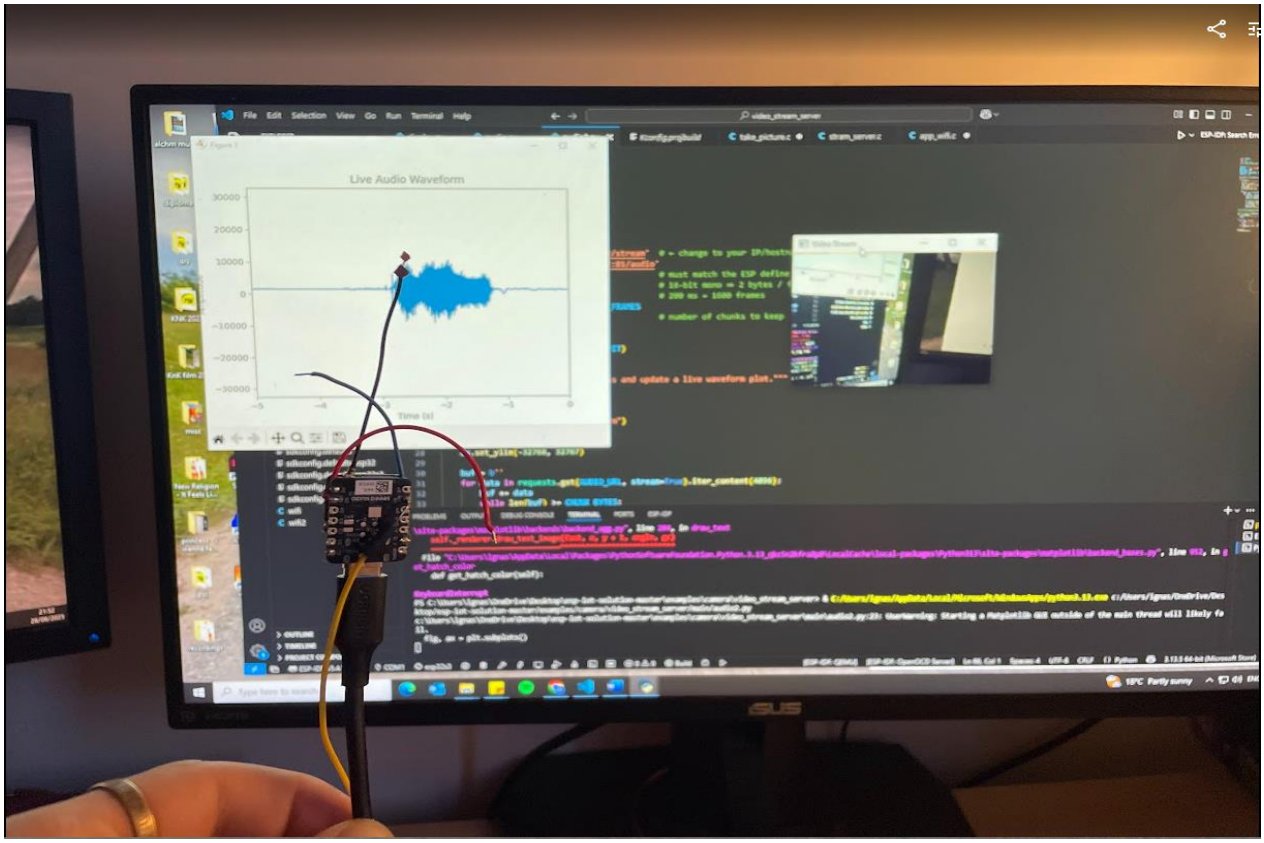
```
void app_main()
{
    app_wifi_main();
    init_microphone();

    xQueueIFrame = xQueueCreate(2, sizeof(camera_fb_t *));
    xQueueAudio = xQueueCreate(2, sizeof(audio_frame_t *));
    TEST_ESP_OK(init_camera(10000000, PIXFORMAT_JPEG, FRAMESIZE_QVGA, 2));

    xTaskCreate(video_capture_task, "video_capture_task", 4096, NULL, 5,
NULL);
```

```
    xTaskCreate(audio_capture_task, "audio_capture_task", 4096, NULL, 5,
NULL);
    TEST_ESP_OK(start_stream_server(xQueueIFrame, xQueueAudio, true));
}
```

Sistema pradedama parengiant Wi-Fi valdiklius ir prisijungiant prie stoties. Inicijuojami garso ir vaizdo jutikliai bei paduodami vaizdo surinkimo parametrai, sukuriamos eilės pagal siunčiamo buferį dydį ir siunčiamų dalykų kiekį, sukuriamos vaizdo ir garso surinkimo užduotys ir įjungiamas serveris.

Rezultatai naudojant Python programą

ESP32S3 jutiklių siunčiami duomenis per Wi-Fi HTTP protokolą stebimi pasirašius nesudėtingą Python skriptą, kuriame prašomos duomenų užklausos pagal atitinkamą (80 ir 81) HTTP prievadą ir su OpenCV paleidžiama vaizdo transliaciją ir su MatPlotLib biblioteka atvaizduojami mikrofono rodmenys.



Išvados

Šiame projekte buvo įgyvendinta funkcionuojanti realaus laiko garso ir vaizdo transliacijos sistema, naudojant ESP32-S3. Pasitelkiant OV3660 kamerą ir jame integruotą JPEG kodavimo įrenginį, I²S MEMS mikrofoną, sėkmingai įvykdyta vaizdo transliacija QVGA/15 kadrų per sekundę greičiu ir 16 kHz mono garsą su minimaliomis procesoriaus apkrovomis. Duomenų surinkimo ir perdavimo funkcionalumas įgyvendintas FreeRTOS užduotimis – nuo DVP ir I²S DMA per HTTP daugiapartinį (MJPEG) ir fragmentuotą garso transliaciją per Wi-Fi – iki Python programos, kuri atvaizdavo tiek vaizdo kadrus, tiek garso bangos formą. Ateityje projektą galima plėsti tobulinant HTTP serverio pajėgumą ir keisti garso atvaizdavimą į tikrą garsą, o ne bangą.

LITERATŪROS SĄRAŠAS

https://github.com/espressif/esp-iot-solution

https://docs.espressif.com/projects/esp-faq/en/latest/application-solution/camera-application.html#when-debugging-the-gc2145-camera-with-esp32-s3-the-maximum-supported-resolution-seems-to-be-1024x768-if-it-is-adjusted-to-a-larger-resolution-such-as-1280x720-it-will-print-cam-hal-ev-eof-ovf-error-how-to-solve-this-issue

https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/peripherals/i2s.html

https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-guides/wifi.html#esp32-s3-wi-fi-station-general-scenario

https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/protocols/esp_http_server.html

Python skriptas

```
import threading
import requests
import cv2
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

# Configuration
STREAM_URL   = 'http://192.168.0.112/stream'  # ← change to your IP/hostname
AUDIO_URL    = 'http://192.168.0.112:81/audio'
SAMPLE_RATE  = 16000                          # must match the ESP define
BYTES_PER_FR = 2                              # 16-bit mono ⇒ 2 bytes / frame
CHUNK_FRAMES = SAMPLE_RATE // 10              # 200 ms = 1600 frames
CHUNK_BYTES  = BYTES_PER_FR * CHUNK_FRAMES
FIFO_LIMIT   = 50                             # number of chunks to keep

# rolling buffer for audio
audio_buffer = deque(maxlen=FIFO_LIMIT)

def audio_plot():
    """Continuously read audio chunks and update a live waveform plot."""
    plt.ion()
    fig, ax = plt.subplots()
    line, = ax.plot([], [], lw=1)
    ax.set_title("Live Audio Waveform")
    ax.set_xlabel("Time (s)")
    ax.set_ylabel("Amplitude")
    ax.set_ylim(-32768, 32767)

    buf = b''
    for data in requests.get(AUDIO_URL, stream=True).iter_content(4096):
        buf += data
        while len(buf) >= CHUNK_BYTES:
            chunk, buf = buf[:CHUNK_BYTES], buf[CHUNK_BYTES:]
            samples = np.frombuffer(chunk, dtype=np.int16)
            audio_buffer.append(samples)

            # concatenate and plot
            all_samples = np.concatenate(audio_buffer)
            t = np.linspace(-len(all_samples)/SAMPLE_RATE, 0,
len(all_samples))
            line.set_data(t, all_samples)
            ax.set_xlim(t[0], t[-1])
            fig.canvas.draw()
            fig.canvas.flush_events()

# start audio plotting in a background thread
threading.Thread(target=audio_plot, daemon=True).start()
```

```
# open and display the MJPEG video stream
cap = cv2.VideoCapture(STREAM_URL)
if not cap.isOpened():
    print(f"Failed to open video stream at {STREAM_URL}")
    exit(1)

while True:
    ret, frame = cap.read()
    if not ret:
        print("Stream ended or cannot fetch frame.")
        break
    cv2.imshow('Video Stream', frame)
    if cv2.waitKey(1) & 0xFF == 27:  # press Esc to exit
        break

cap.release()
cv2.destroyAllWindows()
```