```
// Copyright 2015-2016 Espressif Systems (Shanghai) PTE LTD
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
#include "esp_http_server.h"
#include "esp_timer.h"
#include "esp_camera.h"
#include "img_converters.h"
#include "fb_gfx.h"
#include "esp32-hal-ledc.h"
#include "sdkconfig.h"
#include "camera_index.h"

#if defined(ARDUINO_ARCH_ESP32) && defined(CONFIG_ARDUHAL_ESP_LOG)
#include "esp32-hal-log.h"
#endif

// Face Detection will not work on boards without (or with disabled) PSRAM
#ifdef BOARD_HAS_PSRAM
#define CONFIG_ESP_FACE_DETECT_ENABLED 1
// Face Recognition takes upward from 15 seconds per frame on chips other than
ESP32S3
// Makes no sense to have it enabled for them
#if CONFIG_IDF_TARGET_ESP32S3
#define CONFIG_ESP_FACE_RECOGNITION_ENABLED 1
#else
#define CONFIG_ESP_FACE_RECOGNITION_ENABLED 0
#endif
#else
#define CONFIG_ESP_FACE_DETECT_ENABLED 0
#define CONFIG_ESP_FACE_RECOGNITION_ENABLED 0
#endif

#if CONFIG_ESP_FACE_DETECT_ENABLED

#include <vector>
```

```
#include "human_face_detect_msr01.hpp"
#include "human_face_detect_mnp01.hpp"

#define TWO_STAGE 1 /*<! 1: detect by two-stage which is more accurate but
slower(with keypoints). */
                    /*<! 0: detect by one-stage which is less accurate but
faster(without keypoints). */

#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
#include "face_recognition_tool.hpp"
#include "face_recognition_112_v1_s16.hpp"
#include "face_recognition_112_v1_s8.hpp"

#define QUANT_TYPE 0 //if set to 1 => very large firmware, very slow, reboots
when streaming...

#define FACE_ID_SAVE_NUMBER 7
#endif

#define FACE_COLOR_WHITE 0x00FFFFFF
#define FACE_COLOR_BLACK 0x00000000
#define FACE_COLOR_RED 0x000000FF
#define FACE_COLOR_GREEN 0x0000FF00
#define FACE_COLOR_BLUE 0x00FF0000
#define FACE_COLOR_YELLOW (FACE_COLOR_RED | FACE_COLOR_GREEN)
#define FACE_COLOR_CYAN (FACE_COLOR_BLUE | FACE_COLOR_GREEN)
#define FACE_COLOR_PURPLE (FACE_COLOR_BLUE | FACE_COLOR_RED)
#endif

// Enable LED FLASH setting
#define CONFIG_LED_ILLUMINATOR_ENABLED 1

// LED FLASH setup
#if CONFIG_LED_ILLUMINATOR_ENABLED

#define LED_LEDC_CHANNEL 2 //Using different ledc channel/timer than camera
#define CONFIG_LED_MAX_INTENSITY 255

int led_duty = 0;
bool isStreaming = false;

#endif

typedef struct
{
    httpd_req_t *req;
    size_t len;
} jpg_chunking_t;

#define PART_BOUNDARY "123456789000000000000987654321"
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";

httpd_handle_t stream_httpd = NULL;
httpd_handle_t camera_httpd = NULL;

#if CONFIG_ESP_FACE_DETECT_ENABLED

static int8_t detection_enabled = 0;

// #if TWO_STAGE
// static HumanFaceDetectMSR01 s1(0.1F, 0.5F, 10, 0.2F);
// static HumanFaceDetectMNP01 s2(0.5F, 0.3F, 5);
// #else
// static HumanFaceDetectMSR01 s1(0.3F, 0.5F, 10, 0.2F);
// #endif

#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
static int8_t recognition_enabled = 0;
static int8_t is_enrolling = 0;

#if QUANT_TYPE
    // S16 model
    FaceRecognition112V1S16 recognizer;
#else
    // S8 model
    FaceRecognition112V1S8 recognizer;
#endif
#endif

#endif

typedef struct
{
    size_t size;  //number of values used for filtering
    size_t index; //current value index
    size_t count; //value count
    int sum;
    int *values; //array to be filled with values
} ra_filter_t;

static ra_filter_t ra_filter;

static ra_filter_t *ra_filter_init(ra_filter_t *filter, size_t sample_size)
{
    memset(filter, 0, sizeof(ra_filter_t));

    filter->values = (int *)malloc(sample_size * sizeof(int));
    if (!filter->values)
    {
        return NULL;
    }
    memset(filter->values, 0, sample_size * sizeof(int));

    filter->size = sample_size;
    return filter;
}

#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
static int ra_filter_run(ra_filter_t *filter, int value)
{
    if (!filter->values)
    {
        return value;
    }
    filter->sum -= filter->values[filter->index];
    filter->values[filter->index] = value;
    filter->sum += filter->values[filter->index];
    filter->index++;
    filter->index = filter->index % filter->size;
    if (filter->count < filter->size)
    {
        filter->count++;
    }
    return filter->sum / filter->count;
}
#endif

#if CONFIG_ESP_FACE_DETECT_ENABLED
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
static void rgb_print(fb_data_t *fb, uint32_t color, const char *str)
{
    fb_gfx_print(fb, (fb->width - (strlen(str) * 14)) / 2, 10, color, str);
}

static int rgb_printf(fb_data_t *fb, uint32_t color, const char *format, ...)
{
    char loc_buf[64];
    char *temp = loc_buf;
    int len;
    va_list arg;
    va_list copy;
    va_start(arg, format);
    va_copy(copy, arg);
    len = vsnprintf(loc_buf, sizeof(loc_buf), format, arg);
    va_end(copy);
    if (len >= sizeof(loc_buf))
    {
        temp = (char *)malloc(len + 1);
        if (temp == NULL)
        {
            return 0;
        }
    }
    vsnprintf(temp, len + 1, format, arg);
    va_end(arg);
    rgb_print(fb, color, temp);
    if (len > 64)
    {
        free(temp);
    }
    return len;
}
#endif
static void draw_face_boxes(fb_data_t *fb, std::list<dl::detect::result_t>
*results, int face_id)
{
    int x, y, w, h;
    uint32_t color = FACE_COLOR_YELLOW;
    if (face_id < 0)
    {
        color = FACE_COLOR_RED;
    }
    else if (face_id > 0)
    {
        color = FACE_COLOR_GREEN;
    }
    if(fb->bytes_per_pixel == 2){
        //color = ((color >> 8) & 0xF800) | ((color >> 3) & 0x07E0) | (color &
0x001F);
        color = ((color >> 16) & 0x001F) | ((color >> 3) & 0x07E0) | ((color <<
8) & 0xF800);
    }
    int i = 0;
    for (std::list<dl::detect::result_t>::iterator prediction = results->begin();
prediction != results->end(); prediction++, i++)
    {
        // rectangle box
        x = (int)prediction->box[0];
        y = (int)prediction->box[1];
        w = (int)prediction->box[2] - x + 1;
        h = (int)prediction->box[3] - y + 1;
        if((x + w) > fb->width){
            w = fb->width - x;
        }
        if((y + h) > fb->height){
            h = fb->height - y;
        }
        fb_gfx_drawFastHLine(fb, x, y, w, color);
        fb_gfx_drawFastHLine(fb, x, y + h - 1, w, color);
        fb_gfx_drawFastVLine(fb, x, y, h, color);
        fb_gfx_drawFastVLine(fb, x + w - 1, y, h, color);
#if TWO_STAGE
        // landmarks (left eye, mouth left, nose, right eye, mouth right)
        int x0, y0, j;
        for (j = 0; j < 10; j+=2) {
            x0 = (int)prediction->keypoint[j];
            y0 = (int)prediction->keypoint[j+1];
            fb_gfx_fillRect(fb, x0, y0, 3, 3, color);
        }
#endif
    }
}

#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
static int run_face_recognition(fb_data_t *fb, std::list<dl::detect::result_t>
*results)
{
    std::vector<int> landmarks = results->front().keypoint;
    int id = -1;

    Tensor<uint8_t> tensor;
    tensor.set_element((uint8_t *)fb->data).set_shape({fb->height, fb->width, 
3}).set_auto_free(false);

    int enrolled_count = recognizer.get_enrolled_id_num();

    if (enrolled_count < FACE_ID_SAVE_NUMBER && is_enrolling){
        id = recognizer.enroll_id(tensor, landmarks, "", true);
        log_i("Enrolled ID: %d", id);
        rgb_printf(fb, FACE_COLOR_CYAN, "ID[%u]", id);
    }

    face_info_t recognize = recognizer.recognize(tensor, landmarks);
    if(recognize.id >= 0){
        rgb_printf(fb, FACE_COLOR_GREEN, "ID[%u]: %.2f", recognize.id, 
recognize.similarity);
    } else {
        rgb_print(fb, FACE_COLOR_RED, "Intruder Alert!");
    }
    return recognize.id;
}
#endif
#endif

#if CONFIG_LED_ILLUMINATOR_ENABLED
void enable_led(bool en)
{ // Turn LED On or Off
    int duty = en ? led_duty : 0;
    if (en && isStreaming && (led_duty > CONFIG_LED_MAX_INTENSITY))
    {
        duty = CONFIG_LED_MAX_INTENSITY;
    }
    ledcWrite(LED_LEDC_CHANNEL, duty);
    //ledc_set_duty(CONFIG_LED_LEDC_SPEED_MODE, CONFIG_LED_LEDC_CHANNEL, duty);
    //ledc_update_duty(CONFIG_LED_LEDC_SPEED_MODE, CONFIG_LED_LEDC_CHANNEL);
    log_i("Set LED intensity to %d", duty);
}
#endif

static esp_err_t bmp_handler(httpd_req_t *req)
{
    camera_fb_t *fb = NULL;
    esp_err_t res = ESP_OK;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
    uint64_t fr_start = esp_timer_get_time();
#endif
```

```
    fb = esp_camera_fb_get();
    if (!fb)
    {
        log_e("Camera capture failed");
        httpd_resp_send_500(req);
        return ESP_FAIL;
    }

    httpd_resp_set_type(req, "image/x-windows-bmp");
    httpd_resp_set_hdr(req, "Content-Disposition", "inline;
filename=capture.bmp");
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");

    char ts[32];
    snprintf(ts, 32, "%ld.%06ld", fb->timestamp.tv_sec, fb->timestamp.tv_usec);
    httpd_resp_set_hdr(req, "X-Timestamp", (const char *)ts);


    uint8_t * buf = NULL;
    size_t buf_len = 0;
    bool converted = frame2bmp(fb, &buf, &buf_len);
    esp_camera_fb_return(fb);
    if(!converted){
        log_e("BMP Conversion failed");
        httpd_resp_send_500(req);
        return ESP_FAIL;
    }
    res = httpd_resp_send(req, (const char *)buf, buf_len);
    free(buf);
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
    uint64_t fr_end = esp_timer_get_time();
#endif
    log_i("BMP: %llums, %uB", (uint64_t)((fr_end - fr_start) / 1000), buf_len);
    return res;
}

static size_t jpg_encode_stream(void *arg, size_t index, const void *data, size_t
len)
{
    jpg_chunking_t *j = (jpg_chunking_t *)arg;
    if (!index)
    {
        j->len = 0;
    }
    if (httpd_resp_send_chunk(j->req, (const char *)data, len) != ESP_OK)
    {
        return 0;
    }
    j->len += len;
    return len;
}

static esp_err_t capture_handler(httpd_req_t *req)
{
    camera_fb_t *fb = NULL;
    esp_err_t res = ESP_OK;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
    int64_t fr_start = esp_timer_get_time();
#endif

#if CONFIG_LED_ILLUMINATOR_ENABLED
    enable_led(true);
    vTaskDelay(150 / portTICK_PERIOD_MS); // The LED needs to be turned on ~150ms
before the call to esp_camera_fb_get()
    fb = esp_camera_fb_get();             // or it won't be visible in the frame.
A better way to do this is needed.
    enable_led(false);
#else
    fb = esp_camera_fb_get();
#endif

    if (!fb)
    {
        log_e("Camera capture failed");
        httpd_resp_send_500(req);
        return ESP_FAIL;
    }

    httpd_resp_set_type(req, "image/jpeg");
    httpd_resp_set_hdr(req, "Content-Disposition", "inline;
filename=capture.jpg");
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");

    char ts[32];
    snprintf(ts, 32, "%ld.%06ld", fb->timestamp.tv_sec, fb->timestamp.tv_usec);
    httpd_resp_set_hdr(req, "X-Timestamp", (const char *)ts);

#if CONFIG_ESP_FACE_DETECT_ENABLED
    size_t out_len, out_width, out_height;
    uint8_t *out_buf;
    bool s;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
    bool detected = false;
#endif
    int face_id = 0;
    if (!detection_enabled || fb->width > 400)
    {
#endif
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        size_t fb_len = 0;
#endif
        if (fb->format == PIXFORMAT_JPEG)
        {
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
            fb_len = fb->len;
#endif
            res = httpd_resp_send(req, (const char *)fb->buf, fb->len);
        }
        else
        {
            jpg_chunking_t jchunk = {req, 0};
            res = frame2jpg_cb(fb, 80, jpg_encode_stream, &jchunk) ? ESP_OK :
ESP_FAIL;
            httpd_resp_send_chunk(req, NULL, 0);
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
            fb_len = jchunk.len;
#endif
        }
        esp_camera_fb_return(fb);
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        int64_t fr_end = esp_timer_get_time();
#endif
        log_i("JPG: %uB %ums", (uint32_t)(fb_len), (uint32_t)((fr_end - fr_start)
/ 1000));
        return res;
#if CONFIG_ESP_FACE_DETECT_ENABLED
    }

    jpg_chunking_t jchunk = {req, 0};

    if (fb->format == PIXFORMAT_RGB565
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
     && !recognition_enabled
#endif
     ){
```

```
#if TWO_STAGE
        HumanFaceDetectMSR01 s1(0.1F, 0.5F, 10, 0.2F);
        HumanFaceDetectMNP01 s2(0.5F, 0.3F, 5);
        std::list<dl::detect::result_t> &candidates = s1.infer((uint16_t *)fb-
>buf, {(int)fb->height, (int)fb->width, 3});
        std::list<dl::detect::result_t> &results = s2.infer((uint16_t *)fb->buf,
{(int)fb->height, (int)fb->width, 3}, candidates);
#else
        HumanFaceDetectMSR01 s1(0.3F, 0.5F, 10, 0.2F);
        std::list<dl::detect::result_t> &results = s1.infer((uint16_t *)fb->buf,
{(int)fb->height, (int)fb->width, 3});
#endif
        if (results.size() > 0) {
            fb_data_t rfb;
            rfb.width = fb->width;
            rfb.height = fb->height;
            rfb.data = fb->buf;
            rfb.bytes_per_pixel = 2;
            rfb.format = FB_RGB565;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
            detected = true;
#endif
            draw_face_boxes(&rfb, &results, face_id);
        }
        s = fmt2jpg_cb(fb->buf, fb->len, fb->width, fb->height, PIXFORMAT_RGB565,
90, jpg_encode_stream, &jchunk);
        esp_camera_fb_return(fb);
    } else
    {
        out_len = fb->width * fb->height * 3;
        out_width = fb->width;
        out_height = fb->height;
        out_buf = (uint8_t*)malloc(out_len);
        if (!out_buf) {
            log_e("out_buf malloc failed");
            httpd_resp_send_500(req);
            return ESP_FAIL;
        }
        s = fmt2rgb888(fb->buf, fb->len, fb->format, out_buf);
        esp_camera_fb_return(fb);
        if (!s) {
            free(out_buf);
            log_e("To rgb888 failed");
            httpd_resp_send_500(req);
            return ESP_FAIL;
        }

        fb_data_t rfb;
        rfb.width = out_width;
        rfb.height = out_height;
        rfb.data = out_buf;
        rfb.bytes_per_pixel = 3;
        rfb.format = FB_BGR888;

#if TWO_STAGE
        HumanFaceDetectMSR01 s1(0.1F, 0.5F, 10, 0.2F);
        HumanFaceDetectMNP01 s2(0.5F, 0.3F, 5);
        std::list<dl::detect::result_t> &candidates = s1.infer((uint8_t
*)out_buf, {(int)out_height, (int)out_width, 3});
        std::list<dl::detect::result_t> &results = s2.infer((uint8_t *)out_buf,
{(int)out_height, (int)out_width, 3}, candidates);
#else
        HumanFaceDetectMSR01 s1(0.3F, 0.5F, 10, 0.2F);
        std::list<dl::detect::result_t> &results = s1.infer((uint8_t *)out_buf,
{(int)out_height, (int)out_width, 3});
#endif

        if (results.size() > 0) {
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
            detected = true;
#endif
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
            if (recognition_enabled) {
                face_id = run_face_recognition(&rfb, &results);
            }
#endif
            draw_face_boxes(&rfb, &results, face_id);
        }

        s = fmt2jpg_cb(out_buf, out_len, out_width, out_height, PIXFORMAT_RGB888,
90, jpg_encode_stream, &jchunk);
        free(out_buf);
    }

    if (!s) {
        log_e("JPEG compression failed");
        httpd_resp_send_500(req);
        return ESP_FAIL;
    }
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
```

```
    int64_t fr_end = esp_timer_get_time();
#endif
    log_i("FACE: %uB %ums %s%d", (uint32_t)(jchunk.len), (uint32_t)((fr_end -
fr_start) / 1000), detected ? "DETECTED " : "", face_id);
    return res;
#endif
}

static esp_err_t stream_handler(httpd_req_t *req)
{
    camera_fb_t *fb = NULL;
    struct timeval _timestamp;
    esp_err_t res = ESP_OK;
    size_t _jpg_buf_len = 0;
    uint8_t *_jpg_buf = NULL;
    char *part_buf[128];
#if CONFIG_ESP_FACE_DETECT_ENABLED
    #if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        bool detected = false;
        int64_t fr_ready = 0;
        int64_t fr_recognize = 0;
        int64_t fr_encode = 0;
        int64_t fr_face = 0;
        int64_t fr_start = 0;
    #endif
    int face_id = 0;
    size_t out_len = 0, out_width = 0, out_height = 0;
    uint8_t *out_buf = NULL;
    bool s = false;
#if TWO_STAGE
    HumanFaceDetectMSR01 s1(0.1F, 0.5F, 10, 0.2F);
    HumanFaceDetectMNP01 s2(0.5F, 0.3F, 5);
#else
    HumanFaceDetectMSR01 s1(0.3F, 0.5F, 10, 0.2F);
#endif
#endif

    static int64_t last_frame = 0;
    if (!last_frame)
    {
        last_frame = esp_timer_get_time();
    }

    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");

#if CONFIG_LED_ILLUMINATOR_ENABLED
    isStreaming = true;
    enable_led(true);
#endif

    while (true)
    {
#if CONFIG_ESP_FACE_DETECT_ENABLED
    #if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        detected = false;
    #endif
        face_id = 0;
#endif

        fb = esp_camera_fb_get();
        if (!fb)
        {
            log_e("Camera capture failed");
            res = ESP_FAIL;
        }
        else
        {
            _timestamp.tv_sec = fb->timestamp.tv_sec;
            _timestamp.tv_usec = fb->timestamp.tv_usec;
#if CONFIG_ESP_FACE_DETECT_ENABLED
    #if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
            fr_start = esp_timer_get_time();
            fr_ready = fr_start;
            fr_encode = fr_start;
            fr_recognize = fr_start;
            fr_face = fr_start;
    #endif
            if (!detection_enabled || fb->width > 400)
            {
#endif
                if (fb->format != PIXFORMAT_JPEG)
                {
                    bool jpeg_converted = frame2jpg(fb, 80, &_jpg_buf,
&_jpg_buf_len);
                    esp_camera_fb_return(fb);
                    fb = NULL;
                    if (!jpeg_converted)
                    {
                        log_e("JPEG compression failed");
                        res = ESP_FAIL;
                    }
                }
                else
                {
                    _jpg_buf_len = fb->len;
                    _jpg_buf = fb->buf;
                }
#if CONFIG_ESP_FACE_DETECT_ENABLED
            }
            else
            {
                if (fb->format == PIXFORMAT_RGB565
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
                    && !recognition_enabled
#endif
                ){
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                    fr_ready = esp_timer_get_time();
#endif
#if TWO_STAGE
                    std::list<dl::detect::result_t> &candidates =
s1.infer((uint16_t *)fb->buf, {(int)fb->height, (int)fb->width, 3});
                    std::list<dl::detect::result_t> &results = s2.infer((uint16_t
*)fb->buf, {(int)fb->height, (int)fb->width, 3}, candidates);
#else
                    std::list<dl::detect::result_t> &results = s1.infer((uint16_t
*)fb->buf, {(int)fb->height, (int)fb->width, 3});
#endif
#if CONFIG_ESP_FACE_DETECT_ENABLED && ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                    fr_face = esp_timer_get_time();
                    fr_recognize = fr_face;
#endif
                    if (results.size() > 0) {
                        fb_data_t rfb;
                        rfb.width = fb->width;
                        rfb.height = fb->height;
                        rfb.data = fb->buf;
                        rfb.bytes_per_pixel = 2;
                        rfb.format = FB_RGB565;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                        detected = true;
#endif
                        draw_face_boxes(&rfb, &results, face_id);
                    }
                    s = fmt2jpg(fb->buf, fb->len, fb->width, fb->height,
PIXFORMAT_RGB565, 80, &_jpg_buf, &_jpg_buf_len);
                    esp_camera_fb_return(fb);
                    fb = NULL;
                    if (!s) {
                        log_e("fmt2jpg failed");
                        res = ESP_FAIL;
                    }
#if CONFIG_ESP_FACE_DETECT_ENABLED && ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                    fr_encode = esp_timer_get_time();
#endif
                } else
                {
                    out_len = fb->width * fb->height * 3;
                    out_width = fb->width;
                    out_height = fb->height;
                    out_buf = (uint8_t*)malloc(out_len);
                    if (!out_buf) {
                        log_e("out_buf malloc failed");
                        res = ESP_FAIL;
                    } else {
                        s = fmt2rgb888(fb->buf, fb->len, fb->format, out_buf);
                        esp_camera_fb_return(fb);
                        fb = NULL;
                        if (!s) {
                            free(out_buf);
                            log_e("To rgb888 failed");
                            res = ESP_FAIL;
                        } else {
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                            fr_ready = esp_timer_get_time();
#endif

                            fb_data_t rfb;
                            rfb.width = out_width;
                            rfb.height = out_height;
                            rfb.data = out_buf;
                            rfb.bytes_per_pixel = 3;
                            rfb.format = FB_BGR888;

#if TWO_STAGE
                            std::list<dl::detect::result_t> &candidates =
s1.infer((uint8_t *)out_buf, {(int)out_height, (int)out_width, 3});
                            std::list<dl::detect::result_t> &results =
s2.infer((uint8_t *)out_buf, {(int)out_height, (int)out_width, 3}, candidates);
#else
                            std::list<dl::detect::result_t> &results =
s1.infer((uint8_t *)out_buf, {(int)out_height, (int)out_width, 3});
#endif

#if CONFIG_ESP_FACE_DETECT_ENABLED && ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                            fr_face = esp_timer_get_time();
                            fr_recognize = fr_face;
#endif

                            if (results.size() > 0) {
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                                detected = true;
#endif
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
                                if (recognition_enabled) {
                                    face_id = run_face_recognition(&rfb,
&results);
    #if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                                    fr_recognize = esp_timer_get_time();
    #endif
                                }
#endif
                                draw_face_boxes(&rfb, &results, face_id);
                            }
                            s = fmt2jpg(out_buf, out_len, out_width, out_height,
PIXFORMAT_RGB888, 90, &_jpg_buf, &_jpg_buf_len);
                            free(out_buf);
                            if (!s) {
                                log_e("fmt2jpg failed");
                                res = ESP_FAIL;
                            }
#if CONFIG_ESP_FACE_DETECT_ENABLED && ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
                            fr_encode = esp_timer_get_time();
#endif
                        }
                    }
                }
```

```
            }
#endif
        }
        if (res == ESP_OK)
        {
            res = httpd_resp_send_chunk(req, _STREAM_BOUNDARY,
strlen(_STREAM_BOUNDARY));
        }
        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);
        }
        if (res == ESP_OK)
        {
            res = httpd_resp_send_chunk(req, (const char *)_jpg_buf,
_jpg_buf_len);
        }
        if (fb)
        {
            esp_camera_fb_return(fb);
            fb = NULL;
            _jpg_buf = NULL;
        }
        else if (_jpg_buf)
        {
            free(_jpg_buf);
            _jpg_buf = NULL;
        }
        if (res != ESP_OK)
        {
            log_e("Send frame failed");
            break;
        }
        int64_t fr_end = esp_timer_get_time();

#if CONFIG_ESP_FACE_DETECT_ENABLED && ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        int64_t ready_time = (fr_ready - fr_start) / 1000;
        int64_t face_time = (fr_face - fr_ready) / 1000;
        int64_t recognize_time = (fr_recognize - fr_face) / 1000;
        int64_t encode_time = (fr_encode - fr_recognize) / 1000;
        int64_t process_time = (fr_encode - fr_start) / 1000;
#endif
```

```
        int64_t frame_time = fr_end - last_frame;
        frame_time /= 1000;
#if ARDUHAL_LOG_LEVEL >= ARDUHAL_LOG_LEVEL_INFO
        uint32_t avg_frame_time = ra_filter_run(&ra_filter, frame_time);
#endif
        log_i("MJPG: %uB %ums (%.1ffps), AVG: %ums (%.1ffps)"
#if CONFIG_ESP_FACE_DETECT_ENABLED
                     ", %u+%u+%u+%u=%u %s%d"
#endif
                 ,
                 (uint32_t)(_jpg_buf_len),
                 (uint32_t)frame_time, 1000.0 / (uint32_t)frame_time,
                 avg_frame_time, 1000.0 / avg_frame_time
#if CONFIG_ESP_FACE_DETECT_ENABLED
                 ,
                 (uint32_t)ready_time, (uint32_t)face_time,
(uint32_t)recognize_time, (uint32_t)encode_time, (uint32_t)process_time,
                 (detected) ? "DETECTED " : "", face_id
#endif
        );
    }

#if CONFIG_LED_ILLUMINATOR_ENABLED
    isStreaming = false;
    enable_led(false);
#endif

    return res;
}

static esp_err_t parse_get(httpd_req_t *req, char **obuf)
{
    char *buf = NULL;
    size_t buf_len = 0;

    buf_len = httpd_req_get_url_query_len(req) + 1;
    if (buf_len > 1) {
        buf = (char *)malloc(buf_len);
        if (!buf) {
            httpd_resp_send_500(req);
            return ESP_FAIL;
        }
        if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK) {
            *obuf = buf;
            return ESP_OK;
        }
        free(buf);
    }
    httpd_resp_send_404(req);
    return ESP_FAIL;
}

static esp_err_t cmd_handler(httpd_req_t *req)
{
    char *buf = NULL;
    char variable[32];
    char value[32];

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }
    if (httpd_query_key_value(buf, "var", variable, sizeof(variable)) != ESP_OK
||
        httpd_query_key_value(buf, "val", value, sizeof(value)) != ESP_OK) {
        free(buf);
        httpd_resp_send_404(req);
        return ESP_FAIL;
    }
    free(buf);

    int val = atoi(value);
    log_i("%s = %d", variable, val);
    sensor_t *s = esp_camera_sensor_get();
    int res = 0;

    if (!strcmp(variable, "framesize")) {
        if (s->pixformat == PIXFORMAT_JPEG) {
            res = s->set_framesize(s, (framesize_t)val);
        }
    }
    else if (!strcmp(variable, "quality"))
        res = s->set_quality(s, val);
    else if (!strcmp(variable, "contrast"))
        res = s->set_contrast(s, val);
    else if (!strcmp(variable, "brightness"))
        res = s->set_brightness(s, val);
    else if (!strcmp(variable, "saturation"))
        res = s->set_saturation(s, val);
    else if (!strcmp(variable, "gainceiling"))
        res = s->set_gainceiling(s, (gainceiling_t)val);
    else if (!strcmp(variable, "colorbar"))
        res = s->set_colorbar(s, val);
    else if (!strcmp(variable, "awb"))
        res = s->set_whitebal(s, val);
    else if (!strcmp(variable, "agc"))
        res = s->set_gain_ctrl(s, val);
    else if (!strcmp(variable, "aec"))
        res = s->set_exposure_ctrl(s, val);
    else if (!strcmp(variable, "hmirror"))
        res = s->set_hmirror(s, val);
    else if (!strcmp(variable, "vflip"))
        res = s->set_vflip(s, val);
    else if (!strcmp(variable, "awb_gain"))
        res = s->set_awb_gain(s, val);
    else if (!strcmp(variable, "agc_gain"))
        res = s->set_agc_gain(s, val);
    else if (!strcmp(variable, "aec_value"))
        res = s->set_aec_value(s, val);
    else if (!strcmp(variable, "aec2"))
        res = s->set_aec2(s, val);
    else if (!strcmp(variable, "dcw"))
        res = s->set_dcw(s, val);
    else if (!strcmp(variable, "bpc"))
        res = s->set_bpc(s, val);
    else if (!strcmp(variable, "wpc"))
        res = s->set_wpc(s, val);
    else if (!strcmp(variable, "raw_gma"))
        res = s->set_raw_gma(s, val);
    else if (!strcmp(variable, "lenc"))
        res = s->set_lenc(s, val);
    else if (!strcmp(variable, "special_effect"))
        res = s->set_special_effect(s, val);
    else if (!strcmp(variable, "wb_mode"))
        res = s->set_wb_mode(s, val);
    else if (!strcmp(variable, "ae_level"))
        res = s->set_ae_level(s, val);
#if CONFIG_LED_ILLUMINATOR_ENABLED
    else if (!strcmp(variable, "led_intensity")) {
        led_duty = val;
        if (isStreaming)
            enable_led(true);
    }
#endif

#if CONFIG_ESP_FACE_DETECT_ENABLED
```

```
    else if (!strcmp(variable, "face_detect")) {
        detection_enabled = val;
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
        if (!detection_enabled) {
            recognition_enabled = 0;
        }
#endif
    }
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
    else if (!strcmp(variable, "face_enroll")){
        is_enrolling = !is_enrolling;
        log_i("Enrolling: %s", is_enrolling?"true":"false");
    }
    else if (!strcmp(variable, "face_recognize")) {
        recognition_enabled = val;
        if (recognition_enabled) {
            detection_enabled = val;
        }
    }
#endif
#endif
    else {
        log_i("Unknown command: %s", variable);
        res = -1;
    }

    if (res < 0) {
        return httpd_resp_send_500(req);
    }

    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, NULL, 0);
}

static int print_reg(char * p, sensor_t * s, uint16_t reg, uint32_t mask){
    return sprintf(p, "\"0x%x\":%u,", reg, s->get_reg(s, reg, mask));
}

static esp_err_t status_handler(httpd_req_t *req)
{
    static char json_response[1024];

    sensor_t *s = esp_camera_sensor_get();
    char *p = json_response;
    *p++ = '{';
```

```
if(s->id.PID == OV5640_PID || s->id.PID == OV3660_PID){
    for(int reg = 0x3400; reg < 0x3406; reg+=2){
        p+=print_reg(p, s, reg, 0xFFF);//12 bit
    }
    p+=print_reg(p, s, 0x3406, 0xFF);

    p+=print_reg(p, s, 0x3500, 0xFFFF0);//16 bit
    p+=print_reg(p, s, 0x3503, 0xFF);
    p+=print_reg(p, s, 0x350a, 0x3FF);//10 bit
    p+=print_reg(p, s, 0x350c, 0xFFFF);//16 bit

    for(int reg = 0x5480; reg <= 0x5490; reg++){
        p+=print_reg(p, s, reg, 0xFF);
    }

    for(int reg = 0x5380; reg <= 0x538b; reg++){
        p+=print_reg(p, s, reg, 0xFF);
    }

    for(int reg = 0x5580; reg < 0x558a; reg++){
        p+=print_reg(p, s, reg, 0xFF);
    }
    p+=print_reg(p, s, 0x558a, 0x1FF);//9 bit
} else if(s->id.PID == OV2640_PID){
    p+=print_reg(p, s, 0xd3, 0xFF);
    p+=print_reg(p, s, 0x111, 0xFF);
    p+=print_reg(p, s, 0x132, 0xFF);
}

p += sprintf(p, "\"xclk\":%u,", s->xclk_freq_hz / 1000000);
p += sprintf(p, "\"pixformat\":%u,", s->pixformat);
p += sprintf(p, "\"framesize\":%u,", s->status.framesize);
p += sprintf(p, "\"quality\":%u,", s->status.quality);
p += sprintf(p, "\"brightness\":%d,", s->status.brightness);
p += sprintf(p, "\"contrast\":%d,", s->status.contrast);
p += sprintf(p, "\"saturation\":%d,", s->status.saturation);
p += sprintf(p, "\"sharpness\":%d,", s->status.sharpness);
p += sprintf(p, "\"special_effect\":%u,", s->status.special_effect);
p += sprintf(p, "\"wb_mode\":%u,", s->status.wb_mode);
p += sprintf(p, "\"awb\":%u,", s->status.awb);
p += sprintf(p, "\"awb_gain\":%u,", s->status.awb_gain);
p += sprintf(p, "\"aec\":%u,", s->status.aec);
p += sprintf(p, "\"aec2\":%u,", s->status.aec2);
p += sprintf(p, "\"ae_level\":%d,", s->status.ae_level);
```

```
    p += sprintf(p, "\"aec_value\":%u,", s->status.aec_value);
    p += sprintf(p, "\"agc\":%u,", s->status.agc);
    p += sprintf(p, "\"agc_gain\":%u,", s->status.agc_gain);
    p += sprintf(p, "\"gainceiling\":%u,", s->status.gainceiling);
    p += sprintf(p, "\"bpc\":%u,", s->status.bpc);
    p += sprintf(p, "\"wpc\":%u,", s->status.wpc);
    p += sprintf(p, "\"raw_gma\":%u,", s->status.raw_gma);
    p += sprintf(p, "\"lenc\":%u,", s->status.lenc);
    p += sprintf(p, "\"hmirror\":%u,", s->status.hmirror);
    p += sprintf(p, "\"dcw\":%u,", s->status.dcw);
    p += sprintf(p, "\"colorbar\":%u", s->status.colorbar);
#if CONFIG_LED_ILLUMINATOR_ENABLED
    p += sprintf(p, ",\"led_intensity\":%u", led_duty);
#else
    p += sprintf(p, ",\"led_intensity\":%d", -1);
#endif
#if CONFIG_ESP_FACE_DETECT_ENABLED
    p += sprintf(p, ",\"face_detect\":%u", detection_enabled);
#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
    p += sprintf(p, ",\"face_enroll\":%u,", is_enrolling);
    p += sprintf(p, "\"face_recognize\":%u", recognition_enabled);
#endif
#endif
    *p++ = '}';
    *p++ = 0;
    httpd_resp_set_type(req, "application/json");
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, json_response, strlen(json_response));
}

static esp_err_t xclk_handler(httpd_req_t *req)
{
    char *buf = NULL;
    char _xclk[32];

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }
    if (httpd_query_key_value(buf, "xclk", _xclk, sizeof(_xclk)) != ESP_OK) {
        free(buf);
        httpd_resp_send_404(req);
        return ESP_FAIL;
    }
    free(buf);
    int xclk = atoi(_xclk);
    log_i("Set XCLK: %d MHz", xclk);

    sensor_t *s = esp_camera_sensor_get();
    int res = s->set_xclk(s, LEDC_TIMER_0, xclk);
    if (res) {
        return httpd_resp_send_500(req);
    }

    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, NULL, 0);
}

static esp_err_t reg_handler(httpd_req_t *req)
{
    char *buf = NULL;
    char _reg[32];
    char _mask[32];
    char _val[32];

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }
    if (httpd_query_key_value(buf, "reg", _reg, sizeof(_reg)) != ESP_OK ||
        httpd_query_key_value(buf, "mask", _mask, sizeof(_mask)) != ESP_OK ||
        httpd_query_key_value(buf, "val", _val, sizeof(_val)) != ESP_OK) {
        free(buf);
        httpd_resp_send_404(req);
        return ESP_FAIL;
    }
    free(buf);

    int reg = atoi(_reg);
    int mask = atoi(_mask);
    int val = atoi(_val);
    log_i("Set Register: reg: 0x%02x, mask: 0x%02x, value: 0x%02x", reg, mask,
val);

    sensor_t *s = esp_camera_sensor_get();
    int res = s->set_reg(s, reg, mask, val);
    if (res) {
        return httpd_resp_send_500(req);
    }

    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
```

```c
    return httpd_resp_send(req, NULL, 0);
}

static esp_err_t greg_handler(httpd_req_t *req)
{
    char *buf = NULL;
    char _reg[32];
    char _mask[32];

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }
    if (httpd_query_key_value(buf, "reg", _reg, sizeof(_reg)) != ESP_OK ||
        httpd_query_key_value(buf, "mask", _mask, sizeof(_mask)) != ESP_OK) {
        free(buf);
        httpd_resp_send_404(req);
        return ESP_FAIL;
    }
    free(buf);

    int reg = atoi(_reg);
    int mask = atoi(_mask);
    sensor_t *s = esp_camera_sensor_get();
    int res = s->get_reg(s, reg, mask);
    if (res < 0) {
        return httpd_resp_send_500(req);
    }
    log_i("Get Register: reg: 0x%02x, mask: 0x%02x, value: 0x%02x", reg, mask,
res);

    char buffer[20];
    const char * val = itoa(res, buffer, 10);
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, val, strlen(val));
}

static int parse_get_var(char *buf, const char * key, int def)
{
    char _int[16];
    if(httpd_query_key_value(buf, key, _int, sizeof(_int)) != ESP_OK){
        return def;
    }
    return atoi(_int);
}
```

```c
static esp_err_t pll_handler(httpd_req_t *req)
{
    char *buf = NULL;

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }

    int bypass = parse_get_var(buf, "bypass", 0);
    int mul = parse_get_var(buf, "mul", 0);
    int sys = parse_get_var(buf, "sys", 0);
    int root = parse_get_var(buf, "root", 0);
    int pre = parse_get_var(buf, "pre", 0);
    int seld5 = parse_get_var(buf, "seld5", 0);
    int pclken = parse_get_var(buf, "pclken", 0);
    int pclk = parse_get_var(buf, "pclk", 0);
    free(buf);

    log_i("Set Pll: bypass: %d, mul: %d, sys: %d, root: %d, pre: %d, seld5: %d,
pclken: %d, pclk: %d", bypass, mul, sys, root, pre, seld5, pclken, pclk);
    sensor_t *s = esp_camera_sensor_get();
    int res = s->set_pll(s, bypass, mul, sys, root, pre, seld5, pclken, pclk);
    if (res) {
        return httpd_resp_send_500(req);
    }

    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, NULL, 0);
}

static esp_err_t win_handler(httpd_req_t *req)
{
    char *buf = NULL;

    if (parse_get(req, &buf) != ESP_OK) {
        return ESP_FAIL;
    }

    int startX = parse_get_var(buf, "sx", 0);
    int startY = parse_get_var(buf, "sy", 0);
    int endX = parse_get_var(buf, "ex", 0);
    int endY = parse_get_var(buf, "ey", 0);
    int offsetX = parse_get_var(buf, "offx", 0);
    int offsetY = parse_get_var(buf, "offy", 0);
    int totalX = parse_get_var(buf, "tx", 0);
    int totalY = parse_get_var(buf, "ty", 0);
    int outputX = parse_get_var(buf, "ox", 0);
    int outputY = parse_get_var(buf, "oy", 0);
    bool scale = parse_get_var(buf, "scale", 0) == 1;
    bool binning = parse_get_var(buf, "binning", 0) == 1;
    free(buf);

    log_i("Set Window: Start: %d %d, End: %d %d, Offset: %d %d, Total: %d %d,
Output: %d %d, Scale: %u, Binning: %u", startX, startY, endX, endY, offsetX,
offsetY, totalX, totalY, outputX, outputY, scale, binning);
    sensor_t *s = esp_camera_sensor_get();
    int res = s->set_res_raw(s, startX, startY, endX, endY, offsetX, offsetY,
totalX, totalY, outputX, outputY, scale, binning);
    if (res) {
        return httpd_resp_send_500(req);
    }

    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
    return httpd_resp_send(req, NULL, 0);
}

static esp_err_t index_handler(httpd_req_t *req)
{
    httpd_resp_set_type(req, "text/html");
    httpd_resp_set_hdr(req, "Content-Encoding", "gzip");
    sensor_t *s = esp_camera_sensor_get();
    if (s != NULL) {
        if (s->id.PID == OV3660_PID) {
            return httpd_resp_send(req, (const char *)index_ov3660_html_gz,
index_ov3660_html_gz_len);
        } else if (s->id.PID == OV5640_PID) {
            return httpd_resp_send(req, (const char *)index_ov5640_html_gz,
index_ov5640_html_gz_len);
        } else {
            return httpd_resp_send(req, (const char *)index_ov2640_html_gz,
index_ov2640_html_gz_len);
        }
    } else {
        log_e("Camera sensor not found");
        return httpd_resp_send_500(req);
    }
}

void startCameraServer()
{
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    config.max_uri_handlers = 16;

    httpd_uri_t index_uri = {
        .uri = "/",
        .method = HTTP_GET,
        .handler = index_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t status_uri = {
        .uri = "/status",
        .method = HTTP_GET,
        .handler = status_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t cmd_uri = {
        .uri = "/control",
        .method = HTTP_GET,
        .handler = cmd_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t capture_uri = {
        .uri = "/capture",
        .method = HTTP_GET,
        .handler = capture_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t stream_uri = {
        .uri = "/stream",
        .method = HTTP_GET,
        .handler = stream_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t bmp_uri = {
        .uri = "/bmp",
        .method = HTTP_GET,
        .handler = bmp_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t xclk_uri = {
        .uri = "/xclk",
        .method = HTTP_GET,
        .handler = xclk_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t reg_uri = {
        .uri = "/reg",
        .method = HTTP_GET,
        .handler = reg_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t greg_uri = {
        .uri = "/greg",
        .method = HTTP_GET,
        .handler = greg_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t pll_uri = {
        .uri = "/pll",
        .method = HTTP_GET,
        .handler = pll_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    httpd_uri_t win_uri = {
        .uri = "/resolution",
        .method = HTTP_GET,
        .handler = win_handler,
        .user_ctx = NULL
#ifdef CONFIG_HTTPD_WS_SUPPORT
        ,
        .is_websocket = true,
        .handle_ws_control_frames = false,
        .supported_subprotocol = NULL
#endif
    };

    ra_filter_init(&ra_filter, 20);

#if CONFIG_ESP_FACE_RECOGNITION_ENABLED
    recognizer.set_partition(ESP_PARTITION_TYPE_DATA, ESP_PARTITION_SUBTYPE_ANY,
"fr");

    // load ids from flash partition
    recognizer.set_ids_from_flash();
#endif
    log_i("Starting web server on port: '%d'", config.server_port);
    if (httpd_start(&camera_httpd, &config) == ESP_OK)
    {
        httpd_register_uri_handler(camera_httpd, &index_uri);
        httpd_register_uri_handler(camera_httpd, &cmd_uri);
        httpd_register_uri_handler(camera_httpd, &status_uri);
        httpd_register_uri_handler(camera_httpd, &capture_uri);
        httpd_register_uri_handler(camera_httpd, &bmp_uri);

        httpd_register_uri_handler(camera_httpd, &xclk_uri);
        httpd_register_uri_handler(camera_httpd, &reg_uri);
        httpd_register_uri_handler(camera_httpd, &greg_uri);
        httpd_register_uri_handler(camera_httpd, &pll_uri);
        httpd_register_uri_handler(camera_httpd, &win_uri);
    }

    config.server_port += 1;
    config.ctrl_port += 1;
    log_i("Starting stream server on port: '%d'", config.server_port);
    if (httpd_start(&stream_httpd, &config) == ESP_OK)
    {
        httpd_register_uri_handler(stream_httpd, &stream_uri);
    }
}
```

```
void setupLedFlash(int pin)
{
    #if CONFIG_LED_ILLUMINATOR_ENABLED
    ledcSetup(LED_LEDC_CHANNEL, 5000, 8);
    ledcAttachPin(pin, LED_LEDC_CHANNEL);
    #else
    log_i("LED flash is disabled -> CONFIG_LED_ILLUMINATOR_ENABLED = 0");
    #endif
}
```