```
// 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 "camera_index.h"
#include "Arduino.h"

#include "fb_gfx.h"
#include "fd_forward.h"
#include "fr_forward.h"

#define ENROLL_CONFIRM_TIMES 5
#define FACE_ID_SAVE_NUMBER 7

#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)

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;

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\n\r\n";

```

```
 55 static ra_filter_t ra_filter;
 56 httpd_handle_t stream_httpd = NULL;
 57 httpd_handle_t camera_httpd = NULL;
 58
 59 static mtmn_config_t mtmn_config = { 0 };
 60 static int8_t detection_enabled = 0;
 61 static int8_t recognition_enabled = 0;
 62 static int8_t is_enrolling = 0;
 63 static face_id_list id_list = { 0 };
 64
 65 static ra_filter_t* ra_filter_init(ra_filter_t* filter, size_t sample_size) {
 66     memset(filter, 0, sizeof(ra_filter_t));
 67
 68     filter->values = (int*)malloc(sample_size * sizeof(int));
 69     if (!filter->values) {
 70         return NULL;
 71     }
 72     memset(filter->values, 0, sample_size * sizeof(int));
 73
 74     filter->size = sample_size;
 75     return filter;
 76 }
 77
 78 static int ra_filter_run(ra_filter_t* filter, int value) {
 79     if (!filter->values) {
 80         return value;
 81     }
 82     filter->sum -= filter->values[filter->index];
 83     filter->values[filter->index] = value;
 84     filter->sum += filter->values[filter->index];
 85     filter->index++;
 86     filter->index = filter->index % filter->size;
 87     if (filter->count < filter->size) {
 88         filter->count++;
 89     }
 90     return filter->sum / filter->count;
 91 }
 92
 93 static void rgb_print(dl_matrix3du_t* image_matrix, uint32_t color, const
      char* str) {
 94     fb_data_t fb;
 95     fb.width = image_matrix->w;
 96     fb.height = image_matrix->h;
 97     fb.data = image_matrix->item;
 98     fb.bytes_per_pixel = 3;
 99     fb.format = FB_BGR888;
100     fb_gfx_print(&fb, (fb.width - (strlen(str) * 14)) / 2, 10, color, str);
101 }
102
103 static int rgb_printf(dl_matrix3du_t* image_matrix, uint32_t color, const
      char* format, ...) {
104     char loc_buf[64];
105     char* temp = loc_buf;
106     int len;
107     va_list arg;
108     va_list copy;
```

```
109     va_start(arg, format);
110     va_copy(copy, arg);
111     len = vsnprintf(loc_buf, sizeof(loc_buf), format, arg);
112     va_end(copy);
113     if (len >= sizeof(loc_buf)) {
114         temp = (char*)malloc(len + 1);
115         if (temp == NULL) {
116             return 0;
117         }
118     }
119     vsnprintf(temp, len + 1, format, arg);
120     va_end(arg);
121     rgb_print(image_matrix, color, temp);
122     if (len > 64) {
123         free(temp);
124     }
125     return len;
126 }
127
128 static void draw_face_boxes(dl_matrix3du_t* image_matrix, box_array_t* boxes,
      int face_id) {
129     int x, y, w, h, i;
130     uint32_t color = FACE_COLOR_YELLOW;
131     if (face_id < 0) {
132         color = FACE_COLOR_RED;
133     }
134     else if (face_id > 0) {
135         color = FACE_COLOR_GREEN;
136     }
137     fb_data_t fb;
138     fb.width = image_matrix->w;
139     fb.height = image_matrix->h;
140     fb.data = image_matrix->item;
141     fb.bytes_per_pixel = 3;
142     fb.format = FB_BGR888;
143     for (i = 0; i < boxes->len; i++) {
144         // rectangle box
145         x = (int)boxes->box[i].box_p[0];
146         y = (int)boxes->box[i].box_p[1];
147         w = (int)boxes->box[i].box_p[2] - x + 1;
148         h = (int)boxes->box[i].box_p[3] - y + 1;
149         fb_gfx_drawFastHLine(&fb, x, y, w, color);
150         fb_gfx_drawFastHLine(&fb, x, y + h - 1, w, color);
151         fb_gfx_drawFastVLine(&fb, x, y, h, color);
152         fb_gfx_drawFastVLine(&fb, x + w - 1, y, h, color);
153 #if 0
154         // landmark
155         int x0, y0, j;
156         for (j = 0; j < 10; j += 2) {
157             x0 = (int)boxes->landmark[i].landmark_p[j];
158             y0 = (int)boxes->landmark[i].landmark_p[j + 1];
159             fb_gfx_fillRect(&fb, x0, y0, 3, 3, color);
160         }
161 #endif
162     }
163 }
```

```
164
165 static int run_face_recognition(dl_matrix3du_t* image_matrix, box_array_t*
      net_boxes) {
166     dl_matrix3du_t* aligned_face = NULL;
167     int matched_id = 0;
168
169     aligned_face = dl_matrix3du_alloc(1, FACE_WIDTH, FACE_HEIGHT, 3);
170     if (!aligned_face) {
171         Serial.println("Could not allocate face recognition buffer");
172         return matched_id;
173     }
174     if (align_face(net_boxes, image_matrix, aligned_face) == ESP_OK) {
175         if (is_enrolling == 1) {
176             int8_t left_sample_face = enroll_face(&id_list, aligned_face);
177
178             if (left_sample_face == (ENROLL_CONFIRM_TIMES - 1)) {
179                 Serial.printf("Enrolling Face ID: %d\n", id_list.tail);
180             }
181             Serial.printf("Enrolling Face ID: %d sample %d\n", id_list.tail,
                ENROLL_CONFIRM_TIMES - left_sample_face);
182             rgb_printf(image_matrix, FACE_COLOR_CYAN, "ID[%u] Sample[%u]",
                id_list.tail, ENROLL_CONFIRM_TIMES - left_sample_face);
183             if (left_sample_face == 0) {
184                 is_enrolling = 0;
185                 Serial.printf("Enrolled Face ID: %d\n", id_list.tail);
186             }
187         }
188         else {
189             matched_id = recognize_face(&id_list, aligned_face);
190             if (matched_id >= 0) {
191                 Serial.printf("Match Face ID: %u\n", matched_id);
192                 rgb_printf(image_matrix, FACE_COLOR_GREEN, "Hello Subject %u",
                    matched_id);
193             }
194             else {
195                 Serial.println("No Match Found");
196                 rgb_print(image_matrix, FACE_COLOR_RED, "Intruder Alert!");
197                 matched_id = -1;
198             }
199         }
200     }
201     else {
202         Serial.println("Face Not Aligned");
203         //rgb_print(image_matrix, FACE_COLOR_YELLOW, "Human Detected");
204     }
205
206     dl_matrix3du_free(aligned_face);
207     return matched_id;
208 }
209
210 static size_t jpg_encode_stream(void* arg, size_t index, const void* data,
      size_t len) {
211     jpg_chunking_t* j = (jpg_chunking_t*)arg;
212     if (!index) {
213         j->len = 0;
214     }
```

```
215     if (httpd_resp_send_chunk(j->req, (const char*)data, len) != ESP_OK) {
216         return 0;
217     }
218     j->len += len;
219     return len;
220 }
221
222 static esp_err_t capture_handler(httpd_req_t* req) {
223     camera_fb_t* fb = NULL;
224     esp_err_t res = ESP_OK;
225     int64_t fr_start = esp_timer_get_time();
226
227     fb = esp_camera_fb_get();
228     if (!fb) {
229         Serial.println("Camera capture failed");
230         httpd_resp_send_500(req);
231         return ESP_FAIL;
232     }
233
234     httpd_resp_set_type(req, "image/jpeg");
235     httpd_resp_set_hdr(req, "Content-Disposition", "inline;
        filename=capture.jpg");
236     httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
237
238     size_t out_len, out_width, out_height;
239     uint8_t* out_buf;
240     bool s;
241     bool detected = false;
242     int face_id = 0;
243     if (!detection_enabled || fb->width > 400) {
244         size_t fb_len = 0;
245         if (fb->format == PIXFORMAT_JPEG) {
246             fb_len = fb->len;
247             res = httpd_resp_send(req, (const char*)fb->buf, fb->len);
248         }
249         else {
250             jpg_chunking_t jchunk = { req, 0 };
251             res = frame2jpg_cb(fb, 80, jpg_encode_stream, &jchunk) ? ESP_OK :
                ESP_FAIL;
252             httpd_resp_send_chunk(req, NULL, 0);
253             fb_len = jchunk.len;
254         }
255         esp_camera_fb_return(fb);
256         int64_t fr_end = esp_timer_get_time();
257         Serial.printf("JPG: %uB %ums\n", (uint32_t)(fb_len), (uint32_t)
            ((fr_end - fr_start) / 1000));
258         return res;
259     }
260
261     dl_matrix3du_t* image_matrix = dl_matrix3du_alloc(1, fb->width, fb-
        >height, 3);
262     if (!image_matrix) {
263         esp_camera_fb_return(fb);
264         Serial.println("dl_matrix3du_alloc failed");
265         httpd_resp_send_500(req);
266         return ESP_FAIL;
```

```
267     }
268
269     out_buf = image_matrix->item;
270     out_len = fb->width * fb->height * 3;
271     out_width = fb->width;
272     out_height = fb->height;
273
274     s = fmt2rgb888(fb->buf, fb->len, fb->format, out_buf);
275     esp_camera_fb_return(fb);
276     if (!s) {
277         dl_matrix3du_free(image_matrix);
278         Serial.println("to rgb888 failed");
279         httpd_resp_send_500(req);
280         return ESP_FAIL;
281     }
282
283     box_array_t* net_boxes = face_detect(image_matrix, &mtmn_config);
284
285     if (net_boxes) {
286         detected = true;
287         if (recognition_enabled) {
288             face_id = run_face_recognition(image_matrix, net_boxes);
289         }
290         draw_face_boxes(image_matrix, net_boxes, face_id);
291         free(net_boxes->score);
292         free(net_boxes->box);
293         free(net_boxes->landmark);
294         free(net_boxes);
295     }
296
297     jpg_chunking_t jchunk = { req, 0 };
298     s = fmt2jpg_cb(out_buf, out_len, out_width, out_height, PIXFORMAT_RGB888,
          90, jpg_encode_stream, &jchunk);
299     dl_matrix3du_free(image_matrix);
300     if (!s) {
301         Serial.println("JPEG compression failed");
302         return ESP_FAIL;
303     }
304
305     int64_t fr_end = esp_timer_get_time();
306     Serial.printf("FACE: %uB %ums %s%d\n", (uint32_t)(jchunk.len), (uint32_t)
          ((fr_end - fr_start) / 1000), detected ? "DETECTED " : "", face_id);
307     return res;
308 }
309
310 static esp_err_t stream_handler(httpd_req_t* req) {
311     camera_fb_t* fb = NULL;
312     esp_err_t res = ESP_OK;
313     size_t _jpg_buf_len = 0;
314     uint8_t* _jpg_buf = NULL;
315     char* part_buf[64];
316     dl_matrix3du_t* image_matrix = NULL;
317     bool detected = false;
318     int face_id = 0;
319     int64_t fr_start = 0;
320     int64_t fr_ready = 0;
```

```
321     int64_t fr_face = 0;
322     int64_t fr_recognize = 0;
323     int64_t fr_encode = 0;
324
325     static int64_t last_frame = 0;
326     if (!last_frame) {
327         last_frame = esp_timer_get_time();
328     }
329
330     res = httpd_resp_set_type(req, _STREAM_CONTENT_TYPE);
331     if (res != ESP_OK) {
332         return res;
333     }
334
335     httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
336
337     while (true) {
338         detected = false;
339         face_id = 0;
340         fb = esp_camera_fb_get();
341         if (!fb) {
342             Serial.println("Camera capture failed");
343             res = ESP_FAIL;
344         }
345         else {
346             fr_start = esp_timer_get_time();
347             fr_ready = fr_start;
348             fr_face = fr_start;
349             fr_encode = fr_start;
350             fr_recognize = fr_start;
351             if (!detection_enabled || fb->width > 400) {
352                 if (fb->format != PIXFORMAT_JPEG) {
353                     bool jpeg_converted = frame2jpg(fb, 80, &_jpg_buf,
                        &_jpg_buf_len);
354                     esp_camera_fb_return(fb);
355                     fb = NULL;
356                     if (!jpeg_converted) {
357                         Serial.println("JPEG compression failed");
358                         res = ESP_FAIL;
359                     }
360                 }
361                 else {
362                     _jpg_buf_len = fb->len;
363                     _jpg_buf = fb->buf;
364                 }
365             }
366             else {
367
368                 image_matrix = dl_matrix3du_alloc(1, fb->width, fb->height,
                      3);
369
370                 if (!image_matrix) {
371                     Serial.println("dl_matrix3du_alloc failed");
372                     res = ESP_FAIL;
373                 }
374                 else {
```

```
375                     if (!fmt2rgb888(fb->buf, fb->len, fb->format,
                        image_matrix->item)) {
376                         Serial.println("fmt2rgb888 failed");
377                         res = ESP_FAIL;
378                     }
379                     else {
380                         fr_ready = esp_timer_get_time();
381                         box_array_t* net_boxes = NULL;
382                         if (detection_enabled) {
383                             net_boxes = face_detect(image_matrix,
                        &mtmn_config);
384                         }
385                         fr_face = esp_timer_get_time();
386                         fr_recognize = fr_face;
387                         if (net_boxes || fb->format != PIXFORMAT_JPEG) {
388                             if (net_boxes) {
389                                 detected = true;
390                                 if (recognition_enabled) {
391                                     face_id = run_face_recognition
                        (image_matrix, net_boxes);
392                                 }
393                                 fr_recognize = esp_timer_get_time();
394                                 draw_face_boxes(image_matrix, net_boxes,
                        face_id);
395                                 free(net_boxes->score);
396                                 free(net_boxes->box);
397                                 free(net_boxes->landmark);
398                                 free(net_boxes);
399                             }
400                             if (!fmt2jpg(image_matrix->item, fb->width * fb-
                        >height * 3, fb->width, fb->height, PIXFORMAT_RGB888, 90,
                        &_jpg_buf, &_jpg_buf_len)) {
401                                 Serial.println("fmt2jpg failed");
402                                 res = ESP_FAIL;
403                             }
404                             esp_camera_fb_return(fb);
405                             fb = NULL;
406                         }
407                         else {
408                             _jpg_buf = fb->buf;
409                             _jpg_buf_len = fb->len;
410                         }
411                         fr_encode = esp_timer_get_time();
412                     }
413                     dl_matrix3du_free(image_matrix);
414                 }
415             }
416         }
417         if (res == ESP_OK) {
418             res = httpd_resp_send_chunk(req, _STREAM_BOUNDARY, strlen
                  (_STREAM_BOUNDARY));
419         }
420         if (res == ESP_OK) {
421             size_t hlen = snprintf((char*)part_buf, 64, _STREAM_PART,
                  _jpg_buf_len);
422             res = httpd_resp_send_chunk(req, (const char*)part_buf, hlen);
```

```
423         }
424         if (res == ESP_OK) {
425             res = httpd_resp_send_chunk(req, (const char*)_jpg_buf,
                  _jpg_buf_len);
426         }
427         if (fb) {
428             esp_camera_fb_return(fb);
429             fb = NULL;
430             _jpg_buf = NULL;
431         }
432         else if (_jpg_buf) {
433             free(_jpg_buf);
434             _jpg_buf = NULL;
435         }
436         if (res != ESP_OK) {
437             break;
438         }
439         int64_t fr_end = esp_timer_get_time();
440
441         int64_t ready_time = (fr_ready - fr_start) / 1000;
442         int64_t face_time = (fr_face - fr_ready) / 1000;
443         int64_t recognize_time = (fr_recognize - fr_face) / 1000;
444         int64_t encode_time = (fr_encode - fr_recognize) / 1000;
445         int64_t process_time = (fr_encode - fr_start) / 1000;
446
447         int64_t frame_time = fr_end - last_frame;
448         last_frame = fr_end;
449         frame_time /= 1000;
450         uint32_t avg_frame_time = ra_filter_run(&ra_filter, frame_time);
451         Serial.printf("MJPG: %uB %ums (%.1ffps), AVG: %ums (%.1ffps), %u+%u+%u
              +%u=%u %s%d\n",
452             (uint32_t)(_jpg_buf_len),
453             (uint32_t)frame_time, 1000.0 / (uint32_t)frame_time,
454             avg_frame_time, 1000.0 / avg_frame_time,
455             (uint32_t)ready_time, (uint32_t)face_time, (uint32_t)
                recognize_time, (uint32_t)encode_time, (uint32_t)process_time,
456             (detected) ? "DETECTED " : "", face_id
457         );
458     }
459
460     last_frame = 0;
461     return res;
462 }
463
464 static esp_err_t cmd_handler(httpd_req_t* req) {
465     char* buf;
466     size_t buf_len;
467     char variable[32] = { 0, };
468     char value[32] = { 0, };
469
470     buf_len = httpd_req_get_url_query_len(req) + 1;
471     if (buf_len > 1) {
472         buf = (char*)malloc(buf_len);
473         if (!buf) {
474             httpd_resp_send_500(req);
475             return ESP_FAIL;
```

```
476         }
477         if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK) {
478             if (httpd_query_key_value(buf, "var", variable, sizeof(variable))
                == ESP_OK &&
479                 httpd_query_key_value(buf, "val", value, sizeof(value)) ==
                    ESP_OK) {
480             }
481             else {
482                 free(buf);
483                 httpd_resp_send_404(req);
484                 return ESP_FAIL;
485             }
486         }
487         else {
488             free(buf);
489             httpd_resp_send_404(req);
490             return ESP_FAIL;
491         }
492         free(buf);
493     }
494     else {
495         httpd_resp_send_404(req);
496         return ESP_FAIL;
497     }
498
499     int val = atoi(value);
500     sensor_t* s = esp_camera_sensor_get();
501     int res = 0;
502
503     if (!strcmp(variable, "framesize")) {
504         if (s->pixformat == PIXFORMAT_JPEG) res = s->set_framesize(s,
              (framesize_t)val);
505     }
506     else if (!strcmp(variable, "quality")) res = s->set_quality(s, val);
507     else if (!strcmp(variable, "contrast")) res = s->set_contrast(s, val);
508     else if (!strcmp(variable, "brightness")) res = s->set_brightness(s, val);
509     else if (!strcmp(variable, "saturation")) res = s->set_saturation(s, val);
510     else if (!strcmp(variable, "gainceiling")) res = s->set_gainceiling(s,
          (gainceiling_t)val);
511     else if (!strcmp(variable, "colorbar")) res = s->set_colorbar(s, val);
512     else if (!strcmp(variable, "awb")) res = s->set_whitebal(s, val);
513     else if (!strcmp(variable, "agc")) res = s->set_gain_ctrl(s, val);
514     else if (!strcmp(variable, "aec")) res = s->set_exposure_ctrl(s, val);
515     else if (!strcmp(variable, "hmirror")) res = s->set_hmirror(s, val);
516     else if (!strcmp(variable, "vflip")) res = s->set_vflip(s, val);
517     else if (!strcmp(variable, "awb_gain")) res = s->set_awb_gain(s, val);
518     else if (!strcmp(variable, "agc_gain")) res = s->set_agc_gain(s, val);
519     else if (!strcmp(variable, "aec_value")) res = s->set_aec_value(s, val);
520     else if (!strcmp(variable, "aec2")) res = s->set_aec2(s, val);
521     else if (!strcmp(variable, "dcw")) res = s->set_dcw(s, val);
522     else if (!strcmp(variable, "bpc")) res = s->set_bpc(s, val);
523     else if (!strcmp(variable, "wpc")) res = s->set_wpc(s, val);
524     else if (!strcmp(variable, "raw_gma")) res = s->set_raw_gma(s, val);
525     else if (!strcmp(variable, "lenc")) res = s->set_lenc(s, val);
526     else if (!strcmp(variable, "special_effect")) res = s->set_special_effect
          (s, val);
```

```
527     else if (!strcmp(variable, "wb_mode")) res = s->set_wb_mode(s, val);
528     else if (!strcmp(variable, "ae_level")) res = s->set_ae_level(s, val);
529     else if (!strcmp(variable, "face_detect")) {
530         detection_enabled = val;
531         if (!detection_enabled) {
532             recognition_enabled = 0;
533         }
534     }
535     else if (!strcmp(variable, "face_enroll")) is_enrolling = val;
536     else if (!strcmp(variable, "face_recognize")) {
537         recognition_enabled = val;
538         if (recognition_enabled) {
539             detection_enabled = val;
540         }
541     }
542     else {
543         res = -1;
544     }
545 
546     if (res) {
547         return httpd_resp_send_500(req);
548     }
549 
550     httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
551     return httpd_resp_send(req, NULL, 0);
552 }
553 
554 static esp_err_t status_handler(httpd_req_t* req) {
555     static char json_response[1024];
556 
557     sensor_t* s = esp_camera_sensor_get();
558     char* p = json_response;
559     *p++ = '{';
560 
561     p += sprintf(p, "\"framesize\":%u,", s->status.framesize);
562     p += sprintf(p, "\"quality\":%u,", s->status.quality);
563     p += sprintf(p, "\"brightness\":%d,", s->status.brightness);
564     p += sprintf(p, "\"contrast\":%d,", s->status.contrast);
565     p += sprintf(p, "\"saturation\":%d,", s->status.saturation);
566     p += sprintf(p, "\"sharpness\":%d,", s->status.sharpness);
567     p += sprintf(p, "\"special_effect\":%u,", s->status.special_effect);
568     p += sprintf(p, "\"wb_mode\":%u,", s->status.wb_mode);
569     p += sprintf(p, "\"awb\":%u,", s->status.awb);
570     p += sprintf(p, "\"awb_gain\":%u,", s->status.awb_gain);
571     p += sprintf(p, "\"aec\":%u,", s->status.aec);
572     p += sprintf(p, "\"aec2\":%u,", s->status.aec2);
573     p += sprintf(p, "\"ae_level\":%d,", s->status.ae_level);
574     p += sprintf(p, "\"aec_value\":%u,", s->status.aec_value);
575     p += sprintf(p, "\"agc\":%u,", s->status.agc);
576     p += sprintf(p, "\"agc_gain\":%u,", s->status.agc_gain);
577     p += sprintf(p, "\"gainceiling\":%u,", s->status.gainceiling);
578     p += sprintf(p, "\"bpc\":%u,", s->status.bpc);
579     p += sprintf(p, "\"wpc\":%u,", s->status.wpc);
580     p += sprintf(p, "\"raw_gma\":%u,", s->status.raw_gma);
581     p += sprintf(p, "\"lenc\":%u,", s->status.lenc);
582     p += sprintf(p, "\"vflip\":%u,", s->status.vflip);
```

```
583     p += sprintf(p, "\"hmirror\":%u,", s->status.hmirror);
584     p += sprintf(p, "\"dcw\":%u,", s->status.dcw);
585     p += sprintf(p, "\"colorbar\":%u,", s->status.colorbar);
586     p += sprintf(p, "\"face_detect\":%u,", detection_enabled);
587     p += sprintf(p, "\"face_enroll\":%u,", is_enrolling);
588     p += sprintf(p, "\"face_recognize\":%u", recognition_enabled);
589     *p++ = '}';
590     *p++ = 0;
591     httpd_resp_set_type(req, "application/json");
592     httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*");
593     return httpd_resp_send(req, json_response, strlen(json_response));
594 }
595 
596 static esp_err_t index_handler(httpd_req_t* req) {
597     httpd_resp_set_type(req, "text/html");
598     httpd_resp_set_hdr(req, "Content-Encoding", "gzip");
599     sensor_t* s = esp_camera_sensor_get();
600     if (s->id.PID == OV3660_PID) {
601         return httpd_resp_send(req, (const char*)index_ov3660_html_gz,
          index_ov3660_html_gz_len);
602     }
603     return httpd_resp_send(req, (const char*)index_ov2640_html_gz,
      index_ov2640_html_gz_len);
604 }
605 
606 void startCameraServer() {
607     httpd_config_t config = HTTPD_DEFAULT_CONFIG();
608 
609     httpd_uri_t index_uri = {
610         .uri = "/",
611         .method = HTTP_GET,
612         .handler = index_handler,
613         .user_ctx = NULL
614     };
615 
616     httpd_uri_t status_uri = {
617         .uri = "/status",
618         .method = HTTP_GET,
619         .handler = status_handler,
620         .user_ctx = NULL
621     };
622 
623     httpd_uri_t cmd_uri = {
624         .uri = "/control",
625         .method = HTTP_GET,
626         .handler = cmd_handler,
627         .user_ctx = NULL
628     };
629 
630     httpd_uri_t capture_uri = {
631         .uri = "/capture",
632         .method = HTTP_GET,
633         .handler = capture_handler,
634         .user_ctx = NULL
635     };
636 
```

```
637     httpd_uri_t stream_uri = {
638          .uri = "/stream",
639          .method = HTTP_GET,
640          .handler = stream_handler,
641          .user_ctx = NULL
642     };
643
644
645     ra_filter_init(&ra_filter, 20);
646
647     mtmn_config.type = FAST;
648     mtmn_config.min_face = 80;
649     mtmn_config.pyramid = 0.707;
650     mtmn_config.pyramid_times = 4;
651     mtmn_config.p_threshold.score = 0.6;
652     mtmn_config.p_threshold.nms = 0.7;
653     mtmn_config.p_threshold.candidate_number = 20;
654     mtmn_config.r_threshold.score = 0.7;
655     mtmn_config.r_threshold.nms = 0.7;
656     mtmn_config.r_threshold.candidate_number = 10;
657     mtmn_config.o_threshold.score = 0.7;
658     mtmn_config.o_threshold.nms = 0.7;
659     mtmn_config.o_threshold.candidate_number = 1;
660
661     face_id_init(&id_list, FACE_ID_SAVE_NUMBER, ENROLL_CONFIRM_TIMES);
662
663     Serial.printf("Starting web server on port: '%d'\n", config.server_port);
664     if (httpd_start(&camera_httpd, &config) == ESP_OK) {
665         httpd_register_uri_handler(camera_httpd, &index_uri);
666         httpd_register_uri_handler(camera_httpd, &cmd_uri);
667         httpd_register_uri_handler(camera_httpd, &status_uri);
668         httpd_register_uri_handler(camera_httpd, &capture_uri);
669     }
670
671     config.server_port += 1;
672     config.ctrl_port += 1;
673     Serial.printf("Starting stream server on port: '%d'\n",
        config.server_port);
674     if (httpd_start(&stream_httpd, &config) == ESP_OK) {
675         httpd_register_uri_handler(stream_httpd, &stream_uri);
676     }
677 }
678
```