FILE SERVER

```
#include <stdio.h>
#include <string.h>
#include <sys/param.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include <esp_err.h>
#include <esp_log.h>
#include <esp_vfs.h>
#include <esp_spiffs.h>
#include <esp_http_server.h>


#define FILE_PATH_MAX (ESP_VFS_PATH_MAX + CONFIG_SPIFFS_OBJ_NAME_LEN)
#define MAX_FILE_SIZE (200 * 1024)
#define MAX_FILE_SIZE_STR "200KB"
#define SCRATCH_BUFSIZE 8192


struct file_server_data {
  char base_path[ESP_VFS_PATH_MAX + 1];
  char scratch[SCRATCH_BUFSIZE];
};


static const char *TAG = "file_server";
```

```
static esp_err_t index_html_get_handler(httpd_req_t *req){
    httpd_resp_set_status(req, "307 Temporary Redirect");
    httpd_resp_set_hdr(req, "Location", "/");
    httpd_resp_send(req, NULL, 0);
    return ESP_OK;
}


static esp_err_t favicon_get_handler(httpd_req_t *req){
    extern const unsigned char favicon_ico_start[]
                asm("_binary_favicon_ico_start");
    extern const unsigned char favicon_ico_end[]
                asm("_binary_favicon_ico_end");
    const size_t favicon_ico_size = (favicon_ico_end - favicon_ico_start);
    httpd_resp_set_type(req, "image/x-icon");
    httpd_resp_send(req, (const char*) favicon_ico_start, favicon_ico_size);
    return ESP_OK;
}


static esp_err_t http_resp_dir_html(httpd_req_t *req, const char *dirpath){
    char entrypath[FILE_PATH_MAX];
    char entrysize[16];
    const char *entrytype;
    struct dirent *entry;
    struct stat entrystat;

    DIR *dir = opendir(dirpath);
    const size_t dirpath_len = strlen(dirpath);
```

```
strlcpy(entrypath, dirpath, sizeof(entrypath));
if (!dir) {
    ESP_LOGE(TAG, "Failed to stat dir : %s", dirpath);
    httpd_resp_send_err(req, HTTPD_404_NOT_FOUND,
                        "Directory does not exist");
    return ESP_FAIL;
}

httpd_resp_sendstr_chunk(req, "<!DOCTYPE html><html><body>");
extern const unsigned char upload_script_start[]
        asm("_binary_upload_script_html_start");
extern const unsigned char upload_script_end[]
        asm("_binary_upload_script_html_end");
const size_t upload_script_size =
    (upload_script_end - upload_script_start);
httpd_resp_send_chunk(req, (const char*) upload_script_start,
                      upload_script_end);
httpd_resp_sendstr_chunk(req,
    "<table class=\"fixed\" border=\"1\">"
    "<col width=\"800px\" /><col width=\"300px\" />"
    "<col width=\"100px\" />"
    "<thead><tr><th>Name</th><th>Type</th>
    <th>Size (Bytes)</th><th>Delete</th></tr></thead>"
    "<tbody>");

while ((entry = readdir(dir)) != NULL) {
    entrytype = (entry->d_type == DT_DIR ? "directory" : "file");
    strlcpy(entrypath + dirpath_len, entry->d_name,
        sizeof(entrypath) - dirpath_len);
```

```
if (stat(entrypath, &entry_stat) == -1) {
    ESP_LOGE(TAG, "Failed to stat %s : %s",
        entrytype, entry->d_name);
    continue;
}

sprintf(entrysize, "%ld", entry.st_size);
ESP_LOGI(TAG, "Found %s (%s bytes)", entrytype,
    entry->d_name, entrysize);
httpd_resp_sendstr_chunk(req, "<tr><td><a href=\"");
httpd_resp_sendstr_chunk(req, req->uri);
httpd_resp_sendstr_chunk(req, entry->d_name);
if (entry->d_type == DT_DIR) {
    httpd_resp_sendstr_chunk(req, "/");
}
httpd_resp_sendstr_chunk(req, "\">");
httpd_resp_sendstr_chunk(req, entry->d_name);
httpd_resp_sendstr_chunk(req, "</a></td><td>");
httpd_resp_sendstr_chunk(req, entrytype);
httpd_resp_sendstr_chunk(req, "</td><td>");
httpd_resp_sendstr_chunk(req,
    "<form method=\"POST\" action=\"/delete");
httpd_resp_sendstr_chunk(req, req->uri);
httpd_resp_sendstr_chunk(req, entry->d_name);
httpd_resp_sendstr_chunk(req, "\"><button type=\"submit\">
    Delete</button></form>");
httpd_resp_sendstr_chunk(req, "</td></tr>\n");
}
closedir(dir);
httpd_resp_sendstr_chunk(req, "</tbody></table>");
```

```
    httpd_resp_sendstr_chunk(req, "</body></html>");
    httpd_resp_send_str_chunk(req, NULL);
    return ESP_OK;
}

#define IS_FILE_EXT(filename, ext) \
    (strcasecmp(&filename[strlen(filename) - sizeof(ext) + 1], ext) == 0)

static esp_err_t set_content_type_from_file(
    httpd_req_t *req, const char *filename) {
    if (IS_FILE_EXT(filename, ".pdf")) {
        return httpd_resp_set_type(req, "application/pdf");
    } else if (IS_FILE_EXT(filename, ".html")) {
        return httpd_resp_set_type(req, "text/html");
    } else if (IS_FILE_EXT(filename, ".jpeg")) {
        return httpd_resp_set_type(req, "image/jpeg");
    } else if (IS_FILE_EXT(filename, ".ico")) {
        return httpd_resp_set_type(req, "image/x-icon");
    }
    return httpd_resp_set_type(req, "text/plain");
}

static const char* get_path_from_uri(char *dest,
    const char* base_path, const char *uri, size_t destsize) {
    const size_t base_pathlen = strlen(base_path);
    size_t pathlen = strlen(uri);
```

```
const char * quest = strchr(uri, '?');
if (quest) {
    pathlen = MIN(pathlen, quest - uri);
}
const char *hash = strchr(uri, '#');
if (hash) {
    pathlen = MIN(pathlen, hash - uri);
}
if (base_pathlen + pathlen + 1 > dest_size) {
    return NULL;
}
strcpy(dest, base_path);
strlcpy(dest + base_pathlen, uri, pathlen + 1);
return dest + base_pathlen;
}

static esp_err_t download_get_handler(httpd_req_t *req) {
    char filepath[FILE_PATH_MAX];
    FILE *fd = NULL;
    struct stat file_stat;
    const char *filename = get_path_from_uri(
        filepath, ((struct file_server_data*) req->user_ctx)->
        base_path, req->uri, sizeof(filepath));
    if (!filename) {
        ESP_LOGE(TAG, "Filename is too long");
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
            "Filename too long");
        return ESP_FAIL;
    }
```

```
if (filename[strlen(filename) - 1] == '/') {
    return httpd_resp_dir_html(req, filepath);
}

if (stat(filepath, &file_stat) == -1) {
    if (strcmp(filename, "/index.html") == 0) {
        return index_html_get_handler(req);
    } else if (strcmp(filename, "/favicon.ico") == 0) {
        return favicon_get_handler(req);
    }
    ESP_LOGE(TAG, "failed to stat file : %s", filepath);
    httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                        "Failed to read existing file");
    return ESP_FAIL;
}

ESP_LOGI(TAG, "sending file : %s (%ld bytes)...",
         filename, file_stat.st_size);
set_content_type_from_file(req, filename);

char *chunk = ((struct file_server_data *)
               req->user_ctx)->scratch;
size_t chunksize;
do {
    chunksize = fread(chunk, 1, SCRATCH_BUFSIZE, fd);
    if (httpd_resp_send_chunk(req, chunk, chunksize) != ESP_OK) {
        fclose(fd);
        ESP_LOGE(TAG, "File sending failed!");
        httpd_resp_sendstr_chunk(req, NULL);
        httpd_resp_send_err(req,
            HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to send file");
        return ESP_FAIL;
    }
```

```
} while (chunksize != 0);
fclose(fd);
ESP_LOGI(TAG, "File sending complete");
httpd_resp_send_chunk(req, NULL, 0);
return ESP_OK;
}


static esp_err_t upload_post_handler(httpd_req_t *req) {
    char filepath[FILE_PATH_MAX];
    FILE *fd = NULL;
    struct stat file_stat;

    const char *filename = get_path_from_uri(filepath,
        ((struct file_server_data *)req->user_ctx)->base_path,
        req->uri + sizeof("/upload") - 1, sizeof(filepath));
    if (!filename) {
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                            "Filename too long");
        return ESP_FAIL;
    }
    if (filename[strlen(filename) - 1] == '/') {
        ESP_LOGE(TAG, "Invalid filename : %s", filename);
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                            "Invalid filename");
        return ESP_FAIL;
    }
```

```
if (stat(filepath, &file_stat) == 0) {
    ESP_LOGE(TAG, "File already exists : %s", filepath);
    httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST,
                        "File already exists");
    return ESP_FAIL;
}

if (req->content_len > MAX_FILE_SIZE) {
    ESP_LOGE(TAG, "File too large : %d bytes", req->content_len);
    httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST,
        "File size must be less than " MAX_FILE_SIZE_STR "!");
    return ESP_FAIL;

fd = fopen(filepath, "w");
if (!fd) {
    ESP_LOGE(TAG, "Failed to create file : %s", filepath);
    httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                        "Failed to create file");
    return ESP_FAIL;
}
ESP_LOGI(TAG, "Recieving file : %s ...", filename);
                                        scratch;
char *buf = ((struct file_server_data *)req->user_ctx)->
int recieved;
int remaining = req->content_len;

while (remaining > 0) {
    ESP_LOGI(TAG, "Remaining size : %d", remaining);
    if((recieved = httpd_req_recv(req, buf,
        MIN(remaining, SCRATCH_BUFSIZE))) <= 0) {
```

```
if (recieved == HTTPD_SOCK_ERR_TIMEOUT) {
    continue;
}
fclose (fd);
unlink (filepath);
ESP_LOGE(TAG, "File reception failed!");
httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to recieve file");
return ESP_FAIL;
}
if (recieved && (recieved != fwrite(buf, 1, recieved, fd))) {
    fclose (fd);
    unlink (filepath);
    ESP_LOGE(TAG, "File write failed!");
    httpd_resp_send_err(req, HTTPD_500_INTERANAL_SERVER_ERROR, "Failed to write file to storage");
    return ESP_FAIL;
}
remaining -= recieved;
fclose (fd);
ESP_LOGI(TAG, "File reception complete");
httpd_resp_set_status(req, "303 See Other");
httpd_resp_set_hdr(req, "Location", "/");
httpd_resp_sendstr(req, "File uploaded successfully");
return ESP_OK;
}
```

```
static esp_err_t delete_post_handler(httpd_req_t *req) {
    char filepath[FILE_PATH_MAX];
    struct stat file_stat;

    const char *filename = get_path_from_uri(filepath,
    ((struct file_server_data*)req->user_ctx)->base_path,
    req->uri + sizeof("/delete") -1, sizeof(filepath));

    if(!filename) {
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                                "Filename too long");
        return ESP_FAIL;
    }

    if(filename[strlen(filename) -1] == '/') {
        ESP_LOGE(TAG, "Invalid filename : %s", filename);
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
                                "Invalid filename");
        return ESP_FAIL;
    }
    if(stat(filepath, &file_stat) == -1) {
        ESP_LOGE(TAG, "File does not exist : %s", filename);
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST,
                                "File does not exist");
        return ESP_FAIL;
    }

    ESP_LOGI(TAG, "Deleting file : %s", filename);
    unlink(filepath);
    httpd_resp_set_status(req, "303 See other");
    httpd_resp_set_hdr(req, "Location", "/");
```

```
    httpd_resp_sendstr(req, "File deleted successfully");
    return ESP_OK;
}


esp_err_t start_file_server(const char *base_path) {
    static struct file_server_data *server_data = NULL;
    if(!base_path || strcmp(base_path, "/spiffs") != 0) {
        ESP_LOGE(TAG, "File server presently supports only
            'spiffs' as base_path");
        return ESP_ERR_INVALID_ARG;
    }
    if(server_data) {
        ESP_LOGE(TAG, "File server already started");
        return ESP_ERR_INVALID_STATE;
    }
    server_data = calloc(1, sizeof(struct file_server_data));
    if(!server_data) {
        ESP_LOGE(TAG, "Failed to allocate memory for server data");
        return ESP_ERR_NO_MEM;
    }
    strlcpy(server_data->base_path, base_path,
            sizeof(server_data->base_path));

    httpd_handle_t server = NULL;
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    config.uri_match_fn = httpd_uri_match_wildcard;
```

```
ESP_LOGI(TAG, "Starting HTTP server");
if (httpd_start(&server, &config) != ESP_OK) {
    ESP_LOGE(TAG, "Failed to start file server");
    return ESP_FAIL;
}
httpd_uri_t file_download = {
    .uri = "/*",
    .method = HTTP_GET,
    .handler = download_get_handler,
    .user_ctx = server_data
};
httpd_register_uri_handler(server, &file_download);

httpd_uri_t file_upload = {
    .uri = "/upload/*",
    .method = HTTP_POST,
    .handler = upload_post_handler,
    .user_ctx = server_data,
};
httpd_register_uri_handler(server, &file_upload);

httpd_uri_t file_delete = {
    .uri = "/delete/*",
    .method = HTTP_POST,
    .handler = delete_post_handler,
    .user_ctx = server_data
};
httpd_register_uri_handler(server, &file_delete);
return ESP_OK;
}
```