lujji

2017-01-30

HTTP server with WebSockets on ESP8266

This article will cover implementing a basic HTTP server on top of LwIP for ESP8266 and dive into the implementation of WebSockets.

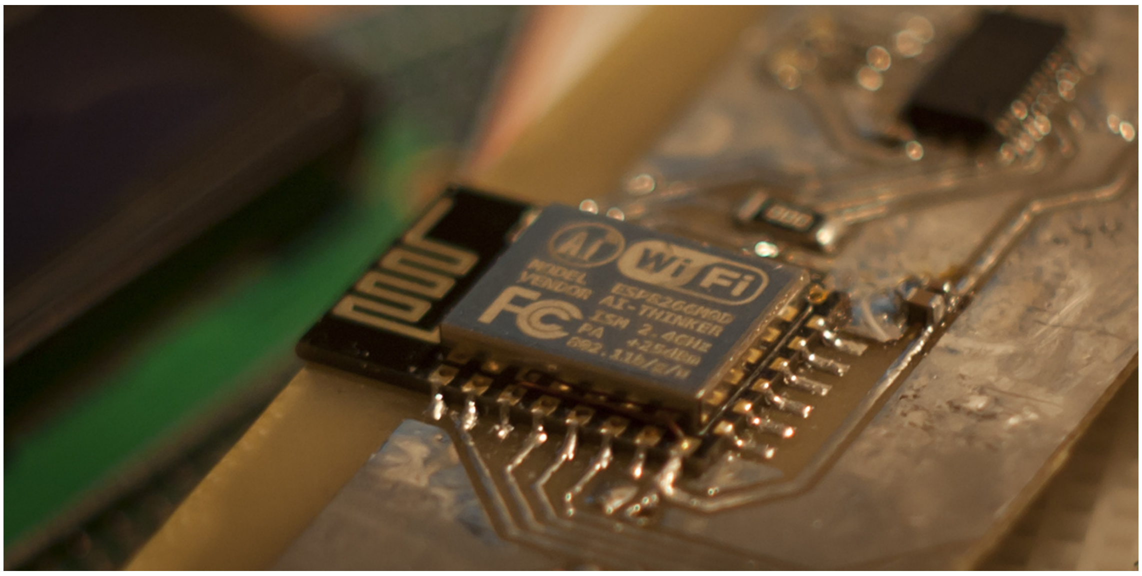
Preface

ESP8266 is an extremely popular device. Chances are, at some point you even bought a few modules for some “future project”. That’s exactly what I did, and for a long time I didn’t find any application for this device.

Contents:

The Hardware

At first glance, ESP8266 looks quite attractive: 32-bit processor, decent amount of RAM, up to 4MB external flash for user code. But once you dive into the specifics of the device, you immediately start facing it's problems: the documentation is rather scarce, power consumption is about 80mA during normal operation, which is a problem for battery-powered applications, and even though you can have a large external flash, the ESP8266 can only map 1 megabyte of it into execution space. The rest of the flash may be used for firmware updates and data storage. Overall, the device itself does not instill a lot of confidence. But hey.. it's cheap.



ESP8266

To get the module up and running we need a 3V3 supply rail and a UART-USB converter for programming. Apart from Rx/Tx the following lines need to be connected from the serial cable to the module:

```
RTS -> Reset
DTR -> Boot/GPIO0
3V3 -> CH_PD
```

ESP-12E modules already have a pull-up resistor on reset line. Optionally, a pull-up should be installed on `GPIO0` .

As for the software, there are two versions of SDK from Espressif - one of them is based on FreeRTOS and the other one is based on callbacks. It seems that most development occurs around the non-RTOS version of SDK. At the moment of writing this post, the latest FreeRTOS version provided by Espressif SDK seems to be 7.5.2, while the latest upstream version is 9.0.0. Luckily, esp-open-rtos addresses this issue. It is a community-developed framework based on the latest version of FreeRTOS, which aims to provide open-source alternatives to the binary blobs of the Espressif SDK.

Simple HTTP server

To get a better understanding of how things work, let's implement the most basic HTTP server. First we need to create a new task called `httpd_task`.

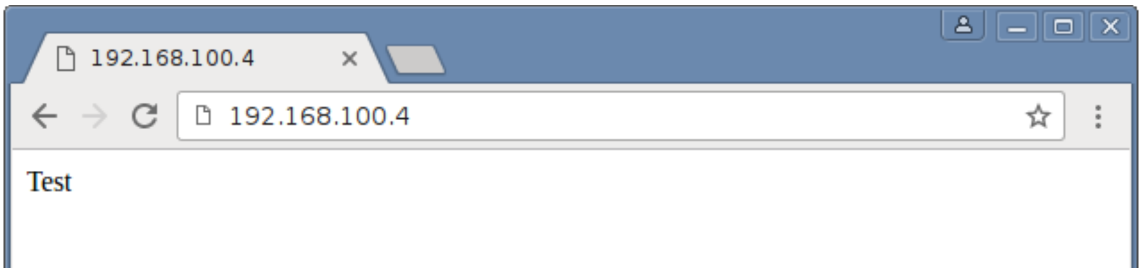
```
xTaskCreate(&httpd_task, "http_server", 1024, NULL, 2, NULL);
```

We are going to use LwIP's `netconn` API for our demo, `<lwip/api.h>` needs to be included.

```
void httpd_task(void *pvParameters)
{
    struct netconn *client = NULL;
    struct netconn *nc = netconn_new(NETCONN_TCP);
    if (nc == NULL) {
        printf("Failed to allocate socket\n");
        vTaskDelete(NULL);
    }
    netconn_bind(nc, IP_ADDR_ANY, 80);
    netconn_listen(nc);
    char buf[512];

    while (1) {
        err_t err = netconn_accept(nc, &client);
        if (err == ERR_OK) {
            struct netbuf *nb;
            if ((err = netconn_recv(client, &nb)) == ERR_OK) {
                void *data;
                u16_t len;
                netbuf_data(nb, &data, &len);
                printf("Received data:\n%.*s\n", len, (char*) d
                snprintf(buf, sizeof(buf),
                        "HTTP/1.1 200 OK\r\n"
                        "Content-type: text/html\r\n\r\n"
                        "Test");
                netconn_write(client, buf, strlen(buf), NETCONN
            }
```

```
                netbuf_delete(nb);
            }
            printf("Closing connection\n");
            netconn_close(client);
            netconn_delete(client);
        }
}
```

The code is pretty straight-forward: we create a new `netconn` , bind it to port 80 (which is used for HTTP) and start listening for incoming TCP connections. In the main loop of the task we call a blocking function `netconn_accept()` . Once the connection from client is accepted we log the request to console and generate a response. Response contains a minimal header that is enough for the browser to treat anything after `\r\n\r\n` as an HTML page.



Hello, world

When browser requests a page it sends a `GET` request, which looks like this:

```
GET / HTTP/1.1
Host: 192.168.100.4
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

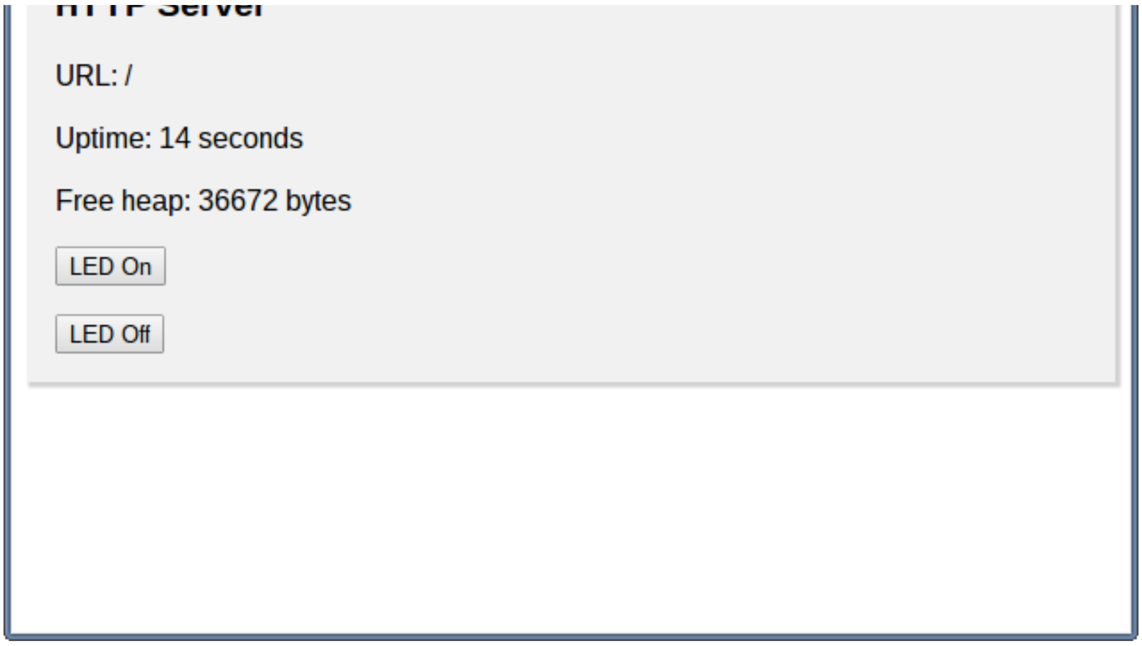
We’re only interested in the first line that contains the URI. To make things a bit more interesting we are going to extract the URI and switch the LED on the device when particular address is requested. We’ll also add some page content just for kicks.

```
void httpd_task(void *pvParameters)
{
    struct netconn *client = NULL;
    struct netconn *nc = netconn_new(NETCONN_TCP);
    if (nc == NULL) {
        printf("Failed to allocate socket.\n");
        vTaskDelete(NULL);
    }
    netconn_bind(nc, IP_ADDR_ANY, 80);
    netconn_listen(nc);
    char buf[512];
    const char *webpage = {
        "HTTP/1.1 200 OK\r\n"
        "Content-type: text/html\r\n\r\n"
        "<html><head><title>HTTP Server</title>"
        "<style> div.main {"
        "font-family: Arial;"
        "padding: 0.01em 16px;"
        "box-shadow: 2px 2px 1px 1px #d2d2d2;"
        "background-color: #f1f1f1;}"
        "</style></head>"
        "<body><div class='main'>"
        "<h3>HTTP Server</h3>"
        "<p>URL: %s</p>"
        "<p>Uptime: %d seconds</p>"
        "<p>Free heap: %d bytes</p>"
        "<button onclick=\"location.href='/on'\" type='button'>
        "LED On</button></p>"
        "<button onclick=\"location.href='/off'\" type='button'
        "LED Off</button></p>"
        "</div></body></html>"
    };
    /* disable LED */
    gpio_enable(2, GPIO_OUTPUT);
    gpio_write(2, true);

    while (1) {
        err_t err = netconn_accept(nc, &client);
        if (err == ERR_OK) {
            struct netbuf *nb;
            if ((err = netconn_recv(client, &nb)) == ERR_OK) {
                void *data;
                u16_t len;
                netbuf_data(nb, &data, &len);
```

```

                    /* check for a GET request */
                    if (!strncmp(data, "GET ", 4)) {
                        char uri[16];
                        const int max_uri_len = 16;
                        char *sp1, *sp2;

                        /* extract URI */
                        sp1 = data + 4;
                        sp2 = memchr(sp1, ' ', max_uri_len);
                        int len = sp2 - sp1;
                        memcpy(uri, sp1, len);
                        uri[len] = '\0';
                        printf("uri: %s\n", uri);

                        if (!strncmp(uri, "/on", max_uri_len))
                            gpio_write(2, false);
                        else if (!strncmp(uri, "/off", max_uri_len)
                            gpio_write(2, true);

                        snprintf(buf, sizeof(buf), webpage,
                                uri,
                                xTaskGetTickCount() * portTICK_PERI
                                (int) xPortGetFreeHeapSize());
                        netconn_write(client, buf, strlen(buf), NET
                    }
                }
                netbuf_delete(nb);
            }
            printf("Closing connection\n");
            netconn_close(client);
            netconn_delete(client);
        }
    }
```

Now we have a slightly more interactive server.

HTTP Server
192.168.100.4
HTTP Server



Simple HTTP server

In case your application needs to serve a simple web-page, this approach might just be good enough.

Although implementing an HTTP server from scratch could be a good exercise, I didn't find it very exciting, so instead of reinventing the wheel I decided to find one that is round enough for my needs.

For my application I decided to use httpd from LwIP/contrib. This server is based on callbacks, so it should work with RTOS and non-RTOS SDK.

WebSockets

WebSocket is a protocol which allows full-duplex communication between client (like web-browser) and server. This means that we can send small messages back and forth for doing things like toggling pins and reading sensor data without having to refresh the web-page and transfer large amounts of HTTP data all the time. We'll have to resort to HTTP only once for the opening handshake, after that all the communication is happening on the TCP layer. Everything we need to know in order to implement WebSocket protocol is described in RFC 6455.

Opening handshake

Probably the hardest part. When client wants to open a WebSocket it sends a specific GET request:

```
GET / HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
```

The server should generate the following response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

The procedure to generate `Sec-WebSocket-Accept` part is as follows:

1. Take the `Sec-WebSocket-Key` part
2. Concatenate it with GUID which is “258EAFA5-E914-47DA-95CA-C5AB0DC85B11”
3. Calculate SHA-1 hash of the resulting string
4. Encode the hash in base-64

Let’s first define some necessary constants inside `httpd.c` .

```
const char WS_HEADER[] = "Upgrade: websocket\r\n";
const char WS_GUID[] = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
const char WS_KEY[] = "Sec-WebSocket-Key: ";
const char WS_RSP[] = "HTTP/1.1 101 Switching Protocols\r\n" \
                      "Upgrade: websocket\r\n" \
                      "Connection: Upgrade\r\n" \
                      "Sec-WebSocket-Accept: %s\r\n\r\n";
```

According to the HTTP specification, comparison of fields like `WS_HEADER` should be case-insensitive. Despite that, we’ll use standard `strnstr()` function, since most browsers follow the convention and generate requests as defined above.

We’ll need to alter `http_parse_request()` function to support opening handshake. In this context `data` is the incoming TCP buffer.

```
if (strnstr(data, WS_HEADER, data_len)) {
    unsigned char encoded_key[32];
    char key[64];
    char *key_start = strnstr(data, WS_KEY, data_len);
    if (key_start) {
        key_start += 19;
        char *key_end = strnstr(key_start, "\r\n", data_len);
        if (key_end) {
            int len = sizeof(char) * (key_end - key_start);
            if (len + sizeof(WS_GUID) < sizeof(key) && len > 0)
                /* Concatenate key */
                memcpy(key, key_start, len);
                strlcpy(&key[len], WS_GUID, sizeof(key));
                printf("Resulting key: %s\n", key);

                /* Get SHA1 */
                unsigned char sha1sum[20];
                mbedtls_sha1((unsigned char *) key, sizeof(WS_G

                /* Base64 encode */
                unsigned int olen;
                mbedtls_base64_encode(NULL, 0, &olen, sha1sum,
                int ok = mbedtls_base64_encode(encoded_key, siz

                if (ok == 0) {
                    hs->is_websocket = 1;
                    encoded_key[olen] = '\0';
                    printf("Base64 encoded: %s\n", encoded_key)

                    /* Send response */
                    char buf[256];
                    u16_t len = snprintf(buf, sizeof(buf), WS_R
                    http_write(pcb, buf, &len, TCP_WRITE_FLAG_C
                    return ERR_OK;
                }
            } else {
                printf("Key overflow\n");
                return ERR_MEM;
            }
        }
    } else {
        printf("Malformed packet\n");
        return ERR_ARG;
    }
}
```

Note: I'm using `sizeof(buf)` quite often to get the array length at compile-time. In this case it works as expected due to the fact that `buf` is always of char type. A more proper solution is to use `sizeof(buf)/sizeof(buf[0])` - this way we get the correct result regardless of the data type.

Transmitting data

On the client side opening a new WebSocket and listening for incoming messages is just a matter of few lines of javascript:

```
/* Open new websocket and register callback */
ws = new WebSocket("ws://192.168.54.29");
ws.onmessage = function(evt) { onMessage(evt) };

function onMessage(evt) {
    console.log(evt.data);
}
```

When server receives data from a client the payload is *always* masked (assuming that client's implementation of the protocol is correct), therefore, we need to unmask the payload before passing it to the user callback. Masking algorithm is rather trivial.
The first byte of the payload contains an opcode. We're only going to support text or binary modes and close request. We shall omit continuation frames to keep things simple.

```
static err_t websocket_parse(struct tcp_pcb *pcb, struct pbuf *
{
    unsigned char *data;
    data = (unsigned char*) p->payload;
    u16_t data_len = p->len;

    if (data != NULL && data_len > 1) {
        uint8_t opcode = data[0] & 0x0F;
        switch (opcode) {
            case 0x01: // text
            case 0x02: // bin
                if (data_len > 6) {
                    data_len -= 6;
                    /* unmask */
                    for (int i = 0; i < data_len; i++)
                        data[i + 6] ^= data[2 + i % 4];
                    /* user callback */
                    websocket_cb(pcb, &data[6], data_len, opcod
                }
                break;
            case 0x08: // close
                return ERR_CLSD;
                break;
        }
        return ERR_OK;
    }
    return ERR_VAL;
}
```

When server sends data to the client it is always unmasked. Our implementation won't support packets larger than 125 bytes for simplicity.

```
void websocket_write(struct tcp_pcb *pcb, const uint8_t *data,
{
    if (len > 125)
        return;

    unsigned char buf[len + 2];
    buf[0] = 0x80 | mode;
    buf[1] = len;
    memcpy(&buf[2], data, len);
    len += 2;

    tcp_write(pcb, buf, len, TCP_WRITE_FLAG_COPY);
}
```

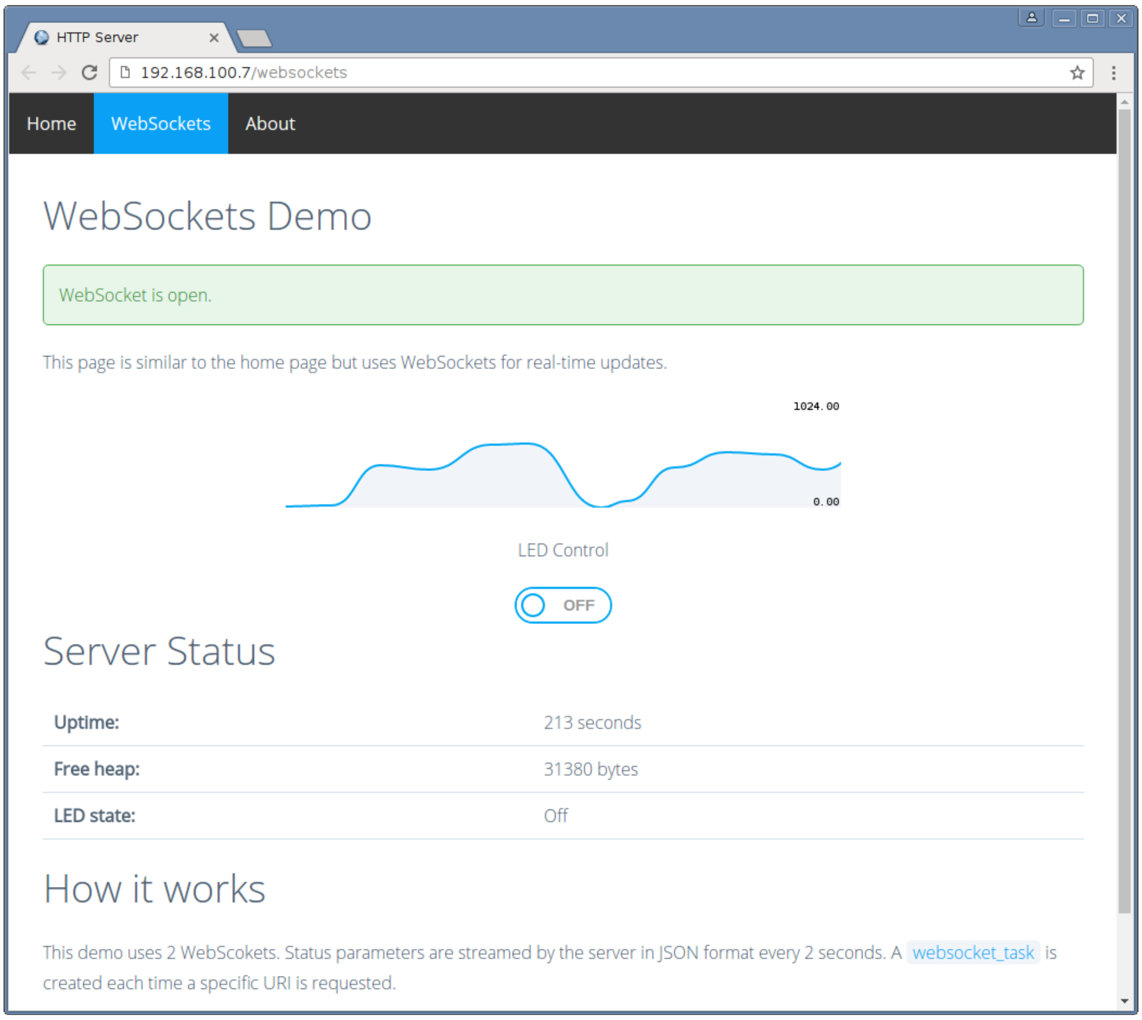
Closing connection

Simply closing a TCP connection is an option, but it's considered to be an *unclean shutdown*. When one side wants to close a websocket, it sends a packet which contains a reason for closing the connection. The other side then echoes this packet back and the connection is considered closed afterwards. Our implementation shall always close the connection with status code `1000` (normal closure).

```
static err_t websocket_close(struct tcp_pcb *pcb)
{
    const char buf[] = {0x88, 0x02, 0x03, 0xe8};
    u16_t len = sizeof(buf);
    return tcp_write(pcb, buf, len, TCP_WRITE_FLAG_COPY);
}
```

Demo

I created a small project to demonstrate basic functionality. In this demo two sockets are used: one for polling by the client, and second one for streaming data from server.



WebSockets Demo

WebSocket is open.

This page is similar to the home page but uses WebSockets for real-time updates.

LED Control

Server Status

Uptime:	213 seconds
Free heap:	31380 bytes
LED state:	Off

How it works

This demo uses 2 WebSockets. Status parameters are streamed by the server in JSON format every 2 seconds. A `websocket_task` is created each time a specific URI is requested.

WebSockets demo

Code is available on github.

#esp8266 #freertos #web

Comments Share

NEWER POSTS

Installing Black Magic via ST-Link bootloader

OLDER POSTS

Reverse-engineering ST-Link firmware - Part 2

10 Comments **lujji's blog** 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Gustavo Massaneiro • 9 months ago

Awesome job! Sure I'll use it on my project.
Thank you!

1 • Reply • Share ›

Andres Santos • 3 months ago

hello, great explanation, I implemented some of your lines into my project and got it working, I was wondering if you know how to get param data from a POST method

• Reply • Share ›

user3492 • 3 months ago

Hi, I want to implement also Websockets to the httpd Webserver. I used your code from the github, but it isn't working. I use raw api. The problem is that tcp_write() is not sending anything, so on the client side I get a timeout. Wenn I check the debugger, the content of the message would be right, but it's not sending out. At the end of http_parse_request(), when I let it at it was by the stack: return http_find_file(hs, uri, is_09);, then the Websocket connection is opening for a few moments, but a default webpage is send with the paket. If I change the end of the function and it looks like yours, then there is nothing been send. Any idea? Thank you

• Reply • Share ›

KALEESWAR ESWAR • 7 months ago

i need to initiate a Web socket client connection from LWIP TCP raw.What is the request header for the websocket client to connect to the websocket server.I am using embedded board to initiate a connection.

• Reply • Share ›

lujji Mod → KALEESWAR ESWAR • 7 months ago

Hi, does this answer your question?

• Reply • Share ›

ARCHIVES

August 2017
July 2017
April 2017
March 2017
February 2017
January 2017
October 2016
September 2016

RECENT POSTS

Serial bootloader for STM8
Mixing C and assembly on STM8
Executing code from RAM on STM8
Bare metal programming: STM8 (Part 2)
Bare metal programming: STM8