ESP8266

An introduction

Martin Heckel

July 8, 2021

Abstract

Micro Controller Units (MCUs) are used for quite some time to control devices in non-industrial scenarios (for industrial use cases, Programmable Logic Controllers (PLCs) are typically used). These MCUs are small computers equipped with a Central Processing Unit (CPU), memory, non-volatile data storage, and some physical interfaces often mapped to registers. Typically, these MCUs support multiple basic protocols for data transfer, such as Two-Wire Interface (TWI) (also known as Inter-Integrated Circuit (I^2C)) and Serial Peripheral Interface (SPI).

Those communication buses have one major disadvantage: They require the devices to be directly connected with multiple wires in order to communicate with each other. Another disadvantage is that those protocols are typically designed for communication between Integrated Circuits (ICs) on the same Printed Circuit Board (PCB). For this reason, they can often only handle short distances compared to other protocols.

In order to communicate with other devices using typical network or wireless protocols (like ethernet, Wireless Local Area Network (WLAN) or Bluetooth), additional hardware needs to be attached to those MCUs.

The ESP8266 is a MCU which contains the hardware to communicated over WLAN, so there is no requirement for additional hardware modules. Also, it has a faster CPU, more memory and more flash than a typical MCU. However, there is the drawback that is consumes more power as well. Because the ESP8266 is pretty cheap, it is the ideal hardware component for several network-related use cases of MCUs.

This semester paper provides an overview of the ESP8266 as well as different newer variants of that MCU. Afterwards, it is demonstrated that the ESP8266 can be easily programmed. There are some aspects of information security of setups with this kind of MCU explained. In the end, some use cases of the ESP8266 are described.

Contents

Contents

List of Figures ii

List of Tables iii

Abbreviations iv

Listings v

1. Introduction 1
- 1.1. ESP8266 . . . 1
- 1.2. Different variants of ESP MCUs . . . 2
- 1.3. Comparison: ESP vs Arduino . . . 2

2. Programming Examples 3
- 2.1. Basic GPIO access . . . 3
- 2.2. WLAN Station (Client) . . . 5
- 2.3. WLAN Access Point (AP) . . . 7
- 2.4. Web server . . . 7
- 2.5. Web client . . . 8

3. Aspects of information security 8

4. Project ideas 10
- 4.1. Typical Projects . . . 10
- 4.2. Deauthenticator-based Projects . . . 11

5. Conclusion 12

References 12

A. Listings used in the examples 15
- A.1. GPIO example . . . 15
- A.2. WLAN Station example . . . 15
- A.3. WLAN AP example . . . 18
- A.4. Web Server example . . . 19
- A.5. Web Client example . . . 23

List of Figures

1. ESP8266 in different form factors 1
2. Connection diagrams of the GPIO example 4
3. Mapping of General-Purpose Input/Output (GPIO) pins on the development board [20] 5
4. Connection diagrams of the WLAN Station example 6
5. Connection diagrams of the Web client example 8

List of Tables

1. Comparison of Arduino Uno, ESP8266 and ESP32 3

Abbreviations

AP	Access Point
API	Application Programming Interface
BLE	Bluetooth Low-Energy
CPU	Central Processing Unit
DoS	Denial of Service
DRAM	Dynamic Random-Access Memory
GPIO	General-Purpose Input/Output
HTTP	Hyper Text Transfer Protocol
I^{2}C	Inter-Integrated Circuit
IC	Integrated Circuit
IP	Internet Protocol
IPC	Inter-Process Communication
LED	Light-Emitting Diode
MCU	Micro Controller Unit
MitM	Man in the Middle
OS	Operating System
OTA	Over The Air
PCB	Printed Circuit Board
PLC	Programmable Logic Controller
QFN	Quad-Flat No-leads
RTOS	Real-Time Operating System
SDK	Software Development Kit
SMT	Surface-Mount Technology
SoC	System on a Chip
SPI	Serial Peripheral Interface
SSID	Service Set IDentifier
TCP	Transmission Control Protocol
THT	Through-Hole Technology
TWI	Two-Wire Interface
URI	Uniform Resource Identifier
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access

Listings

1. GPIO example 15
2. WLAN Station example 16
3. WLAN AP example 18
4. Web Server example 19
5. Web Client example 23

1. Introduction

In this section, the ESP8266 is described in more detail. It is compared to its successors like the ESP32. Afterwards, there is a short overview of the capabilities of the ATmega328P (which is used in the Arduino Uno), the ESP8266 and the ESP32.

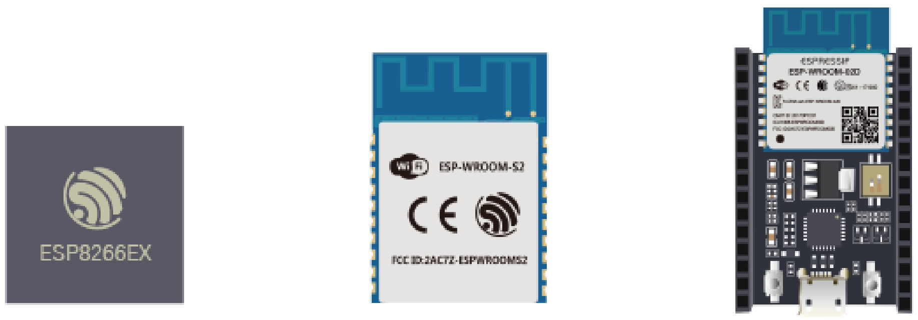
If not noted otherwise, the images in this semester paper were done by myself. I used Fritzing [9] to create the connection diagrams (breadboard views and schematics).

1.1. ESP8266

The ESP8266 is a WLAN-capable MCU produced by Espressif Systems [7]. It can be programmed with multiple programming languages like Arduino (similar to C++), Python, Lua, C, and some others.

Python, for example, requires an interpreter to be executed. Therefore, the ESP8266 has to be flashed with a firmware providing that python interpreter first (in case of python, the micropython firmware [18] is used). Afterwards, the python source code can be transferred and executed.

The ESP8266 is provided in different form factors as shown in Figure 1.



(a) System on a Chip (SoC) [13] (b) Module [12] (c) DevKit [11]

Figure 1: ESP8266 in different form factors

The SoC as shown in Subfigure 1a is shipped in a Quad-Flat No-leads (QFN) package which can soldered to PCBs using automatic soldering machines. Manual soldering is impossible or very hard. The use case for those SoC systems is typically a final product which is built in series production.

In contrast to that, the module as shown in Subfigure 1b provides a small PCB which can be soldered as well but has the soldering pads on the outside of the PCB, so it can be manually soldered and, therefore, used for prototype development.

The Development Kit (*DevKit*) provides a finished PCB with Universal Serial Bus (USB) support and already soldered pin headers. Because of this, no additional programmers are required to flash the

DevKit variant of the ESP8266. Because of the pin headers, it can be directly used in experimental schematics, e.g. on breadboards, without the requirement to solder anything. For these reasons, the DevKit is typically used for testing/experimenting in early development phases.

Typically, the programming (called *flashing*) is done via a serial interface. In case of the DevKit, this can be done directly via USB. For the other form factors, an additional programmer is required for that. While this is not a problem during development, it can be hard to program the devices or prototypes once they are built-in (e.g. a prototype of a sensor station on the roof). To solve this problem, the ESP8266 provides the possibility to do Over The Air (OTA) updates (via network). Thereby, the update can be done remotely without the necessity to physically reach and disassemble the prototype.

1.2. Different variants of ESP MCUs

The ESP8266 became known to a bigger community when the Software Development Kit (SDK) [6] was made publicly available by Espressif in 2014 [17]. Two years later, in 2016, the ESP32 was released. In contrast to the ESP8266, the ESP32 supports Bluetooth and Bluetooth Low-Energy (BLE).

In 2019, the ESP32-S2 was released. Just like the ESP8266, it does not provide support for Bluetooth and BLE. It consumes about $\frac{1}{3}$ of the energy of the ESP32. This is achieved by using a weaker CPU and less memory and flash. One year later, in 2020, the ESP32-C3 was released. It does, like the ESP32, support Bluetooth and BLE, but consumes only about $\frac{2}{3}$ of the energy the ESP32 consumes.

Note that these information are rough estimations based on the datasheets of the ESP32 [2], the ESP32-S2 [4] and the ESP32-C3 [3]. These estimations are necessary because those information were not measured under the same conditions in the datasheets (e.g. some datasheets specify required currents, others required powers).

1.3. Comparison: ESP vs Arduino

This section provides a short comparison of the Arduino Uno, the ESP8266 and the ESP32. Table 1 contains detailed information. Like before, the datasheets of the ESPs are pretty bad, so some of the values are estimations based on some other values in the according data sheets.

In comparison to the Arduino Uno, the ESP8266 and the ESP32 support WLAN 802.11 b/g/n. In difference to both other devices, the ESP32 supports Bluetooth 4.2 as well as BLE. All devices provide GPIO. The Arduino has a 8-bit CPU. In contrast to that, the CPU of the ESPs are 32-bit. The CPU of the ESP8266 is significantly (100 times) faster than the one on the Arduino. There are different configurations of the ESP32. Depending on the used configuration, the CPU is slightly or significantly faster than on the ESP8266 (factor 1.25 up to 3.75). Just like the CPU speed, the ESP8266 provides more memory capacity than the Arduino and the ESP32 provides more memory capacity

Feature	Arduino Uno [1]	ESP8266 [5]	ESP32 [2]
WLAN	✗	802.11 b/g/n	802.11 b/g/n
Bluetooth	✗	✗	BT 4.2 & BLE
GPIO	✓	✓	✓
CPU	8-bit, 16 MIPS with 16 MHz	32-bit, 160 MHz	32-bit, 200, 400, or 600 MIPS
Memory	2 KiB	160 KiB	520 KiB
Flash	32 KiB	up to 16 MiB external flash	up to 16 MiB external flash
Power consumption	4.5 mW at 4 MHz	264 mW average	500 mW average

Table 1: Comparison of Arduino Uno, ESP8266 and ESP32

than the ESP8266. Both ESPs support up to 16 MiB external flash (which is part of the modules). In contrast to that, the Arduino has 32 KiB of flash integrated. The increase of computing power has an effect on the power consumption as well. For this reason, the ESP32 consumes more power than the ESP8266 which consumes more power than the Arduino.

2. Programming Examples

In this section, some basic examples about programming the ESP8266 with the Real-Time Operating System (RTOS) SDK [6] are provided. The complete source code of the examples can be found in the appendix and it is suggested to look at it while reading this section.

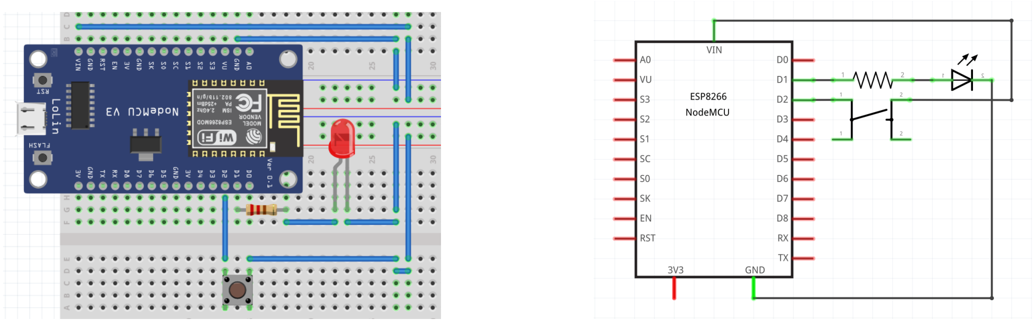
2.1. Basic GPIO access

This example demonstrates the basic principles of accessing the GPIO pins of the ESP8266. In detail, the pins are initialized first. Afterwards, the voltage level at one of the pins is measured. Depending on the measured value, another pin is enabled and disabled in a blinking pattern.

A Light-Emitting Diode (LED) is connected to the pin that is blinking and a switch to the pin that is read. Figure 2 depicts the breadboard and schematic diagrams. The source code referenced in this example is shown in Listing 1 in the appendix.

Before the GPIO pins can be used, they have to be initialized. Therefore, the SDK provides the data structure `gpio_config_t`. The configuration is written to an instance of that data structure and written with the `gpio_config()` function afterwards.

It is important to note that there has to be a defined voltage level at the input pins when a measurement takes place. As shown in Figure 2, there is no defined voltage level at the pin D2 when



(a) Breadboard (b) Schematic

Figure 2: Connection diagrams of the GPIO example

the switch is open. For this reason, the value measured at pin D2 would toggle between *High* and *Low* when the switch is not closed.

This is problematic because a measured level of *High* cold mean that the switch is closed or that the switch is opened. A measured level of *Low* would always mean the switch is opened. To avoid those fluctuations, there is an approach to add a resistor between the pin that is measured (in this case, D2) and some defined level (GND for *Low* and VIN for *High*).

Assume a resistor of a size of 10 kΩ is added between D2 and GND. If the switch is closed, the resistance to GND would be 10 kΩ, the resistance to VIN would be theoretically zero (ignoring the wire resistance). If the switch is opened, the resistance to GND would still be 10 kΩ, the resistance to VIN would be theoretically infinitely high.

The level measured at the pin can be calculated using the 10 kΩ resistor R that was added, the resistance of the switch R_{Switch} (0Ω or $\infty\Omega$) and the voltage of the development board U (5 V):

$$U_{D2} = \frac{U_{VIN} \times R}{R + R_{Switch}}$$

When the switch is closed, the voltage at pin D2 is:

$$U_{D2} = \frac{U_{VIN} \times R}{R + R_{Switch}} = \frac{5\,\text{V} \times 10\,\text{k}\Omega}{10\,\text{k}\Omega + 0\Omega} = 5\,\text{V}$$

When the switch is opened, the voltage at pin D2 is:

$$U_{D2} = \frac{U_{VIN} \times R}{R + R_{Switch}} = \frac{5\,\text{V} \times 10\,\text{k}\Omega}{10\,\text{k}\Omega + \infty\Omega} = 0\,\text{V}$$

Those resistors are called pull-down resistors when they connect the measured pin with GND and pull-up resistors when the connect the measured pin with VIN. The ESP8266 has internal pull-up and pull-down resistors that can be enabled in the configuration process described before. For that reason, it is not necessary to connect them on the breadboard.

The resistor shown in Figure 2 is required to limit the current floating through the LED and thereby preventing the LED to get too much power as well as the ESP8266 to submit too much power. In the figure, it is rather symbolic and the color code of the rings does not represent the actual resistance that should be selected depending on the operation voltage and used LED. However, typically a resistance of 1 kΩ can be used for LEDs when the operation volage is 5 V. The same applies on the value of the resistors before the LEDs in the other figures as well.

When the configuration is done, the GPIO pins can be accessed with the `gpio_get_level()` function to read the voltage level at a pin configured as input and the `gpio_set_level()` function to set the voltage level at a pin configured as output.

The function `ESP_LOGI()` takes a tag and a message and writes them to the serial log which can be accessed by connecting the ESP8266 to a computer and connecting to the serial console of the device. The *I* stands for the log level *Information*. There are other log levels, as *Error*, *Warning*, and *Debug* as well which can be used by calling the corresponding log function (e.g. `ESP_LOGE()`, `ESP_LOGW()`, and `ESP_LOGD()`).

Mind that the numbers used in the GPIO-related functions are the numbers of the GPIO pins and not the numbers of the pins on the development board. Figure 3 depicts the mapping of GPIO pins to the pins of the development board.

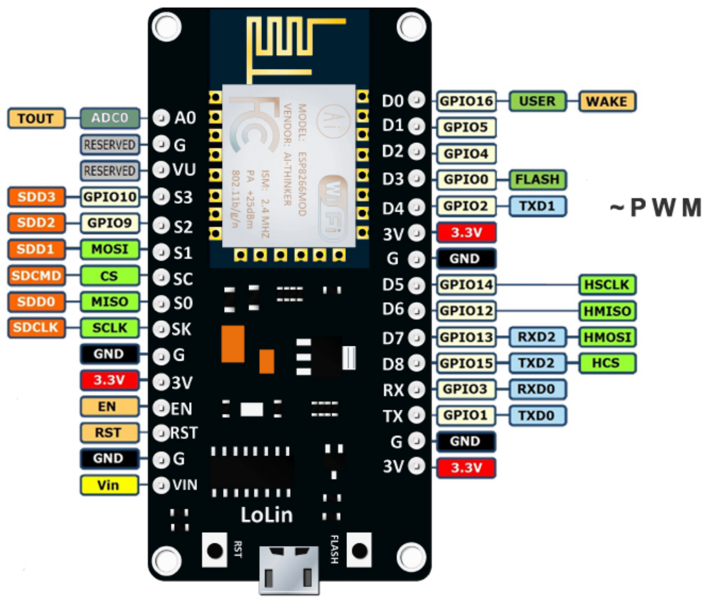
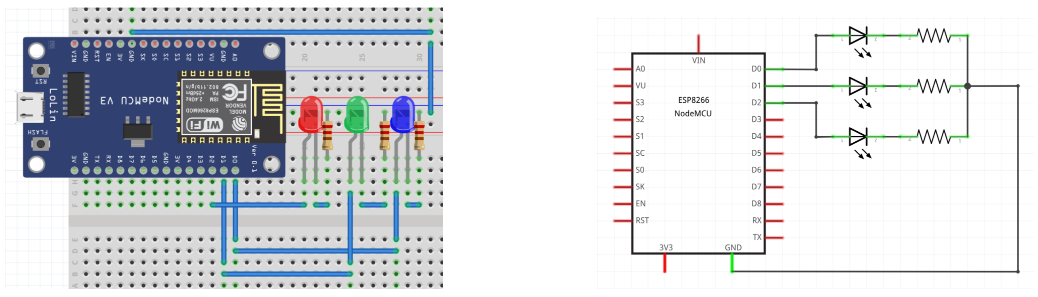


Figure 3: Mapping of GPIO pins on the development board [20]

2.2. WLAN Station (Client)

Figure 4 depicts the schematic used in this example. Listing 2 in the appendix depicts the source code referenced in this section.



(a) Breadboard (b) Schematic

Figure 4: Connection diagrams of the WLAN Station example

As in the example described in the section before, there are some steps for initialization needed first. For the WLAN station, the Transmission Control Protocol (TCP)/Internet Protocol (IP) stack has to be initialized first. This can be done with the function `tcpip_adapter_init()`.

In contrast to the last example, an event loop is used in this one. In the last example, there was a loop checking for a specific condition. If the condition occurred, something was done. In this example, the event loops provided by the RTOS SDK are used for that. The event loop can be created with the function `esp_event_loop_create_default()`.

Afterwards, the physical configuration (e.g. size of the buffers that should be used) of the WLAN interface is set with the function `esp_wifi_init()`.

An event has two identification numbers: One specifies the event type (e.g. IP event, WiFi event) and the other one specifies the exact event that happened within this type (e.g. station got IP). A handler that should be called when an event occurred can be registered. It is possible to submit an additional parameter to the handler function, which should be set to `NULL` if it is not needed. In the example, the handler function `handler` is assigned to all WiFi events and the IP event *station got IP*.

After this, the mode of the WLAN interface is set to station with the function `esp_wifi_set_mode()`. Next, the logical configuration (e.g. Service Set IDentifier (SSID) and passphrase) of the WLAN interface is set with the function `esp_wifi_set_config()`.

The last step is to start the WLAN interface using the function `esp_wifi_start()`.

The handler function that was registered before takes four parameters: an optional argument (the parameter that was set to `NULL` before), the event type, the concrete event, and additional data. Those data is actually a pointer to a data structure depending on the event that occurred. Based on the event, the data structure that parameter is pointing to can be cast to the correct structure. Thereby, the handler can access additional information about the event. The handler function contains conditional branches for three cases: The ESP8266 WLAN station mode was started, the connection to the specified WLAN network failed, or the connection to the specified network was successful.

2.3. WLAN AP

There are no additional components used in this example. For this reason, there is no figure showing the breadboard and schematic view. The source code referenced in this example is shown in Listing 3 in the appendix.

Basically, the programming of a WLAN AP is similar to the programming of a WLAN Station: First, the TCP/IP stack is initialized with the function `tcpip_adapter_init()`. Afterwards, the event loop is created with `esp_event_loop_create_default()` and the physical configuration of the WLAN interface is set with the function `esp_wifi_init()`.

Next,the event handler is registered for any WLAN event. Afterwards, the mode of the WLAN interface is set to AP with the function `esp_wifi_set_mode()` and then, the logical configuration is applied with the function `esp_wifi_set_config()`.

The last step is to start the WLAN interface using the function `esp_wifi_start()`.

2.4. Web server

The web server example is based on the WLAN station example described in Section 2.2. Therefore, the schematic used in the example is the same. Figure 4 depicts that schematic. The source code is shown in Listing 4 in the appendix.

In the beginning, the ESP8266 connects to the WLAN using the SSID and passphrase specified. If the connection was successful and the ESP8266 got an IP, the web server is started. If that was successful, a handler function is registered for a Uniform Resource Identifier (URI) specified. In this example, the handler function is called for the URI `/set` when the Hyper Text Transfer Protocol (HTTP) method GET is used.

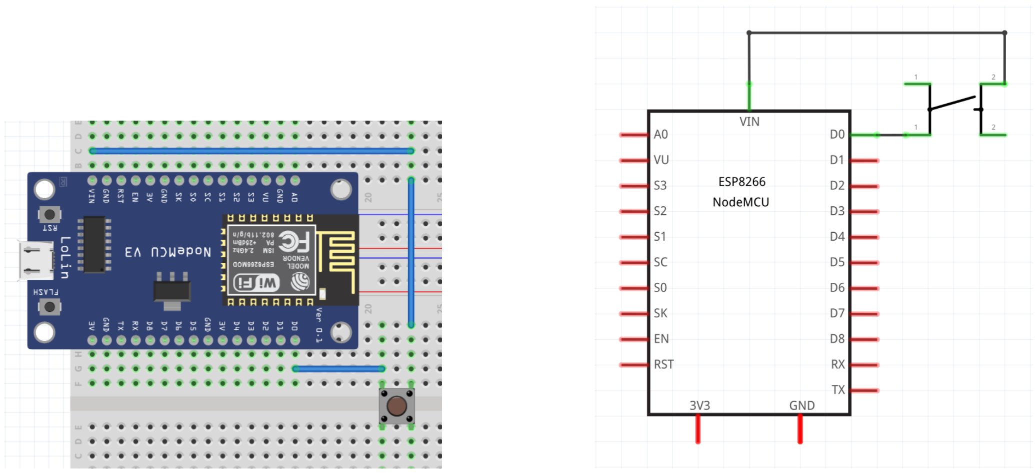
If the connection to the WLAN is lost or can not be established, the web server is stopped.

In the request handler, the full request URI is copied into a buffer first using the function `httpd_req_get_url_query_str()`. Next, the value of the query attribute “led” is stored in a buffer using `httpd_query_key_value()`. After that, the value of the attribute is compared to be “on”, “off” or “toggle”. Depending on its value, the LED connected to GPIO pin 16 is turned on or off.

Afterwards, the buffer is freed and depending on the success of switching the state a message is returned in the HTTP response.

2.5. Web client

The schematic used in this example is depicted in Figure 5. The source code is shown in Listing 5 in the appendix.



(a) Breadboard (b) Schematic

Figure 5: Connection diagrams of the Web client example

In the beginning, the web client connects to the WLAN AP as described in Section 2.3. Then, the state of the switch is checked in an endless loop. If the switch is pressed, the client sends requests to the server described in Section 2.4 in order to let the LED blink three times by turning it on and off alternating and wait 500 ms in between.

Like described in Section 2.1, there has to be a short pause if the switch is not pressed to avoid the ESP8266 being restarted by the watchdog. Therefore, a delay of 10 ms is done in that case.

The request is sent manually by establishing a TCP connection with the web server at port 80 and sending the raw http request using that connection. The request is crafted by replacing the value of the “led” GET parameter with the one submitted as function parameter. The rest of the HTTP request stays the same and, therefore, is just written in a static string.

3. Aspects of information security

Current information systems are so complex that they mostly consist of multiple sub-systems which are built on to of each other: Typically, an application running on such a system depends on the hardware, the Operating System (OS), libraries, runtime components (e.g. dynamically linked components or external web services), etc.

In recent years, researchers found many hardware vulnerabilities [14, 10, 15, 16, 22, 21, 24, 19, 8, 23] in computer systems which showed that the hardware that was believed to be secure and work correctly is far away from that state: The research showed, among others, that is was possible to modify memory without actually writing (Rowhammer), to read memory without actually reading (Meltdown, Spectre), to communicate with other processes without using any Inter-Process Communication (IPC) mechanisms of the OS (DRAMA), etc.

Typically, that research is done on often used systems (e.g. DDR4 Dynamic Random-Access Memory (DRAM), x86 CPUs, etc.) Even though the ESP8266 is less complex than a current x86 CPU, it might be still affected by hardware vulnerabilities which were not found yet.

There might be bugs in the SDKs and standard libraries which were not found until now. There might be bugs in own implementations running on the ESP8266. For that reasons, it is recommended to perform updates of the software running on that devices on a regular basis.

The main security problem with the ESP8266 is their remote accessibility: Due to the ability to connect to an existing WLAN, they can basically be connected directly to the internet. Thereby, depending on the exact configuration, an attacker might be able to access them from a remote location. This increases the attack surface massively because it is possible to automatically scan for those devices and attack them (e.g. when there is a known vulnerability in the SDK).

A basic approach to reduce that risk is to reduce the attack surface. That can be done by putting all ESP8266 and likely devices to an own WLAN which does not have direct access to the internet. Access to those devices can be only established using a proxy which allows, of course, only the operations required for the system to work. The more restrictive that proxy is configured the smaller is the attack surface of the system.

Another important thing is to be aware of the risks. Think what an attacker could do in the worst case with that system. Information security has three possible targets to protect: Confidentially, Integrity, and Availability. What would, in the concrete use case, happen if an attacker could break Confidentially and read data on the system. What about Integrity and writing data or Availability and turning off or breaking the ESP8266.

If there is awareness of that risks, one can evaluate if it is required to do some additional measures to lower them. For example, if an ESP8266 is used as smart light switch, an additional normal light switch might be a good solution as fallback. If anything happens to the ESP8266 (if it is attacked, or simply fails) it would be still possible to control the light manually.

4. Project ideas

In Section 2, some basic examples of programming the ESP8266 were shown. In this section, there is a short overview about some projects that could be implemented with the ESP8266 (or another Bluetooth capable variant if thats required).

4.1. Typical Projects

- **Web-controllable LED strip**
  There are LED strips where each LED is addressed and, thereby, the LEDs are controllable individually. Typically, those strips have two connectors for power supply (Vcc and GND) and one connector for data input. The single LEDs contain chips that implement a protocol for data transmission and control the according LED.

  Typically, there are libraries implementing that protocols for MCUs like the Arduino or ESP8266. As described before, one of the advantages of the ESP8266 over the Arduino is that it supports WLAN by default. For that reason, the ESP8266 can be used to control the LED strip based on a web-based user interface or Application Programming Interface (API) provided by the ESP8266.

  Those addressable LEDs are not only available in strips but can also be bought in Surface-Mount Technology (SMT) or Through-Hole Technology (THT) packages and, thereby, soldered in own projects. This makes it possible to arrange the LEDs in other form factors, like matrices or cubes as well.

- **Gesture-based controller for games**
  In the scope of the bachelors program, we (a group of four students) created a 3D-Snake game as practical work in the *Internet of Things* lecture. In that project we used the approach described before to create a 3D cube with $8 \times 8 \times 8$ WS2812-based LEDs. In difference to the example described before, the cube was controlled by a raspberry pi.

  In order to control the snakes (it was a game for two players), we built two controllers which detected gestures based on light barriers. If a gesture (“up”, “down”, “left”, “right”, “forwards”, “backwards”) is detected, the controller sent the gesture detected to the raspberry pi using a serial Bluetooth-based protocol we designed for that project.

  Those controllers had an ESP32 that contained the logic to detect the gestures and send the detected gestures to the raspberry pi.

- **Controller for gutter heating**
  At home, we have a gutter heating system which measures moisture and temperature and enables the heater based on that parameters. That avoids icicles to grow on the gutter. The

system we have has one problem: the sensor for moisture contains a small heater in order to melt snow if it is cold. Then, it measures if there is water between two contacts based on resistance.

However, when there is much snow, the sensor just melts a small tunnel under the snow and measures “dry” afterwards, independent of the actual condition. I connected an ESP8266 to the control device of the heating system which can overwrite the actual sensor values using transistors that connect resistors to the control device instead of the actual sensors.

Thereby, the conditions “wet” and “cold” can be manually overwritten with the ESP8266 leading to the user being able to enable or disable the heating system manually. The ES8266 provides a web interface to do this remote as well. As described in Section 3, the ESP8266 is in a separate WLAN which can be accessed only over a proxy. Before the proxy allows access, the user has to authenticate at the proxy.

4.2. Deauthenticator-based Projects

- **WLAN deauthenticator**
  The IEEE802.11 standard (WLAN) specifies a deauthentication mechanism using *deauthentication frames*. If such a frame is sent to an access point, it stops the communication with a station and the station has to connect again in order to communicate with the access point. That deauthentication frame does not have to be sent encrypted, even when the station connected to the network using encryption.

  Thereby, an attacker can spoof those deauthentication frames and disconnect stations from a wireless network.

  This approach can be used to deauthenticate some or all clients of a WLAN over and over again, effectively leading to a Denial of Service (DoS).

  The attack works as long as the device sending the deauthentication frames is physically near the AP. For that reason, the ESP8266 can be comfortably used because it does not consume much energy and is relatively cheap. So, an attacker can get a battery, connect it to the ESP8266 and leave it near the AP (fire-and-forget) which would result in the attacked WLAN to be effectively unusable.

- **WPA handshake sniffer** When a device connects to a WLAN which is using Wi-Fi Protected Access (WPA) or WPA2 as authentication mechanism, there is a 4-way handshake done by every device that authenticates to the WLAN. An attacker can sniff those handshakes and mount a password attack (brute-force or directory-based) on that data.

In order to get more WPA handshakes, the attacker can deauthenticate stations from the WLAN forcing them to authenticate again and, thereby, perform the 4-way handshake which the attacker can sniff.

- **WLAN reconnect to evil twin** An attacker can operate an AP with the same configuration than a legitimate AP. The attackers AP is called *evil twin* in that case. If a victim wants to connect to a WLAN network and there is an evil twin with a stronger signal, it will connect to that evil twin. Note that "same configuration" means that the attacker has to know the passphrase of the victims WLAN as well.

  Now, the victim is connected to an AP controlled by the attacker which can be used to mount Man in the Middle (MitM) attacks.

  Again, the deauthenticator can be used to force the victims station to disconnect from the legitimate WLAN first. When it is disconnected, it will try to connect again using the attackers AP when the signal is stronger.

5. Conclusion

The ESP8266 is a small programmable device which is capable to connect to WLAN. Some variants (e.g. the ESP32) are capable to communicate over Bluetooth and BLE as well. Those devices can be used as replacements for other MCU-based systems, like Arduino: They have more connectivity and more computing power.

Because there are multiple libraries/firmwares in for different programming languages, the ESP8266 can be programmed in the programming language preferred by the developer. The OTA update feature makes it easy to keep the software on the devices up-to-date.

Although the documentation from the vendor is not satisfying (documentation style changed between different datasheets, technical information are not complete, many non-technical information in technical datasheets), there are many 3rd party resources for the original SDK as well as other SDKs which are much better documented.

The most important thing is to keep the risks of remote controllable hardware in mind. Before using an ESP8266, one should thing about possible implications when an attacker is able to impact Confidentiality, Integrity and Availability. After this consideration is done, the ESP8266 can be a good and powerful device for several projects.

References

[1] *Datasheet ATmega328P*. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (visited on 06/17/2021).

[2] *Datasheet ESP32*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (visited on 06/17/2021).

[3] *Datasheet ESP32*. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf (visited on 06/17/2021).

[4] *Datasheet ESP32-S2*. URL: https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf (visited on 06/17/2021).

[5] *Datasheet ESP8266*. URL: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (visited on 06/17/2021).

[6] *ESP8266_RTOS_SDK repository*. URL: https://github.com/espressif/ESP8266_RTOS_SDK (visited on 06/17/2021).

[7] *Espressif Systems*. URL: https://www.espressif.com/ (visited on 06/17/2021).

[8] Pietro Frigo et al. "TRRespass: Exploiting the Many Sides of Target Row Refresh". In: *Proceedings of the IEEE Security & Privacy*. May 2020. URL: https://download.vusec.net/papers/trrespass_sp20.pdf.

[9] *Fritzing (Software)*. URL: https://fritzing.org/ (visited on 06/19/2021).

[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[11] *Image of ESP8266 DevKit*. URL: https://www.espressif.com/sites/default/files/dev-board/ESP-WROOM-02D%20.png (visited on 06/17/2021).

[12] *Image of ESP8266 Module*. URL: https://www.espressif.com/sites/default/files/modules/ESP-WROOM-S2_1.png (visited on 06/17/2021).

[13] *Image of ESP8266 SoC*. URL: https://www.espressif.com/sites/default/files/chips/ESP8266EX%20%E5%B0%8F.png (visited on 06/17/2021).

[14] Yoongu Kim et al. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *ACM SIGARCH Computuer Architure News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964. DOI: 10.1145/2678373.2665726.

[15] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P 19)*. 2019.

[16] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[17] *Metadata of ESP8266_RTOS_SDK repository*. URL: https://api.github.com/repos/espressif/ESP8266_RTOS_SDK (visited on 06/17/2021).

[18] *MicroPython*. URL: `https://micropython.org/` (visited on 06/17/2021).

[19] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *41st IEEE Symposium on Security and Privacy (S&P’20)*. 2020.

[20] *NodeMCUv3.0-pinout.jpg*. URL: `https://www.teachmemicro.com/wp-content/uploads/2018/04/NodeMCUv3.0-pinout.jpg` (visited on 06/20/2021).

[21] Peter Pessl et al. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016, pp. 565–581.

[22] Kaveh Razavi et al. “Flip Feng Shui: Hammering a Needle in the Software Stack”. In: *Proceedings of the 25th USENIX Security Symposium*. June 2016. URL: `https://download.vusec.net/papers/flip-feng-shui_sec16.pdf`.

[23] Finn de Ridder et al. “SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript”. In: *Proceedings of the 30th USENIX Security Symposium*. Aug. 2021. URL: `https://download.vusec.net/papers/smash_sec21.pdf`.

[24] Victor van der Veen et al. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *Proceedings of the 23rd ACM Conference on Computer and Communication Security (CCS)*. Oct. 2016. URL: `https://vvdveen.com/publications/drammer.pdf`.

A. Listings used in the examples

A.1. GPIO example

```c
#include "driver/gpio.h"
#include "esp_log.h"
#include "arch/sys_arch.h"

#define GPIO_OUT 5
#define GPIO_IN 4

void setupGPIO(void) {
    gpio_config_t config;
    config.pin_bit_mask = (1<<GPIO_OUT);
    config.mode = GPIO_MODE_OUTPUT;
    config.pull_up_en = GPIO_PULLUP_DISABLE;

    gpio_config(&config);

    config.pin_bit_mask = (1<<GPIO_IN);
    config.mode = GPIO_MODE_INPUT;
    config.pull_up_en = GPIO_PULLDOWN_ENABLE;

    gpio_config(&config);
}

void app_main(void) {
    setupGPIO();

    ESP_LOGI("main", "Started.\n");

    while(1) {
        if(gpio_get_level(GPIO_IN) == 0) {
            ESP_LOGI("main", "Blinking.");
            gpio_set_level(GPIO_OUT, 1);
            sys_delay_ms(500);
            gpio_set_level(GPIO_OUT, 0);
            sys_delay_ms(500);
        } else {
            sys_delay_ms(10);
        }
    }
}
```

Listing 1: GPIO example

A.2. WLAN Station example

```c
#include<string.h>
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "lwip/err.h"
#include "lwip/sys.h"
#include "driver/gpio.h"
#include "arch/sys_arch.h"

#define SSID "myssid"
#define PASSWORD "mypassword"

#define LED1 16
#define LED2 5
#define LED3 4


void setupGPIO(void) {
    gpio_config_t config;
    config.pin_bit_mask = (1UL<<LED1)|(1UL<<LED2)|(1UL<<LED3);
    config.mode = GPIO_MODE_OUTPUT;
    config.pull_up_en = GPIO_PULLUP_DISABLE;
    config.pull_down_en = GPIO_PULLDOWN_DISABLE;
    config.intr_type = GPIO_INTR_DISABLE;

    gpio_config(&config);

    gpio_set_level(LED1, 0);
    gpio_set_level(LED2, 1);
    gpio_set_level(LED3, 0);
}

void blink(int pin, int iter, int endState) {
    for(int i = 0; i < iter; i++) {
        gpio_set_level(pin, 0);
        sys_delay_ms(100);
        gpio_set_level(pin, 1);
        sys_delay_ms(100);
    }
    gpio_set_level(pin, endState);
}

void eventHandler(void *arg, esp_event_base_t eventBase, int32_t eventId, void *
    data) {
    if(eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_START) {
        ESP_LOGI("WiFi", "Connecting to AP...");
        gpio_set_level(LED2, 0);
        blink(LED3, 2, 1);
        esp_wifi_connect();
    } else if (eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_DISCONNECTED)
         {
```

```c
        wifi_event_sta_disconnected_t *event = (wifi_event_sta_disconnected_t*)
            data;
        ESP_LOGI("WiFi", "Disconnected from AP %02x:%02x:%02x:%02x:%02x:%02x
            Reason: %d. Reconnecting...", event->bssid[0], event->bssid[1], event
            ->bssid[2], event->bssid[3], event->bssid[4], event->bssid[5], event
            ->reason);
        gpio_set_level(LED2, 0);
        blink(LED3, 2, 1);
        esp_wifi_connect();
    } else if(eventBase == IP_EVENT && eventId == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t *event = (ip_event_got_ip_t *)data;
        ESP_LOGI("WiFi", "Connected to AP. IP=%s", ip4addr_ntoa(&event->ip_info.
            ip));
        gpio_set_level(LED3, 0);
        blink(LED2, 10, 1);
    }
}

void setIntArr(uint8_t *arr, const char *value) {
    int i = 0;
    for(; value[i] != 0; i++) {
        arr[i] = (uint8_t)(value[i]);
    }
    arr[i] = 0;
}

void initWifi() {
    tcpip_adapter_init();
    esp_event_loop_create_default();

    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init((const wifi_init_config_t *)(&config)));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &
        eventHandler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &
        eventHandler, NULL));

    wifi_config_t wifiConfig;
    setIntArr(wifiConfig.sta.ssid, SSID);
    setIntArr(wifiConfig.sta.password, PASSWORD);
    wifiConfig.sta.threshold.authmode = WIFI_AUTH_WPA2_PSK;

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifiConfig));
    ESP_ERROR_CHECK(esp_wifi_start());
}

void app_main(void) {
    setupGPIO();
    initWifi();
}
```

Listing 2: WLAN Station example

A.3. WLAN AP example

```c
#include<string.h>
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "lwip/err.h"
#include "lwip/sys.h"

#define SSID "myssid"
#define PASSWORD "mypassword"

void eventHandler(void *arg, esp_event_base_t eventBase, int32_t eventId, void *
    data) {
    if(eventId == WIFI_EVENT_AP_STACONNECTED) {
        wifi_event_ap_staconnected_t *event = (wifi_event_ap_staconnected_t*)
            data;
        ESP_LOGI("WiFi", "station %02x:%02x:%02x:%02x:%02x:%02x connected.",
            MAC2STR(event->mac));

    } else if (eventId == WIFI_EVENT_AP_STADISCONNECTED) {
        wifi_event_ap_stadisconnected_t *event = (wifi_event_ap_stadisconnected_t
            *) data;
        ESP_LOGI("WiFi", "station %02x:%02x:%02x:%02x:%02x:%02x disconnected.",
            MAC2STR(event->mac));
    }
}

void setIntArr(uint8_t *arr, const char *value) {
    int i = 0;
    for(; value[i] != 0; i++) {
        arr[i] = (uint8_t)(value[i]);
    }
    arr[i] = 0;
}

void initWifi() {
    tcpip_adapter_init();
    esp_event_loop_create_default();

    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init((const wifi_init_config_t *)(&config));

    esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &eventHandler, NULL)
        ;

    wifi_config_t wifiConfig;
    setIntArr(wifiConfig.ap.ssid, SSID);
    wifiConfig.ap.ssid_len = strlen(SSID);
    setIntArr(wifiConfig.ap.password, PASSWORD);
    wifiConfig.ap.max_connection = 4;
    wifiConfig.ap.beacon_interval = 100;
    wifiConfig.ap.channel = 11;
    wifiConfig.ap.authmode = WIFI_AUTH_WPA_WPA2_PSK;

    esp_wifi_set_mode(WIFI_MODE_AP);
    esp_wifi_set_config(ESP_IF_WIFI_AP, &wifiConfig);
    esp_wifi_start();
}

void app_main(void) {
    initWifi();
}
```

Listing 3: WLAN AP example

A.4. Web Server example

```c
#include<string.h>
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "lwip/err.h"
#include "lwip/sys.h"
#include "driver/gpio.h"
#include "arch/sys_arch.h"
#include "esp_http_server.h"

#define SSID "myssid"
#define PASSWORD "mypassword"

#define LED1 16
#define LED2 5
#define LED3 4

int LEDState = 0;


void setupGPIO(void) {
    gpio_config_t config;
    config.pin_bit_mask = (1UL<<LED1)|(1UL<<LED2)|(1UL<<LED3);
    config.mode = GPIO_MODE_OUTPUT;
    config.pull_up_en = GPIO_PULLUP_DISABLE;
    config.pull_down_en = GPIO_PULLDOWN_DISABLE;
    config.intr_type = GPIO_INTR_DISABLE;

    gpio_config(&config);

    gpio_set_level(LED1, 0);
    gpio_set_level(LED2, 1);
    gpio_set_level(LED3, 0);
}

void blink(int pin, int iter, int endState) {
    for(int i = 0; i < iter; i++) {
        gpio_set_level(pin, 0);
        sys_delay_ms(100);
        gpio_set_level(pin, 1);
        sys_delay_ms(100);
    }
    gpio_set_level(pin, endState);
}

esp_err_t getHandler(httpd_req_t *req) {
    ESP_LOGI("HTTP", "received request");
    char *buf;
    size_t bufSize;
    int success = 0;

    bufSize = httpd_req_get_url_query_len(req) + 1;
    if(bufSize > 1) {
        buf = malloc(bufSize);
        if(httpd_req_get_url_query_str(req, buf, bufSize) == ESP_OK) {
            char value[32];
            value[0] = 0;
            if(httpd_query_key_value(buf, "led", value, sizeof(value)) == ESP_OK)
                 {
                if(strcmp(value, "off") == 0) {
                    gpio_set_level(LED1, 0);
                    LEDState = 0;
                    success = 1;
                } else if(strcmp(value, "on") == 0) {
                    gpio_set_level(LED1, 1);
                    LEDState = 1;
                    success = 1;
                } else if(strcmp(value, "toggle") == 0) {
                    LEDState ^= 1UL;
                    gpio_set_level(LED1, LEDState);
                    success = 1;
                }
            }
        }
        free(buf);
    }

    if(success == 1) {
        buf = "The LED state was set successfully.";
    } else {
        buf = "Unable to set the LED state.";
    }

    httpd_resp_send(req, buf, strlen(buf));

    if(success == 0) {
        blink(LED1, 10, LEDState);
    }

    return ESP_OK;
}

httpd_handle_t startServer(void) {
    httpd_handle_t server = NULL;
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    if(httpd_start(&server, &config) == ESP_OK) {
        httpd_uri_t getURI;
        getURI.uri = "/set";
        getURI.method = HTTP_GET;
        getURI.handler = getHandler;

        httpd_register_uri_handler(server, &getURI);
        return server;
    }

    return NULL;
}

httpd_handle_t stopServer(httpd_handle_t server) {
    httpd_stop(server);
    return NULL;
}

httpd_handle_t server = NULL;


void setIntArr(uint8_t *arr, const char *value) {
    int i = 0;
    for(; value[i] != 0; i++) {
        arr[i] = (uint8_t)(value[i]);
    }
}


void eventHandler(void *arg, esp_event_base_t eventBase, int32_t eventId, void *
    data) {
    if(eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_START) {
        ESP_LOGI("WiFi", "Connecting to AP...");
        gpio_set_level(LED2, 0);
        blink(LED3, 2, 1);
        esp_wifi_connect();
```

```c
    } else if (eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_DISCONNECTED) {
        server = stopServer(server);
        wifi_event_sta_disconnected_t *event = (wifi_event_sta_disconnected_t*) data;
        ESP_LOGI("WiFi", "Disconnected from AP %02x:%02x:%02x:%02x:%02x:%02x Reason: %d. Reconnecting...", event->bssid[0], event->bssid[1], event->bssid[2], event->bssid[3], event->bssid[4], event->bssid[5], event->reason);
        gpio_set_level(LED2, 0);
        blink(LED3, 2, 1);
        esp_wifi_connect();
    } else if(eventBase == IP_EVENT && eventId == IP_EVENT_STA_GOT_IP) {
        server = startServer();
        ip_event_got_ip_t *event = (ip_event_got_ip_t *)data;
        ESP_LOGI("WiFi", "Connected to AP. IP=%s", ip4addr_ntoa(&event->ip_info.ip));
        gpio_set_level(LED3, 0);
        blink(LED2, 10, 1);
    }
}

void initWifi() {
    tcpip_adapter_init();
    esp_event_loop_create_default();

    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init((const wifi_init_config_t *)(&config)));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &eventHandler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &eventHandler, NULL));

    wifi_config_t wifiConfig;
    setIntArr(wifiConfig.sta.ssid, SSID);
    wifiConfig.sta.ssid[strlen(SSID)] = 0;
    setIntArr(wifiConfig.sta.password, PASSWORD);
    wifiConfig.sta.threshold.authmode = WIFI_AUTH_WPA2_PSK;

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifiConfig));
    ESP_ERROR_CHECK(esp_wifi_start());
}


void app_main(void) {
    setupGPIO();
    initWifi();
}
```

Listing 4: Web Server example

A.5. Web Client example

```c
#include<string.h>
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "lwip/err.h"
#include "lwip/sys.h"
#include "lwip/sockets.h"
#include "lwip/netdb.h"
#include "driver/gpio.h"
#include "arch/sys_arch.h"
#include "esp_http_server.h"

#define SSID "myssid"
#define PASSWORD "mypassword"

#define SWITCH 16

int LEDState = 0;

static const char *req = "GET /set?led=%s HTTP/1.1\r\nHost: 192.168.4.2\r\nUser-
    Agent: esp8266\r\n\r\n\n";

char *getRequest(char *state) {
    int bufsize = strlen(req) - 2 + strlen(state);
    char *buf = malloc(sizeof(char) * bufsize);
    buf[bufsize-1] = 0;
    snprintf(buf, bufsize, req, state);
    return buf;
}

void setupGPIO(void) {
    gpio_config_t config;
    config.pin_bit_mask = (1UL<<SWITCH);
    config.mode = GPIO_MODE_INPUT;
    config.pull_down_en = GPIO_PULLDOWN_ENABLE;

    gpio_config(&config);
}

void blink(int pin, int iter, int endState) {
    for(int i = 0; i < iter; i++) {
        gpio_set_level(pin, 0);
        sys_delay_ms(100);
        gpio_set_level(pin, 1);
        sys_delay_ms(100);
    }
    gpio_set_level(pin, endState);
}


```

```c
void sendRequest(char *state) {
    int fd = socket(AF_INET, SOCK_STREAM, 0);

    const struct addrinfo hints = {
        .ai_family = AF_INET,
        .ai_socktype = SOCK_STREAM,
    };
    struct addrinfo *res;

    getaddrinfo("192.168.4.2", "80", &hints, &res);

    if(connect(fd, res->ai_addr, res->ai_addrlen) != 0) {
        ESP_LOGE("HTTP", "Unable to connect to server");
        close(fd);
        return;
    }
    char *request = getRequest(state);
    ESP_LOGI("HTTP", "Sending request:\n%s", request);
    if(write(fd, request, strlen(request)) < 0) {
        ESP_LOGE("HTTP", "Unable to send request to server");
        close(fd);
        free(request);
        return;
    }

    ESP_LOGI("HTTP", "Sent request");
    close(fd);
    free(request);
}

void setIntArr(uint8_t *arr, const char *value) {
    int i = 0;
    for(; value[i] != 0; i++) {
        arr[i] = (uint8_t)(value[i]);
    }
}


void eventHandler(void *arg, esp_event_base_t eventBase, int32_t eventId, void *data) {
    if(eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_START) {
        ESP_LOGI("WiFi", "Connecting to AP...");
        esp_wifi_connect();
    } else if (eventBase == WIFI_EVENT && eventId == WIFI_EVENT_STA_DISCONNECTED) {
        wifi_event_sta_disconnected_t *event = (wifi_event_sta_disconnected_t*)data;
        ESP_LOGI("WiFi", "Disconnected from AP %02x:%02x:%02x:%02x:%02x:%02x Reason: %d. Reconnecting...", event->bssid[0], event->bssid[1], event->bssid[2], event->bssid[3], event->bssid[4], event->bssid[5], event->reason);
        esp_wifi_connect();
    } else if(eventBase == IP_EVENT && eventId == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t *event = (ip_event_got_ip_t *)data;
        ESP_LOGI("WiFi", "Connected to AP. IP=%s", ip4addr_ntoa(&event->ip_info.ip));
    }
}

void initWifi() {
    tcpip_adapter_init();
    esp_event_loop_create_default();

    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init((const wifi_init_config_t *)(&config)));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &eventHandler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &eventHandler, NULL));

    wifi_config_t wifiConfig;
    setIntArr(wifiConfig.sta.ssid, SSID);
    wifiConfig.sta.ssid[strlen(SSID)] = 0;
    setIntArr(wifiConfig.sta.password, PASSWORD);
    wifiConfig.sta.threshold.authmode = WIFI_AUTH_WPA2_PSK;

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifiConfig));
    ESP_ERROR_CHECK(esp_wifi_start());
}


void app_main(void) {
    setupGPIO();
    initWifi();

    int level;
    while(1) {
        level = gpio_get_level(SWITCH);
        if(level != 0) {
            ESP_LOGI("GPIO", "Blink LED");
            sendRequest("on");
            sys_delay_ms(500);
            sendRequest("off");
            sys_delay_ms(500);
            sendRequest("on");
            sys_delay_ms(500);
            sendRequest("off");
            sys_delay_ms(500);
            sendRequest("on");
            sys_delay_ms(500);
            sendRequest("off");
            sys_delay_ms(500);
        } else {
            sys_delay_ms(10);
        }
    }
}
```

Listing 5: Web Client example