

Erwin Ouyang

Hands-On IoT: Wi-Fi and Embedded Web Development

Developing with ESP32, Arduino, C/C++,
HTML, CSS, and JavaScript by Examples

Hands-On IoT: Wi-Fi and Embedded Web Development

*Developing with ESP32, Arduino, C/C++, HTML, CSS, and
JavaScript by Examples*

Erwin Ouyang

*Help others achieve their dreams and
you will achieve yours.*

— LES BROWN

Downloading the Code

The code for the examples shown in this book is available on my GitHub repository, https://github.com/yohanes-erwin/arduino-ml/tree/master/esp32_examples.

Other Resources

Visit the following sites for other tutorials: www.hands-onembedded.com, aiotedge.tech

Preface

Rapid advances in IoT technology demand a lot of devices to be connected to the internet. To design such devices, we usually need knowledges about microcontrollers and computer network. As an example, we often found devices that can be connected to the network and can be configured via web interfaces. These devices implement embedded web server. For example, most of network devices usually use embedded web server as the interface for configuration.

Although there are a lot of books that discuss about microcontrollers or web development, they usually discuss the topics in separate books. Rarely, there is a book that discusses both of the topics in one book, i.e. the book that discusses how to create a web interface for a microcontroller. Therefore, this book is written to fill that gap. The Arduino library is used to program the ESP32, while HTML, CSS, and JavaScript are used to build the web interface.

*Erwin Ouyang
March 2020*

Contents

Preface	i
Contents	ii
Listings	vii
1 Introduction	1
1.1 Internet of Things	1
1.2 ESP32	3
1.2.1 ESP32 Modules	5
1.2.2 ESP32 Development Boards	6
1.3 Wi-Fi	7
1.4 Arduino	8
1.5 Embedded Web Development	8
1.5.1 Back-End	9
1.5.2 Front-End	9
1.6 Prerequisites	10
1.6.1 DOIT Esp32 DevKit v1	11
1.6.2 ESP32 Library	11
1.7 ESP32 Documentations	12

1 Peripherals Programming	13
2 Digital Output	14
2.1 General Purpose Input/Output	14
2.2 GPIO as Output	15
2.3 ESP32 GPIO Peripheral: Output	15
2.4 Example Circuit: LED Circuits	17
2.5 Example Code: Control an LED	17
2.6 Summary	19
3 Digital Input	20
3.1 GPIO as Input	20
3.2 ESP32 GPIO Peripheral: Input	21
3.3 Example Circuit: Button Circuits	21
3.4 Example Code: Read a Button State	23
3.5 Summary	24
4 Serial I/O	25
4.1 Serial Communication	25
4.2 ESP32 UART Peripheral	26
4.2.1 Serial Print	27
4.2.2 Serial Read	29
4.3 Example Code: Serial Print	30
4.4 Example Code: Serial Read	31
4.5 Summary	32
5 Analog Output	33
5.1 Varying the Output Voltage	33
5.2 Pulse Width Modulation	34
5.3 ESP32 LEDC Peripheral	35
5.4 Example Code: Dim an LED with PWM	36
5.5 Summary	38
6 Analog Input	39

6.1	Varying the Input Voltage	39
6.2	Analog-to-Digital Converter	40
6.3	ESP32 ADC Peripheral	41
6.4	Example Circuit: Voltage Divider	41
6.5	Example Code: Read a Trimpot	42
6.6	Summary	44
7	Sensors	45
7.1	Sensing Physical Properties	45
7.2	Example Code: DHT11 Sensor	47
7.3	Example Code: DS1307 Real-Time Clock	50
7.4	Summary	52
II	Wi-Fi Programming	53
8	Wi-Fi Access Point	54
8.1	Wi-Fi Network	54
8.2	Access Point	56
8.3	ESP32 Wi-Fi Networking: Access Point	56
8.4	Example Code: Access Point Mode	58
8.5	Summary	61
9	Wi-Fi Station	62
9.1	Station	62
9.2	ESP32 Wi-Fi Networking: Station	63
9.3	Example Code: Station Mode	65
9.4	Summary	68
10	TCP Server	69
10.1	Internet Protocol Suite	69
10.2	TCP Server	71
10.3	ESP32 TCP Server	71
10.4	Example Code: TCP Server	72

10.5 Summary	77
11 TCP Client	78
11.1 TCP Client	78
11.2 ESP32 TCP Client	79
11.3 Example Code: TCP Client	80
11.4 Summary	84
12 HTTP Server	85
12.1 HyperText Transfer Protocol	85
12.2 HTTP Server	86
12.3 Message Format	87
12.4 ESP32 HTTP Server	88
12.5 Example Code: HTTP Server	89
12.6 Summary	94
13 HTTP Client	95
13.1 HTTP Client	95
13.2 ESP32 HTTP Client	96
13.3 Example Code: HTTP Client	96
13.4 Summary	100
III Embedded Web Development	101
14 Web Server	102
14.1 Web Server	102
14.2 ESPAsyncWebServer Library	103
14.3 Example Code: Simple Web Server	104
14.4 Summary	107
15 HyperText Markup Language (HTML)	108
15.1 HyperText Markup Language (HTML)	108
15.2 Example Code: HTML for Controlling an LED	110

15.3 Example Code: HTML for Dimming an LED	112
15.4 Example Code: HTML for Reading a Physical Button	113
15.5 Summary	115
16 Web Page Data Exchange	116
16.1 Web Page Data Exchange	116
16.2 Example Code: Web Server for Controlling an LED	117
16.3 Example Code: Web Server for Dimming an LED	120
16.4 HTTP GET and POST Methods	125
16.5 Example Code: Web Server for Reading a Physical Button	126
16.6 Summary	130
17 JavaScript (JS)	131
17.1 JavaScript	131
17.2 Example Code: Dimming an LED with JS	132
17.3 Example Code: Reading a Physical Button with JS	137
17.4 Example Code: Reading the DHT11 Sensor with JS	143
17.5 Wrapping Up: AJAX Technique	148
17.6 Summary	148
18 SPI Flash File System	149
18.1 SPI Flash File System	149
18.2 Example Code: Web Server using SPIFFS	150
18.3 Summary	153
19 Cascading Style Sheets (CSS)	154
19.1 Cascading Style Sheets	154
19.2 Bootstrap	156
19.3 Example Code: Bootstrap Label	158
19.4 Example Code: Bootstrap Button	164
19.5 Example Code: Bootstrap Card	170
19.6 Summary	176

20 Gauge and Chart	177
20.1 Gauge and Chart	177
20.2 Example Code: Gauge	178
20.3 Example Code: Chart	185
20.4 Summary	193
Bibliography	194

Listings

2.1	Control an LED	17
3.1	Read a button state	23
4.1	Serial print	30
4.2	Serial read	31
5.1	Dim an LED with PWM	36
6.1	Read a trimpot value	43
7.1	Read temperature and humidity from DHT11	48
7.2	Read timestamp from DS1307	51
8.1	Wi-Fi AP mode	58
9.1	Wi-Fi station mode	65
10.1	TCP server	73
11.1	TCP client	80
12.1	HTTP server	89
13.1	HTTP client	96
14.1	Simple web server using ESPAsyncWebServer	104
15.1	HTML code for controlling an LED	110
15.2	HTML code for dimming an LED	112
15.3	HTML code for reading a physical button	113
16.1	Web server for controlling an LED	117
16.2	Web server for dimming an LED	121
16.3	Web server for reading a physical button	126
17.1	Dimming an LED with JavaScript	132
17.2	Reading a pyhsical button with JavaScript	138

17.3	Reading the DHT11 sensor with JavaScript	143
18.1	Web server using SPIFFS	150
19.1	C code for styling a text using Bootstrap badge class	158
19.2	HTML code for styling a text using Bootstrap badge class	162
19.3	C code for styling HTML buttons using Bootstrap button class	164
19.4	HTML code for styling HTML buttons using Bootstrap button class	168
19.5	C code for creating a container for other HTML elements using Bootstrap card class	171
19.6	HTML code for creating a container for other HTML elements using Bootstrap card class	174
20.1	C code for streaming sensor readings to a real-time gauge	178
20.2	HTML code for streaming sensor readings to a real-time gauge	182
20.3	C code for streaming sensor readings to a real-time chart	185
20.4	HTML code for streaming sensor readings to a real-time chart	189

Chapter 1

Introduction

1.1 Internet of Things

Internet of Things (IoT) is a system of physical objects that are connected to the Internet. The physical objects can be anything from machines, animals, or people (wearable IoT devices). These objects are provided with unique identifiers (UIDs) and the ability to exchange data over a network [1].

The IoT applications are classified into consumer, commercial, industrial, infrastructure, and military [2]:

- **Consumer applications:** smart home, elder care.
- **Commercial applications:** medical and healthcare, transportation, V2X communications, building and home automation.
- **Industrial applications:** manufacturing, agriculture.
- **Infrastructure applications:** metropolitan scale deployments, energy management, environmental monitoring.

- **Military applications:** Internet of Battlefield Things, Ocean of Things.

The IoT stack consists of four technology layers: sensors, controllers, communications, and app platforms.

- **Sensors:** sensors are devices that are used to detect some type of input from the physical environment. This input is converted to electrical signal. The example of sensors are touch sensors, temperature sensors, light sensors, etc.
- **Controllers:** controllers are devices that are used for local data processing, storage, and data pre-processing before they are sent to the communication devices. The example of controllers are microcontrollers, microprocessors, digital signal processors (DSP) field-programmable gate arrays (FPGA).
- **Communications:** communications are devices that used for converting the data into analog or digital signal that can be transmitted over a communication medium either wired or wireless. The example of communications are serial port, USB, Wi-Fi, Bluetooth, LoRa, etc.
- **App platforms:** app platforms are used for device command and control, data collection, data presentation, data analysis, and data report. The data report from this layer can be integrated to business process.

The objectives of this book focus on the controllers and communications layers. The ESP32 microcontroller is used as the controller device and the Wi-Fi is used as the communication device.



Figure 1.1. ESP32 chip (ESP32-D0WDQ6). Retrieved March 14, 2020, from gridconnect.com.

1.2 ESP32

ESP32 is a low-cost microcontroller with integrated Wi-Fi and Bluetooth. It is a successor to the ESP8266. The ESP32 chip is shown in Figure 1.1. It is housed in a 6mm×6mm Quad-Flat No-leads (QFN) package. ESP32 is made by Espressif Systems, a Shanghai-based Chinese fabless semiconductor company. ESP32 is a very popular chip for developing IoT devices.

The ESP32 functional block diagram is shown in Figure 1.2. The ESP32 specifications are listed as the following [3]:

- **Processors:**
 - CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 160 or 240 MHz and performing at up to 600 DMIPS
 - Ultra low power (ULP) co-processor
- **Memory:** 512 KiB SRAM
- **Wireless connectivity:**
 - Wi-Fi: 802.11 b/g/n

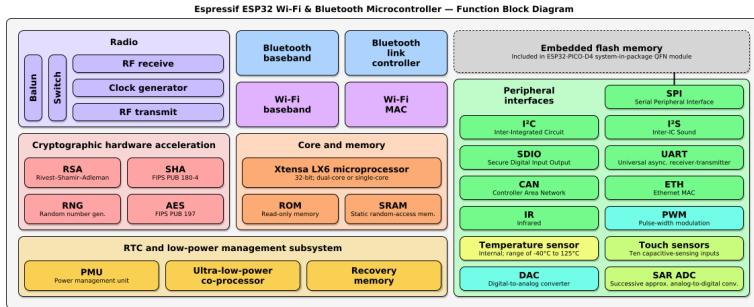


Figure 1.2. ESP32 functional block diagram. Retrieved March 14, 2020, from wikipedia.org.

- Bluetooth: v4.2 BR/EDR and BLE

- **Peripheral interfaces:**

- 12-bit SAR ADC up to 18 channels
- 2 × 8-bit DACs
- 10 × touch sensors (capacitive sensing GPIOs)
- 4 × SPI
- 2 × I₂S interfaces
- 2 × I₂C interfaces
- 3 × UART
- SD/SDIO/CE-ATA/MMC/eMMC host controller
- SDIO/SPI slave controller
- Ethernet MAC interface with dedicated DMA and IEEE 1588 Precision Time Protocol support
- CAN bus 2.0
- Infrared remote controller (TX/RX, up to 8 channels)

- Motor PWM
- LED PWM (up to 16 channels)
- Hall effect sensor
- Ultra low power analog pre-amplifier

- **Security:**

- IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
- Secure boot
- Flash encryption
- 1024-bit OTP, up to 768-bit for customers
- Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)

- **Power management:**

- Internal low-dropout regulator
- Individual power domain for RTC
- $5\mu\text{A}$ deep sleep current
- Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt

1.2.1 ESP32 Modules

Espressif offers ESP32 modules powered by ESP32 chips [4]. One of the most popular modules is the ESP32-WROOM-32xx. It is used in many development boards. The modules support PCB antenna or external antenna with IPEX/U.FL connector. They are shown in Figure 1.3.



Figure 1.3. ESP32 modules: (a) ESP32-WROOM-32 with PCB antenna; (b) ESP32-WROOM-32U with IPEX antenna. Retrieved March 14, 2020, from espressif.com.



Figure 1.4. DOIT Esp32 DevKit v1 board. Retrieved March 14, 2020, from zerynth.com.

1.2.2 ESP32 Development Boards

The ESP32 modules are usually mounted on the other PCB boards. These boards are called development boards. There are many ESP32 development boards available from Espressif [5]. In this book, the DOIT Esp32 DevKit v1 board [6] is used as a reference. This is a low-cost and quite popular development board. The board is shown in Figure 1.4. It uses the ESP32-WROOM-32 module.

Table 1.1. Wi-Fi versions [8].

IEEE Standard	Maximum Linkrate	Frequency
IEEE 802.11ax	600–9608 Mbit/s	2.4/5 GHz, 1–6 GHz ISM
IEEE 802.11ac	433–6933 Mbit/s	5 GHz
IEEE 802.11n	72–600 Mbit/s	2.4/5 GHz
IEEE 802.11g	3–54 Mbit/s	2.4 GHz
IEEE 802.11a	1.5 to 54 Mbit/s	5 GHz
IEEE 802.11b	1 to 11 Mbit/s	2.4 GHz

1.3 Wi-Fi

Wi-Fi is a family of wireless communication technologies, based on the IEEE 802.11 family of standards. Wi-Fi is commonly used for local area networking of devices and Internet access [7]. There are several Wi-Fi versions available as shown in Table 1.1.

There are two commonly used Wi-Fi modes: infrastructure and ad hoc/Wi-Fi direct:

- **Infrastructure:** this mode is the most commonly used mode. On the network, there is one base station and multiple stations. The base station is called Wi-Fi access point (AP)/Wi-Fi router, and the stations are the client devices, such as mobile phones, tablets, PCs/laptops, etc. In this mode, all communications goes through the base station.
- **Ad hoc/Wi-Fi direct:** this mode allows communications from one station to another station without an AP. This mode is popular in multiplayer handheld game consoles, such as Playstation Vita, Nintendo Switch, etc. Wi-Fi direct is popular for file transfer and media sharing between mobile devices, laptops, game consoles, and televisions.

This book focuses on the infrastructure mode, while the ad hoc/Wi-Fi direct mode is beyond the scope of this book. We are going to configure the Wi-Fi of ESP32 as a station that connects to the AP.

1.4 Arduino

Arduino is an open-source hardware and software company [9]. They manufacture microcontroller boards and develop libraries and integrated development environment (IDE) to program the boards. The products are licensed under the GNU Lesser General Public License (LGPL) or the GNU General Public License (GPL).

The Arduino libraries and IDE are not limited to boards manufactured by Arduino, so other boards can use them as well. As an example, in this book we use the Arduino libraries and IDE to program the ESP32. In order to use Arduino IDE to program the ESP32, a custom library for ESP32 is required [10]. It is developed by Espressif.

The Arduino IDE is shown in Figure 1.5. It is a cross-platform application that available for Windows, macOs, and Linux.

1.5 Embedded Web Development

An embedded web is a website that is built into embedded devices. The web page provides a control panel for configuring and monitoring the device. A good example of these devices are computer network devices, such as Ethernet switches, Wi-Fi AP/routers, modems, etc. They usually have a web interface for configuring and monitoring the device.

The web development is divided into two parts: back-end and front-end. A block diagram of embedded web that consists of back-end and front-end is shown in Figure 1.6.

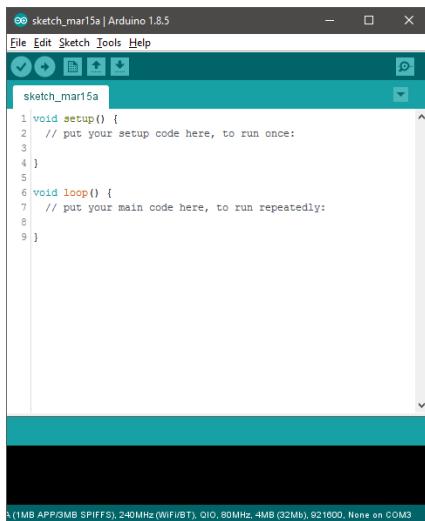


Figure 1.5. Arduino IDE.

1.5.1 Back-End

Back-end refers to the server-side of the web application. It is responsible to ensure everything works. It uses an embedded web (HTTP) server to serve HTTP request from the client (web browsers). In this book, we are going to build embedded web servers in the ESP32. We are going to use a library called ESPAsyncWebServer [11].

1.5.2 Front-End

Front-end refers to the client-side of the web application. It is basically what users see through the web browsers. There are three main programming languages that are used to develop front-end: HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS). HTML provides the elements and structure of web

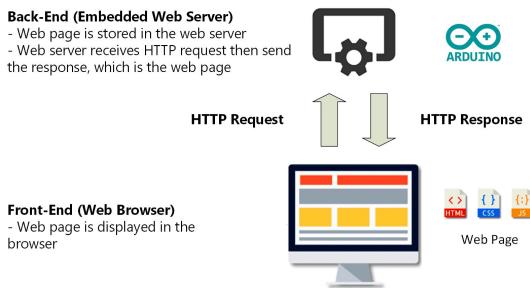


Figure 1.6. A block diagram of an embedded web.

pages. CSS enhances the layout and style of the HTML. JavaScript is used to control the behavior of the elements.

1.6 Prerequisites

In order to follow the example programs in this book, there are several prerequisites:

- **DOIT Esp32 DevKit v1:** the example programs in this book are based on this ESP32 board, but other ESP32 boards should work as well.
- **Arduino IDE:** the Arduino IDE from <https://www.arduino.cc/en/Main/Software> is installed on your development PC. It is required to use Arduino IDE at version 1.8 or later.
- **ESP32 library:** the ESP32 library from <https://github.com/espressif/arduino-esp32> is installed on your Arduino IDE.
- **ESP32 file system uploader:** this is an Arduino IDE plugin for uploading data files to the ESP32. You can get it from [https:](https://)

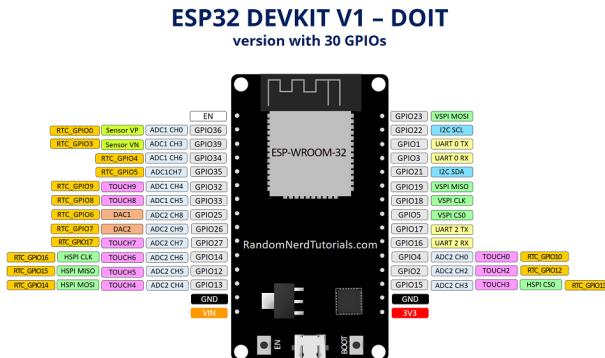


Figure 1.7. DOIT Esp32 DevKit v1 pin diagram. Retrieved March 16, 2020, from randomnerdtutorials.com.

```
//github.com/me-no-dev/arduino-esp32fs-plugin.  
It is used in chapter 18 and later.
```

1.6.1 DOIT Esp32 DevKit v1

There are two versions of DOIT Esp32 DevKit v1: version with 30 pins and version with 36 pins. In this book, we are going to use the version with 30 pins. The pin diagram of this board is shown in Figure 1.7. This board can be powered via the on-board micro B USB connector or via the VIN pin. The recommended external supply voltage for the VIN pin is from 7 to 12V [12].

1.6.2 ESP32 Library

The ESP32 library, it is also called 'Arduino core for the ESP32', is required by the Arduino IDE in order to program the ESP32. Follow the following instructions to install the ESP32 library [13]:

- Start Arduino IDE and open **Preferences** window.

- Enter this link: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json into **Additional Board Manager URLs** field. You can add multiple URLs by separating them with commas.
- Go to menu **Tools** → **Board**, then open **Boards Manager** and install **esp32**.

1.7 ESP32 Documentations

There are several ESP32 documentations that you may need to program the ESP32 as follows:

- **ESP32 Datasheet:** https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
- **ESP-IDF Programming Guide:** <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html>

Part I

Peripherals Programming

Chapter 2

Digital Output

What will you learn in this chapter?

- Concept of digital output.
- Use digital output to control an LED.

2.1 General Purpose Input/Output

General purpose input/output (GPIO) is a standard interface used for connecting microcontrollers to other electronic components. For example, it can be connected to a light bulb (with external driver circuit) to turn it on or off, or it can be connected to a button to count how many times it is pressed.

A GPIO can be customized and controlled by software either as input or output. It works as a digital input or a digital output. In other words, it works in two discrete levels, which are logical high and low. These logic states are usually represented by two different voltages, which are called logic voltage. For example, a logical high is represented as

+5V or +3.3V, and a logical low is represented as 0V.

2.2 GPIO as Output

Digital output is the simplest way that your microcontrollers are going to control other devices. When a GPIO is configured as a digital output, it is usually used to control other electronic components, such as LED, buzzer, relay, DC motor, etc. With a digital output, you can turn them on or off.

Some of these components, such as relay or DC motor, require more voltage and current than a digital output can supply. A typical digital output can only supply current limited to about 20mA. Therefore, you need an intermediary (driver circuit). The most common component that is used to control other components that need higher voltage and current is the transistor.

2.3 ESP32 GPIO Peripheral: Output

The ESP32 chip has 40 physical GPIO pads. However, some of them can't be used or don't have the corresponding pin on the chip package. Each GPIO can be configured as output, input, or connected to internal peripherals, such as ADC, DAC, I2C, SPI, UART, etc. The logic voltage of the GPIO is 3.3V (not 5V-tolerant). There are some GPIO pins with specific limitations that may not suitable for a specific project. You can refer to this reference [14] for the details.

In ESP32 Arduino, there are two important functions for using digital output: `pinMode` and `digitalWrite`. The `pinMode` function is defined as

```
void pinMode(uint8_t pin, uint8_t mode);
```

which is used to configure the GPIO either as output or input. It takes two input arguments: GPIO pin that you want to configure and

GPIO mode (`OUTPUT`, `INPUT`, or `INPUT_PULLUP`). We will get into digital input in the next chapter.

The `digitalWrite` function is defined as

```
void digitalWrite(uint8_t pin, uint8_t val);
```

which is used to write the logic state to the digital output. It takes two input arguments: GPIO pin that you want to write and the output state (`HIGH` or `LOW`).

Those functions are defined in the ESP32 Arduino core in the `esp32-hal-gpio.h` and `esp32-hal-gpio.c`.

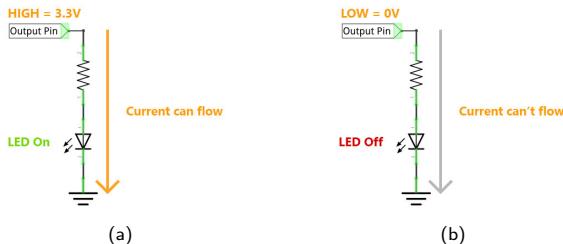


Figure 2.1. Active-high LED circuit: (a) A logical high turns on the LED; (b) A logical low turns off the LED.

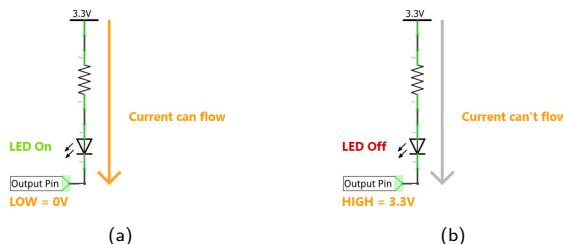


Figure 2.2. Active-low LED circuit: (a) A logical low turns on the LED; (b) A logical high turns off the LED.

2.4 Example Circuit: LED Circuits

A light-emitting diode (LED) is an electronic component that emits light when current flows through it. There are two ways of wiring an LED to the digital output pin, which are active-high and active-low.

- **Active-high:** the active-high LED circuit is shown in Figure 2.1. The LED is on when the output pin is high. It is called active-high because a logical high is needed to activate (turn on) the LED. When a logical high is written to the output pin, the current can flow through the resistor and LED, and then flows to the GND. Therefore, the LED is on. When a logical low is written, the current can't flow. Therefore, the LED is off.
- **Active-low:** the active-low LED circuit is shown in Figure 2.2. The LED is on when the output pin is low. It is called active-low because a logical low is needed to activate (turn on) the LED. When a logical low is written to the output pin, the current can flow from the 3.3V through the resistor and LED, and then flows to the GND. Therefore, the LED is on. When a logical high is written, the current can't flow. Therefore, the LED is off.

2.5 Example Code: Control an LED

The easiest way to see digital output in action is to create an example code to turn on and off an LED. In this code, we are going to blink (turn on and off repeatedly) the on-board LED. The on-board LED is connected to the pin D2 of the DOIT Esp32 DevKit v1. It uses an active-high LED circuit. The code is shown in Listing 2.1.

Listing 2.1. Control an LED

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 22 Feb 2020
3
```

```
4 void setup()
5 {
6     // Set GPIO pin as output for on-board LED
7     pinMode(2, OUTPUT);
8 }
9
10 void loop()
11 {
12     // Turn on LED
13     digitalWrite(2, HIGH);
14     delay(500);
15     // Turn off LED
16     digitalWrite(2, LOW);
17     delay(500);
18 }
```

Here is the step-by-step to control an LED with digital output using the ESP32 Arduino:

1. First, you need to configure a GPIO pin as output. In **line 7**, you need to configure pin D2 as output by calling `pinMode` function.
2. Then, in order to turn on the LED, you need to write a logical high to pin D2 by calling `digitalWrite` function as in **line 13**.
3. After that, in **line 14**, add a delay of 500ms by calling `delay` function. So, the LED is turned on for 500ms.
4. Finally, in **line 16-17**, you need to do the same procedure as in step 2-3 to turn off the LED for 500ms.

Note that if you don't add the delay or use a very short delay, then you may not see the blinking because it is too fast for our eyes to see the changes.

2.6 Summary

In this chapter, you have learned how to configure a GPIO as a digital output. It produces a logical high or logical low that turns on or off the LED, depending on the circuits either active-high or active-low. By adding the delay between on and off states, you can see the blinking LED.

Chapter 3

Digital Input

What will you learn in this chapter?

- Concept of digital input.
- Use digital input to read a button state.

3.1 GPIO as Input

In the previous chapter, you have learned about GPIO output for controlling an LED. In this chapter, you are going to learn about digital input. Digital input is the simplest way that your microcontrollers can read logic state either high or low. When a GPIO is configured as a digital input, it is usually used to read logic state of the other electronic components, such as button, switch, keypad, rotary encoder, etc.

3.2 ESP32 GPIO Peripheral: Input

In this chapter, we are going to configure the GPIO peripheral as input. In ESP32 Arduino, there are two important functions for digital input: `pinMode` and `digitalRead`. The `pinMode` function is defined as

```
void pinMode(uint8_t pin, uint8_t mode);
```

which is used to configure the GPIO either as output or input. It takes two input arguments: GPIO pin that you want to configure and GPIO mode (`OUTPUT`, `INPUT`, or `INPUT_PULLUP`).

The `digitalRead` function is defined as

```
int digitalRead(uint8_t pin);
```

which is used to read the logic state of the digital input. It takes one input argument, which is the GPIO pin that you want to read. It returns the state of the digital input either `HIGH` or `LOW`.

Those functions are defined in the ESP32 Arduino core in the `esp32-hal-gpio.h` and `esp32-hal-gpio.c`.

3.3 Example Circuit: Button Circuits

A button is an electronic component that has momentary "on" position when it is pressed, and it is reverted to "off" position when it is not pressed. A switch is another electronic component has "on" and "off" positions, but it can maintain its "on" position. There are two ways of wiring a button or switch to the digital input pin, which are active-high and active-low.

- **Active-high:** the active-high button circuit is shown in Figure 3.1. In this circuit, the GND is connected to an input pin through a resistor, so there is no current flows to the input pin. Therefore, it receives a logical low. When the button is pressed,

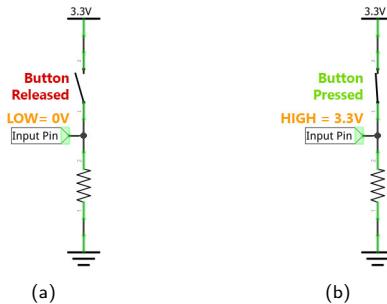


Figure 3.1. Active-high button circuit: (a) Button released, input pin receives a logical low; (b) Button pressed, input pin receives a logical high.

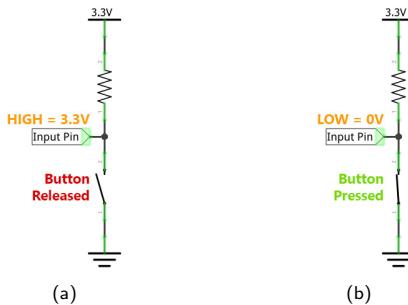


Figure 3.2. Active-low button circuit: (a) Button released, input pin receives a logical high; (b) Button pressed, input pin receives a logical low.

the current flows from VCC to the input pin and also to the resistor. Therefore, the input pin receives a logical high. The resistor in this circuit is called pull-down resistor. The typical pull-down resistor value is $1\text{--}10\text{k}\Omega$.

- **Active-low:** the active-low button circuit is shown in Figure 3.2. In this circuit, the VCC is connected to an input pin through a resistor, so the current can flow to the input pin. Therefore, it

receives a logical high. When the button is pressed, the current flows through the button, because the button's resistance is almost zero, and the input pin has high impedance. Therefore, the input pin receives a logical low. The resistor in this circuit is called pull-up resistor. The typical pull-up resistor value is 1–10k Ω .

3.4 Example Code: Read a Button State

In this example code, we are going to read a button state using the ESP32. The button is connected between the pin D12 and the GND of the DOIT Esp32 DevKit v1. We use the active-low button circuit with internal pull-up resistor. You also need an LED that is used as an indicator of the button state. In this case, you are going to use the on-board LED. The code is shown in Listing 3.1.

Listing 3.1. Read a button state

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 8 Mar 2020
3
4 void setup()
5 {
6     // Set GPIO pin as output for on-board LED
7     pinMode(2, OUTPUT);
8     // Set GPIO pin as input with pull-up resistor for button
9     pinMode(12, INPUT_PULLUP);
10 }
11
12 void loop()
13 {
14     // *** Read button state ***
15     if (digitalRead(12) == LOW)
16     {
17         // If button is pressed, then turn on the LED
18         digitalWrite(2, HIGH);
```

```
19     }
20 else
21 {
22     // If button is not pressed, then turn off the LED
23     digitalWrite(2, LOW);
24 }
25 }
```

Here is the step-by-step to read a button state with digital input using the ESP32 Arduino:

1. First, you need to configure the GPIO pin of the on-board LED as output as in **line 7**.
2. Then, you need to configure a GPIO pin as input. In **line 9**, you need to configure pin D12 as input with internal pull-up resistor by calling `pinMode` function.
3. Finally, in **line 15**, you need to read the state of the input pin by calling the `digitalRead` function.
4. If the button state is `LOW`, it means the button is pressed (because it is an active-low circuit). So, you need to turn on the LED as in **line 18**. Otherwise, you need to turn off the LED as in **line 23**.

3.5 Summary

In this chapter, you have learned how to configure a GPIO as a digital input. It reads a logical high or logical low voltage from external circuit. In this case, we use button as the example, but you can change it to other components that produce digital output.

Chapter 4

Serial I/O

What will you learn in this chapter?

- Concept of serial communication.
- Use serial I/O to transfer data between ESP32 and PC.

4.1 Serial Communication

In this chapter, we begin to discuss how microcontrollers or computers actually talk to each other. There are rules (protocols) that define the encoding of data into voltage pulses and the decoding of voltage pulses back into data. There are many protocols available, such as UART, SPI, I2C, USB, etc. This chapter focuses to the simplest serial protocol, which is the universal asynchronous receive and transmit (UART).

In UART protocol, the data bits are sent one bit at a time. This protocol is widely used for communication between an embedded device and a laptop/PC. On the laptops/PCs side, the serial communication

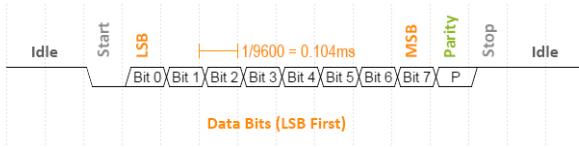


Figure 4.1. UART data frame.

is done through the DB9 RS232 port or the USB port (by using a USB to UART bridge). However, on modern laptops/PCs, the DB9 RS232 port is not usually available, so the UART over USB is more common. On the embedded devices side, it is also common to use the UART over USB (mini or micro) port.

The UART protocol uses two wires for sending and receiving data, namely TX and RX. The UART data frame is shown in Figure 4.1. The length of data bits of each frame is 5–9 data bits with LSB first configuration. The commonly used length is 8 bits (1 byte). Then, there are start bit and stop bit, which are used to indicate where the first and last data bit is, respectively. The start bit is 0, and the stop bit is 1. The length of the stop bit can be configured either 1, 1.5, or 2 bits. The parity bit is an optional bit that is used for error detection. There are two types of parity: even parity and odd parity. There are several standards for the data speed (baud rate), such as 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200 bps.

4.2 ESP32 UART Peripheral

The ESP32 has three UART controllers. They are implemented as hardware on the ESP32 chip. They handle timing requirements and data framing of the UART protocol. Each of them can be independently configured with parameters, such as baud rate, data bit length, number of stop bit, parity bit, etc. The TX and RX of UART controllers have default GPIO pins, but they can also be logically mapped

to any of the available GPIO pins. The GPIO pins work at 3.3V TTL level.

In ESP32 Arduino, there are several important methods¹ for serial I/O: begin, print methods, and read methods. The `begin` method is defined as

```
void begin(unsigned long baud,  
          uint32_t config=SERIAL_8N1,  
          int8_t rxPin=-1,  
          int8_t txPin=-1);
```

This method is used to configure the UART. It has four input arguments:

- `baud`: baud rate, e.g. 9600, 19200, etc.
- `config`: data, parity, and stop bit, e.g. `SERIAL_8N1`. See the `esp32-hal-uart.h` for available definitions.
- `rxPin` and `txPin`: RX GPIO pin and TX GPIO pin, respectively. The pin -1 means that it uses the default GPIO pin.

The last three input arguments are optional because they have default arguments. For default usage, i.e. to send or receive data to or from serial monitor, you need to provide only the first input argument, which is the baud rate.

4.2.1 Serial Print

There are several print methods that can be used for sending data to the serial output: `print`, `println`, and `printf`. You can pass any printable data type as their input arguments. The `printf` method

¹ A method and a function are the same, with different terms. The term 'method' is used in object-oriented programming. A method is a function associated with a class.

Table 4.1. The commonly used format specifiers [15].

Specifier	Output	Example
%d or %i	Signed decimal integer	392
%x	Unsigned hexadecimal integer	7fa
%X	Unsigned hexadecimal integer (uppercase)	7FA
%o	Unsigned octal	610
%f	Decimal floating point	392.65
%c	Character	a
%s	String of characters	sample

Table 4.2. The example of commonly used sub-specifiers [15].

Sub-specifier	Output	Example
%08d	Preceding with zeros	00001977
%#x	Preceding with 0x	0x7fa
%.2f	Number of digit after decimal point	3.14

requires format specifiers. The format specifiers start with a % character, and indicate the location to translate a piece of data (char, int, float, etc.) to characters. The commonly used format specifiers are shown in Table 4.1. The specifiers can also contain sub-specifiers, which are optional. The example of commonly used sub-specifiers are shown in Table 4.2. The print methods are part of `Stream` class that is defined in `Stream.h` and `Stream.cpp`.

How to print a number in binary?

The `printf` method is only able to print a number in base 8 (octal), 10 (decimal), and 16 (hexadecimal), so there is no specifier for binary integer. Alternatively, you can use the `print` or `println` method, which has an optional second parameter specifies the base to use. The optional second parameters are `BIN`, `OCT`, `DEC`, and `HEX`. For example, `Serial.print(78, BIN)` gives "1001110".

4.2.2 Serial Read

There are several read methods that can be used for receiving data from the serial input: `readBytes`, `readBytesUntil`, `readString`, and `readStringUntil`. The `readBytes` and `readBytesUntil` methods are defined as

```
size_t readBytes(char *buffer, size_t length);
size_t readBytesUntil(char terminator,
                      char *buffer,
                      size_t length);
```

They are used for reading characters (`char` or `uint8_t`) from internal stream buffer. The `readBytesUntil` method reads data bytes until it finds the terminator character. It returns the number of characters placed in the buffer.

The `readString` and `readStringUntil` methods are defined as

```
String readString();
String readStringUntil(char terminator);
```

They are used for reading a string from internal stream buffer. The `readStringUntil` method reads a string until it finds the terminator character.

The read methods are part of `Stream` class that is defined in `Stream.h` and `Stream.cpp`.

4.3 Example Code: Serial Print

In this example code, we are going to print a message to the serial monitor. The code is shown in Listing 4.1.

Listing 4.1. Serial print

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 22 Feb 2020
3
4 void setup()
5 {
6     // Initialize serial port
7     Serial.begin(115200);
8 }
9
10 void loop()
11 {
12     // Print text
13     Serial.println("Hello, World!");
14     delay(1000);
15 }
```

Here is the step-by-step to print a message to the serial monitor:

1. First, you need to configure the UART with a baud rate of 115200 as in **line 7**.
2. Then, you need to print the message as in **line 13**.
3. Finally, you need to add a delay of one second as in **line 14**. So, the message is printed only once every one second.

Note that the `Serial` object is never instantiated in this file. This is because the `Serial` is an `extern` object that is already instantiated in the `HardwareSerial.h` file. After you upload the code to the ESP32, you can open serial monitor in the Arduino IDE, and set the baud rate to 115200.

4.4 Example Code: Serial Read

In this example code, we are going to read a message from the serial monitor. Then, we are going to send back (echo) the message to the serial monitor. The code is shown in Listing 4.2.

Listing 4.2. Serial read

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 8 Mar 2020
3
4 void setup()
5 {
6     // Initialize serial port
7     Serial.begin(115200);
8 }
9
10 void loop()
11 {
12     // If there is data in receive buffer
13     if (Serial.available() > 0)
14     {
15         // Read string from receive buffer
16         String input = Serial.readString();
17         // Print to serial monitor
18         Serial.print(input);
19     }
20 }
```

Here is the step-by-step to read a message from the serial monitor:

1. First, you need to configure the UART with a baud rate of 115200 as in **line 7**.
2. Then, in main loop, you need to check whether there is data or not in the receive buffer as in **line 13**.
3. If there is data in the receive buffer, then you need to read the data by calling `readString` method as in **line 16**.
4. Finally, in **line 18**, you need to send back the message to the serial monitor.

4.5 Summary

In this chapter, you have learned how to use the UART for serial communication between ESP32 and PC. Both of them can send and receive data. We will use this serial communication mainly for debugging.

Chapter 5

Analog Output

What will you learn in this chapter?

- Concept of analog output.
- Use analog output to dim an LED.

5.1 Varying the Output Voltage

The world we live in is an analog one. It is more than simply turn a device on and off. For example, sometimes you may want to control the brightness of a lamp, or change the volume of a speaker, or change the speed of a fan. So, you need to vary the output, which is called analog output.

Microcontrollers are digital components. Without an additional component, microcontrollers can't produce an analog voltage. They can only produce logic high or low voltage. In ESP32 case, the logic high is 3.3V and the logic low is 0V.

There are two techniques to produce analog voltage from microcon-

trollers:

- **Pulse width modulation (PWM):** in this technique, you create a "fake" analog voltage. You pulse your digital output pin high and low for a specific time. By using a very small time, a few microseconds or milliseconds, you can create a pseudo-analog voltage. Over time, the average voltage of the pulse can be a range of values between logic high and low.
- **Digital-to-analog conversion:** in this technique, you create a "real" analog voltage. You use a component called digital-to-analog converter (DAC). A DAC converts a digital value to an analog voltage.

Many microcontrollers do not have on-chip DAC. This is because the things that microcontrollers typically control, such as LED or DC motor, can be controlled with PWM technique. Unless you are doing high quality audio or signal processing, the on-chip or external DAC is not needed. PWM technique is very common to be used in microcontrollers.

5.2 Pulse Width Modulation

Pulse width modulation (PWM) is a square wave, which the logic high and low duration can be varied. Figure 5.1 shows an illustration of PWM signals. The time the digital output is high is called pulse width. The ratio of pulse width to the square wave period is called duty cycle. If you reduce the pulse width, you get a lower analog voltage, and vice versa. The pseudo-analog output is the average voltage of the pulse width. For example, if the duty cycle is 50 percent, the average voltage is half the total voltage.

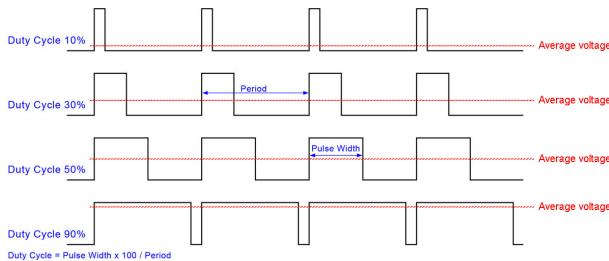


Figure 5.1. PWM with several duty cycles. Retrieved September 11, 2018, from caferacersjpg.com.

5.3 ESP32 LEDC Peripheral

The ESP32 has a LED PWM controller (LEDC) peripheral. It is primarily designed to control the intensity of LEDs, but it can also be used to generate PWM signals for other purposes as well. It has 16 independent channels with configurable periods and duties.

In ESP32 Arduino, there are three important functions for LEDC: `ledcSetup`, `ledcAttachPin`, and `ledcWrite`. The `ledcSetup` function is defined as

```
double ledcSetup(uint8_t chan, double freq,  
      uint8_t bit_num);
```

which is used to configure the PWM properties. It takes three input arguments: PWM channel, frequency, and duty cycle resolution.

The `ledcAttachPin` function is defined as

```
void ledcAttachPin(uint8_t pin, uint8_t chan);
```

which is used to specify to which GPIO pin the PWM output will appear on. It takes two input arguments: GPIO pin and the PWM channel.

The `ledcWrite` function is defined as

```
void ledcWrite(uint8_t channel, uint32_t duty);
```

which is used to set duty cycle of the PWM output. It takes two input arguments: the PWM channel and duty cycle value.

Those functions are defined in the ESP32 Arduino core in the `esp32-hal-ledc.h` and `esp32-hal-ledc.c`.

5.4 Example Code: Dim an LED with PWM

The easiest way to see PWM in action is to create an example code to dim an LED. The code is shown in Listing 5.1.

Listing 5.1. Dim an LED with PWM

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 29 Feb 2020
3
4 // On-board LED's GPIO pin
5 #define LED_ON_BOARD 2
6
7 // PWM parameters
8 const int channel = 0;
9 const int freq = 1000;
10 const int res = 10;
11
12 void setup()
13 {
14     // Configure PWM
15     ledcSetup(channel, freq, res);
16     // Attach the PWM channel to the LED
17     ledcAttachPin(LED_ON_BOARD, channel);
18 }
19
20 void loop()
```

```
21  {
22      // *** Increase brightness ***
23      for (int i = 0; i <= 1023; i += 20)
24      {
25          ledcWrite(channel, i);
26          delay(25);
27      }
28      // *** Decrease brightness ***
29      for (int i = 1023; i >= 0; i -= 20)
30      {
31          ledcWrite(channel, i);
32          delay(25);
33      }
34 }
```

Here is the step-by-step to dim an LED with PWM using the ESP32 Arduino:

1. First, you need to choose a PWM channel from 16 available channels (from 0 to 15). For example, in **line 8**, we use PWM channel 0.
2. Then, you need to set the PWM frequency. In **line 9**, we use a frequency of 1000 Hz.
3. After that, you need to set the duty cycle resolution. In **line 10**, we use 10-bit resolution. So, the PWM has $2^{10} = 1024$ (from 0 to 1023) discrete levels.
4. Then, you need to pass those parameters as the input arguments to the `ledcSetup` function as in **line 15**.
5. Next, you need to attach the PWM channel to the LED pin by calling `ledcAttach` function as in **line 17**. Here, we use the on-board LED, which is wired to pin D2.

6. Then, in **line 23-27**, the brightness of the LED is gradually increased. The brightness is increased from 0 to 1023 by 20 levels every 25 ms. We call the `ledcWrite` function to set the duty cycle.
7. Finally, in **line 29-33**, you need to do the same procedure as in the previous step to gradually decrease the brightness of the LED.

After you program the ESP32, the brightness of the on-board LED will be gradually increased and decreased. Actually, the LED is blinking on and off rather than dimming. This is because our eyes can't detect the changes if it is blinking too fast. To our eyes, the LED appears to be dimmed. If you use PWM frequency that is slow enough for our eyes to see (for example 1 Hz), you will see that the LED is blinking.

5.5 Summary

In this chapter, you have learned how to generate an analog output with PWM technique. It produces a pseudo-analog output that can be used to change the brightness of the LED.

Chapter 6

Analog Input

What will you learn in this chapter?

- Concept of analog input.
- Use analog input to read a trimpot.

6.1 Varying the Input Voltage

As you have seen in the previous chapter, the world we live in is an analog one. It is not just binary: on or off. There are many continuously variable values, such as voltage, current, light intensity, force, etc. In order to read this continuously variable values, microcontrollers need to have input that can read analog voltage, which is called analog input.

Microcontrollers are digital components. Without an additional component, microcontrollers can't read an analog voltage. They can only read logic high or low voltage. In ESP32 case, the logic high is 3.3V and the logic low is 0V. So, they need an additional component to read

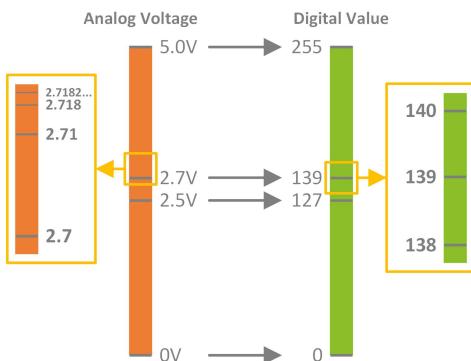


Figure 6.1. Conversion between an analog voltage to a digital value.

the analog voltage, which is an analog-to-digital converter (ADC). With ADC, microcontrollers can read varying voltage levels from, for example variable resistors like potentiometers or analog sensors.

6.2 Analog-to-Digital Converter

Analog-to-digital Converter (ADC) is a component that converts an analog voltage to a digital value. In mathematics, it is said that the analog voltage consists of an infinite number of points between point A and point B, for example between 0–5V. In Figure 6.1, you can see that after 2.7V there are 2.71V, 2.718V, 2.7182V, and so on, but a microcontroller can only represent a finite set of numbers (discrete numbers). For example, 8-bit number can only represent numbers from 0–255, so between 139 and 140 there is no other number.

6.3 ESP32 ADC Peripheral

The ESP32 has two 12-bit ADCs. However, the second ADC is used by the Wi-Fi. Therefore, the application can only use the second ADC when the Wi-Fi is not used. The voltage range of the ADC is between 0V and 3.3V. Because the resolution of this ADC is 12-bit ($2^{12} = 4096$), so the analog input voltage is assigned to a value between 0 and 4095.

In ESP32 Arduino, there is an important functions for ADC—the `analogRead` function. The `analogRead` function is defined as

```
uint16_t analogRead(uint8_t pin);
```

which is used to read the analog input. It takes an input argument—the analog input pin. This function is defined in the ESP32 Arduino core in the `esp32-hal-adc.h` and `esp32-hal-adc.c`.

6.4 Example Circuit: Voltage Divider

A voltage divider is a simple circuit, which divides a higher voltage into a lower one. It consists of two series resistors. An input voltage is applied across the series of two resistors, and the output voltage is between the two resistors, which is a fraction of the input voltage. Figure 6.2(a) shows an example of a voltage divider circuit. Let V_{in} be input voltage and V_{out} be output voltage. The relationship between V_{in} and V_{out} is given by

$$V_{out} = V_{in} \frac{R2}{R1 + R2} \quad (6.1)$$

The V_{out} is directly proportional to the V_{in} and the ratio of $R1$ and $R2$.

Voltage dividers are used in many applications. For example, a potentiometer, which is a variable resistor that can be used to create an

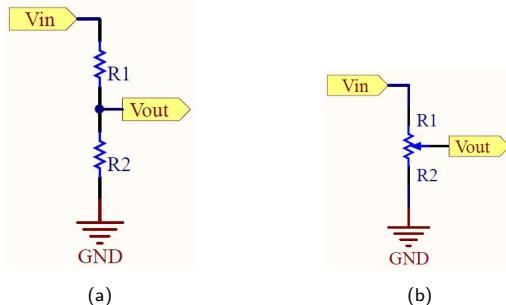


Figure 6.2. Voltage divider: (a) a example of a voltage divider circuit; (b) an adjustable voltage divider.

adjustable voltage divider. The internal of a potentiometer is a resistor that has a wiper. The wiper can be moved to adjust the ratio between both halves as shown in Figure 6.2(b). If a potentiometer is turned all the way in one direction, the output voltage may be zero; turned to the other side, the output voltage approaches the input voltage.

6.5 Example Code: Read a Trimpot

In this example code, we are going to read analog voltage from a trimmer potentiometer (trimpot¹) by using the ADC. The output pin of the trimpot is connected to the pin D27, while the other two pins of the trimpot are connected to the VCC and GND, respectively. A trimpot has no anode or cathode pin, so it doesn't matter which end you connect to VCC and GND. The only difference is the direction of rotation, clockwise to increase the output voltage and counterclockwise to decrease the output voltage, or vice versa. The code is shown in Listing 6.1.

¹ A trimpot is the same as potentiometer. The only key difference between a trimpot and a potentiometer is size and shape. Trimpots usually have smaller form factor compared to potentiometers.

Listing 6.1. Read a trimpot value

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 8 Mar 2020
3
4 void setup()
5 {
6     // Initialize serial port
7     Serial.begin(115200);
8 }
9
10 void loop()
11 {
12     // Read trimpot value using ADC
13     int trimpot = analogRead(27);
14     // Print the value to the serial monitor
15     Serial.println(trimpot);
16     delay(1000);
17 }
```

Here is the step-by-step to read a trimpot with ADC:

1. First, you need to configure UART with a baud rate of 115200 as in **line 7**.
2. Then, you need to read the ADC by using the `analogRead` function as in **line 13**. You need to pass the pin number (27) that is connected to the trimpot as input argument of the function.
3. Finally, in **line 15**, you need to send the trimpot value to the serial monitor, and also you need to add a delay of one second as in **line 16**.

You can open the serial monitor to check the trimpot value. As you turn the trimpot clockwise or counterclockwise, the trimpot value will



Figure 6.3. Serial plotter for plotting the trimpot value.

be changed. You can also use Arduino serial plotter to plot the trimpot value as shown in Figure 6.3.

6.6 Summary

In this chapter, you have learned how to read an analog input with ADC. The input voltage range of the ESP32 ADC is between 0V to 3.3V that corresponds to the discrete values between 0 to 4095 in the default 12-bit mode.

Chapter 7

Sensors

What will you learn in this chapter?

- Concept of sensors.
- Use the DHT11 digital temperature and humidity sensor.
- Use the DS1307 real-time clock.

7.1 Sensing Physical Properties

In the real world, there are many physical properties. Examples of physical properties include: temperature, light, mass, electric current, magnetic field, etc. These physical properties can't be sensed directly using microcontrollers. We need to use additional components called sensor. A sensor converts a particular physical property to electric voltage. For example, a temperature sensor converts room temperature to voltage, for example between 0V and 3.3V.

There are many sensors available today. Many of them come as modules (they have extra electronic circuitry along with the sensor) as

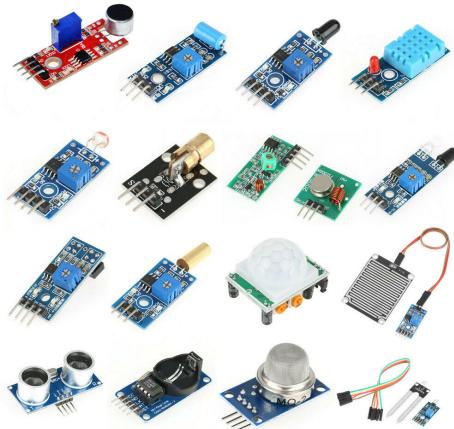


Figure 7.1. Sensor modules. Retrieved April 11, 2020, from ebay.com.

shown in Figure 7.1. Here is the list of several popular sensor modules:

- **Color sensors:** TCS3200, TCS34725.
- **Gas sensors:** MQ2, MQ3, MQ4, MQ5, MQ6, MQ7, MQ8, MQ9, MQ135.
- **Light sensors:** photoresistor, photodiode.
- **Passive infrared sensors:** HC-SR501, HC-SR505.
- **Real-time clock (RTC) sensors:** DS1307, DS3231.
- **Temperature sensors:** LM35, DS18B20, DHT11, DHT22.
- **Ultrasonic distance sensors:** HC-SR04, HY-SRF05.

Some of these sensors use digital interface, such as 1-wire, I2C, SPI, etc. In this case, you don't need to use ADC. However, there are also sensors that use analog interface, so you do need to use ADC.

7.2 Example Code: DHT11 Sensor

DHT11 is a temperature and humidity sensor. This sensor is quite slow, but very low cost, so it is great for hobbyists. Here are the specifications [16]:

- 3 to 5V power and I/O
- 2.5mA max current use during conversion (while requesting data)
- Good for 20-80% humidity readings with 5% accuracy
- Good for 0-50°C temperature readings $\pm 2^\circ\text{C}$ accuracy
- No more than 1 Hz sampling rate (once every second)
- Body size 15.5mm x 12mm x 5.5mm
- 4 pins with 0.1" spacing

This sensor can either be purchased as a sensor or as a module. It uses just one wire to transmit digital data to the microcontroller. There is a library [17] for this sensor in Arduino, so you don't have to deal directly with the communication protocol.

The connection between DHT11 and ESP32 is described in Table 7.1. Here the DATA pin of the DHT11 is connected to pin D22, but you can connect it to any other digital pin.

In this example code, we are going to read the temperature and humidity values from the DHT11 sensor. Then, the values are printed to the serial terminal. The code is shown in Listing 7.1.

Table 7.1. The connection between DHT11 and ESP32.

DHT11 Pins	ESP32 Pins
VCC (1)	3V3
DATA (2)	D22
NC (3)	-
GND (4)	GND

Listing 7.1. Read temperature and humidity from DHT11

```

1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 23 Feb 2020
3
4 #include <DHT.h>
5
6 #define DHTPIN 22
7 #define DHTTYPE DHT11
8
9 // Setup DHT pin and type
10 DHT dht(DHTPIN, DHTTYPE);
11
12 void setup()
13 {
14     // Initialize serial port
15     Serial.begin(115200);
16     // Initialize DHT sensor
17     dht.begin();
18 }
19
20 void loop()
21 {
22     // Read temperature
23     float temp = dht.readTemperature();
24     // Read humidity
25     float hum = dht.readHumidity();
26 }
```

```
27 // Print temperature
28 Serial.print("Temperature: ");
29 Serial.print(temp);
30 Serial.print("C ");
31 Serial.print("Humidity: ");
32 Serial.print(hum);
33 Serial.println("%");
34
35 // Maximum sampling rate of DHT11 is 1Hz
36 // So, the minimum delay is one second
37 delay(1000);
38 }
```

Here is the step-by-step to read the DHT11:

1. First, you need to install the DHT library [17], and import the library as in **line 4**.
2. Then, you need to define the data pin and DHT type as in **line 6 and 7**.
3. After that, you need to instantiate the DHT class as in **line 10**. It takes two input arguments: `DHTPIN` and `DHTTYPE`.
4. Next, in **line 17**, you need to call `begin` method to initialize the DHT sensor.
5. Then, in **line 23 and 25**, you need to read the temperature and humidity by calling `readTemperature` and `readHumidity` methods, respectively.
6. Finally, in **line 28-33**, you need to print the temperature and humidity to the serial monitor.

Table 7.2. The connection between DS1307 and ESP32.

DS1307 Pins	ESP32 Pins
VCC	3V3
SCL	D32
SDA	D33
GND	GND

7.3 Example Code: DS1307 Real-Time Clock

DS1307 is a real-time clock (RTC) that is used to keep track of time. It can accurately keep track of seconds, minutes, hours, days, months, and years. Here are the specifications [18]:

- 56-byte, battery-backed, nonvolatile (NV) RAM for data storage
- Two-wire (I2C) serial interface
- Programmable squarewave output signal
- Automatic power-fail detect and switch circuitry
- Consumes less than 500nA in battery backup mode with oscillator running

It uses the I2C protocol to control the module. There is a library [19] for this module in Arduino, so you don't have to deal directly with the communication protocol.

The connection between DS1307 and ESP32 is described in Table 7.2. The SCL (clock) and SDA (data) pins are the I2C pins.

In this example code, we are going to read timestamp from the DS1307. Then, the timestamp is printed to the serial terminal. The code is shown in Listing 7.2.

Listing 7.2. Read timestamp from DS1307

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 8 Mar 2020
3
4 #include <Wire.h>
5 #include <RtcDS1307.h>
6
7 // I2C pins
8 #define I2C_SDA 33
9 #define I2C_SCL 32
10
11 // RTC object declaration
12 RtcDS1307<TwoWire> rtc(Wire);
13
14 void setup()
15 {
16     // Initialize serial port
17     Serial.begin(115200);
18
19     // Initialize I2C pins
20     Wire.begin(I2C_SDA, I2C_SCL, 400000);
21
22     // Initialize RTC
23     rtc.Begin();
24     // *** Set RTC date and time to code compiled time ***
25     RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);
26     rtc.SetDateTime(compiled);
27     rtc.SetIsRunning(true);
28 }
29
30 void loop()
31 {
32     // Read RTC date and time
33     RtcDateTime now = rtc.GetDateTime();
34     // Print RTC date and time to serial monitor
35     Serial.printf("%04d/%02d/%02d %02d:%02d:%02d\n",
```

```
36     now.Year(), now.Month(), now.Day(),
37     now.Hour(), now.Minute(), now.Second());
38   delay(1000);
39 }
```

Here is the step-by-step to read the DS1307:

1. First, you need to install the RTC library [19], and import the library along with the Wire (I2C) library as in **line 4 and 5**.
2. Then, you need to instantiate the `RtcDS1307` class as in **line 12**. It takes an input argument—the `Wire` object.
3. After that, you need to initialize the I2C by calling the `begin` method as in **line 20**. It takes three input arguments: data pin, clock pin, and I2C clock frequency.
4. Next, in **line 25-27**, you need to set the DS1307's time to the compiled time, and then start the DS1307. The `SetIsRunning` method is used to start or stop the DS1307.
5. Finally, in **line 33-38**, you need to read the current timestamp and print it to the serial monitor every 1 second.

7.4 Summary

In this chapter, you have learned about sensors. You have learned the DHT11 temperature and humidity sensor. You have also learned the DS1307 RTC. Both of them use digital interface to the microcontroller.

Part II

Wi-Fi Programming

Chapter 8

Wi-Fi Access Point

What will you learn in this chapter?

- Concept of Wi-Fi networking and Wi-Fi access point.
- Connect a Wi-Fi device to an ESP32 access point.

8.1 Wi-Fi Network

Wi-Fi network is a wireless networking technology that enables multiple devices, such as computers, laptops, tablets, smartphones, gaming devices, televisions, etc., to connect to the Internet and each other. It is the easiest way to setup an Internet network without obtrusive cables. However, in theory, Wi-Fi network is less secure than the Ethernet network. Because in Wi-Fi network, the data travel through the air, it is possible to be intercepted by attacker. In Ethernet network, it is impossible to do this attack because attacker need physical access to do so.

Figure 8.1 shows an example of a Wi-Fi network. Laptops and phones

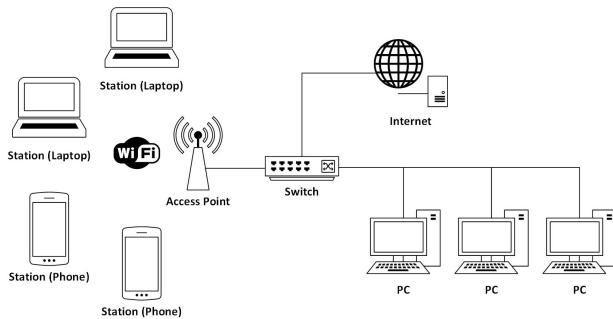


Figure 8.1. An example of Wi-Fi network.

connect to a Wi-Fi network via an access point (AP). A Wi-Fi network consists of an AP and multiple stations.

- **Access point:** a device that allows Wi-Fi devices (stations) to connect to each other, to a wired network, and to the Internet.
- **Station:** devices with Wi-Fi capability, such as computers, laptops, tablets, smartphones, gaming devices, televisions, etc.

Todays, many APs come with integrated router and modem. These are called wireless modem routers. Some of these also supports Internet connection via 3G/4G network.

Beside that, there is also a technology called wireless ad hoc network. In ad hoc network, the stations can connect to each other without using an AP. For example, it can be used for peer-to-peer file transfer or multiplayer video game. The wireless ad hoc network is beyond the scope of this book.

8.2 Access Point

A access point (AP) is a device in Wi-Fi network that allows Wi-Fi stations to connect to each other, to a wired network, and to the Internet. The stations connect to each other via this AP, rather than directly to each other. There is also a term called soft AP. It is an abbreviated term for software enabled AP. The software enables a station which hasn't been specifically made to be an AP to become an AP.

8.3 ESP32 Wi-Fi Networking: Access Point

ESP32 Wi-Fi implements TCP/IP, MAC, baseband, and radio. It supports the IEEE 802.11 b/g/n specifications. The Wi-Fi supports three modes:

- **AP mode:** this mode is also called soft AP mode. In this mode, stations connect to the ESP32.
- **Station mode:** this mode is also called STA mode or Wi-Fi client mode. In this mode, the ESP32 connects to an AP.
- **Combined AP-STA mode:** in this mode, the ESP32 is both an AP and a station connected to another AP.

By configuring the ESP32 in AP mode, any Wi-Fi devices can connect directly to the ESP32 without the need of another Wi-Fi AP. This mode is shown in Figure 8.2.

In this chapter, we are going to focus on the AP mode. We will get into the station mode in the next chapter. First of all, in order to program the ESP32 Wi-Fi using ESP32 Arduino core, you should include the `WiFi.h` library:

```
#include <WiFi.h>
```

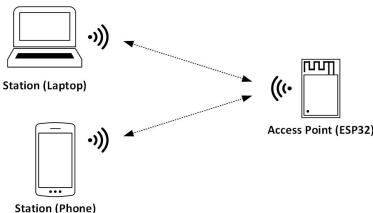


Figure 8.2. ESP32 in AP mode.

There are two important methods to setup ESP32 as AP: `softAP` and `softAPConfig`. The `softAP` method is defined as

```
bool softAP(const char* ssid,
            const char* passphrase = NULL,
            int channel = 1,
            int ssid_hidden = 0);
```

This method is used to start the ESP32 in AP mode. It takes four input arguments:

- `ssid`: SSID name (AP name). Maximum of 63 characters.
- `passphrase`: AP password. Minimum of 8 characters.
- `channel`: Wi-Fi channel number (1-13).
- `ssid_hidden`: 0 = broadcast SSID, 1 = hide SSID.

The last three input arguments are optional because they have default arguments. So, it means that this method can be called with just providing one input argument—the SSID name.

The `softAPConfig` method is defined as

```
bool softAPConfig(IPAddress local_ip,
                  IPAddress gateway,
```

```
 IPAddress subnet);
```

This method is used to set the IP address, gateway, and subnet of the ESP32. It takes those parameters as the input arguments. By default, ESP32 uses the default IP address (192.168.4.1).

The ESP32 AP methods are defined in the ESP32 Arduino core in the WiFiAP.h and WiFiAP.c.

8.4 Example Code: Access Point Mode

In this example code, you are going to set the ESP32 in AP mode. Then, you print (to serial terminal) how many stations that are connected to this AP. The code is shown in Listing 8.1.

Listing 8.1. Wi-Fi AP mode

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 29 Feb 2020
3
4 #include <WiFi.h>
5
6 const char ssid[] = "ESP32-AP";
7 const char pass[] = "esp32accesspoint";
8
9 IPAddress ip(192,168,1,1);
10 IPAddress gateway(192,168,1,1);
11 IPAddress subnet(255,255,255,0);
12
13 int currentNumOfStation;
14
15 void setup()
16 {
17     // Setup serial communication
18     Serial.begin(115200);
19
20     // *** Create a WiFi access point ***
```

```
21   Serial.print("Setting AP ... ");
22   Serial.println(
23     WiFi.softAP(ssid, pass) ? "Ready" : "Failed!");
24   Serial.print("Setting AP configuration ... ");
25   Serial.println(
26     WiFi.softAPConfig(ip, gateway, subnet) ?
27       "Ready" : "Failed!");
28
29 // *** Print WiFi AP configuration ***
30 Serial.printf("IP address: %s\n",
31   WiFi.softAPIP().toString().c_str());
32 Serial.printf("MAC address: %s\n",
33   WiFi.softAPmacAddress().c_str());
34 }
35
36 void loop()
37 {
38 // *** Print number of stations that are
39 //      connected to AP ***
40 if (WiFi.softAPgetStationNum() != currentNumOfStation)
41 {
42   currentNumOfStation = WiFi.softAPgetStationNum();
43   Serial.printf("Stations connected to AP: %d\n",
44     currentNumOfStation);
45 }
46 }
```

Here is the step-by-step to set the ESP32 in AP mode:

1. First, you need to include the library `WiFi.h` as in **line 4**.
2. Then, you need to define your AP name and password as in **line 6-7**. You can modify the name and password to whatever you want.
3. After that, you need to define your IP addresses as in **line 9-11**.
4. Next, you need to pass the AP name and password as input arguments of the `softAP` method as in **line 22-23**. If this method returns `true`, it means that the AP name and password are successfully configured.
5. Next, you need to pass the IP addresses as the input arguments of the `softAPConfig` method as in **line 25-27**. It returns `true`, if the IP addresses are successfully configured.
6. Then, in **line 30-33**, you need to call `softAPIP` and `softAPmacAddress` methods in order to print the ESP32's current IP and MAC address.
7. Finally, in **line 40-45**, you can get the number of stations that are connected to ESP32 by calling `softAPgetStationNum` method. We store the number of stations in variable `currentNumOfStation`. The value of this variable is updated and printed only if it changes.

Note that the `WiFi` object is never instantiated in this file. This is because the `WiFi` is an external object that is already instantiated in the `WiFi.h` file with `extern` keyword. Figure 8.3 shows the output logs of this example code in serial terminal. It shows two stations are connected.

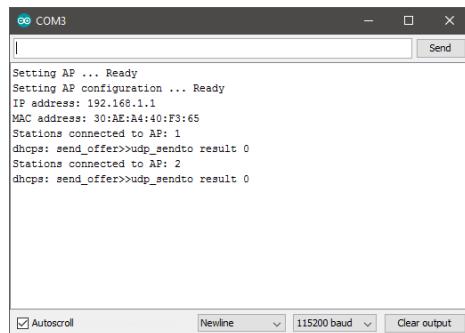


Figure 8.3. An example of ESP32 AP logs in serial terminal.

8.5 Summary

In this chapter, you have learned how to set ESP32 in AP mode. It allows stations, such as laptops or smartphones, to connect to the ESP32.

Chapter 9

Wi-Fi Station

What will you learn in this chapter?

- Concept of Wi-Fi station.
- Connect an ESP32 station to a Wi-Fi access point.

9.1 Station

The previous chapter explained the concept of Wi-Fi network. A Wi-Fi network can consist of an AP and multiple stations. The AP is a device that allows stations to connect to each other. The stations can be any devices that have Wi-Fi capability, such as computers, laptops, tablets, smartphones, gaming devices, televisions, etc. Today, many smart home appliances also have Wi-Fi capability. These stations connect to each other via the AP, rather than directly to each other.

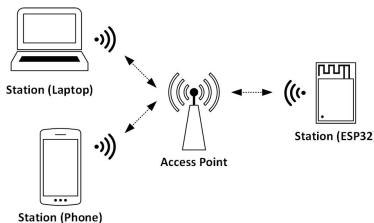


Figure 9.1. ESP32 in station mode.

9.2 ESP32 Wi-Fi Networking: Station

ESP32 Wi-Fi implements TCP/IP, MAC, baseband, and radio. It supports the IEEE 802.11 b/g/n specifications. The Wi-Fi supports three modes:

- **AP mode:** this mode is also called soft AP mode. In this mode, stations connect to the ESP32.
- **Station mode:** this mode is also called STA mode or Wi-Fi client mode. In this mode, the ESP32 connects to an AP.
- **Combined AP-STA mode:** in this mode, the ESP32 is both an AP and a station connected to another AP.

In this chapter, we are going to focus on the station mode. In station mode, you need to connect the ESP32 station to a Wi-Fi AP. The topology is shown in Figure 9.1. The ESP32 station can talk to other stations via the AP.

First of all, in order to program the ESP32 Wi-Fi using ESP32 Arduino core, you should include the `WiFi.h` library:

```
#include <WiFi.h>
```

There are three important methods to setup ESP32 as station: `mode`, `begin`, and `waitForConnectResult`. The `mode` method is defined as

```
bool mode(wifi_mode_t m);
```

This method is used to set the mode of ESP32. It takes one input argument `m`, which is the ESP32 mode (`WIFI_OFF`, `WIFI_STA`, `WIFI_AP`, or `WIFI_AP_STA`). It returns `true` when the ESP32 is successfully configured.

The `begin` method is defined as

```
wl_status_t begin(char* ssid,  
                  char *passphrase = NULL,  
                  int32_t channel = 0,  
                  const uint8_t* bssid = NULL,  
                  bool connect = true);
```

This method is used to start the ESP32 in station mode. It takes five input arguments:

- `ssid`: SSID name of AP.
- `passphrase`: Password of AP (optional).
- `channel`: Wi-Fi channel of AP (optional).
- `bssid`: BSSID/MAC of AP(optional).
- `connect`: 0 = don't connect to AP, 1 = connect to AP (optional).

This method can be called by providing just one input argument, which is the SSID name of AP, because the other input arguments are optional. It returns Wi-Fi connection status, which is defined as

```
typedef enum {
    WL_NO_SHIELD      = 255,
    WL_IDLE_STATUS     = 0,
    WL_NO_SSID_AVAIL   = 1,
    WL_SCAN_COMPLETED   = 2,
    WL_CONNECTED        = 3,
    WL_CONNECT_FAILED   = 4,
    WL_CONNECTION_LOST   = 5,
    WL_DISCONNECTED     = 6
} wl_status_t;
```

The `waitForConnectResult` method is defined as

```
uint8_t waitForConnectResult();
```

This method is used to wait for Wi-Fi connection to reach a result. It has no input argument, and returns one of the value defined in `wl_status_t`.

The ESP32 station methods are defined in the ESP32 Arduino library in the `WiFiSTA.h` and `WiFiSTA.c`.

9.3 Example Code: Station Mode

In this example code, you are going to set the ESP32 in station mode. Then, you connect it to an AP. The code is shown in Listing 9.1.

Listing 9.1. Wi-Fi station mode

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 29 Feb 2020
3
4 #include <WiFi.h>
5
6 const char ssid[] = "Huawei-E5573";
7 const char pass[] = "huaweiie5573";
8
```

```
9 void setup()
10 {
11     // Setup serial communication
12     Serial.begin(115200);
13
14     // *** Connect to a WiFi access point ***
15     Serial.printf("Connecting to %s ... \n", ssid);
16     WiFi.mode(WIFI_STA);
17     WiFi.begin(ssid, pass);
18     if (WiFi.waitForConnectResult() != WL_CONNECTED)
19     {
20         Serial.printf("WiFi connect failed! Rebooting ... \n");
21         delay(1000);
22         ESP.restart();
23     }
24     Serial.printf("Connected\n");
25
26     // *** Print WiFi station configuration ***
27     Serial.printf("SSID: %s\n", WiFi.SSID().c_str());
28     Serial.printf("RSSI: %d dBm\n", WiFi.RSSI());
29     Serial.printf("IP address: %s\n", WiFi.localIP()
30                 .toString().c_str());
31     Serial.printf("Subnet mask: %s\n", WiFi.subnetMask()
32                 .toString().c_str());
33     Serial.printf("Gateway: %s\n", WiFi.gatewayIP()
34                 .toString().c_str());
35     Serial.printf("MAC address: %s\n", WiFi.macAddress()
36                 .c_str());
37 }
38
39 void loop()
40 {
41 }
```

Here is the step-by-step to set the ESP32 in station mode:

1. First, you need to include the library `WiFi.h` as in **line 4**.
2. Then, you need to define the AP name and password as in **line 6-7**. You should modify the name and password to your Wi-Fi AP's or router's.
3. After that, you need to set the ESP32 in station mode by calling `mode` method as in **line 16**. Note that in the previous chapter when you set the ESP32 in AP mode, you don't have to call the `mode` method because the default mode of the ESP32 is soft AP.
4. Next, you need to pass the AP name and password as input arguments of `begin` method as in **line 17**.
5. Then, in **line 18-23**, you need to wait until the ESP32 is connected to the AP by calling the `waitForConnectResult` method. If the ESP32 is not connected yet after 1 second, then it will be restarted. If necessary, you can change the delay.
6. Finally, in **line 27-36**, you can print the Wi-Fi station configuration, such as SSID, IP address, MAC address, etc.

You can use this example code as bare minimum code for configuring the ESP32 in station mode. Figure 8.3 shows the output logs of this example code in serial terminal.

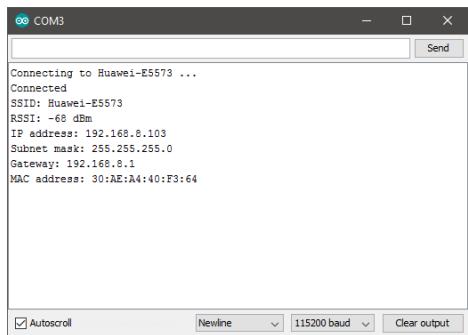


Figure 9.2. An example of ESP32 station logs in serial terminal.

9.4 Summary

In this chapter, you have learned how to set ESP32 in station mode. It is connected to an AP. It can talk to other stations via the AP.

Chapter 10

TCP Server

What will you learn in this chapter?

- Concept of TCP/IP model and TCP server.
- Create a TCP server on an ESP32 station.

10.1 Internet Protocol Suite

Internet protocol suite or commonly known as TCP/IP is a model used in computer network. The model consists of communication protocols that are grouped into several layers. The foundational protocols are Transmission Control Protocol (TCP) and Internet Protocol (IP). This is the reason why it is called TCP/IP model. The complete layers of the model are defined as follows:

- **Network interface:** this is the first or the lowest layer of the model. This layer converts the data into transmittable format. So, it can be transmitted over physical communication media

wired or wirelessly. Moreover, this layer is separated into two layers: data link and physical.

The physical layer (PHY) is responsible to convert digital bits (raw data) into electrical, radio, or optical signals. This layer is usually implemented by a PHY chip. Examples of protocols present in this layer are 1-wire, Bluetooth, CAN bus, Ethernet PHY, I2C, LoRa, RS232, SPI, USB PHY, Wi-Fi PHY, etc.

The data link layer is concerned with reliable transmission of data frames between two nodes connected by a physical layer. This layer may be implemented by a chip. Examples of protocols present in this layer are Ethernet Media Access Control (MAC) (IEEE 802.3), Wi-Fi MAC (IEEE 802.11), ZigBee MAC (IEEE 802.15.4), Point-to-Point Protocol (PPP), etc.

- **Internet:** this is the second layer of the model. This layer is responsible for managing a multi-node network. This layer is concerned with transmission of data packet through a multi-node network including addressing, routing, and traffic control. Each node on a network has a unique address called Internet Protocol (IP) address. So, the transmission of data packet can be done by providing the content of packet and the address of the destination node. The packet is also possibly routed through intermediate routers. Examples of protocols present in this layer are Internet Protocol (IPv4/IPv6), Internet Control Message Protocol (ICMP), etc.
- **Transport:** this is the third layer of the model. This layer provides host-to-host communication service for applications. This layer is responsible for providing a reliable communication service between applications through flow control, segmentation/desegmentation, and error control. This layer also provides multiplexing (port), so that a single node can have multiple endpoints. Examples of protocols present in this layer are Transmission Control Protocol (TCP), User Datagram Protocol (UDP), etc.

- **Application:** this is the fourth layer of the model. This layer provides communication protocols used by applications for exchanging data over the network. Examples of protocols present in this layer are File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), Secure Shell (SSH), MQ Telemetry Transport (MQTT), Modbus, etc.

Furthermore, there are other networking models that model the computer network, such as OSI model (seven layers), TCP/IP 5-layer reference model, etc. The number of layers varies between three and seven.

10.2 TCP Server

TCP connections work in client-server model. The server creates a TCP socket with a specified port number and waits for a connection request from the client. Once a connection has been established, data can be sent in either direction. The connection remains open until either the client or server terminates the connection.

10.3 ESP32 TCP Server

In this chapter, we are going to focus on the TCP server. We will get into the TCP client in the next chapter. A TCP server creates a TCP socket that listen to a specified port and waits for a connection request from the client.

First of all, in order to use TCP server in ESP32 Arduino core, you should include the `WiFi.h` library:

```
#include <WiFi.h>
```

In ESP32 Arduino Wi-Fi library, TCP server class is named as `WiFiServer`. The constructor¹ of this class is defined as

```
WiFiServer(uint16_t port=80, uint8_t max_clients=4) :  
    sockfd(-1),  
    _port(port),  
    _max_clients(max_clients),  
    _listening(false){}
```

The default port of TCP server is 80, and the maximum number of clients is four.

There are two important methods to setup a TCP server in ESP32: `begin` and `available`. The `begin` method is defined as

```
void begin();
```

This method is used to start the TCP server to listen on a specified port.

The `available` method is defined as

```
WiFiClient available();
```

This method is used to accept a connection request from a client. Once the connection has been accepted, it returns a TCP client object which is named as `WiFiClient`.

The ESP32 TCP server methods are defined in the ESP32 Arduino library in the `WiFiServer.h` and `WiFiServer.c`.

10.4 Example Code: TCP Server

In this example code, you are going to create a TCP server for controlling an LED from TCP client by sending specified commands. First,

¹ A constructor is a method of a class which initializes the object of the class. It has same name as the class itself.

you need to set the ESP32 in station mode. Then, you connect it to an AP. Next, you need to create a TCP server that listen on port 23. After that, you need to listen to an incoming command from the client. Finally, you need to process the command. The code is shown in Listing 10.1.

Listing 10.1. TCP server

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 1 Mar 2020
3
4 #include <WiFi.h>
5
6 // On-board LED's GPIO pin
7 #define LED_ON_BOARD 2
8
9 const char ssid[] = "Huawei-E5573";
10 const char pass[] = "huaweie5573";
11
12 // TCP server port
13 const uint16_t port = 23;
14
15 // TCP server object declaration
16 WiFiServer server(port);
17
18 void setup()
19 {
20     // Set GPIO pin as output for on-board LED
21     pinMode(LED_ON_BOARD, OUTPUT);
22     // Setup serial communication
23     Serial.begin(115200);
24
25     // *** Connect to a WiFi access point ***
26     Serial.printf("Connecting to %s ...\\n", ssid);
27     WiFi.mode(WIFI_STA);
28     WiFi.begin(ssid, pass);
29     if (WiFi.waitForConnectResult() != WL_CONNECTED)
```

```
30     {
31         Serial.printf("WiFi connect failed! Rebooting ...\\n");
32         delay(1000);
33         ESP.restart();
34     }
35     Serial.printf("Connected\\n");
36     Serial.printf("IP address: %s\\n", WiFi.localIP()
37         .toString().c_str());
38
39     // *** Start TCP server ***
40     server.begin();
41     Serial.printf("TCP server started at port %d\\n", port);
42 }
43
44 void loop()
45 {
46     // Get TCP client
47     WiFiClient client = server.available();
48
49     // If a client connected
50     if (client)
51     {
52         Serial.printf("Client connected\\n");
53         while (client.connected())
54         {
55             if (client.available())
56             {
57                 // Read command from receive buffer
58                 String cmd = client.readStringUntil('\\n');
59
60                 // *** Process the command ***
61                 if (cmd == "LED+ON")
62                 {
63                     // Turn on LED
64                     digitalWrite(LED_ON_BOARD, HIGH);
65                 }
66                 else if (cmd == "LED+OFF")
```

```
67      {
68          // Turn off LED
69          digitalWrite(LED_ON_BOARD, LOW);
70      }
71      else if (cmd == "LED")
72      {
73          // Query the LED value
74          if (digitalRead(LED_ON_BOARD))
75          {
76              client.println("LED is on");
77          }
78          else
79          {
80              client.println("LED is off");
81          }
82      }
83      else
84      {
85          Serial.println("Unknown command");
86      }
87  }
88 }
89 Serial.printf("Client disconnected\n");
90 }
91 }
```

Here is the step-by-step to create a TCP server:

1. First, you need to define a port number for the TCP server, and then you need to pass it into the `WiFiServer` object as in **line 13 and 16**.
2. Then, you need to set the GPIO for on-board LED, and then you need to start the serial I/O as in **line 21 and 23**.
3. After that, in **line 26 and 37**, you need to set the ESP32 as a station and connect it to an AP.

4. Next, in **line 40**, you need to start the TCP server to listen to the port 23 by calling the `begin` method.
5. Then, in **line 47**, you need to accept a connection request from a client by calling the `available` method. It returns the `client` object.
6. After that, in **line 53-88**, while the client is connected, the server always waits for a command, then processes it.
7. The `available` method in **line 55** is used to return the number of characters which have arrived in receive buffer.
8. In **line 58**, you need to read the characters from receive buffer until you find the `\n` character, which is the newline character that comes at the end of every command.
9. Finally, in **line 61-86**, you need to process the command. There are three defined commands: `LED+ON`, `LED+OFF`, and `LED`. The first two commands are used for turning on and off the LED, respectively. The last command is used for reading the current LED state.

Since you set the ESP32 in station mode, you need to connect the ESP32 to an AP. Then, from another station e.g. your laptop/PC, you need to make a connection request to the ESP32. In order to make a connection request, you need to use a TCP client software tool. I use a software tool named Hercules SETUP utility. It is a freeware application that you can download from <https://www.hw-group.com/software/hercules-setup-utility>.

From Hercules SETUP utility, you can send the command to control the LED or read the current state of the LED. You need to end every command with the newline character, which is defined as `<LF>` in this tool. Figure 10.1 shows the logs of the TCP client.

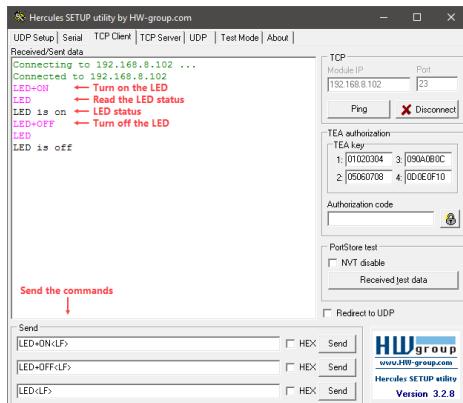


Figure 10.1. An example of TCP client using Hercules SETUP utility.

10.5 Summary

In this chapter, you have learned how to create a TCP server in the ESP32. The server allows a connected TCP client to be able to control the LED and read the current state of the LED. So, both the server and client can send and receive data.

Chapter 11

TCP Client

What will you learn in this chapter?

- Concept of TCP client.
- Create a TCP client on an ESP32 station.

11.1 TCP Client

The previous chapter explained the concept of TCP/IP model. It consists of four layers: network interface, internet, transport, and application. In case of ESP32, the network interface layer corresponds to the Wi-Fi PHY (baseband) and Wi-Fi MAC. They are implemented as hardware on ESP32 chip. The internet and transport layers correspond to the IP and TCP, respectively. They are implemented as software libraries that run on the processor of ESP32. Most of the time, we work on the transport layer (TCP) or the application layer. We will get into the application layer in the next chapter. In this chapter, we are going to focus on the TCP client.

The TCP server creates a TCP socket with a specified port number and waits for a connection request from the client. So, the client should make a connection request to the server. Once the connection has been accepted, both client or server can send and receive data. The connection remains open until either the client or server terminates the connection.

11.2 ESP32 TCP Client

In this chapter, we are going to focus on the TCP client. A TCP client connects to a TCP server that listen on a specified IP address and port number.

First of all, in order to use TCP client in ESP32 Arduino core, you should include the `WiFi.h` library:

```
#include <WiFi.h>
```

In ESP32 Arduino Wi-Fi library, the TCP client class is named as `WiFi Client`. There are two important methods for using TCP client in ESP32: `connect` and `stop`. The `connect` method is defined as

```
int connect(IPAddress ip, uint16_t port);
int connect(IPAddress ip, uint16_t port,
            int32_t timeout);
```

This method is used to connect the TCP client to a TCP server that listens on a specified IP address and port number. This method has two implementations with the same name, but the input arguments are different. The ability to create multiple functions/methods of the same name with different implementations is called function/method overloading. The first implementation takes two input arguments: IP address and port number of the server. The second implementation takes three input arguments: IP address, port number, and timeout. Once the client is connected to the server, these method return one.

The `stop` method is defined as

```
void stop();
```

which is used to terminate the connection to the server.

The ESP32 TCP client methods are defined in the ESP32 Arduino library in the `WiFiClient.h` and `WiFiClient.c`.

11.3 Example Code: TCP Client

In this example code, you are going to create a TCP client that connects to a TCP server. After the connection between the client and server has been established, the client sends some data to the server. In the server, you need to have a feature for echoing the received data. So, the received data will be retransmitted back to the client. The code is shown in Listing 11.1.

Listing 11.1. TCP client

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 1 Mar 2020
3
4 #include <WiFi.h>
5
6 const char ssid[] = "Huawei-E5573";
7 const char pass[] = "huaweiie5573";
8
9 // *** TCP server IP and port ***
10 const char ip[] = "192.168.8.101";
11 const uint16_t port = 23;
12
13 void setup()
14 {
15     // Setup serial communication
16     Serial.begin(115200);
17 }
```

```
18 // *** Connect to a WiFi access point ***
19 Serial.printf("Connecting to %s ...\\n", ssid);
20 WiFi.mode(WIFI_STA);
21 WiFi.begin(ssid, pass);
22 if (WiFi.waitForConnectResult() != WL_CONNECTED)
23 {
24     Serial.printf("WiFi connect failed! Rebooting ...\\n");
25     delay(1000);
26     ESP.restart();
27 }
28 Serial.printf("Connected\\n");
29 Serial.printf("IP address: %s\\n", WiFi.localIP()
30             .toString().c_str());
31 }
32
33 void loop()
34 {
35     // TCP client object declaration
36     WiFiClient client;
37
38     // *** Connect to a TCP server ***
39     Serial.printf("Connecting to %s ...\\n", ip);
40     if (!client.connect(ip, port))
41     {
42         Serial.printf("Connection failed\\n");
43         Serial.println("Wait 5 sec ...");
44         delay(5000);
45         return;
46     }
47     Serial.printf("Connected\\n");
48
49     // Send a message to the TCP server
50     client.printf("Message from TCP client\\n");
51
52     // *** Read an echo from the TCP server ***
53     String response = client.readStringUntil('\\n');
54     Serial.print("Echoed message: ");
```

```
55     Serial.println(response);
56
57     // *** Close TCP connection ***
58     Serial.printf("Closing connection\n");
59     client.stop();
60
61     Serial.println("Wait 5 sec ...");
62     delay(5000);
63 }
```

Here is the step-by-step to create a TCP client:

1. First, you need to define the IP address and port number of the TCP server as in **line 10 and 11**.
2. Then, you need to set the ESP32 as a WiFi station that connects to an AP as in **line 19-30**.
3. After that, you need to instantiate the `WiFiClient` class as in **line 36**.
4. Next, in **line 40-47**, you need to connect the client to the server by calling the `connect` method. If the connection request is failed, then it will retry after 5 seconds.
5. Once the connection has been established, you can send a message to server by calling `printf` method as in **line 50**.
6. Then, you need to wait for the echoed message from the server by calling `readStringUntil` method as in **line 53**. The echoed message will be printed to serial terminal.
7. Finally, in **line 59**, you need to call the `stop` method to terminate the connection.

You need to connect your ESP32 to an AP. Then, from another station e.g. your laptop/PC, you need to start a TCP server. As an example,

I use the Hercules SETUP utility to create a TCP server that listens on port 23, and the server echo option is enabled as shown in Figure 11.1.

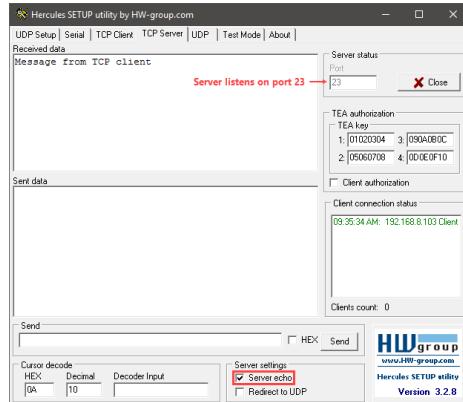


Figure 11.1. A TCP server using Hercules SETUP utility.

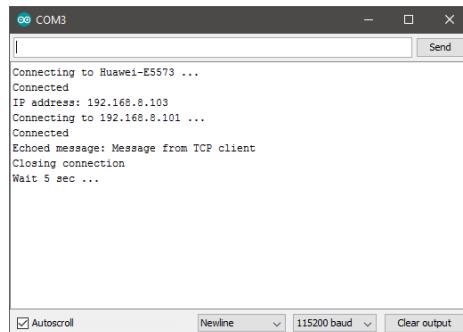


Figure 11.2. An example of client logs in serial terminal.

The client will connect to the server, and it will send a message. After

that, the message will be retransmitted back to the client. Finally, the client terminates the connection. The client logs are printed to serial terminal as shown in Figure 11.2.

11.4 Summary

In this chapter, you have learned how to create a TCP client in the ESP32. The client connects to a TCP server. Both the client and server can send and receive data as long as the connection is established.

Chapter 12

HTTP Server

What will you learn in this chapter?

- Concept of HTTP server.
- Create a simple HTTP server on an ESP32 station.

12.1 HyperText Transfer Protocol

HyperText Transfer Protocol (HTTP) is a fundamental application protocol for the world wide web (www). HyperText is a text displayed on a computer display with hyperlinks or simply links to other resources that the user can easily access by a mouse click or by tapping the screen in a web browser. The HTTP communication takes place over a TCP connection, which provides reliable transport service. The default TCP port for HTTP is 80.

HTTP is based on requests and responses in a client-server model. For example, a web browser may be the client, and an application running on a computer hosting a website (web server) may be the

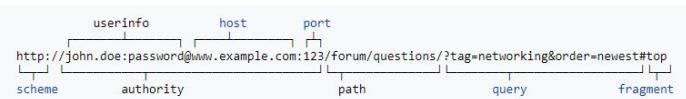


Figure 12.1. An example of a URL including all optional components. Retrieved April 5, 2020, from wikipedia.org.

server. The client sends an HTTP request message to the server. The server returns an HTTP response message to the client. The response message may contain resources, such as HTML files and other contents, or perform other functions on behalf of the client.

12.2 HTTP Server

An HTTP server is a piece of software that understands the HTTP and Uniform Resource Locator (URL) or web address. It can be addressed from the client (web browser) by using its URL. URL is a reference to a resource that specifies its location on a computer network and a mechanism for retrieving it. A example of a URL is shown in Figure 12.1.

This is an example of a typical URL: `http://www.example.com/index.html`. It indicates a protocol (HTTP), a host name (`www.example.com`), and a file name (`index.html`). In case of a server that runs on a local network, most likely the URL could be like this: `http://192.168.1.1/index.html`. It uses its IP address as the host name, because it doesn't have a domain name¹.

A block diagram of an HTTP server is shown in Figure 12.2. An HTTP server and its resources are called web server. We will get into the details of an embedded web server in the next part. A web browser accesses the web server which is identified by its URL. The

¹ A domain name is translated into its IP address by using a Domain Name Server (DNS).

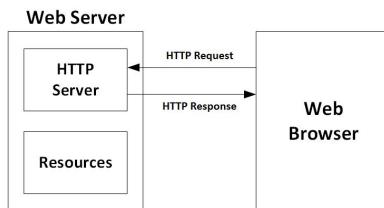


Figure 12.2. A block diagram of an HTTP server.

web browser sends HTTP request messages, and the server sends the HTTP response messages. From the hardware point of view, servers typically could be computer servers, PCs, or embedded devices. The client typically could be PCs, tablets, smartphones, or embedded devices. Sometimes, a device can be both client and server. When a device requests data from a server, it acts as a client. When a device provides data, it acts as a server.

12.3 Message Format

There are two types of HTTP messages: request which is sent by the client and response which is sent by the server. Examples of HTTP request and response messages are shown in Figure 12.3. HTTP requests and responses are composed of:

- **Start-line:** for request message, the start-line describes HTTP request method. For response message, the start-line describes HTTP response status.
- **HTTP headers:** HTTP headers are optional. They provide information about the request or response message, or about the object sent in the body.

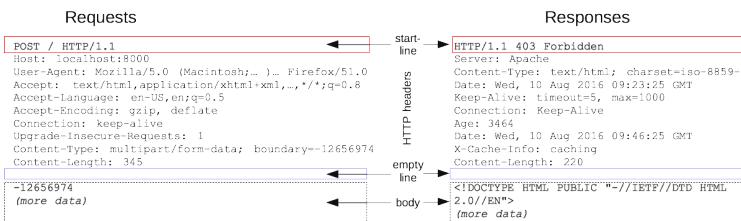


Figure 12.3. Example of HTTP messages. Retrieved April 5, 2020, from developer.mozilla.org.

- **Empty line:** the blank line indicates that all meta-information of the message has been sent.
- **Body:** the body part is optional. It contains data associated with the request, for example HTML files, images, or other files. The HTTP headers specify the content of the body.

There are several commonly used HTTP methods: GET, POST, PUT, and DELETE. They describe an action to be performed. There are several commonly used status codes: 200 (OK), 303 (see other), 403 (forbidden), and 404 (not found).

12.4 ESP32 HTTP Server

In this chapter, we are going to use only the TCP server class, i.e. the `WiFiServer` class of the ESP32 Arduino. The reason is that the HTTP communication takes place over TCP. So, you only need a TCP server runs on the ESP32, and you would be able to send and receive HTTP messages. By not using any HTTP server or web server library, you can understand how the HTTP request and response work. We will use and get into the details of a web server library in the next part.

12.5 Example Code: HTTP Server

In this example code, you are going to create a simple HTTP server on top of a TCP server. A client (web browser) can connect to this server. Then, it sends an HTTP request to the server to request a web page. The request is printed to serial monitor. After that, the server sends an HTTP response which contains a simple web page. Finally, the web browser displays the web page. The code is shown in Listing 12.1.

Listing 12.1. HTTP server

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 1 Mar 2020
3
4 #include <WiFi.h>
5
6 const char ssid[] = "Huawei-E5573";
7 const char pass[] = "huaweiie5573";
8
9 // TCP server port for HTTP
10 const uint16_t port = 80;
11
12 // TCP server object declaration
13 WiFiServer server(port);
14
15 String httpResponseHeader =
16   "HTTP/1.1 200 OK\r\n" \
17   "Content-Type: text/html\r\n" \
18   "Connection: close\r\n" \
19   "\r\n";
20 String webView =
21   "<!DOCTYPE HTML>\r\n" \
22   "<html>\r\n" \
23   "<head>\r\n" \
24   "<title>ESP32 HTTP Server</title>\r\n" \
25   "</head>\r\n" \
```

```
26  "<body>\r\n" \
27  "<p>A web page from ESP32</p>\r\n" \
28  "</body>\r\n" \
29  "</html>\r\n";
30
31 void setup()
32 {
33     // Setup serial communication
34     Serial.begin(115200);
35
36     // *** Connect to a WiFi access point ***
37     Serial.printf("Connecting to %s ... \n", ssid);
38     WiFi.mode(WIFI_STA);
39     WiFi.begin(ssid, pass);
40     if (WiFi.waitForConnectResult() != WL_CONNECTED)
41     {
42         Serial.printf("WiFi connect failed! Rebooting ... \n");
43         delay(1000);
44         ESP.restart();
45     }
46     Serial.printf("Connected\n");
47     Serial.printf("IP address: %s\n", WiFi.localIP()
48                 .toString().c_str());
49
50     // *** Start TCP server ***
51     server.begin();
52     Serial.printf("TCP server started at port %d\n", port);
53 }
54
55 void loop()
56 {
57     // Get TCP client
58     WiFiClient client = server.available();
59
60     // *** If a client connected ***
61     if (client)
62     {
```

```
63     Serial.printf("[Client connected]\n");
64
65     // An http request ends with a blank line
66     boolean currentLineIsBlank = true;
67     while (client.connected())
68     {
69         if (client.available())
70         {
71             // *** Read a character from client ***
72             char c = client.read();
73             Serial.write(c);
74
75             // The new line is a blank line
76             // The http request has ended
77             if (c == '\n' && currentLineIsBlank)
78             {
79                 // Send HTTP response header
80                 client.print(httpResponseHeader);
81                 // Send web page
82                 client.print(webPage);
83                 Serial.printf("[HTTP response sent]\n");
84
85                 break;
86             }
87
88             // Every line of HTTP request ends with \r\n
89             if (c == '\n')
90             {
91                 // This is the last character of every line
92                 currentLineIsBlank = true;
93             }
94             else if (c != '\r')
95             {
96                 // The new line is not a blank line
97                 currentLineIsBlank = false;
98             }
99     }
```

```
100     }
101     delay(1);
102
103     // *** Close client ***
104     client.stop();
105     Serial.printf("[Client disconnected]\n\n");
106 }
107 }
```

Here is the step-by-step to create a simple HTTP server:

1. First, you need to set the ESP32 in station mode as in **line 37-48**.
2. Then, you need to start the TCP server to listen to the port 80 as in **line 51**. The port number and server object are declared in **line 10 and 13**, respectively.
3. After that, you need to accept a connection request from a client as in **line 58**. If there is a client connected (**line 61**), then the server will process it.
4. Next, you need to read every characters from receive buffer until you reach the end of HTTP request message as in **line 66-100**. It is indicated by a blank line. In this case, we assume that the HTTP request from the web browser doesn't have body data.
5. At the end of HTTP request, you need to send HTTP response header and HTTP response body (web page) as in **line 80 and 82**, respectively. The HTTP response header is defined in **line 15-19**, while the web page is defined in **line 20-29**.
6. Then, in **line 101**, you need to add a delay to give time for the web browser to receive the data.
7. Finally, in **line 104**, the server terminates the connection to the client.

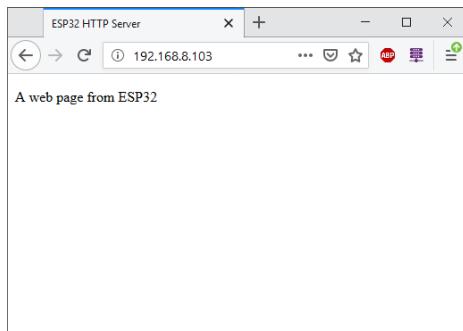


Figure 12.4. A web page from the HTTP server.

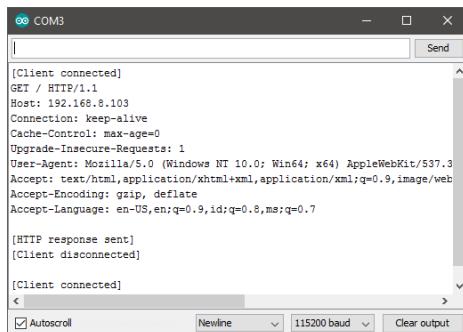


Figure 12.5. An example of HTTP server logs in serial terminal.

In order to test this example, you need to connect the ESP32 to an AP. Then, from your laptop/PC, you need to open a web browser. Type your ESP32's IP address in the address bar. A web page will be loaded as shown in Figure 12.4. This simple web page is written in HTML. We will get into the details of HTML in the next part.

The HTTP server logs are shown in Figure 12.5. First, a client is connected. Then, the client sends an HTTP request message. Note

that the HTTP request message is ended with a blank line. After that, the server sends the HTTP response message. Finally, the connection is terminated.

12.6 Summary

In this chapter, you have learned how to create a simple HTTP server in the ESP32. A web browser can connect to the HTTP server. A simple HTML web page is stored in the ESP32. Then, a web browser loads and displays the web page.

Chapter 13

HTTP Client

What will you learn in this chapter?

- Concept of HTTP client.
- Create a simple HTTP client on a ESP32 station.

13.1 HTTP Client

The previous chapter explained the concept of HTTP server. A server waits for a request message from a client. Once the client request has been received, then the server processes it, and returns a response message which contains a resource for the client. Any devices can act as client, for example PCs, tablets, smartphones, embedded devices, and even computer servers.

Suppose an embedded device provides weather information to a smartphone. In this scenario, the embedded device acts as a server. In another scenario, an embedded device requests weather information from an online server. So, in this scenario, the embedded device acts

as a client.

13.2 ESP32 HTTP Client

As in the previous chapter, we are going to use only the TCP class, which is the TCP client class. In ESP32 Arduino library, the TCP client is called `WiFiClient`. Again, the reason is that the HTTP communication takes place over TCP. So, you only need the a TCP client runs on the ESP32, and you will be able to send and receive HTTP messages.

13.3 Example Code: HTTP Client

In this example code, you are going to create a simple HTTP client on top of a TCP client. The ESP32 that acts as an HTTP client connects to an HTTP server, which is the google.com. Then, the client sends an HTTP request to the server. After that, the server sends the HTTP response which contains the Google home page. The code is shown in Listing 13.1.

Listing 13.1. HTTP client

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 1 Mar 2020
3
4 #include <WiFi.h>
5
6 const char ssid[] = "Huawei-E5573";
7 const char pass[] = "huaweiie5573";
8
9 const char host[] = "www.google.com";
10 const uint16_t port = 80;
11
12 String httpRequest =
13     "GET / HTTP/1.1\r\n \\"
```

```
14     "Host: " + String(host) + "\r\n" \
15     "Connection: close\r\n\r\n";
16
17 void setup()
18 {
19     // Setup serial communication
20     Serial.begin(115200);
21
22     // *** Connect to a WiFi access point ***
23     Serial.printf("Connecting to %s ...\\n", ssid);
24     WiFi.mode(WIFI_STA);
25     WiFi.begin(ssid, pass);
26     if (WiFi.waitForConnectResult() != WL_CONNECTED)
27     {
28         Serial.printf("WiFi connect failed! Rebooting...\\n");
29         delay(1000);
30         ESP.restart();
31     }
32     Serial.printf("Connected\\n");
33     Serial.printf("IP address: %s\\n", WiFi.localIP()
34             .toString().c_str());
35 }
36
37 void loop()
38 {
39     // TCP client object declaration
40     WiFiClient client;
41
42     // *** Connect to host server ***
43     Serial.printf("Connecting to %s ...\\n", host);
44     if (!client.connect(host, port))
45     {
46         Serial.printf("Connection failed\\n");
47         Serial.println("Wait 5 sec ...");
48         delay(5000);
49         return;
50     }
```

```
51     Serial.printf("Connected\n");
52
53     // Send HTTP request
54     client.print(httpRequest);
55
56     // *** Wait until server give response ***
57     unsigned long timeout = millis();
58     while (client.available() == 0)
59     {
60         if (millis() - timeout > 5000)
61         {
62             Serial.printf("Client Timeout\n");
63             client.stop();
64             return;
65         }
66     }
67
68     // *** Read HTTP response from server ***
69     while(client.available())
70     {
71         String line = client.readStringUntil('\n');
72         Serial.print(line);
73     }
74
75     // *** Close TCP connection ***
76     Serial.printf("\nClosing connection\n");
77     client.stop();
78
79     Serial.println("Wait 5 sec ...");
80     delay(5000);
81 }
```

Here is the step-by-step to create a simple HTTP client:

1. First, you need to set the ESP32 in station mode as in **line 23-34**.

2. Then, you need to create a TCP client object, and connect it to the google.com as in **line 40-51**. The host name and port number are declared in **line 9 and 10**, respectively.
3. After that, an HTTP request is defined in **line 12-25**, and it is sent to the server as in **line 54**.
4. Next, you need to wait until the server sends the HTTP response as in **line 57-66**.
5. Then, you need to read every characters from the HTTP response as in **line 68-73**. The HTTP response is printed to serial terminal.
6. Finally, you need to terminate the connection to the server as in **line 77**.

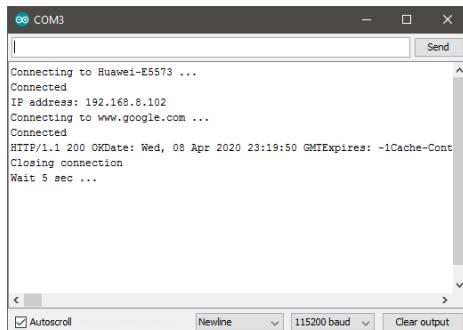


Figure 13.1. An example of HTTP client logs in serial terminal.

In order to test this example, the ESP32 should be connected to an AP that has an Internet access. Then, the client connects to the google.com to request the home page. The server gives the response, and it is printed in serial terminal. Finally, the client terminates the

connection to the server. The HTTP client logs are shown in Figure 13.1

13.4 Summary

In this chapter, you have learned how to create a simple HTTP client in the ESP32. The client can connect to google.com to request the Google home page. Then, the web page is printed to serial terminal.

Part III

Embedded Web Development

Chapter 14

Web Server

What will you learn in this chapter?

- Concept of web server.
- Create a web server on an ESP32 station.

14.1 Web Server

In chapter 12, you have learned about the HTTP server. It was built only with the TCP library. It can only receive simple HTTP requests and send simple HTTP responses. In this chapter, you are going to learn how to build a web server using a web server library. With a web server library, you can focus with the content of your web page, because the HTTP server functionality is already handled by the library. You just need to create your web pages, and then set handlers to them.

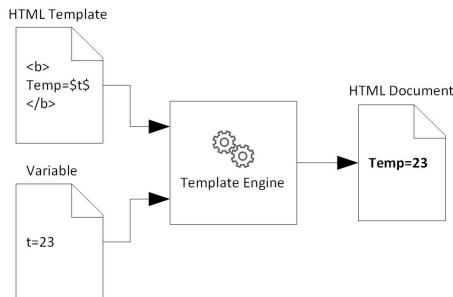


Figure 14.1. The concept of web template.

14.2 ESPAsyncWebServer Library

In this chapter, we are going to use a web server library called ESPAsyncWebServer [20]. This library uses asynchronous network, so that you can handle more than one connection at the same time. It runs on top of a base library called AsyncTCP [21]. The web server does not run on the main loop. Instead, it uses handlers. A handler is a routine that deals with an event. The routine would be executed when the event occurs, i.e. the client request event. Inside a handler, you can't use yield or delay or any function that uses them. Furthermore, the server is smart enough to know when to close the connection and free resources.

ESPAsyncWebServer has a simple template processing engine. A web template helps you to create dynamic elements based on request parameters from a client. The concept of web template is shown in Figure 14.1. The content and template are combined through the template processing engine to produce the final HTML document. The ESPAsyncWebserver supports only replacing template placeholders with actual values from variables. The practical use of web template will be explained in more detail later.

14.3 Example Code: Simple Web Server

In this example code, you are going to create a simple web server. It accepts connection requests from web browsers (clients). Then, it sends a simple HTML code that will display 'Hello, World!'. After that, the web browser engines render the web page from the received the HTML code. The code is shown in Listing 14.1.

Listing 14.1. Simple web server using ESPAsyncWebServer

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 const char ssid[] = "Huawei-E5573";
8 const char pass[] = "huaweie5573";
9
10 // TCP server port for HTTP
11 const uint16_t port = 80;
12
13 // Create AsyncWebServer object
14 AsyncWebServer server(port);
15
16 // Web page
17 const char webpage[] PROGMEM = R"=====(
18 <html>
19   <head>
20     <title>ESP32 Web Page</title>
21   </head>
22   <body>
23     <p>Hello, World!</p>
24   </body>
25 </html>
26 )=====;"
```

```
28 void setup()
29 {
30     // Setup serial communication
31     Serial.begin(115200);
32
33     // *** Connect to a WiFi access point ***
34     Serial.printf("Connecting to %s ...\\n", ssid);
35     WiFi.mode(WIFI_STA);
36     WiFi.begin(ssid, pass);
37     if (WiFi.waitForConnectResult() != WL_CONNECTED)
38     {
39         Serial.printf("WiFi connect failed! Rebooting ...\\n");
40         delay(1000);
41         ESP.restart();
42     }
43     Serial.printf("Connected\\n");
44     Serial.printf("IP address: %s\\n", WiFi.localIP()
45                 .toString().c_str());
46
47     // Handler for root request
48     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
49     {
50         request->send(200, "text/html", webpage);
51     });
52
53
54     // Start server
55     server.begin();
56 }
57
58 void loop()
59 {
60 }
```

Here is the step-by-step to create a simple web server:

1. First, you need to set the ESP32 in station mode as in **line 34-45**.
2. Then, you need to instantiate the `AsyncWebServer` class as an object named `server` as in **line 14**. It takes one input argument—the default HTTP port number (80).
3. After that, you need to add a handler for the '/' (root) request by calling `on` method as in **line 48-52**. The request is specified as HTTP GET. When the event is occurred, it sends the HTML code that is defined in `webpage`.
4. The `webpage` is defined in **line 17-26**. We use a variable modifier `PROGMEM` to store data in flash (program) memory instead of SRAM. We define the HTML code as raw string literal. A raw string literal starts with `R"=====(` and ends with `)====="`. In raw string literal, the escape characters (like `\n`, `\t`, or `"`) of C++ are not processed, so you would get a single line of text with a content identical with what you have in the source code.
5. Finally, in **line 55**, you need to start the web server by calling `begin` method.

In order to test this example, you need to connect the ESP32 to an AP. Then, from your laptop/PC, you need to open a web browser. Type your ESP32's IP address in the address bar. A web page will be loaded as shown in Figure 14.2. This simple web page is written in HTML. We will get into the details of HTML in the next chapter.

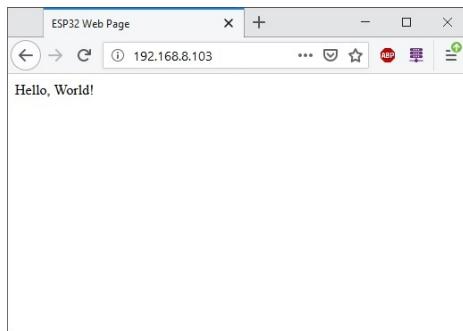


Figure 14.2. A web page from the web server.

14.4 Summary

In this chapter, you have learned how to create a simple web server in the ESP32. The result is same as in chapter 12, but we use the ESPAsyncWebServer library. So, we just need to define our web content and leave the rest of HTTP processing to the library.

Chapter 15

HyperText Markup Language (HTML)

What will you learn in this chapter?

- Concept of HyperText Markup Language (HTML).
- Create web pages for interfacing with hardware.

15.1 HyperText Markup Language (HTML)

HyperText Markup Language (HTML) is the standard programming language for developing web pages. It defines the structure of web pages. HTML usually works together with Cascading Style Sheets (CSS) and JavaScript (JS) to create web pages. CSS defines the presentation/appearance of the web pages, and JS defines the behaviour of the web pages. The HTML, CSS, and JS form the front-end design of websites.

As you have seen in the previous chapter, web browsers receive HTML

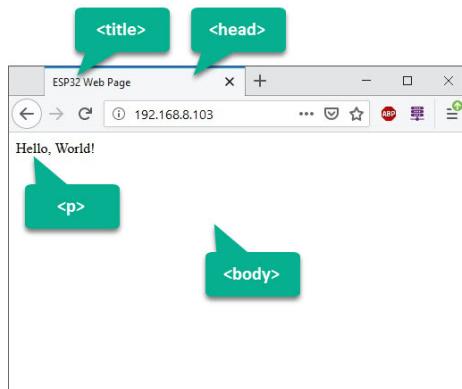


Figure 15.1. A simple HTML document.

documents from a web server or from a local storage. The web browser engines render the HTML documents into web pages. HTML uses various tags to format the content. The following is an example of a simple HTML document from the previous chapter:

```
<html>
  <head>
    <title>ESP32 Web Page</title>
  </head>
  <body>
    <p>Hello, World!</p>
  </body>
</html>
```

A web browser renders this HTML to a web page as shown in Figure 15.1. This HTML code uses several basic tags:

- `<html>`: this tag encloses a complete HTML document.

- `<head>`: this tag is a container for the head elements, such as `<title>`, `<meta>`, `<script>`, etc.
- `<title>`: this tag defines the title of HTML document.
- `<body>`: this tag contains all the contents of HTML document, such as `<p>`, `<dir>`, etc.
- `<p>`: this tag represents a paragraph.

Most of the tags have their corresponding closing tags. For example `<head>` has its closing tag `</head>`.

15.2 Example Code: HTML for Controlling an LED

In this example code, you are going to create a web page in HTML for controlling an LED on the ESP32. In this chapter, you are going to create only the HTML code. You will get into the web server and its functionality in the next chapter.

Figure 15.2 shows the HTML web page. To control an LED, you can use two buttons (on and off). The HTML code is shown in Listing 15.1.

Listing 15.1. HTML code for controlling an LED

```
1 <html>
2   <head>
3     <title>ESP32 Web Page</title>
4   </head>
5   <body>
6     <p>Write LED State</p>
7     <a href="/on"><button>ON</button></a>
8     <a href="/off"><button>OFF</button></a>
9   </body>
10 </html>
```

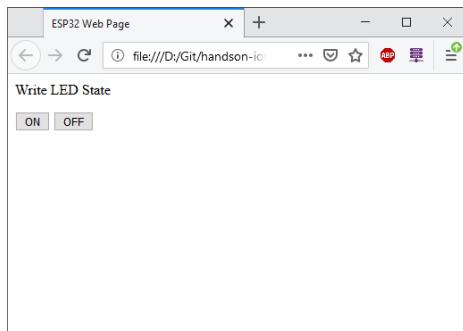


Figure 15.2. A web page for controlling an LED.

This HTML code uses several tags:

1. First, in **line 3**, you need to define the title.
2. Then, in **line 6**, you need to define the description within `<p>` tag.
3. After that, in **line 7 and 8**, you need to define the buttons (ON and OFF) within `<button>` tag. The `<button>` tag creates a button.
4. Finally, you need to put the button within `<a>` tag. The `<a>` tag defines a hyperlink. Later, this hyperlink will be associated with the LED functionality.

In this HTML code, we use hyperlinks to control the LED. When you press the button, the web browser will send an HTTP GET request to the URL that corresponds to the button. For example, if you press the ON button, the web browser will send an HTTP GET to 192.168.8.102/on.

15.3 Example Code: HTML for Dimming an LED

In this example code, you are going to create a web page in HTML for dimming an LED on the ESP32 with PWM. The web page is similar to the previous web page, but instead of using a button to send digital control (on or off), you are going to use a slider to send the PWM value.

Figure 15.3 shows the HTML web page. To dim an LED using a slider, you can use an HTML input element of type range. The HTML code is shown in Listing 15.2.

Listing 15.2. HTML code for dimming an LED

```
1 <html>
2   <head>
3     <title>ESP32 Web Page</title>
4   </head>
5   <body>
6     <p>Write LED Value</p>
7     <form action="/led_pwm" method="POST">
8       <input type="range" name="pwm_value"
9         min="0" max="1023" value="0"><br>
10      <input type="submit">
11    </form>
12  </body>
13 </html>
```

This HTML code uses several tags:

1. First, in **line 3**, you need to define the title, and in **line 6**, you need to define the description within `<p>` tag.
2. Then, in **line 7-11**, you need to define the slider and submit button within the `<form>` tag. This tag has two attributes—`action` and `method`. The `action` attribute is the URL, and the `method` attribute is the HTTP method.

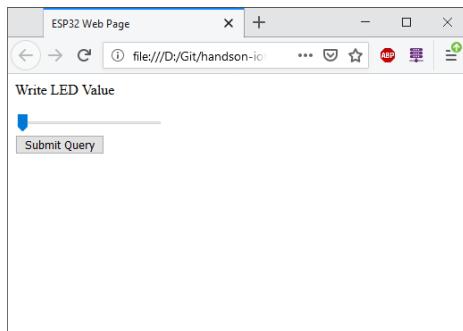


Figure 15.3. A web page for dimming an LED with PWM.

3. Next, in **line 8 and 9**, you need to define the slider. The range of the PWM value is from 0 to 1023.
4. Finally, in **line 10**, you need to define a submit button. This is another way to define a button instead of using `<button>` tag.

In this HTML code, instead of using HTTP GET, we use HTTP POST to submit the PWM value.

15.4 Example Code: HTML for Reading a Physical Button

In this example code, you are going to create a web page in HTML for reading a physical button on the ESP32. You will use a `<p>` tag to display the physical button state. Figure 15.4 shows the HTML web page. The HTML code is shown in Listing 15.3.

Listing 15.3. HTML code for reading a physical button

```
1 <html>
2   <head>
```

```
3      <title>ESP32 Web Page</title>
4      <meta http-equiv="refresh" content="3">
5  </head>
6  <body>
7      <p>Read Button State</p>
8      <p><b>%STATE%</b></p>
9  </body>
10 </html>
```

This HTML code uses several tags:

1. First, in **line 3**, you need to define the title, and in **line 7**, you need to define the description within `<p>` tag.
2. Next, in **line 4**, you need to use `meta` tag. Within this tag, you need to use `http-equiv="refresh"` attribute, and set the `content` attribute to 3. This code is used to tell the web browser to automatically refresh the web page every 3 seconds. So, the new state of the switch would be updated.
3. Finally, in **line 8**, you need to define a placeholder for displaying the button state within the `<p>` and `` (bold text) tags. Later, the web server replaces this placeholder the actual physical button state.

In this code, we set the HTML code to automatically refresh the web page every 3 seconds. This is a very simple method to read a state or value in the ESP32. In this method, the web browser flashes every 3 seconds, so it is quite annoying. In the next couple of tutorial, we are going to improve this by using JavaScript. By using JavaScript, the web browser can refresh only part of the web page that displays the button state.

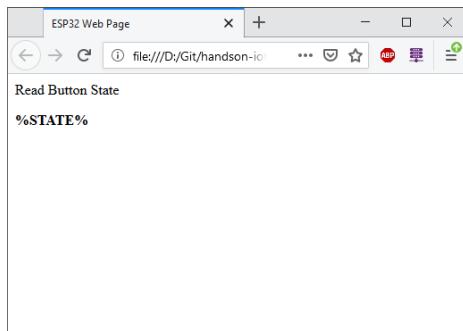


Figure 15.4. A web page for reading a physical button.

15.5 Summary

To summarize, in this chapter you have created several HTML web pages for interfacing with hardware components, such as LED or button. In the next chapter, you will host these web pages on the ESP32 web server and associate the HTML elements to the corresponding hardware components.

Chapter 16

Web Page Data Exchange

What will you learn in this chapter?

- Create web servers for interfacing with hardware.
- HTTP GET and HTTP POST methods.

16.1 Web Page Data Exchange

In the previous chapter, we have learned how to create web pages in HTML for interfacing with hardware, such as LED and button. In this chapter, we are going to host the web pages on the ESP32 and associate the HTML elements to the corresponding hardware components. The web pages can interact with the hardware components through the HTTP GET or HTTP POST method.

There are three examples in this chapter. First, we are going to control the LED on and off. Second, we are going to dim the LED with PWM. Third, we are going to read the physical button state.

16.2 Example Code: Web Server for Controlling an LED

In the first example code, you are going to create a web server for controlling an LED. The code is shown in Listing 16.1. You are going to use the HTML code that has been designed in the previous chapter. It has two HTML buttons for turning on and off the LED. When a user presses the HTML button, the web browser will send HTTP GET to a specified URL, either to turn on or off the LED.

Listing 16.1. Web server for controlling an LED

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 // On-board LED's GPIO pin
8 #define LED_ON_BOARD 2
9
10 const char ssid[] = "Huawei-E5573";
11 const char pass[] = "huaweiie5573";
12
13 // TCP server port for HTTP
14 const uint16_t port = 80;
15
16 // Create AsyncWebServer object
17 AsyncWebServer server(port);
18
19 // Web page
20 const char webpage[] PROGMEM = R"=====(
21 <html>
22   <head>
23     <title>ESP32 Web Page</title>
24   </head>
25   <body>
```

```
26     <p>Write LED State</p>
27     <a href="/on"><button>ON</button></a>
28     <a href="/off"><button>OFF</button></a>
29   </body>
30 </html>
31 ) =====";
32
33 void setup()
34 {
35   // Set GPIO pin as output for on-board LED
36   pinMode(LED_ONBOARD, OUTPUT);
37   // Setup serial communication
38   Serial.begin(115200);
39
40   // *** Connect to a WiFi access point ***
41   Serial.printf("Connecting to %s ...\\n", ssid);
42   WiFi.mode(WIFI_STA);
43   WiFi.begin(ssid, pass);
44   if (WiFi.waitForConnectResult() != WL_CONNECTED)
45   {
46     Serial.printf("WiFi connect failed! Rebooting ...\\n");
47     delay(1000);
48     ESP.restart();
49   }
50   Serial.printf("Connected\\n");
51   Serial.printf("IP address: %s\\n", WiFi.localIP()
52     .toString().c_str());
53
54   // Handler for root request
55   server.on("/", 
56     HTTP_GET, [] (AsyncWebServerRequest *request)
57   {
58     request->send_P(200, "text/html", webpage);
59   });
60
61   // Handler for turning on the LED
62   server.on("/on",
```

```
63     HTTP_GET, [](AsyncWebServerRequest *request) {
64     printf("%s/on\n", WiFi.localIP().toString().c_str());
65     digitalWrite(LED_ON_BOARD, HIGH);
66     printf("LED On\n");
67     request->redirect("/");
68 });
69
70 // Handler for turning off the LED
71 server.on("/off",
72     HTTP_GET, [](AsyncWebServerRequest *request) {
73     printf("%s/off\n", WiFi.localIP().toString().c_str());
74     digitalWrite(LED_ON_BOARD, LOW);
75     printf("LED Off\n");
76     request->redirect("/");
77 });
78
79 // Start server
80 server.begin();
81 }
82
83 void loop()
84 {
85 }
```

Here is the step-by-step to create a web server for controlling an LED:

1. First, you need to define the HTML code as in **line 20-31**.
2. Then, in **line 36**, you need to initialize the GPIO pin associated with the on-board LED as output.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 41-52**.
4. Next, in **line 55-59**, you need to define a handler for the / request. When there is a request from a web browser, the web server will send the HTML code.

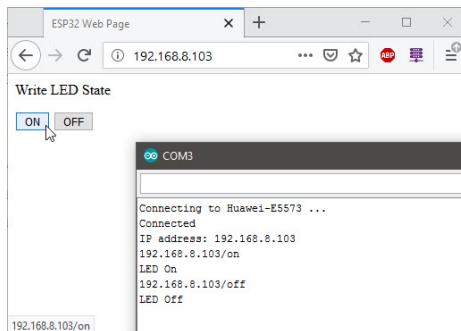


Figure 16.1. A web server for controlling an LED.

5. Next, in **line 62-68**, you need to define a handler for `/on` request. When the ON button is pressed, this handler will be executed. It will turn on the LED, and redirects back to the `/`.
6. Next, in **line 71-77**, you need to define a handler for `/off` request. When the OFF button is pressed, this handler will be executed. It will turn off the LED, and redirects back to the `/`.
7. Finally, in **line 80**, you need to start the web server.

Note that if you do not redirect the web page back to the `/` as in **line 67 and 76**, you would see a blank page in the web browser. This is because the `/on` and `/off` URLs don't have any HTML code associated with them. The result of this example code is illustrated in Figure 16.1.

16.3 Example Code: Web Server for Dimming an LED

In the second example code, you are going to create a web server for dimming an LED with PWM. The code is shown in Listing 16.2. In

the HTML code, it uses a slider and a button to submit the PWM value to the web server. In this case, we use HTTP POST to send the PWM value.

Listing 16.2. Web server for dimming an LED

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 // On-board LED's GPIO pin
8 #define LED_ON_BOARD 2
9
10 // PWM parameters
11 const int channel = 0;
12 const int freq = 1000;
13 const int res = 10;
14
15 const char ssid[] = "Huawei-E5573";
16 const char pass[] = "huaweie5573";
17
18 // TCP server port for HTTP
19 const uint16_t port = 80;
20
21 // Create AsyncWebServer object
22 AsyncWebServer server(port);
23
24 // Web page
25 const char webpage[] PROGMEM = R"=====(
26 <html>
27   <head>
28     <title>ESP32 Web Page</title>
29   </head>
30   <body>
31     <p>Write LED Value</p>
```

```
32      <form action="/led_pwm" method="POST">
33          <input type="range" name="pwm_value"
34              min="0" max="1023" value="0"><br>
35          <input type="submit">
36      </form>
37  </body>
38 </html>
39 ) =====";
40
41 void setup()
42 {
43     // Set GPIO pin as output for on-board LED
44     pinMode(LED_ON_BOARD, OUTPUT);
45     // Configure PWM
46     ledcSetup(channel, freq, res);
47     // Attach the PWM channel to the LED
48     ledcAttachPin(LED_ON_BOARD, channel);
49
50     // Setup serial communication
51     Serial.begin(115200);
52
53     // *** Connect to a WiFi access point ***
54     Serial.printf("Connecting to %s ... \n", ssid);
55     WiFi.mode(WIFI_STA);
56     WiFi.begin(ssid, pass);
57     if (WiFi.waitForConnectResult() != WL_CONNECTED)
58     {
59         Serial.printf("WiFi connect failed! Rebooting ... \n");
60         delay(1000);
61         ESP.restart();
62     }
63     Serial.printf("Connected\n");
64     Serial.printf("IP address: %s\n", WiFi.localIP()
65                 .toString().c_str());
66
67     // Handler for root request
68     server.on("/",
```

```
69         HTTP_GET, [] (AsyncWebServerRequest *request)
70     {
71         request->send_P(200, "text/html", webpage);
72     });
73
74     // Handler for button request
75     server.on("/led_pwm",
76             HTTP_POST, [] (AsyncWebServerRequest *request)
77     {
78         // Get HTTP POST parameter
79         AsyncWebParameter *p = request->getParam(0);
80         if (p->isPost())
81     {
82         // Print HTTP POST parameter to the serial monitor
83         Serial.printf("POST[%s]: %s\n",
84                     p->name().c_str(), p->value().c_str());
85     }
86         // Write PWM value to the LED
87         ledcWrite(channel, p->value().toInt());
88
89         request->redirect("/");
90     });
91
92     // Start server
93     server.begin();
94 }
95
96 void loop()
97 {
98 }
```

Here is the step-by-step to create a web server for dimming an LED:

1. First, you need to define the HTML code as in **line 25-39**.
2. Then, in **line 44-48**, you need to initialize the GPIO pin and the PWM for the on-board LED.

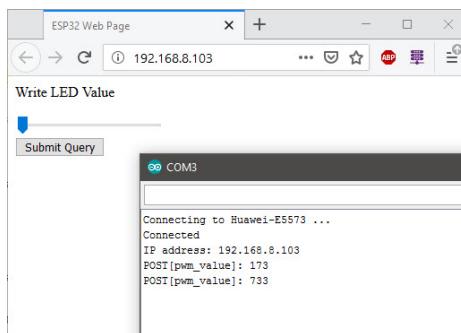


Figure 16.2. A web server for dimming an LED.

3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 54-65**.
4. Next, in **line 68-72**, you need to define a handler for the `/` request.
5. Next, in **line 75-90**, you need to define a handler for `/led_pwm` request. It uses HTTP POST method. Within the handler, you need to read the HTTP POST parameter by calling `getParam` method as in **line 79**. You can print the name-value pair to the serial monitor for debugging as in **line 80-85**. In **line 87**, set the PWM value to the LED.
6. Finally, in **line 93**, you need to start the web server.

The result of this example code is illustrated in Figure 16.2. When you press the submit button, the web browser sends an HTTP POST to the web server. The PWM value is inserted as name-value pair of the HTTP POST parameter.

```

GET /led_pwm?pwm_value=152 HTTP/1.1
Host: 192.168.1.98
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,id;q=0.8,ms;q=0.7

HTTP/1.1 302 Found
Content-Length: 0
Connection: close
Location: /
Accept-Ranges: none

```

(a)


```

POST /led_pwm HTTP/1.1
Host: 192.168.1.98
Connection: keep-alive
Content-Length: 13
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://192.168.1.98
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://192.168.1.98/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,id;q=0.8,ms;q=0.7

pwm_value=152HTTP/1.1 302 Found
Content-Length: 0
Connection: close
Location: /
Accept-Ranges: none

```

(b)

Figure 16.3. (a) HTTP GET name–pair in URL; (b) HTTP POST name–pair in message body.

16.4 HTTP GET and POST Methods

There are two HTTP methods that can be used to send data to the server—HTTP GET and HTTP POST. The main difference is that GET carries request parameters appended in URL. While POST carries request parameters in message body. POST is a more secure way of transferring data from client to server in HTTP.

For the dimming LED example, you can also use GET to submit the PWM value. If you use GET, the name–value pair is appended in URL as shown in Figure 16.3(a). If you use POST, the name–value

pair is appended in message body as shown in Figure 16.3(b). In GET method, it is very easy to alter the data because it is appended in URL. While in POST method, it is more secure because you can't easily alter the message body.

16.5 Example Code: Web Server for Reading a Physical Button

In the third example code, you are going to create a web server for reading a physical button. The code is shown in Listing 16.3. In the HTML code, it uses a template placeholder to display the physical button state. Whenever the web browser requests the web page, the button state will be updated.

Listing 16.3. Web server for reading a physical button

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date  : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 // Button's GPIO pin
8 #define BUTTON 12
9
10 const char ssid[] = "Huawei-E5573";
11 const char pass[] = "huaweie5573";
12
13 // TCP server port for HTTP
14 const uint16_t port = 80;
15
16 // Create AsyncWebServer object
17 AsyncWebServer server(port);
18
19 // Web page
20 const char webpage[] PROGMEM = R"=====("
```

```
21 <html>
22   <head>
23     <title>ESP32 Web Page</title>
24     <meta http-equiv="refresh" content="3">
25   </head>
26   <body>
27     <p>Read Button State</p>
28     <p><b>%STATE%</b></p>
29   </body>
30 </html>
31 ) ======";
32
33 // Template processor for replacing placeholder
34 // with button state
35 String processor(const String& var)
36 {
37     String button_state;
38
39     if (var == "STATE")
40     {
41         // Read active-low button state
42         if (digitalRead(BUTTON))
43         {
44             button_state = "Released";
45             printf("Button released\n");
46         }
47         else
48         {
49             button_state = "Pressed";
50             printf("Button pressed\n");
51         }
52         return button_state;
53     }
54
55     return String();
56 }
57 }
```

```
58 void setup()
59 {
60     // Set GPIO pin as input for button
61     pinMode(BUTTON, INPUT_PULLUP);
62     // Setup serial communication
63     Serial.begin(115200);

64
65     // *** Connect to a WiFi access point ***
66     Serial.printf("Connecting to %s ...\\n", ssid);
67     WiFi.mode(WIFI_STA);
68     WiFi.begin(ssid, pass);
69     if (WiFi.waitForConnectResult() != WL_CONNECTED)
70     {
71         Serial.printf("WiFi connect failed! Rebooting ...\\n");
72         delay(1000);
73         ESP.restart();
74     }
75     Serial.printf("Connected\\n");
76     Serial.printf("IP address: %s\\n", WiFi.localIP()
77         .toString().c_str()));

78
79     // Handler for root request
80     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
81     {
82         request->send_P(200, "text/html", webpage, processor);
83     });
84
85
86     // Start server
87     server.begin();
88 }
89
90 void loop()
91 {
92 }
```

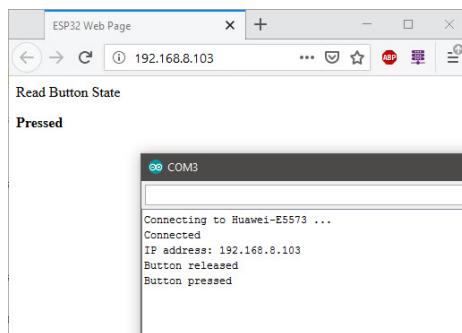


Figure 16.4. A web server for reading a physical button.

Here is the step-by-step to create a web server for reading a physical button:

1. First, you need to define the HTML code as in **line 20-31**.
2. Then, in **line 61**, you need to initialize a GPIO pin as input for a button.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 66-77**.
4. Next, in **line 80-84**, you need to define a handler for the `/` request. When the server sends the web page, it calls a template processor function named `processor`. This function is defined in **line 35-56**. This function replaces the template placeholder `STATE` to the actual button state.
5. Finally, in **line 87**, you need to start the web server.

The result of this example code is illustrated in Figure 16.4. The web browser will automatically refresh the web page every 3 seconds. So,

the button state is get updated. In the next chapter, we are going to improve this by using JavaScript. So, the web browser can refresh only part of the web page that displays the button state.

16.6 Summary

In this chapter, you have learned how to exchange data between client and server using HTTP. There are two methods for requesting data to server, which are HTTP GET and HTTP POST.

Chapter 17

JavaScript (JS)

What will you learn in this chapter?

- The concept of JavaScript for front-end.
- Bank-end data loading using JavaScript.

17.1 JavaScript

JavaScript is one of the three languages for front-end development. It is used to define the behaviour of the web pages. JavaScript code is executed by a JavaScript engine of the web browser. JavaScript can manipulate web pages on the fly which makes web pages more dynamic and interactive. It can also be used for back-end data loading while you are doing other processing.

JavaScript can be implemented within the `<script>` tag in a web page. Normally, the `<script>` tag can be placed within the `<body>` tag or the `<head>` tag, depending on when you want the script to be loaded. Here is an example of JavaScript code to display hello world

on the web page:

```
<html>
  <head>
    <title>ESP32 Web Page</title>
  </head>
  <body>
    <script>
      document.write("Hello, World!");
    </script>
  </body>
</html>
```

JavaScript is a case-sensitive language. In JavaScript, the semicolon character at the end of every line is optional. However, it is a good programming practice to use semicolons.

In this chapter, we are going to use JavaScript to improve the web pages in the previous chapter. There are three examples in this chapter. First, we are going to dim the LED with PWM. This example is similar to the previous one, but instead of HTML form, we are going to use JavaScript to send the HTTP request. Second, we are going to read the physical button state using JavaScript. This example is also similar to the previous one, but we are going to update only the part that displays the button state. Third, we are going to read the DHT11 sensor using the same method as in the second example.

17.2 Example Code: Dimming an LED with JS

In the first example code, you are going to create a web server for dimming an LED. The code is shown in Listing 17.1. We are going to use a technique called AJAX. AJAX stands for Asynchronous JavaScript And XML. It is a technique for accessing a web server from a web page. We are going to send the PWM value to the server using HTTP POST with AJAX.

Listing 17.1. Dimming an LED with JavaScript

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 // On-board LED's GPIO pin
8 #define LED_ON_BOARD 2
9
10 // PWM parameters
11 const int channel = 0;
12 const int freq = 1000;
13 const int res = 10;
14
15 const char ssid[] = "Huawei-E5573";
16 const char pass[] = "huaweiie5573";
17
18 // TCP server port for HTTP
19 const uint16_t port = 80;
20
21 // Create AsyncWebServer object
22 AsyncWebServer server(port);
23
24 // Web page
25 const char webpage[] PROGMEM = R"=====(
26 <html>
27   <head>
28     <title>ESP32 Web Page</title>
29   </head>
30   <body>
31     <p>Write LED Value using AJAX</p>
32     <input type="range" id="range_pwm"
33       min="0" max="1023" value="0">
34     <script>
35       document.getElementById("range_pwm")
```

```
36         .oninput = function() {
37             var post_request = "pwm_value=" +
38                 document.getElementById("range_pwm").value;
39             var xhttp = new XMLHttpRequest();
40             xhttp.open("POST", "led_pwm_ajax", true);
41             xhttp.send(post_request);
42         };
43     </script>
44     </body>
45 </html>
46 )=====;
47
48 void setup()
49 {
50     // Set GPIO pin as output for on-board LED
51     pinMode(LED_ONBOARD, OUTPUT);
52     // Configure PWM
53     ledcSetup(channel, freq, res);
54     // Attach the PWM channel to the LED
55     ledcAttachPin(LED_ONBOARD, channel);
56
57     // Setup serial communication
58     Serial.begin(115200);
59
60     // *** Connect to a WiFi access point ***
61     Serial.printf("Connecting to %s ...\\n", ssid);
62     WiFi.mode(WIFI_STA);
63     WiFi.begin(ssid, pass);
64     if (WiFi.waitForConnectResult() != WL_CONNECTED)
65     {
66         Serial.printf("WiFi connect failed! Rebooting ...\\n");
67         delay(1000);
68         ESP.restart();
69     }
70     Serial.printf("Connected\\n");
71     Serial.printf("IP address: %s\\n", WiFi.localIP()
72         .toString().c_str());
```

```
73
74     // Handler for root request
75     server.on("/", [
76         HTTP_GET, [](AsyncWebServerRequest *request) {
77             request->send_P(200, "text/html", webpage);
78         });
79
80
81     // Handler for button AJAX request
82     server.on("/led_pwm_ajax",
83         HTTP_POST, [](AsyncWebServerRequest *request) {
84
85         // Get HTTP POST parameter
86         AsyncWebParameter *p = request->getParam(0);
87         if (p->isPost())
88         {
89             // Print HTTP POST parameter to the serial monitor
90             Serial.printf("POST[%s]: %s\n",
91                 p->name().c_str(), p->value().c_str());
92         }
93         // Write PWM value to the LED
94         ledcWrite(channel, p->value().toInt());
95
96         // Tell client that the HTTP POST has been performed
97         request->send(303);
98     });
99
100    // Start server
101    server.begin();
102 }
103
104 void loop()
105 {
106 }
```

Here is the step-by-step for dimming an LED with JavaScript:

1. First, you need to define the HTML code as in **line 25-46**.
2. Then, in **line 51-55**, you need to initialize the GPIO pin and the PWM for the on-board LED.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 61-72**.
4. Next, in **line 75-79**, you need to define a handler for the `/` request.
5. Next, in **line 82-98**, you need to define a handler for the HTTP POST from AJAX request. Within the handler, you need to read the HTTP POST parameter which is the PWM value, and then you need to set the PWM value to the LED. After that, you need to send HTTP 303 response as in **line 97**. This response tells the client that the HTTP POST has been performed.
6. Finally, in **line 101**, you need to start the web server.

Here is the details on how the web page works. In this web page, we only use a slider without a submit button as we do in the previous chapter. The slider is defined as

```
<input type="range" id="range_pwm"  
min="0" max="1023" value="0">
```

We should define the `id` attribute for this slider because we need to access it from JavaScript code. The JavaScript code itself is defined as

```
<script>  
document.getElementById("range_pwm")  
    .oninput = function() {  
        var post_request = "pwm_value=" +
```

```
document.getElementById("range_pwm").value;
var xhttp = new XMLHttpRequest();
xhttp.open("POST", "led_pwm_ajax", true);
xhttp.send(post_request);
};
</script>
```

In this JavaScript code, we set a handler for the `range_pwm` slider. The `getElementById` method returns the HTML element that has a specified ID attribute. Then, we use `oninput` event, so when the user move the slider, the code inside this handler is executed. Inside the handler, we get the PWM value from the slider, then we create an HTTP request parameter which is stored in a variable named as `post_request`. Next, we create an `XMLHttpRequest` object. With this object, we can exchange data with the web server behind the scenes. Then, we set the HTTP method to POST, and the URL to `led_pwm_ajax`. Finally, we send the POST request by calling `send` method.

The result of this example code is illustrated in Figure 17.1. When you move the slider, the web browser sends the PWM value to the web server. Then, the server sets the PWM value to the LED.

17.3 Example Code: Reading a Physical Button with JS

In the second example code, you are going to create a web server for reading a physical button. The code is shown in Listing 17.2. We are going to use AJAX. With AJAX, we can update the web page asynchronously by exchanging data with a web server behind the scenes. We can update parts of the web page without reloading the whole page.

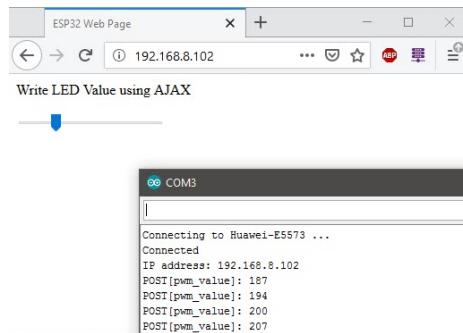


Figure 17.1. A web server for dimming an LED with AJAX.

Listing 17.2. Reading a physical button with JavaScript

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6
7 // Button's GPIO pin
8 #define BUTTON 12
9
10 const char ssid[] = "Huawei-E5573";
11 const char pass[] = "huaweiie5573";
12
13 // TCP server port for HTTP
14 const uint16_t port = 80;
15
16 // Create AsyncWebServer object
17 AsyncWebServer server(port);
18
19 // Web page
20 const char webpage[] PROGMEM = R"=====("
```

```
21 <html>
22   <head>
23     <title>ESP32 Web Page</title>
24   </head>
25   <body>
26     <p>Read Button State using AJAX</p>
27     <p><b><span id="button_state">N/A</span></b></p>
28     <script>
29       setInterval(function() {
30         getButtonState();
31       }, 1000);
32       function getButtonState() {
33         var xhttp = new XMLHttpRequest();
34         xhttp.onreadystatechange = function() {
35           if (this.readyState == 4 && this.status == 200) {
36             document.getElementById("button_state")
37               .innerHTML = this.responseText;
38           }
39         };
40         xhttp.open("GET", "button_ajax", true);
41         xhttp.send();
42       }
43     </script>
44   </body>
45 </html>
46 )=====;
47
48 void setup()
49 {
50   // Set GPIO pin as input for button
51   pinMode(BUTTON, INPUT_PULLUP);
52   // Setup serial communication
53   Serial.begin(115200);
54
55   // *** Connect to a WiFi access point ***
56   Serial.printf("Connecting to %s ...\\n", ssid);
57   WiFi.mode(WIFI_STA);
```

```
58     WiFi.begin(ssid, pass);
59     if (WiFi.waitForConnectResult() != WL_CONNECTED)
60     {
61         Serial.printf("WiFi connect failed! Rebooting ...\\n");
62         delay(1000);
63         ESP.restart();
64     }
65     Serial.printf("Connected\\n");
66     Serial.printf("IP address: %s\\n", WiFi.localIP()
67                 .toString().c_str());
68
69     // Handler for root request
70     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
71     {
72         request->send_P(200, "text/html", webpage);
73     });
74
75
76     // Handler for button AJAX request
77     server.on("/button_ajax",
78             HTTP_GET, [] (AsyncWebServerRequest *request)
79     {
80         if (digitalRead(BUTTON))
81         {
82             request->send(200, "text/plain", "Released");
83             printf("Button released\\n");
84         }
85         else
86         {
87             request->send(200, "text/plain", "Pressed");
88             printf("Button pressed\\n");
89         }
90     });
91
92     // Start server
93     server.begin();
94 }
```

```
95  
96 void loop()  
97 {  
98 }
```

Here is the step-by-step to create a web server for reading a physical button with JavaScript:

1. First, you need to define the HTML code as in **line 20-46**.
2. Then, in **line 51**, you need to initialize a GPIO pin as input for a button.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 56-67**.
4. Next, in **line 70-74**, you need to define a handler for the `/` request.
5. Next, in **line 77-90**, you need to define a handler for the HTTP GET from AJAX request. Within the handler, you need to read the button state, and then send the state as plain text to the client.
6. Finally, in **line 93**, you need to start the web server.

Note that in this example, we don't use template placeholder as we do in the previous chapter. Instead, the JavaScript code will update the button state. Here is the details on how the web page works. The button state is defined within `<p>` tag as

```
<p><b><span id="button_state">N/A</span></b></p>
```

We use `` tag to mark up a part of a text. When it is marked, we can style it with CSS, or manipulate it with JavaScript. The JavaScript code is defined as

```
<script>
    setInterval(function() {
        getButtonState();
    }, 1000);
    function getButtonState() {
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                document.getElementById("button_state")
                    .innerHTML = this.responseText;
            }
        };
        xhttp.open("GET", "button_ajax", true);
        xhttp.send();
    }
</script>
```

We call `setInterval` method to execute the `getButtonState` function every one second. Within the `getButtonState` function, we create an `XMLHttpRequest` object. Then, we define a handler (with `onreadystatechange` method) to process the HTTP response from the server. The handler will update the button state. Finally, we set the HTTP method to GET, the URL to `button_ajax`, and then send it to the server.

When you run this example, you will get the same web page as in the previous chapter, but the web page will not be automatically reloaded every second. This is because we can exchange the button state data behind the scenes, and we reload only the text within the `` tag.

17.4 Example Code: Reading the DHT11 Sensor with JS

In the third example code, you are going to create a web server for reading the DHT11 sensor. The code is shown in Listing 17.3. We are going to use AJAX to send HTTP request to the server. Then, the server will send the response which is the temperature and humidity values. The temperature and humidity values are formatted in JSON.

Listing 17.3. Reading the DHT11 sensor with JavaScript

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 7 Mar 2020
3
4 #include <WiFi.h>
5 #include <ESPAsyncWebServer.h>
6 #include <DHT.h>
7 #include <ArduinoJson.h>
8
9 #define DHTPIN  22
10 #define DHTTYPE DHT11
11
12 // Setup DHT pin and type
13 DHT dht(DHTPIN, DHTTYPE);
14
15 const char ssid[] = "Huawei-E5573";
16 const char pass[] = "huaweiie5573";
17
18 // TCP server port for HTTP
19 const uint16_t port = 80;
20
21 // Create AsyncWebServer object
22 AsyncWebServer server(port);
23
24 // Web page
25 const char webpage[] PROGMEM = R"=====(
26 <html>
```

```
27 <head>
28     <title>ESP32 Web Page</title>
29 </head>
30 <body>
31     <p>Read DHT11 Sensor using AJAX</p>
32     <p>Temperature:<br>
33         <b><span id="temp_value">N/A</span>&deg;C</b></p>
34     <p>Humidity:<br>
35         <b><span id="hum_value">N/A</span>%</b></p>
36     <script>
37         setInterval(function() {
38             getDHT11Value();
39         }, 1000);
40         function getDHT11Value() {
41             var xhttp = new XMLHttpRequest();
42             xhttp.onreadystatechange = function() {
43                 if (this.readyState == 4 && this.status == 200) {
44                     var response = JSON.parse(this.responseText);
45                     document.getElementById("temp_value")
46                         .innerHTML = Math.round(response.temp);
47                     document.getElementById("hum_value")
48                         .innerHTML = Math.round(response.hum);
49                 }
50             };
51             xhttp.open("POST", "dht11_ajax", true);
52             xhttp.send();
53         }
54     </script>
55     </body>
56 </html>
57 )=====;
```

```
58
59 void setup()
60 {
61     // Setup serial communication
62     Serial.begin(115200);
63     // Initialize DHT sensor
```

```
64     dht.begin();  
65  
66     // *** Connect to a WiFi access point ***  
67     Serial.printf("Connecting to %s ...\\n", ssid);  
68     WiFi.mode(WIFI_STA);  
69     WiFi.begin(ssid, pass);  
70     if (WiFi.waitForConnectResult() != WL_CONNECTED)  
71     {  
72         Serial.printf("WiFi connect failed! Rebooting ...\\n");  
73         delay(1000);  
74         ESP.restart();  
75     }  
76     Serial.printf("Connected\\n");  
77     Serial.printf("IP address: %s\\n", WiFi.localIP()  
78         .toString().c_str());  
79  
80     // Handler for root request  
81     server.on("/",  
82         HTTP_GET, [] (AsyncWebServerRequest *request)  
83     {  
84         request->send_P(200, "text/html", webpage);  
85     });  
86  
87     // Handler for DHT11 AJAX request  
88     server.on("/dht11_ajax",  
89         HTTP_POST, [] (AsyncWebServerRequest *request)  
90     {  
91         // Read temperature and humidity  
92         float temp = dht.readTemperature();  
93         float hum = dht.readHumidity();  
94         printf("Temperature: %.2f, Humidity: %.2f\\n",  
95             temp, hum);  
96  
97         // Send JSON response  
98         AsyncResponseStream *response =  
99             request->beginResponseStream("application/json");  
100        DynamicJsonBuffer jsonBuffer;
```

```
101     JsonObject &dht11 = jsonBuffer.createObject();
102     dht11["temp"] = String(temp);
103     dht11["hum"] = String(hum);
104     dht11.printTo(*response);
105     request->send(response);
106   );
107
108 // Start server
109 server.begin();
110 }
111
112 void loop()
113 {
114 }
```

Here is the step-by-step to create a web server for reading the DHT11 sensor with JavaScript:

1. First, you need to define the HTML code as in **line 25-57**.
2. Then, in **line 64**, you need to initialize the DHT11.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 67-78**.
4. Next, in **line 81-85**, you need to define a handler for the `/` request.
5. Next, in **line 88-106**, you need to define a handler for the HTTP POST from AJAX request. Within the handler, you need to read the temperature and humidity (**line 92 and 93**), and then send them in JSON format to the client (**line 98-105**).
6. Finally, in **line 109**, you need to start the web server.

Here is the details on how the web page works. The temperature and humidity value are defined within `<p>` tag as

```
<p>Temperature:  
    <b><span id="temp_value">N/A</span>&deg;C</b></p>  
<p>Humidity:  
    <b><span id="hum_value">N/A</span>%</b></p>
```

We use `` tags to mark up the text for temperature and humidity. We are going to update them from the JavaScript. The JavaScript code is defined as

```
<script>  
    setInterval(function() {  
        getDHT11Value();  
    }, 1000);  
    function getDHT11Value() {  
        var xhttp = new XMLHttpRequest();  
        xhttp.onreadystatechange = function() {  
            if (this.readyState == 4 && this.status == 200) {  
                var response = JSON.parse(this.responseText);  
                document.getElementById("temp_value")  
                    .innerHTML = Math.round(response.temp);  
                document.getElementById("hum_value")  
                    .innerHTML = Math.round(response.hum);  
            }  
        };  
        xhttp.open("POST", "dht11_ajax", true);  
        xhttp.send();  
    }  
</script>
```

We call `setInterval` method to execute the `getDHT11Value` function every one second. Within the `getDHT11Value` function, we create an `XMLHttpRequest` object. Then, we define a handler to process the HTTP response from the server. The handler will update the temperature and humidity value to the text that are marked up with `` tags. Finally, we set the HTTP method to POST, the URL to `dht11_ajax`, and then send it to the server.

We use JSON to format the temperature and humidity value. JSON stands for JavaScript Object Notation. It is a lightweight data interchange format. It is easy for humans to read and understand the JSON format. The JSON format for this example is defined as

```
{"temp": "29.00", "hum": "88.00"}
```

JSON objects are written in key/value pairs. A Key and value are separated by a colon, and each key/value pair is separated by a comma. To create and parse JSON in Arduino, you can use ArduinoJson [22] library.

17.5 Wrapping Up: AJAX Technique

In AJAX, HTML elements are associated with JavaScript event handlers. When an event is occurred (e.g. the page is loaded, a button is clicked), an `XMLHttpRequest` object is created by JavaScript. After that, the object sends an HTTP request to a web server. Next, the server processes the request, and sends a response back to the web page. Finally, the response is read and a proper action (like page update) is performed by JavaScript.

17.6 Summary

In this chapter, you have learned how to exchange data between client and server using AJAX technique. With AJAX, you can exchange data between client and server behind the scenes. You can also update parts of the web page without reloading the whole page.

Chapter 18

SPI Flash File System

What will you learn in this chapter?

- Separating front-end code from back-end code.
- Store web files in SPI flash file system.

18.1 SPI Flash File System

So far, we have always included the front-end code (HTML and JS) in our Arduino code as raw string literals. However, the problem is as soon as your project gets bigger this becomes hard to maintain. SPI Flash File System (SPIFFS) is a light-weight file system for microcontrollers with an SPI flash chip. With SPIFFS, you can store the front-end code as files as if it was in your computer. Therefore, you don't need to define them as raw string literals.

The ESP32 chip has 4-16 MB of SPI flash memory depending on the version. So, it has plenty of space for storing your web pages. To use the SPIFFS, you just need to include the library, and then you



Figure 18.1. ESP32 flash memory partition scheme.

can upload the files to the SPIFFS from the Arduino IDE by using Arduino ESP32 file system uploader [23]. To install the plugin, you should follow the instructions there. You can also change the partition scheme of the SPIFFS. Figure 18.1 shows ESP32 flash memory partition scheme. The flash memory is divided into two partitions, which are for application (APP) and data (SPIFFS).

18.2 Example Code: Web Server using SPIFFS

In this example code, you are going to create a web server that stores its web pages in the SPIFFS. The code is shown in Listing 18.1. We are going to create a simple web page in an HTML file named as `index.html`. We are going to upload this file into the SPIFFS. Then, when the web browser requests the file, web server reads the file from SPIFFS and sends it to the web browser.

Listing 18.1. Web server using SPIFFS

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
6 #include <ESPAsyncWebServer.h>
```

```
7
8 const char ssid[] = "Huawei-E5573";
9 const char pass[] = "huaweie5573";
10
11 // TCP server port for HTTP
12 const uint16_t port = 80;
13
14 // Create AsyncWebServer object
15 AsyncWebServer server(port);
16
17 void setup()
18 {
19     // Setup serial communication
20     Serial.begin(115200);
21
22     // Initialize SPIFFS
23     if (!SPIFFS.begin(true))
24     {
25         Serial.println("An error has occurred while \
26                         mounting SPIFFS");
27         return;
28     }
29
30     // *** Connect to a WiFi access point ***
31     Serial.printf("Connecting to %s ...\\n", ssid);
32     WiFi.mode(WIFI_STA);
33     WiFi.begin(ssid, pass);
34     if (WiFi.waitForConnectResult() != WL_CONNECTED)
35     {
36         Serial.printf("WiFi connect failed! Rebooting ...\\n");
37         delay(1000);
38         ESP.restart();
39     }
40     Serial.printf("Connected\\n");
41     Serial.printf("IP address: %s\\n", WiFi.localIP()
42                     .toString().c_str());
43
```

```
44 // Handler for root request
45 server.on("/", 
46     HTTP_GET, [] (AsyncWebServerRequest *request) {
47     {
48         request->send(SPIFFS, "/index.html");
49     });
50
51 // Start server
52 server.begin();
53 }
54
55 void loop()
56 {
57 }
```

Here is the step-by-step to create a web server using SPIFFS:

1. First, you need to include the SPIFFS library as in **line 4**.
2. Then, you need to initialize the SPIFFS as in **line 22-28**.
3. After that, in **line 45-49**, you need to set a handler for the `/` request. Note that in the `send` function, in **line 48**, we call the `index.html` that is stored in the SPIFFS.
4. Finally, in **line 52**, you need to start the web server.

In order to upload the `index.html` into the SPIFFS, you need to put the file within a folder named as 'Data'. As an example, Figure 18.2 shows the folder structure of this project. To upload the files inside the 'Data' folder, you can go to menu **Tools** → **ESP32 Sketch Data Upload**.

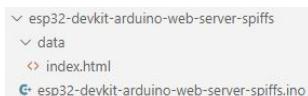


Figure 18.2. Folder structure of the project.

18.3 Summary

In this chapter, you have learned how to create a web server using SPIFFS. By using this technique, you can separate the front-end code (HTML, CSS, and JS) from the back-end code (C/C++). The front-end code can be stored as files as if it was in your computer.

Chapter 19

Cascading Style Sheets (CSS)

What will you learn in this chapter?

- The concept of Cascading Style Sheets (CSS) for front-end.
- Style web pages using Bootstrap (a CSS library).

19.1 Cascading Style Sheets

So far, we have learned about HTML and JavaScript. The last language of the three languages for front-end development is Cascading Style Sheets (CSS). With only HTML and JavaScript, we can create fully functional web pages, but it is like building a house without any colors and design. CSS is used to describe the style of an HTML document. So, we can make our web pages more attractive.

There are three ways to add CSS to the HTML elements:

- **Inline:** the CSS is written in the `style` attribute of an HTML element that you want to apply that style to.

- **Internal:** the CSS is written in the `<style>` tag in the `<head>` section.
- **External:** the CSS is written in a separate `.css` file.

For a small and simple project, I would recommend you to write the CSS in the `<style>` tag. But, for a big and complex project, I would recommend you to write the CSS in a separate CSS file. Here is an example of CSS code to change the color of a text using inline CSS:

```
<html>
  <head>
    <title>ESP32 Web Page</title>
  </head>
  <body>
    <p style="color: blue;">This is a blue text.</p>
  </body>
</html>
```

You can do the same thing using internal CSS as follows:

```
<html>
  <head>
    <title>ESP32 Web Page</title>
    <style>
      p {color: blue;}
    </style>
  </head>
  <body>
    <p>This is a blue text.</p>
  </body>
</html>
```

To use an external CSS file, you need to add a link to it in the `<head>` section:

```
<html>
  <head>
```



Figure 19.1. CSS syntax. Retrieved June 1, 2020, from w3schools.com.

```
<title>ESP32 Web Page</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
    <p>This is a blue text.</p>
</body>
</html>
```

Here is how the 'style.css' looks:

```
p {
    color: blue;
}
```

The CSS syntax is shown in Figure 19.1. The selector points to the HTML element that you want to style. Within the curly brackets, you can put declarations of styles separated with semicolons. Each declaration has a name–value pair separated by a colon.

In this book, we are going to use a CSS library called Bootstrap to style the previous web pages. The entire CSS syntax and rules are beyond the scope of this book. You can refer to this tutorial [24] instead.

19.2 Bootstrap

Bootstrap is a free and open-source CSS library. It helps you design websites faster and easier. It helps you create responsive designs¹.

¹ Responsive web design is an approach to design websites that automatically adjust themselves for a variety of devices and screen sizes.

Bootstrap is originally created by developers at Twitter, and then it is released as an open source product in 2011.

There are two ways to use Bootstrap in your websites. First, you can download Bootstrap from getbootstrap.com, and follow the instructions there. Then, you can add links to them in the `<head>` section as shown in the following codes. It is assumed that you put the files in the same directory as the HTML file.

```
<head>
  <title>ESP32 Web Page</title>
  <link rel="stylesheet" href="bootstrap.min.css">
  <script src="jquery-3.3.1.min.js"></script>
  <script src="popper.min.js"></script>
  <script src="bootstrap.min.js"></script>
</head>
```

The second way to import Bootstrap is by using content delivery network (CDN). CDN is a group of geographically distributed servers which provide fast delivery of Internet content. With CDN, you don't need to download Bootstrap to your computer and then store it in ESP32, but your ESP32 needs to be connected to the Internet. In order to use CDN, you just need to add the CDN links as shown in the following code.

```
<head>
  <title>ESP32 Web Page</title>
  <link rel="stylesheet" href="https://stackpath
    .bootstrapcdn.com/bootstrap/4.3.1/css
    /bootstrap.min.css">
  <script src="https://code.jquery.com
    /jquery-3.3.1.min.js"></script>
  <script src="https://cdnjs.cloudflare.com
    /ajax/libs/popper.js/1.14.7/umd
    /popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com
    /bootstrap/4.3.1/js
```

```
/bootstrap.min.js"></script>
</head>
```

There are three examples in this chapter. First, we are going to style a text using Bootstrap label. Second, we are going to style a button using Bootstrap button. Third, we are going to create a container for other HTML elements using Bootstrap card.

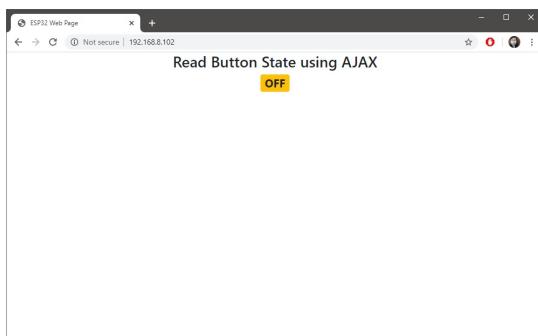


Figure 19.2. Using Bootstrap badge class to style a text.

19.3 Example Code: Bootstrap Label

In the first example code, you are going to style a text using Bootstrap badge class. The text displays a physical button state that is connected to the ESP32. The result is shown in Figure 19.2. The C code for this example is shown in Listing 19.1.

Listing 19.1. C code for styling a text using Bootstrap badge class

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
```

```
6 #include <ESPAsyncWebServer.h>
7
8 // Button's GPIO pin
9 #define BUTTON 12
10
11 const char ssid[] = "Huawei-E5573";
12 const char pass[] = "huaweiie5573";
13
14 // TCP server port for HTTP
15 const uint16_t port = 80;
16
17 // Create AsyncWebServer object
18 AsyncWebServer server(port);
19
20 void setup()
21 {
22     // Set GPIO pin as input for button
23     pinMode(BUTTON, INPUT_PULLUP);
24     // Setup serial communication
25     Serial.begin(115200);
26
27     // Initialize SPIFFS
28     if (!SPIFFS.begin(true))
29     {
30         Serial.println("An error has occurred while \
31             mounting SPIFFS");
32         return;
33     }
34
35     // *** Connect to a WiFi access point ***
36     Serial.printf("Connecting to %s ...\\n", ssid);
37     WiFi.mode(WIFI_STA);
38     WiFi.begin(ssid, pass);
39     if (WiFi.waitForConnectResult() != WL_CONNECTED)
40     {
41         Serial.printf("WiFi connect failed! Rebooting ...\\n");
42         delay(1000);
```

```
43     ESP.restart();
44 }
45 Serial.printf("Connected\n");
46 Serial.printf("IP address: %s\n", WiFi.localIP()
47     .toString().c_str());
48
49 // Handler for root request
50 server.on("/", 
51     HTTP_GET, [] (AsyncWebServerRequest *request)
52 {
53     request->send(SPIFFS, "/index.html");
54 });
55
56 // Handler for button AJAX request
57 server.on("/button_ajax",
58     HTTP_GET, [] (AsyncWebServerRequest *request)
59 {
60     if (digitalRead(BUTTON) )
61     {
62         request->send(200, "text/plain", "OFF");
63     }
64     else
65     {
66         request->send(200, "text/plain", "ON");
67     }
68 });
69
70 // Handler for Bootstrap libraries
71 server.on("/bootstrap.min.css",
72     HTTP_GET, [] (AsyncWebServerRequest *request) {
73     request->send(SPIFFS, "/bootstrap.min.css");
74 });
75 server.on("/jquery-3.3.1.min.js",
76     HTTP_GET, [] (AsyncWebServerRequest *request) {
77     request->send(SPIFFS, "/jquery-3.3.1.min.js");
78 });
79 server.on("/popper.min.js",
```

```
80     HTTP_GET, [](AsyncWebServerRequest *request) {
81     request->send(SPIFFS, "/popper.min.js");
82   });
83   server.on("/bootstrap.min.js",
84     HTTP_GET, [](AsyncWebServerRequest *request) {
85     request->send(SPIFFS, "/bootstrap.min.js");
86   });
87
88 // Start server
89 server.begin();
90 }
91
92 void loop()
93 {
94 }
```

Here is the step-by-step how to create the web server:

1. First, in **line 23**, you need to initialize a GPIO pin as input for a button.
2. Then, you need to initialize the SPIFFS as in **line 28-33**.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 36-47**.
4. Next, in **line 50-54**, you need to define a handler for the `/` request.
5. Next, in **line 57-68**, you need to define a handler for the HTTP GET from AJAX request. Within the handler, you need to read the button state, and then send the state as plain text to the client.
6. Next, in **line 71-86**, you need to define handlers for Bootstrap library files.

- Finally, in **line 89**, you need to start the web server.

Subsequently, you need to create the HTML code in a separate file named as `index.html` as shown in Listing 19.2. You need to put this file together with the Bootstrap library files inside the 'Data' folder.

Listing 19.2. HTML code for styling a text using Bootstrap badge class

```
1 <html>
2 <head>
3   <title>ESP32 Web Page</title>
4   <link rel="stylesheet" href="bootstrap.min.css">
5   <script src="jquery-3.3.1.min.js"></script>
6   <script src="popper.min.js"></script>
7   <script src="bootstrap.min.js"></script>
8 </head>
9 <body>
10  <div class="container-fluid">
11    <h3 class="text-center">
12      Read Button State using AJAX
13    </h3>
14    <h3 class="text-center">
15      <strong>
16        <span id="button_state"
17          class="badge badge-warning">
18          N/A</span>
19      </strong>
20    </h3>
21  </div>
22  <script>
23    setInterval(function() {
24      getButtonState();
25    }, 1000);
26    function getButtonState() {
27      var xhttp = new XMLHttpRequest();
28      xhttp.onreadystatechange = function() {
29        if (this.readyState == 4
```

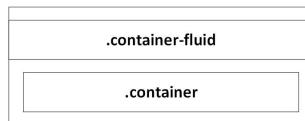


Figure 19.3. Bootstrap containers.

```
30             && this.status == 200) {
31                 document.getElementById("button_state")
32                     .innerHTML = this.responseText;
33             }
34         };
35         xhttp.open("GET", "button_ajax", true);
36         xhttp.send();
37     }
38     </script>
39 </body>
40 </html>
```

Here we use four Bootstrap classes:

- `.container-fluid`: in **line 10**, we use this class to provide a full width container that span the entire width of the viewport. Instead of `.container-fluid`, you can also use `.container`. The difference is illustrated in Figure 19.3.
- `.text-center`: in **line 11 and 14**, we use this class to center text horizontally.
- `.badge` and `.badge-warning`: in **line 17**, we use this class to provide background color to the text. You can choose other colors as shown in Figure 19.4.



Figure 19.4. Bootstrap badge variations [25].

19.4 Example Code: Bootstrap Button

In the second example code, you are going to style HTML buttons using Bootstrap button class. We are going to style two HTML buttons for turning on and off an LED on the ESP32. The result is shown in Figure 19.5. We also center the content of web page both horizontally and vertically. The C code for this example is shown in Listing 19.3.

Listing 19.3. C code for styling HTML buttons using Bootstrap button class

```

1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
6 #include <ESPAsyncWebServer.h>
7
8 // On-board LED's GPIO pin
9 #define LED_ON_BOARD 2
10
11 const char ssid[] = "Huawei-E5573";
12 const char pass[] = "huaweiie5573";
13
14 // TCP server port for HTTP
15 const uint16_t port = 80;
16

```

```
17 // Create AsyncWebServer object
18 AsyncWebServer server(port);
19
20 void setup()
21 {
22     // Set GPIO pin as output for on-board LED
23     pinMode(LED_ONBOARD, OUTPUT);
24     // Setup serial communication
25     Serial.begin(115200);
26
27     // Initialize SPIFFS
28     if (!SPIFFS.begin(true))
29     {
30         Serial.println("An error has occurred while \
31             mounting SPIFFS");
32         return;
33     }
34
35     // *** Connect to a WiFi access point ***
36     Serial.printf("Connecting to %s ...\\n", ssid);
37     WiFi.mode(WIFI_STA);
38     WiFi.begin(ssid, pass);
39     if (WiFi.waitForConnectResult() != WL_CONNECTED)
40     {
41         Serial.printf("WiFi connect failed! Rebooting ...\\n");
42         delay(1000);
43         ESP.restart();
44     }
45     Serial.printf("Connected\\n");
46     Serial.printf("IP address: %s\\n", WiFi.localIP()
47         .toString().c_str());
48
49     // Handler for root request
50     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
51     {
52         request->send(SPIFFS, "/index.html");
```

```
54     });
55
56     // Handler for turning on the LED
57     server.on("/on",
58         HTTP_POST, [](AsyncWebServerRequest *request) {
59         digitalWrite(LED_ON_BOARD, HIGH);
60         request->send(303);
61     });
62
63     // Handler for turning off the LED
64     server.on("/off",
65         HTTP_POST, [](AsyncWebServerRequest *request) {
66         digitalWrite(LED_ON_BOARD, LOW);
67         request->send(303);
68     });
69
70     // Handler for Bootstrap libraries
71     server.on("/bootstrap.min.css",
72         HTTP_GET, [](AsyncWebServerRequest *request) {
73         request->send(SPIFFS, "/bootstrap.min.css");
74     });
75     server.on("/jquery-3.3.1.min.js",
76         HTTP_GET, [](AsyncWebServerRequest *request) {
77         request->send(SPIFFS, "/jquery-3.3.1.min.js");
78     });
79     server.on("/popper.min.js",
80         HTTP_GET, [](AsyncWebServerRequest *request) {
81         request->send(SPIFFS, "/popper.min.js");
82     });
83     server.on("/bootstrap.min.js",
84         HTTP_GET, [](AsyncWebServerRequest *request) {
85         request->send(SPIFFS, "/bootstrap.min.js");
86     });
87
88     // Start server
89     server.begin();
90 }
```

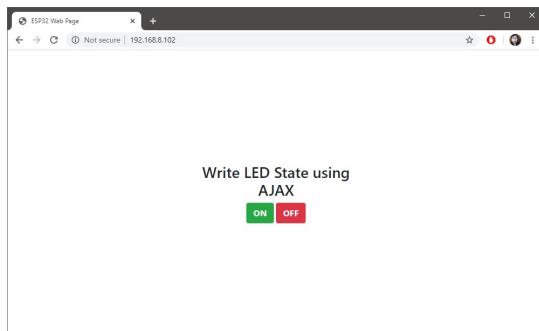


Figure 19.5. Using Bootstrap button class to style HTML buttons.

```
91
92 void loop()
93 {
94 }
```

Here is the step-by-step how to create the web server:

1. First, in **line 23**, you need to initialize the GPIO pin as output for the on-board LED.
2. Then, you need to initialize the SPIFFS as in **line 28-33**.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 36-47**.
4. Next, in **line 50-54**, you need to define a handler for the `/` request.
5. Next, in **line 57-68**, you need to define a handler for the HTTP GET from AJAX request. Within the handler, you need to turn on or off the LED.

6. Next, in **line 71-86**, you need to define handlers for Bootstrap library files.
7. Finally, in **line 89**, you need to start the web server.

Subsequently, you need to create the HTML code in a separate file named as `index.html` as shown in Listing 19.4.

Listing 19.4. HTML code for styling HTML buttons using Bootstrap button class

```
1 <html>
2 <head>
3   <title>ESP32 Web Page</title>
4   <link rel="stylesheet" href="bootstrap.min.css">
5   <script src="jquery-3.3.1.min.js"></script>
6   <script src="popper.min.js"></script>
7   <script src="bootstrap.min.js"></script>
8 </head>
9 <body>
10  <div class="container-fluid">
11    <div class="row h-100">
12      <div class="col-4 offset-4 my-auto">
13        <h3 class="text-center">
14          Write LED State using AJAX
15        </h3>
16        <div class="text-center">
17          <button id="btn_on" class="btn btn-success">
18            <strong>ON</strong>
19          </button>
20          <button id="btn_off" class="btn btn-danger">
21            <strong>OFF</strong>
22          </button>
23        </div>
24      </div>
25    </div>
26    <div>
27      <script>
```

```
28     document.getElementById("btn_on")
29         .onclick = function() {
30             var xhttp = new XMLHttpRequest();
31             xhttp.open("POST", "on", true);
32             xhttp.send();
33         };
34     document.getElementById("btn_off")
35         .onclick = function() {
36             var xhttp = new XMLHttpRequest();
37             xhttp.open("POST", "off", true);
38             xhttp.send();
39         };
40     </script>
41 </body>
42 </html>
```

Here we use Bootstrap grid system [26]. It uses containers, rows, and columns classes to layout and align contents:

- `.row` and `.h-100`: `.row` is used to make a row within the main container. `.h-100` is used to make a content take up 100% height of the web browser.
- `col-4`, `offset-4`, and `my-auto`: `.col-4` is used to make a column with size 4 within the row container. `offset-4` is used to offset the column by 4. `my-auto` is used to center the content vertically.

We also use button classes (`.btn`, `.btn-success`, and `.btn-danger`) to change the style of the default HTML buttons. You can choose other colors as shown in Figure 19.6.



Figure 19.6. Bootstrap button variations [27].

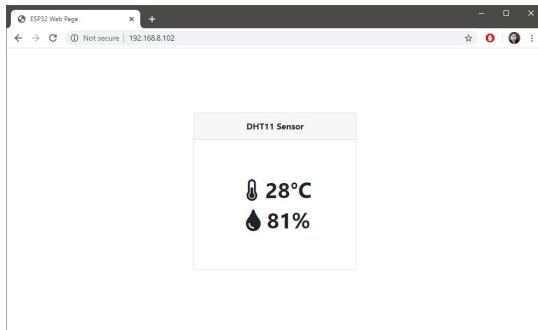


Figure 19.7. Using Bootstrap card class to create a container for other HTML elements.

19.5 Example Code: Bootstrap Card

In the third example code, you are going to create a container for other HTML elements using Bootstrap card class. We are going to wrap two texts that display temperature and humidity from DHT11 with the Boostrap card. The result is shown in Figure 19.7. The C code for this example is shown in Listing 19.5.

Listing 19.5. C code for creating a container for other HTML elements using Bootstrap card class

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
6 #include <ESPAsyncWebServer.h>
7 #include <DHT.h>
8 #include <ArduinoJson.h>
9
10 #define DHTPIN 22
11 #define DHTTYPE DHT11
12
13 // Setup DHT pin and type
14 DHT dht(DHTPIN, DHTTYPE);
15
16 const char ssid[] = "Huawei-E5573";
17 const char pass[] = "huaweiie5573";
18
19 // TCP server port for HTTP
20 const uint16_t port = 80;
21
22 // Create AsyncWebServer object
23 AsyncWebServer server(port);
24
25 void setup()
26 {
27     // Setup serial communication
28     Serial.begin(115200);
29     // Initialize DHT sensor
30     dht.begin();
31
32     // Initialize SPIFFS
33     if (!SPIFFS.begin(true))
34     {
35         Serial.println("An error has occurred while \\"
```

```
36         mounting SPIFFS");
37     return;
38 }
39
40 // *** Connect to a WiFi access point ***
41 Serial.printf("Connecting to %s ...\\n", ssid);
42 WiFi.mode(WIFI_STA);
43 WiFi.begin(ssid, pass);
44 if (WiFi.waitForConnectResult() != WL_CONNECTED)
45 {
46     Serial.printf("WiFi connect failed! Rebooting ...\\n");
47     delay(1000);
48     ESP.restart();
49 }
50 Serial.printf("Connected\\n");
51 Serial.printf("IP address: %s\\n", WiFi.localIP()
52             .toString().c_str());
53
54 // Handler for root request
55 server.on("/", 
56     HTTP_GET, [] (AsyncWebServerRequest *request)
57 {
58     request->send(SPIFFS, "/index.html");
59 });
60
61 // Handler for DHT11 AJAX request
62 server.on("/dht11_ajax",
63     HTTP_POST, [] (AsyncWebServerRequest *request)
64 {
65     // Read temperature and humidity
66     float temp = dht.readTemperature();
67     float hum = dht.readHumidity();
68
69     // Send JSON response
70     AsyncResponseStream *response =
71         request->beginResponseStream("application/json");
72     DynamicJsonBuffer jsonBuffer;
```

```
73     JsonObject &dht11 = jsonBuffer.createObject();
74     dht11["temp"] = String(temp);
75     dht11["hum"] = String(hum);
76     dht11.printTo(*response);
77     request->send(response);
78 }
79
80 // Handler for Bootstrap libraries
81 server.on("/bootstrap.min.css",
82     HTTP_GET, [](AsyncWebServerRequest *request) {
83     request->send(SPIFFS, "/bootstrap.min.css");
84 });
85 server.on("/jquery-3.3.1.min.js",
86     HTTP_GET, [](AsyncWebServerRequest *request) {
87     request->send(SPIFFS, "/jquery-3.3.1.min.js");
88 });
89 server.on("/popper.min.js",
90     HTTP_GET, [](AsyncWebServerRequest *request) {
91     request->send(SPIFFS, "/popper.min.js");
92 });
93 server.on("/bootstrap.min.js",
94     HTTP_GET, [](AsyncWebServerRequest *request) {
95     request->send(SPIFFS, "/bootstrap.min.js");
96 });
97 server.on("/a076d05399.js",
98     HTTP_GET, [](AsyncWebServerRequest *request) {
99     request->send(SPIFFS, "/a076d05399.js");
100 });
101
102 // Start server
103 server.begin();
104 }
105
106 void loop()
107 {
108 }
```

Here is the step-by-step how to create the web server:

1. First, in **line 30**, you need to initialize DHT11 sensor.
2. Then, you need to initialize the SPIFFS as in **line 33-38**.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 41-52**.
4. Next, in **line 55-59**, you need to define a handler for the `/` request.
5. Next, in **line 62-78**, you need to define a handler for the HTTP POST from AJAX request. Within the handler, you need to read the temperature and humidity (**line 66 and 67**), and then send the them in JSON format to the client (**line 70-77**).
6. Next, in **line 81-100**, you need to define handlers for Bootstrap library files.
7. Finally, in **line 103**, you need to start the web server.

Subsequently, you need to create the HTML code in a separate file named as `index.html` as shown in Listing 19.6.

Listing 19.6. HTML code for creating a container for other HTML elements using Bootstrap card class

```
1 <html>
2 <head>
3   <title>ESP32 Web Page</title>
4   <link rel="stylesheet" href="bootstrap.min.css">
5   <script src="jquery-3.3.1.min.js"></script>
6   <script src="popper.min.js"></script>
7   <script src="bootstrap.min.js"></script>
8   <script src="a076d05399.js"></script>
9 </head>
10 <body>
```

```
11   <div class="container-fluid">
12     <div class="row h-100">
13       <div class="col-4 offset-4 my-auto">
14         <div class="card">
15           <div class="card-header text-center">
16             <strong>DHT11 Sensor</strong>
17           </div>
18           <div class="card-body text-center">
19             <div class="m-5">
20               <h1>
21                 <i class="fa-fw fas
22                   fa-thermometer-half"></i>
23                 <strong>
24                   <span id="temp_value">N/A</span>
25                   &deg;C
26                 </strong>
27               </h1>
28               <h1>
29                 <i class="fa-fw fas fa-tint"></i>
30                 <strong>
31                   <span id="hum_value">N/A</span>
32                   %
33                 </strong>
34               </h1>
35             </div>
36           </div>
37         </div>
38       </div>
39     </div>
40   <div>
41     <script>
42       setInterval(function() {
43         getDHT11Value();
44       }, 1000);
45       function getDHT11Value() {
46         var xhttp = new XMLHttpRequest();
47         xhttp.onreadystatechange = function() {
```

```
48     if (this.readyState == 4
49         && this.status == 200) {
50         var response = JSON.parse(this.responseText);
51         document.getElementById("temp_value")
52             .innerHTML = Math.round(response.temp);
53         document.getElementById("hum_value")
54             .innerHTML = Math.round(response.hum);
55     }
56 };
57 xhttp.open("POST", "dht11_ajax", true);
58 xhttp.send();
59 }
60 </script>
61 </body>
62 </html>
```

Here we use several Bootstrap classes:

- `.card`, `.card-header`, and `.card-body`: these classes are used to create card container. We define the title within the `.card-header` as in **line 16**, and the content within the `.card-body` as in **line 19-35**.
- `.fa-fw`, `.fas`, `.fa-thermometer-half`, and `.fa-tint`: these classes are used to create temperature and humidity icons.

19.6 Summary

In this chapter, we have learned how to use Bootstrap which is a CSS library to style the HTML web pages. With Bootstrap, you can style your web pages faster and easier.

Chapter 20

Gauge and Chart

What will you learn in this chapter?

- Streaming sensor readings to a real-time gauge and chart.

20.1 Gauge and Chart

So far, we have used a simple text with `<p>` tag to display sensor readings. We have also used Bootstrap to style the web pages. In this chapter, we are going to use a real-time gauge and chart to stream sensor readings.

Here, we are going to use Bootstrap together with these two libraries to stream sensor readings.

- **JustGage** [28]: this is a JavaScript library that provides animated gauges.
- **Highcharts** [29]: this is a JavaScript library that provides charts for data visualization.

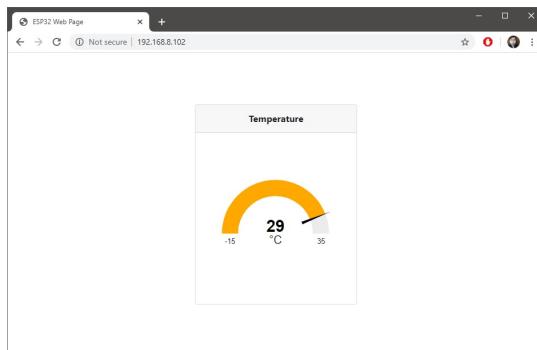


Figure 20.1. Using a gauge to display real-time sensor readings.

20.2 Example Code: Gauge

In the first example code, you are going to use a gauge to display real-time sensor readings. The result is shown in Figure 20.1. The C code for this example is shown in Listing 20.1.

Listing 20.1. C code for streaming sensor readings to a real-time gauge

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
6 #include <ESPAsyncWebServer.h>
7 #include <DHT.h>
8 #include <ArduinoJson.h>
9
10 #define DHTPIN 22
11 #define DHTTYPE DHT11
12
13 // Setup DHT pin and type
14 DHT dht(DHTPIN, DHTTYPE);
```

```
15
16 const char ssid[] = "Huawei-E5573";
17 const char pass[] = "huaweie5573";
18
19 // TCP server port for HTTP
20 const uint16_t port = 80;
21
22 // Create AsyncWebServer object
23 AsyncWebServer server(port);
24
25 // Global variable for storing temperature value
26 int temp;
27
28 void setup()
29 {
30     // Setup serial communication
31     Serial.begin(115200);
32     // Initialize DHT sensor
33     dht.begin();
34
35     // Initialize SPIFFS
36     if (!SPIFFS.begin(true))
37     {
38         Serial.println("An error has occurred while \
39                         mounting SPIFFS");
40         return;
41     }
42
43     // *** Connect to a WiFi access point ***
44     Serial.printf("Connecting to %s ...\\n", ssid);
45     WiFi.mode(WIFI_STA);
46     WiFi.begin(ssid, pass);
47     if (WiFi.waitForConnectResult() != WL_CONNECTED)
48     {
49         Serial.printf("WiFi connect failed! Rebooting ...\\n");
50         delay(1000);
51         ESP.restart();
```

```
52     }
53     Serial.printf("Connected\n");
54     Serial.printf("IP address: %s\n", WiFi.localIP()
55         .toString().c_str()));
56
57     // Handler for root request
58     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
59     {
60         request->send(SPIFFS, "/index.html");
61     });
62
63     // Handler for DHT11 AJAX request
64     server.on("/temperature",
65         HTTP_POST, [] (AsyncWebServerRequest *request)
66     {
67         // Send temperature
68         request->send(200, "text/plain",
69             readDHT11Temperature());
70     });
71
72     // Handler for Bootstrap libraries
73     server.on("/bootstrap.min.css",
74         HTTP_GET, [] (AsyncWebServerRequest *request) {
75         request->send(SPIFFS, "/bootstrap.min.css");
76     });
77     server.on("/jquery-3.3.1.min.js",
78         HTTP_GET, [] (AsyncWebServerRequest *request) {
79         request->send(SPIFFS, "/jquery-3.3.1.min.js");
80     });
81     server.on("/popper.min.js",
82         HTTP_GET, [] (AsyncWebServerRequest *request) {
83         request->send(SPIFFS, "/popper.min.js");
84     });
85     server.on("/bootstrap.min.js",
86         HTTP_GET, [] (AsyncWebServerRequest *request) {
87         request->send(SPIFFS, "/bootstrap.min.js");
```

```
89     });
90     server.on("/raphael-2.1.4.min.js",
91         HTTP_GET, [](AsyncWebServerRequest *request) {
92         request->send(SPIFFS, "/raphael-2.1.4.min.js");
93     });
94     server.on("/justgage.js",
95         HTTP_GET, [](AsyncWebServerRequest *request) {
96         request->send(SPIFFS, "/justgage.js");
97     });
98
99     // Start server
100    server.begin();
101 }
102
103 void loop()
104 {
105 }
106
107 String readDHT11Temperature()
108 {
109     // Read temperature
110     int t = (int)dht.readTemperature();
111     // Update temperature value only
112     // if it is not NaN and between -15 and 35
113     temp = (isnan(t)) ? temp :
114         (((t >= -15) && (t <= 35)) ? t : temp);
115
116     return String(temp);
117 }
```

Here is the step-by-step how to create the web server:

1. First, in **line 33**, you need to initialize DHT11 sensor.
2. Then, you need to initialize the SPIFFS as in **line 36-41**.

3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 44-55**.
4. Next, in **line 58-62**, you need to define a handler for the `/` request.
5. Next, in **line 65-71**, you need to define a handler for the HTTP POST from AJAX request. Within the handler, you need to read the temperature by calling `readDHT11Temperature` function, and then send the it as plain text to the client.
6. The `readDHT11Temperature` function is defined in **line 107-117**.
7. Next, in **line 74-97**, you need to define handlers for Bootstrap and JustGage libraries.
8. Finally, in **line 100**, you need to start the web server.

Subsequently, you need to create the HTML code in a separate file named as `index.html` as shown in Listing 20.2.

Listing 20.2. HTML code for streaming sensor readings to a real-time gauge

```
1 <html>
2 <head>
3   <meta http-equiv="Content-Type" content="text/html;
4     charset=utf-8">
5   <title>ESP32 Web Page</title>
6   <link rel="stylesheet" href="bootstrap.min.css">
7   <script src="jquery-3.3.1.min.js"></script>
8   <script src="popper.min.js"></script>
9   <script src="bootstrap.min.js"></script>
10  <script src="raphael-2.1.4.min.js"> </script>
11  <script src="justgage.js"> </script>
12  <style>
13    .card {
14      min-height: 350px;
```

```
15      }
16      .card-body {
17          display: flex;
18          flex-direction: column;
19          align-items: center;
20          justify-content: center;
21      }
22  </style>
23 </head>
24 <body>
25  <div class="container-fluid">
26      <div class="row h-100">
27          <div class="col-4 offset-4 my-auto">
28              <div class="card">
29                  <div class="card-header text-center">
30                      <strong>Temperature</strong>
31                  </div>
32                  <div class="card-body text-center">
33                      <div id="gauge_temp" style="width:250px;
34                          height:250px"></div>
35                  </div>
36              </div>
37          </div>
38      </div>
39  <div>
40      <script>
41          // *** Gauge for temperature ***
42          var gauge_temp;
43          gauge_temp = new JustGage({
44              id: "gauge_temp",
45              min: -15,
46              max: 35,
47              donut: false,
48              pointer: true,
49              gaugeWidthScale: 0.8,
50              counter: true,
51              hideInnerShadow: true,
```

```
52     title: "",  
53     titlePosition: "below",  
54     levelColors: ["#ffa800", "#ffa800", "#ffa800"],  
55     titleFontColor : "#292b2c",  
56     label: "C",  
57     labelFontColor: "#292b2c",  
58     labelMinFontSize: 16,  
59     relativeGaugeSize: true  
60   });  
61  
62   setInterval(function() {  
63     var xhttp = new XMLHttpRequest();  
64     xhttp.onreadystatechange = function() {  
65       if (this.readyState == 4  
66         && this.status == 200) {  
67         gauge_temp  
68           .refresh(parseInt(this.responseText));  
69       }  
70     };  
71     xhttp.open("POST", "/temperature", true);  
72     xhttp.send();  
73   }, 1000);  
74 </script>  
75 </body>  
76 </html>
```

Here is the step-by-step how to create the gauge:

1. First, you need to use Bootstrap main and card container as in the previous chapter.
2. Then, within the body of card container you need to add a placeholder for gauge as in **line 33-34**. You also need to add custom CSS styles to the card class as shown in **line 12-22**.
3. After that, in **line 42-60**, you need to create and customize the gauge.

- Finally, in **line 62-73**, you need to set a function that will be executed every one second. Within this function, you need to send an HTTP POST to request the temperature value. The server sends the temperature value, and the gauge value is refreshed by calling `refresh` method.

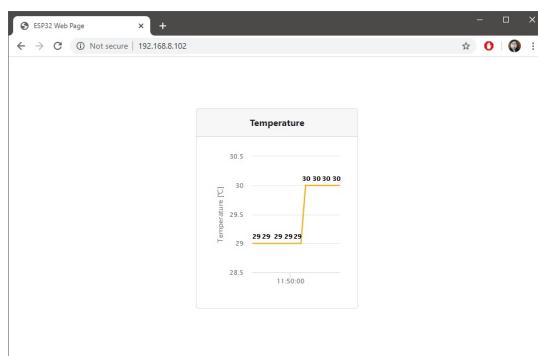


Figure 20.2. Using a chart to display real-time sensor readings.

20.3 Example Code: Chart

In the second example code, you are going to use a chart to display real-time sensor readings. The result is shown in Figure 20.2. The C code for this example is shown in Listing 20.3.

Listing 20.3. C code for streaming sensor readings to a real-time chart

```
1 // Author: Erwin Ouyang, aiotedge.tech
2 // Date   : 8 Mar 2020
3
4 #include <SPIFFS.h>
5 #include <WiFi.h>
6 #include <ESPAsyncWebServer.h>
```

```
7 #include <DHT.h>
8 #include <ArduinoJson.h>
9
10 #define DHTPIN 22
11 #define DHTTYPE DHT11
12
13 // Setup DHT pin and type
14 DHT dht(DHTPIN, DHTTYPE);
15
16 const char ssid[] = "Huawei-E5573";
17 const char pass[] = "huaweie5573";
18
19 // TCP server port for HTTP
20 const uint16_t port = 80;
21
22 // Create AsyncWebServer object
23 AsyncWebServer server(port);
24
25 // Global variable for storing temperature value
26 int temp;
27
28 void setup()
29 {
30     // Setup serial communication
31     Serial.begin(115200);
32     // Initialize DHT sensor
33     dht.begin();
34
35     // Initialize SPIFFS
36     if (!SPIFFS.begin(true))
37     {
38         Serial.println("An error has occurred while \
39                         mounting SPIFFS");
39         return;
40     }
41
42     // *** Connect to a WiFi access point ***
43 }
```

```
44     Serial.printf("Connecting to %s ...\\n", ssid);
45     WiFi.mode(WIFI_STA);
46     WiFi.begin(ssid, pass);
47     if (WiFi.waitForConnectResult() != WL_CONNECTED)
48     {
49         Serial.printf("WiFi connect failed! Rebooting ...\\n");
50         delay(1000);
51         ESP.restart();
52     }
53     Serial.printf("Connected\\n");
54     Serial.printf("IP address: %s\\n", WiFi.localIP()
55                 .toString().c_str());
56
57     // Handler for root request
58     server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
59     {
60         request->send(SPIFFS, "/index.html");
61     });
62
63
64     // Handler for DHT11 AJAX request
65     server.on("/temperature",
66             HTTP_POST, [] (AsyncWebServerRequest *request)
67     {
68         // Send temperature
69         request->send(200, "text/plain",
70                     readDHT11Temperature());
71     });
72
73
74     // Handler for Bootstrap libraries
75     server.on("/bootstrap.min.css",
76             HTTP_GET, [] (AsyncWebServerRequest *request) {
77         request->send(SPIFFS, "/bootstrap.min.css");
78     });
79     server.on("/jquery-3.3.1.min.js",
80             HTTP_GET, [] (AsyncWebServerRequest *request) {
81         request->send(SPIFFS, "/jquery-3.3.1.min.js");
```

```
81     });
82     server.on("/popper.min.js",
83         HTTP_GET, [](AsyncWebServerRequest *request) {
84         request->send(SPIFFS, "/popper.min.js");
85     });
86     server.on("/bootstrap.min.js",
87         HTTP_GET, [](AsyncWebServerRequest *request) {
88         request->send(SPIFFS, "/bootstrap.min.js");
89     });
90     server.on("/highcharts.js",
91         HTTP_GET, [](AsyncWebServerRequest *request) {
92         request->send(SPIFFS, "/highcharts.js");
93     });
94
95     // Start server
96     server.begin();
97 }
98
99 void loop()
100 {
101 }
102
103 String readDHT11Temperature()
104 {
105     // Read temperature
106     int t = (int)dht.readTemperature();
107     // Update temperature value only
108     // if it is not NaN and between -15 and 35
109     temp = (isnan(t)) ? temp :
110         (((t >= -15) && (t <= 35)) ? t : temp);
111
112     return String(temp);
113 }
```

Here is the step-by-step how to create the web server:

1. First, in **line 33**, you need to initialize DHT11 sensor.
2. Then, you need to initialize the SPIFFS as in **line 36-41**.
3. After that, you need to set the ESP32 as WiFi station, and connect it to an AP as in **line 44-55**.
4. Next, in **line 58-62**, you need to define a handler for the `/` request.
5. Next, in **line 65-71**, you need to define a handler for the HTTP POST from AJAX request. Within the handler, you need to read the temperature by calling `readDHT11Temperature` function, and then send the it as plain text to the client.
6. The `readDHT11Temperature` function is defined in **line 103-113**.
7. Next, in **line 73-93**, you need to define handlers for Bootstrap and Highcharts libraries.
8. Finally, in **line 96**, you need to start the web server.

Subsequently, you need to create the HTML code in a separate file named as `index.html` as shown in Listing 20.4.

Listing 20.4. HTML code for streaming sensor readings to a real-time chart

```
1 <html>
2 <head>
3   <meta http-equiv="Content-Type" content="text/html;
4     charset=utf-8">
5   <title>ESP32 Web Page</title>
6   <link rel="stylesheet" href="bootstrap.min.css">
7   <script src="jquery-3.3.1.min.js"></script>
8   <script src="popper.min.js"></script>
```

```
9   <script src="bootstrap.min.js"></script>
10  <script src="highcharts.js"> </script>
11  <style>
12    .card {
13      min-height: 350px;
14    }
15    .card-body {
16      display: flex;
17      flex-direction: column;
18      align-items: center;
19      justify-content: center;
20    }
21  </style>
22 </head>
23 <body>
24   <div class="container-fluid">
25     <div class="row h-100">
26       <div class="col-4 offset-4 my-auto">
27         <div class="card">
28           <div class="card-header text-center">
29             <strong>Temperature</strong>
30           </div>
31           <div class="card-body text-center"
32             id="chart_container">
33             <div id="chart_temperature"></div>
34           </div>
35         </div>
36       </div>
37     </div>
38   <div>
39     <script>
40       // *** Chart for number of users ***
41       var chartTemp = new Highcharts.Chart({
42         chart: {
43           renderTo : 'chart_temperature',
44         },
45         title: {
```

```
46         text: ''
47     },
48     series: [{
49         showInLegend: false,
50         data: []
51     }],
52     plotOptions: {
53         line: {
54             animation: false,
55             dataLabels: {
56                 enabled: true
57             }
58         },
59         series: {
60             color: '#ffa800'
61         }
62     },
63     xAxis: {
64         type: 'datetime',
65         dateLabelFormats: {
66             second: '%H:%M:%S'
67         }
68     },
69     yAxis: {
70         title: {
71             text: 'Temperature [C]'
72         }
73     },
74     credits: {
75         enabled: false
76     }
77 });
78
79 $("#chart_temperature").highcharts()
80     .setSize($("#chart_container").width(), 250);
81 $(window).on('resize', function(){
82     $("#chart_temperature").highcharts()
```

```
83         .setSize($("#chart_container").width(), 250);
84     });
85
86     setInterval(function() {
87         var xhttp = new XMLHttpRequest();
88         xhttp.onreadystatechange = function() {
89             if (this.readyState == 4
90                 && this.status == 200) {
91                 var x = (new Date()).getTime();
92                 var y = parseInt(this.responseText);
93
94                 if (chartTemp.series[0].data.length > 40) {
95                     chartTemp.series[0]
96                         .addPoint([x, y], true, true, true);
97                 } else {
98                     chartTemp.series[0]
99                         .addPoint([x, y], true, false, true);
100                }
101            }
102        };
103        xhttp.open("POST", "/temperature", true);
104        xhttp.send();
105    }, 1000);
106    </script>
107 </body>
108 </html>
```

Here is the step-by-step how to create the chart:

1. First, you need to use Bootstrap main and card container as in the previous chapter.
2. Then, within the body of card container you need to add a placeholder for chart as in **line 33**. You also need to add custom CSS styles to the card class as shown in **line 11-21**.

3. After that, in **line 41-84**, you need to create and customize the chart.
4. Finally, in **line 86-105**, you need to set a function that will be executed every one second. Within this function, you need to send an HTTP POST to request the temperature value. The server sends the temperature value, and a new data point is added to the chart by calling `addPoint` method.

20.4 Summary

In this chapter, we have learned how to use real-time gauge and chart to stream sensor readings. By using gauge and chart, we can make the user interface of the web page more attractive. Furthermore, you can integrate these gauge and chart to your IoT dashboard.

Bibliography

- [1] Wikipedia, “Internet of things.” https://en.wikipedia.org/wiki/Internet_of_things. Accessed on 2020-3-14.
- [2] Wikipedia, “Internet of things, applications.” https://en.wikipedia.org/wiki/Internet_of_things#Applications. Accessed on 2020-3-14.
- [3] Wikipedia, “Esp32.” <https://en.wikipedia.org/wiki/ESP32>. Accessed on 2020-3-14.
- [4] Espressif, “Esp32 modules.” <https://www.espressif.com/en/products/hardware/modules>. Accessed on 2020-3-14.
- [5] Espressif, “Esp32 development boards.” <https://www.espressif.com/en/products/hardware/development-boards>. Accessed on 2020-3-14.
- [6] Zerynth, “Doit esp32 devkit v1.” https://docs.zerynth.com/latest/official/board.zerynth.doit_esp32/docs/index.html. Accessed on 2020-3-14.
- [7] Wikipedia, “Wi-fi.” <https://en.wikipedia.org/wiki/Wi-Fi>. Accessed on 2020-3-15.

- [8] Wikipedia, “Wi-fi versions.” <https://en.wikipedia.org/wiki/Wi-Fi#Versions>. Accessed on 2020-3-15.
- [9] Wikipedia, “Arduino.” <https://en.wikipedia.org/wiki/Arduino>. Accessed on 2020-3-15.
- [10] Espressif, “Arduino core for the esp32.” <https://github.com/espressif/arduino-esp32>. Accessed on 2020-3-15.
- [11] M. N. Dev, “Espasyncwebserver.” <https://github.com/me-no-dev/ESPAsyncWebServer>. Accessed on 2020-3-15.
- [12] Zerynth, “Doit esp32 devkit v1 power.” https://docs.zerynth.com/latest/official/board.zerynth.doit_esp32/docs/index.html#power. Accessed on 2020-3-16.
- [13] Espressif, “Installation instructions using arduino ide boards manager.” https://github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/boards_manager.md. Accessed on 2020-3-15.
- [14] R. N. Tutorials, “Esp32 pinout reference: Which gpio pins should you use?” <https://randomnerdtutorials.com/esp32-pinout-reference-gpios/>. Accessed on 2020-3-17.
- [15] cplusplus, “printf.” wwwcplusplus.com/reference/cstdio/printf/. Accessed on 2018-10-16.
- [16] Adafruit, “Dht11 vs dht22.” <https://learn.adafruit.com/dht/overview>. Accessed on 2020-4-11.
- [17] Adafruit, “Dht-sensor-library.” <https://github.com/adafruit/DHT-sensor-library>. Accessed on 2020-4-11.

- [18] M. Integrated, “Ds1307.” <https://datasheets.maximintegrated.com/en/ds/DS1307.pdf>. Accessed on 2020-4-11.
- [19] Makuna, “Rtc.” <https://github.com/Makuna/Rtc>. Accessed on 2020-4-11.
- [20] M. N. Dev, “Espasyncwebserver.” <https://github.com/me-no-dev/ESPAsyncWebServer>. Accessed on 2020-4-13.
- [21] M. N. Dev, “Asynctcp.” <https://github.com/me-no-dev/AsyncTCP>. Accessed on 2020-4-13.
- [22] ArduinoJson, “Arduinojson.” <https://arduinojson.org/>. Accessed on 2020-5-2.
- [23] M. N. Dev, “Arduino esp32 filesystem uploader.” <https://github.com/me-no-dev/arduino-esp32fs-plugin>. Accessed on 2020-6-6.
- [24] w3schools.com, “Css tutorial.” <https://www.w3schools.com/css/default.asp/>. Accessed on 2020-6-1.
- [25] Bootstrap, “Badges.” <https://getbootstrap.com/docs/4.4/components/badge/>. Accessed on 2020-6-6.
- [26] Bootstrap, “Grid.” <https://getbootstrap.com/docs/4.0/layout/grid/>. Accessed on 2020-6-6.
- [27] Bootstrap, “Button.” <https://getbootstrap.com/docs/4.0/components/buttons/>. Accessed on 2020-6-6.
- [28] B. Djuricic, “Justgage.” <https://github.com/toorshia/justgage/>. Accessed on 2020-6-6.
- [29] Highsoft, “Highcharts.” <https://github.com/highcharts/highcharts/>. Accessed on 2020-6-6.