

# KHOÁ HỌC ESP32-IDF

Mentor: Ngô Vũ Trường Giang



<https://www.facebook.com/groups/deviot.vn>



<https://deviot.vn/>

# MỤC ĐÍCH CỦA KHÓA HỌC

- Nắm được các khái niệm của ESP32-IDF, cách sử dụng ESP32-IDF và có thể tự tìm hiểu nghiên cứu các ứng dụng IoT khác.
- Biết thêm các kiến thức về IoT như Wifi Station, Wifi Access Point, HTTP, MQTT, Web Server,... và một số các vấn đề về bảo mật.

# TÀI LIỆU HỌC LẬP TRÌNH

- ESP32 Datasheet: [https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)
- ESP32 reference manual: [https://www.espressif.com/sites/default/files/documentation/esp32\\_technical\\_reference\\_manual\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf)
- ESP32 WROOM schematic: <https://docs.zohopublic.com/file/kohvpddd1cff937ec4de5a828ac624e60a74d>
- ESP32 Document: <https://docs.espressif.com/projects/esp-idf/en/v4.2.2/esp32/api-reference/index.html>

# LỘ TRÌNH HỌC

Buổi 1: Làm quen, hiểu cấu trúc của Project. Biết cách sử dụng ESP-IDF. Bắt đầu dự án đầu tiên với Blink Led và Hello World, làm quen với FreeRTOS.

Buổi 2: Lập trình PWM, RMT WS2812, Đọc dữ liệu cảm biến DHT11

Buổi 3: Lập trình Wifi Station, HTTP Server basic.

Buổi 4: Tìm hiểu sơ bộ HTML, CCS, JavaScript, các tự chủ giao diện 1 trang web Local. Đẩy dữ liệu DHT11. Nhận dữ liệu Button.

Buổi 5: Lập trình HTTP Client Write, Read Data lên ThingSpeak.

Buổi 6: Nhúng đồ thị ThingSpeak vào Web Server, điều khiển RGB, hoàn thiện dự án.

Buổi 7: Lập trình Smart Config cấu hình SSID, PWD. Lập trình WiFi Access Point. Chạy HTTP Server.

Buổi 8: Lập trình MQTT, MQTT + Secure (SSL). Sử lý chuỗi Json.

Buổi 9: Lập trình Sử dụng Flash, học cách phân chia các vùng nhớ trên Flash.

Buổi 10: Lập trình OTA, tạo Server Local lưu Firmware Update.

Buổi 11: Hoàn thiện Project, điều khiển đóng cắt Relay.

Buổi 12: Sử dụng tính năng Flash Encrytion để mã hóa dữ liệu Flash, bảo vệ Firmware. Giới thiệu các chức năng BLE.

# SO SÁNH MCU

## AT89C51

- 80C51 Core Architecture
- CPU: 0 Hz to 33 MHz
- RAM: 256 byte
- Flash: 8Kbytes

## STM32F103C8

- ARM® 32-bit Cortex®-M3
- CPU: 72 MHz
- RAM: 20 Kbytes
- Flash: 64 or 128 Kbytes

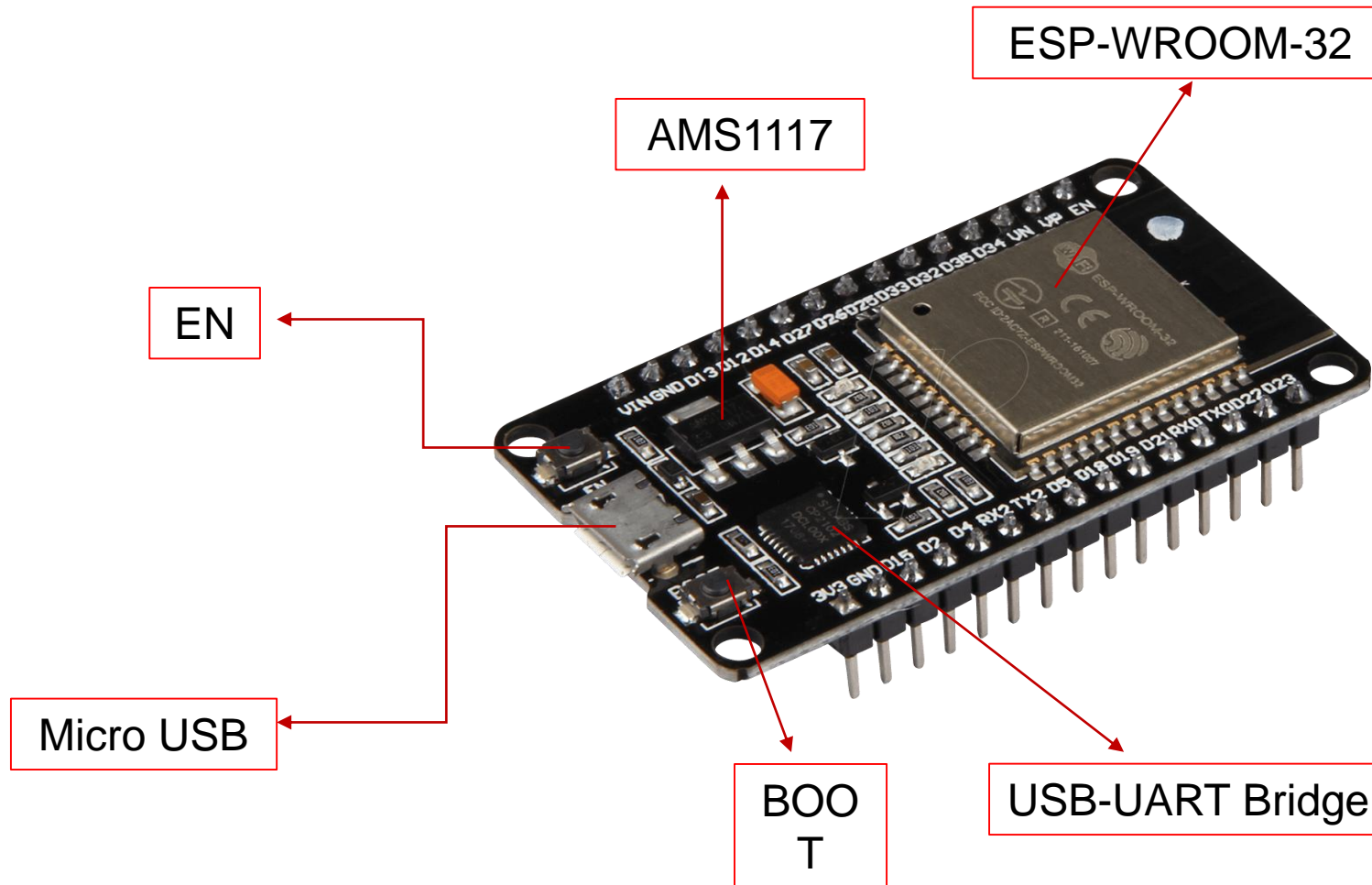
## ESP8266EX

- 32-bit RISC CPU : Tensilica Xtensa LX106
- CPU: up to 160 MHz
- RAM: < 50 Kbytes
- Flash ngoài: 512 Kbytes đến 4Mbytes

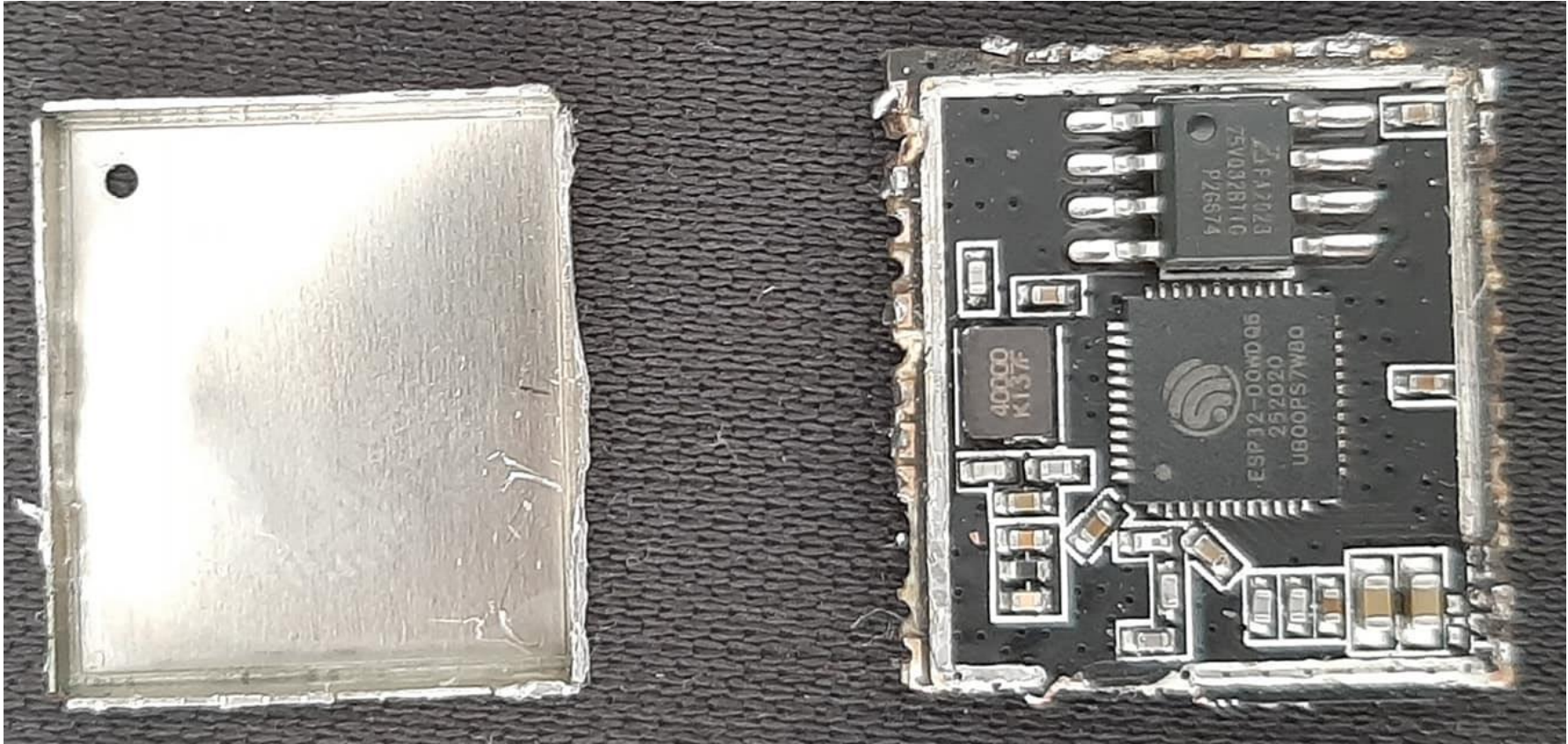
## ESP32

- 2 core low-power Xtensa® 32-bit LX6
- CPU: up to 240 MHz
- RAM: 520 KB
- Flash nội: 448 KB
- Flash ngoài: 4Mbytes

# CẤU TẠO CỦA DEVKIT



# BÊN TRONG MODULE ESP32



# XÂY DỰNG PROJECT ĐẦU TIÊN

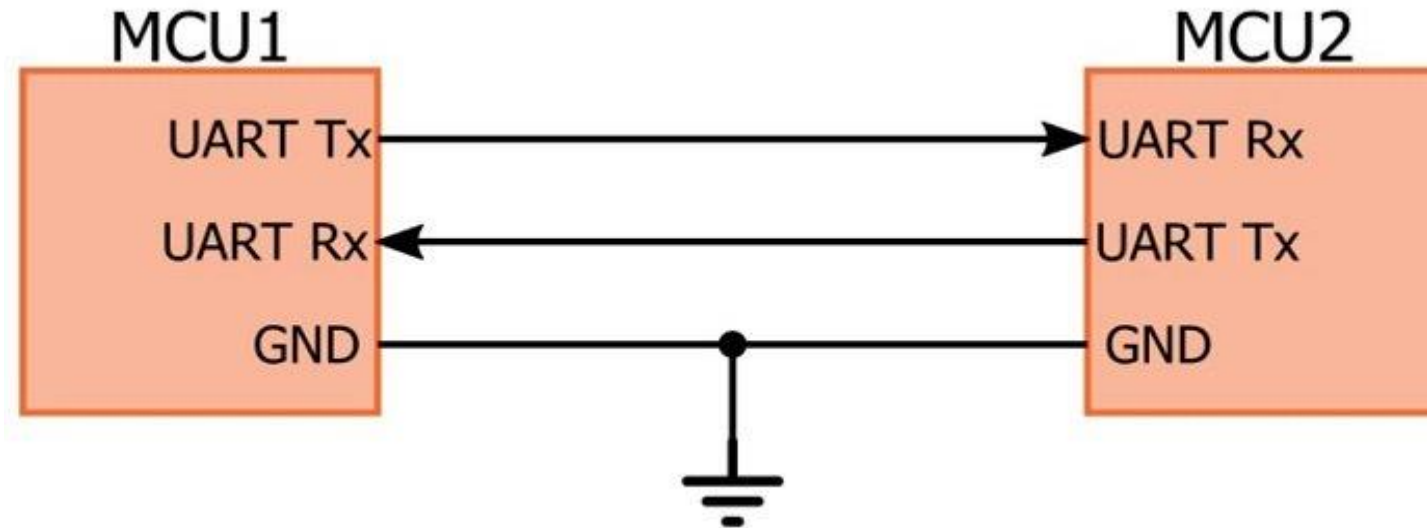
Level	Tên folder
1	D:
2	esp
3	esp-idf
	esp-project1
	esp-project2
	esp-projectN

1. Cấu trúc tạo 1 workspace làm việc
2. Vào esp-idf copy 1 example template đặt ngang hàng với thư mục esp-idf.
3. Chạy VSC để edit code.
4. Chạy ESP-IDF 4.2 PowerShell để build và nạp code xuống VDK.



# CHUẨN GIAO TIẾP UART

UART (Universal Asynchronous Receiver – Transmitter) là một chuẩn giao tiếp truyền nhận bất đồng bộ, bất đồng bộ có nghĩa là không có sự kiểm soát khi dữ liệu được gửi hoặc bất kì đảm bảo 2 bên có cùng tốc độ hay không.

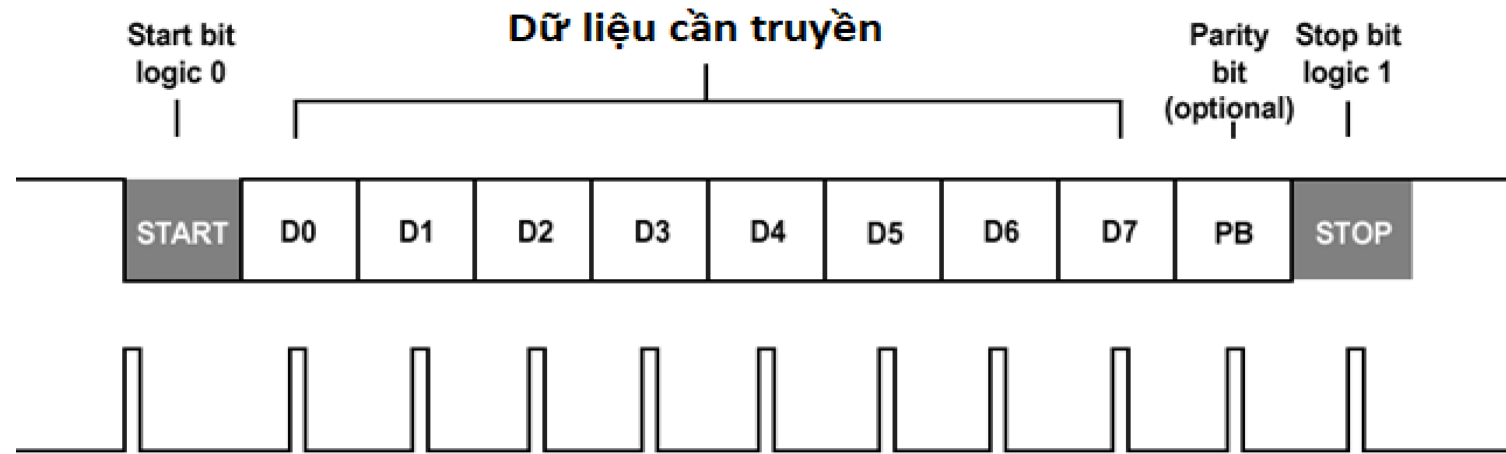


# CÁC KHÁI NIỆM TRONG UART



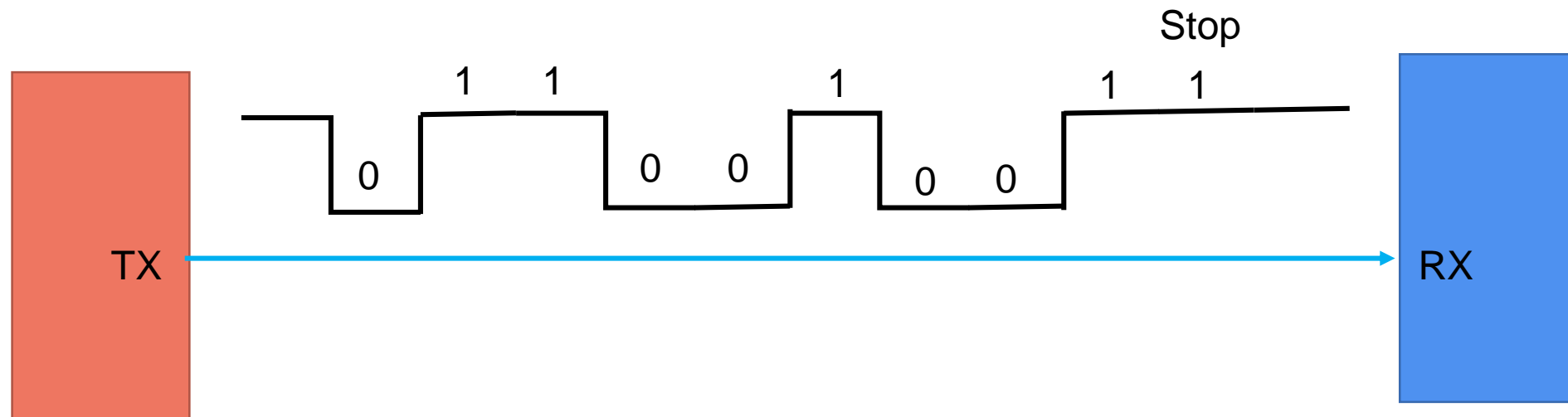
- **Baud rate** (tốc độ baud): số bit truyền nhận trong 1s, tốc độ hay sử dụng phải là bội của 8 thường là 2400, 4800, 9600, 19200, 38400, 56000, 115200... cả 2 thiết bị truyền nhận cho nhau phải cùng 1 tốc độ baud.
- **Frame** (khung truyền ): Khung truyền quy định về số bit trong mỗi lần truyền. Thông thường chúng ta sẽ chọn khung truyền 8 bit .
- **Start bit** : là bit đầu tiên được truyền trong 1 Frame. Báo hiệu cho thiết bị nhận có một gói dữ liệu sắp đc truyền đến. Bit bắt buộc.
- **Data** : dữ liệu cần truyền. Bit có trọng số nhỏ nhất LSB được truyền trước sau đó đến bit MSB.
- **Parity bit** : bit kiểm tra chẵn lẻ.
- **Stop bit** : là 1 hoặc các bit báo cho thiết bị rằng các bit đã được gửi xong. Thiết bị nhận sẽ tiến hành kiểm tra khung truyền nhằm đảm bảo tính đúng đắn của dữ liệu. Bit bắt buộc.

# KHUNG TRUYỀN UART



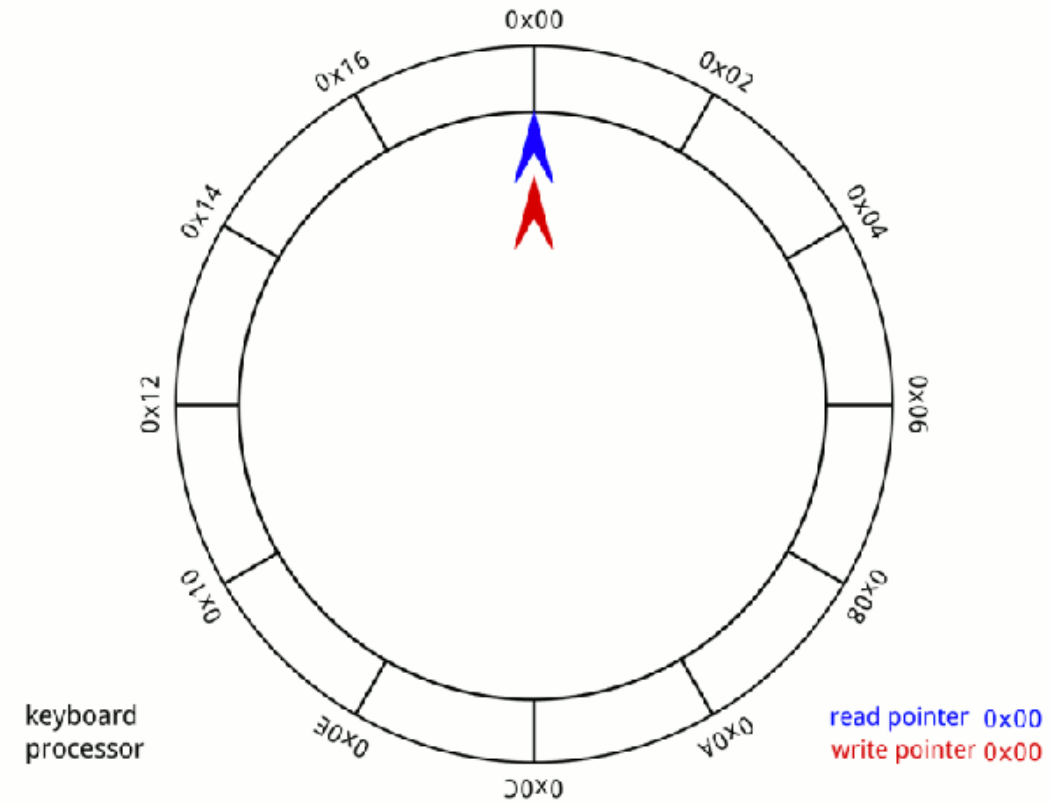
Xung clock được xác định theo thời gian

Giả sử ta cần truyền 1 byte, **0x93**;  
**0x93** = 1001 0011



# RING BUFFER

- Là 1 kiểu bộ đệm FIFO (First In First Out).  
Như tên gọi của nó (array-base), bộ đệm này được thực hiện dựa trên một mảng. Kèm theo đó là 2 con trỏ Write và Read.
  - Mỗi khi nhận lệnh ghi, con trỏ pWrite sẽ ghi data vào bộ đệm, sau đó sẽ tăng lên 1 đơn vị  $pWrite++$ .
  - Mỗi khi nhận lệnh đọc, con trỏ pRead sẽ tăng lên một. Sau đó đọc giá trị từ bộ đệm ra.
- Khi 1 con trỏ tới được cuối mảng, nó sẽ cuộn lại vị trí đầu tiên. Đó là lý do vì sao gọi đây là bộ đệm vòng.
- Nếu  $(pRead+1) == pWrite$ , chứng tỏ bộ đệm đang trống, không có gì để đọc.
  - Nếu  $pWrite == pRead$ , chứng tỏ bộ đệm đang đầy, không thể ghi thêm.

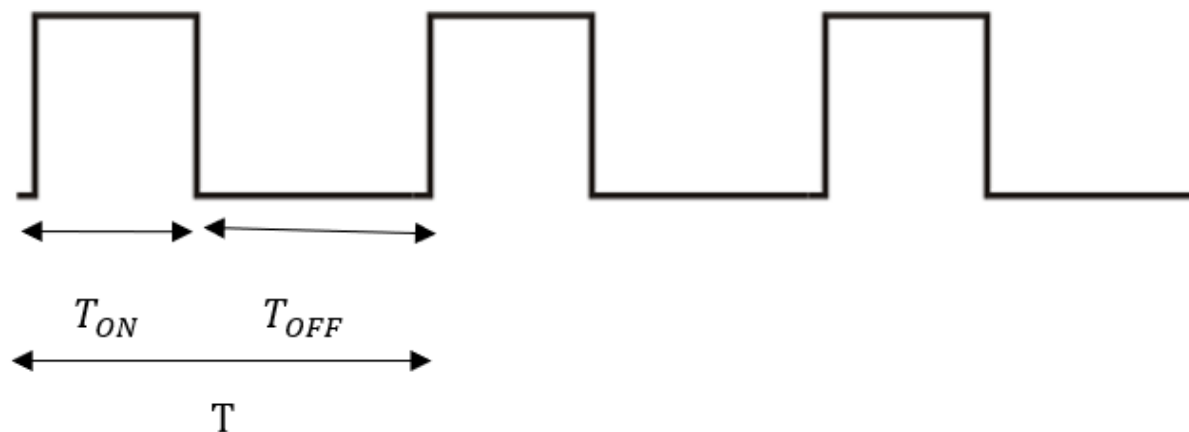


# PWM

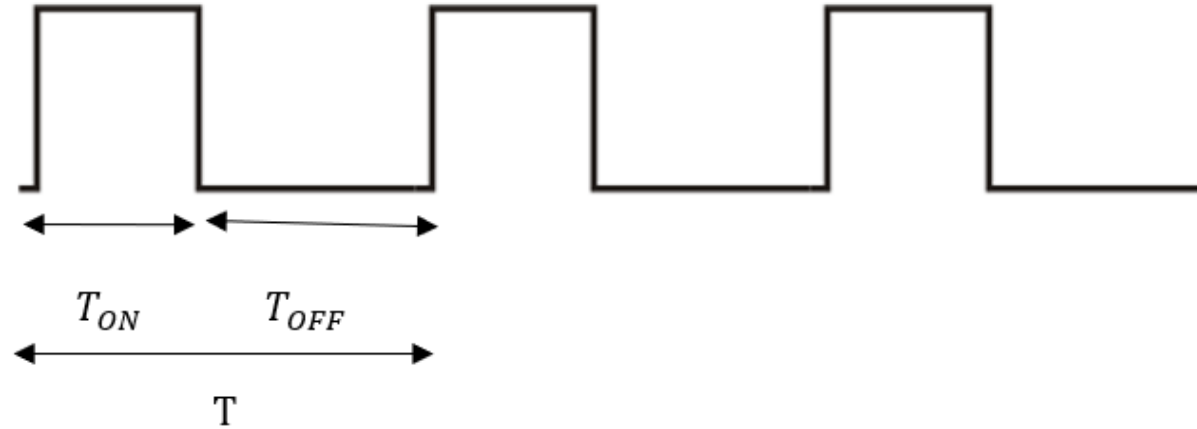
**PWM** là viết tắt của Pulse Width Modulation (Điều chế độ rộng xung) và nó là một kỹ thuật phổ biến được sử dụng để thay đổi độ rộng xung.

## Mục đích của PWM:

Khi tín hiệu 1 chân được bật tắt liên tục ở tần số cao kết hợp với việc thay đổi độ rộng của xung ( $T_{ON}$ ) thì độ sáng bóng đèn sẽ được thay đổi, ngoài ra sóng PWM còn có chức năng điều khiển tốc độ động cơ DC.



# CÁC THÔNG SỐ PWM



$T_{ON}$  (s): Thời gian xung ở mức cao (3.3V).

$T_{OFF}$  (s): Thời gian xung ở mức thấp (0V).

$T = T_{ON} + T_{OFF}$  (s) : Chu kì của xung.

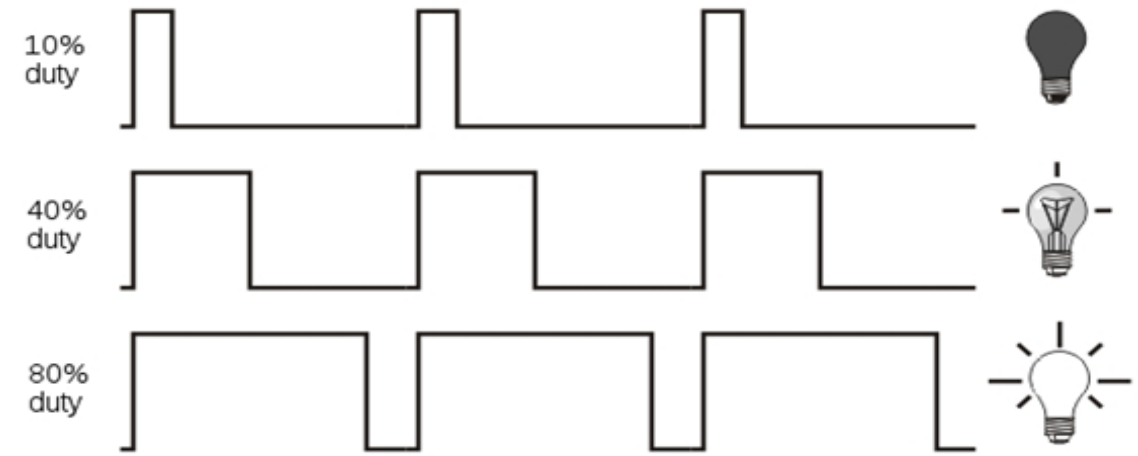
$f = \frac{1}{T}$  (Hz) : Tần số của xung.

Duty cycle  $= \frac{T_{ON}}{T_{ON} + T_{OFF}} = \frac{T_{ON}}{T}$  (%) : phần trăm thời gian xung ở mức cao trong 1 chu kì tín hiệu.

$V = 3.3 * \text{Duty cycle}$  : điện áp ra trung bình.

# NGUYÊN LÝ THAY ĐỔI ĐỘ SÁNG

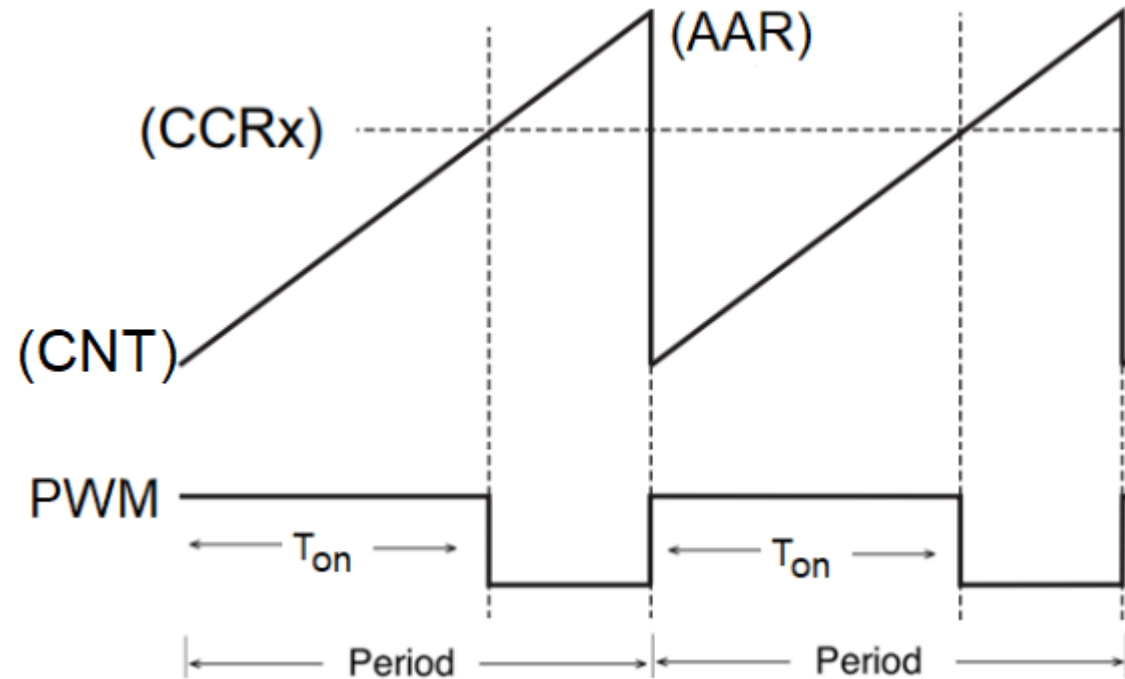
Duty cycle	Điện áp ra trung bình	Tình trạng đèn
10% duty	$V = 3.3 \cdot 10 / 100 = 0.33$ V	Sáng mờ
40% duty	$V = 3.3 \cdot 40 / 100 = 1.32$ V	Sáng trung bình
80% duty	$V = 3.3 \cdot 80 / 100 = 2.64$ V	Sáng rõ



# NGUYÊN LÝ TẠO XUNG PWM

## Nguyên lý tạo PWM

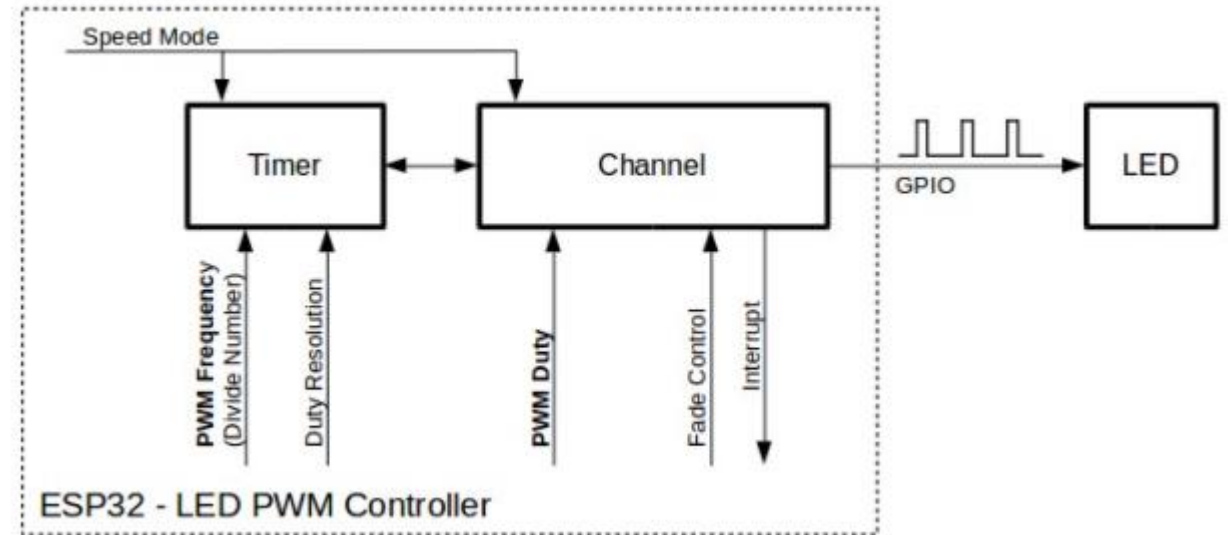
- Khi thanh ghi CNT bắt đầu đếm thì ngõ ra PWM sẽ ở mức 1.
- Khi CNT đến bằng giá trị trong thanh ghi CCRx thì ngõ ra PWM sẽ bị reset về 0.
- Khi CNT tăng đến bằng giá trị ARR thì CNT sẽ bị reset về 0, và ngõ ra PWM sẽ đặt lên 1.
- Tiếp tục chu trình như vậy sẽ tạo ra dạng sóng PWM.





# MODULE LEDC

- Được thiết kế để điều khiển LED và tạo ra các sóng PWM cho mục đích khác nhau.
- Có 16 kênh có thể tạo ra tín hiệu PWM trong đó 8 kênh High Speed và 8 kênh Low Speed.
- Tần số tối đa có thể tạo ra của PWM là 40MHZ với độ phân giải 1bit.
- Độ phân giải của PWM có thể cấu hình từ 1 bit tới 16 bit. Tần số càng cao thì độ phân giải đạt được càng nhỏ đi.



# DHT11

Link Datasheet: [tại đây](#)

Item	Measurement Range	Humidity Accuracy	Temperature Accuracy	Resolution	Package
DHT11	20-90%RH 0-50 °C	±5%RH	±2°C	1	4 Pin Single Row

- One communication process is about 4ms.
- Data consists of decimal and integral parts. A complete data transmission is 40bit, and the sensor sends higher data bit first.
- **Data format:** 8bit integral RH data + 8bit decimal RH data + 8bit integral T data + 8bit decimal T data + 8bit check sum.

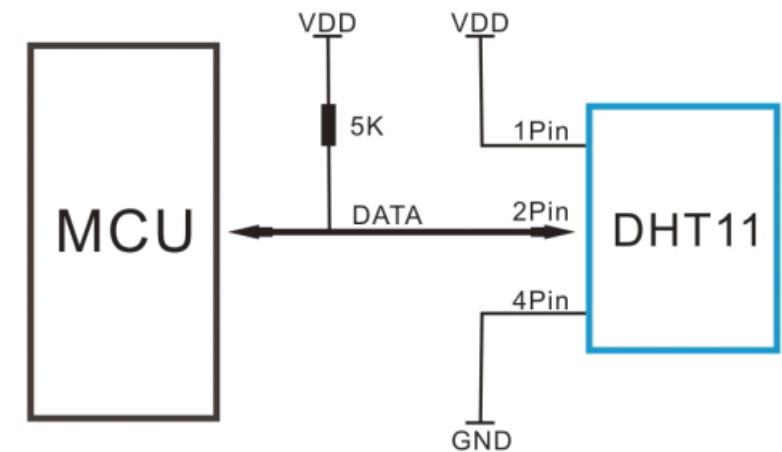
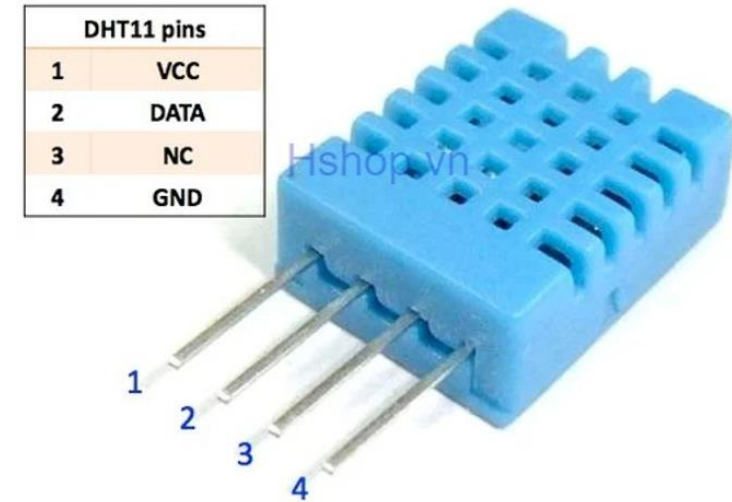


Figure 1 Typical Application

# MCU START READ DHT11

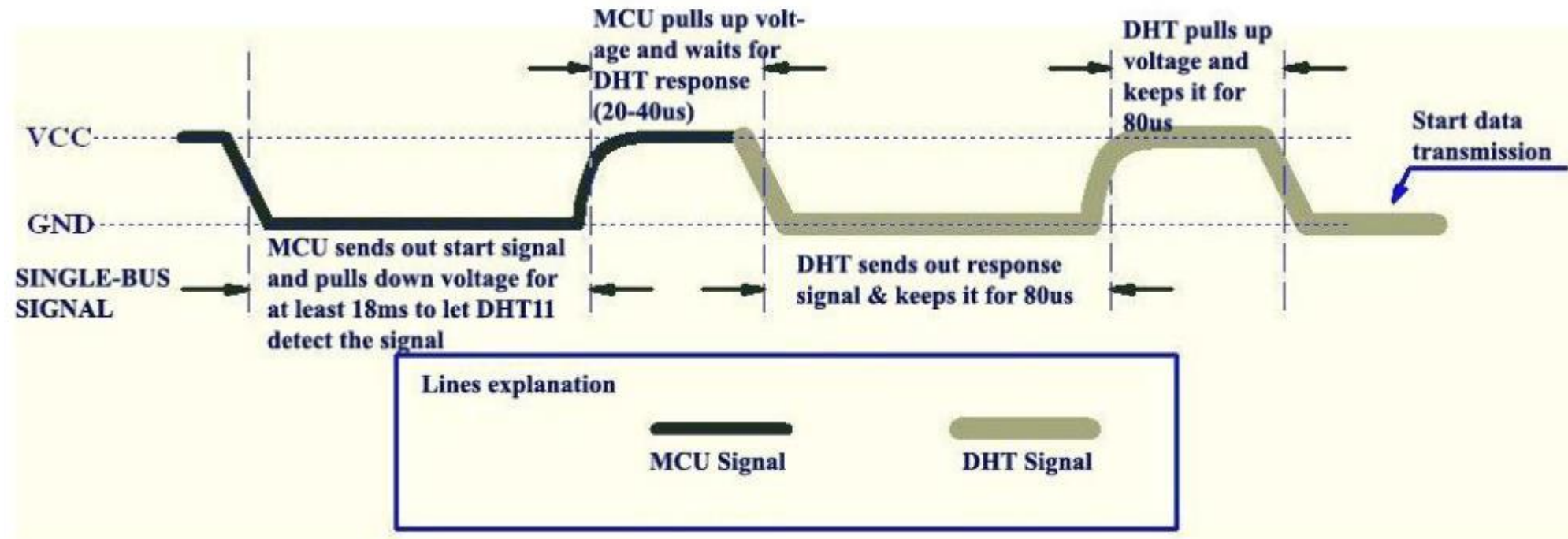
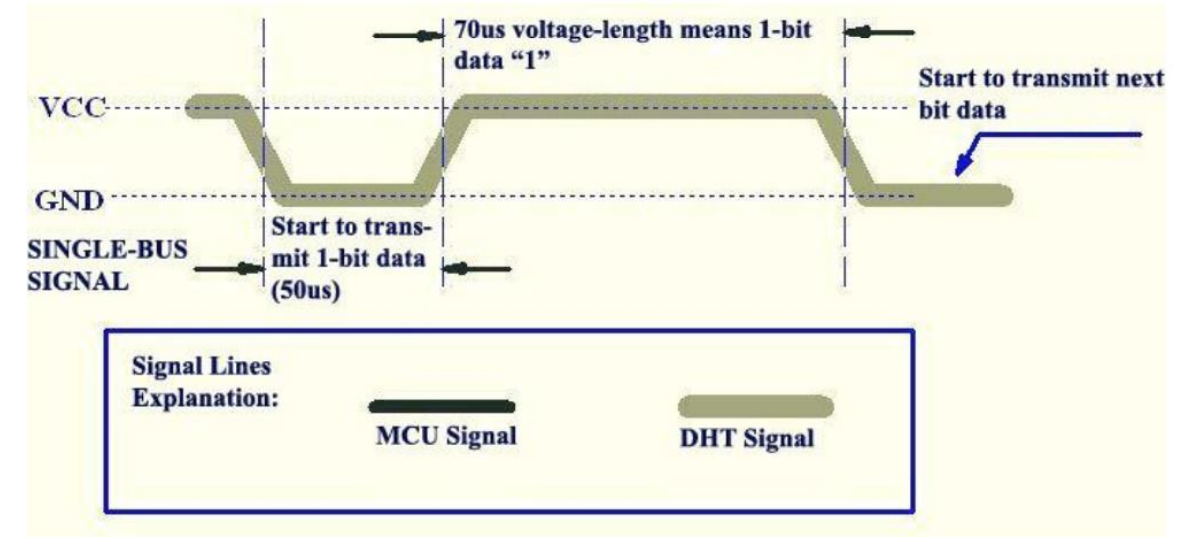
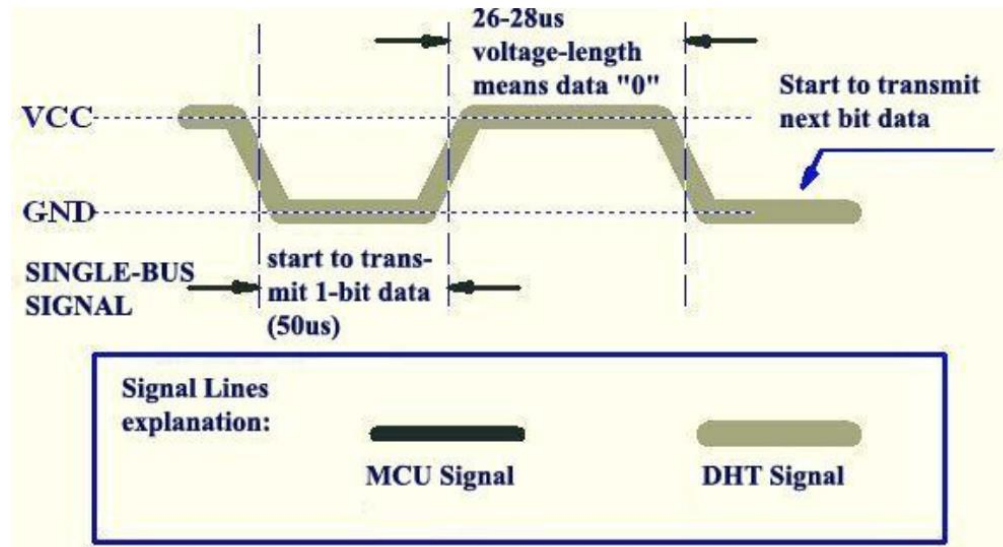


Figure 3 MCU Sends out Start Signal & DHT Responses

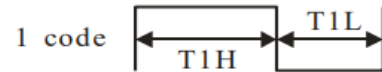
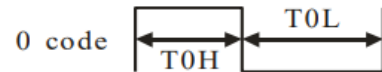
# QUY ƯỚC DỮ LIỆU DHT11



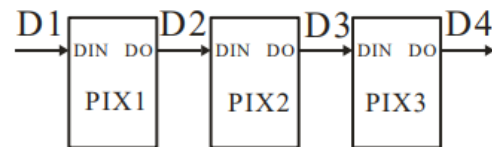
Link Datasheet: [tại đây](#)

**Data transfer time**(  $T_H + T_L = 1.25\mu s \pm 600ns$  )

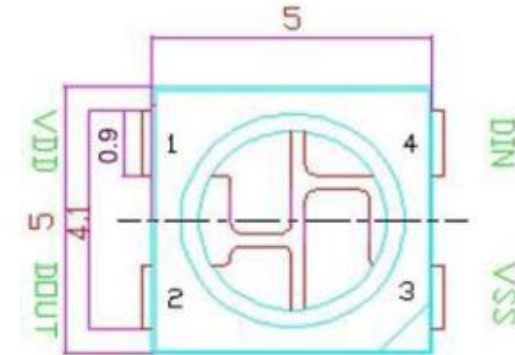
T0H	0 code ,high voltage time	0.4us	$\pm 150ns$
T1H	1 code ,high voltage time	0.8us	$\pm 150ns$
T0L	0 code , low voltage time	0.85us	$\pm 150ns$
T1L	1 code ,low voltage time	0.45us	$\pm 150ns$
RES	low voltage time	Above 50 $\mu s$	



**Cascade method:**



**PIN configuration**

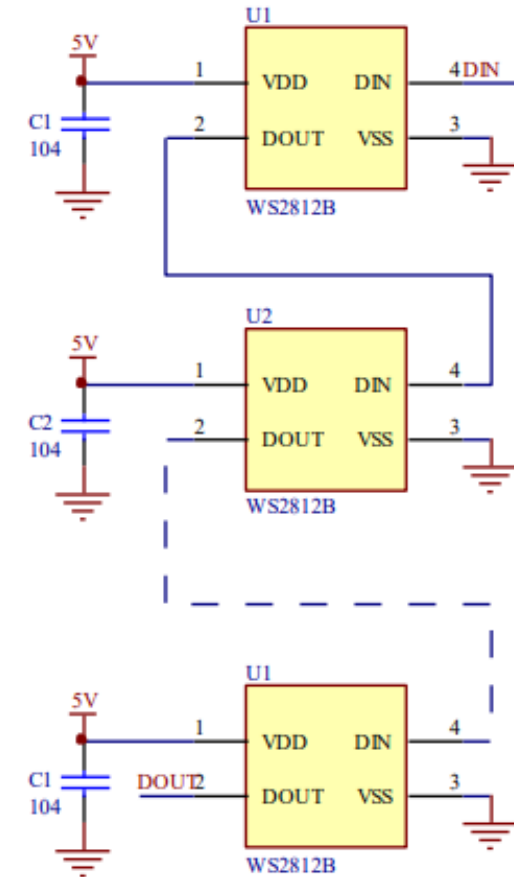
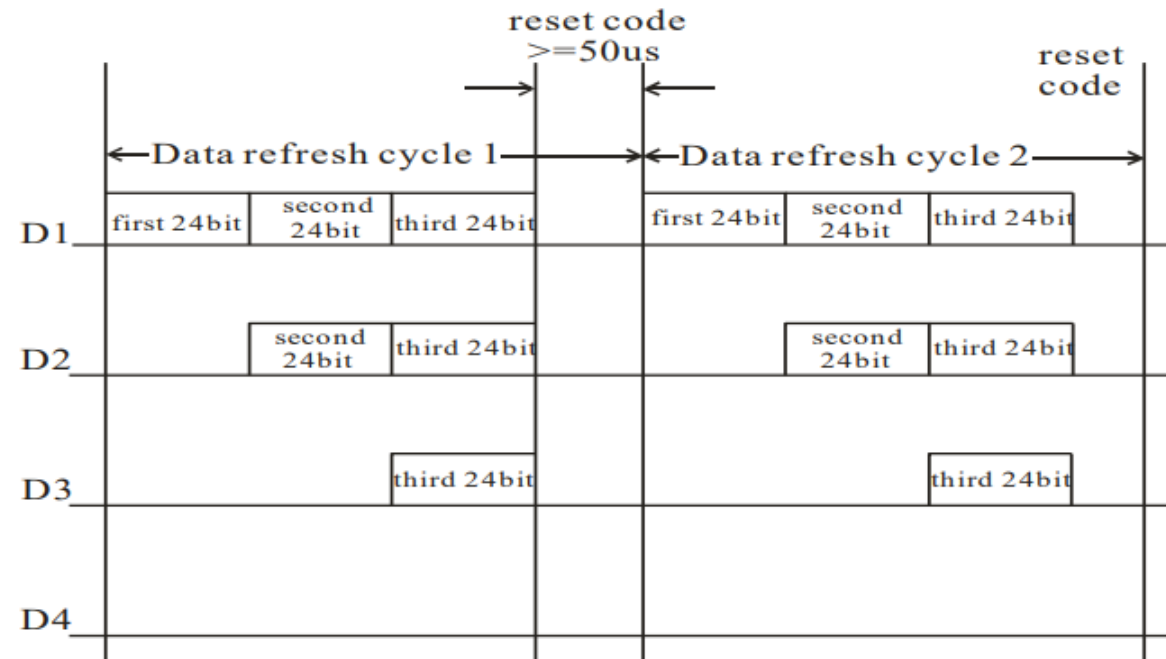


**PIN function**

NO.	Symbol	Function description
1	VDD	Power supply LED
2	DOUT	Control data signal output
3	VSS	Ground
4	DIN	Control data signal input

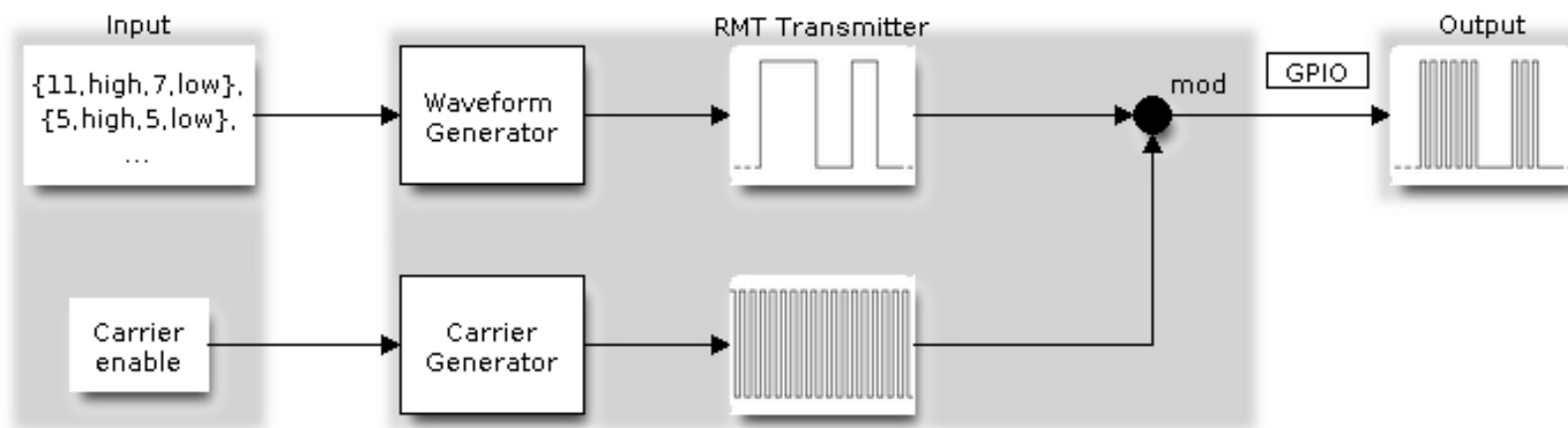
# WS2812

## Data transmission method:



# RMT

Trình điều khiển mô-đun RMT (Điều khiển từ xa) có thể được sử dụng để gửi và nhận tín hiệu điều khiển từ xa hồng ngoại. Do tính linh hoạt của mô-đun RMT, trình điều khiển cũng có thể được sử dụng để tạo hoặc nhận nhiều loại tín hiệu khác.



*RMT Transmitter Overview*

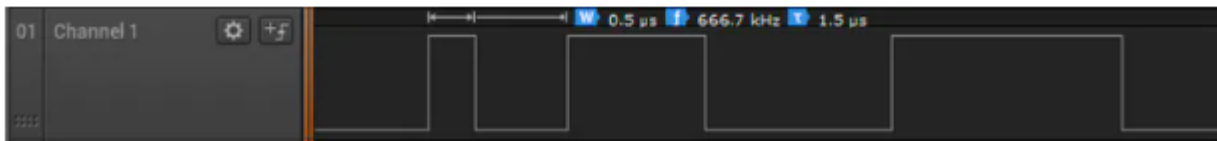
# RMT

## Example

```
var rmt_transmitter = Transmitter.Register(GPIO: 15);  
// transmitter configuration  
rmt_transmitter.CarrierEnabled = false;  
rmt_transmitter.isSource80MHz = true;  
rmt_transmitter.ClockDivider = 4; // base period 80 MHz / 4 => 20 MHz -> 0.05 us  
rmt_transmitter.IsTransmitIdleEnabled = true;  
rmt_transmitter.TransmitIdleLevel = false;  
  
// create new pulse command for to transmit  
// IDLE 0.5us 1us 1.5 us 2 us 2.5 us IDLE..  
// ---|----|-----|-----|-----|-----|  
// ---|----|-----|-----|-----|-----|  
var commandlist = new PulseCommandList();  
commandlist  
    .AddLevel(true, 10)  
    .AddLevel(false, 20)  
    .AddLevel(true, 30)  
    .AddLevel(false, 40)  
    .AddLevel(true, 50);  
rmt_transmitter.Send(commandlist);  
// destroy transmitter instance and free associated RMT channel  
rmt_transmitter.Dispose();
```

Khi chọn **CarrierEnable = false**  
thì sẽ không tạo ra Sóng mang.

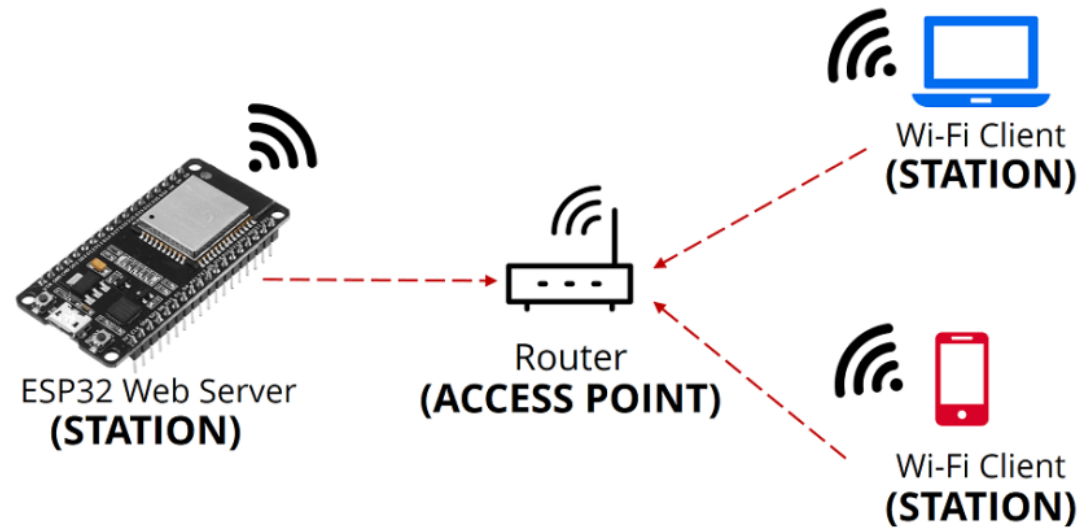
Let' see output on GPIO15:



GPIO15 output



# Wifi Station



Khi muốn kết nối vào mạng WiFi cục bộ thì ESP32 cần phải hoạt động ở chế độ Station (STA), đồng thời nó phải được cung cấp tên (SSID) và mật khẩu mạng WiFi.

Mỗi Access Point đều yêu cầu một phương thức mã hóa để Station sử dụng nhằm tạo kết nối - ví dụ các phương thức WEP, WPA2, tuy nhiên chúng ta có lẽ không cần quan tâm nhiều, vì ESP32 sẽ tự động thực hiện các thao tác lựa chọn phương thức mã hóa.

Khi kết nối thành công vào mạng WiFi thì ESP32 sẽ khởi động DHCP Client (mặc định) để xin cấp phát địa chỉ IP trước khi bắt đầu các kết nối IP. Do đó, nếu như vì lý do gì đó, mà Access Point của bạn không có DHCP Server để cấp phát IP thì bạn phải cấu hình IP tĩnh cho ESP32.

# Wifi Access Point



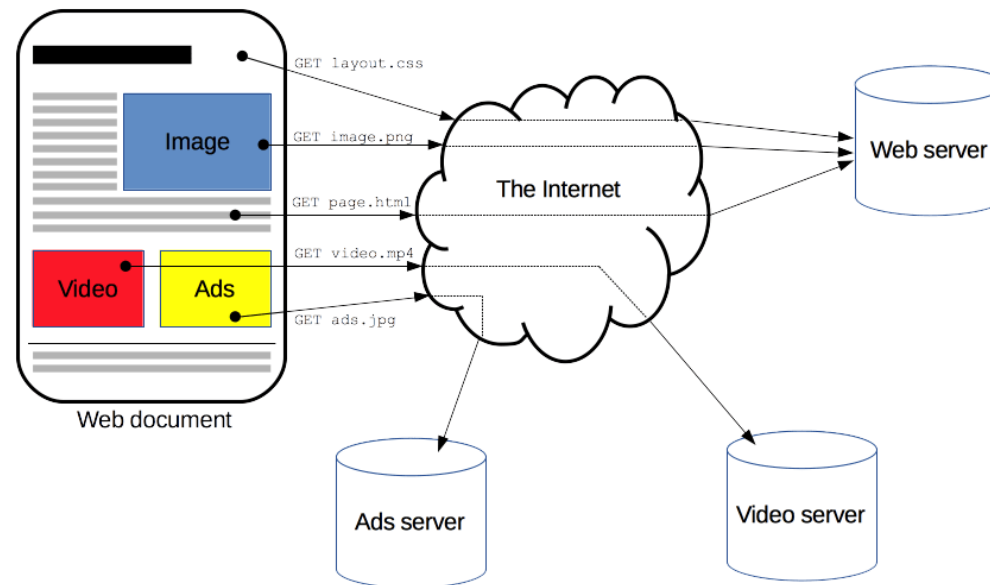
ESP32 có khả năng cho phép các thiết bị khác (Station - STA) truy cập vào và hoạt động như là 1 Access Point, có thể tự thiết lập 1 mạng WiFi nội bộ, với khả năng khởi động DHCP Server và cung cấp được IP cho các Client kết nối tới. Do giới hạn về RAM, nên số lượng tối đa các STA có thể kết nối đến một ESP32 là giới hạn.



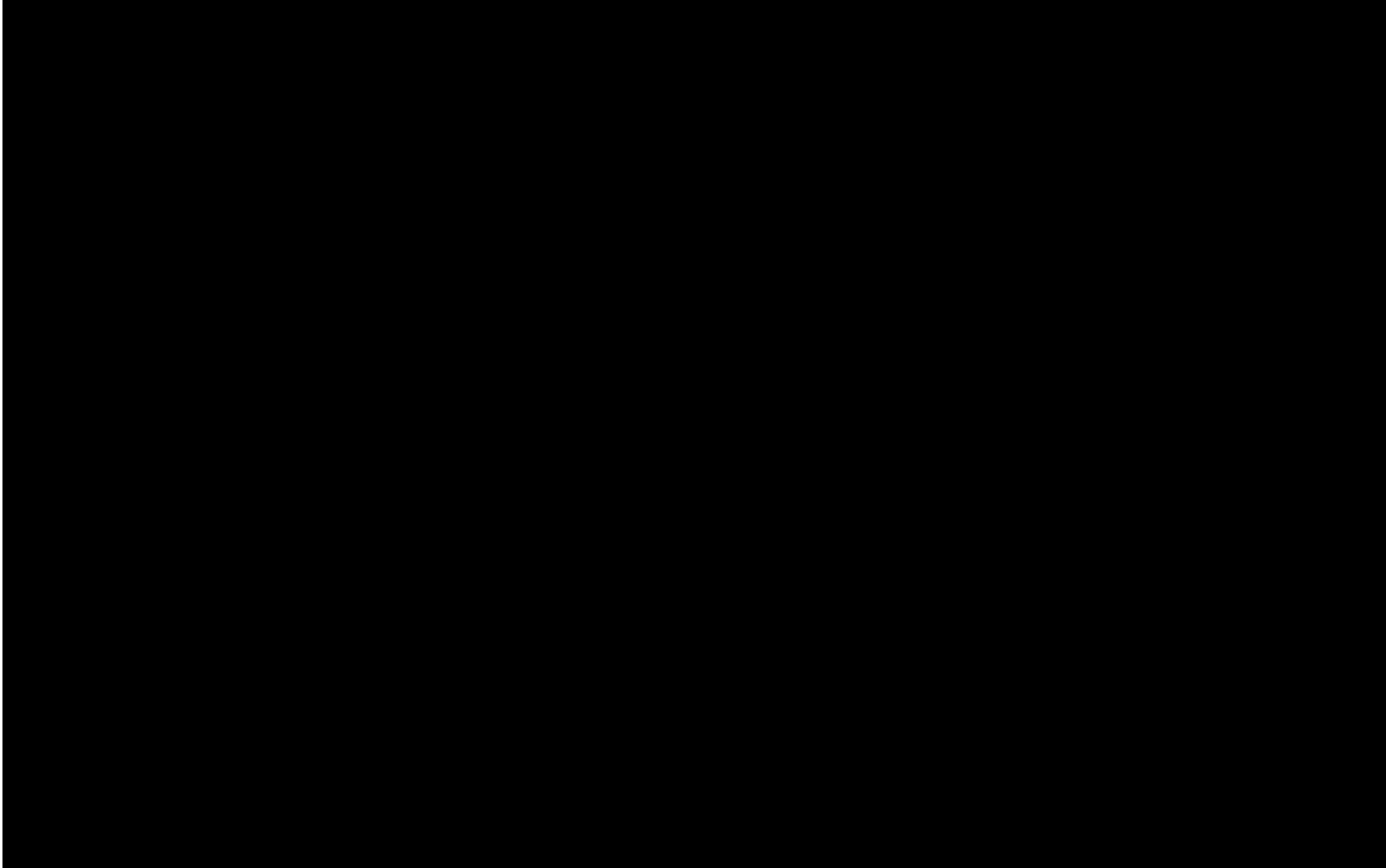
Lưu ý rằng, mạng WiFi khởi tạo bởi hàm softAP sẽ sử dụng địa chỉ IP mặc định là 192.168.4.1, và chạy 1 DHCP Server cung cấp dải IP cho client kết nối tới là 192.168.1.x.

# Giao thức HTTP

- HTTP - Hypertext Transfer Protocol (giao thức truyền dẫn siêu văn bản), là giao thức để truyền dữ liệu giữa các máy tính qua www (World Wide Web), với dữ liệu có thể là dạng text, file, ảnh, hoặc video.
- HTTP được thiết kế để trao đổi dữ liệu giữa Client và Server trên nền TCP/IP, nó vận hành theo cơ chế yêu cầu/trả lời, stateless - không lưu trữ trạng thái. Trình duyệt Web chính là Client, và một máy chủ chứa Web Site là Server. Client sẽ kết nối tới Server, gửi dữ liệu đến server bao gồm các thông tin header. Server nhận được thông tin và căn cứ trên đó gửi phản hồi lại cho Client. Đồng thời đóng kết nối.

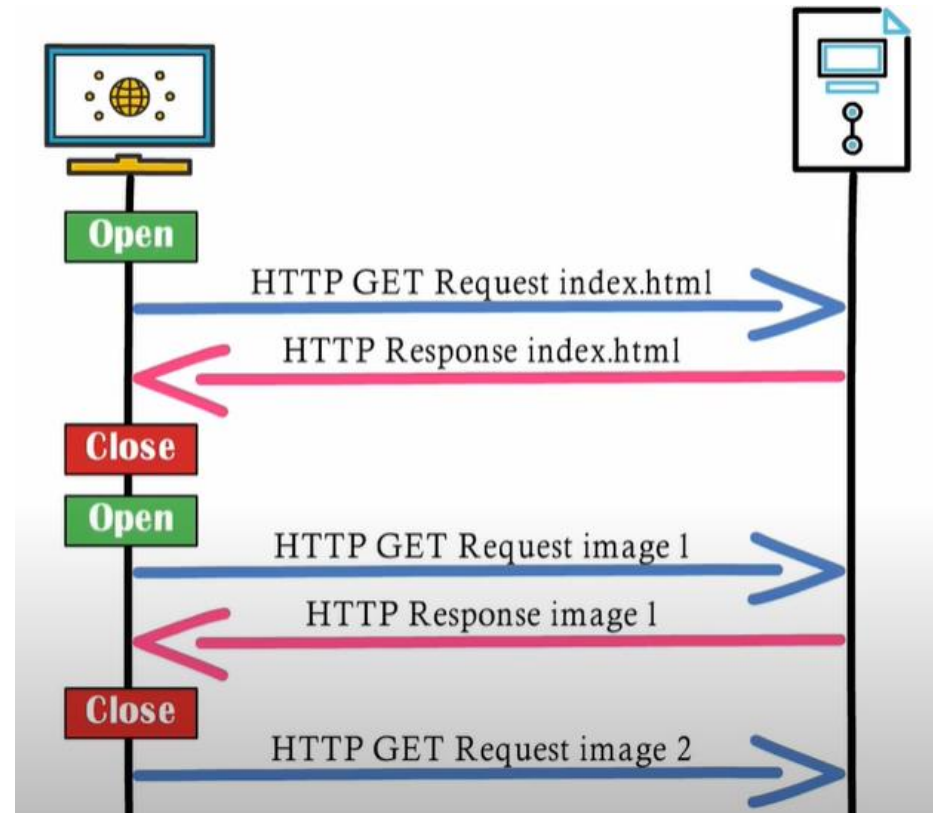


# Giao thức HTTP

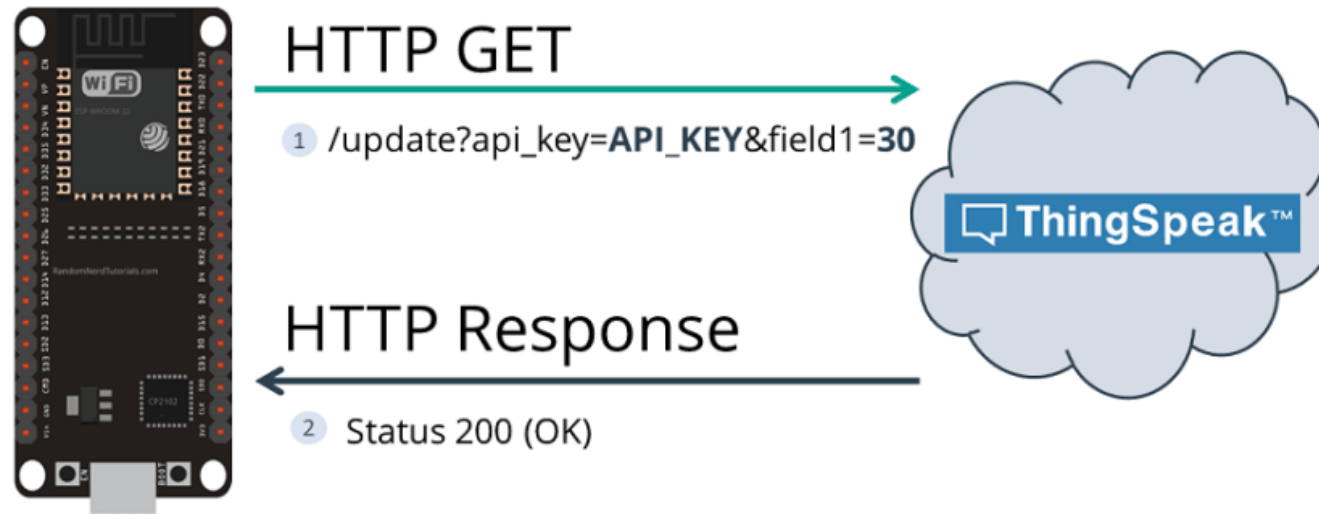


# HTTP CLIENT

Trong giao thức HTTP, việc thiết lập kết nối chỉ có thể xuất phát từ phía client ( lúc này có thể gọi là HTTP Client ). Khi client gửi yêu cầu, cùng với URL và payload ( dữ liệu muốn lấy ) tới server. Server ( HTTP Server ) lắng nghe mọi yêu cầu từ phía client và trả lời các yêu cầu ấy. Khi trả lời xong kết nối được chấm dứt.



# HTTP REQUEST



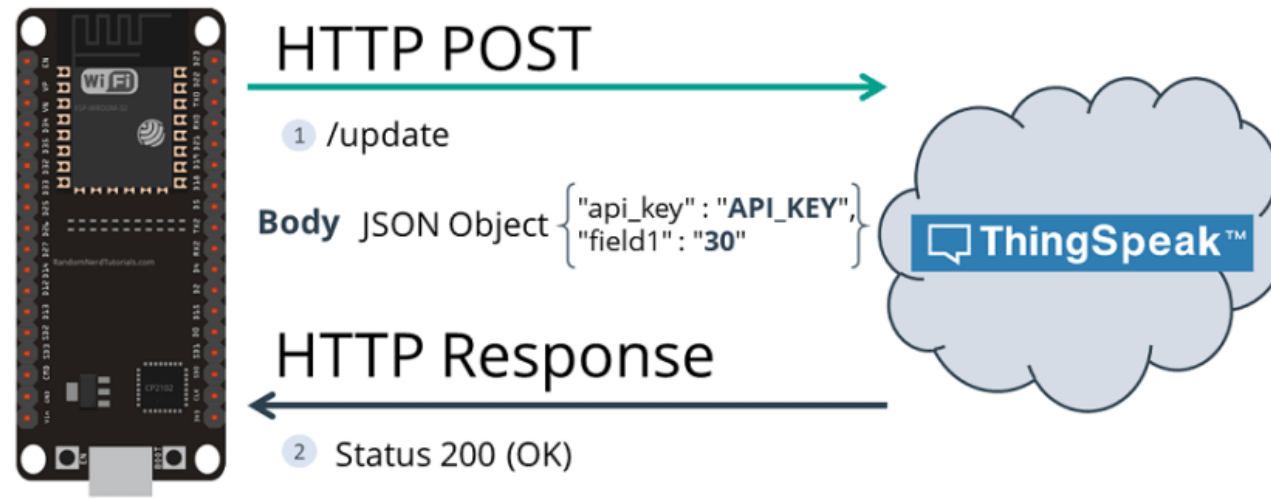
## GET

Là phương thức yêu cầu dữ liệu đơn giản và thường sử dụng nhất của HTTP. Phương thức GET yêu cầu server chỉ trả về dữ liệu bằng việc cung cấp các thông tin truy vấn trên URL, thông thường Server căn cứ vào thông tin truy vấn đó trả về dữ liệu mà không thay đổi nó. path và query trong URL chứa thông tin truy vấn.

### Example:

```
GET /update.json?api_key=XXXXXXXXXXXXXXXXX&field1=10&field2=10 HTTP/1.1
Host: api.thingspeak.com
User-Agent: esp-idf/1.0 esp32
```

# HTTP REQUEST



## POST

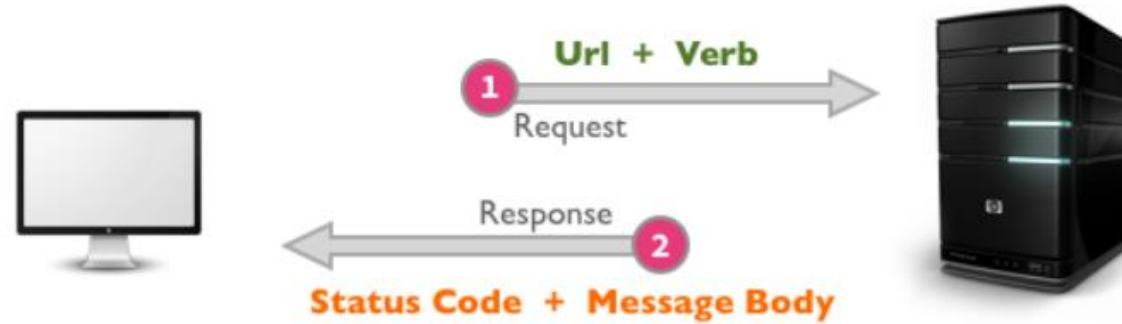
Tương tự như GET, nhưng POST có thể gửi dữ liệu về Server.

### Example:

```
POST /update.json HTTP/1.1
Host: api.thingspeak.com
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length:44
```

```
api_key=XXXXXXXXXXXXXXXXX&field1=10&field2=10
```

# HTTP RESPONSE



## STATUS CODES

- **1xx : Informational**

Đã nhận / đã xử lý yêu cầu

- **2xx : Success**

Truy cập thành công

- **3xx : Redirect**

Yêu cầu thực hiện thêm hành động

- **4xx : Client Error**

Request một nội dung không tồn tại

- **5xx : Server Error**

Server không thực hiện yêu cầu.

**200** - OK

**201** - OK created

**301** - Moved to new URL

**304** - Not modified (Cached version)

**400** - Bad request

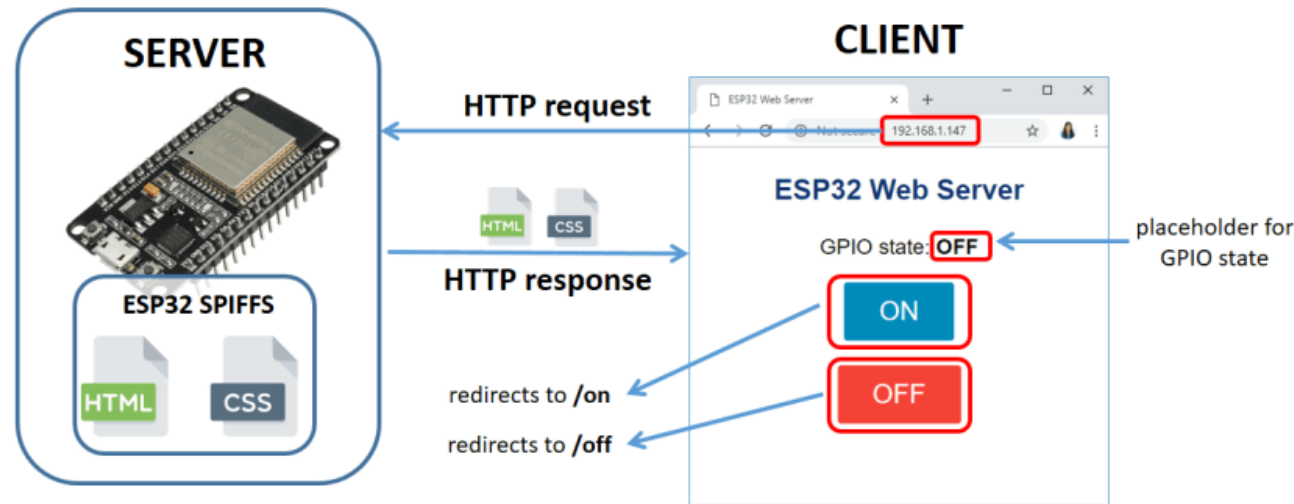
**401** - Unauthorized

**404** - Not found

**500** - Internal server error



# WEB SERVER



## Web Server là gì?

Web Server là một máy chủ Web mà khi có bất kỳ một Web Client nào (chẳng hạn Web Browser) truy cập vào, thì nó sẽ căn cứ trên các thông tin yêu cầu truy cập để xử lý, và phản hồi lại nội dung.

Đa phần các nội dung Web Server phục vụ là HTML, Javascript, CSS, JSON và bao gồm cả các dữ liệu Binary. Mặc định các Web Server phục vụ trên Port 80, và 443 cho dịch vụ Web có bảo mật HTTPS

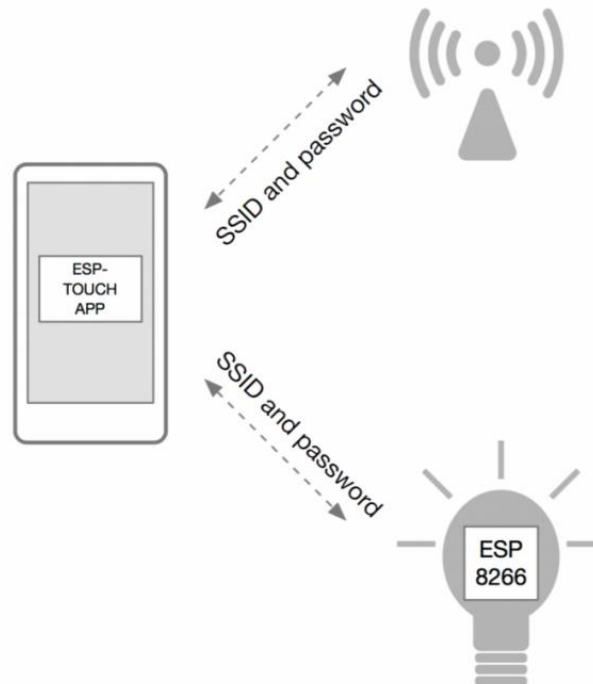
## HTML - Javascript - CSS

HTML, Javascript và CSS là ba ngôn ngữ để xây dựng và phát triển Web. Những hiểu biết cơ bản về chúng sẽ tạo điều kiện thuận lợi cho các quá trình tiếp theo sau được dễ dàng hơn.

# SMART CONFIG

## SmartConfig là gì ?

SmartConfig là một giao thức được tạo ra nhằm cấu hình cho các thiết bị kết nối với mạng WiFi một cách dễ dàng nhất bằng smart phone. Nói một cách đơn giản, để kết nối WiFi cho thiết bị ESP8266, ESP32 ta chỉ cần cung cấp thông tin mạng wifi (bao gồm SSID và password) cho ESP thông qua 1 ứng dụng trên smart phone.

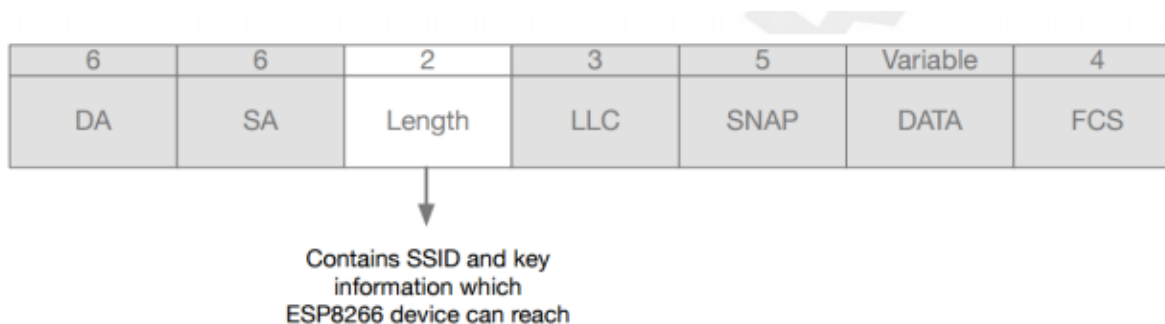


# SMART CONFIG

## SmartConfig hoạt động như thế nào ?

Cách thức để giao thức ESPTOUCH thực hiện việc gửi thông tin SSID và mật khẩu cho thiết bị như sau:

- ESP32 sẽ vào chế độ lắng nghe, lần lượt từng kênh.
- Điện thoại phải kết nối vào mạng WiFi được mã hóa.
- Ứng dụng trên điện thoại sẽ tiến hành gửi các gói tin với nội dung bất kỳ, nhưng có độ dài n theo từng ký tự của SSID và mật khẩu. Ví dụ, ssid của mạng là mynetwork thì sẽ có ký tự m, với ký tự ascii = 109, Ứng dụng sẽ gửi gói tin có độ dài 109 với nội dung bất kỳ, và lặp lại cho đến hết ký tự k, cũng như mật khẩu, và các ký tự khác như CRC.
- Có thể giao thức ESPTOUCH sẽ mã hóa cả các thông số gửi đi, nhưng vẫn giữ nguyên tắc như trên.
- ESP32 sẽ phát hiện ra các gói tin với độ dài thay đổi này và ghép nối lại thành SSID và password để kết nối vào mạng.
- Khi ESP32 kết nối thành công đến mạng, ESP32 sẽ kết nối đến IP của Điện thoại, được cung cấp thông qua ESPTOUCH, và gửi thông tin kết nối thành công đến ứng dụng trên điện thoại.

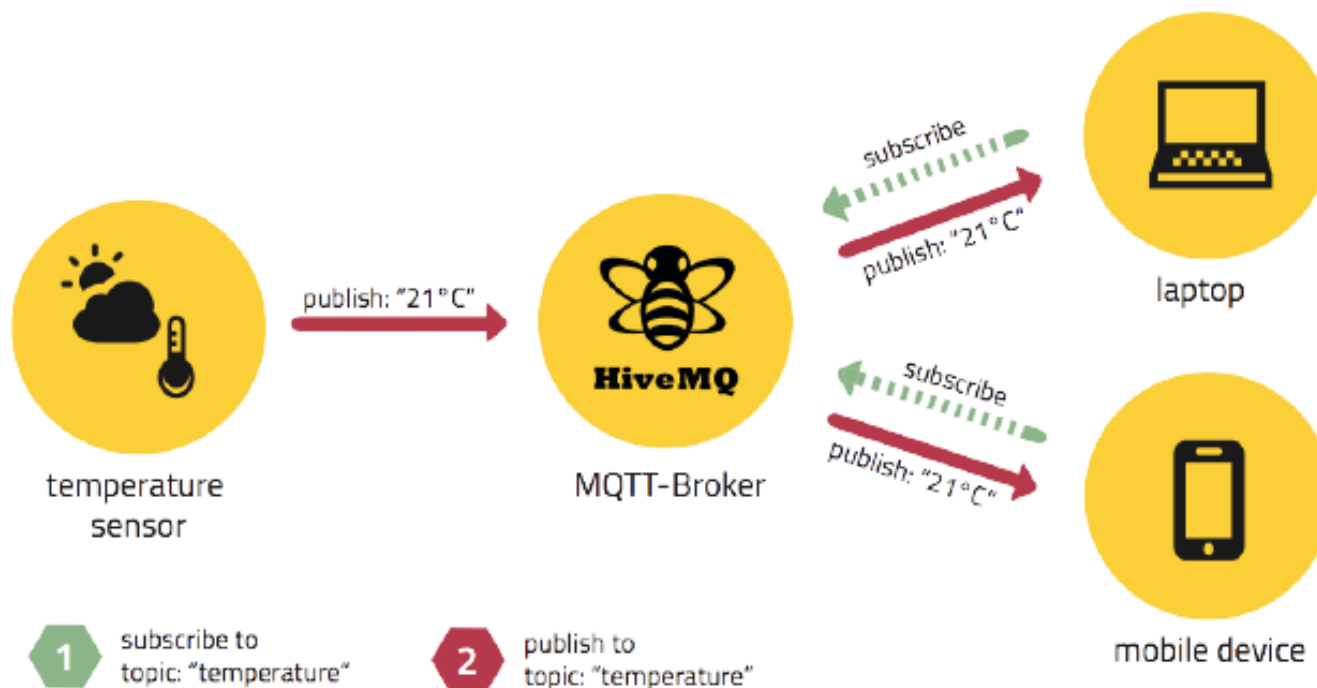


# MQTT

## MQTT là gì ?

MQTT (Message Queuing Telemetry Transport) là một giao thức gửi dạng publish/subscribe sử dụng cho các thiết bị Internet of Things với băng thông thấp, độ tin cậy cao và khả năng được sử dụng trong mạng lưới không ổn định.

Bởi vì giao thức này sử dụng băng thông thấp trong môi trường có độ trễ cao nên nó là một giao thức lý tưởng cho các ứng dụng M2M.



## Một số khái niệm quan trọng

### **Publish, subscribe**

Trong một hệ thống sử dụng giao thức MQTT, nhiều node trạm (gọi là mqtt client - gọi tắt là client) kết nối tới một MQTT Server (gọi là Broker). Mỗi client sẽ đăng ký một vài kênh (topic), ví dụ như "/client1/channel1", "/client1/channel2". Quá trình đăng ký này gọi là "subscribe", giống như chúng ta đăng ký nhận tin trên một kênh Youtube vậy. Mỗi Client sẽ nhận được dữ liệu khi bất kỳ trạm nào khác gửi dữ liệu vào kênh đã đăng ký. Khi một Client gửi dữ liệu tới kênh đó, gọi là "publish".

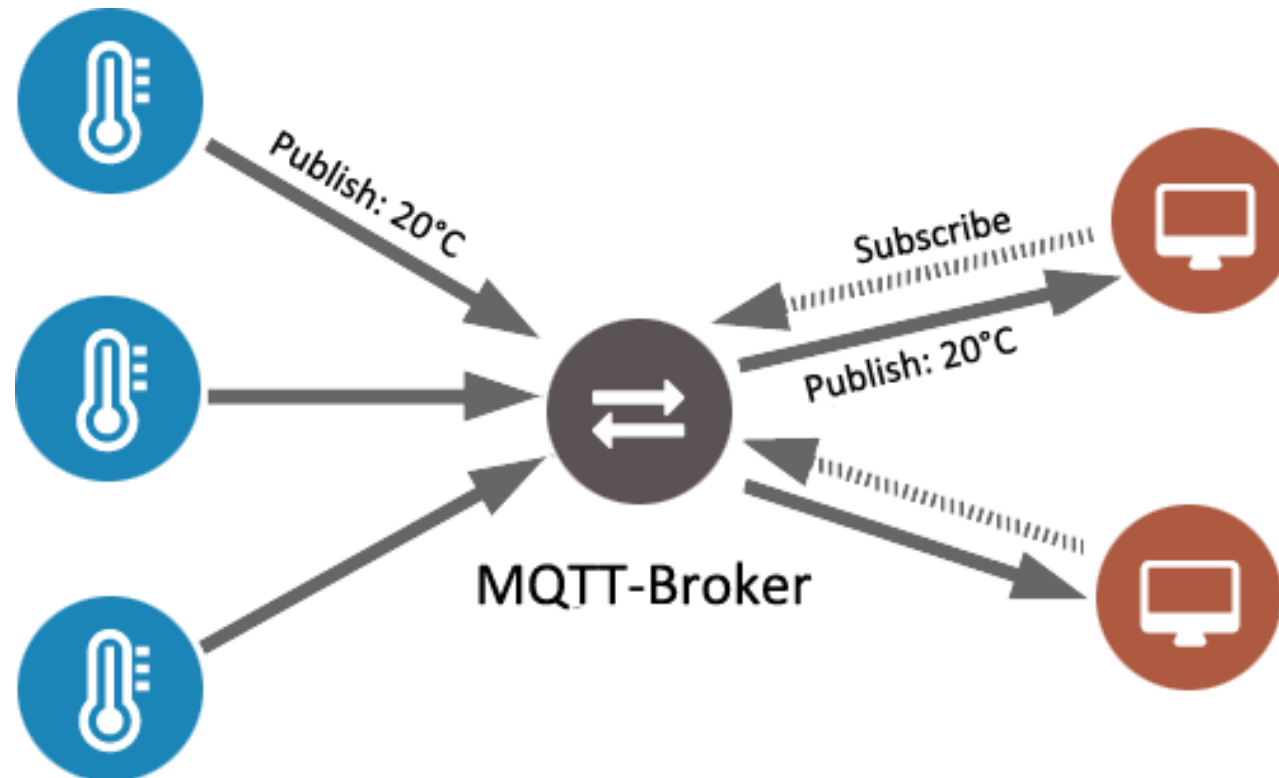
### **QoS**

Ở đây có 3 tùy chọn QoS (Qualities of service) khi "publish" và "subscribe":

- QoS0 Broker/Client sẽ gửi dữ liệu đúng 1 lần, quá trình gửi được xác nhận bởi chỉ giao thức TCP/IP, giống kiểu đem con bỏ chợ.
- QoS1 Broker/Client sẽ gửi dữ liệu với ít nhất 1 lần xác nhận từ đầu kia, nghĩa là có thể có nhiều hơn 1 lần xác nhận đã nhận được dữ liệu.
- QoS2 Broker/Client đảm bảo khi gửi dữ liệu thì phía nhận chỉ nhận được đúng 1 lần, quá trình này phải trải qua 4 bước bắt tay.

# MQTT CLIENT

Như chúng ta đã tìm hiểu ở phần trước, 2 thành phần publisher và subscriber là đặc trưng tạo nên giao thức MQTT. Các MQTT Client không kết nối trực tiếp với nhau, mọi gói dữ liệu được gửi đi đều thông qua MQTT Broker. Để có thể triển khai các ứng dụng của MQTT Client, chúng ta cần MQTT Broker (sẽ được trình bày trong phần sau).



# MQTT CLIENT

Một số các MQTT Client điển hình:

- MQTT LEN
- MQTT FX
- ESP8266/32 MQTT CLIENT
- MOSQUITO MQTT CLIENT

MQTT FX Download: <http://www.jensd.de/apps/mqttx/1.7.1/>



# MQTT BROKER



MQTT Broker có chức năng điều phối các bản tin giữa các MQTT Client. Một số MQTT Broker miễn phí public có thể kể đến như:

Name	Broker Address	TCP Port	TLS Port	WebSocket Port	Message Retention	Persistent Session
+ Eclipse	mqtt.eclipse.org	1883	N/A	80, 443	YES	YES
+ Mosquitto	test.mosquitto.org	1883	8883, 8884	80	YES	YES
+ HiveMQ	broker.hivemq.com	1883	N/A	8000	YES	YES
+ Flespi	mqtt.flespi.io	1883	8883	80, 443	YES	YES
+ Dioty	mqtt.dioty.co	1883	8883	8080, 8880	YES	YES
+ Fluux	mqtt.fluux.io	1883	8883	N/A	N/A	N/A
+ EMQX	broker.emqx.io	1883	8883	8083	YES	YES

Ngoài ra bạn cũng có thể tự cài dịch vụ MQTT Broker chạy local:

<https://mosquitto.org/download/>



# JSON



JSON (JavaScript Object Notation) là 1 định dạng trao đổi dữ liệu để giúp việc đọc và viết dữ liệu trở nên dễ dàng hơn, máy tính cũng sẽ dễ phân tích và tạo ra JSON. Chúng là cơ sở dựa trên tập hợp của ngôn ngữ lập trình JavaScript. JSON là 1 định dạng kiểu text mà hoàn toàn độc lập với các ngôn ngữ hoàn chỉnh, thuộc họ hàng với các ngôn ngữ trong họ hàng của C, gồm có C, C++, C#, Java, JavaScript, Perl, Python, và nhiều ngôn ngữ khác. Những đặc tính đó đã tạo nên JSON 1 ngôn ngữ Internet Of Things (IoT) cho người mới bắt đầu 51/155 hoán vị dữ liệu lý tưởng.

JSON là tập hợp của các cặp tên và giá trị name-value.

## Ví dụ:

```
{
  "id": 12,
  "name": "tom",
  "sex": "man",
  "pass": true
}
```

```
{
  "name": "ws2812b",
  "on"   : true,
  "rgb":
  {
    "r": 100,
    "g": 50,
    "b": 70
  }
}
```

```
{
  "name": "Awesome 4K",
  "resolutions": [
    {
      "width": 1280,
      "height": 720
    },
    {
      "width": 1920,
      "height": 1080
    },
    {
      "width": 3840,
      "height": 2160
    }
  ]
}
```

# OTA Update Firmware



## Tại sao phải Update Firmware ?

Trong vòng đời của sản phẩm, việc nâng cấp tính năng mới diễn ra rất thường xuyên.

Việc này đòi hỏi thiết bị cần phải có 1 cơ chế để nạp lại code từ xa.

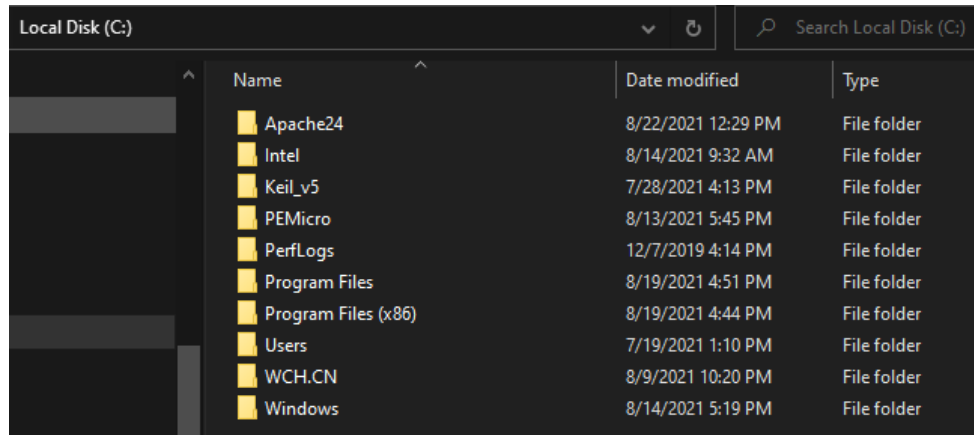
Vậy mục đích của OTA bao gồm:

- Fix các bug còn tồn đọng của bản Firmware hiện tại
- Update thêm các tính năng mới cho thiết bị
- Cải thiện hiệu suất của thiết bị

# Dựng HTTP Server chứa File

**B1:** Tải Apache: <https://www.apachelounge.com/download/>

**B2:** Copy thư mục giải nén vào ổ C

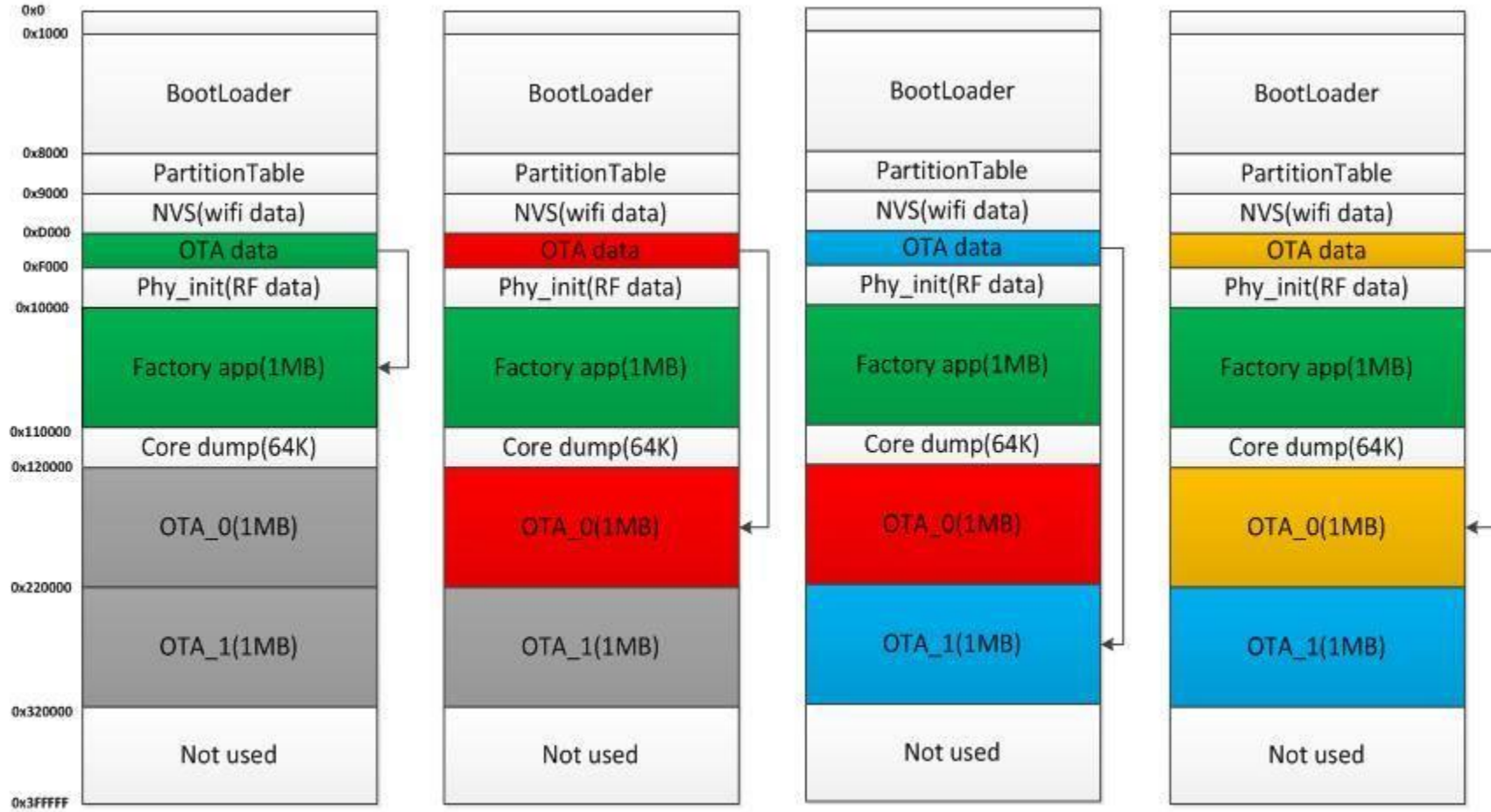


**B3:** Chạy File C:\Apache24\bin\httpd.exe để khởi chạy service. Vào trình duyệt gõ local host để test Service đã chạy thành công.

**B4:** Copy File FW vào thư mục C:\Apache24\htdocs

**B5:** Bạn đã có link download FW: [http://your\\_computer\\_ip/tên\\_File](http://your_computer_ip/tên_File)

# OTA FLOW



Flash出厂状态

OTA升级一次

OTA升级两次 [://blog.csdn.net/i16611](http://blog.csdn.net/i16611)

OTA升级三次

# OTA FLOW

1. ESP32 SPI Flash có (ít nhất) bốn phân vùng được liên kết với nâng cấp: OTA data, Factory App, OTA\_0, OTA\_1. Mặc định Firmware lần đầu tiên sẽ được nạp vào phân vùng Factory App.
2. Khi nâng cấp OTA được thực hiện lần đầu tiên, OTA Demo ghi phần FW mới vào phân vùng OTA\_0 và sau khi ghi xong, cập nhật dữ liệu phân vùng OTA data và khởi động lại.
3. Dữ liệu phân vùng OTA data được tính toán khi hệ thống được khởi động lại và nó được quyết định nhảy tới phân vùng OTA\_0 để thực thi.
4. Tương tự, nếu ESP32 đã thực thi FW trong OTA\_0 sau khi nâng cấp, OTA Demo sẽ ghi phần FW mới vào phân vùng OTA\_1 khi nâng cấp.  
Sau khi bắt đầu lại, nó sẽ thực thi FW tại phân vùng OTA\_1. Tương tự, chương trình nâng cấp tiếp theo luôn được ghi giữa hai phân vùng OTA\_0 và OTA\_1 mà không ảnh hưởng đến FW tại phân vùng Factory App.

# SECURE ESP32

Sau khi hoàn thành xong 1 dự án thì vấn đề về bảo mật luôn được nhắc tới. Đặc biệt là các sản phẩm IOT có quan hệ mật thiết với hệ thống Cloud và chúng cũng rất dễ bị cho vào tầm ngắm để các đối tượng xấu clone lại ra các sản phẩm tương tự mà không cần biết chương trình code ra sao.

Vì vậy chúng ta sẽ luôn luôn cần tìm cách để bảo mật cho các nội dung code chứa trong thiết bị.

Có 2 mức độ bảo mật Firmware mà ESP-IDF cung cấp sẵn cho chúng ta:

- **Secure Boot**
- **Flash Encryption**

# SECURE BOOT

Bảo vệ tính xác thực và tính toàn vẹn của phần FW được lưu trữ trong bộ nhớ Flash SPI bên ngoài.

Kẻ tấn công có thể dễ dàng sửa đổi nội dung của FW và chạy mã độc hại của họ trên ESP32. Khởi động an toàn hiện diện để bảo vệ chống lại kiểu sửa đổi này.

Khởi động an toàn tạo ra một chuỗi tin cậy từ Bootloader cho đến phần FW ứng dụng. Nó đảm bảo mã chạy trên thiết bị là mã chính hãng và không thể sửa đổi nếu không ký mã nhị phân (sử dụng private key). Nếu không, thiết bị sẽ không thực thi các file FW chưa được ký.

# FLASH ENCRYPTION

Flash encryption là một tính năng để mã hóa nội dung của Flash SPI đính kèm của ESP32.

Khi Flash encryption được enable, khả năng đọc vật lý của flash SPI không đủ để khôi phục hầu hết các nội dung flash.

Khi enable Flash encryption, các loại dữ liệu Flash sau được mã hóa theo mặc định:

- Bootloader
- Partition Table
- App partition
- Các loại dữ liệu Flash khác được mã hóa theo chỉ thị cờ “encrypted”

Các nội dung được mã hóa sử dụng thuật toán AES256.

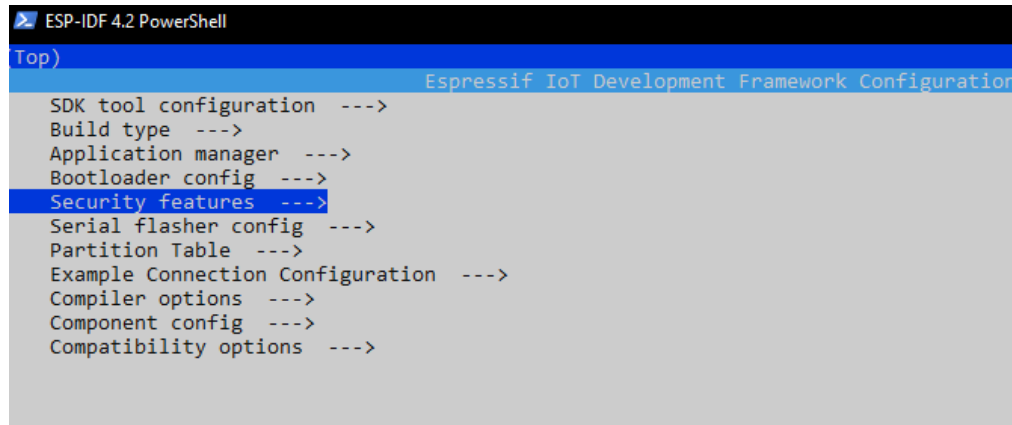
Ví dụ file Partitions.csv sử dụng Flash Encryption.

```
# Name, Type, SubType, Offset, Size, Flags
# Note: if you have increased the bootloader size, make sure
nvs, data, nvs, , 0x10000,
otadata, data, ota, , 0x2000, encrypted
phy_init, data, phy, , 0x1000, encrypted
clientcert, data, 0xff, 0x1d000 , 0x1000, encrypted
clientkey, data, 0xff, 0x1e000 , 0x1000, encrypted
cacrt, data, 0xff, 0x1f000 , 0x1000, encrypted
deviceinfo, data, 0xff, 0x20000 , 0x4000, encrypted
ota_0, app, ota_0, , 0x1A0000,
ota_1, app, ota_1, , 0x1A0000,
```



# ENABLE FLASH ENCRYPTION

**B1:** Vào menuconfig enable chức năng security, đổi vị trí Partition Table từ 0x8000 → 0x9000.



```
ESP-IDF 4.2 PowerShell
Top)
Espressif IoT Development Framework Configuration
SDK tool configuration --->
Build type --->
Application manager --->
Bootloader config --->
Security features --->
Serial flasher config --->
Partition Table --->
Example Connection Configuration --->
Compiler options --->
Component config --->
Compatibility options --->
```

**B2:** Tạo 1 file scrip (.bat) và copy paste nội dung dưới đây

```
idf.py -p $port erase_flash

echo generate encrypt flash key
esptool.py generate_flash_encryption_key my_flash_encryption_key.bin

echo burn key to flash
esptool.py -p $port burn_key flash_encryption my_flash_encryption_key.bin

esptool.py burn_efuse FLASH_CRYPT_CONFIG 0xf

esptool.py burn_efuse FLASH_CRYPT_CNT

echo flash all application to flash
idf.py -p $port encrypted-flash monitor

echo flash only application to flash
idf.py -p $port encrypted-app-flash monitor
```

# ESPTOOL

**WRITE (ghi 1 file bất kỳ vào bộ nhớ Flash của ESP32)**

```
esptool.py --port PORT write_flash 0x1000 my_app-0x01000.bin
```

**READ (đọc nội dung Flash của ESP32 ra ngoài)**

```
esptool.py -p PORT -b 460800 read_flash 0 0x200000 flash_contents.bin
```

**ERASE (Xóa toàn bộ nội dung trên Flash của ESP32)**

```
esptool.py -p PORT erase_flash
```

**ENCRYPTION CUSTOMER FILE (tự mã hóa 1 tập tin bất kỳ để ghi vào Flash ESP32)**

```
espsecure.py encrypt_flash_data --keyfile my_flash_encryption_key.bin --address 0x10000 -o  
build/my-app-encrypted.bin build/my-app.bin
```