



PCBWay HIGH-QUALITY PCB



ONLY \$5 FOR 10 PIECES

- Rogers, HDI, aluminum and rigid-flex PCB are available now
- Production time 24 hours

PCB ASSEMBLY
Free shipping + Free stencil

ONLY \$30

- Component sourcing
- Quality assurance



ESP32 EEPROM Tutorial & Library Examples (Arduino IDE)

by Khaled Magdy

This is a comprehensive guide for ESP32 EEPROM Memory & Library For Arduino Core. The EEPROM is one type of NVM (Non-Volatile Memories), which means the data stored in it doesn't get lost when the ESP32 loses power or goes into a hard reset.

You'll learn how it works and discover that the ESP32 EEPROM library is emulating the functionality of EEPROMs by using the FLASH memory basically. We'll create a couple of example projects to practice and test the ESP32 EEPROM library functions.

Table of Contents

1. ESP32 EEPROM
2. ESP32 EEPROM Library (Arduino Core)
3. ESP32 EEPROM Example (Arduino IDE)
4. Download Attachments
5. Concluding Remarks
6. FAQ & Answers

Before proceeding with this tutorial, you should have installed the ESP32 Arduino Core in your Arduino IDE in order to be able to compile and build projects for ESP32 in Arduino IDE. Follow the tutorial below in order to get started if you haven't done that already.

- Installing ESP32 in Arduino IDE

If you're just starting out with ESP32, it's highly recommended to begin with the following tutorial to kickstart your journey with ESP32 microcontrollers. Then, go to the second link which directs you to the main page for ESP32 Tutorials Series, where you'll find all ESP32 tutorials ordered and categorized in a logical way that guarantees you systematic progress in learning ESP32 programming and IoT.

- Getting Started With ESP32
- ESP32 Tutorials Series (Main Page)

 **Note**

Please, be advised that the EEPROM library is **deprecated** and replaced by the **Preferences.h** library for ESP32. And it's generally better for new projects to use the Preferences library instead. The ESP32 Preferences library makes use of the **ESP32 NVS** driver's capabilities and is generally considered a wrapper layer on top of the ESP32 NVS.

Another option is to use the **External SPI FLASH** which also has simple library interfaces that will help in most simple applications. You can learn more about ESP32 NVS (Non-Volatile Storage) Options by checking the tutorials suggested below.

ESP32 EEPROM

An EEPROM (Electrically Erasable Programmable Read-Only Memory) is a type of NVM (Non-Volatile Memory) just like FLASH. Both memories are used for permanent data storage, as the data saved in EEPROM or FLASH doesn't get lost when the microcontroller loses power or goes into a hard reset.

First of all, the **ESP32 does not have an internal EEPROM** memory at all. The ESP32 EEPROM library however does emulate the functionality by writing and reading data to/from the FLASH memory. It was an early implementation that was based on the ESP32 NVS (Non-Volatile Storage) driver code and it looks very similar to the Arduino well-known EEPROM library that everyone is used to.

As of the time of writing this tutorial, the EEPROM library for ESP32 is declared deprecated and replaced with a recent more robust implementation that also uses the FLASH memory of the ESP32 and it's called the Preferences library. And you can learn more about it in this tutorial.

Applications For ESP32 EEPROM & NVS

Non-Volatile Memories are typically used when we need to save some data and need the microcontroller to remember it even if it lost power or got restarted. Typical use cases for data that we need to store in NVS include the following application examples:

- Passwords
-

- WiFi Credentials
- User Configuration
- App-Level Parameters
- Sensor Last Reading
- State Variables (for some state machines)
- Sensor Calibration Data
- and much more...

ESP32 EEPROM Size

Despite the fact that the ESP32 EEPROM library is using the FLASH memory on-board which is 4MB in size, the **ESP32 EEPROM allowable size is 20kB “theoretically”**. This is according to the memory map for ESP32 FLASH partitions that you can see below (check the reference).

#	Name	Type	SubType	Offset	Size
2	nvs	data	nvs	0x9000	0x5000
3	otadata	data	ota	0xe000	0x2000
4	app0	app	ota_0	0x10000	0x140000
5	app1	app	ota_1	0x150000	0x140000
6	spiffs	data	spiffs	0x290000	0x160000
7	coredump	data	coredump	0x3F0000	0x10000

As you can see in the memory map table above, the NVS section which is used by the **EEPROM library has a size of 0x5000 bytes which is equal to 20kB**. Which is again, theoretically, the maximum allowable memory space for data storage in the NVS.

I’ve done some testing and I couldn’t write/read in any location after **11kB which was the maximum size I could use** for data storage with the EEPROM library. This doesn’t align with the memory map table that I found on the ESP32 Arduino repo, but it’s much better than what I found in other articles online that claim ESP32 EEPROM has a maximum of 512 bytes or 4kB.

Out of curiosity, I brought in another ESP32 board (identical to the one used in the first test) and conducted the same tests again. The result was: I could easily and consistently write and read from memory locations within **12kB-13kB**. The limit was a little bit less than 13kB, and of course, the test setup and code were exactly the same.

All in all, it turned out to be something around the 11kB mark which is a lot of memory space for most applications. Not that bad considering that the better and newer library

implementation (**Preferences**) has a **maximum allowable size of 20kB** given that it's built on top of the NVS driver.

ESP32 EEPROM & FLASH Write Cycles

Generally speaking, the EEPROM memories tend to have 10x times more write/erase cycles than FLASH memory. A typical EEPROM can have 100,000 up to 1,000,000 write/erase cycles while FLASH memories have typical write/erase cycles of around 10,000 up to 100,000 at maximum.

Given that ESP32 doesn't have an EEPROM and it's actually using the FLASH memory under the hood, we're expecting to have around 10,000 write cycles for each memory location in the FLASH. In other words, if you keep writing a byte of data to the same address location in the FLASH memory, it'd take only 10,000 write operations to permanently damage that memory location forever.

! Note

Please, be cautious when using any sort of NVS memory as it can easily get damaged permanently if you're not wisely using it. Try to avoid unnecessary regular writing cycles to the NVS unless it's mandatory. Better library implementations do have some **wear and tear leveling** algorithms in order to distribute the memory utilization across all of the sectors so you have a longer FLASH memory lifetime.

If you keep writing to the same memory section addresses over and over again, it'll eventually get permanently damaged and this can happen rather quickly if no attention is paid.

ESP32 EEPROM Library (Arduino Core)

The only and biggest advantage of using the ESP32 EEPROM library for saving data to the FLASH memory is that it's very similar to using the Arduino EEPROM library which most of us are already familiar with.

By using the ESP32 EEPROM library, you'll have up to 11kB of memory to use in the FLASH memory for data storage applications. In other words, you have 11264 distinct addresses for memory locations that you can use for saving data. Each memory location is a single byte that can hold 8-Bit data (a value between 0 and up to 255) in each and every memory address.

EEPROM Write

To use the ESP32 EEPROM library, you have to include the EEPROM.h header file.

```
// Include The EEPROM.h Library To Read & Write To The FLASH Memory
#include <EEPROM.h>
```

Then, you need to specify the maximum memory size that you'd need in your application for saving data in the EEPROM. If you know exactly how many bytes you need, then use it as a limit. Otherwise, allocate a slightly larger chunk of memory to give yourself a safety buffer.

```
// Define The Number of Bytes You Need
#define EEPROM_SIZE 1 // This is 1-Byte
```

Next, you need to start the EEPROM by calling the `begin(EEPROM_SIZE)` function. Which takes the size of EEPROM memory that you've previously defined as an argument. And now, you've successfully allocated the memory needed for saving your data into ESP32 NVS (FLASH).

```
void setup()
{
    ...
    // Allocate The Memory Size Needed
    EEPROM.begin(EEPROM_SIZE);
}
```

Now, you can use the EEPROM for writing data by calling the `EEPROM.write(address, data)` function. Which takes two arguments: the address and the data itself. The address can be any location within the memory limit that you've defined and the data is an 8-bit value (from 0 to 255).

```
// Write Data To EEPROM  
EEPROM.write(address, data);
```

Given that the EEPROM library is based on the NVS Flash driver, and it's actually a Flash memory, it doesn't support single-byte addressing. Generally, EEPROMs are known for the ability to read/write a single address location at a time. However, Flash memories are page-addressed on the other hand. A typical Flash page is 64-Bytes. Therefore, you need to read-modify-write an entire page in order to swap two bytes for example.

The EEPROM library deals with this by saving the data to a buffer when you call the **write()** function. And it's not written to the actual FLASH memory until you call **EEPROM.commit()** function. And it's the next and final step to do.

```
// Call commit() Function To Actually Save The Data You've Written E  
EEPROM.commit();
```

EEPROM Read

To read a byte (single memory location) from the FLASH, you should use the **EEPROM.read(address)** function which takes the address of the memory location that you want to get its content.

```
// Read The Data Stored @ a Specific Address Location  
data = EEPROM.read(address);
```

EEPROM Write/Read String

To Write a String in ESP32 EEPROM, use the following function from the EEPROM library.

```
// Write a String starting from a specific address location  
EEPROM.writeString(address, &myString);
```

To read a String from ESP32 EEPROM, use the function shown below.

```
// This will return the String stored @ the specified address
String myStr = EEPROM.readString(address);
```

EEPROM Write/Read Float

To write or read a Float variable in ESP32 EEPROM memory (FLASH), you don't need to check how it's stored in the memory byte-by-byte. Just call the following two functions for write and read operations.

```
// Write a Floating Point Variable To EEPROM
EEPROM.writeFloat(address, myFloat);

// Read a Float Variable From EEPROM @ Specific Address
myFloat = EEPROM.readFloat(address);
```

EEPROM Read Write Other Data Types

Here are some APIs in the ESP32 EEPROM library that give you the ability to write and read all known data types.

```
// [EEPROM Write Functions]
size_t writeByte(int address, uint8_t value);
size_t writeChar(int address, int8_t value);
size_t writeUChar(int address, uint8_t value);
size_t writeShort(int address, int16_t value);
size_t writeUShort(int address, uint16_t value);
size_t writeInt(int address, int32_t value);
size_t writeUInt(int address, uint32_t value);
size_t writeLong(int address, int32_t value);
size_t writeULong(int address, uint32_t value);
size_t writeLong64(int address, int64_t value);
size_t writeULong64(int address, uint64_t value);
size_t writeFloat(int address, float_t value);
size_t writeDouble(int address, double_t value);
size_t writeBool(int address, bool value);
size_t writeString(int address, const char* value);
size_t writeString(int address, String value);
size_t writeBytes(int address, const void* value, size_t len);
```



```
// [EEPROM Read Functions]
uint8_t readByte(int address);
int8_t readChar(int address);
uint8_t readUChar(int address);
int16_t readShort(int address);
uint16_t readUShort(int address);
int32_t readInt(int address);
uint32_t readUInt(int address);
int32_t readLong(int address);
uint32_t readULong(int address);
int64_t readLong64(int address);
uint64_t readULong64(int address);
float_t readFloat(int address);
double_t readDouble(int address);
bool readBool(int address);
size_t readString(int address, char* value, size_t maxlen);
String readString(int address);
size_t readBytes(int address, void * value, size_t maxlen);
```

EEPROM Read Write All Data Types

There are also two interesting APIs (functions) in the ESP32 EEPROM library that you can use in order to write and read any known data type or even user-defined data types like structures and so on. Those are: the **put()** and **get()** functions.

```
// Write Any Data Type To ESP32 EEPROM
EEPROM.put(address, myData);

// Read Any Data Type From ESP32 EEPROM
EEPROM.get(address, myData);
```

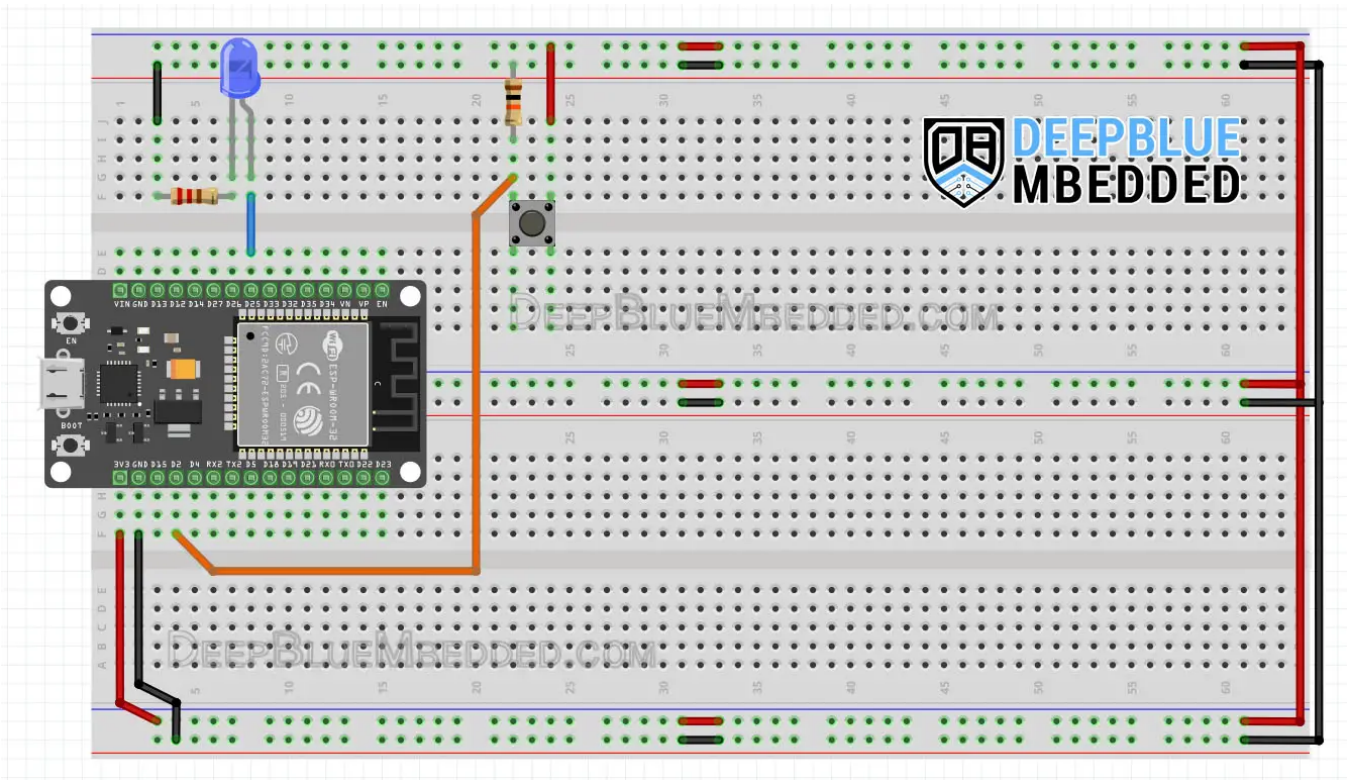
ESP32 EEPROM Example (Arduino IDE)

In this example, we'll test the EEPROM memory by saving the last LED state. The LED is toggled by a push button, and whenever its state changes, the last state will be saved to EEPROM. We'll reset the ESP32 board and it should retrieve the last saved state from

EEPROM.

Wiring

Here is how to hook up the input push button and the LED output.



Example Code

Here is the full code listing for this example.

<div>Pdf document...</div> <div>Download</div>	<div>Zip folder icon</div> <div>Download</div>	<div>A</div> <div>S</div>
--	--	---------------------------

```

* LAB Name: ESP32 EEPROM Library Demo
* Author: Khaled Magdy
* DeepBlueMbedded 2023
* For More Info Visit: www.DeepBlueMbedded.com
*/
// Include The EEPROM.h Library To Read & Write To The FLASH Memory
#include <EEPROM.h>

// Define The Number of Bytes You Need
#define EEPROM_SIZE 1
// Define The LED Output GPIO Pin & Button Input GPIO Pin
#define LED_GPIO 25
#define BTN_GPIO 2

// Global Variables For Button Reading & Debouncing
int ledState = LOW;
int btnState = LOW;
int lastBtnState = LOW;
int lastDebounceTime = 0;
int debounceDelay = 50;
int eeprom_address = 0;

void setup() {
    Serial.begin(115200);
    pinMode(LED_GPIO, OUTPUT);
    pinMode(BTN_GPIO, INPUT);
    EEPROM.begin(EEPROM_SIZE);
    ledState = EEPROM.read(eeprom_address);
    digitalWrite(LED_GPIO, ledState);
}

void loop() {
    int reading = digitalRead(BTN_GPIO);
    if (reading != lastBtnState) {
        lastDebounceTime = millis();
    }
    if ((millis() - lastDebounceTime) > debounceDelay) {
        if (reading != btnState) {
            btnState = reading;
            if (btnState == HIGH) {
                ledState = !ledState;
                digitalWrite(LED_GPIO, ledState);
                EEPROM.write(eeprom_address, ledState);
                EEPROM.commit();
                Serial.println("State Changed & Saved To FLASH!");
            }
        }
    }
}

```

```
    }  
  }  
}  
lastBtnState = reading;  
}
```

Code Explanation

The code example does simply read the button input pin and debounces it to make sure it's not picking up any noise. And toggle the LED if the button is actually pressed. The LED toggle event does also a write operation to the **eeeprom_address** location and saves the current **ledState** in the FLASH memory.

We start by including the EEPROM.h Library

```
// Include The EEPROM.h Library To Read & Write To The FLASH Memory  
#include <EEPROM.h>
```

Then, we define the needed memory size of EEPROM which is only 1 byte. And also define the GPIO pins we'll be using.

```
// Define The Number of Bytes You Need  
#define EEPROM_SIZE 1  
// Define The LED Output GPIO Pin & Button Input GPIO Pin  
#define LED_GPIO 25  
#define BTN_GPIO 2
```

We also define some global variables to save the LED state, and button state, and perform the button debouncing logic.

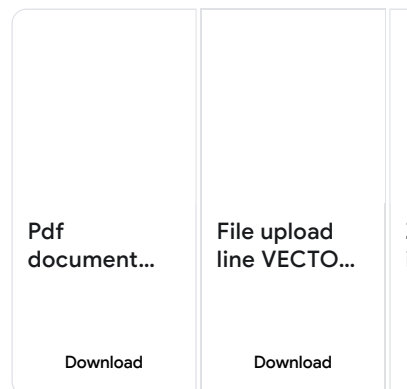
```
// Global Variables For Button Reading & Debouncing  
int ledState = LOW;  
int btnState = LOW;  
int lastBtnState = LOW;  
int lastDebounceTime = 0;  
int debounceDelay = 50;  
int eeeprom address = 0;
```

setup()

in the setup() function, we initialize serial communication for debugging, the pin modes, and the EEPROM with the pre-defined size.

```
Serial.begin(115200);  
pinMode(LED_GPIO, OUTPUT);  
pinMode(BTN_GPIO, INPUT);  
EEPROM.begin(EEPROM_SIZE);
```

We also read the last saved LED state from the FLASH memory and apply it to the LED output. This happens only at startup, hence it's done in setup() function.



```
ledState = EEPROM.read(eeprom_address);  
digitalWrite(LED_GPIO, ledState);
```

loop()

in the loop() function, it's mostly about reading the button and debouncing it. Once, it's clear that the button is held HIGH and it's not a noise, we take the LED toggle action +

saving the current LED state to the EEPROM (Flash Memory).

Pdf document note icc

Download

```
int reading = digitalRead(BTN_GPIO);
if (reading != lastBtnState)
{
    lastDebounceTime = millis();
}
if ((millis() - lastDebounceTime) > debounceDelay)
{
    if (reading != btnState)
    {
        btnState = reading;
        if (btnState == HIGH)
        {
            ledState = !ledState;
            digitalWrite(LED_GPIO, ledState);
            EEPROM.write(eeprom_address, ledState);
            EEPROM.commit();
            Serial.println("State Changed & Saved To FLASH!");
        }
    }
}
lastBtnState = reading;
```

Testing Results

Here is a short demo video for the testing result of this example. Note that the ESP32 is remembering the last LED state after every reset operation. And that's the most significant application for using non-volatile memories.



Parts List

Here is the full components list for all parts that you'd need in order to perform the practical LABs mentioned here in this article and for the whole ESP32 series of tutorials found here on DeepBlueMbedded. Please, note that those are affiliate links and we'll receive a small commission on your purchase at no additional cost to you, and it'd definitely support our work.

ESP32 Course Kit List

Download Attachments

You can download all attachment files for this Article/Tutorial (project files, schematics, code, etc..) using the link below. Please consider supporting my work through the various support options listed in the link down below. Every small donation helps to keep this website up and running and ultimately supports our community.

[DOWNLOAD](#)

[DONATE HERE](#)

Concluding Remarks

To conclude this tutorial, we'll highlight the fact that ESP32 does not have an EEPROM and the library called EEPROM.h is just an old wrapper for NVS that uses the FLASH memory under the hood. It's also deprecated as of the time of writing this tutorial, and the ESP32 EEPROM library alternative is currently the **Preference.h** library.

Please, pay attention to the writing operations of NVS as it can quickly get damaged if you're not dealing carefully with it. Try to avoid any redundant write operations and commit your changes to the memory location as less as you can.

FAQ & Answers

Does ESP32 Have EEPROM?

ESP32 does not have an internal EEPROM. However, the EEPROM library for ESP32 is actually using the NVS (FLASH Memory) of ESP32 under the hood to emulate the EEPROM functionality. The EEPROM Library is currently deprecated and replaced by the Preferences library.

Where is EEPROM in ESP32?

It's not in the ESP32 microcontroller at all, as ESP32 does not have an internal EEPROM. However, there is an EEPROM library for ESP32 which has the same well-known and user-friendly function of Arduino EEPROM that most users are already familiar with. But in the end, this EEPROM library is built on top of the NVS (Flash Memory) driver of the ESP32.

Does the ESP32 have internal EEPROM?

No, it doesn't. However, there is an ESP32 library that's called EEPROM which is basically built on top of the NVS (Flash Memory) driver to emulate the EEPROM functionality by using FLASH memory.

How to save data in EEPROM in ESP32?

Simply use the EEPROM library or its new alternative Preferences library that has replaced the old ESP32 EEPROM library.

What is the maximum EEPROM size ESP32?

Theoretically, it's about 20kB. But in practice, it can be around 11kB or 12kB, while online articles claim it is 512bytes or even 4kB. The best answer can be to test it yourself or use the new Preferences library that's guaranteed to give you up to 20kB of FLASH space.

Share This Page With Your Network!



Join Our +25,000 Newsletter Subscribers!

Stay Updated With All New Content Releases. You Also Get Occasional FREE Coupon Codes For Courses & Other Stuff!

Subscribe

📁 Embedded Systems, Embedded Tutorials, ESP32, IoT

🔑 ESP32 Arduino, ESP32 Core

◀ ESP32 Keypad Matrix (Kaypad-LCD Example) – Arduino IDE

▶ ESP32 Flash Memory (Save Permanent Data) – Arduino IDE

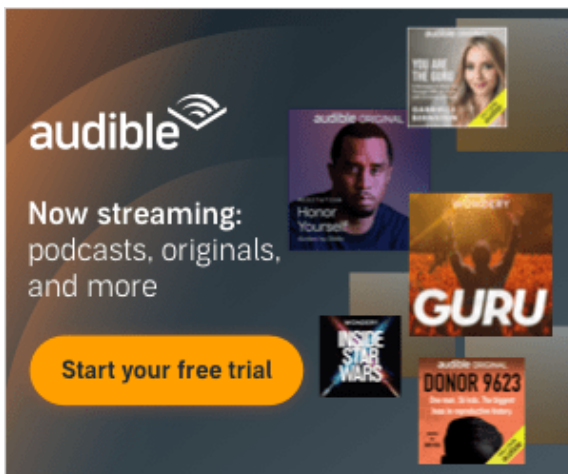
Author

Khaled Magdy

I'm an embedded systems engineer with several years of experience in embedded

software and hardware design. I work as an embedded SW engineer in the Automotive industry. However, I still do Hardware design and SW development for DSP, Control Systems, Robotics, Ai/ML, and other fields I'm passionate about. I love reading, writing, creating projects, and Technical training. A reader by day and a writer by night, it's my lifestyle. You can view my profile or follow me via contacts.

Leave a Reply



Learn ESP32 Basics

Getting Started With ESP32

ESP32 GPIO (Inputs/Outputs)

ESP32 Serial Print

ESP32 PWM Tutorial

ESP32 Change CPU Clock Speed

ESP32 ADC Tutorial

ESP32 External Interrupt Pins

ESP32 I2C Tutorial

ESP32 Timers & Timer Interrupts

ESP32 Sleep Modes

ESP32 DAC & Audio Tutorial

ESP32 EEPROM Library

ESP32 Flash Memory Tutorial

ESP32 Connectivity

ESP32 Bluetooth Classic

ESP32 WiFi Tutorial

ESP32 Networking

Connect To a WiFi Network

ESP32 WiFi Scanner

ESP32 WiFi Signal Strength

ESP32 MAC Address Get & Set

ESP32 Static IP Address

WiFi Hostname Change

ESP32 Displays

LCD 16x2 Display

I2C LCD 16x2 (PCF8574)

ESP32 Sensors

LM35 Temperature Sensor

ESP32 Modules

ESP32 Keypad 4x4 Interfacing

Graduate from a cutting-edge institution abroad. Aim for academic excellence with IELTS.

British Council



[report this ad](#)

Search The Blog

Categories

Select Category



Subscribe To Our Newsletter To Get All New Updates

Type your email...

Subscribe

Access to Top US Brand

Get a Free 30 Day Premi
Membership Today!



[report this ad](#)

Categories

Select Category



Search The Website

Search

Resources

[STM32 ARM MCUs Programming Course](#)

[Embedded Systems - PIC Course](#)

[DeepBlue Patreon Page](#)

[PayPal Donation](#)

[Books Recommendation List](#)

ABOUT DEEPBLUE

DeepBlueMbedded is an educational website where you can find technical content (Articles – Tutorials – Projects – etc..). Mainly on Embedded Systems & ECE Related topics.

topics. You'll find also downloadable resources like firmware code examples, schematics, hardware designs, and more.

It's been and will always be a free resource of information. So, please consider supporting this work if possible.

SHARE & SUPPORT

You can always show your support by sharing my articles and tutorials on social networks. It's the easiest way to support my work that costs nothing and would definitely be appreciated!



OR You Can Support Me Through This Link

SUPPORT

DISCLOSURE

DeepBlueMbedded.com is a participant in the Amazon Services LLC Associates Program, eBay Partner Network EPN, affiliate advertising programs designed to provide a means for sites to earn advertising fees by advertising and linking to Amazon.com and eBay.com

You can also check my Full [Disclaimer](#) Page For More Information.

Copyright © 2023 DeepBlueMbedded.com . All Rights Reserved.

[Privacy Policy](#) | [Trademark Information](#) | [Disclaimer](#)