Arduino IDE and ESP32forth

Marc PETREMANN

Version 1.0 - January 23, 2026

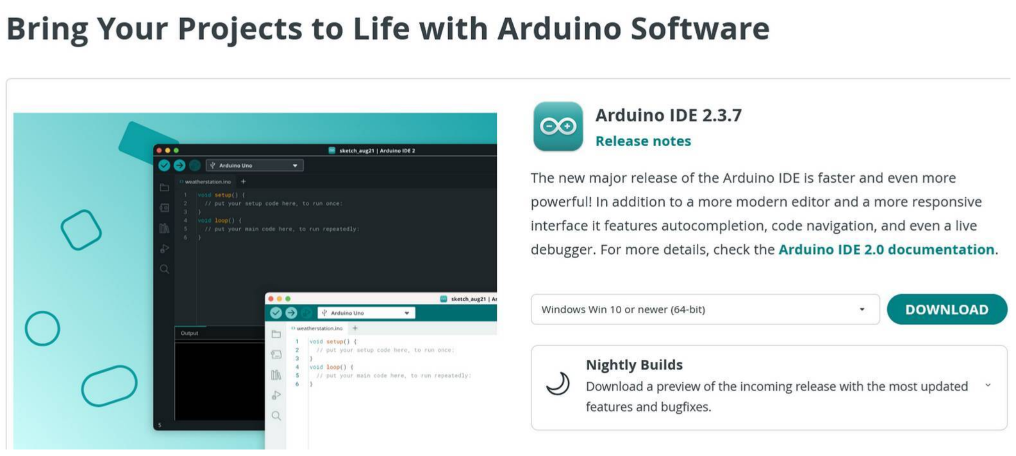
Table of Contents

Preamble

This manual is intended to help you understand how to use the ARDUINO IDE program to compile ESP32forth for EPS32 boards and incidentally all other programs in C language for the range of boards in the ARDUINO environment.

Install ARDUINO IDE

Download the latest ARDUINO IDE version, available on the official website;
https://www.arduino.cc/en/software/



Bring Your Projects to Life with Arduino Software

Arduino IDE 2.3.7
Release notes

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger. For more details, check the **Arduino IDE 2.0 documentation**.

Windows Win 10 or newer (64-bit)

DOWNLOAD

Nightly Builds
Download a preview of the incoming release with the most updated features and bugfixes.

Select the target system, then click *Download* .

Once downloaded, follow the instructions specific to your system (Windows, Linux, MacOS…).

Installation under Windows

Here, the downloaded version has the filename *arduino-ide_2.3.7_Windows_64bit.exe* .

After downloading, the file should be located in the *Downloads folder* .

To begin the installation, run this file. Follow the steps requested during the installation process.

ARDUINO IDE

In the context of Arduino, the acronym **IDE** stands for **Integrated Development Environment** , which translates into French as **Environnement de Développement Intégré** .

This is the indispensable tool that serves as a "command center" for creating your electronic projects.

The IDE is a single software package that brings together all the tools a programmer needs to transform an idea into code, and then into action on an Arduino board. It mainly consists of three parts:

- **The Text Editor** : This is where you write your code (called a "sketch" in Arduino). It highlights keywords to help you avoid syntax errors.
- **The Compiler** : It translates your code (human-readable) into machine language (0s and 1s) that the microcontroller on the board can understand.
- **The Programmer (Uploader)** : It sends the compiled code to your Arduino board via the USB cable.

Before the existence of IDEs, developers had to use separate software: one for writing, one for compiling, and another for uploading the file to the chip. The Arduino IDE integrates all of this into a simple interface, making electronics accessible even to beginners.

The two main versions:

- Arduino IDE 1.8.x: The classic version, very stable and lightweight.
- Arduino IDE 2.x: The modern version with autocompletion (it guesses what you are going to write) and an integrated debugger.

To compile and download ESP32forth, you will need version 2.x.

Download and prepare ESP32forth

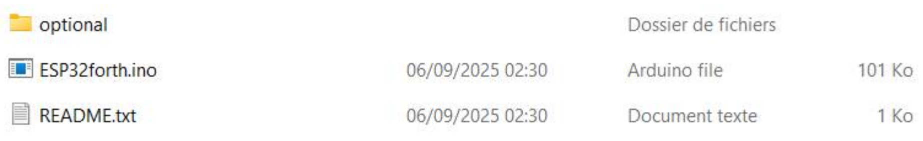
To download ESP32forth, go here:
https://esp32forth.appspot.com/ESP32forth.html

Prefer version 7.0.7.21. This is the version compatible with ARDUINO IDE 2.x.

Preparing files for ESP32forth

Open the Zip file containing ESP32forth and extract its contents to your development workspace. It is best to choose a workspace somewhere other than your Windows desktop.

Once extracted, you should have an ESP32forth folder containing these files:



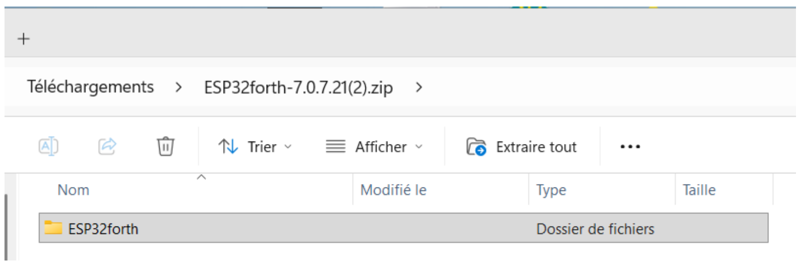
optional		Dossier de fichiers	
ESP32forth.ino	06/09/2025 02:30	Arduino file	101 Ko
README.txt	06/09/2025 02:30	Document texte	1 Ko

If you have never compiled and installed ESP32forth, leave the contents of these folders as they are.

If you have never developed for Arduino, ESP32 or any other electronic board, create a **"developments" folder** in your Windows user space:

`c: → Users → myself → developments → ESP32`

In this **development folder** , create another folder named **ESP32** . Copy the contents of the previously downloaded Zip file into this folder.



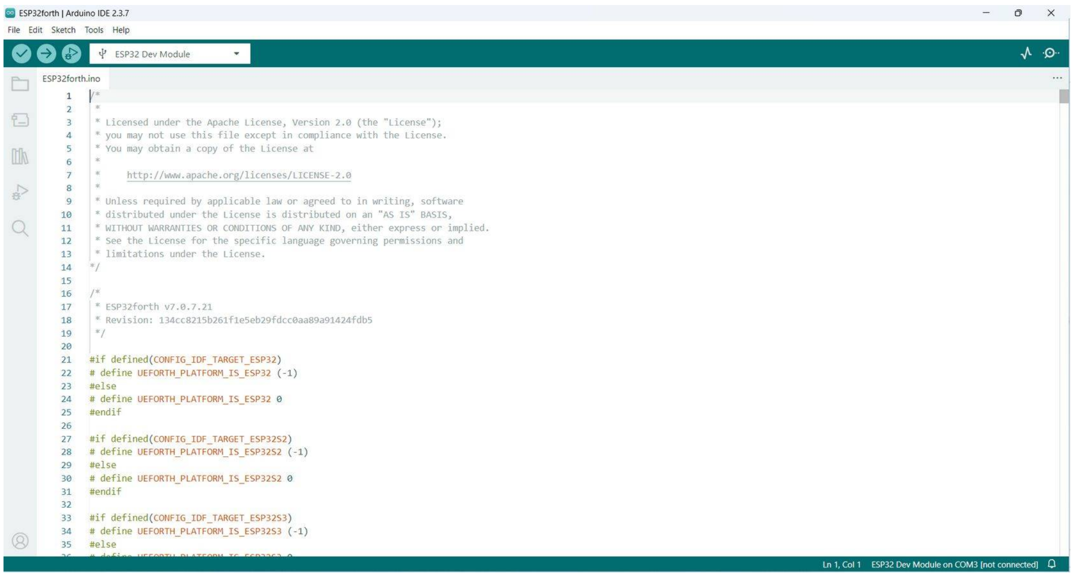
Téléchargements > ESP32forth-7.0.7.21(2).zip >

Nom	Modifié le	Type	Taille
ESP32forth		Dossier de fichiers	

If you have other codes to create for ESP32, you will define other directories in the **ESP32 folder** .

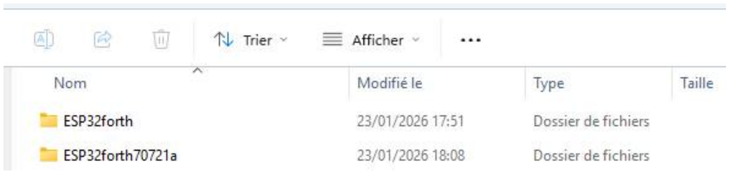
Open a project

Open the Arduino IDE. Then open the previously downloaded project. To do this, click on *File* and select *Open* . Locate the ESP32forth project. Select the **ESP32forth.ino file** . You will see this window:


ESP32forth | Arduino IDE 2.3.7
File Edit Sketch Tools Help
ESP32 Dev Module
ESP32forth.ino
Ln 1, Col 1 ESP32 Dev Module on COM3 [not connected]

```
/*
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
*/

/*
 * ESP32forth v7.0.7.21
 * Revision: 134cc8215b261f1e5eb29fdcc0aa89a91424fdb5
 */

#if defined(CONFIG_IDF_TARGET_ESP32)
# define UEFORTH_PLATFORM_IS_ESP32 (-1)
#else
# define UEFORTH_PLATFORM_IS_ESP32 0
#endif

#if defined(CONFIG_IDF_TARGET_ESP32S2)
# define UEFORTH_PLATFORM_IS_ESP32S2 (-1)
#else
# define UEFORTH_PLATFORM_IS_ESP32S2 0
#endif

#if defined(CONFIG_IDF_TARGET_ESP32S3)
# define UEFORTH_PLATFORM_IS_ESP32S3 (-1)
#else
```

IMPORTANT: The project must have the same name as the folder that contains it. For example, if you want to make a copy of this project to make changes, create a new folder named **ESP32forth70721a** and place the copy of **ESP32forth.ino** in this new folder. Then rename **ESP32forth** to **ESP32forth70721a** .



Nom	Modifié le	Type	Taille
ESP32forth	23/01/2026 17:51	Dossier de fichiers	
ESP32forth70721a	23/01/2026 18:08	Dossier de fichiers	

You will be able to make changes in this new folder without altering the contents of the **ESP32forth folder** .

At this stage, we need to manage a few parameters before uploading our project to the ESP32 board.

The tools

The tools can be accessed by clicking on *Tools* in the menu bar. There are many of these tools:

```
Auto Format
Archive Sketch
Manage Libraries
Serial Monitor
Serial Plotter
---------------------------------
Firmware Updater
Upload SSL Root Certification
---------------------------------
Board:
Port:
Reload Bord Data
Get Board Info
---------------------------------
CPU Frequency:
Core Debug Level
Erase All Flash Before Sketch Upload:
Events Run On:
Flash Frequency:
Flash Mode:
Flash Size:
JTAG Adapter:
Arduino Runs On:
Partition Scheme:
PSRAM:
Upload Speed:
ZigBee Mode:
---------------------------------
Program
Burn Bootloader
```

Our aim is not to detail all these tools. We will focus primarily on the parameters essential to our **ESP32forth project** .

Auto Format

Auto Format function is the best friend of developers who like clean work (or those who are a little messy).

In practical terms, it is used to automatically reorganize the layout of your code to make it perfectly readable, without changing how it works.

In programming, readability is crucial. Auto Format addresses several aspects:

- **Indentation** : It correctly aligns left-aligned elements. For example, everything inside a ` `void loop() { ... }` function` will be indented one space to visually show the hierarchy.
- **The curly braces** : It replaces the `{` and `}` according to precise programming standards, thus avoiding getting tangled up in control structures.
- **Spaces** : He adds spaces where they are missing (for example around an = sign or after a comma) to make the text easier to read.

Archive Sketch

The *Archive Sketch option* is a very useful yet often overlooked feature in the Arduino IDE. It allows you to instantly create a compressed backup copy of your current project.

When working on an electronics project, you often accumulate several files (the main code, .h header files, secondary tabs). This option allows you to neatly "package" everything.

What *Archive Sketch actually does* :

- **.zip compression** : It takes the entire folder of your project and transforms it into a compressed archive.
- **Automatic timestamping** : The IDE automatically names the file with today's date (for example: my_project_jan23a.zip). This allows you to create restore points without overwriting your previous versions.
- **Organization** : The compressed file is placed directly in your project folder, ready to be moved.

The actual usefulness:

- **Easy sharing** : If you want to send your code to a friend or teacher, sending a single *.zip file* is much simpler than sending a whole folder with multiple files.
- **Version History** (Backup): Before attempting a risky modification that could "break" everything, a quick Archive *Sketch* allows you to keep a copy of the current working state.
- **Cleanup** : This allows you to store your old projects compactly on your computer or in the cloud.

Manage Libraries

The **Manage Libraries option** is one of the most powerful tools in the Arduino IDE. It allows you to extend the capabilities of your board without having to reinvent the wheel.

, a **library** is a set of code already written by professionals or the community to make a specific component work (an LCD screen, a temperature sensor, a motor, etc.) *.*

It's a kind of free "application store" for your code. Instead of manually configuring every pin of a complex screen, you install a library that provides you with simple commands like screen.write("Hello") .

The manager is used for:

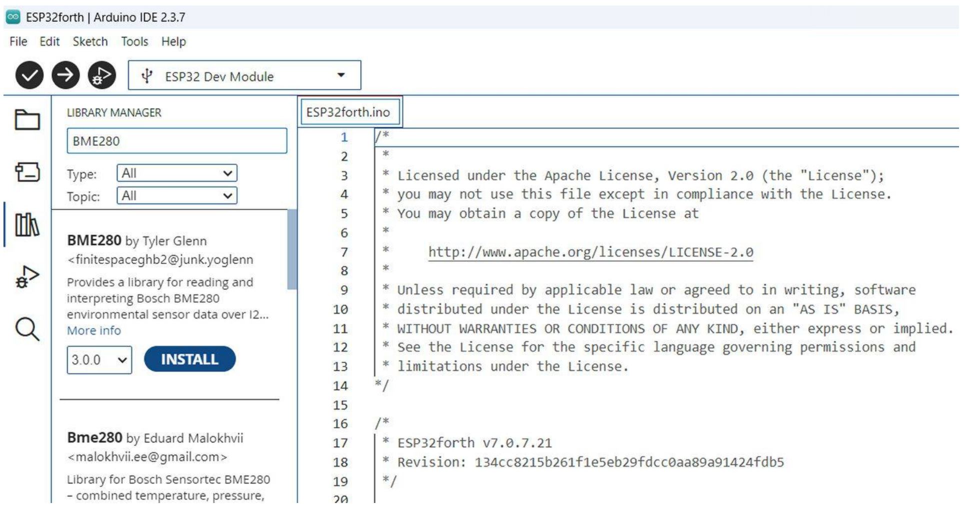
- **Search** for libraries by name or by functionality.
- **Install** new libraries with one click.
- **Update** your libraries to fix bugs or add features.
- **Manage versions** (if an update breaks your code, you can revert to an older version).

Use :

1. **Opening:** Go to **Tools > Manage Libraries** or click on the "books" icon in the left sidebar (in version 2.x).
2. **Search:** Type the name of a component (e.g., "DHT11" for a humidity sensor).
3. **Installation:** Click the "Install" button. The IDE will download and install everything in the correct location for you.

For example, you buy a **BME280** pressure sensor , and you probably won't know how to interpret its raw electrical signals. By going to *Manage Libraries , you search for "BME280", you install the*

Adafruit version, and suddenly, your Arduino understands the `sensor.readPressure()` command .


ESP32forth | Arduino IDE 2.3.7
File Edit Sketch Tools Help
ESP32 Dev Module
LIBRARY MANAGER
BME280
Type: All
Topic: All
BME280 by Tyler Glenn
<finitespaceghb2@junk.yoglenn
Provides a library for reading and interpreting Bosch BME280 environmental sensor data over I2...
More info
3.0.0
INSTALL
Bme280 by Eduard Malokhvii
<malokhvii.ee@gmail.com>
Library for Bosch Sensortec BME280
– combined temperature, pressure,
ESP32forth.ino

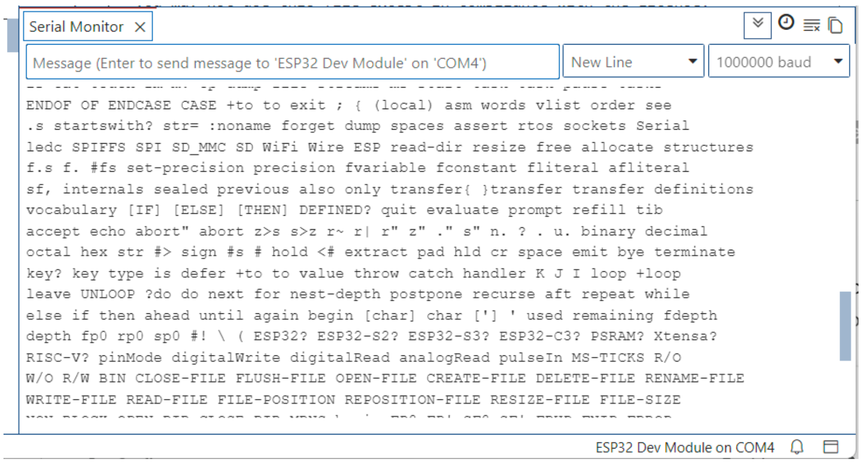
```
/*
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * ESP32forth v7.0.7.21
 * Revision: 134cc8215b261f1e5eb29fdcc0aa89a91424fdb5
 */
```

In the screenshot above, if you click *INSTALL , the* **BME280** library will be installed. However, as long as your project does not explicitly call upon the functions of this library, this installation will have no impact on the size of the binary code generated during the compilation of the application code.

Serial Monitor

The **Serial Monitor** is the essential diagnostic tool integrated into the IDE. It is the communication channel that allows your Arduino board to "talk" to your computer in real time.

Since an ESP32 board does not have a screen by default, *Serial Monitor* serves as a window into what is happening inside the microcontroller:


Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM4')
New Line
1000000 baud
ESP32 Dev Module on COM4

```
ENDOF OF ENDCASE CASE +to to exit ; { (local) asm words vlist order see
.s startswith? str= :noname forget dump spaces assert rtos sockets Serial
ledc SPIFFS SPI SD_MMC SD WiFi Wire ESP read-dir resize free allocate structures
f.s f. #fs set-precision precision fvariable fconstant fliteral afliteral
sf, internals sealed previous also only transfer{ }transfer transfer definitions
vocabulary [IF] [ELSE] [THEN] DEFINED? quit evaluate prompt refill tib
accept echo abort" abort z>s s>z r~ r| r" z" ." s" n. ? . u. binary decimal
octal hex str #> sign #s # hold <# extract pad hld cr space emit bye terminate
key? key type is defer +to to value throw catch handler K J I loop +loop
leave UNLOOP ?do do next for nest-depth postpone recurse aft repeat while
else if then ahead until again begin [char] char ['] ' used remaining fdepth
depth fp0 rp0 sp0 #! \ ( ESP32? ESP32-S2? ESP32-S3? ESP32-C3? PSRAM? Xtensa?
RISC-V? pinMode digitalWrite digitalRead analogRead pulseIn MS-TICKS R/O
W/O R/W BIN CLOSE-FILE FLUSH-FILE OPEN-FILE CREATE-FILE DELETE-FILE RENAME-FILE
WRITE-FILE READ-FILE FILE-POSITION REPOSITION-FILE RESIZE-FILE FILE-SIZE
```

Here, we executed `page vlist` in this terminal window.

For the text to display correctly, the communication speed (the **Baud Rate**) set in the Serial Monitor must be **the same** as that specified in your code (usually **9600** or **115200**). If the speeds do not match, you will see strange or unreadable characters ("gibberish").

Board

The **Board option** in the Arduino IDE is the most crucial setting before uploading your code. It tells the software exactly what type of hardware it is addressing.

Since each chip has a different brain (number of pins, processing speed, memory), the IDE needs to know whether it is preparing the code for a small Arduino Uno or for a powerhouse like the **ESP32** .

Importance for the ESP32

Unlike Arduino, which manufactures its own boards, the **ESP32** is a microcontroller produced by the company *Espressif* , but used by dozens of different manufacturers (DOIT, Adafruit, LilyGO, Heltec...).

Each board has a different pinout. If you choose the wrong board, you risk:

- The code doesn't upload at all.
- That your connections do not match (e.g. you are asking to turn on pin 2, but on your specific model it is pin 4).

Examples of common ESP32 boards

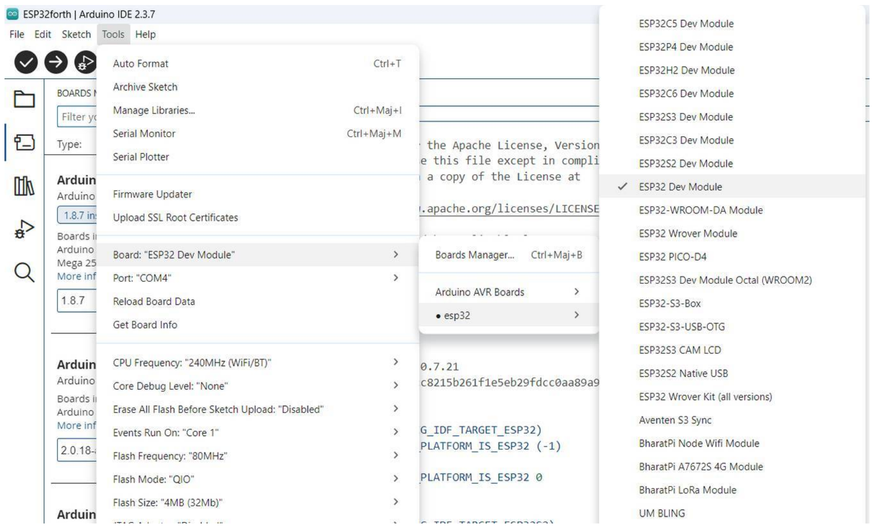
When you go to *Tools → Board → ESP32* , you will see a huge list. Here are the most frequent:

Name in the IDE	**Typical use**
DOIT ESP32 DEVKIT V1	The most classic and inexpensive version (30 pins). Ideal for beginners.
Adafruit Feather ESP32	Compact size with integrated battery connector.
ESP32 Development Module	The "universal" setting. If you cannot find your exact model, this one often works by default.
Wemos D1 Mini ESP32	A miniature version for projects where space is limited.

Configuring an ESP32 board

By default, the Arduino IDE does not recognize the ESP32. It must first be "trained":

1. **Adding the URL:** In *Preferences* , add the link to the Espressif card manager.
2. **Installation:** In *Tools → Board → Boards Manager* , search for "ESP32" and install the package.
3. **Selection:** Once installed, go to *Tools → Board → ESP32* and select your specific model (e.g., *DOIT ESP32 DEVKIT V1*).



Board: "ESP32 Dev Module"
Port: "COM4"
CPU Frequency: "240MHz (WiFi/BT)"
Core Debug Level: "None"
Erase All Flash Before Sketch Upload: "Disabled"
Events Run On: "Core 1"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "4MB (32Mb)"

Warning: Once you have chosen the card, also check the port (COM) in the *Tools menu* , otherwise your computer will not know which "pipe" to send the code through!

Port

This tool allows you to select the serial port to which the ESP32 board to be programmed is connected.

The ARDUINO IDE only displays active or recently used ports. If your ESP32 board is connected to a USB port, the IDE can detect the board and indicate the associated COM port.

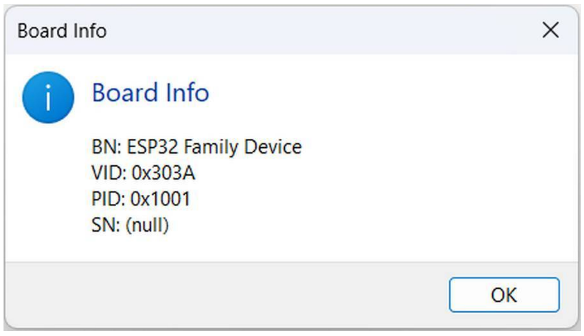
Get Board Info

The **Get Board Info option** is a sort of "digital ID card" that queries the hardware plugged into your USB port.

This is the perfect tool to check if your computer actually "sees" the Arduino and to precisely identify which electronic component is responsible for communication.

This option opens a small window that displays raw technical information sent by the board's USB chip. This is useful for:

1. **Check the connection:** If the window displays an error, it means that the cable is faulty or the drivers are not installed.
2. **Identifying a clone:** If you have purchased an "Arduino compatible" board, this option will tell you which component is used for communication (often a different chip than the original).
3. **Finding the port:** If you have multiple devices plugged in, this confirms that you have selected the correct COM port.



Board Info

BN: ESP32 Family Device
VID: 0x303A
PID: 0x1001
SN: (null)

The window usually displays three key codes:

- **BN (Board Name):** The name of the card (if it is officially recognized).
- **VID (Vendor ID):** The identifier of the manufacturer of the USB component.
- **PID (Product ID):** The identifier of the product model.

- **SN (Serial Number):** The unique serial number of the chip (if available).

It is common for an **ESP32** or a **generic Arduino Nano** to display the message **"Unknown Board"** .

> **Why?** > Because these boards often use generic USB chips (like the **CH340** or **CP2102**). The IDE recognizes that there is "something" at the end of the cable (the VID/PID is displayed), but it cannot guarantee that it is an official Arduino board.

That's okay: as long as you see a VID and PID number, it means the communication is working. You can ignore the "Unknown Board" message and upload your code as usual.

CPU Frequency

The *CPU Frequency option* determines the "thinking speed" of your microcontroller. It's the chip's heartbeat, regulated by an oscillator.

The higher the frequency, the more instructions the ESP32 executes per second.

On a standard Arduino Uno board, the frequency is fixed (16 MHz). But on an **ESP32** , you often have a choice (usually between **80 MHz, 160 MHz, and 240 MHz**). Changing this setting mainly serves two purposes:

1. Performance (Computing Power)

If you are doing signal processing, video display, or complex mathematical calculations, you will choose the maximum frequency (**240 MHz**). The processor will process data much faster.

2. Autonomy (Energy Saving)

This is the most common use for lowering the frequency.

- **The higher the frequency** , the more power the chip consumes and the more heat it generates.
- **The lower the frequency** , the more you save your battery.

> If your project simply involves reading a temperature once a minute, running the processor at 240 MHz is a waste of energy. Switching to 80 MHz can significantly extend your battery life.

The consequences of a frequency change

Note that changing the frequency is not technically "free":

- **WiFi and Bluetooth:** On the ESP32, some wireless functions require a minimum frequency (often 80 MHz or higher) to work correctly. If you go too low, the WiFi connection will be lost.
- **Timing:** Although the IDE handles this most of the time, some timing calculations (such as very precise delays or communication protocols) can be disrupted if you change the frequency in the middle of a project.

Summary of typical settings:

Frequency	**Ideal use**
240 MHz	Active WiFi, Bluetooth, intensive computing, touchscreens.
160 MHz	Good compromise between power and fuel consumption.
80 MHz	Simple projects, battery-powered IoT sensors, occasional WiFi.
< 80 MHz	Very low power mode (Deep Sleep), no wireless.

Core Debug Level

The **Core Debug Level option** is a feature specific to powerful boards like the **ESP32** . It allows you to choose the amount of technical detail that the board sends to the **Serial Monitor** regarding its internal operation.

It's a bit like asking your car to either tell you "Everything is fine" or to list every single drop of gasoline injected into the engine.

What is the purpose of the different levels?

The ESP32's "Core" handles complex tasks in the background (WiFi connection, memory management, Bluetooth). If your code crashes for no apparent reason, increasing the debug level will allow you to see the exact error the system encountered.

Here are the available settings (from least talkative to most verbose):

- **None:** This is the default setting for production. The card remains silent, which saves memory and energy.
- **Error:** The card only speaks if something serious happens (a crash, a defective component).
- **Warning:** Displays errors, but also warning messages about suspicious behavior that has not yet caused the system to crash.
- **Info (Information):** Displays key steps, such as "WiFi connection successful" or "IP address obtained". Very useful for monitoring the normal process.
- **Debug:** Provides technical details about internal functions. Useful for advanced developers.
- **Verbose:** The card narrates *everything* it does, every millisecond. Warning: this slightly slows down execution and floods your Serial Monitor with text.

Why use it?

1. **Tracking down "Gurus" (Crashes):** The ESP32 often displays an error message called "Guru Meditation Error" when it crashes. With the debug level set to **Info** or **Debug** , you'll get more details about the reason for the crash (memory full, division by zero, etc.).
2. **Monitor the WiFi:** If you are unable to connect to your box, setting the level to **Info** will allow you to see if it is a password problem or a signal that is too weak.

Practical tip: Leave this option set to **None** or **Error** most of the time. Only use **Info** or **Verbose** when you are actively investigating the cause of a specific bug.

Erase All Flash Before Sketch Upload

"Erase All Flash Before Sketch Upload" option is a "deep cleanup" command for your ESP32.

By default, when you upload a new program (sketch) to your card, the IDE only replaces the memory area needed for the code. The rest of the Flash memory (where your WiFi settings, SPIFFS/LittleFS files, or saved data are stored) remains untouched.

Enabling this option instructs the IDE to **fully format** the chip before writing new code.

It is not necessary (nor recommended) to use it every time, as this slows down the upload process. Here are the cases where it becomes essential:

- **Erratic behavior:** Your code is correct, but the card is acting strangely, restarting in a loop, or remembering old WiFi settings that you no longer want.
- **File system change:** If you are switching from a project using **SPIFFS** to a project using **LittleFS** , it is best to erase all Flash to avoid partition conflicts.

- **After a severe crash:** If the memory has been corrupted due to a memory management bug, a complete wipe restores the card to its "factory state".
- **Sale or donation of the card:** To ensure that no sensitive data (WiFi password, API keys) remains stored in the recesses of the memory.

The consequences

Warning: As its name suggests, this operation is irreversible.

- All your **saved preferences** (via the Preferences.h library) will be deleted.
- All your **stored files** (images, web pages, logs) will be erased.
- The upload time will be a little longer because the IDE has to send a full erase command (chip_erase) before writing.

How do I activate it?

1. Go to the **Tools menu** .
2. Look for the line **Erase All Flash Before Sketch Upload** .
3. Select **Enabled** .
4. **Important:** Once your problem is solved, switch this option back to **Disabled** to save time on your next attempts.

Events Run On

The **Events Run On option** is an advanced setting, specific to the **ESP32 architecture** , which has the particularity of having **two cores** (Core 0 and Core 1) instead of just one.

This option allows you to choose which "brain" the system should handle **system events on** (mainly WiFi, Bluetooth and TCP/IP network tasks).

Why separate the tasks?

By default, the ESP32 distributes its workload as follows:

- **Core 1:** This is where your main Arduino code runs (the `setup()` and `loop() functions`).
- **Core 0:** This is generally where the "operating system" (FreeRTOS) handles complex wireless communications.

Events Run On option allows you to force event handling on one or the other.

The available options:

1. **Core 0 (Default/Recommended):** The system manages Wi-Fi and Bluetooth on core 0. This leaves core 1 completely free to execute your code without interruption. This is the most stable configuration for the majority of projects.
2. **Core 1:** The system handles everything on the same core as your code. This can be useful in very specific synchronization cases, but it's risky: if your code performs a calculation that takes too long (like a large `delay() loop`), it can prevent the WiFi from responding, causing a disconnection.

You will probably never need to change this setting, unless:

- **You are developing a critical real-time application:** If you need absolute precision on Core 1 (e.g., controlling a motor very precisely), you absolutely want WiFi events (Core 0) not to disrupt your timing.
- **Latency optimization:** In highly specialized network projects, moving event handling can sometimes reduce response time by a few microseconds.

Important note: If you change this setting and your ESP32 starts rebooting continuously or the WiFi becomes unstable, immediately revert to the default setting (**Core 0**).

Flash Frequency

The **Flash Frequency option** defines the communication speed between the ESP32 processor and its external Flash memory chip.

Unlike a computer where RAM and storage are separate, the ESP32 retrieves your program's instructions directly from its Flash chip for execution. The speed of this data transfer directly impacts the overall execution speed of your code.

In the Arduino IDE, you will generally have a choice between several frequencies:

- **40 MHz:** This is the safest and most stable setting. It's the standard speed that works with 100% of ESP32 modules on the market.
- **80 MHz:** This is the "performance" setting. Communication is twice as fast. Most modern modules (ESP32-WROOM, etc.) support this frequency very well.

Why choose one or the other?

1. Performance gain (80 MHz)

By switching to 80 MHz, your program loads into memory faster. If you have very large code or if you handle many files stored on the Flash memory (images for a screen, web server files), you will see an improvement in responsiveness.

2. Stability and Reliability (40 MHz)

If your card starts behaving strangely:

- The code crashes randomly.
- The error message `"Flash read err"` appears in the Serial Monitor.
- The upload is failing regularly. **So, switch back to 40 MHz.** Some low-quality Flash chips or excessively long cables on "homemade" modules cannot support high speeds.

Speed is only part of the equation. It often works in tandem with **Flash Mode** (such as QIO or DIO).

- **40 MHz + DIO:** Maximum security.
- **80 MHz + QIO:** Maximum performance (double the speed and 4 data wires instead of 2).

Practical tip: For most projects, you can leave this setting at **80 MHz** . If you notice that your ESP32 is overheating or restarting for no reason, try lowering it to **40 MHz** to diagnose the problem.

Flash Mode

Flash **Mode** defines how the ESP32's processor communicates with its Flash memory chip (where your program is stored). It's a "bandwidth" setting: it determines how many data wires are used simultaneously to read instructions.

It's a bit like comparing a road to one or more lanes: the more lanes there are, the faster the data flows.

In the Arduino IDE menu, you will generally find these four options:

1. QIO (Quad I/O) – Fastest

- **Operation:** Uses **4** data wires for addresses and for data.
- **Performance:** This is the highest performing mode.

- **Compatibility:** Requires that the Flash chip and wiring support Quad mode. If your ESP32 fails to boot after uploading in this mode, your hardware is not compatible.

2. QOUT (Quad Output)

- **Operation:** Uses **4 wires** only for data (addresses pass through a single wire).
- **Performance:** Slightly slower than QIO, but more stable on some chips.

3. DIO (Dual I/O) - The standard

- **Operation:** Uses **2** data wires for addresses and data.
- **Performance:** Average speed.
- **Usage:** This is the default mode for many ESP32 modules because it is very reliable.

4. DOUT (Dual Output) - The most compatible

- **Operation:** Uses **2 wires** only for data.
- **Performance:** The slowest of the four.
- **Usage:** Use as a last resort if no other method works. It is compatible with virtually all Flash chips on the market.

The choice depends on the exact model of your ESP32 module (the small metal square on your board):

- **For a standard ESP32-WROOM or WROVER: DIO** mode is the safest choice. Many support **QIO** , but the speed gain isn't always noticeable for simple programs.
- **If you need performance (graphics display, audio): Try QIO** mode . If the card restarts in a loop with a "Flash read err" error message, switch back to **DIO mode** .
- **Be careful with the pins used:** Quad mode (QIO/QOUT) uses specific pins (GPIO 6 to 11). If you try to use these pins for something else in your project, Quad mode will create a conflict and your program will crash.

In summary:

Fashion	**Number of data threads**	**Relative speed**	**Stability**
QIO	4	⭐⭐⭐⭐	Average
QOUT	4	⭐⭐⭐	Good
DIO	2	⭐⭐	Excellent
DOUBT	2	⭐	Maximum

Flash Size

The **Flash Size option** tells the Arduino IDE the total storage capacity of the memory chip present on your ESP32 module.

Unlike a computer where the system automatically detects the size of the hard drive, here the compiler needs to know this exact limit to organize the space (the "partitioning") before sending your program.

The ESP32's Flash memory contains everything that needs to survive a power outage:

1. **Your program** (the compiled code).
2. **The operating system** (FreeRTOS).

3. **User data** (LittleFS/SPIFFS files, WiFi settings).
4. **The OTA** (Over-The-Air) zone: a reserved area for updating the wireless map.

If you set the IDE to **2MB** when your card has **4MB** , you will only use half of the available capacity. Conversely, if you set it to **4MB** for a card that only has **2MB** , the upload will fail or the card will crash as soon as it tries to access a non-existent memory area.

Common sizes for the ESP32

- **4MB (4 Megabytes):** This is the absolute standard. The vast majority of commercially available **ESP32-WROOM-32 modules have 4MB.**
- **2MB:** Often found on miniature or very low-cost versions.
- **8MB / 16MB:** Found on high-end cards (like some M5Stack models or ESP32-S3) designed to store a lot of images or sounds.

How do I know the size of my card?

If you don't know the capacity of your card, you can use the **"Get Board Info" option** we saw earlier, but it doesn't always give the exact size of the Flash.

The most reliable method is to look at the Serial Monitor at startup or during upload. The `esptool tool` (used by the IDE) often displays a line like this:

```
Detected Flash size: 4MB
```

The size of the Flash memory is inextricably linked to the **Partition Scheme . For example, on a 4MB** card , you can choose to distribute the space as follows:

- **Default:** ~1.2MB for the code, 1.5MB for the files, the rest for updates.
- **No OTA (Large APP):** ~3MB for the code (if your program is huge), but you lose the ability to update the map via WiFi.

Tip: If you're unsure, leave this setting at **4MB** . This is the "universal" setting for 90% of ESP32 development boards (DevKit V1, NodeMCU-32S, etc.).

JTAG Adapter

JTAG Adapter option in the Arduino IDE for ESP32 is a cutting-edge setting that allows you to choose the hardware tool used for **real-time debugging** .

While the *Serial Monitor* only allows you to display text to understand what is happening, **JTAG** allows you to "freeze" the processor, inspect each variable and go through the code line by line while it is running.

The JTAG (*Joint Test Action Group*) is a standard physical interface used to test and debug microcontrollers. On an ESP32, this typically requires an external box (the adapter) connected between your computer and specific pins on the board.

What is the purpose of this option in the IDE?

In Arduino IDE 2.x, you have a "Start Debugging" button (the insect icon with a play arrow). For this button to work, the IDE needs to know what hardware is bridging the gap between your PC and the chip.

Here are the common choices you will find in the menu:

1. **Integrated JTAG (ESP32-S3 / C3):** Recent chips like the ESP32-S3 integrate their own JTAG converter. You don't need any additional hardware, just a USB cable.
2. **ESP-Prog:** This is Espressif's official tool. It's a small printed circuit board that connects to the GPIO pins of the ESP32.
3. **J-Link / FT2232:** Very fast universal professional adapters.

Why use it?

- **Freeze frame (Breakpoints):** You can tell the ESP32: "Stop as soon as you reach line 42". You can then check if your variables have the correct values.
- **Crash analysis:** If your card keeps restarting, JTAG allows you to see exactly which instruction caused the system error.
- **Speed:** For very large projects, JTAG allows code to be uploaded faster than via the classic serial port.

Why not always use it?

- **Complexity:** This requires configuring specific drivers (often with a utility like *Zadig*).
- **Occupied pins:** JTAG uses several GPIO pins (often 12, 13, 14, and 15). If you are using these pins for sensors or displays, you will not be able to use JTAG simultaneously.

Tip: If you're a beginner and don't have an ESP-Prog box or an S-series chip, leave this option as default. Debugging with `Serial.print()` remains the simplest and most widely used method.

Arduino Runs On

Arduino Runs On option is very similar to the *Events Run On option* we saw previously, but this time it concerns **your own code** (the "Sketch").

Since the ESP32 has two cores (Core 0 and Core 1), this option allows you to decide which brain will be responsible for executing your `setup()` and `loop() functions .`

On a standard ESP32, you will have the choice between:

1. **Core 1 (Default / Recommended):** This is the standard setting. Your code runs on core 1, while core 0 is reserved for "noble" system tasks (WiFi, Bluetooth, TCP/IP stack).
    - **Advantage:** Your program does not slow down the wireless connection, and vice versa.
2. **Core 0:** Your code is moved to the same core as the network functions.
    - **Usage:** This setting is very rarely used. It is sometimes done to completely free up Core 1 in order to manually launch ultra-priority tasks timed to the microsecond via FreeRTOS (the internal operating system).

The ESP32 uses a real-time operating system called **FreeRTOS** . Even if you feel like you are writing simple code, the IDE creates a "Task" in the background that contains your loop `()` .

- **If you choose Core 1:** If your code "crashes" or stops (for example with a poorly coded infinite loop), the WiFi on Core 0 can continue to work for a few moments before the watchdog *notices* the problem.
- **If you choose Core 0:** If your code is too large or contains excessively long `delay() functions` , you will overwork the processor. The Wi-Fi will no longer have enough processing time to maintain the connection, and your network card will constantly disconnect.

In summary:

Option	**Destination of your code**	**Use**
Core 1	Application Core	**99% of the time.** This is the stable configuration.
Core 0	Core System	Very advanced projects where the aim is to reverse the roles.

Partition Scheme

The **Partition Scheme option** is one of the most important settings for the ESP32. It defines how the total Flash memory on your card (e.g., 4MB) is divided into virtual "drawers".

Since the ESP32 has to store your code, system data, and sometimes files (images, web pages), you need to tell it in advance what size to allocate to each section.

over-the-air (**OTA)** updates . This means that the IDE reserves two identical areas for your program: one for the current code and an empty one to receive the update.

As a result, out of 4 MB, you can only use about 1.2 MB for your actual code. If your program exceeds this size, you will receive a *"Sketch too big"* error . This is where the partition change comes in.

The most common options

Here are the settings you will usually find in the **Tools > Partition Scheme menu** :

- **Default (4MB with OTA):**
  - Code: ~1.2 MB
  - Files (SPIFFS/LittleFS): ~1.5 MB
  - *Usage:* The standard for classic projects with WiFi upgrades.
- **No. OTA (2MB APP / 2MB SPIFFS):**
  - Code: 2 MB
  - Files: 2 MB
  - *Usage:* For projects with many images or data, but without wireless updates.
- **Huge APP (3MB No OTA):**
  - Code: **3 MB**
  - Files: Very few (approximately 190 KB).
  - *Use:* Essential for huge programs (simple artificial intelligence, heavy Bluetooth, or complex graphical interfaces).
- **Minimal SPIFFS:**
  - It gives almost all the space to the code and reduces the file area to the bare minimum.

What is the impact on your code?

The choice of partition modifies the memory structure **during upload** .

Warning: If you change your partition scheme, the location of your data will change. This can erase files you had stored in SPIFFS or LittleFS. Therefore, it is recommended that you choose your scheme at the beginning of your project and not change it again.

If you are using ESP32forth with many options, choose the second option:

Tools → **Partition Scheme** :

- "No OTA (2 MB APP / 2 MB SPIFFS)" = 2 MB for your code
- " **Huge APP (3 MB No OTA)** " = 3 MB for your ESP32forth code

Even though it reduces the FLASH memory space for SPIFFS, it leaves enough room for FORTH source code.

How to choose?

1. **Is your code too large?** Switch to **"Huge APP"** .

2. **Do you need to store a lot of files (Sounds, HTML)?** Switch to **"No OTA (Large SPIFFS)"** .
3. **Do you want to be able to update your map via WiFi? Stay in "with OTA"** mode .

View the compilation report

A good habit to get into: monitoring the size of your program helps avoid a lot of frustration when uploading.

Here's how to read and interpret this information in the **Arduino IDE** .

Each time you click **Check** (the checkmark icon) or **Upload** , the IDE displays a summary in the black console at the bottom of the window.

This is what it looks like:

```
The sketch uses 262,144 bytes (20%) of the program storage space. The maximum is
1,310,720 bytes. Global variables use 15,320 bytes (4%) of the dynamic memory...
```

- **Storage space (Flash):** This is the size of your code "drawer" as defined by your **Partition Scheme** . If you reach 90-95%, it's time to move to a larger partition (like *Huge APP*).
- **Dynamic memory (RAM):** This is the random access memory. If it is too high (close to 100%), your ESP32 may freeze or restart unpredictably during operation.

PSRAM

PSRAM (Pseudo-Static Random Access Memory) is an extension of the ESP32's random access memory (RAM) **.**

To understand its usefulness, imagine the ESP32's internal RAM as a small, very fast workspace. If you have a huge project (image processing, audio, artificial intelligence), this workspace becomes too small. The PSRAM is like a **large architect's table added next to it** : it's slightly slower than the main workspace, but it offers a huge amount of extra space.

The standard ESP32 has approximately **520 KB** of internal RAM. This is a lot for simple electronics, but very little for:

- **Video/Photo:** Store a single high-resolution image from a camera (ESP32-CAM).
- **Audio:** Create a buffer to play streaming music without interruption.
- **Display:** Managing color touchscreens with fluid graphical interfaces (LVGL).
- **The Web:** Running a complex server with a lot of dynamic data.

With PSRAM, you can go from 520KB to **2MB, 4MB or even 8MB** of available memory.

If your board has a PSRAM chip (like the **WROVER** or **ESP32-S3-WROOM-1 models**), it is not necessarily active by default.

1. Go to the **Tools menu** .
2. Look for the **PSRAM option** .
3. Select **Enabled** (or "OPI PSRAM" / "QSPI PSRAM" depending on your model).

Limits to be aware of

- **Speed:** PSRAM is connected via a serial bus (SPI). It is therefore slower than internal RAM. It is used to store large data, but not ultra-fast mathematical calculations.
- **Consumption:** It consumes a little more energy, which should be monitored when on battery power.

- **Pin conflict:** On some older ESP32 models, using PSRAM "steals" some GPIO pins (often 16 and 17). Check your pinout carefully.

How can you tell if you have it? Look at the label on the metal module of your card. If it says **WROVER** , you have PSRAM. If it says **WROOM** , you generally don't (except for special versions).

Upload Speed

The **Upload Speed option** defines the speed at which the Arduino IDE sends your compiled program to the ESP32 board via the USB cable.

This speed is measured in **bauds** (bits per second). It's the "flow rate" of the connection between your computer and your card during the programming phase.

Unlike Arduino Uno boards, which are limited to 115,200 baud, the ESP32 is a real speed demon. You'll generally find these options:

- **115,200:** The "safe" speed. It's slow but very reliable.
- **460 800:** An excellent compromise, very widely used.
- **921,600:** The current maximum speed. That's almost 10 times faster than the standard Arduino.

Why choose one speed over another?

1. Time saving

If you are developing a large project (with many libraries such as WiFi or Bluetooth), your binary file may weigh several megabytes.

- At **115,200** , the upload may take 30 to 40 seconds.
- At **921,600** , it only takes 5 to 10 seconds. Over a workday with dozens of tests, the difference is enormous.

2. Reliability (Stability)

If you set the speed too high, you risk transfer errors (`Packet content mismatch` or `Failed to connect`). This usually happens because of:

- **A poor quality USB cable:** Too long or poorly shielded.
- **A cheap USB-to-Serial converter:** Some "clone" cards use chips that do not support high speeds well.
- **Interference:** If your setup includes noisy motors or power supplies nearby.

Does this change the speed of my program?

No. That's a common misconception.

- **Upload Speed:** Only concerns the transfer of the code (the programming phase).
- **Baud Rate (Serial.begin):** This is the communication speed while the program is running (for the Serial Monitor).

You can easily upload to **921 600** and communicate with the Serial Monitor at **115 200**

Zigbee Mode

The **Zigbee Mode option** appeared with newer generations of chips, specifically the **ESP32-H2** and **ESP32-C6** . Unlike the classic ESP32, which only supports WiFi and Bluetooth, these chips integrate a radio compatible with the **IEEE 802.15.4 standard** .

This option in the Arduino IDE allows you to define the role your board will play within a smart home network (home automation).

Zigbee is the ideal alternative to WiFi for connected objects (IoT) because:

- **Ultra-low power consumption:** A sensor can operate for years on a button cell battery.
- **Mesh Network:** Each device plugged into mains power can repeat the signal to extend the range.
- **Home automation standard:** This is the language used by Philips Hue, IKEA TRÅDFRI or Xiaomi sensors.

The different modes available

In the menu, you will generally have a choice between three types of devices:

1. Zigbee Coordinator

It is the "brain" of the network.

- **Role:** It creates the network, authorizes new devices to connect, and manages security.
- **Note:** There can only be one coordinator per Zigbee network (often your home automation box or a dedicated USB key).

2. Zigbee Router

It is an intermediate device that remains powered at all times (e.g., a light bulb or a smart plug).

- **Role:** It performs its tasks (turning on the light) while relaying messages from other sensors to the coordinator.

3. Zigbee End Device

This is the device that sleeps most of the time to save its battery (e.g., a motion or temperature sensor).

- **Role:** It wakes up, sends its data, and goes back to sleep. It never relays messages from others.

Why is this setting in the IDE?

The ESP32 must configure its software stack differently depending on the chosen mode. For example, if you choose **End Device , the IDE will include Deep** *Sleep* management functions . If you choose **Router** , it will enable network routing functions.

Matter and Zigbee

Note that these chips also form the basis of the new **Matter standard** . Zigbee mode is often the first step in creating bridges between your old Zigbee devices and your new Matter devices.

Warning: If you have a classic ESP32 (WROOM-32), this option will not appear or will have no effect, because the radio hardware is not compatible.

Resources

Official ARDUINO IDE website

https://www.arduino.cc/en/software/