Module 3

Getting Started with the ESP32 With WOWKI Platform

The ESP32 is a series of low-cost and low-power System on a Chip (SoC) microcontrollers developed by Espressif that include Wi-Fi and Bluetooth wireless capabilities and dual-core processor. If you're familiar with the ESP8266, the ESP32 is its successor, loaded with lots of new features.

Table of Contents

Introducing the ESP32

First, to get started, **what is an ESP32**? The ESP32 is a series of chip microcontrollers developed by Espressif.

Why are they so popular? Mainly because of the following features:

- **Low-cost**: you can get an ESP32 starting at $6, which makes it easily accessible to the general public;
- **Low-power**: the ESP32 consumes very little power compared with other microcontrollers, and it supports low-power mode states like deep sleep to save power;
- **Wi-Fi capabilities**: the ESP32 can easily connect to a Wi-Fi network to connect to the internet (station mode), or create its own Wi-Fi wireless network (access point mode) so other devices can connect to it—this is

essential for IoT and Home Automation projects—you can have multiple devices communicating with each other using their Wi-Fi capabilities;

- **Bluetooth**: the ESP32 supports Bluetooth classic and Bluetooth Low Energy (BLE)—which is useful for a wide variety of IoT applications;
- Dual-core: most ESP32 are dual-core— they come with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1.
- Rich peripheral input/output interface—the ESP32 supports a wide variety of input (read data from the outside world) and output (to send commands/signals to the outside world) peripherals like capacitive touch, ADCs, DACs, UART, SPI, I2C, PWM, and much more.
- Compatible with the Arduino "programming language": those that are already familiar with programming the Arduino board, you'll be happy to know that they can program the ESP32 in the Arduino style.
- Compatible with MicroPython: you can program the ESP32 with MicroPython firmware, which is a re-implementation of Python 3 targeted for microcontrollers and embedded systems.

ESP32 Specifications

If you want to get a bit more technical and specific, you can take a look at the following detailed specifications of the ESP32 (source: http://esp32.net/)—for more details, check the datasheet):

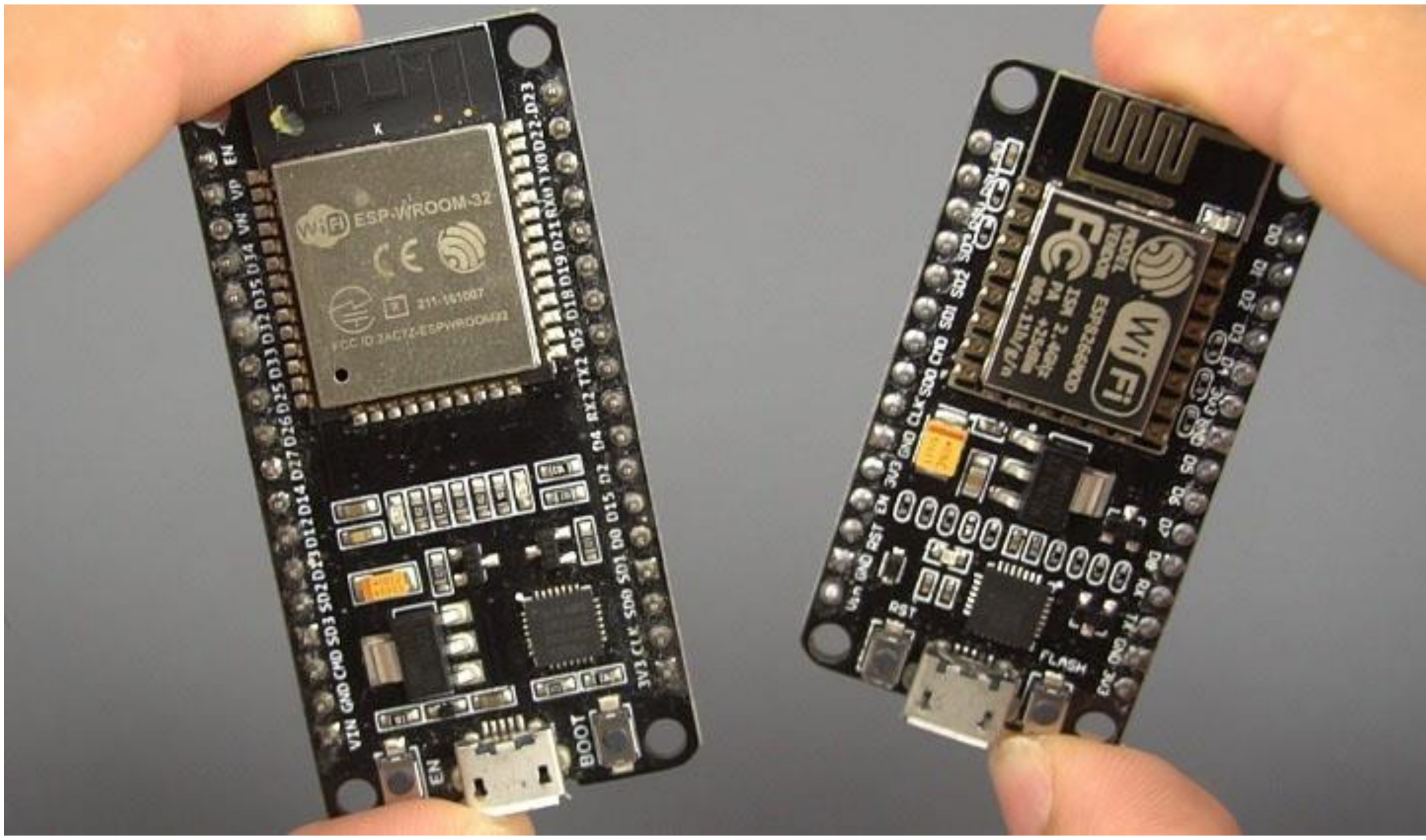
ESP32 module: ESP-WROOM-32

- **Wireless connectivityWiFi:** 150.0 Mbps data rate with HT40
  - **Bluetooth:** BLE (Bluetooth Low Energy) and Bluetooth Classic
  - **Processor:** Tensilica Xtensa Dual-Core 32-bit LX6 microprocessor, running at 160 or 240 MHz
- **Memory**:

    - **ROM:** 448 KB (for booting and core functions)
    - **SRAM:** 520 KB (for data and instructions)
    - **RTC fast SRAM**: 8 KB (for data storage and main CPU during RTC Boot from the deep-sleep mode)
    - **RTC slow SRAM**: 8KB (for co-processor accessing during deep-sleep mode)
    - **eFuse**: 1 Kbit (of which 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID)
    - **Embedded flash**: flash connected internally via IO16, IO17, SD_CMD, SD_CLK, SD_DATA_0 and SD_DATA_1 on ESP32-D2WD and ESP32-PICO-D4.
        - 0 MiB (ESP32-D0WDQ6, ESP32-D0WD, and ESP32-S0WD chips)
        - 2 MiB (ESP32-D2WD chip)
        - 4 MiB (ESP32-PICO-D4 SiP module)
- **Low Power:** ensures that you can still use ADC conversions, for example, during deep sleep.
- **Peripheral Input/Output:**
    - peripheral interface with DMA that includes capacitive touch
    - ADCs (Analog-to-Digital Converter)
    - DACs (Digital-to-Analog Converter)
    - I^2C (Inter-Integrated Circuit)
    - UART (Universal Asynchronous Receiver/Transmitter)
    - SPI (Serial Peripheral Interface)
    - I^2S (Integrated Interchip Sound)
    - RMII (Reduced Media-Independent Interface)
    - PWM (Pulse-Width Modulation)

- **Security:** hardware accelerators for AES and SSL/TLS

- ## Main Differences Between ESP32 and ESP8266



Previously, we mentioned that the ESP32 is the ESP8266 successor. **What are the main differences between ESP32 and ESP8266 boards?**
The ESP32 adds an extra CPU core, faster Wi-Fi, more GPIOs, and supports Bluetooth 4.2 and Bluetooth low energy. Additionally, the ESP32 comes with touch-sensitive pins that can be used to wake up the ESP32 from deep sleep, and built-in hall effect sensor.
Both boards are cheap, but the ESP32 costs slightly more. While the ESP32 can cost around $6 to $12, the ESP8266 can cost $4 to $6 (but it really depends on where you get them and what model you're buying).

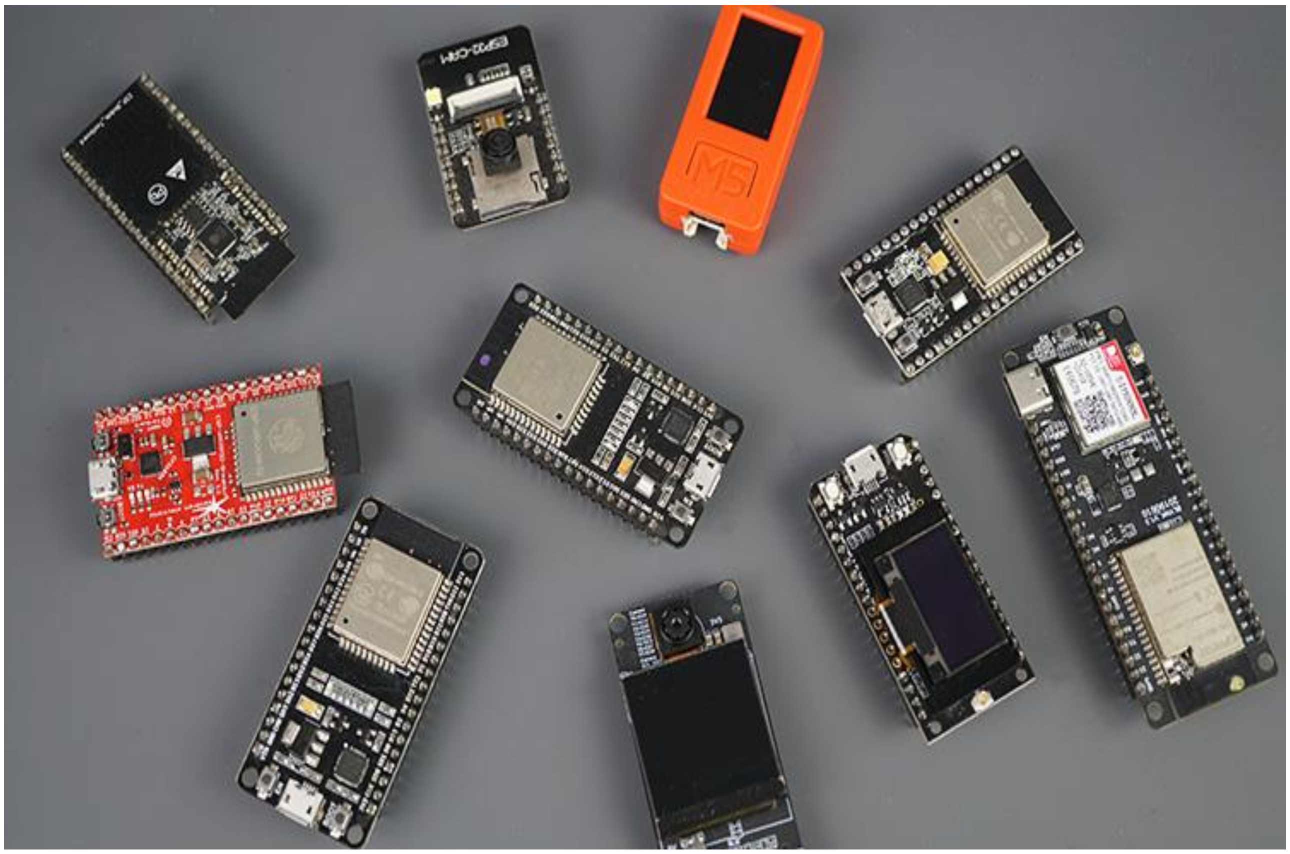
So, in summary:

- The ESP32 is faster than the ESP8266;
- The ESP32 comes with more GPIOs with multiple functions;
- The ESP32 supports analog measurements on 18 channels (analog-enabled pins) versus just one 10-bit ADC pin on the ESP8266;
- The ESP32 supports Bluetooth while the ESP8266 doesn't;
- The ESP32 is dual-core (most models), and the ESP8266 is single core;
- The ESP32 is a bit more expensive than the ESP8266.

For a more detailed analysis of the differences between those boards, we recommend reading the following article: ESP32 vs ESP8266 – Pros and Cons.

ESP32 Development Boards

ESP32 refers to the bare ESP32 chip. However, the "ESP32" term is also used to refer to ESP32 development boards. Using ESP32 bare chips is not easy or practical, especially when learning, testing, and prototyping. Most of the time, you'll want to use an ESP32 development board.



These development boards come with all the needed circuitry to power and program the chip, connect it to your computer, pins to connect peripherals, built-in power and control LEDs, an antenna for wi-fi signal, and other useful features. Others even come with extra hardware like specific sensors or modules, displays, or a camera in the case of the ESP32-CAM.

How to Choose an ESP32 Development Board?

Once you start searching for ESP32 boards online, you'll find there is a wide variety of boards from different vendors. While they all work in a similar way, some boards may be more suitable for some projects than others. When looking for an ESP32 development board there are several aspects you need to take into account:

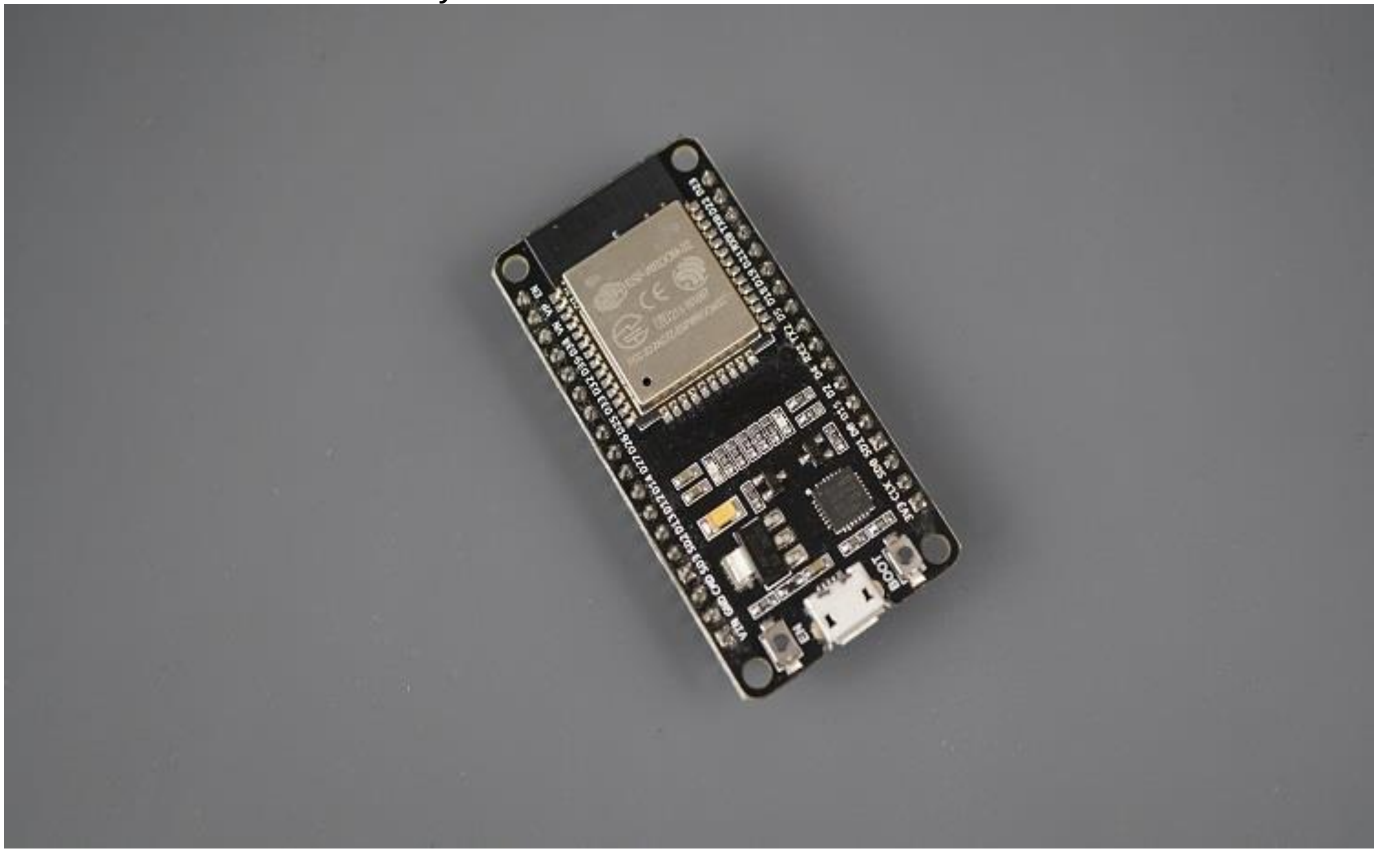
- **USB-to-UART interface and voltage regulator circuit**. Most full-featured development boards have these two features. This is important to easily connect the ESP32 to your computer to upload code and apply power.
- **BOOT and RESET/EN buttons** to put the board in flashing mode or reset (restart) the board. Some boards don't have the BOOT button. Usually, these boards go into flashing mode automatically.
- **Pin configuration and the number of pins.** To properly use the ESP32 in your projects, you need to have access to the board pinout (like a map that shows which pin corresponds to which GPIO and its features). So make sure you have access to the pinout of the board you're getting. Otherwise, you may end up using the ESP32 incorrectly.

- **Antenna connector**. Most boards come with an onboard antenna for Wi-Fi signal. Some boards come with an antenna connector to optionally connect an external antenna. Adding an external antenna increases your Wi-Fi range.
- **Battery connector**. If you want to power your ESP32 using batteries, there are development boards that come with connectors for LiPo batteries—this can be handier. You can also power a "regular" ESP32 with batteries through the power pins.
- **Extra hardware features**. There are ESP32 development boards with extra hardware features. For example, some may come with a built-in OLED display, a LoRa module, a SIM800 module (for GSM and GPRS), a battery holder, a camera, or others.

What is the best ESP32 development board for beginners?

For beginners, we recommend an ESP32 board with a vast selection of available GPIOs, and without any extra hardware features. It's also important that it comes with voltage regular and USB input for power and upload code.

In most of our ESP32 projects, we use the ESP32 DEVKIT DOIT board, and that's the one we recommend for beginners. There are different versions of this board with a different number of available pins (30, 36, and 38)—all boards work in a similar way.

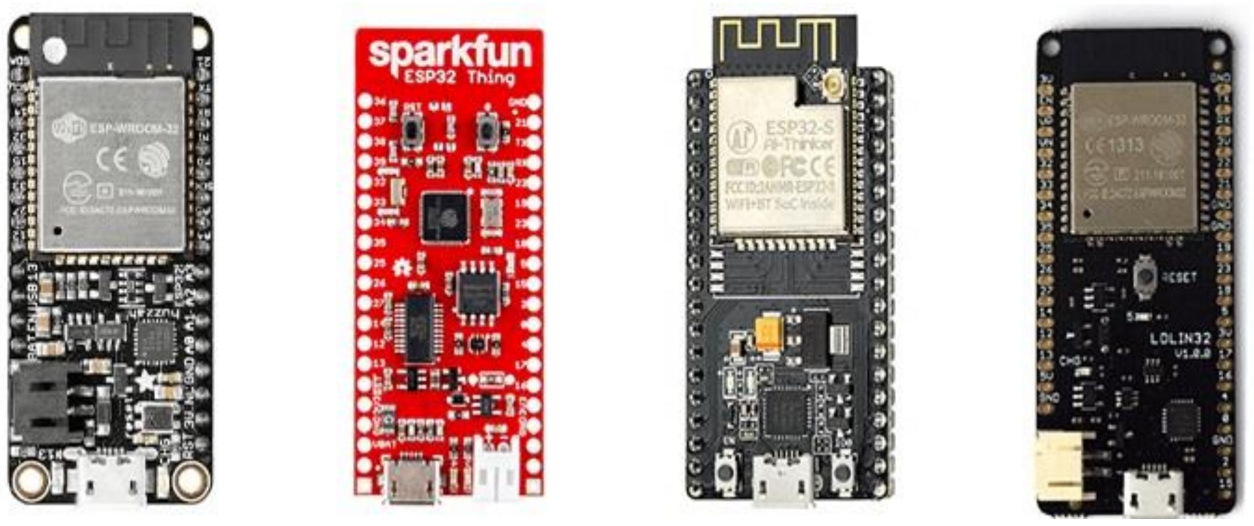


Where to Buy?

You can check the following link to find the ESP32 DEVKIT DOIT board in different stores:

- ESP32 DEVKIT DOIT board

Other similar boards with the features mentioned previously may also be a good option like the Adafruit ESP32 Feather, Sparkfun ESP32 Thing, NodeMCU-32S, Wemos LoLin32, etc.



ESP32 DEVKIT DOIT

In this article, we’ll be using the ESP32 DEVKIT DOIT board as a reference. If you have a different board, don’t worry. The information on this page is also compatible with other ESP32 development boards.

The picture below shows the ESP32 DEVKIT DOIT V1 board, version with 36 GPIO pins.

Number of cores	2 (dual core)
Wi-Fi	2.4 GHz up to 150 Mbits/s
Bluetooth	BLE (Bluetooth Low Energy) and legacy Bluetooth
Architecture	32 bits
Clock frequency	Up to 240 MHz
RAM	512 KB
Pins	30, 36, or 38 (depending on the model)
Peripherals	Capacitive touch, ADC (analog to digital converter), DAC (digital to analog converter), I2C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Netwokr), SPI (Serial Peripheral Interface), I2S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.
Built-in	RESET and BOOT buttons

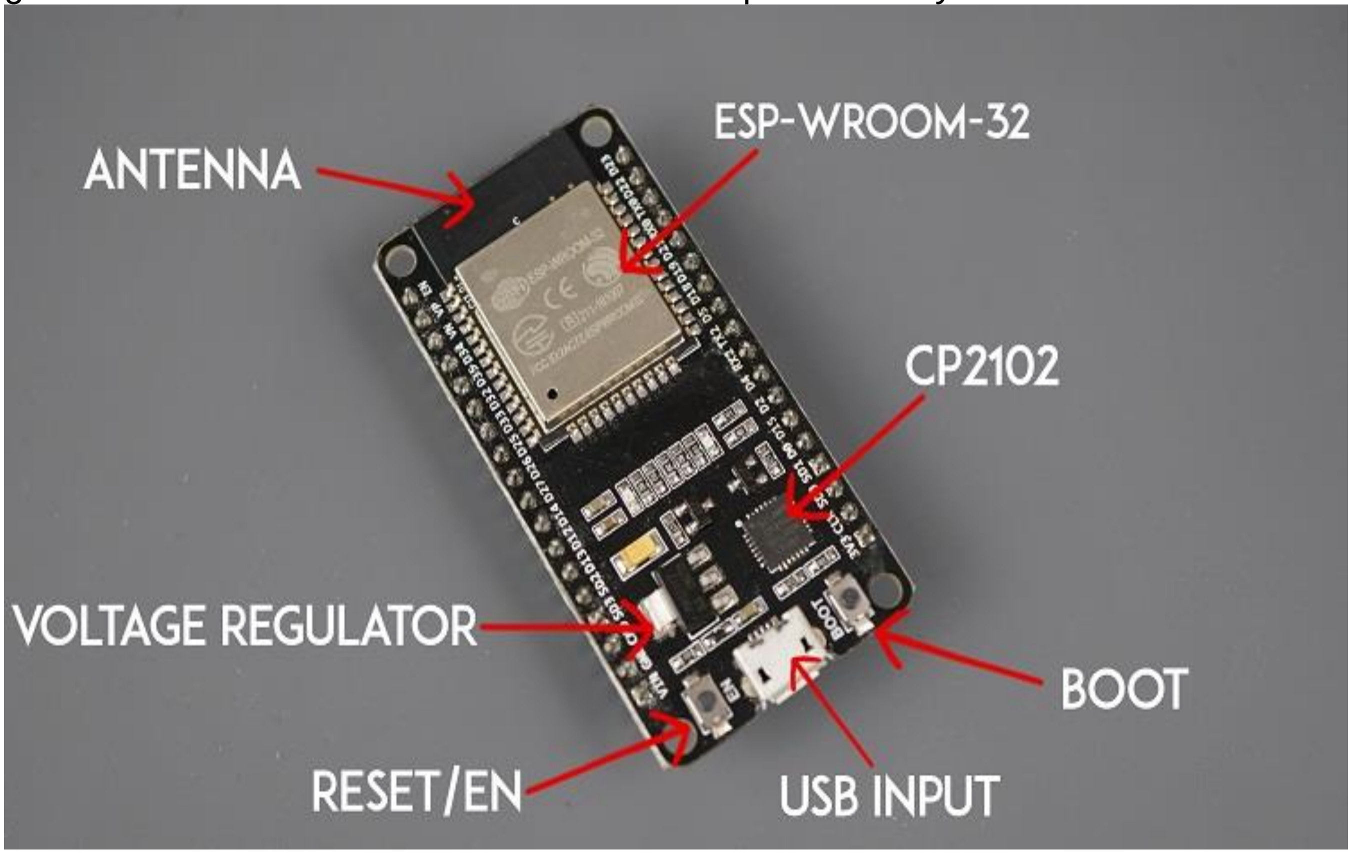
buttons	
Built-in LEDs	built-in blue LED connected to GPIO2; built-in red LED that shows the board is being powered
USB to UART bridge	CP2102

Specifications – ESP32 DEVKIT V1 DOIT

The following table shows a summary of the ESP32 DEVKIT V1 DOIT board features and specifications:

This particular ESP32 board comes with 36 pins, 18 on each side. The number of available GPIOs depends on your board model.

To learn more about the ESP32 GPIOs, read our GPIO reference guide: ESP32 Pinout Reference: Which GPIO pins should you use?

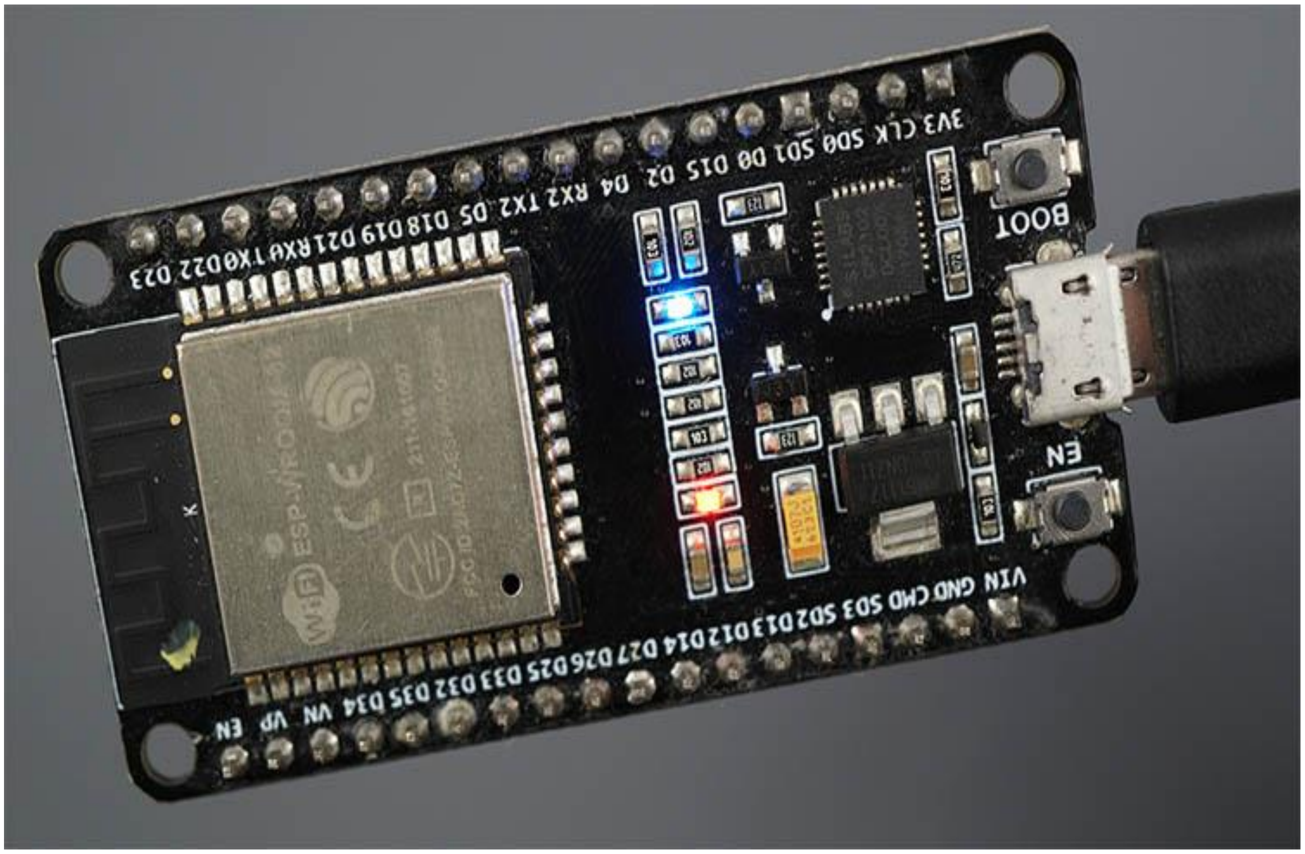


It comes with a microUSB interface that you can use to connect the board to your computer to upload code or apply power.

It uses the CP2102 chip (USB to UART) to communicate with your computer via a COM port using a serial interface. Another popular chip is the CH340. Check what’s the USB to UART chip converter on your board because you’ll need to install the required drivers so that your computer can communicate with the board (more information about this later in this guide).

This board also comes with a RESET button (may be labeled EN) to restart the board and a BOOT button to put the board in flashing mode (available to receive code). Note that some boards may not have a BOOT button.

It also comes with a built-in blue LED that is internally connected to GPIO 2. This LED is useful for debugging to give some sort of visual physical output. There’s also a red LED that lights up when you provide power to the board.


3V3 CLK SD0 SD1 D0 D15 D2 D4 RX2 TX2 D5 D18 D19 D21 RX0 TX0 D22 D23
BOOT
EN
VIN GND CMD SD3 SD2 D13 D12 D14 D27 D26 D25 D33 D32 D35 D34 VN VP EN

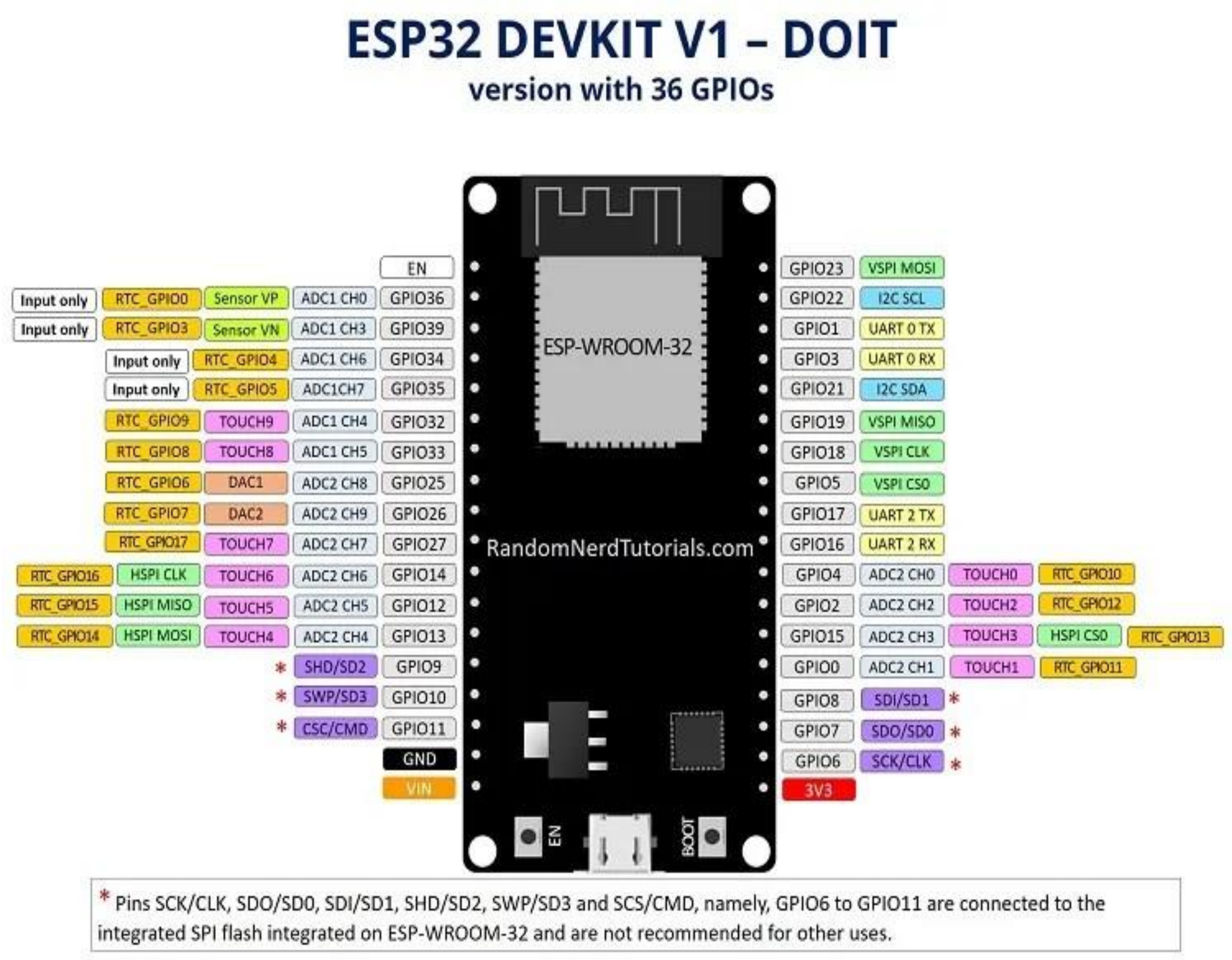
ESP32 GPIOs Pinout Guide

The ESP32 chip comes with 48 pins with multiple functions. Not all pins are exposed in all ESP32 development boards, and some pins should not be used. The ESP32 DEVKIT V1 DOIT board usually comes with 36 exposed GPIOs that you can use to connect peripherals.

Power Pins
Usually, all boards come with power pins: 3V3, GND, and VIN. You can use these pins to power the board (if you're not providing power through the USB port), or to get power for other peripherals (if you're powering the board using the USB port).

General Purpose Input Output Pins (GPIOS)
Almost all GPIOs have a number assigned and that's how you should refer to them—by their number.

With the ESP32 you can decide which pins are UART, I2C, or SPI – you just need to set that on the code. This is possible due to the ESP32 chip's multiplexing feature that allows to assign multiple functions to the same pin. If you don't set them on the code, the pins will be configured by default as shown in the figure below (the pin location can change depending on the manufacturer). Additionally, there are pins with specific features that make them suitable or not for a particular project.


ESP32 DEVKIT V1 – DOIT
version with 36 GPIOs
ESP-WROOM-32
RandomNerdTutorials.com
EN
Input only RTC_GPIO0 Sensor VP ADC1 CH0 GPIO36
Input only RTC_GPIO3 Sensor VN ADC1 CH3 GPIO39
Input only RTC_GPIO4 ADC1 CH6 GPIO34
Input only RTC_GPIO5 ADC1CH7 GPIO35
RTC_GPIO9 TOUCH9 ADC1 CH4 GPIO32
RTC_GPIO8 TOUCH8 ADC1 CH5 GPIO33
RTC_GPIO6 DAC1 ADC2 CH8 GPIO25
RTC_GPIO7 DAC2 ADC2 CH9 GPIO26
RTC_GPIO17 TOUCH7 ADC2 CH7 GPIO27
RTC_GPIO16 HSPI CLK TOUCH6 ADC2 CH6 GPIO14
RTC_GPIO15 HSPI MISO TOUCH5 ADC2 CH5 GPIO12
RTC_GPIO14 HSPI MOSI TOUCH4 ADC2 CH4 GPIO13
* SHD/SD2 GPIO9
* SWP/SD3 GPIO10
* CSC/CMD GPIO11
GND
VIN
GPIO23 VSPI MOSI
GPIO22 I2C SCL
GPIO1 UART 0 TX
GPIO3 UART 0 RX
GPIO21 I2C SDA
GPIO19 VSPI MISO
GPIO18 VSPI CLK
GPIO5 VSPI CS0
GPIO17 UART 2 TX
GPIO16 UART 2 RX
GPIO4 ADC2 CH0 TOUCH0 RTC_GPIO10
GPIO2 ADC2 CH2 TOUCH2 RTC_GPIO12
GPIO15 ADC2 CH3 TOUCH3 HSPI CS0 RTC_GPIO13
GPIO0 ADC2 CH1 TOUCH1 RTC_GPIO11
GPIO8 SDI/SD1 *
GPIO7 SDO/SD0 *
GPIO6 SCK/CLK *
3V3
EN
BOOT

* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and SCS/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

We have a detailed guide dedicated to the ESP32 GPIOs that we recommend you read: ESP32 Pinout Reference Guide. It shows how to use the ESP32 GPIOs and explains what are the best GPIOs to use depending on your project.
The placement of the GPIOs might be different depending on your board model. However, usually, each specific GPIO works in the same way regardless of the development board you're using (with some exceptions). For example, regardless of the board, usually GPIO5 is always the VSPI CS0 pin, GPIO 23 always corresponds to VSPI MOSI for SPI communication, etc.

How to Program the ESP32?

The ESP32 can be programmed using different firmware and programming languages. You can use:

- Arduino C/C++ using the Arduino core for the ESP32
- Espressif IDF (IoT Development Framework)
- Micropython
- JavaScript
- LUA
- …

Our preferred method to program the ESP32 is with C/C++ “Arduino programming language”. We also have some guides and tutorials using MicroPython firmware.
Throughout this guide, we’ll cover programming the ESP32 using the Arduino core for the ESP32 board. If you prefer using MicroPython, please refer to this guide: Getting Started with MicroPython on ESP32.

Programming ESP32 with Arduino IDE

To program your boards, you need an IDE to write your code. For beginners, we recommend using Arduino IDE. While it’s not the best IDE, it works well and is simple and intuitive to use for beginners. After getting familiar with Arduino IDE and you start creating more complex projects, you may find it useful to use VS Code with the Platformio extension instead.
If you’re just getting started with the ESP32, start with Arduino IDE. At the time of writing this tutorial, **we recommend using the legacy version**

(1.8.19) with the ESP32. While version 2 works well with Arduino, there are still some bugs and some features that are not supported yet for the ESP32.

Installing Arduino IDE

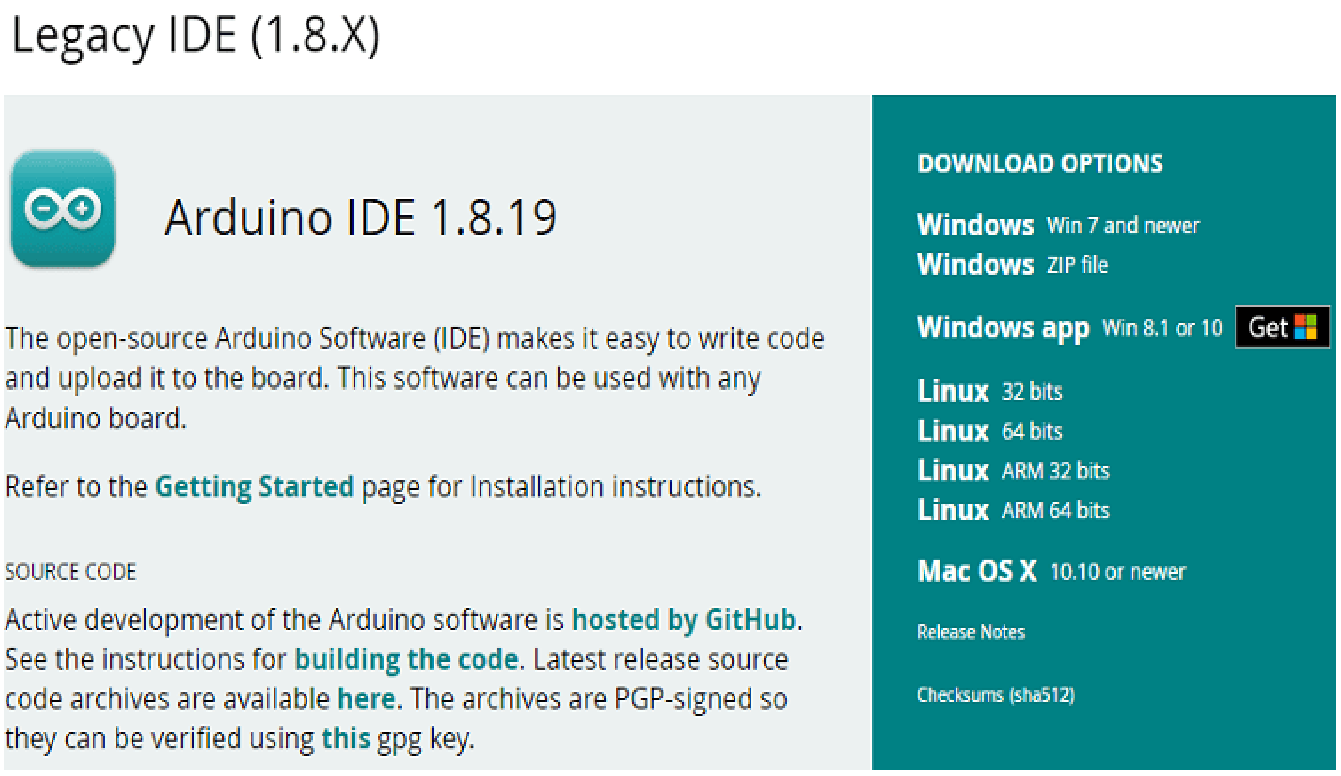
To run Arduino IDE, you need JAVA installed on your computer. If you don't, go to the following website to download and install the latest version: http://java.com/download.

Downloading Arduino IDE

To download the Arduino IDE, visit the following URL:

- https://www.arduino.cc/en/Main/Software

Don't install the 2.0 version. At the time of writing this tutorial, **we recommend using the legacy version (1.8.19)** with the ESP32. While version 2 works well with Arduino, there are still some bugs and some features that are not supported yet for the ESP32.
Scroll down until you find the legacy version section.



Legacy IDE (1.8.X)

Arduino IDE 1.8.19

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board.

Refer to the Getting Started page for Installation instructions.

SOURCE CODE

Active development of the Arduino software is hosted by GitHub. See the instructions for building the code. Latest release source code archives are available here. The archives are PGP-signed so they can be verified using this gpg key.

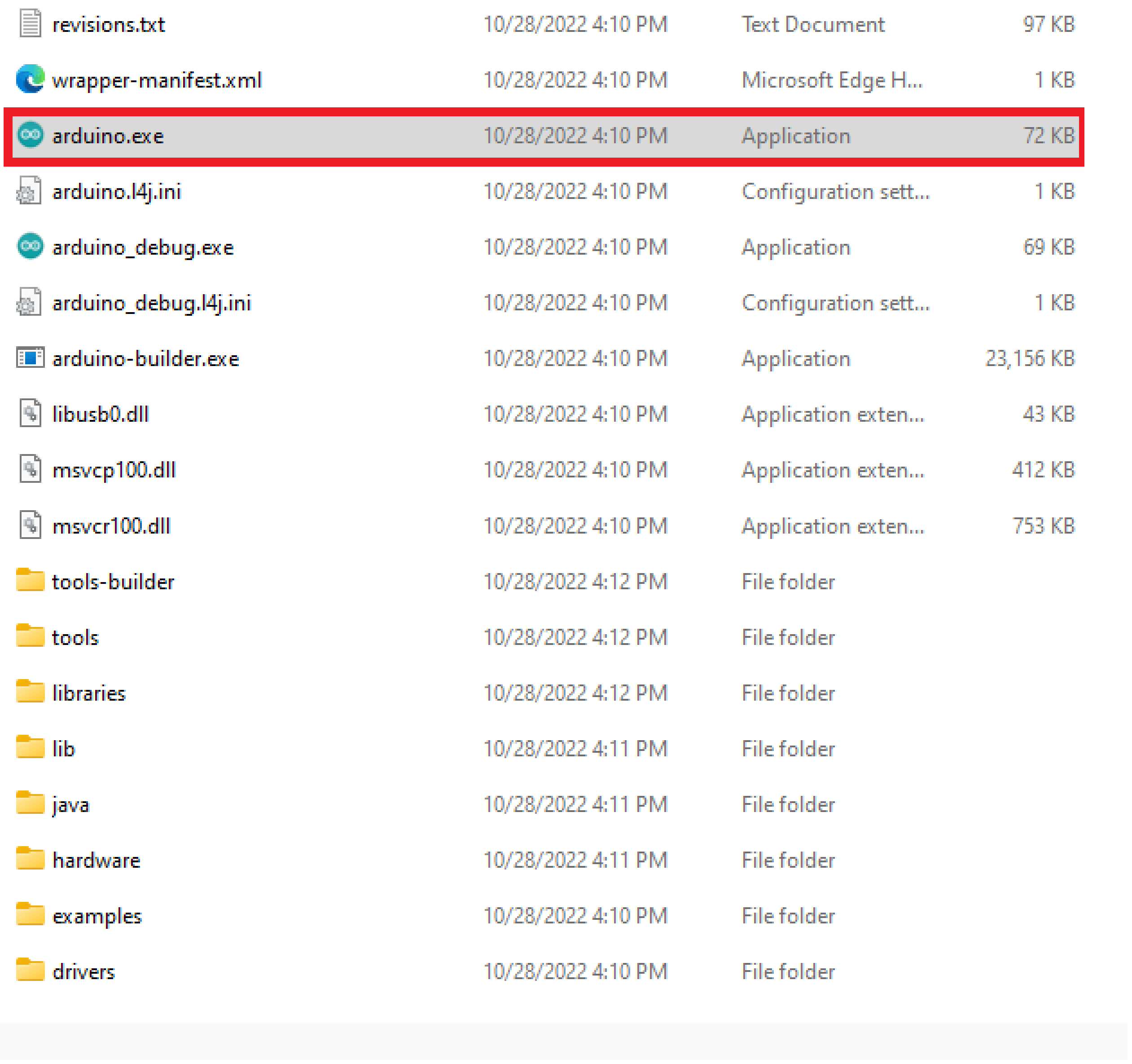
DOWNLOAD OPTIONS

Windows Win 7 and newer
Windows ZIP file
Windows app Win 8.1 or 10 Get
Linux 32 bits
Linux 64 bits
Linux ARM 32 bits
Linux ARM 64 bits
Mac OS X 10.10 or newer
Release Notes
Checksums (sha512)

Select your operating system and download the software. For Windows, we recommend downloading the “**Windows ZIP file**“.

Running Arduino IDE

Grab the folder you’ve just downloaded and unzip it. Run the executable file called arduino.exe (highlighted below).



revisions.txt	10/28/2022 4:10 PM	Text Document	97 KB
wrapper-manifest.xml	10/28/2022 4:10 PM	Microsoft Edge H...	1 KB
arduino.exe	10/28/2022 4:10 PM	Application	72 KB
arduino.l4j.ini	10/28/2022 4:10 PM	Configuration sett...	1 KB
arduino_debug.exe	10/28/2022 4:10 PM	Application	69 KB
arduino_debug.l4j.ini	10/28/2022 4:10 PM	Configuration sett...	1 KB
arduino-builder.exe	10/28/2022 4:10 PM	Application	23,156 KB
libusb0.dll	10/28/2022 4:10 PM	Application exten...	43 KB
msvcp100.dll	10/28/2022 4:10 PM	Application exten...	412 KB
msvcr100.dll	10/28/2022 4:10 PM	Application exten...	753 KB
tools-builder	10/28/2022 4:12 PM	File folder	
tools	10/28/2022 4:12 PM	File folder	
libraries	10/28/2022 4:12 PM	File folder	
lib	10/28/2022 4:11 PM	File folder	
java	10/28/2022 4:11 PM	File folder	
hardware	10/28/2022 4:11 PM	File folder	
examples	10/28/2022 4:10 PM	File folder	
drivers	10/28/2022 4:10 PM	File folder	

Simulation Features

Peripheral	ESP32	S2	S3	C3	C6	Notes
Processor core(s)	✔	✔	✔	✔	✔	
GPIO	✔	✔	✔	✔	✔	Interrupts supported
IOMUX	✔	✔	✔	✔	✔	
PSRAM	✔	✔	✔	—	✕	4MB of external SRAM *
UART	✔	✔	✔	✔	✔	
USB	—	✔	✔	✕	✕	Support for UART over USB (CDC)
I2C	✔	✔	✔	✔	✔	Master only. 10-bit addressing not supported.
I2S	✕	✕	✕	✕	✕	Open for voting
SPI	✔	✔	✔	✔	✔	
TWAI	✕	✕	✕	✕	✕	
RMT	□	□	□	□	□	Transmit-only, use to control NeoPixels
LEDC PWM	✔	✔	✔	✔	✔	Used by analogWrite(), Servo, Buzzer, etc.
MCPWM	✕	—	✕	—	✕	
DMA	□	□	✕	✕	✕	
WiFi	✔	✔	✔	✔	✔	See the ESP32 WiFi Guide
Bluetooth	✕	—	✕	✕	✕	Open for voting
Timers	□	✔	✔	✔	✔	
Watchdog	✕	✕	✕	✕	✕	
RTC	□	□	□	□	□	Only RTC Pull-up / Pull-down resistors
ADC	✔	✔	✔	✔	✔	Note: analogRead() returns values up to 4095
RNG	✔	✔	✔	✔	✔	Random Number Generator
AES Accelerator	✔	✔	✔	✔	✔	
SHA Accelerator	✔	✔	✔	✔	✔	
RSA Accelerator	✔	✔	✔	✔	✔	
Hall Effect Sensor	✕	—	✕	—	—	
ULP Processor	✕	✕	✕	—	✔	
GDB Debugging	✔	✔	✔	✔	✔	Works with Wokwi for VS Code

Legend:
✔ - Simulated
□ - Partial implementation/work in progress
✕ - Not implemented (but if you need it, please open a feature request)
— - Not available on this chip

* The amount of SRAM can be customized using the "psramSize" attribute.

Advanced Usage

Flash and memory size

You can customize the size of flash and PSRAM by adding the following attributes to the chip:

Attribute	Description	Default
flashSize	Flash size in MB. Valid values: "2", "4", "8", "16", "32".	"4"
psramSize	PSRAM size in MB. Valid values: "2", "4", "8".	"4"

- ESP32 Custom flash size example

Custom Partition Table

You can specifiy a custom partititon table by adding a "partitions.csv" file to your project. Check out the ESP32 Partition Table Guide for the exact format of this file.

- ESP32 Custom partition table code example

Custom firmware offset

When loading a custom firmware, you can specify the offset of the firmware in the flash memory. By default, Wokwi will look at the firmware binary and try to figure out the offset automatically, based on the presence of the bootloader and the type of the chip. If Wokwi can't figure out the offset, it will assume that your firmware is an application firmware and load it at offset 0x10000.

You can specify the offset manually by adding the following attribute to the chip:

Attribute	Description	Default
firmwareOffset	Offset of the firmware in the flash memory, in bytes.	""

Changing the MAC address

You can change the MAC address of the WiFi interface by adding the following attribute to the chip:

Attribute	Description	Default
macAddress	MAC address of the WiFi interface, e.g. "24:0a:c4:12:45:56"	"24:0a:c4:00:01:10"

IOT PROTOCOLS

Why are IoT protocols important?

IoT protocols are an integral part of the IoT technology stack. Without IoT protocols and standards, hardware would be useless. This is because IoT protocols are the things that enable that communication—that is, the exchange of data or sending commands—among all those various devices. And, out of these transferred pieces of data and commands, end users can extract useful information as well as interact with and control devices.

With that in mind, we're going to look at some of the most important IoT protocols and standards that your business may use in 2023.

How many IoT protocols are there?

In short, a lot. The IoT is heterogeneous, meaning that there are all sorts of different smart devices, protocols, and applications involved in a typical IoT system. Different projects and use cases might require different kinds of devices and protocols.

For example, an IoT network meant to collect weather data over a wide area needs a bunch of different types of sensors, and lots of them. With that many sensors, the devices need to be lightweight and low-power, otherwise the energy required to transmit data would be enormous. In such an instance, low-power is the main priority, over, say, security or speed of transmission.

If, however, an IoT system consists of medical sensors on ambulances, sensors that transmit patient data ahead to hospitals, time is clearly going to be of the essence. What's more, HIPAA requires special security protocols for health data. So a higher-power, faster, and more secure protocol would be necessary.

Since there are so many different types of IoT systems and so many different applications, experts have figured out a way to sort all the components of IoT architecture into different categories, called layers. These layers allow IT teams to home in on the different parts of a system that may need maintenance, as well as to promote interoperability. In other words, if every system adheres to, or can be defined by, a specific set of layers, the systems are more likely to be able to communicate with each other through those layers.

One of the best frameworks for how you can understand IoT layers is the Open Systems Interconnection (OSI) model, which defines seven different layers in a top-down architecture. Top-down just means that the layers are defined starting from what the typical person uses to interact with an IoT system, like a smartphone app or website, and going down all the way to, for example, the ethernet cables that work in the background to transmit data.

Here are the layers:

1. **The application layer**, which encompasses the mobile and web applications through which you might interact with the devices in an IoT system.
2. **The presentation layer**, which encrypts and transforms data collected by IoT devices so that the application layer can present the information in a readable format.

3. **The session layer**, which acts as a kind of scheduler for incoming and outgoing data. Whenever two devices need to communicate within an IoT system, the system needs to schedule that communication by opening a session.
4. **The transport layer**, which is like a fleet of trucks in a shipping company, except that this layer transports packets of data instead of shipping containers.
5. **The network layer**, which is like the post office for data, coordinating where and when the system transfers data. Routers are the primary part of the network layer that tell data packets how to get to their destinations.
6. **The data link layer**, which corrects errors due to abnormalities or damaged hardware at the physical layer and links different devices so they can transfer data through the network layer.
7. **The physical layer**, which is made up of ethernet cables, cell towers, etc.

OSI is a conceptual model, meaning that it's not always how IoT systems actually appear. Just like a factory might be divided into different maintenance zones to simplify the process of finding and solving problems with different machines, the seven layers of OSI allow IT teams to zero in on where the cause of an error might lie. Plus, the OSI model presents a convenient method for explaining in detail the different possible parts of an IoT network.

Experts also simplify IoT architecture into three, four, or five-layer models, each of which can be broken down further if necessary based on the OSI construct. For instance, a three-layer model has an application layer, just like an OSI model. Then there's a network/internet layer. Lastly, there's a perception/sensing layer composed of all sensors in an IoT system.

A four-layer model is the easiest to compare to an OSI model. There's the application layer, which encompasses the application, presentation, and session layers of an OSI model. Then there's the transport layer and the network/internet layer for transferring data digitally. Finally, there's the physical network access layer, which encompasses any ethernet cables, routers, modems, etc. For a clearer understanding of the relationship between OSI and the four-layer model, see the image below.

Lastly, a five-layer model adds a business layer on top of other layers. The business layer might also be called the data analytics layer or the cloud database layer. This is the layer where data collected from sensors becomes actionable. With the increase in data-based maintenance and operations, most IoT systems have some form of a business layer today. The image below compares the three and five-layer architectures.

Again, one of the purposes behind organizing all these components into layers is to zero in on the different functions of the various protocols. For example, application protocols, which are included in all of the above models, are what transform/present the data in such a way that a typical user can understand it.

Application protocols can include:

- Extensible Messaging and Presence Protocol (XMPP)
- Message Queuing Telemetry Transport (MQTT)
- Constrained Application Protocol (CoAP)
- Simple Object Access Protocol (SOAP)
- Hypertext Transfer Protocol (HTTP)

In an ideal world, any protocol in this application layer should be able to communicate with any other protocol.

Network layer protocols, which allow devices in an IoT network to communicate with each other, can include:

- Bluetooth
- Ethernet
- Wi-Fi
- Zigbee
- Thread
- Cellular networks (4G or 5G)

And so on. Again, in an ideal IoT system, any of these protocols should be compatible with any other protocols in this layer.

What protocols do IoT-qualified devices use?

In essence, IoT protocols and standards are broadly classified into two separate categories. These are:

1. IoT data protocols (Presentation / Application layers)
2. Network protocols for IoT (Datalink / Physical layers)

IoT Data Protocols

Let's take a closer look at the protocols involved in each category.

IoT data protocols are used to connect low-power IoT devices. They provide communication with hardware on the user side—without the need for any internet connection. The connectivity in IoT data protocols and standards is through a wired or cellular network. Some examples of IoT data protocols are:

Extensible Messaging and Presence Protocol (XMPP)

XMPP is a fairly flexible data transfer protocol that is the basis for some instant messaging technologies, including Messenger and Google Hangouts. XMPP is an open protocol, freely available, and easy to use. That's why many use the protocol for machine-to machine (M2M) communication between IoT devices, or for communication between a device and the main server.

XMPP gives devices an ID that's similar to an email address. Then the devices can communicate reliably and securely with each other. Experts can tailor XMPP to different use cases and for transferring both unstructured data or structured data like a fully formatted text message.

MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight IoT data protocol. It features a publisher-subscriber messaging model and allows for simple data flow between different devices. MQTT's main selling point is its architecture. Its genetic make-up is basic and lightweight and, therefore, it's able to provide low power consumption for devices. It also works on top of a TCP/IP protocol.

IoT data protocols were designed to tackle unreliable communication networks. This became a need in the IoT world due to the increasing number of small, cheap, and lower-power objects that have appeared in the network over the past few years.

Despite MQTT's wide adoption—most notably as an IoT standard with industrial applications—it doesn't support a defined data representation and device management structure mode. As a result, the implementation of data and device management capabilities is entirely platform- or vendor-specific. Plus, the protocol has no built-in security measures, so security has to be managed at a device and/or application level.

If you want to learn more about MQTT, check out 'The Pros and Cons of Using MQTT in IoT'

CoAP (Constrained Application Protocol)

A CoAP is an application layer protocol. It's designed to address the needs of HTTP-based IoT systems. HTTP is the foundation of data communication for the World Wide Web.

While the existing structure of the internet is freely available and usable by any IoT device, it's often too heavy and power-consuming for IoT applications. This has led many within the IoT community to dismiss HTTP as a protocol not suitable for IoT.

However, CoAP has addressed this limitation by translating the HTTP model into usage in restrictive devices and network environments. It has incredibly low overheads, is easy to employ, and has the ability to enable multicast support.

Therefore, CoAP is ideal for use in devices with resource limitations, such as IoT microcontrollers or WSN nodes. It's traditionally used in applications involving smart energy and building automation.

Want to have direct remote control of an IoT device over the internet? Create a secure connection by using Nabto Edge alongside CoAP's request/response IoT protocol.

AMQP (Advanced Message Queuing Protocol)

An AMQP is an open standard application layer protocol used for transactional messages between servers.

The main functions of this IoT protocol are as follows:

- Receiving and placing messages in queues
- Storing messages
- Setting up a relationship between these components

With its high level of security and reliability, AMQP is most commonly employed in settings that require server-based analytical environments, such as the banking industry. However, it's not widely used elsewhere. Due to its heaviness, AMQP is not suitable for IoT sensor devices with limited memory. As a result, its use is still quite limited within the world of the IoT.

DDS (Data Distribution Service)

DDS is another scalable IoT protocol that enables high-quality communication in IoT. Similar to the MQTT, DDS also works to a publisher-subscriber model.

It can be deployed in multiple settings, from the cloud to very small devices. This makes it perfect for real-time and embedded systems. Moreover, unlike MQTT, the DDS protocol allows for interoperable data exchange that is independent of the hardware and the software platform. In fact, DDS is considered the first open international middleware IoT standard.

HTTP (HyperText Transfer Protocol)

We've briefly touched on the HTTP model before. As mentioned, the HTTP protocol is not preferred as an IoT standard because of its cost, battery life, huge power consumption, and weight issues.

That being said, it is still used within some industries. For example, manufacturing and 3-D printing rely on the HTTP protocol due to the large amounts of data it can publish. It enables PC connection to 3-D printers in the network and printing of three-dimensional objects.

Traverse firewalls securely by creating a TCP tunnel using Nabto Edge – which can transmit via an HTTP request/response protocol. Learn more here.

WebSocket

WebSocket was initially developed back in 2011 as part of the HTML5 initiative. WebSocket allows messages to be sent between the client and the server via a single TCP connection.

Like CoAp, WebSocket has a standard connectivity protocol that helps simplify many of the complexities and difficulties involved in the management of connections and bi-directional communication on the internet.

WebSocket can be applied to an IoT network where data is communicated continuously across multiple devices. Therefore, you'll find it used most commonly in places that act as clients or servers. This includes runtime environments or libraries.

Network Protocols for IoT

Now that we've covered IoT data protocols, we're going to look at the different network protocols for IoT.

IoT network protocols are used to connect devices over a network. These sets of protocols are typically used over the internet. Here are some examples of various IoT network protocols.

Lightweight M2M (LWM2M)

Many IoT systems rely on resource-constrained devices and sensors, devices that use very little power. The problem is that any communication between these devices or from the devices to a central server must also be extremely lightweight and require little energy.

Gathering meteorological data is one use case that requires hundreds or thousands of sensors over a wide area. Imagine the impact on the environment if all of those devices needed a lot of energy to function and transfer data. Instead, experts rely on lightweight communication protocols to reduce the energy consumption of such systems.

One of the network protocols that greatly facilitates lightweight communication is the Lightweight M2M (LWM2M) protocol. LWM2M provides remote, long-distance connectivity between devices in an IoT system. The protocol primarily transfers data in small, efficient packages.

Cellular

Cellular networks are another method for connecting IoT devices. Most IoT systems rely on the 4G cellular protocol, though 5G has some applications as well. Cellular networks generally take more power than most of the protocols mentioned above, but perhaps more importantly, a cellular connection requires you to pay for a sim card. When you need a lot of devices over a wide area, the cost can quickly become prohibitive. .

Still, cellular networks are very low-latency and can support high speeds for data transfer. 4G in particular was a huge step above 3G, since 4G is ten times faster. When you use your smartphone's data to control or interact with some IoT devices remotely, you're typically using 4G or 5G. An example would be when you use your cell data to view the feed from your smart security cameras when you're away from home.

Wi-Fi

Wi-Fi is the most well-known IoT protocol on this list. However, it's still worth explaining how the most popular IoT protocol works.

In order to create a Wi-Fi network, you need a device that can send wireless signals. These include:

- Telephones
- Computers

- Routers

Wi-Fi provides an internet connection to nearby devices within a specific range. Another way to use Wi-Fi is to create a Wi-Fi hotspot. Mobile phones or computers may share a wireless or wired internet connection with other devices by broadcasting a signal.

Wi-Fi uses radio waves that broadcast information on specific frequencies, such as 2.4 GHz or 5GHz channels. Furthermore, both of these frequency ranges have a number of channels through which different wireless devices can work. This prevents the overflowing of wireless networks.

A range of 100 meters is common for a Wi-Fi connection. That being said, the most common is limited to 10 to 35 meters. The main impacts on the range and speed of a Wi-Fi connection are the environment and whether it provides internal or external coverage.

Bluetooth

Compared to other IoT network protocols listed here, Bluetooth tends to frequency hop and has a generally shorter range. However, it's gained a huge user base due to its integration into modern mobile devices—smartphones and tablets, to name a couple—as well as wearable technology, such as wireless headphones. Since many different devices and systems can use bluetooth, it's great for promoting interoperability, enabling communication between vastly different systems and applications.

Standard Bluetooth technology uses radio waves in the 2.4 GHz ISM frequency band and is sent in the form of packets to one of 79 channels. However, the latest Bluetooth 4.0 standard has 40 channels and a bandwidth of 2Mhz. This guarantees a maximum data transfer of up to 3 Mb/s. This new technology is known as Bluetooth Low Energy (BLE) and can be the foundation for IoT applications that require significant flexibility, scalability, and low power consumption.

BLE wasn't originally a Bluetooth technology. Nokia developed the concept, which soon gained the notice of the Bluetooth Special Interest Group. Since then, Bluetooth fully adopted BLE as an important communication protocol.

But because of the low range of Bluetooth, it's not ideal for all use cases. Additionally, since any Bluetooth-enabled device is discoverable to any other bluetooth enabled device for pairing, BLE doesn't have a built-in security feature to prevent unauthorized connections. You would need to add security measures at the application layer.

So while BLE makes a great option for smart homes and offices, where there are a reasonable number of IoT devices in proximity that make up a single IoT system, BLE is not so great for transferring sensitive data over long distances or remotely accessing an IoT system from a long distance.

ZigBee

ZigBee-based networks are similar to Bluetooth networks in the sense that ZigBee already has a significant user base in the world of IoT.

However, its specifications slightly eclipse the more universally used Bluetooth. ZigBee has lower power consumption, low data-range, high security, and has a longer range of communication. (ZigBee can reach 200 meters, while Bluetooth maxes out at 100 meters.)

Zigbee is a relatively simple protocol, and is often implemented in devices with small requirements, such as microcontrollers and sensors. Furthermore, it easily scales to thousands of nodes. No surprise that many suppliers are offering devices that support ZigBee's open standard self-assembly and self-healing grid topology model.

Thread

Thread is a new IoT protocol based on Zigbee and is a method for providing radio-based internet access to low-powered devices within a relatively small area. Thread is similar very to Zigbee or Wi-Fi, but it is highly power efficient. Also, a Thread network is self-healing, meaning certain devices in the network can act as routers to take the place of a broken or failing router.

Thread can connect up to 250 devices, with up to 32 of those devices being active routers in the network. The newly developed Matter protocol, which operates at the application level to support interoperability (compatibility) between different IoT devices and protocols, often runs on top of Thread.

If you're a developer looking to implement one of these IoT Protocols and Standards in 2023, Check out our comparison of the best IoT Protocols for Developers.

Z-Wave

Z-Wave is an increasingly-popular IoT protocol. It's a wireless, radio frequency cased communication technology that's primarily used for IoT home applications. It operates on the 800 to 900MHz radio frequency, while Zigbee operates on 2.4GHz, which is also a major frequency for Wi-Fi.

By operating in its own range, Z-Wave rarely suffers from any significant interference problems. However, the frequency that Z-Wave devices operate on is location-dependent, so make sure you buy the right one for your country.

Z-Wave is an impressive IoT protocol. However, like ZigBee, it's best used within the home and not within the business world.

LoRaWAN (Long Range WAN)

LoRaWAN is a media access control (MAC) IoT protocol. LoRaWAN allows low-powered devices to communicate directly with internet-connected applications over a long-range wireless connection. Moreover, it has the capability to be mapped to both the 2nd and 3rd layer of the OSI model. LoRaWAN is implemented on top of LoRa or FSK modulation for industrial, scientific, and medical (ISM) radio bands.