£9.95 | €10.95 (DE) | €13.50 (other EU countries) | $12.95

elektorMAG
design › share › earn

24001
* SINCE 1961 *

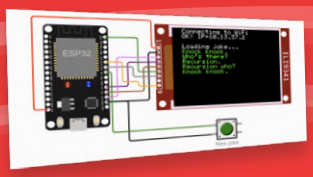
DEC. 2023 & JAN. 2024
ELEKTORMAGAZINE.COM

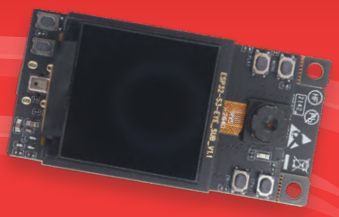
SPECIAL EDITION

140 Pages

Special edition
guest-edited by

ESPRESSIF





Prototyping With **Espressif Chips**

Try it with ESP Launchpad

ADF, IDF, and Other **SDKs**

Insights from
Espressif Engineers

ESP32 and **ChatGPT**

Automation **With** Rainmaker **and** Matter

Trying Out the **ESP32-S3-BOX-3**

In this issue

and much more!

9 770932 546006 01

INELTEK

Design-In Expertise & Service

Your Partner for ESPRESSIF

- **Ineltek** — Leading Espressif franchise distribution partner
- **Experience** — 36 years of experience in the semiconductor industry
- **Latest News** — Ineltek is always up to date regarding Espressif's innovations – Contact us!
- **Innovation** — Features are missing? We address your requests for next product generations
- **Application Support** — Dedicated and focused in-house support offering
- **Time to Market** — Espressif & Ineltek's FAE team are in close & regular contact to speed up your designs
- **Information** — We inform you on Espressif's products & wireless trends in trainings & webinars
- **Supply** — Popular Espressif SoCs, modules and EVKs available from stock

Contact us for Parts & Support

www.ineltek.com

Ineltek is the worldwide independent distributor with a **Passion for Innovation** and a **Commitment to Service**.

Founded in 1987, Ineltek gained the trust of thousands industrial customers as a technical semiconductor and design-in service provider. We work with your team to ensure our mutual success by providing the highest level of technical and commercial services for your projects.

Start your career with us!

Apply now at personal@ineltek.com

- Field Sales Engineer (m/f/x)
- Field Application Engineer (m/f/x)
- Line Management / Marketing (m/f/x)

Follow us!

INELTEK GmbH

Heidenheim, Wien, Castelfranco, London
Hamburg, München, Frankfurt, Dresden

Volume 49, No. 24001 (No. 526)
December 2023 & January 2024
ISSN 0932-5468

Elektor Magazine is published 8 times a year by
Elektor International Media b.v.
PO Box 11, 6114 ZG Susteren, The Netherlands
Phone: +31 46 4389444
www.elektor.com | www.elektormagazine.com

For all your questions
service@elektor.com

Become a Member
www.elektormagazine.com/membership

Advertising & Sponsoring
Büsra Kas
Tel. +49 (0)241 95509178
busra.kas@elektor.com
www.elektormagazine.com/advertising

Print
Senefelder Misset, Mercuriusstraat 35,
7006 RK Doetinchem, The Netherlands

Distribution
IPS Group, Carl-Zeiss-Straße 5
53340 Meckenheim, Germany
Phone: +49 2225 88010

Accelerating IoT Innovation

C. J. Abate *(Content Director, Elektor)*
Jens Nickel *(Editor-in-Chief, Elektor)*

Teo Swee Ann
(Founder/CEO, Espressif Systems)

Following the resounding success of both the SparkFun (2021) and the Arduino (2022) collaborations, we're delighted to have Espressif on board as our 2023 guest editor. Thank you, Espressif!

With groundbreaking solutions such as the ESP8266 and ESP32, the Espressif team has consistently delivered cutting-edge tools to the world's most innovative professional engineers, serious makers, and forward-thinking students. With this guest-edited *Elektor Mag* and the resulting exciting collaboration with Espressif's talented engineers, we are putting diverse examples of the company's solutions and expertise into the hands of Elektor members around the globe.

So, what is in store for you as you read this guest edition of *Elektor Mag*? In typical Elektor fashion, we have packed this issue with a wide range of content, from projects to tutorials, on topics including song recognition on an ESP32, embedded development with Rust, ESP32 and ChatGPT, facial recognition with the ESP32-S3-EYE, handy engineering insights, and more. We're confident that the magazine — along with the free Bonus Edition that we're sharing on our site and via our newsletter with the community — will generate months of innovation and dozens of exciting new electronics projects and applications.

Whether you're experienced with Espressif solutions or you're new to the company's products, make sure you post all of your Espressif-based projects on the Elektor Labs online platform at *elektormagazine.com/labs*. We look forward to learning about your designs. Enjoy this issue, and good luck with your next project!

ESPRESSIF × elektor

Welcome to this special guest edition of *Elektor Mag* — well-known for its dedication to fostering innovation, collaboration, and knowledge sharing — co-authored by Espressif Systems. We'd like to extend our heartfelt thanks to the fantastic Elektor team and editors for their incredible work and unwavering support in making this publication a reality. It's your contributions that have truly made this all happen.

We also want to express our deepest appreciation to the Espressif community and our esteemed partners, who have not only supported us, but have also actively contributed to this edition. Your firm dedication and valuable insights have been instrumental in shaping the content of this publication.

This special edition is a testament to the collaborative spirit and shared passion for open-source technology that our community and partners embody. In this edition, we dive deep into the principles that have propelled Espressif Systems to the forefront of the IoT industry. Get a first-hand look at our open-source initiatives and how they've helped build a robust developer community, which, in turn, played a pivotal role in our journey of innovation and empowerment.

We're also looking to the future, seeing the transformative potential of generative AI technologies,-such as ChatGPT. We'll see how they're poised to reshape industries, and how Espressif plans to harness them for innovation, with insights from our community and partners.

Join us on this journey as we explore technology, innovation, and community. Together, we'll uncover the amazing content in this guest edition, which will ignite your enthusiasm for the boundless possibilities in today's connected world.

Keep Innovating, and Keep Sharing!

THIS EDITION

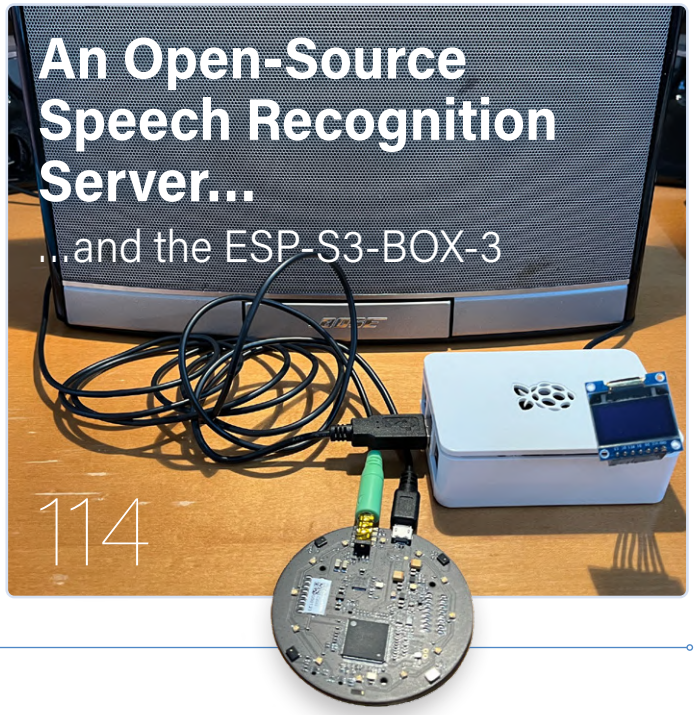
Volume 49, No. 24001 (No. 526)
December 2023 & January 2024

Regulars

Features

Industry

ESP32-C2-Based Coin Cell Switch
Up to a Five-Year Battery Life
126



Projects

Next Edition

Elektor Magazine January & February 2024
As usual, we'll have an exciting mix of projects, circuits, fundamentals and tips and tricks for electronics engineers and makers. This time, we'll focus on Power & Energy.

From the Contents:

- ESP32 Energy Meter
- Optimizing Balcony Power Plants
- Variable Linear Power Supply Ensemble
- Simple PV Power Regulator
- Programming for PC, Tablet, and Smartphone
- Project in Brief: Charger/Discharger
- Smart Kitchen Grocery Container

Elektor Magazine January / February 2024 edition will be published around **January 10th**. Arrival of printed copies for Elektor Gold members is subject to transport.

BONUS EDITION!
Want more content from Elektor and Espressif? In the coming weeks, we will publish a bonus edition of Elektor Mag — also guest-edited by Espressif — that's packed with more projects, tutorials, and background articles: an Espressif-style Dekatron, an ESP32-based authentication dongle, an ESP-BOX-based talking ChatGPT system, an interview with Home Assistant founder Paulus Schoutsen, and much more! Subscribe to the Elektor E-Zine (elektormagazine.com/ezine), and we'll deliver the bonus edition directly to your inbox!

A Color E-Ink Wi-Fi Picture Frame

By Jeroen Domburg (Espressif)

E-ink displays are widely used in the market, but they come predominantly in black and white, and the few color ones are rather expensive and difficult to adapt to a DIY project. In this article, we discuss using a reasonably priced seven-color display to get good color fidelity — achieved through dithering techniques — and with the ability to connect to a Wi-Fi network, using an ESP32-C3-WROOM-02 module from Espressif.

A while ago, our little one was born. Unfortunately, since her (great) grandparents are located somewhat spread out over the globe, real-life visits can't happen too often. Obviously, video calling is a thing nowadays, so we do keep in contact, but I wanted everyone to also be able to see her when we weren't online together. So, I thought: Why not build a picture frame around a color display and a Wi-Fi chip? Especially with her growing so fast, it would be nice to send a new photo every day and have it always show a new and up-to-date picture.

You've probably seen an E-ink display by now: if not in an E-book-reader, then surely in some local supermarket or other store where they've replaced the old paper price tags. They're great for displaying static images, as they retain their most recent image without any power consumption. Most of these screens are black-and-white, with sometimes either red or yellow thrown in as an extra color option. These are nice, but a black-and-white picture doesn't do justice to the vibrant colors of a happy baby derping around with her colorful toys.

On the Actual Market

At the point of writing (early 2023), E-book readers with color screens are finally starting to appear. Their quality seems good, with colors approaching those of a newspaper. Unfortunately, they're expensive, and the screens themselves don't seem to be available as spare parts yet, so there's no chance of easy re-use in a DIY project.

However, there has been one type of color E-ink display that's been available for use in DIY projects. It seems mostly sold by Waveshare, a Chinese company catering to the maker market, and the most common model is a 5.65-inch, seven-color 640×448 unit (**Figure 1**). With only seven colors and a refresh time of about a minute, it seems intended to be the big brother to the price tag variants, displaying static information with the colors used to, for example, have some flat background colors. They're certainly not intended to render photorealistic images.

That is a bit unfortunate: An E-ink display makes for a great "printed photo" simulacrum, as it doesn't require any backlight and is entirely static. I'm not the only one who thinks so, and a fair number of people ([1], [2], [3]) have already attempted to use dithering to make the screen into something that can display pictures with some fidelity. This is my attempt at doing so.

Hardware Requirements

Firstly, I needed to build the hardware. I began my design with some requirements. I wanted the picture frame to connect to a server over Wi-Fi once a day to pick up

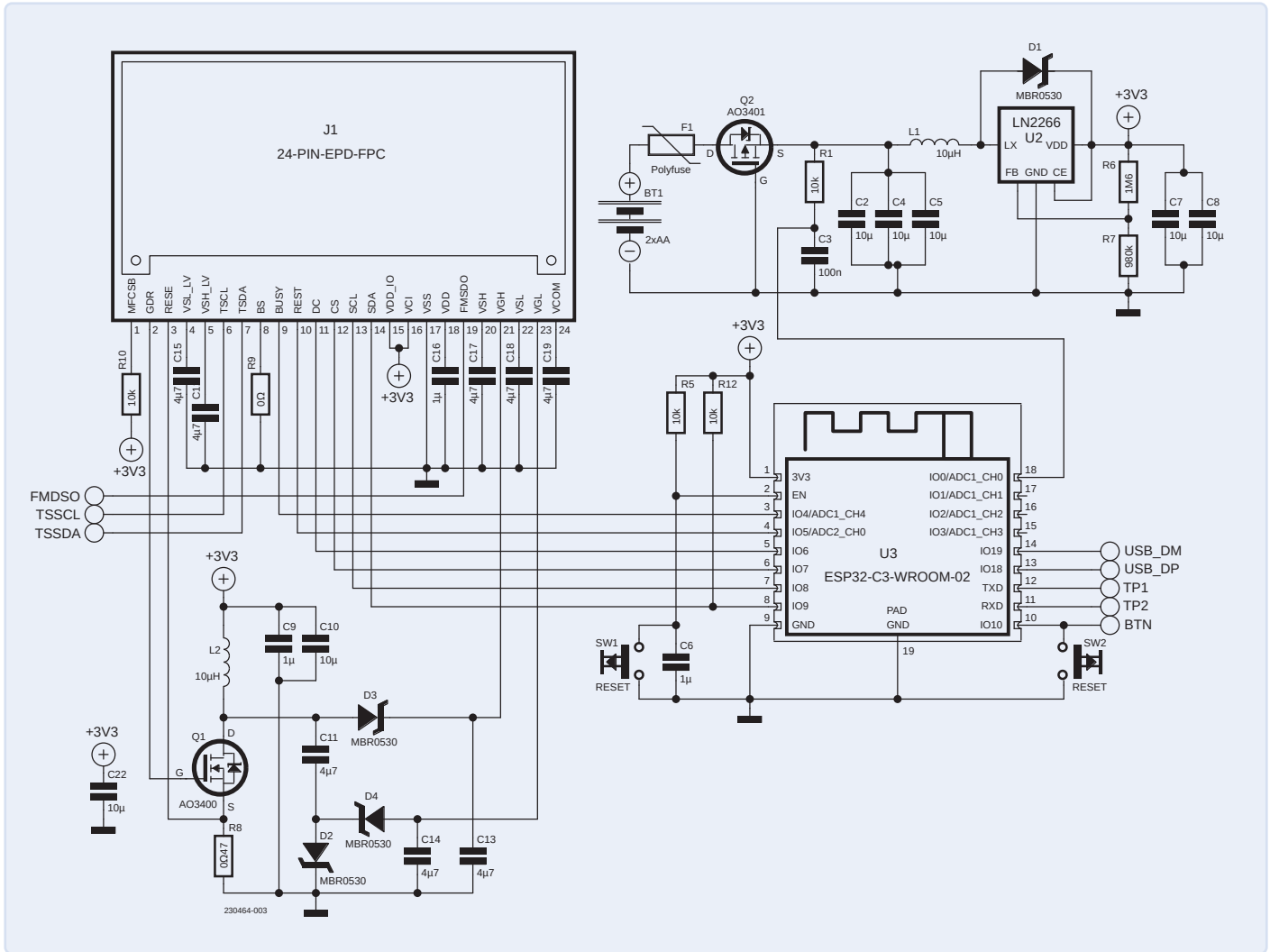
any new images. It would then display one on the E-ink display. If it managed to download new images, it would show one of those, else it would continue to show an old image. It would then go back to sleep for 24 hours. Picking a chip for this was not difficult, as I happen to work for Espressif, a company that makes great Wi-Fi-enabled microcontrollers that also have a deep-sleep mode that works pretty well. I picked an ESP32C3 for this: It's nice and cheap and has all the features I need.

Furthermore, I also had some requirements for the power supply. I wanted this thing to resemble a normal picture frame as much as possible (aside from the fact that it changed pictures nightly), so any external power supply was out of the question; the power supply had to be some kind of internal battery. I also wanted to ship it internationally, so it would be best if it could ship without a battery. That requirement ruled out any kind of rechargeable Li-ion cell. As you may see in the schematics in **Figure 2**, I settled on using 2 AA batteries; they're available everywhere, and even the grandparents would have little issue replacing them when needed.

Figure 1: The 5.65-inch, seven-color, 640×448 display from Waveshare. (Source: Waveshare)

As 2 AA batteries give a total voltage that starts off at about 3.2 volts, which decreases during the lifespan, I needed a boost circuit to get that up to the 3.3 V that the E-ink display and the ESP32C3 needed. Looking at the discharge graphs [4], if I wanted to get the most out of a pair of alkaline batteries, the boost circuit would need to work with an input voltage of down to 2.0 V or so. Additionally, as the ESP32C3 would need to be properly

Figure 2: The complete project schematic diagram, including the 3.3 V DC-DC converter, the ESP32-C3-WROOM-02 module, and the power supply for the display.



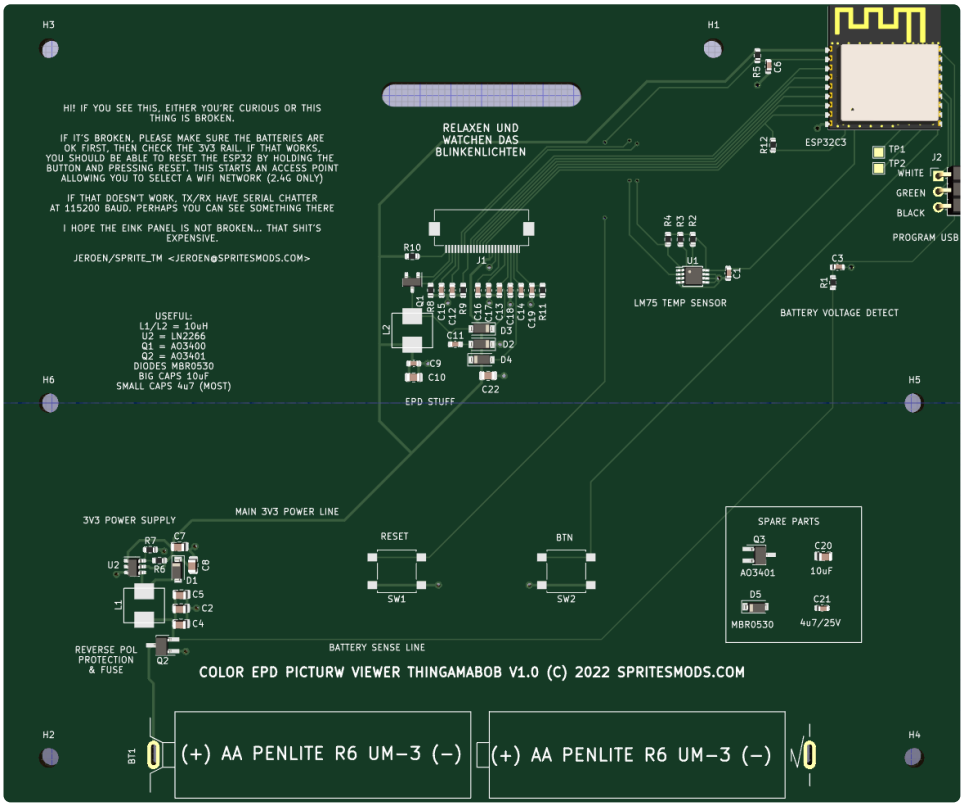


Figure 3: Silkscreen (component side) of the roomy PCB of the project.

powered even in deep sleep, the quiescent current would have to be suitably low. With these requirements, I started hunting for a feasible boost converter.

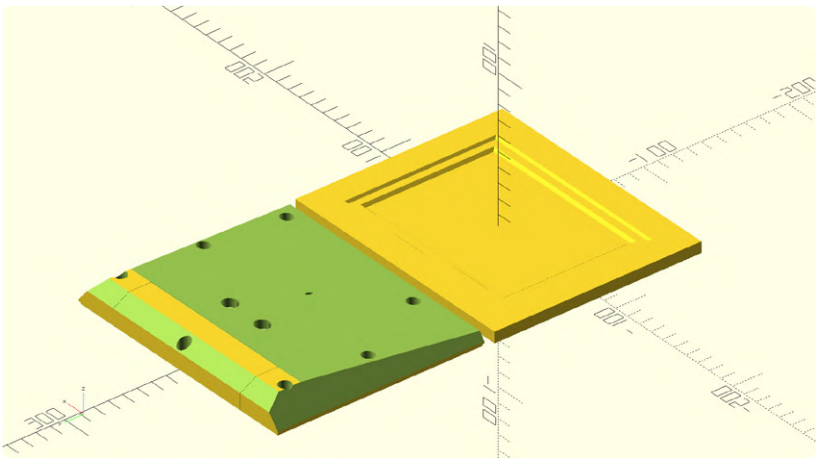
I settled on the Natlinear LN2266. This diminutive chip is made by a Chinese manufacturer, meaning it's generally a little bit less susceptible to the current silicon shortages. It works from 2 V and up, has a no-load current of 56 µA, and can provide the 500 mA or so of Wi-Fi startup current that the ESP32C3 needs. I designed the circuit so that the LN2266 pulls its power from the two batteries via a P-channel MOSFET that's configured to protect the batteries if they're inserted the wrong way around. The battery voltage is also connected via an RC filter to one of the ESP32-C3's ADC pins so it can get an idea of how much juice is still left. Initially, I also included a small polyfuse in the design, which was originally fitted between the battery and Q2, but that turned out to have too large a voltage drop to work. I replaced it with a 0-ohm resistor in my PCBs and removed it entirely from the design files.

The ESP32-C3 comes in the form of an ESP32-C3-WROOM-02 module, visible on the right in the schematics from Figure 2. This module contains the Wi-Fi microprocessor plus 4 MB of flash, as well as all of the RF components needed, including a trace antenna. To program it, I added an internal header, to which I can solder a USB cable. The internal USB-JTAG-serial converter takes care of the rest. I added an OTA firmware update feature to the firmware (i.e. the frame can download new firmware from my server), so, if I need to do any updates to units that are already closed up and out in the field, I can do that by pushing the new firmware remotely.

Then, there's the E-ink screen. It's connected to the PCB using a flat flex cable, and it needs some odds and ends to work, as is visible on the bottom-left of Figure 2: a MOSFET, an inductor, and some caps and diodes in order to generate the voltages it needs, some decoupling caps and a resistor or two. The E-ink display also has a connection for an external LM75 temperature sensor. I designed one in, just in case, but the current firmware doesn't use it, as the screen works just as well without it.

All of this is placed on a very roomy PCB, whose component-side silkscreen is shown in **Figure 3**. As the bare E-ink display is glass and somewhat fragile, I designed the PCB to work as a backing to make the total product less prone to breakage. The PCB is a bit bigger than that, as the mounting holes, any through-hole components (such

Figure 4: The case structure, designed with OpenSCAD.



as the battery contacts), and the Wi-Fi antenna must all be placed outside of where the E-ink display is mounted.

Housing

Finally, there's the case, which I designed in OpenSCAD, as shown in **Figure 4**. I used some tricks to make it look less bulky — it needs a fair amount of thickness because the AA batteries need to fit, and I did not want a large bulge in the bottom, so I used some large chamfers and a slope in the back. Those tend to be hidden if you look at the picture frame from the front, so, effectively, the overall picture frame looks a lot sleeker. The picture frame also looked a bit bulky compared to the visible area of the E-ink display. To fix that, I added a matte (also called "passe-partout"). As the matte is printed in white and contrasts with the black case nicely, it breaks up the "sea of black" and makes everything look in proportion.

In the back, there's the battery component. It's opened by unscrewing one screw (I didn't want to rely on fragile 3D-printed clips to keep it in place) and, as the silkscreen of the PCB has the correct battery orientation on it, replacing batteries should be simple.

Software/Firmware and Wi-Fi Connection

As the ESP32C3 is powered by batteries, I tried to make it do as little as possible. It should connect to Wi-Fi, talk to my server, see if there are new images it hasn't downloaded yet and, if so, download them, figure out what the best image to show is (newest or least times shown), display that, and shut down. Obviously, there are other things as well that need to be done, such as error handling and low-battery handling.

To connect to Wi-Fi, the device needs an SSID and a password. I didn't want to hard-code this, in case it needed to be changed. As such, the firmware comes with a copy of ESP32-WiFi-Manager [5], modified to fix some bugs. Specifically, when you press one of the buttons on the back of the picture frame while resetting the frame with the other button, it will start an access point that you can connect to. An embedded web server then gives you a user interface for picking a new SSID and setting the password for it.

After the picture frame has connected to Wi-Fi, it tries to retrieve data from a hard-coded URL to see what it's supposed to do. The URL also encodes some status data, such as the device's MAC, the battery voltage, and the current firmware ID. The server returns the ID of the most up-to-date firmware for that specific device, as well as an index of the ten most-recently-uploaded images. Some preferences are also sent, such as the time zone and the time the picture frame should try to wake up and run an update.

▲ *Figure 5: The sample image used for testing purposes.*

The picture frame actually has storage for ten images in its flash, and it will replace the oldest ones with newly downloaded images, should the server have images that are not in local storage. This way, it always has a stash of relatively recent images, which is good if, for whatever reason, connectivity drops out. It even allows the picture frame to be moved to a different place: While it will recycle old pictures without a Wi-Fi connection, it will still show something different every day.

Most of the server-side software isn't that complicated: There's a simple front-end webpage created around *Cropper.js* [6], which you can open on your phone or PC. It allows you to select a picture and crop out the bit that you'd like to show on the picture frame. There's a bit of JavaScript that then crops and scales the picture on the client side and sends the resulting data to the server. The server takes this, converts it to raw data for the E-ink display, and stores it in a MariaDB database.

When a picture frame connects, the server stores the data it sends, so I have a log of battery voltages and I can see if a firmware update actually "took." The server then checks the MariaDB database for the latest ten images, as well as other information, such as the last firmware version, and encodes that in JSON and sends the JSON back. All of that is pretty trivial.

Dithering

The only actually complicated bit is converting the RGB image into the seven pretty specific colors that the E-ink display can show. Take the image in **Figure 5**, for example.

If we wanted to convert this into black and white, we could simply check the luminance (lightness) for each pixel, and, if it's closer to black, make the result black; if it's closer to white, we'd make the result white. In other words, we take the closest "color" (restricting the "colors"

Figure 6: First converstion attempt, using 100% black and 100% white.

Figure 7: Black-and-white conversion with a diffusion process.

Figure 8: The image from Figure 7 with the addition of RGB values (see text).

Figure 9: The sample image with the application of the CIEDE2000 standard.

to only 100% black and 100% white) and change the resulting picture to what's shown in **Figure 6**.

That obviously is not a very close resemblance to the original picture. Even with black and white, we can do better using something called *error diffusion*. Effectively, every time we set a gray pixel black or white, we take the difference between the luminance of the pixel in the original photo and the luminance of the pixel we actually show on the E-ink screen (the "error" in "error diffusion") and partially add it to the surrounding pixels (the "diffusion"). The diffusion process can be done in multiple ways, Floyd-Steinberg being the most common, and it renders a pretty good black-and-white dithered image, as illustrated in **Figure 7**.

We can use that for our seven-color screen as well. The issue is that the definition of "closest color" gets a lot more complicated, as well as the definition of "adding." Even the definition of "color" gets hairy, as an E-ink display does not have backlight and, as such, the perceived colors differ, depending on what illuminates it: Light it with a candle flame, and you'll see different colors than when the display is seen in bright sunlight.

To get the colors, I took a temperature-adjustable light, set it to 4800 K (the average color temperature I think the display will be viewed at), displayed the 7 colors as flat rectangles on the E-ink display, and took a photo of it. I imported this into my computer and adjusted the colors manually until the on-screen ones looked as close as I could get to the E-ink ones. I then took the average RGB values of the seven colors and entered them into my program.

Of these seven colors, to get the "closest" to it on any pixel in the source image, we need a way to compare two colors, and, as we're using actual colors and not just black and white, we can no longer get away with simply using the luminance. A quick-and-dirty way to compare two colors is to see the (linearized) RGB space as a three-dimensional space and use the Euclidean distance to measure how close two colors are. With this model, adding colors can be done by simply adding the

RGB values. If we modify the Floyd-Steinberg dithering to adopt this method of picking the closest color, we get a reasonably acceptable image, as shown in **Figure 8**.

That's actually not bad! However, there are a few strange color artifacts. The most obvious one is that the flowerpot is not the same shade of blue: This is because the E-ink display simply doesn't have the available colors to replicate that shade, and no algorithm can compensate for that. But, there's more weirdness in the form of strange color banding, for instance in the shadow under the monkey's left arm, and the belly of the monkey is more orange than in the original picture.

Color Spaces Approach

The thing about calculating color differences in RGB space is that your eyes don't actually work in linear RGB space. The difference between two colors is actually a lot harder to define, and there have been multiple approaches to achieving this. One of the earliest approaches was to convert the RGB colors to CIELAB color space and take the difference. This approach is widely recommended on the internet, but it turns out it doesn't really give good results for colors that are not fully saturated. For me, the best approach turned out to be using the CIEDE2000 standard [7]. This is one of the more up-to-date perception models, and, while it's not trivial to calculate, it does render the best results, as can be seen in **Figure 9**. It's a good thing I already decided to do this server-side, so I wouldn't have to drain the batteries while doing this expensive calculation.

Note that the color banding in the shadows is gone and the monkey's belly is a more accurate shade of red. There are still flaws in the picture, such as the color of the flower pot, but, as I mentioned before, this is because the E-ink simply does not have the colors available to properly display that particular hue.

For speed, all of this logic is implemented in a simple C program. After the picture has been cropped in JavaScript on the client (using *Cropper.js*), it's sent to the server, where the PHP script calls this C script to convert the image into E-ink pixels, which are stored in a MariaDB table for the picture frames to pick up whenever they connect.

The web page is also suited for mobile use, so, when I or anyone else with access to the page snaps a particularly nice image, we can immediately crop it and queue it up for distribution to all the picture frames out there (**Figure 10**).

Result

With the PCB made and the 3D-printed parts printed, it's time to put it all together, as shown in **Figure 11**.

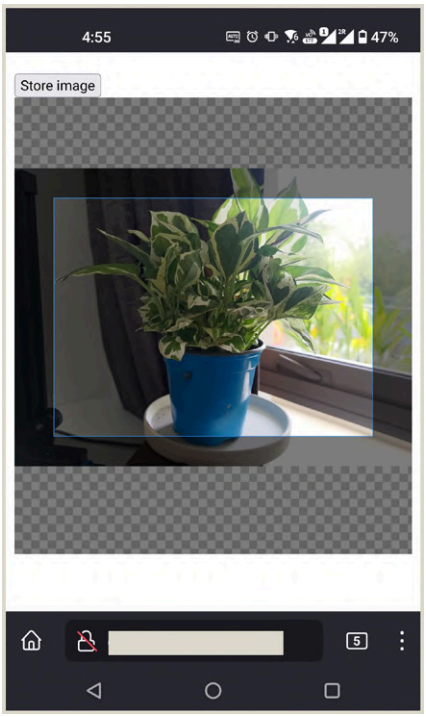


Figure 10: Quick-crop function for mobile use.

◀

Figure 11: The completed PCB assembly.

◀

Figure 12: The two sections of the case realized for this project.

◀

The back of the PCB contains all the electronics. The PCB is actually designed for repairability: If it breaks, and I'm not around to fix it, I might have a friend or two around who are good with electronics and can take a look at it. That means there are some debug instructions on the back and even some spare components in case something breaks. Otherwise, the PCB is pretty sparsely populated: Its dimensions are mostly defined by the size of the E-ink display on the other side. The amount of space would have allowed me to use larger components, but I have reels and reels of 0603 parts around, so I stuck with those for the jellybean bits; using a good soldering iron and a binocular microscope, I'm perfectly comfortable soldering all of that by hand.

Figure 13: The battery compartment with its own little flap.

Figure 14: The final result, shown next to the original.

The case is shown in **Figure 12** with the white matte in it. The matte has a cutout for the E-ink display: Even if the adhesive that sticks it to the PCB were somehow to come loose, it still would be kept in place by that. The case is closed using screws that are screwed into brass inserts. The brass inserts are fixed in place by heating them up using a soldering iron (with an old corroded solder tip inserted — I don't want to damage any usable tips) and melting them in place.

Everything goes together with a bunch of M3 screws. The battery compartment has its own little flap (**Figure 13**), so you don't need to disassemble the entire thing to replace the batteries. The batteries should last more than a year, according to my calculations. Also note that there are two buttons on the back. One is connected directly to the ESP32 reset, and the other to a general-purpose GPIO. You can use the reset button to make the device do a manual connect-and-refresh cycle, and this has the side effect that it'll show the next picture in line. When the other button is held while the picture frame is being reset, it starts up an access point that allows you to connect to the frame and reconfigure the Wi-Fi connection details.

Finally, in **Figure 14**, you may admire the end result. As pictures go, this one doesn't have the best fidelity ever, but the fact that we can send the frame a new photo every day from the other side of the world more than makes up for it.

As usual for me, this project is open source: With a 3D-printer, access to a server, and some skills, you can make your own version of this picture frame. All sources, PCB artwork, etc. can be found on GitHub [8].

230464-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the author at jeroen@spritesmods.com or the Elektor editorial team at editor@elektor.com.

About the Author

Jeroen Domburg is a Senior Software and Technical Marketing Manager at Espressif Systems. With more than 20 years of embedded experience, he is involved with both the software as well as the hardware design process of Espressifs SoCs. In his private time, he likes to tinker with electronics as well to make devices that are practically useful as well as devices that are less so.

Related Products

- **Waveshare 5.65″ ACeP 7-Color E-Paper E-Ink Display Module (600×448)**
  www.elektor.com/19847
- **ESP32-DevKitC-32E**
  www.elektor.com/20518
- **Dogan Ibrahim, *The Complete ESP32 Projects Guide*, Elektor 2019**
  www.elektor.com/18860

WEB LINKS

[1] GitHub 7-Color E-Paper Photo Frame: https://github.com/robertmoro/7ColorEPaperPhotoFrame
[2] Raspberry Pi E-paper frame: https://reddit.com/r/raspberry_pi/comments/10dbhnj/i_built_a_raspberry_pi_epaper_frame_that_shows_me
[3] E-paper picture project on YouTube: https://youtu.be/YawP9RjPcJA
[4] Discharge graphs: https://powerstream.com/AA-tests.htm
[5] ESP32-WiFi-Manager: https://github.com/tonyp7/esp32-wifi-manager
[6] Cropper.js: https://fengyuanchen.github.io/cropperjs
[7] CIEDE2000 standard: |https://en.wikipedia.org/wiki/Color_difference#CIEDE2000
[8] GitHub repository: https://github.com/Spritetm/picframe_colepd

ESP-Launchpad Tutorial

From Zero to Flashing in Minutes

By Dhaval Gujar, Espressif

Are you developing with Espressif chips? ESP-Launchpad is a web-based tool that simplifies firmware evaluation and testing. Let's take a closer look.

This article introduces ESP-Launchpad, a web-based tool that makes evaluating and testing firmware for Espressif chips hassle-free. It simplifies the development environment setup, leverages web browsers for flashing firmware, and benefits developers and non-developer users alike.

Why ESP-Launchpad?

When you build an open-source project, you want to have an easy way for users to evaluate it. As embedded developers, this evaluation means setting up the development host and programming tools, cloning the project, compilation and then programming the hardware.

With variations in the development host and its software environment, there are many possibilities for one of the steps to go wrong. Even if everything works fine, for the users who are not developers but who just want to try out the project, it's too much work.

If you are developing with any of the Espressif chips, you now have a better way of evaluating your project easily. ESP-Launchpad simplifies this process, making it quick and hassle-free to evaluate and test firmware designed for the ESP platform.

Ready, Set, ESP-Launchpad!

ESP-Launchpad is a web-based experience that harnesses the latest browser capabilities to provide an easy, no-frills way for flashing pre-built firmware onto ESP devices. This circumvents the need for traditional, dedicated software installations for developer host setup, and leverages the accessibility of web platforms. It uses the web serial interface supported by Chrome, Edge, and Opera browsers to communicate with the development board from the browser to program it and to provide serial console access.

Let's take a look at how you can easily launch your firmware with ESP-Launchpad.

Prepping the Firmware

Once your firmware has been finalized, the next step is to build it for all the targets that you wish to support and convert them to single binaries. Fret not, `esptool.py` (Espressif's one-stop-shop Python utility for all things related to flashing) provides a convenient `merge_bin` option that can help you easily create such a binary. You can find out more about this on our documentation page [1]. These single binaries should be uploaded to a publicly accessible URL that we will be using later.

ESP-Launchpad is a web-based experience that harnesses the latest browser capabilities to provide an easy, no-frills way for flashing pre-built firmware onto ESP devices.

Note: This tutorial won't go into more details on how that can be accomplished.

Fun fact: The secret sauce that powers ESP-Launchpad is, in fact, a JavaScript port of *esptool*, called *esptool-js*, that makes use of the WebSerial API under the hood.

Understanding the Basics: The Power of TOML

Before diving into the hands-on process, let's familiarize ourselves with the ESP Launchpad configuration TOML structure, a central aspect of ESP-Launchpad. ESP Launchpad configuration TOML files serve as simple configuration blueprints that detail your firmware's components, supported hardware, and complementary phone apps.

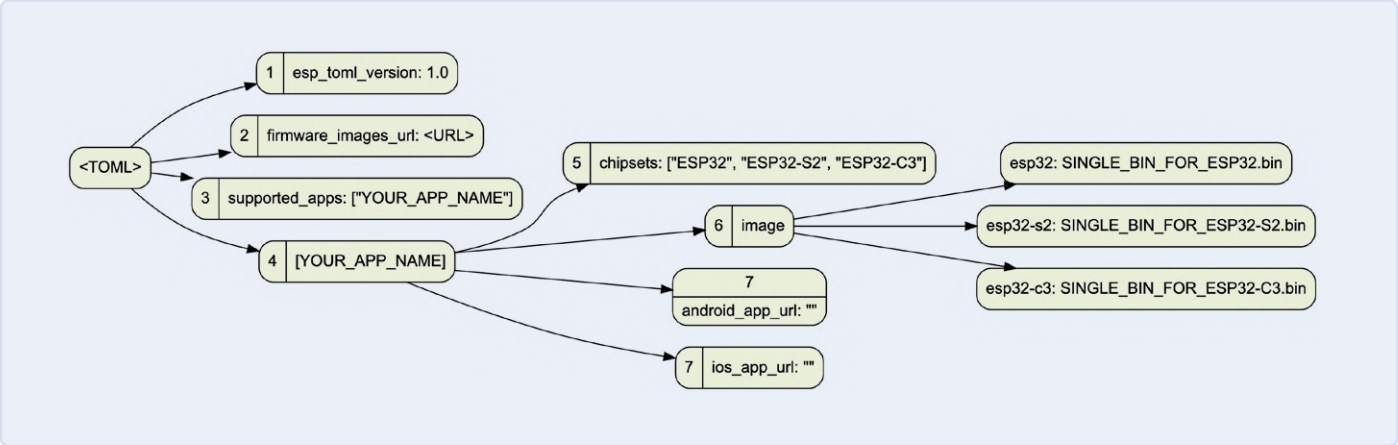


Figure 1: TOML structure tree diagram.

Here's an example!

```
esp_toml_version = 1.0
firmware_images_url = "<URL to your firmware images
directory>"
supported_apps = ["YOUR_APP_NAME"]

[YOUR_APP_NAME]
chipsets = ["ESP32", "ESP32-S2", "ESP32-C3"]
image.esp32 = "SINGLE_BIN_FOR_ESP32.bin"
image.esp32-s2 = "SINGLE_BIN_FOR_ESP32-S2.bin"
image.esp32-c3 = "SINGLE_BIN_FOR_ESP32-C3.bin"
android_app_url = ""
ios_app_url = ""
```

To better understand this, take a look at **Figure 1**.

- `esp_toml_version` specifies the version of the TOML schema.
- `firmware_images_url` is a publicly accessible URL of the file server where your firmware binaries are available for download. Pro tip: We host both our TOML files and binaries on GitHub, using GitHub Pages!
- `supported_apps` is an array of the list of apps that you support and for which the binaries are available. You can have multiple apps, and the ESP-Launchpad UI will show these apps in the available apps dropdown. The three values we just looked at were the top-level key-value pairs.
- `[YOUR_APP_NAME]` — This is a table that is effectively a collection of key-value pairs. It includes:
  - `chipsets` — An array containing a list of ESP chipsets that you will be providing pre-built firmware for. Note the naming convention here, all-caps, *ESP32-C3*.
  - `image.<chip-name>` — These are dotted keys associating the name of the binary image, with each of the supported targets. This will be appended to the `firmware_images_url` to obtain the final URL for the binary. Note again, the naming convention here, small case, *esp32-c3*.
  - `ios_app_url` and `android_app_url` are optional but special key-value pairs under the same app table, that allow you to show links in the form of QR codes that users can easily scan after your app has been flashed successfully.

Let's see what ESP-Launchpad looks like (**Figure 2**) if we provide the example TOML we just created, using the following imaginary URL:

https://espressif.github.io/esp-launchpad/?flashConfigURL=https://some-nice-url/my_app.toml

Okay, Looks Good! What's Next?

Just three simple steps for the user!

- Plug: Connect the ESP device to your computer's serial USB port.
- Connect: Use the tool's menu to connect with the device.
- Choose & Flash: Select the pre-built firmware and flash it.

After flashing, the user is greeted with a console showing the device log and a popup containing the QR codes of the phone apps, if you had added them to the TOML (**Figure 3**).

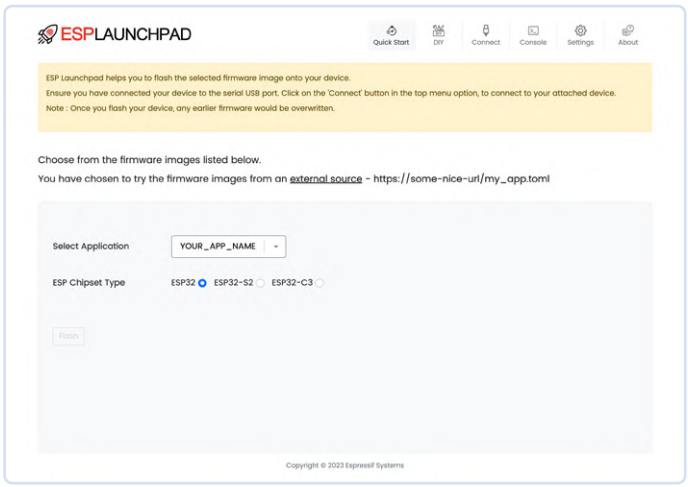


Figure 2: ESP-Launchpad with example config loaded.

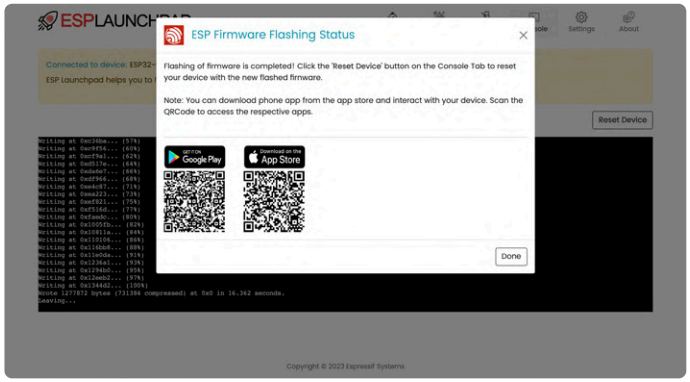


Figure 3: ESP-Launchpad final screen.

Congratulations, you have just published your app using ESP-Launchpad! Pro Tip: We made a badge (**Figure 4**) that you can add to your project's *README* or webpage and add a hyperlink to your app's flash config URL! You can find more info on the project's *README*. [2]

Figure 4: A badge to add!

ESP-Launchpad: Where Firmware Meets Community

What's unique about the ESP-Launchpad is its ability to let users share their firmware apps for others to try. By referencing a simple configuration file, developers can determine where the pre-built binaries of their firmware are picked from, the supported hardware, and even link to any complementary phone apps. This holistic approach makes sure the firmware isn't shared only as a binary, but as an experience that the developer intends.

WEB LINKS

[1] Basic Commands — Merge Binaries for Flashing: merge_bin: https://tinyurl.com/esp32mergebinaries
[2] The project's GitHub repository: https://github.com/espressif/esp-launchpad

Closing Thoughts

The true potential of any tool is realized when it's in the hands of its users. We urge you to explore ESP-Launchpad, integrate it into your workflows, and share your experiences. As we strive to refine our offerings, your insights and contributions will be invaluable. Together, let's redefine the open-source evaluation experience.

230596-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the author at dhaval.gujar@espressif.com or the Elektor editorial team at editor@elektor.com.

About the Author

Dhaval Gujar is an Engineer at Espressif Systems, with a focus on embedded systems. He's driven by the dynamic confluence of technologies like cloud, IoT, and more. Passionate about the evolving tech landscape, Dhaval delves into modern technological shifts with enthusiasm.

Related Products

- **ESP32-DevKitC-32E**
  www.elektor.com/20518
- **LILYGO T-Display-S3 ESP32-S3 Development Board (with Headers)**
  www.elektor.com/20299

ESP32 and ChatGPT

On the Way to a Self-Programming System...

Figure 1: The Arduino Nano ESP32 boards, running Arduino Framework (left) and MicroPython (right).

By Saad Imtiaz (Elektor Lab)

We combined the power of two Arduino Nano ESP32 microcontrollers, wireless communication, and the ChatGPT API to create a smart and interactive communication system. One of the boards is connected to ChatGPT; answers and code to be returned from this popular AI Tool will be transferred wirelessly to the other Nano Board. Can this two-board system possibly program itself with the help of ChatGPT?
Follow the step-by-step instructions and gain insights into how the boards communicate and how the ChatGPT API works.

In the field of embedded systems and Internet of Things (IoT), the Espressif ESP32 microcontroller has gained popularity for its versatility and robust capabilities. With its integrated Wi-Fi connectivity and powerful processing capabilities, it opens a world of possibilities for building smart and connected devices. In this article, we examine a demonstration project that utilizes the ChatGPT API, an AI language model created by OpenAI, to elevate two Arduino Nano ESP32 boards to new heights.

The aim of this project is to create a seamless communication system between the Nano ESP32 boards, one running on the Arduino IDE framework and the other on the MicroPython framework. By leveraging the ChatGPT API, we enable these boards to engage in witty and interactive conversations. Imagine the possibilities of having your microcontrollers sent with code snippets or provide creative solutions to your queries.

Throughout this article, we will go through the technical aspects of setting up the hardware, configuring the software, and establishing communication between the Nano ESP32 boards. We will explore the benefits and limitations of each framework Arduino IDE and MicroPython shedding light on their unique features and use cases. Additionally, we will provide practical code snippets, explanations, and insights to help you understand the inner workings of this project.

By the end of this article, you will clearly understand how to build your own Nano ESP32-based communication system, harnessing the power of AI and IoT. So, let's dive in and start on this exciting journey of creating an intelligent and interactive environment with Nano ESP32 and ChatGPT!

The Hardware

To begin, we will need a few essential components. Let's take a closer look at the hardware requirements:

1. Arduino Nano ESP32 Modules (×2)
The heart of our project lies in the ESP32 microcontroller. For communication, we will need two Arduino Nano ESP32 boards. These boards are widely available and offer a range of features such as Wi-Fi connectivity, ample processing power, and GPIO pins for interfacing with external components.

2. USB Type-C Cables (×2)
The USB Type-C cables are required to power and program the Nano ESP32 boards. These cables allow us to establish a connection between the boards and our computer, facilitating programming and data transfer.

3. Wi-Fi Network
A stable Wi-Fi network is essential for establishing communication between the two boards and the ChatGPT API. Make sure you have

access to a reliable Wi-Fi network with internet connectivity. This will enable the first Nano ESP32 board, (here called "Nano ESP32-1") to connect to the ChatGPT API and exchange messages.

Now that we have the required hardware components, let's move on to the software and programming aspects of the project.

Software and Programming

To set up the communication system and program the boards, we will need the following software tools:

1. Arduino IDE

The Arduino Integrated Development Environment (IDE) is most widely used IDE for programming Arduino Firmware-based microcontrollers. Now with its new IDE 2.0, it provides a more user-friendly interface, a simplified programming language, and a vast library of pre-built functions that make it easier to work with the Nano ESP32 boards.

2. MicroPython Firmware

MicroPython is a lightweight version of the Python programming language optimized for microcontrollers. It offers a more flexible and interactive programming experience compared to the Arduino IDE. To program the second Nano ESP32 ("ESP32-2") using MicroPython, we will need to install the MicroPython firmware on board. In **Figure 1** you can see both the boards connected. The board on the left is the running on Arduino framework and the one on the right is running on MicroPython.

3. ChatGPT API Access

To connect the Nano ESP32-1 board to the ChatGPT API, you will need access to the API. OpenAI provides various options for accessing the ChatGPT API, including API keys or tokens.To create your API key, you will have to head over to [1], create an Open AI account and then head over to [2] and then click on *Create New Secret Key*; you can also go by clicking the profile icon in the top-right corner of the webpage and by selecting *View API Keys* from the drop down menu. Once you created the API key, keep it saved at a safe place as you won't be able to copy the key again.

4. Programming

We are going to start with the Nano ESP32-1 running on the Arduino IDE. First we add the relevant libraries to our code. *WiFi.h* library is required so that we can add the functionality of the Nano ESP32 to connect to Wi-Fi Network. Then *HTTPClient.h* is used for sending our data, means the ChatGPT response to the second Nano ESP32 board. Finally we are using *ArduinoJson.h* (the library for using JSON on pretty much every board out there). It gives us the ability to JSON serialization and JSON deserialization, which we will use to access the ChatGPT API, send prompts and then receive responses. The codes for both boards are also available on GitHub [3].

```
// Load Wi-Fi library
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
// Replace with your network credentials
const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PWD";
//chatgpt api key
const char* apiKey =
  "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
```

Now we will be adding the IP Address of the second Nano ESP32 board (this IP address will be displayed in the serial monitor in the IDE, when the second Nano ESP32 board is connected to the Wi-Fi network).

```
const char* serverIP = "192.168.1.82";
  // IP address of the second Nano ESP32
const uint16_t serverPort = 80;
  // Port number on the second Nano ESP32
```

A function was made specifically to connect to the ChatGPT API and communicate with it, see **Listing 1**.

Now, let's move on to the code of the second Nano ESP32 board. Before we start with the programming; let's talk about how we are running MicroPython on the Nano ESP32: We will be using The Arduino Lab for MicroPython. To start, we will first head over to [4] and download the Arduino MicroPython firmware tool to flash the Arduino Nano ESP32, as shown in the **Figure 2** below.

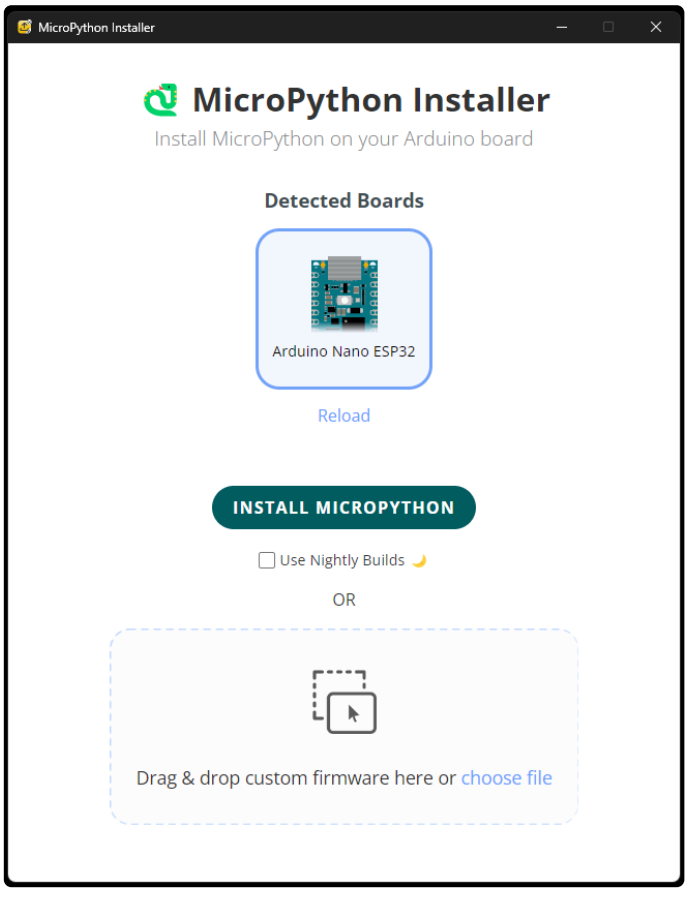


Figure 2: MicroPython Firmware tool by Arduino Labs.

Listing 1: Arduino Code to send requests to ChatGPT.

```
String sendChatGPTRequest(String prompt) {
 String received_response= "";
 String apiUrl = "https://api.openai.com/v1/completions";
 String payload = "{\"prompt\":\"" + prompt +
                  "\",\"max_tokens\":100, \"model\": \"text-davinci-003\"}";
 HTTPClient http;
 http.begin(apiUrl);
 http.addHeader("Content-Type", "application/json");
 http.addHeader("Authorization", "Bearer " + String(apiKey));

 int httpResponseCode = http.POST(payload);
 if (httpResponseCode == 200) {
 String response = http.getString();

 // Parse JSON response
 DynamicJsonDocument jsonDoc(1024);
 deserializeJson(jsonDoc, response);
 String outputText = jsonDoc["choices"][0]["text"];
 received_response = outputText;
 //Serial.println(outputText);
 } else {
 Serial.printf("Error %i \n", httpResponseCode);
 }
 return received_response;
}
```

Simply connect your Arduino Nano ESP32 to your computer, once the board is detected, click *Install Micropython* and after a few seconds your board will be flashed with the latest firmware, and you will be ready to use MicroPython on the Nano ESP32.

Now, we will be downloading Arduino Labs for MicroPython IDE from its official website at [5] and then download the latest version. After extracting the file simply launch the software by running the *Arduino Lab for Micropython.exe* file.

On the second Nano ESP32 board, the code is short and simple; we will first import the libraries. We will be using the *network* and *socket* libraries to connect to the Wi-Fi network and then to have our server port made where we can receive the code or responses from the Nano ESP32-1. Furthermore, *machine* and *time* libraries are required to use the GPIOs and the timer function for the Nano ESP32.

```
import network
import socket
import machine
import time

# Wi-Fi credentials
wifi_ssid = "WIFI_SSID"
wifi_password = "WIFI_PWD"

# Connect to Wi-Fi
wifi = network.WLAN(network.STA_IF)
wifi.active(True)
wifi.connect(wifi_ssid, wifi_password)

# Wait until connected to Wi-Fi
while not wifi.isconnected():
 pass
# Print the Wi-Fi connection details
print("Connected to Wi-Fi")
print("IP Address:", wifi.ifconfig()[0])
```

Then a socket server is created to receive data from Nano ESP32-1 and then the MCU goes into a `while` loop where it is waiting to receive any responses; when any code or response is received, the Nano ESP32 executes the code and then starts waiting again for any further instructions (see **Listing 2**).

In the **textbox "Arduino IDE, MicroPython and Others"**, we discuss the advantages and disadvantages of the two frameworks and understand why one may be more suitable than the other depending on the use case scenario.

Communication Between the Nano ESP32 Boards

Now, let's dive into the technical details of how the two Nano ESP32 boards communicate with each other over Wi-Fi in this project. This section aims to provide a more detailed explanation of the communication process for a better understanding.

Nano ESP32-1 (Arduino IDE Framework)
Nano ESP32-1, running on the Arduino IDE framework, establishes a Wi-Fi connection to communicate with external services and devices. It connects to a specific Wi-Fi network using the provided credentials. Once connected, Nano ESP32-1 waits for user input on the Serial Monitor of the Arduino IDE.

When the user types *GPT* and hits , Nano ESP32-1 prompts the user to enter a message. The user's prompt message is then sent from Nano ESP32-1 to the ChatGPT API for processing. This transmission occurs over the established Wi-Fi connection, enabling Nano ESP32-1 to communicate with the external API. In **Figure 3**, you can see the entire communication and connection loop between the two boards and the ChatGPT API.

The ChatGPT API, an interface to the AI language model, receives the prompt message from Nano ESP32-1. Using advanced natural language processing techniques and machine learning algorithms, the API analyzes the prompt to understand its context and generate a witty response or code snippet. The ChatGPT model within the API leverages its vast training data and language understanding capabilities to provide a relevant and engaging output.

Nano ESP32-1 receives the generated code snippet from the ChatGPT API and stores it in memory. This code snippet represents the AI-generated response to the user's prompt. Nano ESP32-1 then displays the received code snippet on the serial monitor, allowing users to review the instructions generated by the AI. In **Figure 4** you can see the serial monitor output of both the boards, where the Nano ESP32-1 is sending the code over to Nano ESP32-2 after getting the response from the ChatGPT.

Nano ESP32-2 (MicroPython Framework)
On the other side, Nano ESP32-2 runs on the MicroPython framework. Like Nano ESP32-1, Nano ESP32-2 establishes a Wi-Fi connection to the same network using the provided credentials. This allows it to receive instructions wirelessly from Nano ESP32-1.

Nano ESP32-2 sets up a socket connection and listens on a specific port, typically port 80. A socket is a software endpoint that enables communication between two devices over a network. By listening on a specific port, Nano ESP32-2 is ready to receive incoming data from Nano ESP32-1.

When Nano ESP32-1 wants to send the generated code snippet, it establishes a connection to Nano ESP32-2 through the socket connection. The code snippet is then transmitted to Nano ESP32-2, where it is received and processed.

Listing 2: Receive and execute code on the second board (Python).

```
# Create a socket server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', 80))
server.listen(1)

# Accept and handle incoming connections
while True:
 print("Waiting for connection...")
 client, addr = server.accept()
 print("Client connected:", addr)

 # Receive the code snippet from the first ESP32
 code = ""
 while True:
 data = client.recv(1024)
 if not data:
 break
 code += data.decode()

 # Execute the received code
 try:
 exec(code)
 response = "Code executed successfully"
 print (response)
 except Exception as e:
 response = "Error executing code: " + str(e)

 # Send the response back to the first ESP32
 client.sendall(response.encode())
 # Close the connection
 client.close()
 print("Client disconnected")
```

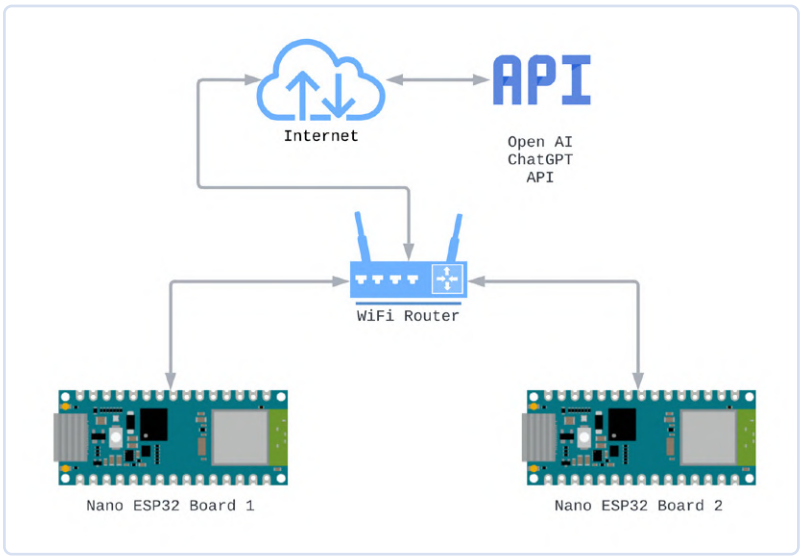


Figure 3: Communication between the two Arduino Nano ESP32 boards and ChatGPT API.

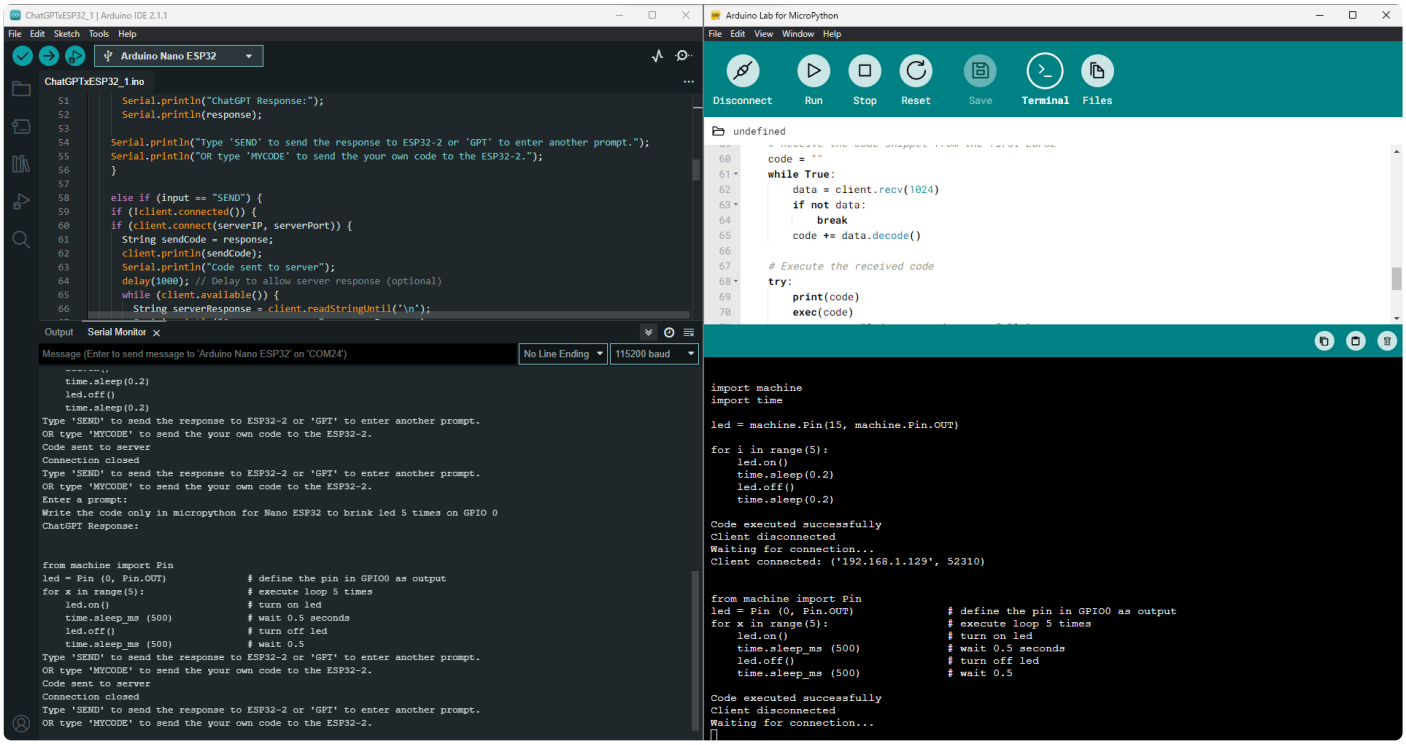


Figure 4: Communication between the two Nano ESP32, Arduino IDE (left) and Arduino Lab for MicroPython (right).

Nano ESP32-2 processes the received code snippet, extracting the instructions embedded within it. These instructions define specific tasks to be executed by Nano ESP32-2. For example, the code snippet might instruct Nano ESP32-2 to blink an LED for a certain duration or perform other actions based on the AI-generated response.

After executing the instructions, Nano ESP32-2 sends a response back to Nano ESP32-1 over the established socket connection. This response serves as confirmation that the received code snippet was successfully executed. It ensures that Nano ESP32-1 is aware of the completion of the requested task.

By following this communication process, the two Nano ESP32 boards can exchange messages and collaborate effectively. Nano ESP32-1 interacts with the ChatGPT API, leveraging its AI capabilities, while Nano ESP32-2 acts as the recipient and executor of the AI-generated instructions.

Limitations of ChatGPT in Programming

While ChatGPT is a remarkable AI language model, it does have certain limitations when it comes to generating functional code, especially in more complex programming scenarios. During testing, it was found that ChatGPT was able to provide functional code for a simple blink sketch only about 60% of the time. This indicates that the model's ability to generate accurate and working code snippets is not guaranteed in every instance.

One challenge arises when requesting code snippets from ChatGPT. In some cases, the response includes additional text before and after the actual code, which can cause issues when attempting to send the entire response to the second Nano ESP32 board. This text may lack proper formatting or syntax, rendering it incompatible with direct execution. In **Figure 5** you can see the response of ChatGPT after prompting it to provide a code for checking the sensor value at pin 36 of the ESP32.

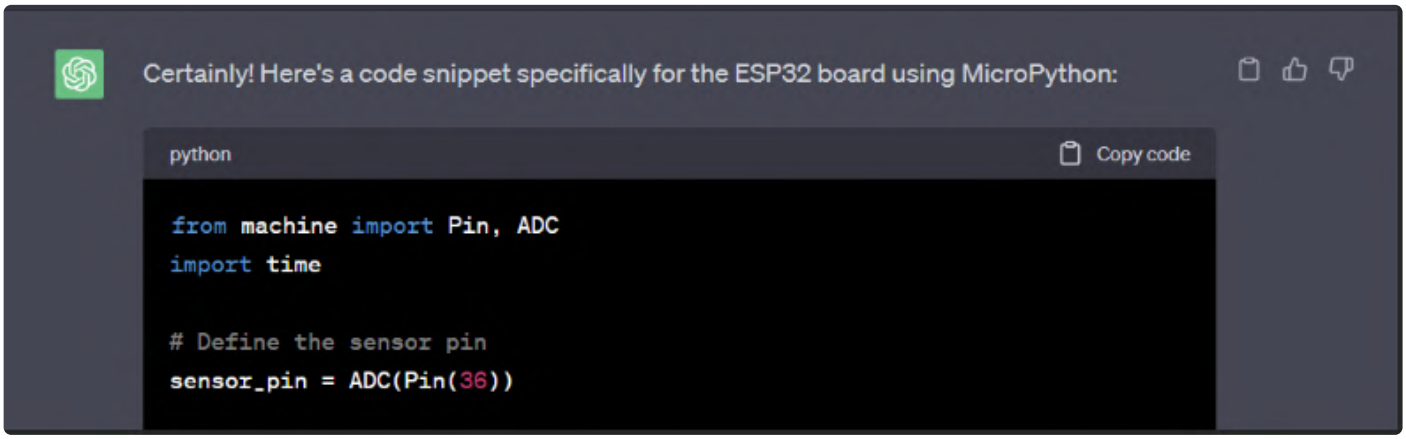


Figure 5: Response by ChatGPT after prompting it to write a code snippet.

Applications and Use Cases

The project of integrating Nano ESP32 boards with the ChatGPT API opens a range of exciting applications and use cases. Here are a few notable examples:

- **Smart Home Automation:** By leveraging the ChatGPT API, users can interact with their smart home systems through natural language prompts. They can control lights, adjust temperature settings, or even ask for recommendations on energy-saving practices.
- **Prototyping and Rapid Development:** The combination of Nano ESP32 boards and AI-generated code snippets allows for rapid prototyping of IoT projects. Users can quickly test and iterate ideas by receiving instant code suggestions for various functionalities.
- **Educational Tool:** The project provides a valuable educational tool for programming learners. Students can engage in interactive coding conversations with the AI, receive guidance, and learn new programming techniques.
- **Personal Assistant and Information Retrieval:** The ChatGPT integration can be used to develop a personal assistant application. Users can ask questions, seek recommendations, or obtain information on various topics, all powered by AI-driven responses.
- **Creative Coding Inspiration:** The project serves as a source of creative coding inspiration. It can generate unique and innovative ideas for animations, visualizations, or interactive projects, sparking the creativity of developers.

To overcome this limitation, an additional feature was implemented in the code, that allows users to have the option to enter the code themselves and use ChatGPT as a reference or guide to optimize and reduce the code lines. This approach allows for greater control over the generated code and addresses the formatting and syntax issues encountered when relying solely on ChatGPT's output.

It is important to acknowledge that while ChatGPT can provide valuable insights and suggestions, it still has room for improvement in the realm of programming. As developers, we must exercise caution and not solely rely on ChatGPT's responses for critical or complex programming tasks. Using ChatGPT as a reference or for code optimization can enhance the development process, but it should be supplemented with human expertise and thorough code review.

As AI language models continue to evolve, we can expect improvements in their ability to generate more accurate and reliable code. These advancements will open new horizons for leveraging AI in programming and enable even greater collaboration between humans and machines. To make the most of ChatGPT, it's important to leverage it as a tool and not rely solely on its outputs. Combining human expertise and judgment with the AI-generated responses can lead to more accurate and reliable outcomes. By understanding and considering these limitations, users can effectively utilize ChatGPT while mitigating potential risks and challenges.

Exploring Alternative Approaches

In addition to the previously discussed project setup, there are alternative methods and approaches that could be utilized to achieve Nano ESP32 communication and AI integration. Let's explore a few different ways this project can be implemented on the Nano ESP32 platform:

1. MQTT Protocol

The MQTT (Message Queuing Telemetry Transport) protocol is a lightweight and efficient messaging protocol commonly used in IoT applications. Instead of relying on Wi-Fi and socket connections, Nano ESP32 boards can communicate with each other using the MQTT protocol. MQTT brokers can be set up to facilitate messaging between the boards, enabling seamless data exchange and AI integration.

2. Direct Web Socket Connection

Another approach is to establish a direct Web Socket connection between the Nano ESP32 boards. Web Socket is a communication protocol that allows full-duplex communication over a single TCP connection. By implementing Web Socket on the Nano ESP32 boards, they can establish a persistent and bidirectional connection, enabling real-time communication and AI integration.

3. ESP-NOW Protocol

ESP-NOW is a communication protocol specifically designed for low-power devices, allowing for fast and reliable data transmission between Nano ESP32 boards. This protocol can be leveraged to establish direct communication between the boards, bypassing the need for Wi-Fi connections. By integrating AI capabilities on one board and utilizing ESP-NOW for communication, real-time responses and code snippets can be exchanged efficiently.

4. On-Device AI Integration

Instead of relying on external APIs, AI models can be deployed directly on the Nano ESP32 boards themselves. TensorFlow Lite, for example, allows AI models to be deployed and executed locally on microcontrollers. By training or fine-tuning a model for specific tasks, such as generating code snippets, the Nano ESP32 boards can provide AI-driven responses without the need for external connections.

These alternative approaches offer different advantages and considerations depending on the specific project requirements and constraints. Factors such as power consumption, latency, complexity, and scalability should be considered when choosing the most suitable approach.

By exploring these alternative methods, developers can extend the capabilities of Nano ESP32 boards, integrate AI at different levels, and tailor the communication system to their specific needs.

The project of Nano ESP32 communication and AI integration is not limited to a single approach. By considering alternative methods like MQTT, Web Socket, ESP-NOW, or on-device AI integration, developers can customize the project to meet their unique requirements and take advantage of the full potential of the Nano ESP32 platform.

Arduino IDE, MicroPython, and Others

Arduino IDE Framework

The Arduino IDE is a popular choice for beginners and hobbyists due to its simplicity and ease of use. It provides a beginner-friendly programming environment with a simplified version of the C++ programming language. Here are some advantages and disadvantages of using the Arduino IDE framework with Nano ESP32:

Advantages

- Easy to Learn: The Arduino IDE offers a gentle learning curve, making it accessible to those new to programming or microcontrollers.
- Large Community and Library Support: Arduino has a vast community of enthusiasts, extensive online resources, and a wide range of libraries and examples available, simplifying development.
- Quick Prototyping: Arduino IDE allows for rapid prototyping with its simplified syntax and library support, enabling faster development cycles.

Disadvantages

- Limited Language Features: The Arduino IDE has a simplified version of C++, which can limit the use of advanced programming features compared to other languages.
- Memory and Performance: The Arduino IDE may not be as efficient in terms of memory usage and performance compared to other frameworks.
- Less Flexibility: Arduino IDE imposes certain conventions and limitations, which can restrict the full potential of the Nano ESP32's capabilities.

MicroPython Framework

MicroPython is a lightweight implementation of the Python programming language designed for microcontrollers. It offers a more flexible and interactive programming experience. Here are some advantages and disadvantages of using the MicroPython framework with Nano ESP32:

Advantages

- Python Language: MicroPython allows developers to leverage the powerful and expressive Python language, making it easier to write complex code and algorithms.
- Interactive REPL: MicroPython provides a Read-Eval-Print Loop (REPL) environment, allowing developers to interactively test and experiment with code directly on the Nano ESP32 board.
- Efficient Memory Usage: MicroPython is designed to be memory-efficient, making it suitable for resource-constrained devices like the Nano ESP32.

Disadvantages

- Learning Curve: MicroPython may have a steeper learning curve for beginners not familiar with the Python language.
- Limited Library Support: Although MicroPython has a growing collection of libraries, it may have fewer options compared to the extensive Arduino library ecosystem.
- Performance Trade-offs: While MicroPython offers flexibility, the interpreted nature of the language may result in slightly slower execution compared to compiled languages like C++.

So, unleash your creativity, experiment with different approaches, and build innovative IoT systems that combine AI and Nano ESP32 in exciting new ways.

Fascinating Intersection of AI & IoT

The integration of Nano ESP32 boards with the ChatGPT API represents a fascinating intersection of AI, IoT, and programming. By enabling Nano ESP32 boards to communicate and interact with an AI language model, we unlock a world of possibilities. From smart home automation to rapid prototyping and educational tools, the applications are vast (see **textbox "Use Cases"**).

However, it's crucial to keep in mind the limitations of ChatGPT and exercise caution when working with AI-generated responses, particularly in programming and coding scenarios. Combining human expertise and judgment with AI-generated content will yield the best outcomes.

With this project, users can unleash their creativity, explore new horizons in IoT development, and enhance their coding skills. So, grab your Nano ESP32 boards, dive into the world of AI-powered interactions, and embrace the limitless potential of this exciting integration!

230485-01

Questions or Comments?

If you have technical questions or comments about this article, you can contact Saad Imtiaz at saad.imtiaz@elektor.com or the Elektor editorial team at editor@elektor.com.

About the Author

Saad Imtiaz is an engineer with experience in embedded systems, mechatronic systems, and product development. He has collaborated with more than 200 companies, ranging from startups to global enterprises, on product prototyping and development. Saad has also spent time in the aviation industry and has led a technology startup company. Recently, he joined Elektor in 2023 and drives project development in both software and hardware.

Other Frameworks

One of the notable advantages of the Nano ESP32 is its flexibility to work with different frameworks, besides Arduino IDE, MicroPython there are PlatformIO, ESP-IDF, JavaScript (Node.js), and Lua frameworks. The Nano ESP32 offers compatibility with several other frameworks, further expanding its versatility. This flexibility allows developers to choose the framework that best suits their project requirements and their familiarity with programming languages.

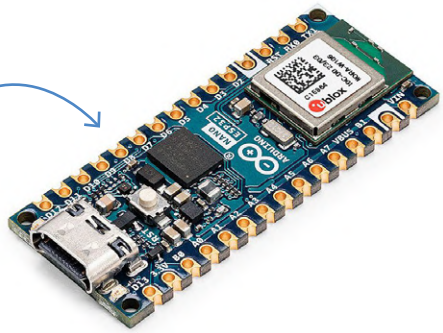
Let's explore a few more frameworks that can be used with the Nano ESP32 and their benefits:

Mongoose OS: Mongoose OS is an open-source operating system for microcontrollers, including the Nano ESP32. It provides a platform-agnostic environment with built-in cloud connectivity and supports multiple programming languages such as JavaScript, C, and C++. Mongoose OS simplifies the development process by offering an easy-to-use command-line interface, rich libraries, and remote device management capabilities.

Zephyr RTOS: Zephyr RTOS is a real-time operating system designed for resource-constrained systems, including the Nano ESP32. It offers a scalable and modular architecture, making it suitable for projects that require multitasking, real-time processing, and advanced device management. Zephyr's broad hardware support and extensive set of drivers and libraries enable developers to build complex IoT applications with ease.

ESPHome: ESPHome, designed for ESP32 and ESP8266 devices, is a versatile IoT framework with a modular architecture akin to Zephyr RTOS. It offers broad hardware support, an array of drivers, and libraries for simplified development of resource-constrained systems. This framework facilitates real-time processing, multitasking, and advanced device management, making it an excellent choice for home automation IoT projects. Choosing the right framework depends on factors such as project requirements, familiarity with the programming language, available libraries and community support, and desired level of control over hardware and performance. Each framework brings its set of benefits and trade-offs, allowing developers to select the most suitable one for their specific use cases.

With the Nano ESP32's compatibility with various frameworks, developers have the freedom to explore different programming languages, ecosystems, and development approaches. This flexibility empowers them to create innovative IoT solutions, leverage existing tools and libraries, and efficiently utilize the Nano ESP32's capabilities.



Related Products

- **Arduino Nano ESP32**
  www.elektor.com/20562
- **Arduino Nano ESP32 with Headers**
  www.elektor.com/20529
- **Dogan Ibrahim and Ahmet Ibrahim, *The Official ESP32 Book* (E-book) (Elektor 2017)**
  www.elektor.com/18330
- **Günter Spanner, *MicroPython for Microcontrollers* (Elektor 2021)**
  www.elektor.com/19736

WEB LINKS

[1] Open AI: https://platform.openai.com
[2] Open AI - API Keys: https://platform.openai.com/account/api-keys
[3] Project - GitHub Repository : https://github.com/elektor-labs/ChatGPTxESP32
[4] Arduino Labs - MicroPython Installer: https://labs.arduino.cc/en/labs/micropython-installer
[5] Arduino Labs - MicroPython IDE: https://labs.arduino.cc/en/labs/micropython

Walkie-Talkie with ESP-NOW

Not Quite Wi-Fi, Not Quite Bluetooth!

By Clemens Valens (Elektor)

Imagine your wireless project needs both fast response times and long-range capabilities? Wi-Fi and Bluetooth are unsuitable for such applications. Maybe ESP-NOW is a good alternative? Connections are established almost instantaneously, and ranges of several hundred meters are possible. In this article, we try it out in a simple walkie-talkie (wireless intercom) application.

The ESP32 from Espressif is often used for its Wi-Fi and Bluetooth capabilities, a domain in which it excels. Wi-Fi and Bluetooth are great protocols for all sorts of wireless applications, but they have their limitations.

An inconvenience of Wi-Fi is the time needed to establish a connection. Also, Wi-Fi doesn't allow for direct communication (peer-to-peer, **Figure 1**) between devices. There is always a router involved. Because of this, Wi-Fi is not really suited for simple low-latency remote controls to open a garage door or to switch a light on and off. Such tasks require immediate response. To work around this, Wi-Fi applications tend to be powered on and connected all the time. As a result, they consume a lot of energy, even when idle.

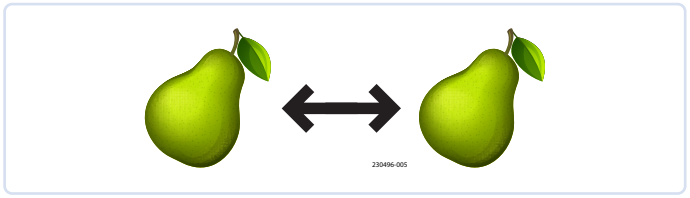


Figure 1: Pear-to-pear communication as shown here is not possible with Wi-Fi; a router is always needed between the two nodes.

Bluetooth, on the other hand, features fast connection setup and peer-to-peer communication and is excellent for low-latency remote controls. However, Bluetooth is intended for short-range applications with communicating devices spaced up to, say, ten meters apart. True, long-range Bluetooth exists, but it is not widely available yet.

The Solution: ESP-NOW

Espressif's ESP-NOW [1] wireless protocol is a solution for situations that require both quick response times and long range while using the same frequency band as Wi-Fi and Bluetooth.

The protocol combines the advantages of Wi-Fi and Bluetooth. ESP-NOW is targeted at home automation and the smart home. As it allows for one-to-many and many-to-many topologies (**Figure 2**), it needs no router, gateway, or worse, a cloud.

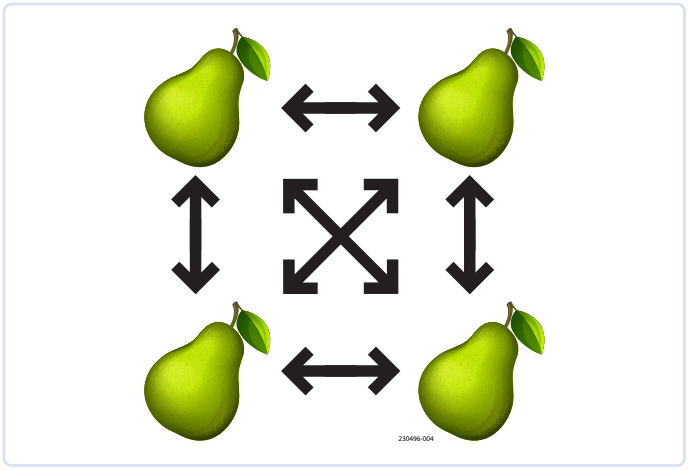


Figure 2: ESP-NOW supports many-to-many networking. In such a network, each node can talk to the other nodes directly without requiring a router.

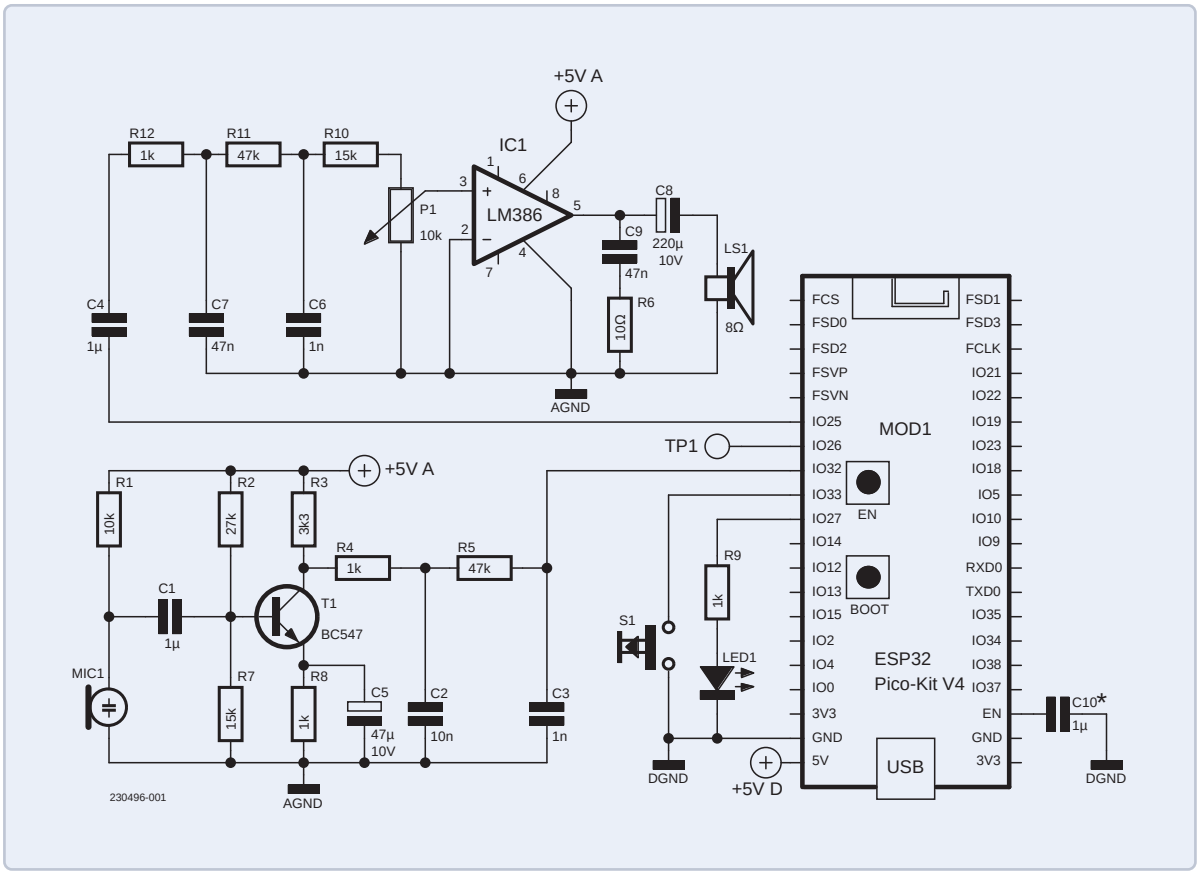


Figure 3: A simple microphone preamplifier on the input and a classic LM386 as power amplifier at the output. Note how the power supplies for the analog and digital parts are separated.

ESP-NOW does not implement fancy connection or high-level communication protocols. Addressing is based on the node's Ethernet MAC address, and a pairing step is required to make them talk to each other. Also, data packets are not guaranteed to arrive in order. For simple remote control applications, this all is fine.

The data rate of ESP-NOW is 1 Mbit/s by default (configurable), and a data packet can have a payload of up to 250 bytes. Together with header and checksum bytes, etc., this results in a maximum packet size of 255 bytes.

Let's Build a Walkie-Talkie

My objective was to create a walkie-talkie-like device or an intercom based on ESP-NOW. A quick glance at the specifications of the ESP32 shows that it integrates everything needed for this: an analog-to-digital converter (ADC), a digital-to-analog converter (DAC), lots of computing power, and, of course, all the radio stuff. In practice, however, things are a little less rosy.

The 12-bit wide ADC turns out to be rather slow, I measured a maximum sample rate of around 20 kHz. Somewhere online, it was mentioned that its analog bandwidth is only 6 kHz. The DAC is eight bits wide (but there are two), which limits the possible audio quality even more.

However, a walkie-talkie can get away with these numbers if the audio bandwidth is limited to the standard telephony bandwidth of 3.5 kHz. A sample rate of 8 kHz results in a data rate of (8,000 / 250) × 255 × 8 = 65,280 bit/s (remember, the maximum payload size is 250 bytes). This is way below the default rate of 1 Mbit/s. These specifications won't get us high-fidelity audio, but that is not our goal anyway. Intelligibility is more important.

The Circuit

To keep things simple, I used a one-transistor band-limited condenser microphone preamplifier as audio input and added a classic LM386-based amplifier as audio output. The schematic is shown in **Figure 3**. The input bandwidth is limited at the low end by C1 and C5, which are slightly under-dimensioned. The high end is limited by low-pass filters R4/C2 and R5/C3. Similar low-pass filters are placed at the DAC's output. The signal at the hot side of P1 should not be larger than 400 mV_{PP}.

As ESP32 module, I opted for the ESP32-PICO-KIT. There exist many other modules, but they do not all expose the DAC outputs on GPIO25 and GPIO26. Also, we need an ADC input. I used GPIO32 for this, which corresponds to ADC1, channel 4. The test point TP1 on GPIO26 (the second DAC output) is provided as a monitor output for the microphone signal. A push button on GPIO33 provides push-to-talk (PTT) functionality, and the LED on GPIO27 is the obligatory multifunction microcontroller-circuit LED.

Note how the power supply is split into an analog and a digital part. The reason for this is not to avoid high-speed digital switching noise coupling into the audio input, but to avoid a clicking sound in the output. Apparently, a task running on the ESP32 produces periodic power surges that can become audible when the circuit is not wired carefully.

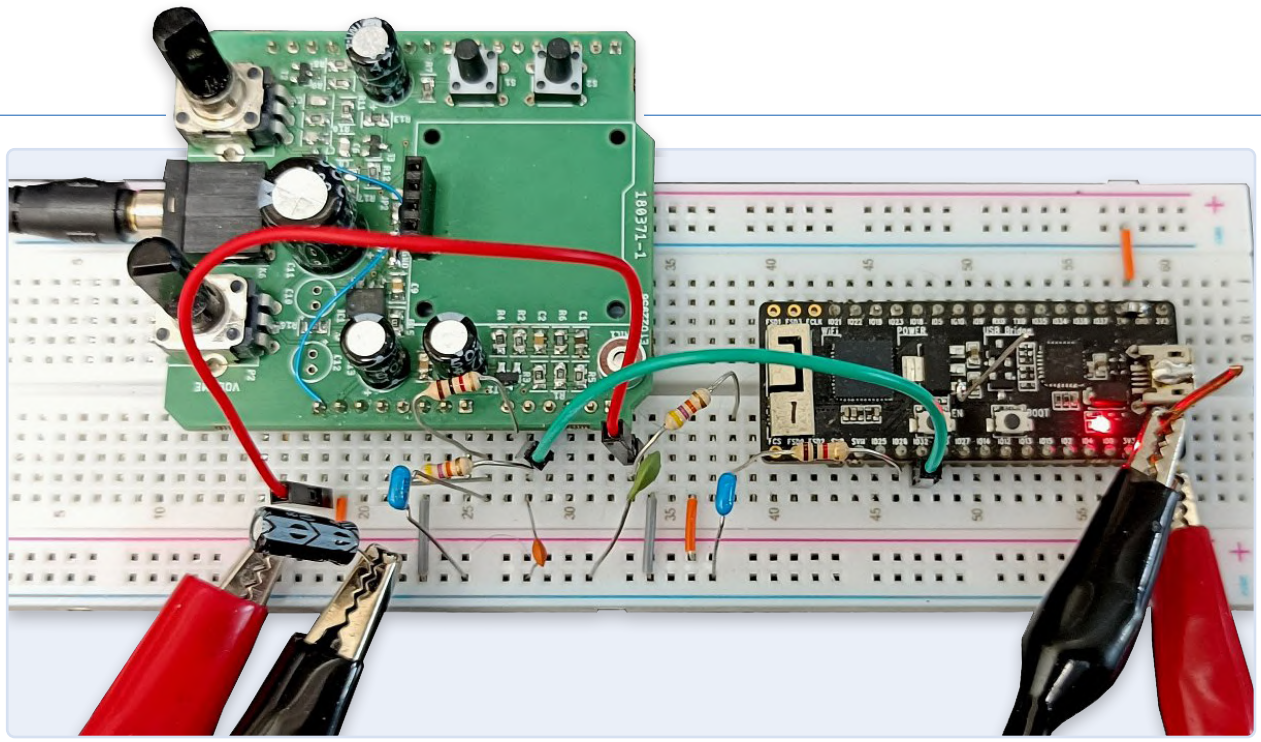


Figure 4: A proof-of-concept built on a breadboard with an ESP32-PICO-KIT and a slightly modified Elektor Snore Shield [2] for the input and output amplifiers. Note the two pairs of crocodile clips that provide the separate analog and digital power supplies.

The best way I found to avoid this is by using two separate power supplies (**Figure 4**). The ESP32 module must be treated as a component that needs a power supply (like the LM386), and not as a module that can also provide power to the rest of the circuit; in this application, it can't. Keep in mind that the LM386 has a power supply range from 4 V to 12 V.

C10 is optional and is only needed in some rare cases of early ESP32 modules that won't boot properly when they are not connected to a computer (or the like). As it happens, I have a few of these early modules, and so I included C10 in my design.

The Software

I based the program for the walkie-talkie on the *ESPNow_Basic_Master* example that comes with the Arduino ESP32 boards package from Espressif. After adapting it to my needs, I added audio sampling and playback to it. There are a few things that you may want to know about the program.

Audio sampling and playback is controlled by a timer interrupt running at 8 kHz. For sampling, the sample rate timer interrupt service routine (ISR) only raises a flag to signal that a new sample should be acquired. The `loop()` function polls this flag and takes the necessary actions. This is because the ADC should not be read inside an ISR when using the ADC API provided by Espressif. The `adc1_get_raw()` function used here calls all sorts of other functions that can do things over which you have no control. As the ESP32 software runs in a multitasking environment, ensuring thread safety therefore is important. When using Arduino for ESP32 programming, a lot of this is handled for you, but if you plan to port my program to the ESP-IDF, you may have to be more careful.

Audio playback is easy as the sample rate timer ISR simply writes a sample to the DAC if one is available. If not, it fixes the DAC output at half the ESP32 supply, i.e., 1.65 V. The only thing to be aware of here is that a so-called ping-pong buffer is used for streamlining digital audio reception (**Figure 5**). Such a buffer consists of two further buffers, one of which is being filled while the other is being read. This allows for overlapping. In theory, this should not happen as the sender and receiver use the same sample rate and timing logic, but in reality it does because of timing tolerances. A ping-pong or double buffer helps to avoid annoying clicks during playback. Note that out-of-order reception of data packets is not handled.

Pairing

The walkie-talkie firmware is a master-slave system. The master functions in Wi-Fi station (STA) mode, while a slave is in access-point (AP) mode. The master connects immediately to a slave when it detects one, and it can start sending data right away. However, when the master connects to the slave, this does not also connect the slave to the master. The slave cannot send data to the master and two-way operation is not possible (at least, I didn't succeed; if you know better, please let me know).

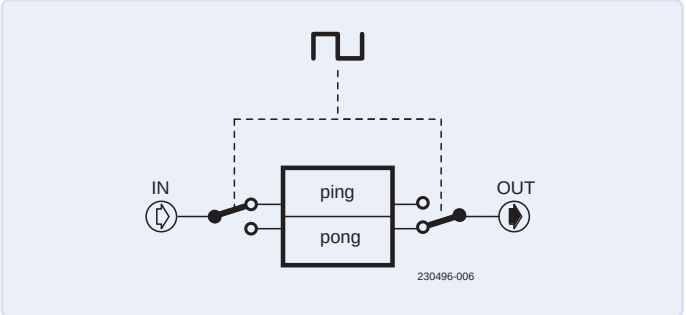


Figure 5: Double- or ping-pong buffering helps to avoid discontinuities in a data stream.

A way to make the slave connect to the master is by using the data reception callback. When data is received, the sender's address is passed to this function together with the data. Therefore, as soon as the slave receives something, it can connect to the sender of the data. For this, I used the same functions and procedure as used by the master to connect to the slave. There is, however, one subtlety that is not very well (if at all) documented: The slave must set its Wi-Fi interface field to `ESP_IF_WIFI_AP`, or it will not work. This field defaults to `ESP_IF_WIFI_STA` as needed by the master, so the program(mer) doesn't have to set it explicitly. As a result, the field doesn't appear anywhere in the example programs, leaving the user unaware of its existence.

Push-to-Talk

When ESP-NOW is streaming continuously, the MCU gets pretty hot. In the walkie-talkie application there is no reason to stream continuously, and so I added a push-to-talk (a.k.a. PTT) button (S1). Press this button and keep it pressed while talking. If the sender is paired with the receiver, the LED will light up. On the receiver-side, the LED will also light up, indicating that a call is coming in. To avoid audio feedback, the audio output on the sender's side is muted when the PTT button is being pressed. Therefore, even though communication is in principle full-duplex, the two peers should not try to talk both at the same time. This is a great opportunity to incorporate "roger" and "over" in your sentences.

One Program Fits All

The program consists of one Arduino *.ino* file ("sketch"). Besides the Espressif ESP32 boards package, no other libraries are required. The walkie-talkie needs a master and a slave device. To compile the program for a master device, comment out line 12, which says `NODE_TYPE_SLAVE`. For the slave device, this macro must be defined. You can reconfigure some other settings too if you like. It is also possible to compile without audio input (`AUDIO_SOURCE`) and/or output (`AUDIO_SINK`) support. This is practical for debugging or for an application that only needs one-way communication.
The source code can be downloaded from [3].

Higher Fidelity?

It shouldn't be too complicated to stream high-quality audio data over ESP-NOW if, instead of using the simple microphone amplifier and the ESP32's built-in ADC and DAC, you switch to I^2S. This makes the circuit and program a bit more complex, but would allow — at least in theory — for streaming 16-bit audio data at a 48 kHz sample rate. However, the possible out-of-order reception of packets must be handled properly. But hey, wasn't Bluetooth designed to do this?

Range Test

To see if ESP-NOW allows for long-range communication, I wrote a simple program to send a ping message to the slave once per second. The slave was nothing more than an ESP32-PICO-KIT with an LED connected to GPIO27, powered bu a USB power bank. Every time a ping is received, the LED flashes briefly (100 ms).

With the transmitter placed outside at 1 m above the ground, I obtained a line-of-sight (LOS) communication distance of about 150 m. At this distance, reception became intermittent, and the slave had to be held up high (approx. 2 m above the ground). This situation can probably be improved by carefully positioning (and designing) the two peers.

230496-01

Questions or Comments?

Do you have technical questions or comments about his article? Email the author at clemens.valens@elektor.com or contact Elektor at editor@elektor.com.

About the Author

After a career in marine and industrial electronics, Clemens Valens started working for Elektor in 2008 as editor-in-chief of Elektor France. He has held different positions since and recently moved to the product developmentteamt. His main interests include signal processing and sound generation.

Related Products

- **ESP32-PICO-KIT V4**
  www.elektor.com/18423
- **Elektor ESP32 Smart Kit Bundle**
  www.elektor.com/19033

WEB LINKS

[1] More about ESP-NOW: https://espressif.com/en/solutions/low-power-solutions/esp-now
[2] The Elektor Snore Shield: https://elektormagazine.com/180481-01
[3] Downloads for this article: https://elektormagazine.com/230496-01

From Idea to Circuit with the ESP32-S3

A Guide to Prototyping with Espressif Chips

By Liu Jing Hui, Espressif

Many products and devices — especially low-volume products and one-offs — that contain Espressif chips are initially prototyped on a development board. But engineers usually migrate their design to use a module or a bare chip if they are working on higher-volume products or if they require a product that looks better. There are some things to pay attention to when doing so. Let's review the most common pitfalls.

We'll take the ESP32-S3 as an example to introduce how to design with Espressif chips. This chip is a highly integrated, low-power, 2.4-GHz Wi-Fi + Bluetooth LE (5) System-on-Chip (SoC) solution. It contains two fast CPU cores, RAM that can be expanded externally, and peripherals that should be enough for a wide range of applications. Relevant here: It also integrates advanced calibration circuitry that compensates for radio imperfections. This makes it easier to get it working in your design and eliminates the need for specialized testing equipment. The ESP32-S3 is an ideal choice for a wide variety of application scenarios related to AI and Artificial Intelligence of Things (AIoT). Note that you can purchase it both in the form of a "bare" chip as well as in the form of a module (e.g., the ESP32-S3-WROOM-1) that integrates the ESP32-S3 with the needed crystal, flash, RF circuitry and either a trace antenna or a connector for an external antenna. We will include details for both using the "bare" chip as well as using a module here.

If you want to design this chip into your design, there are four main things to pay attention to:

- Circuit design and schematic creation
- PCB layout design
- RF and crystal tuning process
- Download firmware and troubleshooting

Circuit Design and Schematic Creation

To get started, you may want to go through the documentation of the chip and/or module in question. We provide various documents online at [1]. Documents like the Datasheet, Hardware Design Guidelines and Module Reference Designs are written to show what the chip needs in order to perform optimally.

If you are using a chip, we'll show an ESP32-S3 chip plus all the components it generally requires in **Figure 1**. Note that unlike chips that do not integrate a radio, decoupling is pretty important here. Specifically, we verified the number and values of decoupling caps on each power pin, so it's important to use these and not leave off capacitors. Especially at pin 2 and 3, there must be a CLC filter circuit (C8, L1 and C9 in the schematic). Generally, for most Espressif chips, there are power pins related to RF power, so a CLC/CCL filter is always required to suppress high harmonics.

Another critical point concerning power is supply current for ESP32-S3 is at least 500 mA. Basically, this is universal, applying to all Espressif chips released until now. (An exception is the ESP32-H2, which only requires at least 350 mA.) Not having enough power supply current generally leads to brownout resets and other weirdness, most often while initializing Wi-Fi.

The ESP32-S3 requires a main crystal as the clock source for the whole system. Please add a series inductor in the XTAL_P line to help suppress harmonics due to crystal. The values of the load capacitors C1 and C4 depend on the crystal, as well as the parasitic impedance of the traces and pads. Adafruit has a good tutorial on how to calculate an initial value for these [2], although for the best range and certification

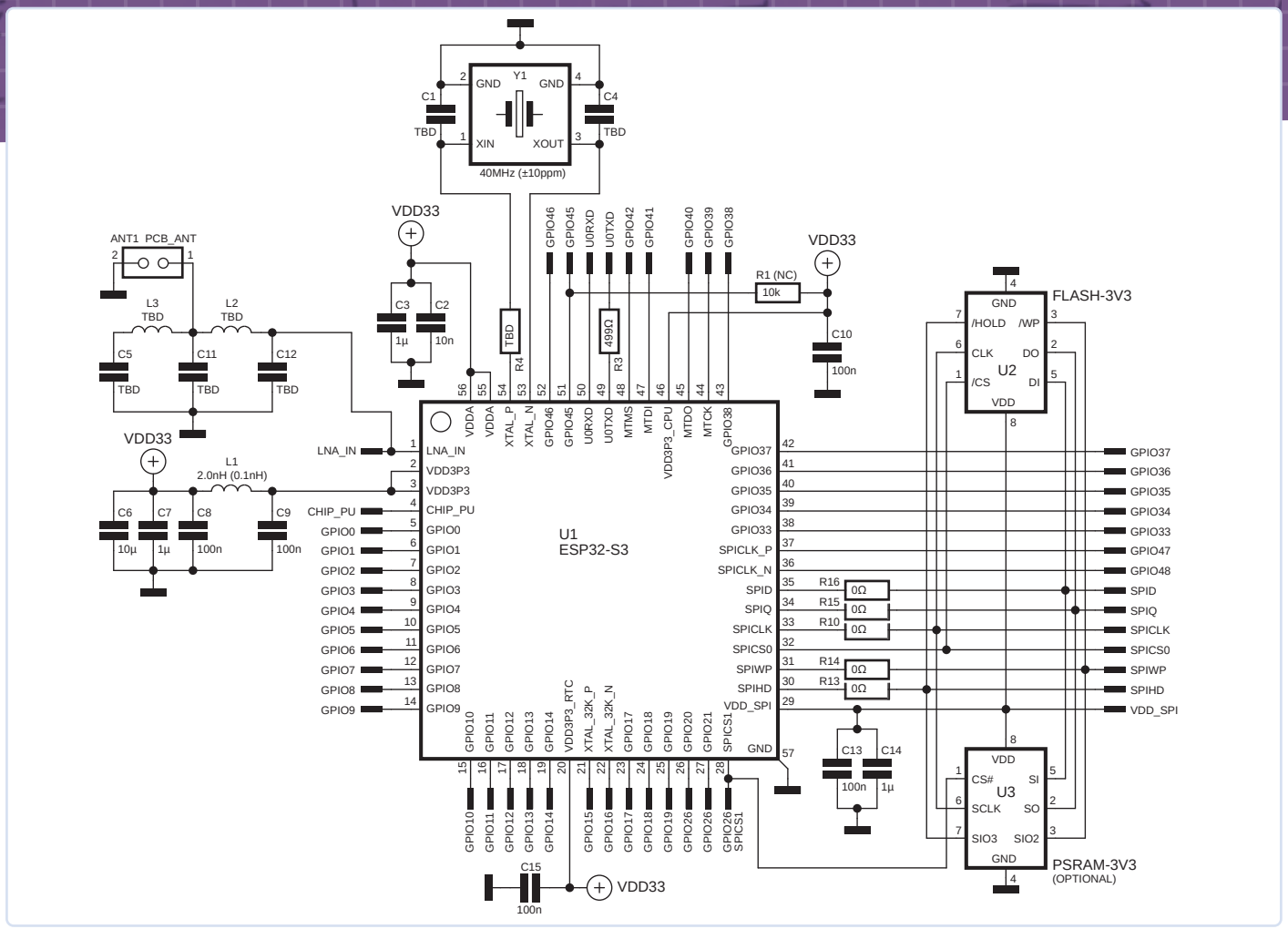


Figure 1: ESP32-S3 chip plus all the components it generally requires. Decoupling is pretty important here.

compliance, you should measure and tweak those once you have the physical PCB. See later in this article for more info.

An ESP32-S3 always needs flash memory to store its program in, and it can optionally be connected to PSRAM if the extra memory is required. Note that some variations of the ESP32-S3 already include the flash and/or PSRAM in the package, in which case you can leave out U2 and/or U3. This means you don't have to have the footprint for these chips, which really helps reduce the size of PCB boards. Do keep the decoupling capacitors on VDD_SPI in this case, as they will still be needed to decouple the internal flash/PSRAM.

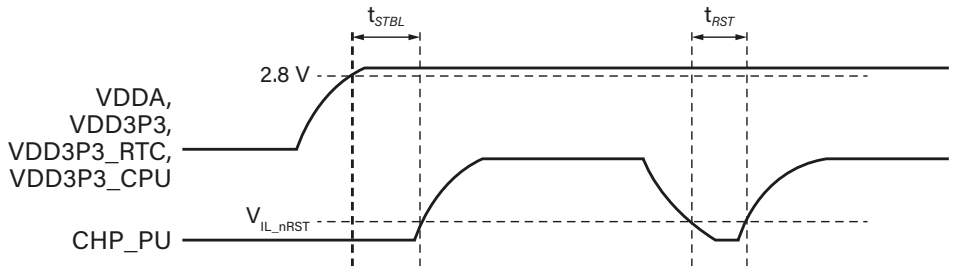
To get the best radio range and to make sure the device complies with EMC certification requirements, you need to make sure the impedances in the path from the ESP32-S3 to the antenna match. Generally, the transmission line (trace) between the ESP32 and the antenna has a 50 Ω impedance, but the ESP32-S3 LNA_IN pad does not and the antenna you use might not either. In other words: connecting the ESP32 to the transmission line requires a π-CLC matching circuit close to the ESP32-S3 to impedance match it to 50 Ω. Connecting an antenna to the trace may also require a CLC matching circuit near it, but you can omit this if you can guarantee, e.g. by simulation, that the antenna impedance is 50 Ω. Note that in the schematic, the CLC network (consisting of C11, L2 and C12) does not have values assigned. This is because the values needed depend on the PCB design and material and need to be determined after the design phase. See later in this article for the details.

If you are using a module, you do not have to worry about any of the above, as the module already includes the correct components and values.

If you are using a module or a chip: The CHIP_PU pin (marked as EN on the module) works as both an enable and reset pin; it should not be left floating. In order to reliably start the ESP32-S3 chip, the CHIP_PU pin should only be activated after the power supply is stable. (Refer to the "Timing Parameters for Power-up and Reset" text box.) Generally, an RC circuit is added to this pin to generate a delay. Some products need to work in a scenario where the power ramp is very slow, or requires frequent power-on and power-off, or the power supply is unstable. (This can happen, for example, if an ESP32-S3 is powered from a solar panel.) In this case, only using an RC circuit may not meet the power-on/off sequence timings; this can lead to the chip booting up unsuccessfully. In this case, we suggest using other ways to meet the requirements, such as using an external power supervisor chip or watchdog chip. If your design allows it, you can also control the CHIP_PU pin by another MCU, or simply add a reset button.

The final step to run ESP32-S3 successfully is to pay attention to strapping pins. At each startup or reset, Espressif chips require some initial configuration parameters, such as in which boot mode to start the chip, the voltage of flash memory, etc. These parameters are selected via the strapping pins. After reset, the strapping pins operate as regular IO pins. As usual, you can find the particular details for this in the chips datasheet.

Timing Parameters for Power-up and Reset



Visualization of timing parameters for power-up and reset. (Source [5])

Parameter	Description	Min (µs)
t_{STBL}	Time reserved for the 3.3 V rails to stabilize before the CHIP_PU pin is pulled high to activate the chip	50
t_{RST}	Time reserved for CHIP_PU to stay below V_{IL_nRST} to reset the chip	50

With all of this taken care of, our ESP32-S3 can successfully boot. However, you probably want to connect it to other chips, sensors and actuators as well. What GPIOs can you use for this? Good news: All Espressif chips have a function called the GPIO Matrix. It makes it really easy to hook things up to the ESP32-S3, as it allows you to map ANY available GPIO to any signal of most general peripherals (e.g., I2C, SPI, SDIO, etc.). Of course, there are still some peripherals that use fixed GPIOs (e.g., ADC). Refer to the Chip Datasheet for more information if needed. As a general reminder, remember to read technical documents first — it's better to re-read information you already know than to have to re-spin a PCB because of a preventable error.

PCB Layout Design

If you are using a chip, you should now have a perfect schematic waiting for layout. For Wi-Fi and Bluetooth products, the PCB layout is critical for qualified RF performance. Even if you only use an Espressif chip as MCU, without adding an antenna or making use of the radio, the guidelines below will result in a stable system and increased reliability. Note that all the exact details can be found in the Hardware Design Guidelines. Here we will just generalize the important points.

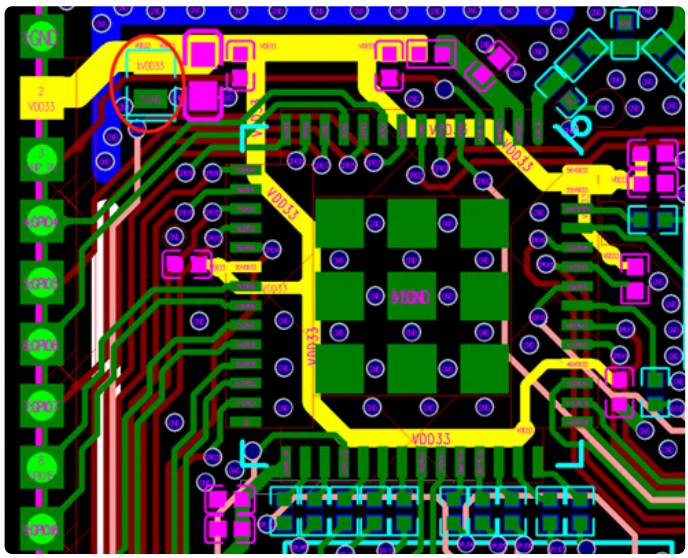


Figure 2: Each decoupling cap should be as close as possible to power pins.

Let's start with the layer stack-up. Four or more layers are recommended since, that way, we can have an unbroken and large ground plane adjacent to the chip layer as a reference ground layer. If you really want to use two layers, do not forget to make sure you still have a good ground plane.

The second important point is the routing of the power traces. We suggest routing them in a star-way on an inner layer. Each decoupling cap and CLC filter circuit should be as close as possible to power pins (**Figure 2**).

The third point is to take care of your RF traces. You should tell the factory to have a controlled 50 Ω impedance for all RF traces. Doing this starts at the PCB layout stage, where you are supposed to calculate the proper trace width and gap to ground based on your stack-up. The factory then may adjust these slightly. If the components are in the 0201 SMD package size, please use a stub in the PCB design of the RF matching circuit near the chip.

Next is the crystal. The series inductor should be very close to the ESP32-S3 chip, and the two load capacitors should be placed on two sides of the crystal (**Figure 3**). Enough ground and dense via stitching are very important for crystal.

A final thing to pay attention to are the flash and UART traces, as well as traces that go to other peripheral GPIO pads. Those traces may generate high-frequency harmonics, so we'd advise to route them on the inner layers and surround them with ground as much as possible.

If you are using a module, the critical point is how to place it to obtain the best RF performance. The following guidelines only cover modules with a PCB antenna; if you use a module with an external antenna, you have more flexibility in where to place it on the PCB.

As you see in **Figure 4**, we suggest you place the module on a corner of the board. Specifically, it's best to place it so that the feedpoint is closest to an edge as well. (Also note that not all module types have the feedpoint on the same side; again, check the datasheet before designing the PCB.) We recommend the antenna to be completely outside the board. In case this is not possible, please ensure that there is clearance under and at least 15 mm around the antenna: this means that there should be no traces, planes, vias or components there (**Figure 5**).

While it is pretty easy to solder most connections of our Wroom modules with a standard soldering iron, there is a ground connection underneath that is normally inaccessible to solder by hand. If you have the capability to do so (e.g., using a reflow oven or a hot air rework station), we'd advise you to connect it to a ground plane, as it helps dissipate heat and give better grounding for the radio, but in normal circumstances the module will also work without it.

RF and Crystal Tuning Process

If you are using a chip: Given you followed all the above points and got a working PCB back from the factory, for best reliability and to pass EMC certification, you'll unfortunately need some pretty professional instruments to tune RF the matching values and crystal caps. In case you have access to that (or in case cheap products like the MicroVNA get good enough to use for this), we'll explain what this procedure comes down to.

You probably picked the values of the load capacitors according to what the guesstimated parasitic capacitance of the traces is. Now is the time to fine-tune this: because the main 2.4 GHz is derived from this crystal, optimizing how it works will get you a better range and lower harmonics. We'll optimize this by using the ESP RF Test Tool you can download at our site [3].

1. Select TX tone mode using the Certification and Test Tool.
2. Observe the 2.4 GHz signal with a radio communication analyzer or a spectrum analyzer and demodulate it to obtain the actual frequency offset.
3. Adjust the frequency offset to be within ±10 ppm (recommended) by adjusting the external load capacitance.
   - When the center frequency offset is positive, it means that the equivalent load capacitance is too small, and the external load capacitance needs to be increased.
   - When the center frequency offset is negative, it means the equivalent load capacitance is too large, and the external load capacitance needs to be reduced.
   - The external load capacitances at the two sides are usually equal, but in special cases, they may have slightly different values.

With the crystal load capacitors adjusted, we move on to the antenna matching network. In the matching circuit, we define the port near the chip as Port 1 and the port near the antenna as Port 2. S11 describes the ratio of the signal power reflected back from Port 1 to the input signal power, and S21 is used to describe the transmission loss of signal from Port 1 to Port 2. For the ESP32-S3 series of chips, if S11

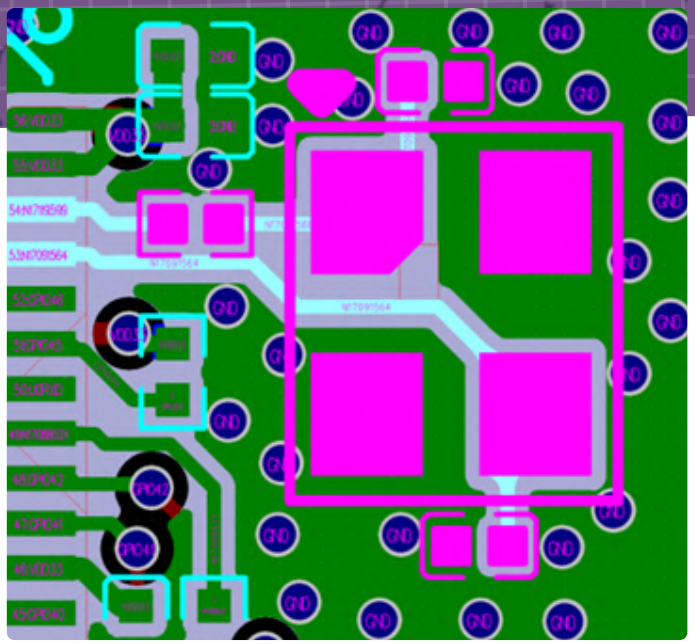


Figure 3: The two load capacitors should be placed on two sides of the crystal.

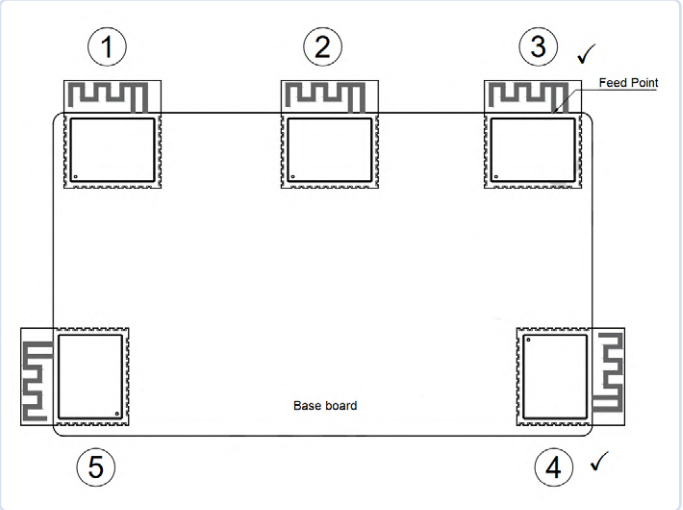


Figure 4: Positioning a module on a base board. (Source [6])

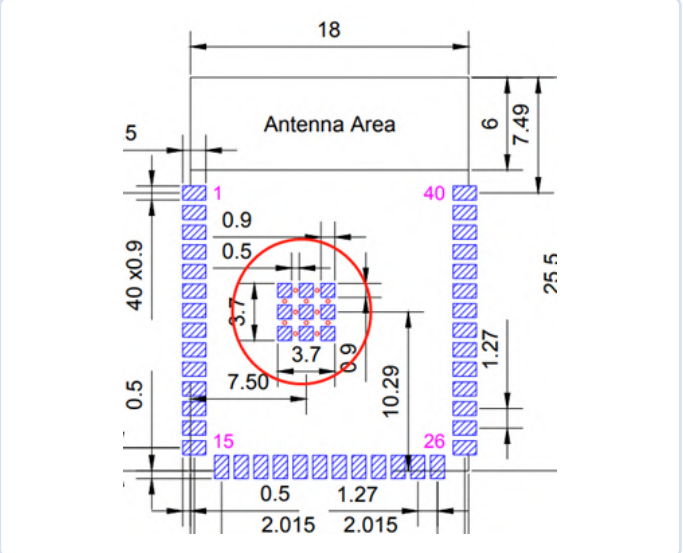


Figure 5: ESP32-S3-WROOM-1 recommended PCB land pattern. (Source: [6])

is less than or equal to -10 dB and S21 is less than or equal to -35 dB when transmitting 4.8 GHz and 7.2 GHz signals, the matching circuit can satisfy transmission requirements.

Connect the two ends of the matching circuit to the network analyzer (refer to the "Matching Circuit" text box), and test its signal reflection parameter S11 and transmission parameter S21. Adjust the values of the components in the circuit until S11 and S21 meet the requirements. If your PCB design of the chip strictly follows the PCB design guidelines and the antenna is well-designed, you can refer to the value ranges in the text box to debug the matching circuit.

So, what should you do when you want to design an ESP32-S3 chip into your board, but you do not have the fancy equipment to do this properly? Well, if you followed all the other points in this article, you may be able to get away with the guesstimated load capacitor values and no CLC network. (If you designed one into your PCB anyway, you can leave off C11/C12 and change L2 to a zero-ohm resistor. By doing this, you can still add a CLC network later without adjusting the PCB design.) In our experience, not having a tuned impedance matching network does not affect the range too much; its main use is to reduce the devices' EMV (Error Magnitude Vector) which is needed to pass certification requirements.

If you are using a module: No need for any of the above, your module comes with all its components pre-tuned for optimal performance.

Firmware Downloading and Trouble Shooting

If you are using either a module or a chip: Before your ESP32-S3 will actually do anything, you will need to load some firmware into its flash (called *programming* or *downloading firmware*). For all Espressif chips, the download process is: reset the chip into download mode — download — reset and run the chip in *SPI Boot* mode.

You can program Espressif chips generally in two ways: all Espressif chips will support downloading firmware over the UART, but the newer chips generally also have a USB peripheral which allows you to download firmware directly over a USB connection to your computer. Below are the detailed steps for ESP32-S3.

To download via UART:

1. Before the download, make sure to set the chip or module to *Download Boot* mode, i.e. strapping pin GPIO0 (pulled up by default) is pulled low and pin GPIO46 (pulled down by default) is left floating or pulled low. Configure pin GPIO45 appropriately according to the strapping pin table, or leave it floating in case you're using a module.
2. Power on the chip or module and check whether it has entered *UART Download* mode via UART0 serial port. If the log shows "waiting for download", the chip or module has entered *Download Boot* mode.
3. Download your firmware into flash via UART. You can use whatever programming option your programming environment (Arduino, ESP-IDF, VSCode, ...) gives you; or you can use the standalone Flash Download Tool for this.
4. After the firmware has been downloaded, pull IO0 high or leave it floating to make sure that the chip or module enters *SPI Boot* mode.
5. Power on the module again. The chip will read and execute the new firmware during initialization.

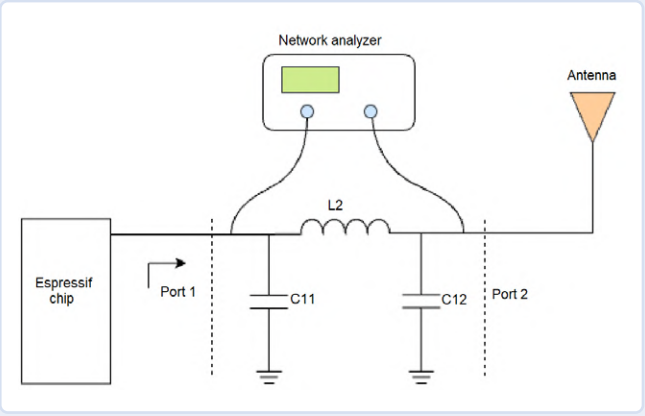
Note that most development boards have a schematic that allows software to automatically get the ESP32 chip into and out of download mode. Refer to, for example, the ESP32-S3-Devkit-C-1 schematics if you want to know how to implement this [4].

To download via USB:

1. In most cases, when the chip has working firmware, the flashing tool should be able to get the chip into *USB download* mode via the USB connection. In this case you can proceed to the next step. If not, you need to put the chip or module into *Download Boot* mode, i.e., make sure strapping pin GPIO0 (pulled up by default) is pulled low and pin GPIO46 (pulled down by default) is left floating or pulled low. Configure pin GPIO45 appropriately according to its strapping pin table, or leave floating if using a module.
2. Power on the chip or module and check whether it has entered *UART Download* mode via USB serial port. If the log shows *waiting for download*, the chip or module has entered *Download Boot* mode.
3. Download your firmware into flash via UART. You can use whatever programming option your programming environment (Arduino, ESP-IDF, VSCode, ...) gives you or you can use the standalone Flash Download Tool for this.
4. After the firmware has been downloaded and you entered *USB download* mode automatically, you're done. If not, pull IO0 high or leave it floating to make sure that the chip or module enters *SPI Boot* mode. Reset or power cycle the chip or module, and it will read and execute the new firmware during initialization.

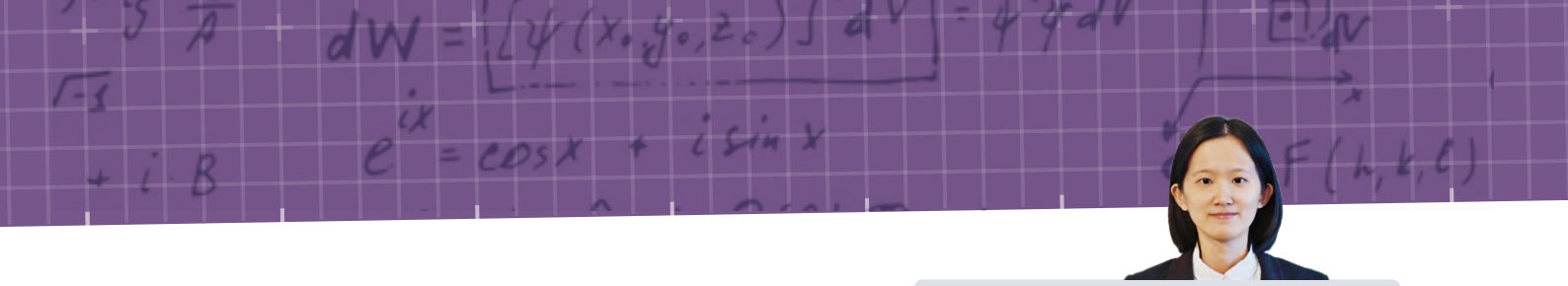
If you fail to download firmware via UART, check UART0 log through a serial terminal tool. The ESP32-S3 startup message will tell you the real latching value of the strapping pins, which allows you to figure out which strapping pin value is wrong. The value after `boot:` shows the

Matching Circuit



Source [7]

Reference Designator	Recommended Value	Serial No.
C11	1.2 ~ 1.8 pF	GRM0335C1H1RXBA01D
L2	2.4 ~ 3.0 nH	LQP03TN2NXB02D
C12	1.8 ~ 1.2 pF	GRM0335C1H1RXBA01D



About the Author

Liu Jinghui, after completing her graduate studies in electronic engineering, joined Espressif and has been engaged in solving hardware problems. For five years, she has served as the window for Espressif products and customers' usage and has a clear understanding of product hardware features and problems that may arise when customers use them.

strapping pin values in hexadecimal: bit 2 = GPIO46, bit 3 = GPIO0, bit 4 = GPIO45, bit 5 = GPIO3:

```
ets Jun 8 2016 00:22:57
rst:0x1 (POWERON_RESET),boot:0x3 (DOWNLOAD_BOOT(UART0/
UART1/SDIO_REI_REO_V2))
```

If you fail to recognize USB device or USB connection is unstable, try to run into download mode first, then download. Also remember that on the computer side, only one program should have the serial port open at the same time: trying to flash while also having a serial terminal open on the same port will fail.

Note that while getting into download mode over USB is generally reliable, but there are a few scenarios in which it can fail:

- USB PHY is disabled by the application;
- USB port is reconfigured for other tasks, e.g., USB host, USB standard device;
- USB GPIOs are reconfigured, e.g. as outputs for LEDC, SPI, or as generic GPIOs.

In this case, it is necessary to manually enter download mode. As such, at least while doing firmware development we'd suggest always having some way (buttons, jumpers, ...) to set the strapping pins to force a download mode boot.

Another thing to note: Sometimes people find that their board will not boot up correctly (that is, start executing their program in flash) unless they manually reset it. This can happen when a capacitor is placed at a strapping pin (e.g., GPIO0 on the ESP32-S3). A capacitor like this sometimes is placed to debounce a button, but when the device is powered up, it leads to the level on GPIO0 rising slowly, which the ESP32-S3 will latch as a low value.

Modules Do the Magic

As you can see in this article, while making something that "just works" is certainly possible, designing a board containing an ESP32-S3 in such a way that performance is optimized and the device can be successfully certified is not trivial and needs some in-depth knowledge and tools. As such, if you have the space to accommodate one, we would suggest using a module instead; all the tricky RF magic is already done by Espressif which makes integrating one into your design a lot easier.

If all this still sounds too complicated, as stated at the beginning of the article, Espressif also has a wide range of development boards, from simple ones that fan out all GPIOs to easy to use headers like the ESP32-S3-DevkitC-1, to all-integrated voice AI powerhouses like the ESP32-S3-Box-3 and the camera-enabled ESP32-S3-Eye. Generally, the schematics and board designs for these development boards are available, so even if you aren't planning on using them in your product, they can be a great place to get your design started.

230563-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the author at liujinghui@espressif.com or the Elektor editorial team at editor@elektor.com.

Related Products

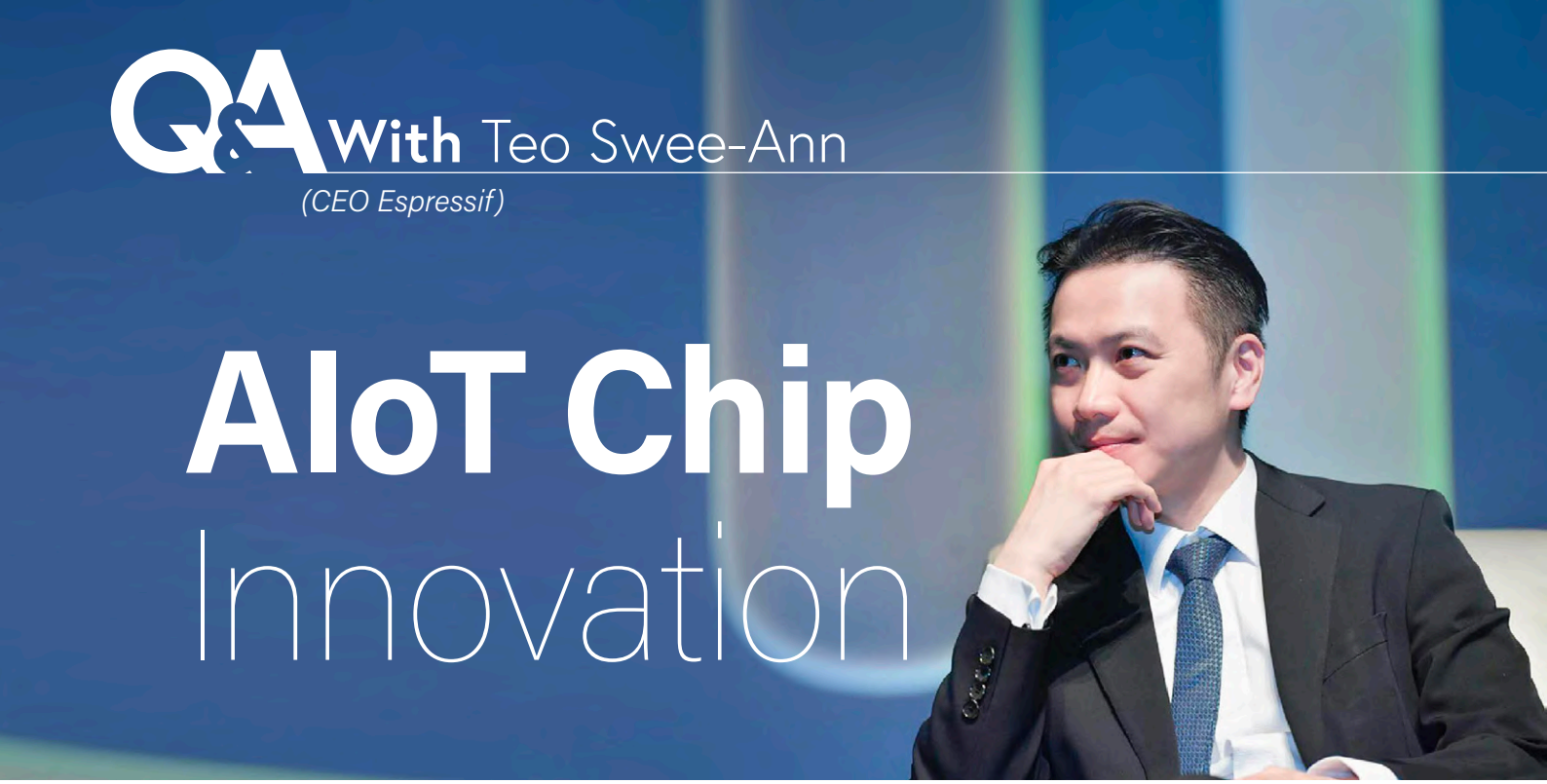
- **ESP32-S3-WROOM-1**
  www.elektor.com/20696

WEB LINKS

[1] Technical Documents Espressif:
https://www.espressif.com/en/support/documents/technical-documents?keys=&field_type_tid%5B%5D=842
[2] Adafruit, "Choosing the Right Crystal and Caps for your Design," 2012:
https://blog.adafruit.com/2012/01/24/choosing-the-right-crystal-and-caps-for-your-design/
[3] ESP RF Test Tool: https://www.espressif.com/en/support/download/other-tools
[4] ESP32-S3-DevKitC-1 Schematic: https://dl.espressif.com/dl/schematics/SCH_ESP32-S3-DevKitC-1_V1.1_20221130.pdf
[5] ESP32 Series Datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
[6] ESP32-H2-Series Hardware Design Guidelines:
https://www.espressif.com/sites/default/files/documentation/esp32-h2_hardware_design_guidelines_en.pdf
[7] ESP32-S3-WROOM-1 Datasheet:
https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf

Q&A with Teo Swee-Ann

(CEO Espressif)



AIoT Chip Innovation

An Interview With Espressif CEO Teo Swee-Ann

What does it take to bring an innovative electronic solution to the market at a disruptive speed? Teo Swee-Ann, CEO of Espressif, shares his insights. We learn about his career path and how he came to design and deploy popular chipsets, modules, and AIoT solutions.

Elektor: When did you first become interested in electronics? Were you inspired by a relative or a teacher? Or, perhaps you became curious while reverse-engineering things and tinkering with electronic products?

Teo Swee-Ann: My interest in electronics began when I was quite young. Growing up, I was always fascinated by gadgets and machines. We did the usual things, such as taking apart radios and TVs, trying to understand how they worked. It wasn't so much about reverse engineering at that time, but rather a sheer curiosity to peek behind the curtains. As usual, taking things apart didn't answer any questions, but made us even more curious.

No one in my immediate family was deeply involved in electronics, so my inspiration wasn't directly from a relative. But, I do seem to have a knack for software and mathematics. My university theses were about computational electromagnetics. I invented a couple of techniques that were useful for computing very efficiently — Green's functions for layered media. It turns out that this plus a technique called "partial elements equivalent circuit" could be useful for the characterization of inductors in-chip, in particular, to account for substrate losses. So, I think that's how I found a job in the chip industry. When I was given the chance to work on circuit design, I immediately got a few classic IC design books by Paul Gray, Philip Allen, Ken Martin, and practically memorized them. There was no looking back thereafter.

Elektor: Were engineering and entrepreneurship always on the cards for you? When you first attended university, did you have thoughts about launching and running a business?

Teo Swee-Ann: I didn't think about the business side of things. I am more of an idealist; I believe that it's knowledge that will transform the world — not business. But, of course, my later experience taught me that sometimes one must do more, adapt, and create a business platform for the technology as well.

Elektor: Tell us about your technical interests. What did you focus on for your Master's degree at the National University of Singapore?

Teo Swee-Ann: In my university days, I preferred mathematics and software and shunned anything that involved hardware. So, my Master's thesis was on the characterization of microwave passives in layer media using computation electromagnetism. A layer media could be a thing such as a PCB or an IC, which is fabricated layer by layer, and its material will vary between layers. The objective was to compute the high-frequency characteristics of passive microwave components such as inductors in such media.

We faced many problems then: the difficulty of creating a single Green's function to represent both the near field and far field, how to extract the poles of the functions, and how to do the Sommerfeld

transforms of the functions from the frequency domain to spatial. So, my work was divided into two parts.

The first part was to show how we could create a single function to represent both the near and far field and have an efficient computation method for computing their Sommerfeld transforms (something like Laplace transform but with a Bessel or Hankel kernel). We know that these functions have singularities and branch cuts. So, it's like treading upon a mathematical minefield with some fences.

The crux was to first extract the singularities of these functions and second, the creation of "synthetic poles" that have closed-form solutions in the later computation, to help accelerate the convergence of the computation (a problem that arose because of the inclusion of the poles for singularity extraction). The remaining small portion of functions that were unaccounted for will then be computable and have rapid convergence.

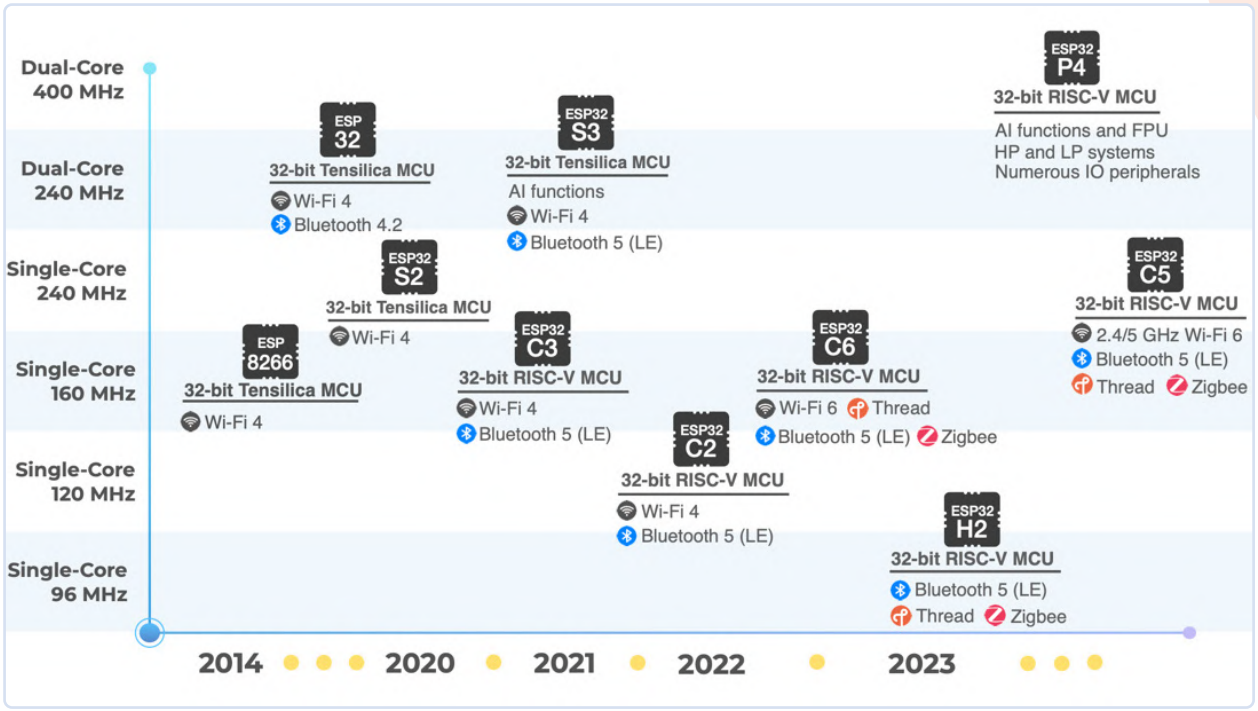
Lastly, the remaining problem was about how we could extract the singularities in the complex domain. I created an alternative technique, which I name "Cauchy Tomography" to extract these singularities, by using the Cauchy residue theorem and recovering their locations via the Generalized Pencil of Function Technique (GPOF). In a serendipitous turn of events, I still sometimes use this technique today in circuit design to compute the transfer function of a system based on its transient response. It turns out that the transient settling response is a kind of time domain integration output of the system frequency response. By converting between the two domains, one could sort of see how the system poles would create a certain kind of transient response and vice versa.

Elektor: Your response to Elektor's invitation to be the next guest editor was a firm and immediate "yes." When did you get to know Elektor? Did Elektor influence or affect you and Espressif in any way?

Teo Swee-Ann: Back in 2015, we first came across Elektor. What stood out about the magazine was its distinctive ability to identify and highlight emerging trends in the industry. In fact, I believe Elektor was among the pioneers in featuring Espressif's chips, starting notably with the ESP8266, thereby playing a significant role in popularizing these chips. We regularly turn to Elektor as a source of inspiration, always eager to gauge the evolving needs and interests of its readership. Moreover, an invaluable role that Elektor fulfills is serving as a knowledge hub for those aspiring to enter the electronics sector. It underscores a shared obligation that we in the industry feel: To disseminate our knowledge and, in so doing, ensure that the realm of electronics continues to draw in and nurture fresh talent.

Elektor: What upsides and possible downsides does your engineering background bring to your current role as CEO of Espressif?

Teo Swee-Ann: I don't see much downside to it. I love being at the front line, and I still work on circuits. Working alongside the team gives me a better perspective of what issues we're



"One of our initial decisions was that an IoT chip should ideally have 600 DMIPS. This performance level offers a balance between power, surpassing many MCUs, and cost-effectiveness, thanks to its dual-core architecture and efficient cache system."

facing and the difficulties of building each feature. This enables me to better allocate resources, identify key issues, communicate with the team, and build consensus.

Elektor: What are your interests outside of engineering and electronics?

Teo Swee-Ann: I play a lot of music while I think about engineering problems. It somehow unblocks the brain. I used to play the classical guitar a lot, and recently, I picked up the cello. I am now going through the Bach cello suites bit by bit and working on improving my intonation.

Elektor: You launched Espressif in 2008. Your first product was the ESP8086, which came out in 2013. Was that product the company's main focus during those five years, or were you also providing other services?

Teo Swee-Ann: During the first few years, it was a very small operation. I worked on software to create a language to describe circuits. Hence, the name "Espressif," which meant to express our circuit ideas. However, selling software is hard, and we pivoted to being a consultancy company, building analog IPs for our customers. Eventually, we were inspired by the concept of an impending technological singularity and decided that we should build IoT chips, which would act as the nervous system for such an intelligent system.

Elektor: In 2014, Espressif launched its first IoT SoC, the ESP8266EX. What sort of engineering challenges were you targeting with that solution?

Teo Swee-Ann: Around that time, there were floating ideas around of creating a low-cost Wi-Fi chip. We decided that we could take on this challenge by integrating all the RF external components into the chip. Historically, you'd see modules cluttered with upwards of 50 to 100 components. We revolutionized this design by incorporating the balun, antenna switch, PA, and LNA directly into the chip. The primary challenge was managing the power amplifier's peak power of 27 dBm. Without a sufficiently robust switch, the LNA risked being damaged.

We also changed the way power amplifiers were driven internally. Instead of inductors, we used baluns, which improved the stability of the system. Also, in the past, most Wi-Fi chips were loading the software images into the memory in a rather static fashion. We adopted a cache design so that software could be continuously read from the flash instead. It would be much slower, but it worked. We also put much of the logic of the system into the software, rather than hardware. This helped in the development.

The culmination of these incremental innovations led to the creation of a highly compact design. Our final product was at a quarter to one-third of the cost of what our competitors were offering. Today, we take pride in acknowledging that many of the techniques we pioneered are now standard practices in the industry, contributing significantly to reducing the costs associated with IoT chips.

Elektor: 2016 was an important year for Espressif. Tell us about the development of the ESP32 and its release.

Teo Swee-Ann: Before the introduction of ESP32, our approach was somewhat spontaneous and less structured. The ESP32 marked a significant shift in our mindset, leading us to approach marketing more systematically. One of our initial decisions was that an IoT chip should ideally have 600 DMIPS. This performance level offers a balance between power, surpassing many MCUs, and cost-effectiveness, thanks to its dual-core architecture and efficient cache system. The journey to shape the ESP32 involved numerous discussions, deliberations, and revisions.

It was during this phase that we formally established our team structure, refined our work process, and aligned our work processes with the product's design and specifications. This was a departure from our previous, less formal, "garage-style" approach. Concurrently with the ESP32's development, our software team launched ESP-IDF. Decisions, such as those regarding our open-source policies, were solidified during this period, setting a robust foundation for future advancements. To summarize, the ESP32's evolution underscored the increasing importance of software, rather than focusing solely on hardware. This transition was further enriched by the active engagement and invaluable feedback from our dedicated developer community.

Elektor: Tell us about Espressif's Matter-compatible solutions. Matter seems to be something of a priority in 2023 and moving into 2024, correct?

Teo Swee-Ann: So far, the smart home industry hasn't reached its potential, and it's still hard for consumers to use the products

effectively. The use cases are limited and the experience that it offers is fragmented. Matter attempts to bring a change by making the devices talk the same language, and it is very hopeful to see most of the major players coming together to solve these problems. Most importantly, we are seeing positive responses from the customers. Hence, it is a natural priority for us.

Espressif's approach to building Matter solutions is not limited to hardware. While we have different chips supporting Matter over Thread and Wi-Fi protocols, we go beyond by understanding specific customer problems in Matter adoption and trying to create a solution for those. Our ESP ZeroCode modules, Certificate Provisioning Service, and Certification Assistance service are good examples of this approach.

Elektor: What was involved in creating ESP RainMaker?

Teo Swee-Ann: ESP RainMaker's conception also was to address customer pain points. Previously, we saw customers either reinventing the wheel when it came to building an IoT cloud platform or building products based on third-party cloud platforms with minimal differentiation and control over the data. On the other hand, serverless cloud architecture held so much promise for building such a cloud without worrying about managing infrastructure. It is still difficult, though, for device makers to build such a platform from scratch. That's why we created ESP RainMaker with the vision of building a cloud platform for customers that gives them full control and customizability to build their own IoT cloud with significantly less engineering investment.

Elektor: AIoT systems and products transmit lots of data. This, of course, creates concerns about data protection and privacy. Does Espressif think that the market for AIoT products will "boom" once international standards regarding data protection and privacy are agreed upon?

Teo Swee-Ann: In the present AIoT landscape, data privacy and regulatory issues are certainly of significance, and we provide our clients with multiple tools to alleviate those. First of all, we encourage edge AI by incorporating support for it in our hardware: For instance, the ESP32-S3 and upcoming ESP32-P4 have AI instructions for fast processing of deep learning models, so devices can process data locally rather than have to send it to servers. For the data that is stored on servers, we provide clients with tools that do this in a regulatory-compliant way: For instance, as reviewed by TÜV Rheinland, Espressif's ESP RainMaker cloud implementation provides all the features needed to achieve GDPR compliance when customers build their own IoT cloud based on ESP RainMaker.

Security also plays a role here: Bad actors getting into a system and making off with all the data is a somewhat common occurence nowadays. We have committed to provide secure hardware and software components that comply with the US Cybersecurity Labeling program and other international initiatives, such as Singapore's Cybersecurity Labelling Scheme. Devices certified under programs such as these can provide customers with peace of mind when it comes to their security.

A more pressing challenge we've identified is the lack of embedded software developers. Additionally, the market fragmentation further complicates the situation, leading to increased complexities. At Espressif, we're actively addressing these challenges. We've introduced many solutions. For instance, we have ESP-ZeroCode for developing Matter applications, ESP-SkaiNet for local AI voice command recognition, ESP-RainMaker for creating your own cloud platform, ESP-ADF for creating your own Wi-Fi audio system, etc.

Our goal is to equip our clients with the tools they need to create solutions without starting from scratch, to ensure efficiency and speed in development. For a significant positive shift in the AIoT landscape, we believe that reducing fragmentation is critical. It's imperative for different platforms to collaborate and foster a unified standard. While the Matter standard shows promise in this direction, its full potential and long-term impact are yet to be ascertained.

Elektor: Did the global chip shortage boost the use of Espressif-based technology in commercial products? If so, how can those products help further establish the Espressif brand?

Teo Swee-Ann: During the challenging period of the global chip shortage, Espressif implemented a strategic approach by rationing chip sales to our valued customers. This decision was met with appreciation from many of our clients. Firstly, it ensured that they continued to receive a consistent supply of chips, even if it was in limited quantities. Secondly, this strategy prevented our customers from hastily overstocking their inventories in response to the shortage. And, importantly, throughout this period, we remained committed to not increasing our prices, ensuring that market volatility did not adversely impact our pricing structure. I believe our stability and modest growth in 2023 reflect the effectiveness and foresight of these policies.

Elektor: In an interview several years ago with Elektor, you said AI would be the next big thing, and that has come true. What do you think the next big thing in electronics will be?

Teo Swee-Ann: In the realm of electronics, there are two distinct trajectories that we must consider. Firstly, there's the trajectory leading toward a more immersive virtual or mixed-reality ecosystem. Envision a future where, within the next few decades, we could possess implants that not only augment traditional senses, such as vision and hearing, but also introduce novel sensory experiences or cognitive pathways previously uncharted by natural evolution. This would mean that our emotions, cognitive capacities, and sensory experiences would be significantly enhanced or

"As we look to the future, our main objective remains the creation of pioneering solutions that prioritize energy efficiency and smaller form factors."

About Teo Swee-Ann

Teo Swee-Ann is the founder and CEO of the Shanghai-listed semiconductor firm Espressif Systems. He holds a master's degree in electrical engineering from the National University of Singapore. Teo worked with US chipmakers Transilica and Marvell Technology Group and China's Montage Technology before founding his own firm in 2008.

modulated by AI. Realizing this vision demands significant technological innovation, specifically in achieving higher computational capacities at reduced power consumption and voltages, with the most compact form factors.

On the other hand, the second direction emphasizes the application of AI in the design facet of electronics. To provide a comparison, if AI is capable of autonomously operating vehicles, it certainly possesses the potential to craft sophisticated circuit designs. This shift addresses a notable challenge in our field, which is the tendency for designs to be over-engineered. Incorporating AI into the design process presents an opportunity to refine these designs, making them more streamlined and effective, thereby accelerating the realization of the first trajectory.

Elektor: What is your vision for Espressif? Where do you see the company in, say, five years?

Teo Swee-Ann: At Espressif, we position ourselves primarily as a leading AIoT chip company. That being said, our capabilities extend far beyond just hardware. We have a strong foothold in software development, with our proprietary software system, ESP-IDF, and extensive solution frameworks catering to cloud integration, edge AI, mesh networking, audio and camera applications, and more. What truly sets us apart is our thriving community and ecosystem, where professionals and amateurs come together to share and develop groundbreaking ideas. As we look to the future, our main objective remains the creation of pioneering solutions that prioritize energy efficiency and smaller form factors. In essence, we hope that Espressif represents a distinctive ethos and a progressive attitude towards engineering, innovation, and fostering a collaborative community spirit.

230614-01

WEB LINKS

[1] C. Valens, "The Reason Behind the Hugely Popular ESP8266?: Interview with Espressif Founder and CEO Teo Swee Ann on the new ESP32," Elektor Business, 1/2018.: https://elektormagazine.com/magazine/elektor-63

Simulate ESP32 with Wokwi

Your Project's Virtual Twin

By Uri Shaked, Wokwi

Wokwi is a simulator for embedded systems and IoT devices. It provides an online environment where you can prototype, debug, and share your ESP32 projects without needing the physical hardware. The simulator runs right in your web browser, and can simulate all the chips in the ESP32 family: the classic ESP32, as well as the S2, S3, C3, C6, and H2 variants. Additional microcontroller families are available as well.

Wokwi enables you to create a digital twin of your project: You can simulate a variety of input and output devices [1], such as LCD screens, sensors, motors, LEDs, buttons, speakers, and potentiometers, and you can even create your own simulation models for new devices. The simulator allows you to iterate faster, work on your firmware code even when the hardware is away or not available, use powerful debugging tools, share your projects ,and collaborate with other engineers in a simple way.

Start Your Wokwi Journey

You can use your favorite programming language with Wokwi. If you are a beginner, MicroPython or Arduino Core are probably good choices. For professionals and advanced users, you can use the ESP-IDF directly, use the Rust programming language, or use an embedded RTOS such as Zephyr or NuttX.

We recommend browsing some of the existing examples to get ideas of what you could build with Wokwi:

- MicroPython [2]
- ESP32 with Arduino Core [3]
- Rust [4]
- DeviceScript (TypeScript for ESP32) [5]

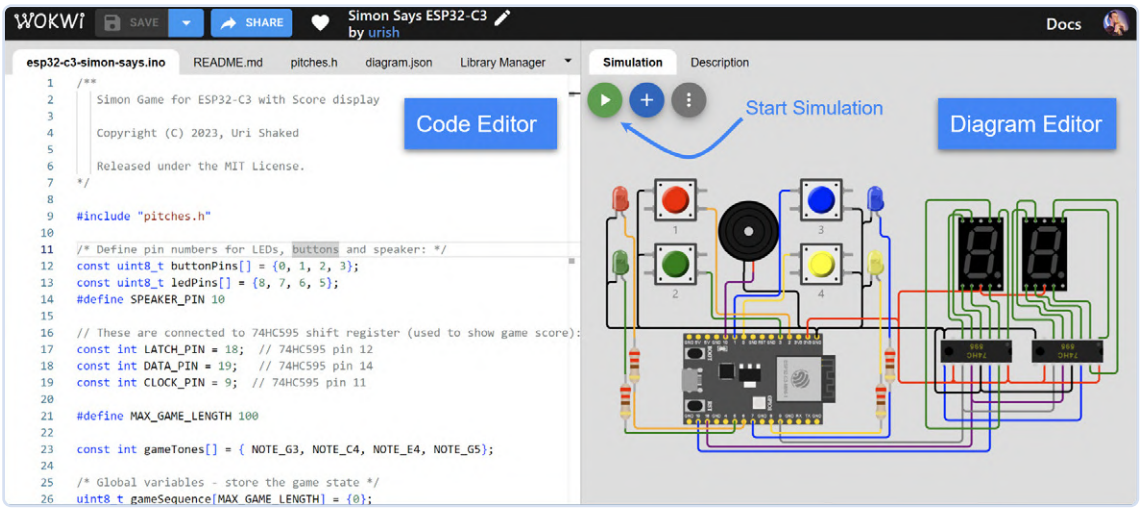


Figure 1: The Wokwi User Interface, "Simon Says" game.

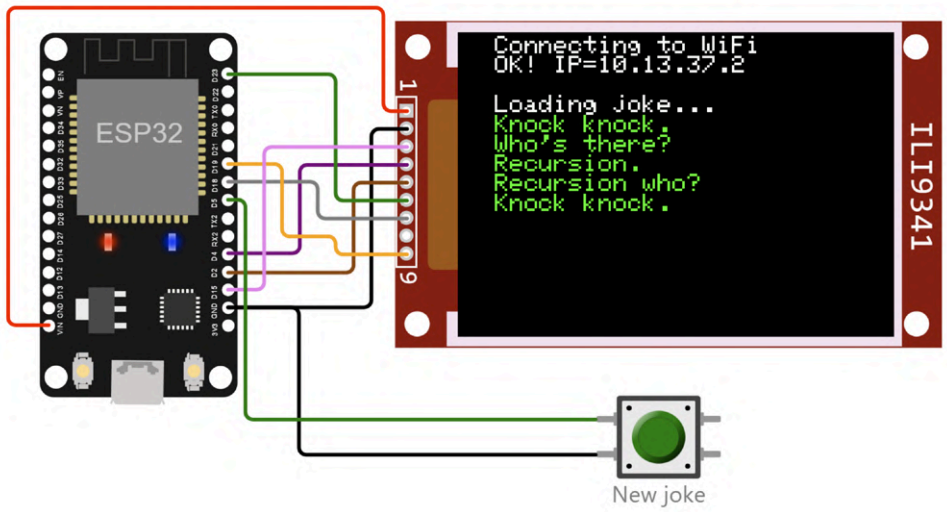


Figure 2: Wi-Fi Simulation Client example [8].

After opening an example, press the green *play* button to start the simulation and interact with the project. Some of the projects also have a `README` file, where you can learn about the project and how to use it. To start a new project from scratch, go to [6], choose your microcontroller and programming language, and click *Start*.

The Wokwi User Interface

The Web version of Wokwi is split into two: The left pane is where you edit your firmware's source code (the *Code Editor*), and the right pane is where you draw the project diagram by adding different devices and connecting them to your microcontroller (the *Diagram Editor*), shown in **Figure 1**.

The Diagram Editor

Add new parts to the diagram by clicking on the blue + button and selecting the desired part from the list. Drag the parts to the desired position. To draw a wire, click on the starting pin, and then click on the target pin. If you want the wire to be routed in a specific direction, you can guide it by clicking where you want it to go after selecting the first pin.

If you want a neat-looking result, turn on the grid by pressing *G*. This will align all the parts and the wiring to a 0.1" (2.54 mm) grid (the standard pin header spacing). Some useful keyboard shortcuts: *D* to duplicate the selected part, *R* to rotate it, the numbers 0 to 9 to quickly set the color of a wire, LED, or push button, and pressing *Shift* while dragging the mouse to select multiple parts at once. For more shortcuts, check out the Diagram Editor guide [7].

The Library Manager

Use the Library Manager to quickly add any standard Arduino library to your project. Paid users can also upload any custom Arduino library and include it in their project.

For other programming languages, you can include libraries by editing the project configuration files (e.g., *Cargo.toml* for Rust), or by directly adding the source code of the library to your project (e.g., for MicroPython).

Wi-Fi Simulation

The ESP32 has built-in Wi-Fi functionality, making it a popular choice for IoT projects. Wokwi simulates the chip's Wi-Fi functionality. The simulator (**Figure 2**) provides a virtual access point called *Wokwi-GUEST*. It's an open access point (so no password required). The access point is connected to the internet, enabling your simulated projects to connect to HTTP servers (**Figure 3**), MQTT brokers, and other cloud services, such as Firebase, ThingsSpeak, Blynk, and Azure IoT.

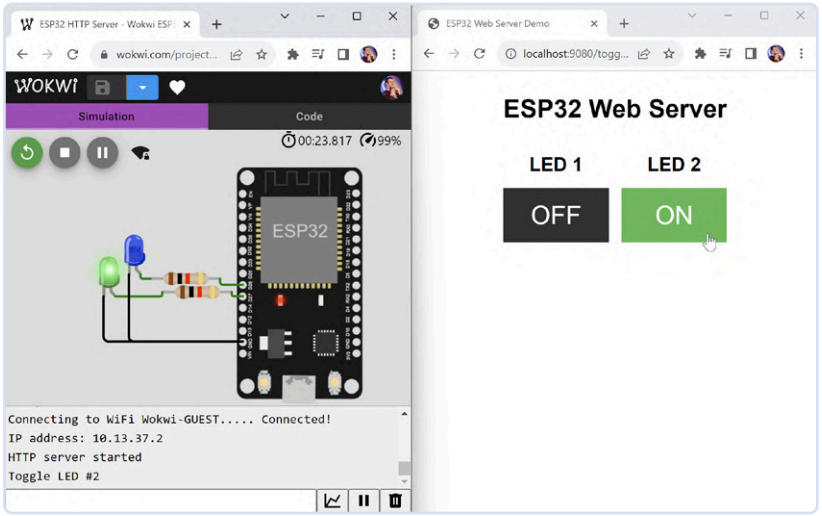


Figure 3: Wi-Fi Simulation Web Server example [9].

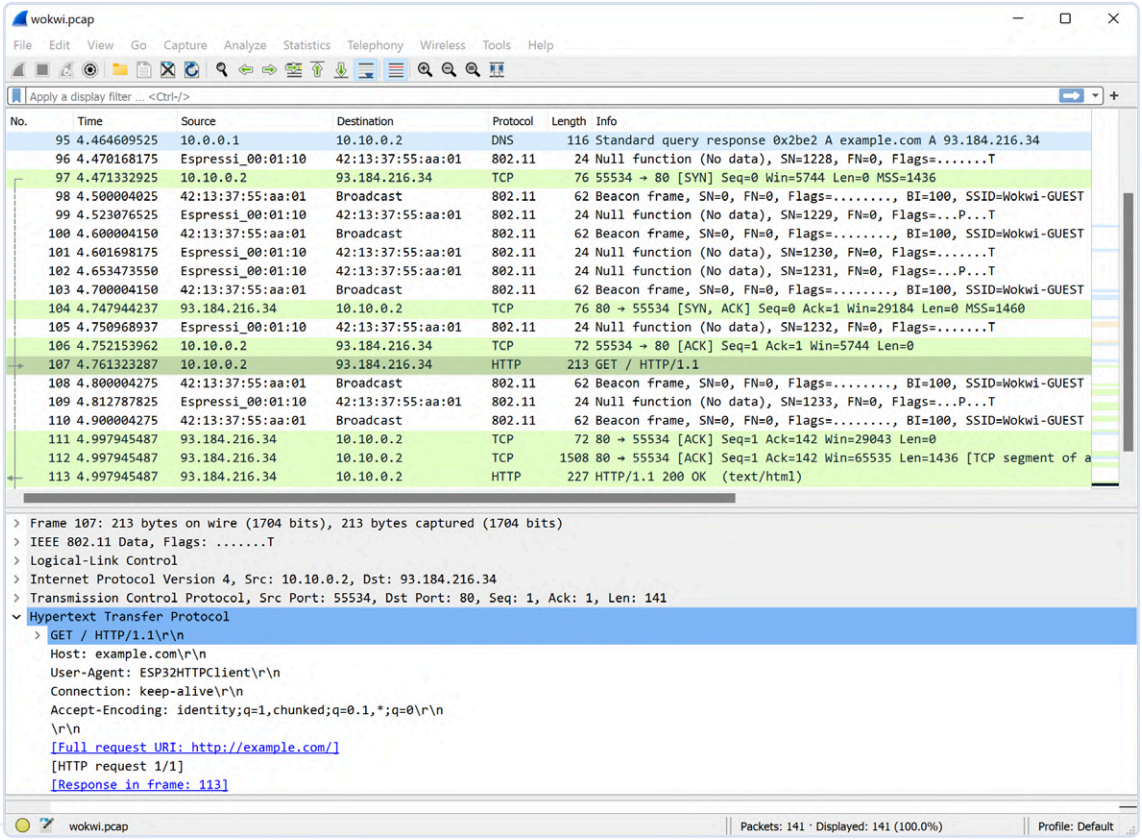


Figure 4: Viewing Wi-Fi traffic capture.

Paid users can also run a special IoT Gateway [10] on their computers, enabling them to connect to local servers from the simulation, as well as to connect from their computer to an HTTP server (or any other kind of server) running inside the simulator. Under the hood, Wokwi simulates a complete network stack: starting at the lowest 802.11 MAC Layer, through the IP and TCP/UDP layers, all the way up to protocols such as DNS, HTTP, MQTT, CoAP, etc. You can capture the raw network traffic [11] and view it in a network protocol analyzer such as Wireshark, as shown in **Figure 4**.

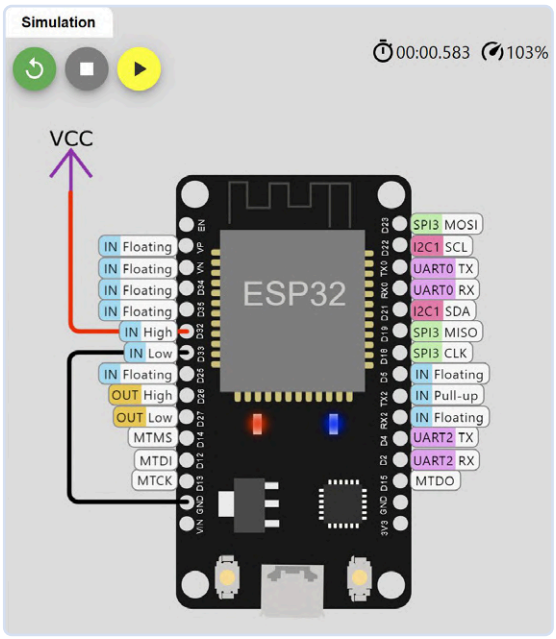


Figure 5: Simulation paused, with state indication for each pin.

Pin Function Debugger

The ESP32 has a flexible peripheral pin mapping: Software can define which peripheral is connected to each of the pins by configuring IO MUX, GPIO Matrix, and the low-power/RTC peripherals. Defining the pin mapping in software can be very convenient, but it also makes it more challenging to figure out the actual pinout for your firmware. Luckily, with Wokwi, all you have to do is just to pause the simulation and you can immediately see the current function and state of each pin, as shown in **Figure 5**.

Visual Studio Code Integration

Using Wokwi in your web browser is great for learning, quick prototyping, and sharing your work. However, for more complex projects, we recommend using Wokwi alongside your standard IDE and development tools. Wokwi integrates seamlessly with Visual Studio Code. All you need to do is to install the *Wokwi for VS Code* extension [12], and create a simple configuration file [13] that tells Wokwi where to find your compiled firmware.

You can also integrate Wokwi with VS Code's built-in debugger just by adding a few lines of configuration [14]. Unlike real hardware, Wokwi has an unlimited number of breakpoints, allowing you to debug more efficiently (**Figure 6**).

For IoT projects, you can define TCP port forwarding [15]. This allows you to connect from your computer to a web server (or any other kind of TCP server) running inside the simulated ESP32 chip.

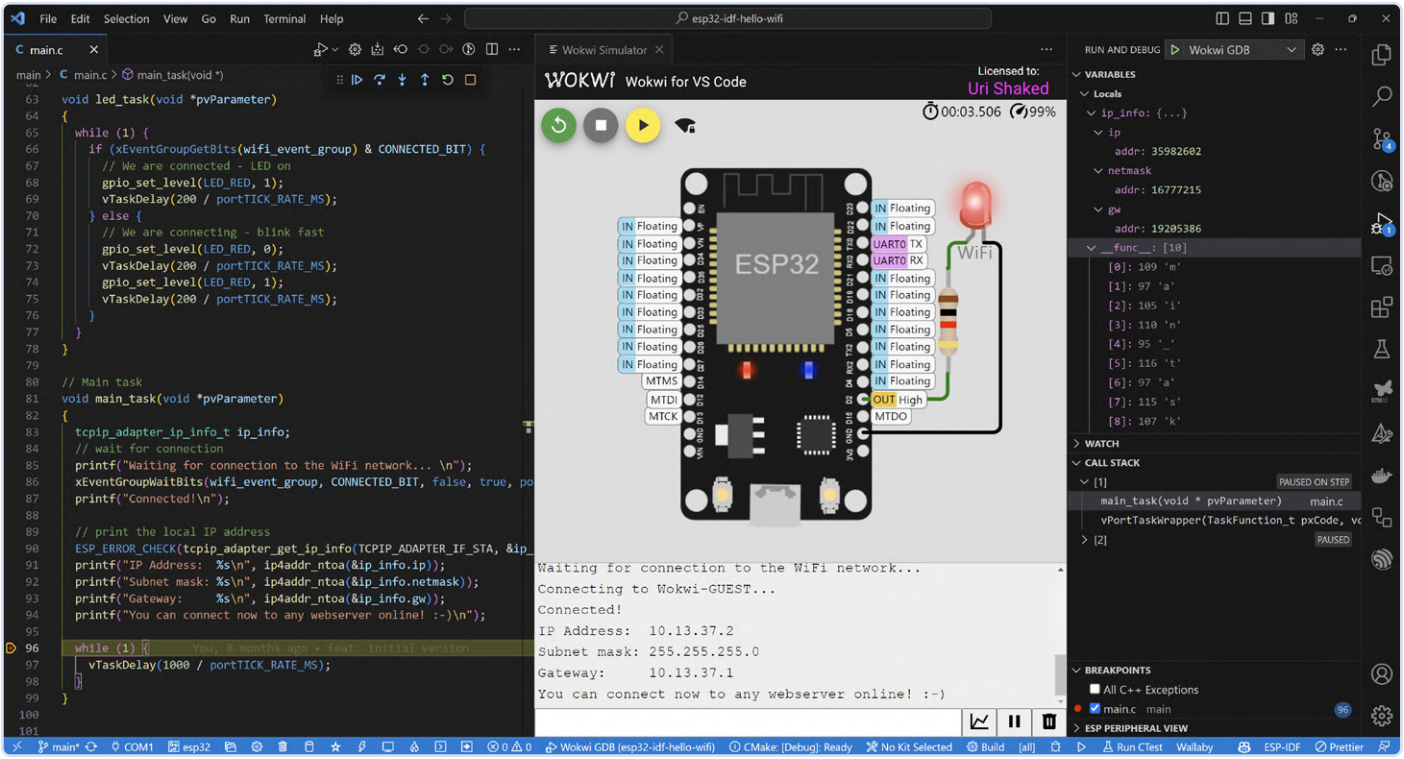


Figure 6: Debugging an ESP-IDF project with VS Code and Wokwi.

Espressif IDE Integration

For those of you who love the Espressif IDE, you can also set up a launch configuration to start your project in the simulator. The output of your program (UART) will go right into the IDE console. For full instructions, check out *How to Use Wokwi Simulator with Espressif-IDE* [16].

Custom Chips

Custom Chips let you create new parts in Wokwi, and extend its functionality. You can create new sensors, displays, memories, testing instruments (**Figure 7**), and even simulate your own custom hardware.

To create a custom chip, you need to define the pinout in a JSON file, and then write code to implement the logic of the chip. The code is usually written in C, and the API resembles common embedded chip hardware abstraction layers (HAL), so firmware developers should feel at home. There's also experimental support for other languages such as Rust, AssemblyScript, and Verilog. For more information and examples, check out the Chips API documentation [17].

Wokwi for CI

Simulation can be a huge time saver when prototyping, but it can also help you catch bugs as you keep developing your product. Continuous integration (CI) is a modern software engineering practice: Build and test your project whenever you modify the code. Testing with real hardware, however, is very time-consuming and difficult to automate. It becomes even more complicated when you want full system integration. Just try to imagine how you'd set up automated Wi-Fi testing, or capture the image your project displays on an LCD screen? How much work would it take to make this setup robust and reliable?

Wokwi for CI takes away all that complexity. If you are using GitHub Actions, `wokwi-ci-action` [18] allows you to integrate simulation into your existing workflow with just a few lines of code. For other

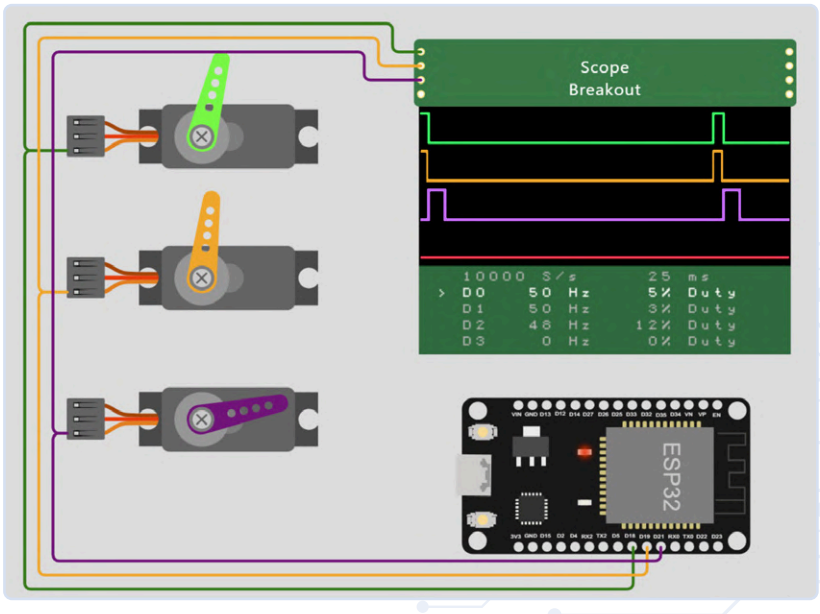


Figure 7: A digital scope one user created using the Custom Chips API.

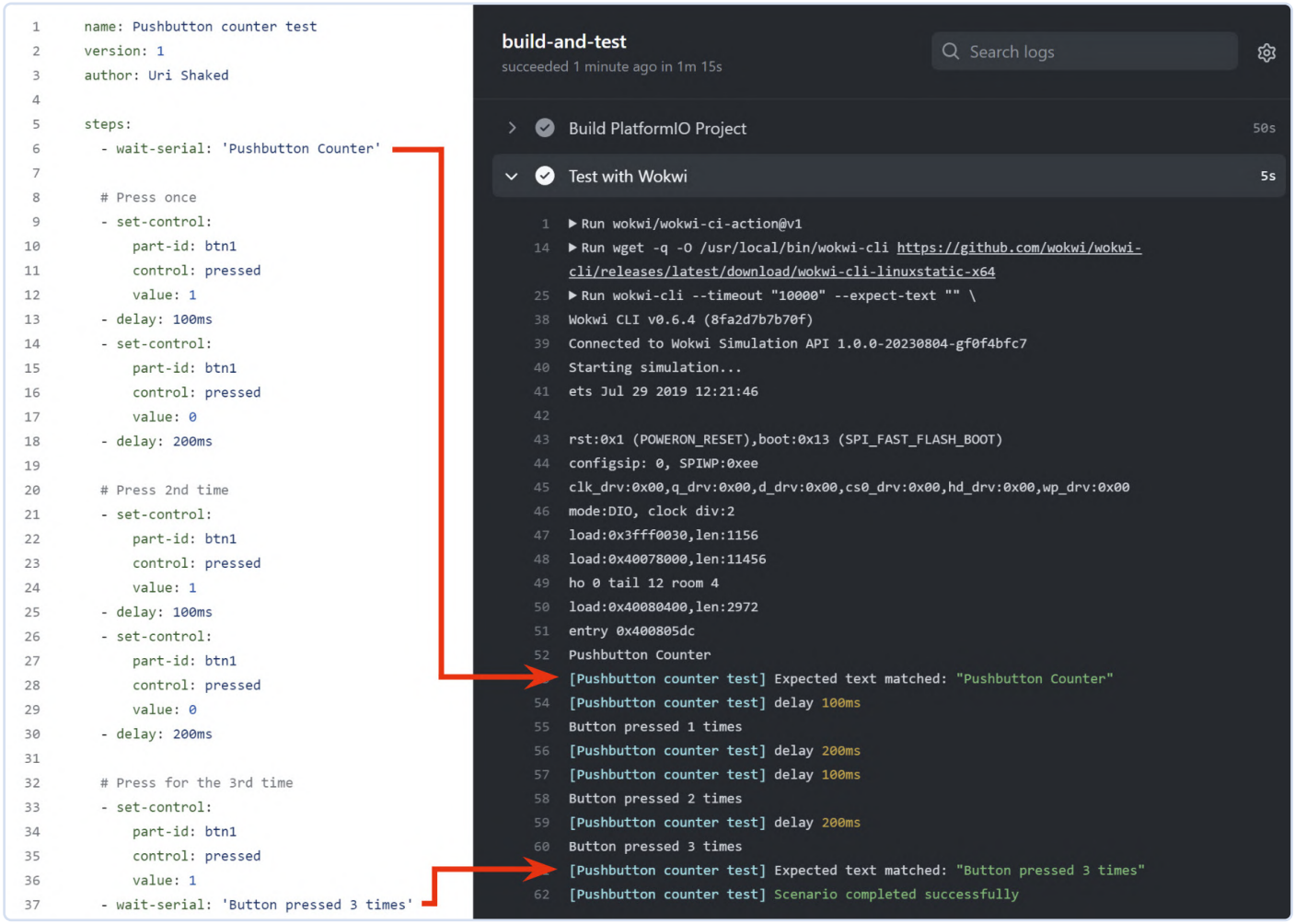


Figure 8: The Automation Scenario code [20] (left) with the corresponding GitHub Action output (right).

CI environments, `wokwi-cli` [19] provides a simple command-line interface for running and testing your firmware in simulation. The CI Simulation Server is available as a managed cloud service, and also for on-premise deployment.

Automation Scenarios

Automation scenarios (**Figure 8**) allow you to perform complex tests and assertions on your firmware. Writing them is simple: Each scenario is a YAML file that contains a list of commands such as pushing a button, setting a temperature sensor value, or looking for a string in the firmware's serial output.

Limitations

Wokwi has many capabilities, but it also has some limitations you should be aware of. The main focus of the simulator is running embedded firmware code. The simulation engine is mostly digital, with limited support for analog simulation. This means that Wokwi won't shout at you when you forget a current-limiting resistor, and nor will the magic smoke be released (one day, maybe).

230523-01

Questions or Comments?

Do you have technical questions or comments about this article? Contact Elektor at editor@elektor.com.

About the Author

Uri Shaked is a longtime maker. He's currently working on Wokwi, an online IoT and embedded systems simulation platform, and on Tiny Tapeout, making custom ASIC manufacturing affordable and accessible.

Related Products

- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324
- **Koen Vervloesem,** *Getting Started with ESPHome* **(Elektor, 2021)**
  www.elektor.com/19738

WEB LINKS

[1] Wokwi Supported Hardware: https://docs.wokwi.com/getting-started/supported-hardware
[2] Wokwi Online Embedded Python Simulator: https://wokwi.com/micropython
[3] Wokwi Online ESP32 Simulator: https://wokwi.com/esp32
[4] Wokwi Online Embedded Rust Simulator: https://wokwi.com/rust
[5] Wokwi Online DeviceScript Simulator: https://wokwi.com/devicescript
[6] New Arduino Uno Project Start Page: https://wokwi.com/projects/new
[7] Diagram Editor Keyboard Shortcuts: https://docs.wokwi.com/guides/diagram-editor#keyboard-shortcuts
[8] *esp32-jokes-api* Wi-Fi Simulation Example: https://wokwi.com/projects/342032431249883731
[9] ESP32 HTTP Server Example: https://wokwi.com/projects/320964045035274834
[10] ESP32 Wi-Fi Network: The Private Gateway: https://docs.wokwi.com/guides/esp32-wifi#the-private-gateway
[11] ESP32 Wi-Fi Network: Viewing Wi-Fi Traffic with Wireshark: https://tinyurl.com/wsviewtraffic
[12] Wokwi Simulator — Visual Studio Code Extension: https://tinyurl.com/wokwivsext
[13] Configuring Your Project (*wokwi.toml*): https://docs.wokwi.com/vscode/project-config
[14] Wokwi — Debugging Your Code: https://docs.wokwi.com/vscode/debugging
[15] Configuring Your Project — IoT Gateway (Port Forwarding): https://docs.wokwi.com/vscode/project-config#iot-gateway-esp32-wifi
[16] How to Use Wokwi Simulator with Espressif-IDE: https://tinyurl.com/wokwiespide
[17] Getting Started with the Wokwi Custom Chips C API: https://docs.wokwi.com/chips-api/getting-started
[18] *wokwi-ci-action*: https://github.com/wokwi/wokwi-ci-action
[19] *wokwi-cli*: https://github.com/wokwi/wokwi-cli
[20] *platform-io-esp32-counter-ci* — Automation Scenario Code: https://tinyurl.com/pushcnttst

Figure 1: The ESP32-S3-BOX-3 kit unboxed. Not shown are the USB cable and the tiny RGB LED module with the four jumper wires. The nectarine is not included.

Trying Out the ESP32-S3-BOX-3

A Comprehensive AIoT Development Platform

By Clemens Valens (Elektor)

The ESP32-S3-BOX-3 from Espressif is a platform for developing applications like personal assistants, smart speakers, and other voice-controlled devices. Let's have a closer look.

Based on an ESP32-S3, the ESP32-S3-BOX-3 [1] kit is marketed as "Your next AIoT development tool." AIoT stands for Artificial Intelligence of Things, and is closely related to, but not to be confused with IoT, which means Internet of Things (as you surely already knew). It is shaped like a small console with a touch screen. Suggested applications for the kit are AI chatbots, Matter (a unifying protocol for home automation), robot controllers and smart sensor devices.

Inside the Box

The ESP32-S3-BOX-3 (**Figure 1**) comes as one of those slow-opening boxes (to improve the Wow! effect, as I was told by someone who had worked at Apple). When opened, you see a display module (55 mm × 50 mm) with a kind of stand for it, and a short (approx. 30 cm) USB-C cable. This is only the top layer. Under the stand hides a second, somewhat smaller stand. Under the display module, you'll find yet another smaller stand and even a much smaller fourth one that plugs onto a breadboard.

Tucked away with the USB cable is a little bag containing a small RGB LED module and four jumper wires.

Display Module

The display module is surprisingly heavy (60 g) for such a small object (WHD 55 mm × 50 mm × 12 mm). This is the unit that packs all the power: an ESP32-S3-WROOM-1-N16R16V module featuring 2.4 GHz Wi-Fi (802.11b/g/n) and Bluetooth 5 (LE). The display itself is a 2.4", 300 × 240 pixels TFT display with capacitive touch. In case you wondered, there is no battery inside (**Figure 2**).

On the front are, besides the display, two tiny holes (the microphones) and a red circle (the return button).

There are two pushbuttons (Boot & Rst) and a USB-C connector on the left side and on the right side something that looks like a microSD-card slot, but which in reality is a loudspeaker. On the top side is a mute button and two LEDs; a PCIe connector sticks out the bottom side. There is nothing on the rear of the module.

Stands and Brackets

The PCIe connector mates with a socket on one of the stands or brackets. As mentioned before, there are four of them, each with different functions (**Figure 3**). The one shown on top of the box is the DOCK. It has two Pmod extension sockets (2 × 6 female header) on the back together with a USB-C socket for power (in & out). A USB-A host socket is available on the left side. A label shows the names of the signals that are accessible on the Pmod connectors.

Sensor Bracket

The largest stand is called SENSOR. It has a temperature and humidity sensor with tiny On/Off switch on the back, a microSD card slot (not a loudspeaker this time) on the left and a USB-C power-in socket on the right. On the front side is an IR sensor and LED, and a really cool feature if you ask me: a radar. This stand has room inside for one 18650-type battery.

The third stand is the BRACKET, and it can be attached to something else thanks to the two screws sticking out of its backside. This stand also has two Pmod headers and a USB-C socket. There are no labels here, so we will suppose that the Pmod headers are wired in the same way as on the DOCK.

The smallest stand, the BREAD, is a simple PCIe-to-breadboard 24-way break-out board. The pins that go into the breadboard are conveniently labelled.

First Power-on

When you power the display module on for the first time, it boots into what later turns out to be help or hint mode. This mode takes you through a few hint screens that explain some basic functionality of the device. It then enters normal mode. In this mode, you can scroll through six functions: Sensor Monitor, Device Control, Network, Media Player, Help and About Us.

Voice Control

According to the help screens, you can voice activate some things by saying "Hi ee es pea" (ESP pronounced letter by letter). As nothing happened when I tried this, I downloaded the user guide [2]. For this, a QR code is printed on the bottom of the box. According to the manual, voice control works in every screen (unless the mute LED is on). Knowing this, I tried again and managed to make it work a couple of times.

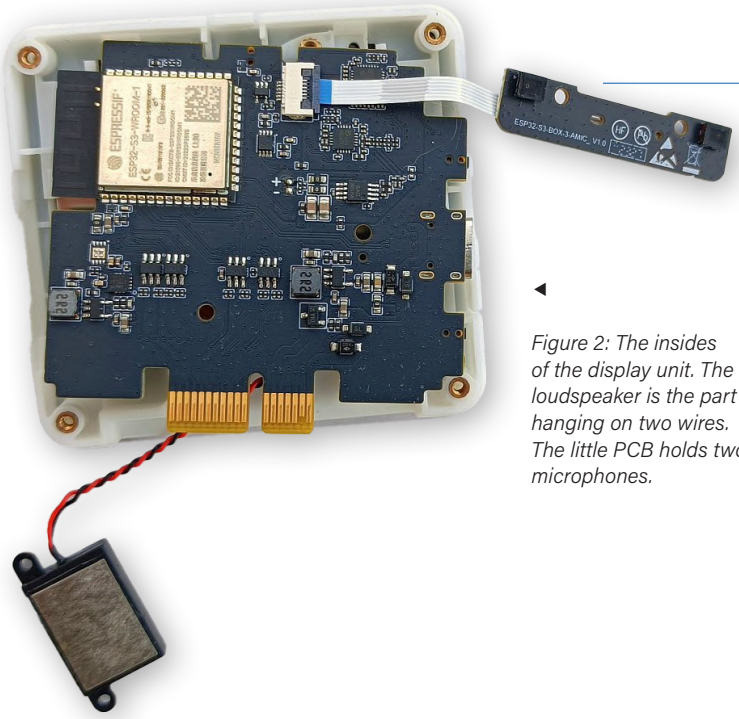


Figure 2: The insides of the display unit. The loudspeaker is the part hanging on two wires. The little PCB holds two microphones.

If you succeed, you will hear a ping sound and the display shows a microphone. You now have six seconds to pronounce a command. After six seconds of inactivity, the device says, "Timeout, see you next time" and returns to the mode it was in before it was activated. I found it very hard to get the device to wake up, but once woken up, it recognized my commands without any problem. Curious.

Radar

I discovered an interesting function after coming back from refilling my coffee mug. The display had switched off while I was away and suddenly switched back on. That must be the work of the built-in radar. If you don't move for a while, it will switch off too.

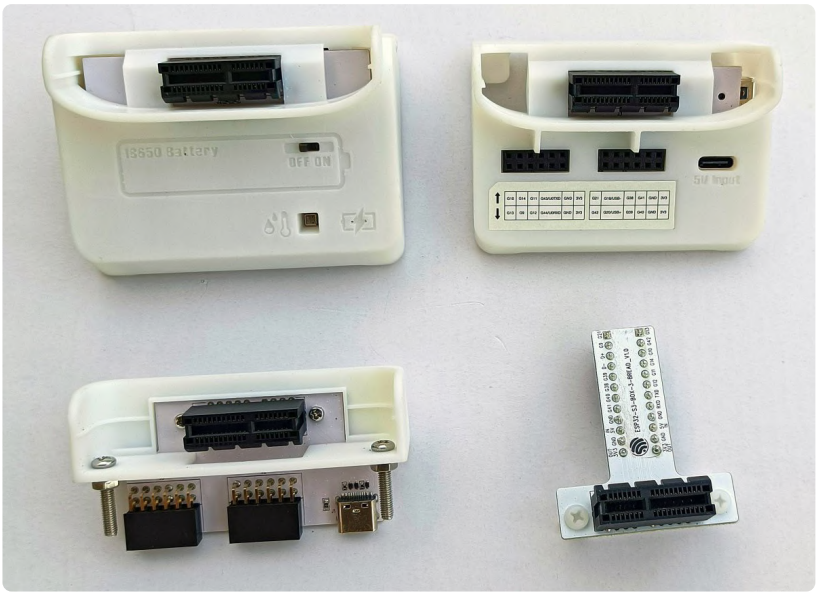
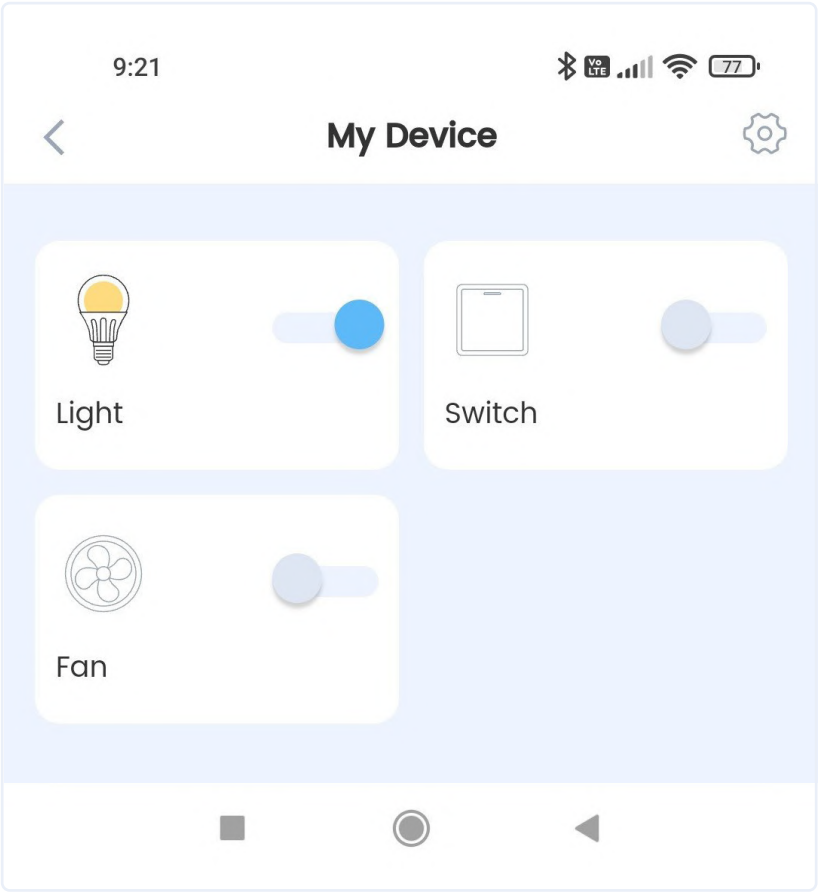


Figure 3: Four brackets allow for all sorts of applications.



▲

Figure 4: The ESP BOX app provides a connection between the kit and the Rainmaker cloud service from Espressif.

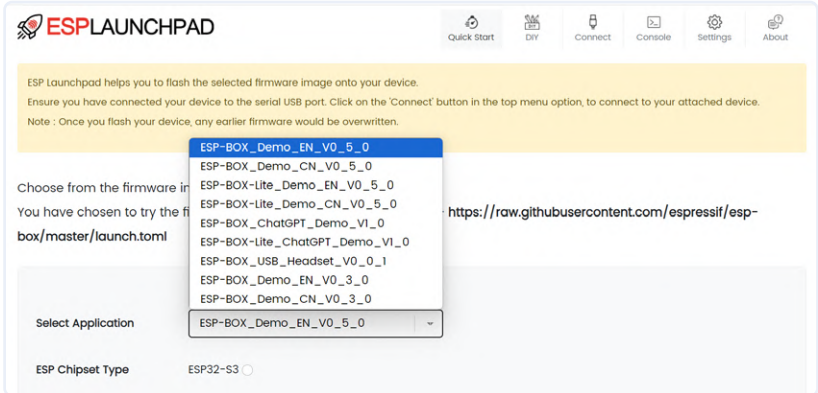
The Box in the Cloud

After fiddling around a bit with the built-in "apps" (power-cycle the display in case you hot-plugged it on the Sensor bracket; otherwise it won't work), I moved on to the ESP-BOX app. You can get it through the Network page after tapping the *To install APP* button and scanning the QR code.

With the app, you can connect the kit (and other devices) through a Wi-Fi network to the Espressif Rainmaker cloud service. After adding the device to the app, it provides you with access to the same controls as on the Device Control page (Light, Switch and Fan, but not Air, **Figure 4**). On your phone, you can now toggle these functions and the result is shown on the ESP32-S3-BOX-3 display. However, this appears to be one way communication only. Toggling a button on the display does not update the app.

Figure 5: ESP Launchpad is an online firmware-programming tool and serial terminal.

▼



If, in the app, you tap the button's icon instead of the slide switch, a settings page opens. Here you can define which GPIO pins are controlled by the button and what the voice control commands do. You can also redefine the voice commands simply by typing in a new command phrase. Some hints for good commands are given in the user guide. Unfortunately, as I couldn't get the device to wake up, I was unable to try any new commands.

I did not find a document explaining how to create your own applications to use with the ESP-BOX app.

Serial Port

Know that you can monitor what is going on inside the device if you connect the display unit to a computer and open a serial terminal. This feature turned out to be very valuable for the next section.

ESP Launchpad

On the bottom of the box of the ESP32-S3-BOX-3 are three QR codes. One is for ESP Launchpad [3]. This app only works in Chrome (at least on Windows) and is intended to reprogram the kit with other firmware. It seems that you can even publish your own firmware here, so other people can try it too (**Figure 5**).

At first, it couldn't connect to my device for some reason, but after rebooting it, I got a list of demo programs.

ChatGPT Demo

From this list, I selected (of course) *ESP-BOX_ChatGPT_Demo_V1_0*, eagerly uploaded it to the display unit and... nothing. The screen remained black. I then tried the next-best demo, *ESP-BOX-Lite_ChatGPT_Demo_V1_0*, but got a similar result. After trying a couple of examples more, it suddenly started to dawn on me (thank you! serial port): these files are not for the ESP32-S3-BOX-3, they are intended for previous versions of the kit, the EPS32-S3-BOX (without '-3') and the ESP32-S3-BOX-Lite. Hopefully, this issue has been solved when you read these lines.

Building Your Own Applications

At this point, I found myself on GitHub, in the repository that supports all three versions of the ESP32-S3-BOX [4]. Now, why didn't they print a large QR code for this page on the box? It is not mentioned in the user guide, either. This repository has lots of information about the kit and also the source code for the example programs.

Requires ESP-IDF V5.1 or Later

There are no precompiled easy-to-try executables, at least I didn't find them, so you have to build them yourself. To do so, you must clone the repository first. You also need ESP-IDF V5.1 or later.

Compilation and uploading to the device took a while, but once it was done, I was finally back at the beginning of this review when the display unit was still working. To my surprise, voice control seemed to respond better, and it was easier to wake up the device. The software version was still V1.1.1 but my EPS-IDF version had become 'v5.11-dirty' (was 'v5.2-dev-2164-g3befd5fff7-dirty').

ChatGPT Demo, Take 2

Now that I had a development environment installed for the ESP32-S3-BOX-3, I decided to try building the ChatGPT demo according to the provided instructions. This went fine.

After compiling and uploading the program to the display unit, the demo booted normally and presented instructions on how to configure it [5]. When done, it connected to my Wi-Fi network, and I could start talking to it. However, the chatbot never replied to my questions, but instead showed the message "API Key is not valid." In the serial terminal I found this line:

```
E (369916) OpenAI: You exceeded your current quota, please check your plan and billing details.
```

As it turns out, you need some credit on your ChatGPT account to use API keys. My credit had expired. You get some for free when you create an account for the first time (requires a phone number). As I didn't want to set up a payment plan, I stopped here.

Image Display Example

Finally, I decided to try a last example, the Image Display. The description doesn't say what it does, so that would be a nice surprise. This example is a bit smaller than the others, and therefore compiling and uploading it is quicker. The result? A list of six smileys from which you can choose one to display (**Figure 6**).

Complex and a Lot to Offer

This review has taken me a bit more time than I expected when I started. The reason is the complexity of the product. The ESP32-S3-BOX-3 certainly has a lot to offer, but to get something useful out of it, you must invest time and effort.

The place to start is the GitHub repository and not the QR codes on the back of the box. On GitHub you will not only find the User Guide, but also other useful documents and example programs. It would have saved me a lot of time if I had known this right from the start.

Even though there is quite a lot of information, it still feels a bit incomplete. A tutorial on how to work with the ESP-BOX app or how to create a voice-controlled application from scratch would be appreciated. There isn't much about AIoT, yet that is what this product is all about, isn't it?

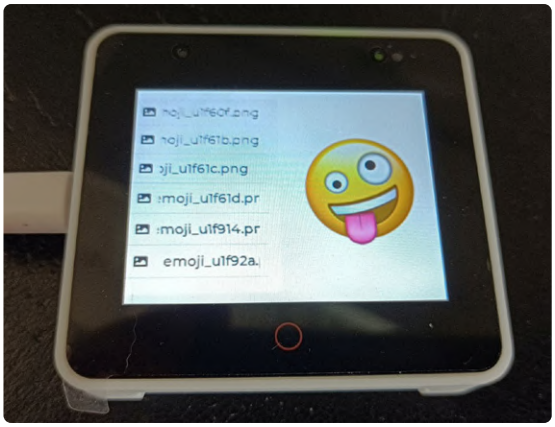


Figure 6: Artificial Intelligence of Things (AIoT), still a long way to go?

On the hardware side, all I can say is that it is really nicely done. The build quality is good, the brackets fit perfectly, the buttons work, the display looks smart, touch responds smoothly, and everything comes packed in a quality box. The display unit, with or without a bracket, makes for a cool gadget on a desk, or in a bedroom or living room.

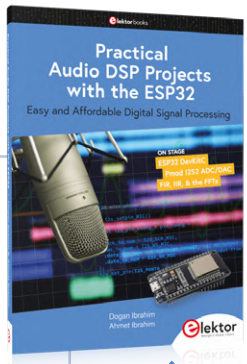
230555-01

Questions or Comments?

Do you have technical questions or comments about his article? Email the author at clemens.valens@elektor.com or contact Elektor at editor@elektor.com.

Related Products

- **ESP32-S3-BOX-3**
  www.elektor.com/20627
- **Arduino Nano ESP32**
  www.elektor.com/20562
- **Practical Audio DSP Projects with the ESP32**
  www.elektor.com/20558



WEB LINKS

[1] ESP32-S3-BOX-3: www.espressif.com/en/news/ESP32-S3-BOX-3
[2] User Guide: https://qr10.cn/CoahPA
[3] ESP Launchpad: https://qr10.cn/DCgKrD
[4] ESP32-S3-BOX-3 on GitHub: https://github.com/espressif/esp-box
[5] ChatGPT secret key: https://platform.openai.com/account/api-keys

Electronics Workspace Essentials

Insights and Tips from Espressif Engineers

The electronics workspace: It's where the magic happens. Are you curious about what other creative engineers have in their workspaces? Looking for some tips about stocking and organizing your electronics workbench? A few Espressif engineers offer some insights.

Omar Chebib
Location: Shanghai, China

Organize and Keep Notes

My workspace is my desk at home. It may not be large, but I try to always keep it clean and organized. I only keep the required tools, devices and components I need for my current tasks. Moreover, consistently placing them in the designated spot lets me uphold efficiency in my tasks. Naturally, this also necessitates a well-organized shelf that includes compartments separating pure electronics, with through-hole and SMD components, as well as embedded tools, with a logic analyzer, some ESP32 chips, or even I²C devices. This enabled me to work on several projects involving logic gates, a Z80 8-bit processor, FPGA, through-hole and surface-mount soldering, including the "terrifying" BGA package.

Advice: On the hardware side, only keep the tools, devices, and cables that you need for the current task on your desk, only get new tools when you need them, and clean up after finishing. On the software side, always version-control your work, even if it is locally. It is never too late to start versioning an already existing project. On a more general note, when context-switching between projects, keep a note explaining what you already did, where you left off, and what the next step is. This will help a lot to get back into the swing of things. Current projects: In my free time, I designed a Zeal 8-bit Computer — a Z80-based computer — from scratch, from PCB design to software. In my work time, I implement ESP32-C3 emulation on QEMU.

Essentials
- Logic analyzer
- Multimeter
- Oscilloscope
- Electronic microscope
- Soldering station + flux
- A good fume extractor!
- Linux computer

Wishlist
- Soldering hot plate
- Better microscope
- Better chair