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

elektor MAG
design > share > earn

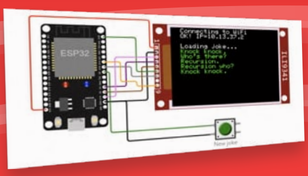
24001
* SINCE 1961 *
DEC. 2023 & JAN. 2024
ELEKTORMAGAZINE.COM

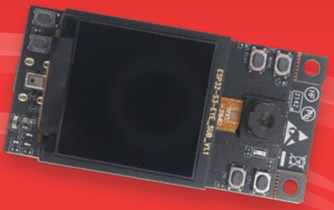
SPECIAL EDITION

140 Pages

Special edition
guest-edited by

ESPRESSIF



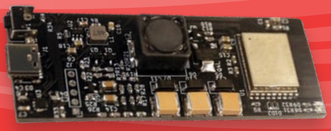


Prototyping With **Espressif Chips**

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

- An Open-Source Speech Recognition Server
- Power Duo: Rust + Embedded
- Facial Recognition With ESP32-S3-EYE
- Walkie-Talkie with ESP-NOW
- Another Elektor Christmas Tree Project

and much more!

Unleashing the ESP32-P4
The Next Era of Innovative Microcontrollers
p. 59

Acoustic Fingerprinting
Song Recognition With ESP32
p. 80

A Vision for the AIoT
Interview with Espressif CEO Teo Swee-Ann
p. 35

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

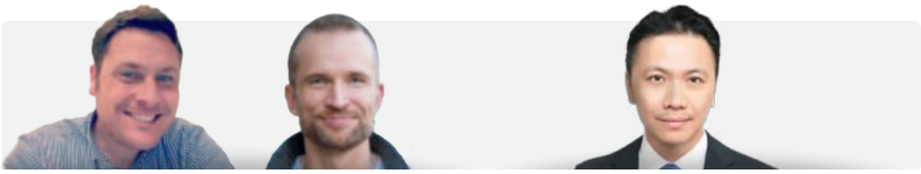
Advertising & Sponsoring
Büsra Kas
Tel. +49 (0)241 95509178
busra.kas@elektor.com
www.elektormagazine.com/advertising

The circuits described in this magazine are for domestic and educational use only. All drawings, photographs, printed circuit board layouts, programmed integrated circuits, digital data carriers, and article texts published in our books and magazines (other than third-party advertisements) are copyright Elektor International Media b.v. and may not be reproduced or transmitted in any form or by any means, including photocopying, scanning and recording, in whole or in part without prior written permission from the Publisher. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature. Patent protection may exist in respect of circuits, devices, components etc. described in this magazine. The Publisher does not accept responsibility for failing to identify such patent(s) or other protection. The Publisher disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from schematics, descriptions or information published in or in relation with Elektor magazine.

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!

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!

The Team

International Editor-in-Chief: Jens Nickel | **Content Director:** C. J. Abate | **International Editorial Staff:** Asma Adhimi, Roberto Armani, Eric Bogers, Jan Buiting, Stuart Cording, Rolf Gerstendorf (RG), Ton Giesberts, Hedwig Hennekens, Saad Imtiaz, Alina Neacsu, Dr. Thomas Scherer, Jean-Francois Simon, Clemens Valens, Brian Tristam Williams | **Regular Contributors:** David Ashton, Tam Hanna, Ilse Joostens, Prof. Dr. Martin Ossmann, Alfred Rosenkränzer | **Graphic Design & Prepress:** Harmen Heida, Sylvia Sopamena, Patrick Wielders | **Publisher:** Erik Jansen | **Technical questions:** editor@elektor.com

THIS EDITION

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

Regulars

3 **Colophon**

50 **Electronics Workspace Essentials**
Insights and Tips from Espressif Engineers

106 **Your AIoT Solution Provider**
Insights From Espressif

138 **Tech the Future: Where Is Smart Home IoT Headed?**

Features

13 **ESP Launchpad Tutorial**
From Zero to Flashing in Minutes

28 **From Idea to Circuit With the ESP32-S3**
A Guide to Prototyping With Espressif Chips

35 **AIoT Chip Innovation**
An Interview With Espressif CEO Teo Swee-Ann

40 **Simulate ESP32 with Wokwi**
Your Project's Virtual Twin

46 **Trying Out the ESP32-S3-BOX-3**
A Comprehensive AIoT Development Platform

53 **The ESP RainMaker Story**
How We Built "Your" IoT Cloud

56 **Assembling the Elektor Cloc 2.0 Kit**
An Elektor Product Unboxed by Espressif

59 **Unleashing the ESP32-P4**
The Next Era of Microcontrollers

64 **Rust + Embedded**
A Development Power Duo

70 **Who Are the Rust-Dacious Embedded Developers?**
How Espressif Is Cultivating Embedded Rust for the ESP32

74 **Espressif's Series of SoCs**

76 **Building a PLC with Espressif Solutions**
With the Capabilities and Functionality of the ISOBUS Protocol

78 **The ESP32-S3 VGA Board**
Bitluni's Exciting Journey Into Product Design

100 **Interview With Arduino on the Nano ESP32**

110 **Streamlining MCU Development With ESP-IDF Privilege Separation**

132 **The Smart Home Leaps Forward with Matter**
Unlocking Smart Home IoT Potential

Industry

89 **A Simpler and More Convenient Life**
An Amateur Project Based on the Espressif ESP8266 Module

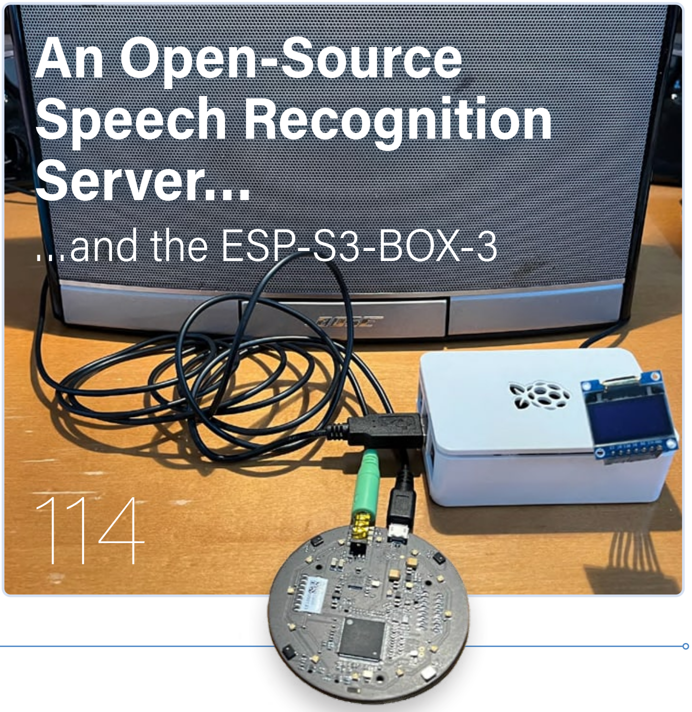
90 **How to Build IoT Apps without Software Expertise**
With Blynk IoT Platform and Espressif Hardware

92 **Building a Smart User Interface on ESP32**

94 **Quick & Easy IoT Development with M5Stack**

99 **A Value-Added Distributor for IoT and More**

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



Projects

6 **A Color E-Ink Wi-Fi Picture Frame**

16 **ESP32 and ChatGPT**
On the Way to a Self-Programming System...

24 **Walkie-Talkie with ESP-NOW**
Not Quite Wi-Fi, Not Quite Bluetooth, Either...

80 **Acoustic Fingerprinting on ESP32**
Song Recognition With Open-Source Project Olaf

84 **Circular Christmas Tree 2023**
A High-Tech Way to Celebrate the Holiday Season

96 **Prototyping an ESP32-Based Energy Meter**

114 **An Open-Source Speech Recognition Server...**
...and the ESP32-S3-BOX-3

119 **The Thinking Eye**
Facial Recognition and More Using the ESP32-S3-EYE

126 **ESP32-C2-Based Coin Cell Switch**
Design and Performance Evaluation

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

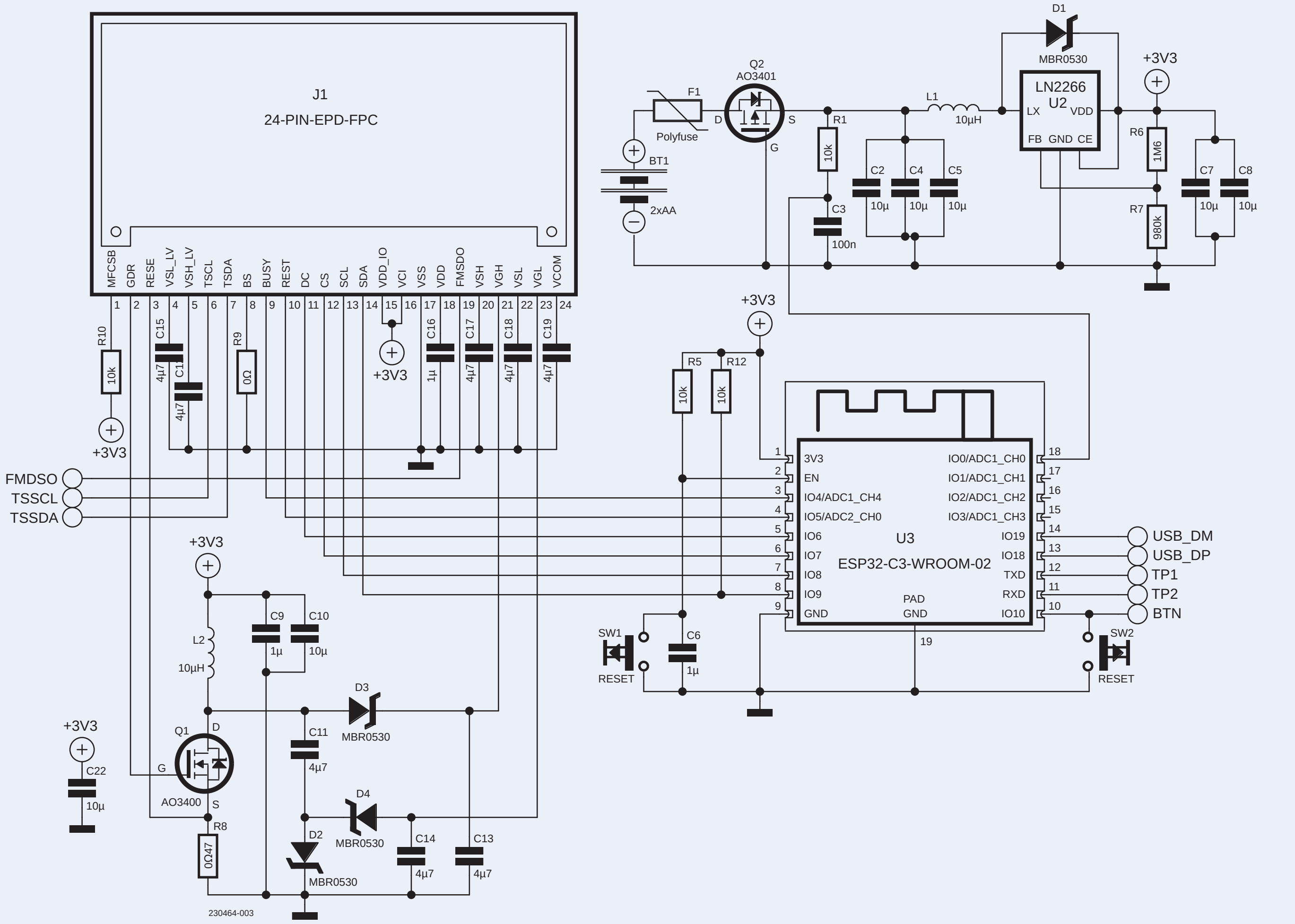
Figure 1: The 5.65-inch, seven-color, 640×448 display from Waveshare. (Source: Waveshare)

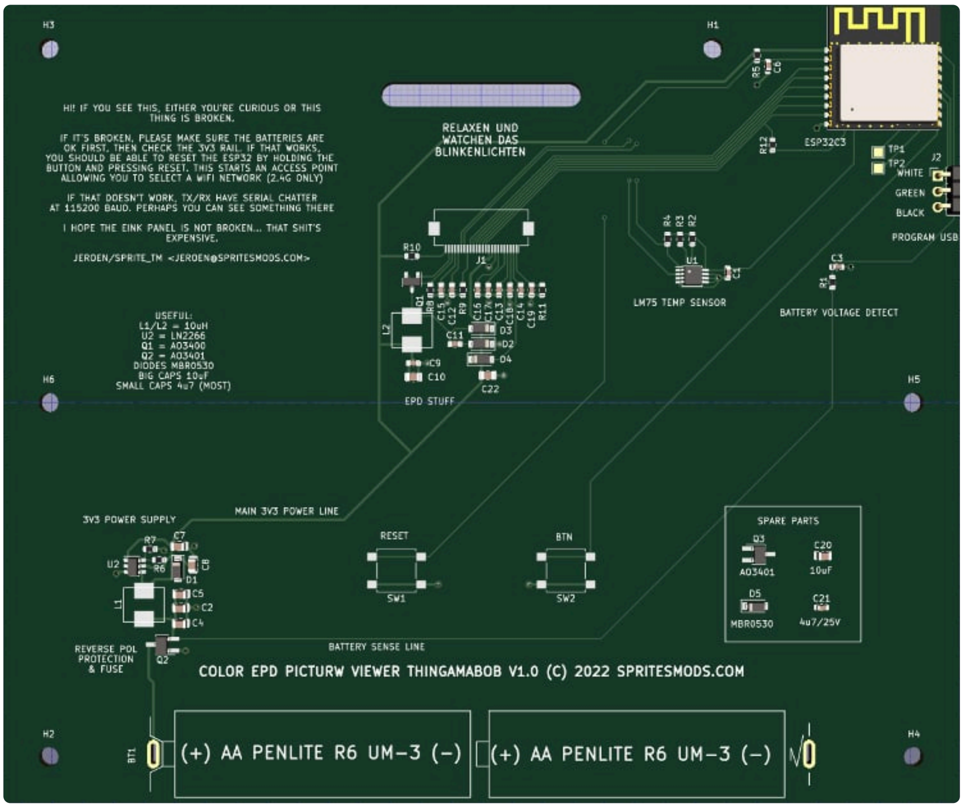
any new images. It would then display one on the E-ink display. If it managed to download new images, it would show one of those, else it would continue to show an old image. It would then go back to sleep for 24 hours. Picking a chip for this was not difficult, as I happen to work for Espressif, a company 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.

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.





HI! IF YOU SEE THIS, EITHER YOU'RE CURIOUS OR THIS THING IS BROKEN.

IF IT'S BROKEN, PLEASE MAKE SURE THE BATTERIES ARE OK FIRST. THEN CHECK THE 3V3 RAIL. IF THAT WORKS, YOU SHOULD BE ABLE TO RESET THE ESP32 BY HOLDING THE BUTTON AND PRESSING RESET. THIS STARTS AN ACCESS POINT ALLOWING YOU TO SELECT A WIFI NETWORK (2.4G ONLY)

IF THAT DOESN'T WORK, TX/RX HAVE SERIAL CHATTER AT 115200 BAUD. PERHAPS YOU CAN SEE SOMETHING THERE

I HOPE THE EINK PANEL IS NOT BROKEN... THAT SHIT'S EXPENSIVE.

JEROEN/SPRITE_TM <JEROEN@SPRITESMODS.COM>

USEFUL:
L1/L2 = 10uH
U2 = LN2266
Q1 = AO3400
Q2 = AO3401
DIODES MBR0530
BIG CAPS 10uF
SMALL CAPS 4u7 (MOST)

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

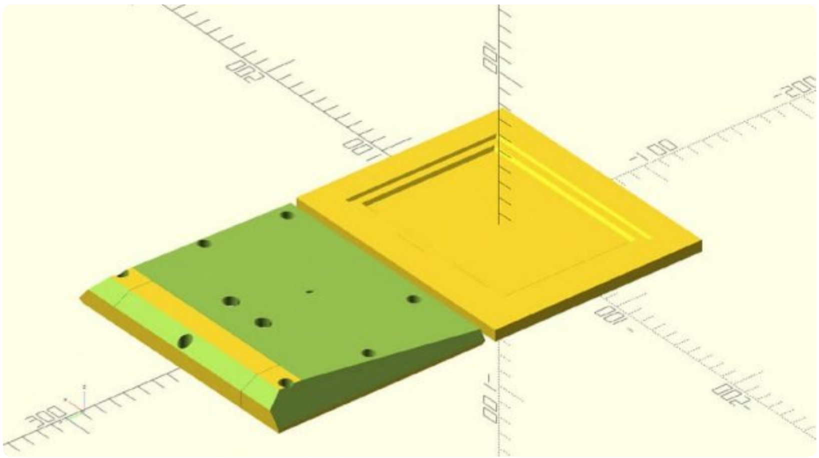


Figure 4: The case structure, designed with OpenSCAD.

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.

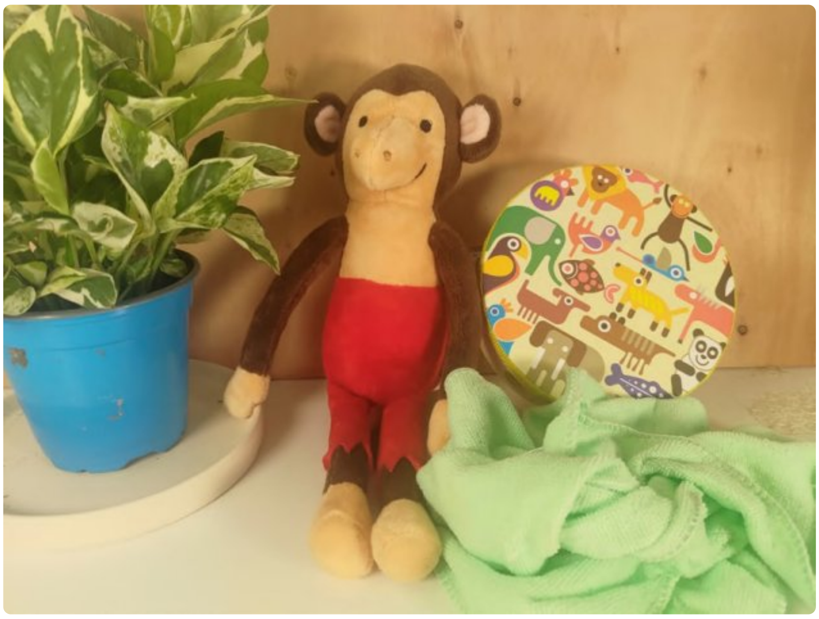
In the back, there's the battery component. It's opened by unscrewing one screw (I didn't want to rely on fragile 3D-printed clips to keep it in place) and, as the silkscreen of the PCB has the correct battery orientation on it, replacing batteries should be simple.

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"

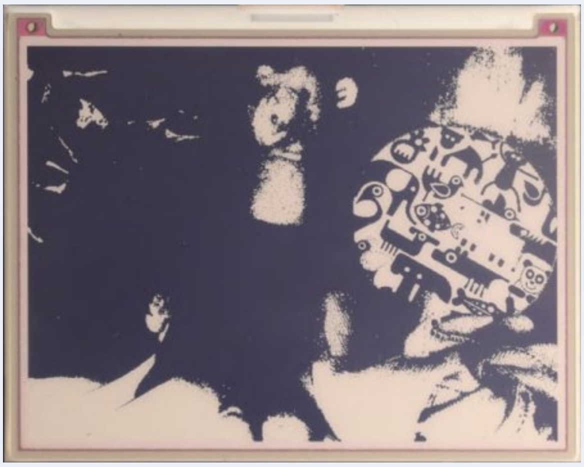


Figure 6: First converstion attempt, using 100% black and 100% white.

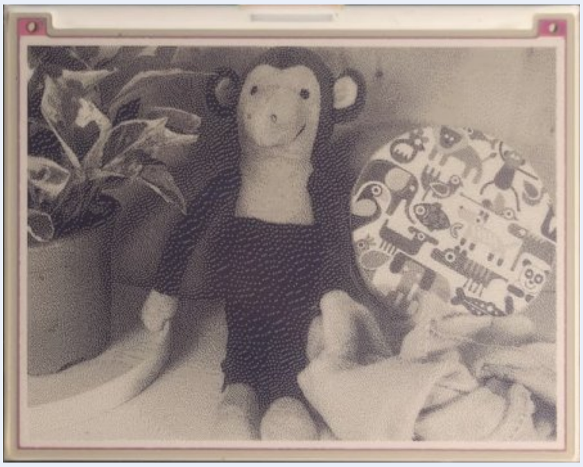


Figure 7: Black-and-white conversion with a diffusion process.

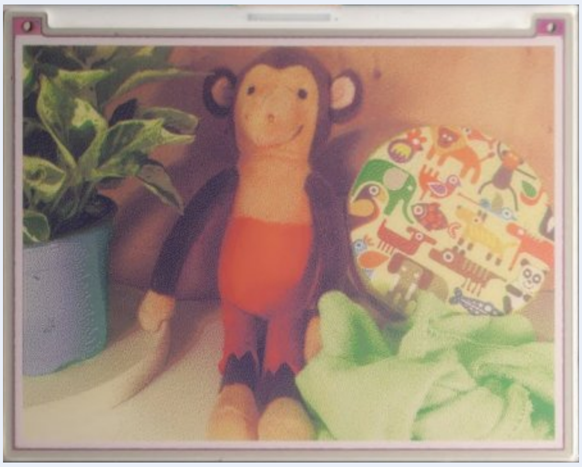


Figure 8: The image from Figure 7 with the addition of RGB values (see text).

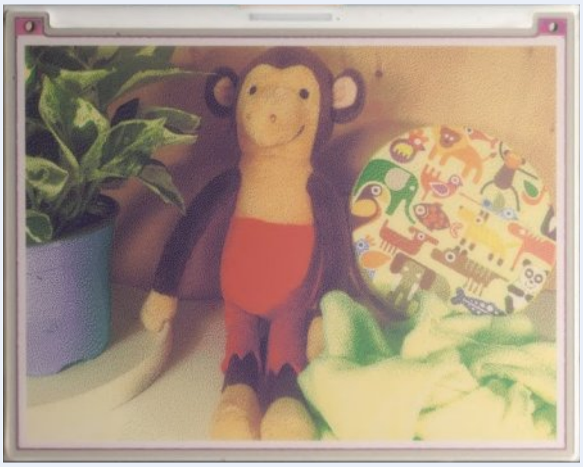


Figure 9: The sample image with the application of the CIEDE2000 standard.

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**.

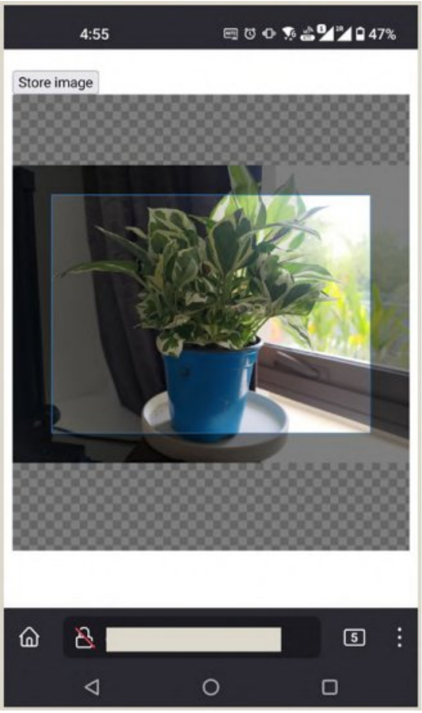


Figure 10: Quick-crop function for mobile use.

◄

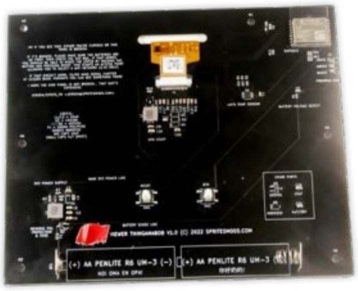


Figure 11: The completed PCB assembly.

◄

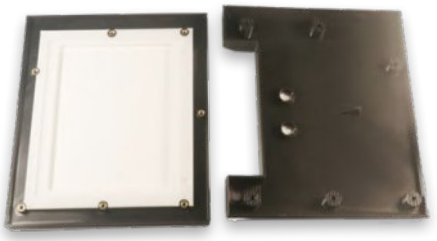


Figure 12: The two sections of the case realized for this project.

◄

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.

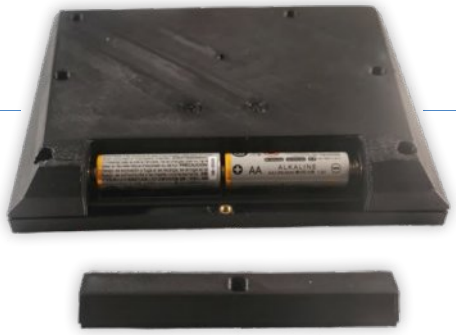


Figure 13: The battery compartment with its own little flap.

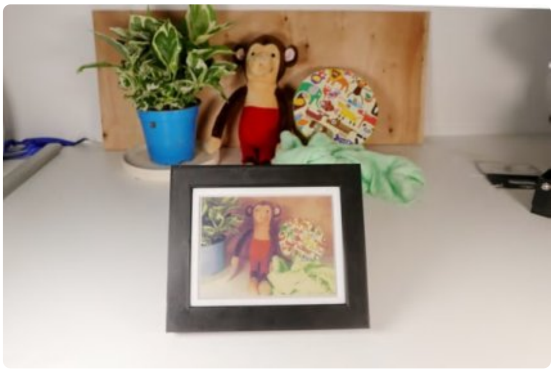


Figure 14: The final result, shown next to the original.

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.

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.

> “
>
> *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.*

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

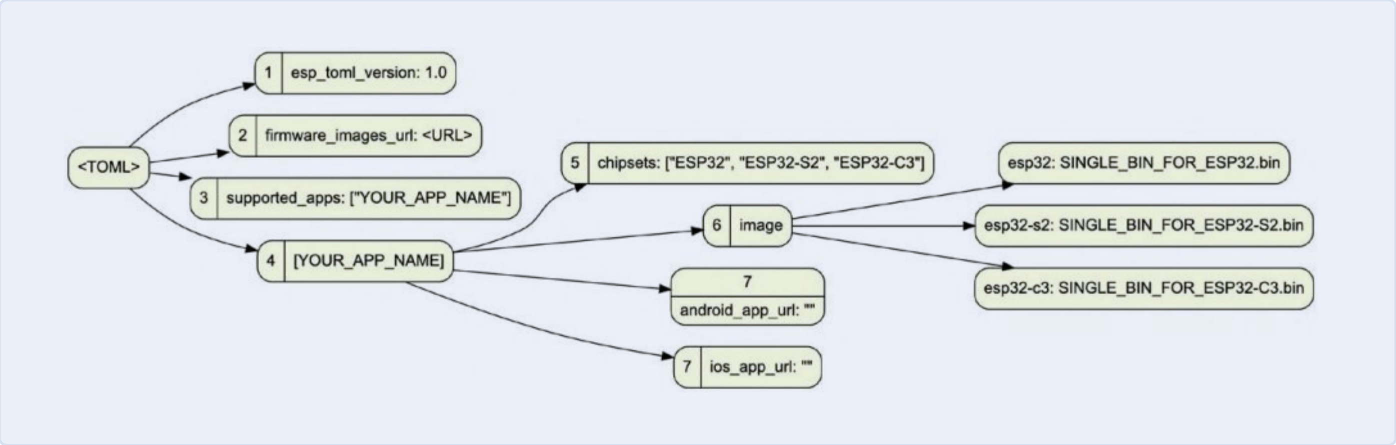
Once your firmware has been finalized, the next step is to build it for all the targets that you wish to support and convert them to single binaries. Fret not, `esptool.py` (Espressif’s one-stop-shop Python utility for all things related to flashing) provides a convenient `merge_bin` option that can 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.

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.



<TOML>
1 esp_toml_version: 1.0
2 firmware_images_url: <URL>
3 supported_apps: ["YOUR_APP_NAME"]
4 [YOUR_APP_NAME]
5 chipsets: ["ESP32", "ESP32-S2", "ESP32-C3"]
6 image
7 android_app_url: ""
7 ios_app_url: ""
esp32: SINGLE_BIN_FOR_ESP32.bin
esp32-s2: SINGLE_BIN_FOR_ESP32-S2.bin
esp32-c3: SINGLE_BIN_FOR_ESP32-C3.bin

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:

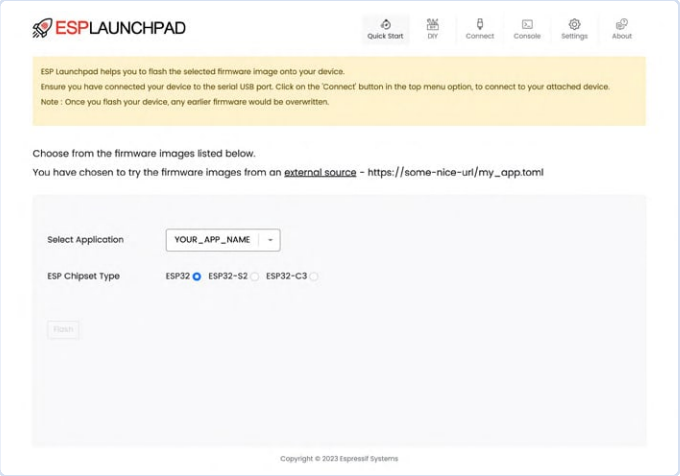
https://espressif.github.io/esp-launchpad/?flashConfigURL=https://some-nice-url/my_app.toml

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**).



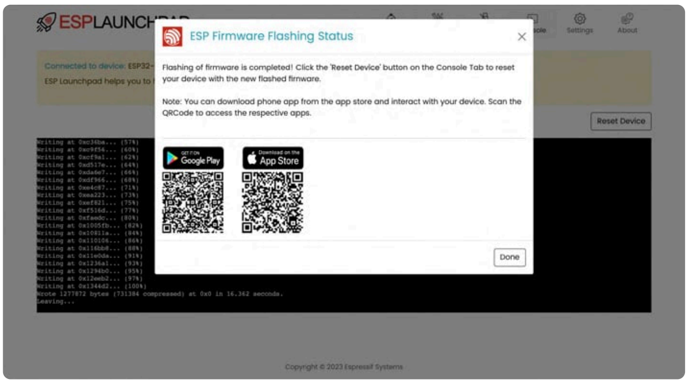
ESP Launchpad helps you to flash the selected firmware image onto your device.
Ensure you have connected your device to the serial USB port. Click on the 'Connect' button in the top menu option, to connect to your attached device.
Note : Once you flash your device, any earlier firmware would be overwritten.

Choose from the firmware images listed below.
You have chosen to try the firmware images from an external source - https://some-nice-url/my_app.toml

Select Application: YOUR_APP_NAME
ESP Chipset Type: ESP32 ● ESP32-S2 ○ ESP32-C3 ○

Copyright © 2023 Espressif Systems

Figure 2: ESP-Launchpad with example config loaded.


ESPLAUNCHPAD
Settings
About
Connected to device: ESP32-
ESP Launchpad helps you to
Reset Device
GET IT ON Google Play
Download on the App Store
Done
Copyright © 2023 Espressif Systems

ESP Firmware Flashing Status

Flashing of firmware is completed! Click the 'Reset Device' button on the Console Tab to reset your device with the new flashed firmware.

Note: You can download phone app from the app store and interact with your device. Scan the QRCode to access the respective apps.

```
Writing at 0xc36ba... (57%)
Writing at 0xc9f56... (60%)
Writing at 0xcf9a1... (62%)
Writing at 0xd517e... (64%)
Writing at 0xda6e7... (66%)
Writing at 0xdf966... (68%)
Writing at 0xe4c87... (71%)
Writing at 0xea223... (73%)
Writing at 0xef821... (75%)
Writing at 0xf516d... (77%)
Writing at 0xfaedc... (80%)
Writing at 0x1005fb... (82%)
Writing at 0x10811a... (84%)
Writing at 0x110106... (86%)
Writing at 0x116bb8... (88%)
Writing at 0x11e0da... (91%)
Writing at 0x1236a1... (93%)
Writing at 0x1294b0... (95%)
Writing at 0x12eeb2... (97%)
Writing at 0x1344d2... (100%)
Wrote 1277872 bytes (731384 compressed) at 0x0 in 16.362 seconds.
Leaving...
```

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]

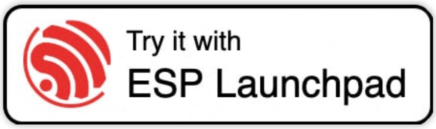


Figure 4: A badge to add!

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...

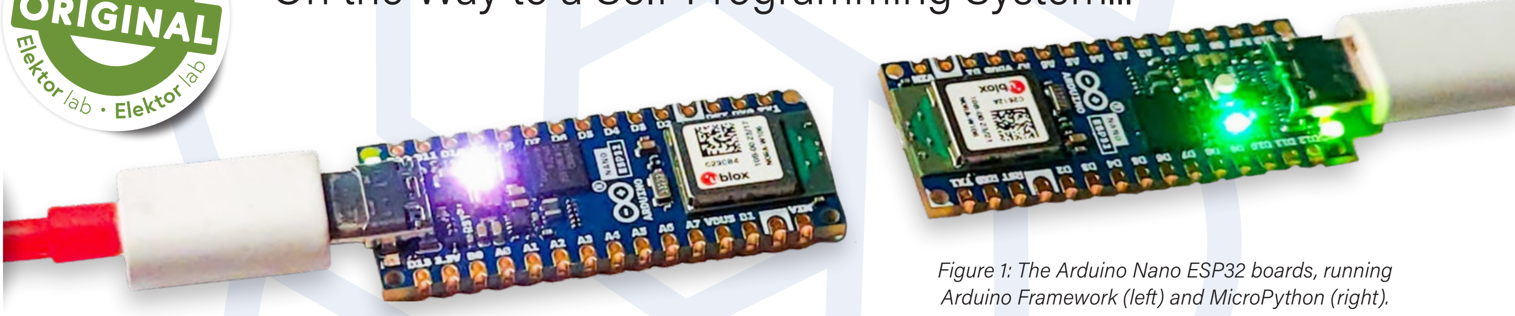


Figure 1: The Arduino Nano ESP32 boards, running Arduino Framework (left) and MicroPython (right).

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-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
```

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.

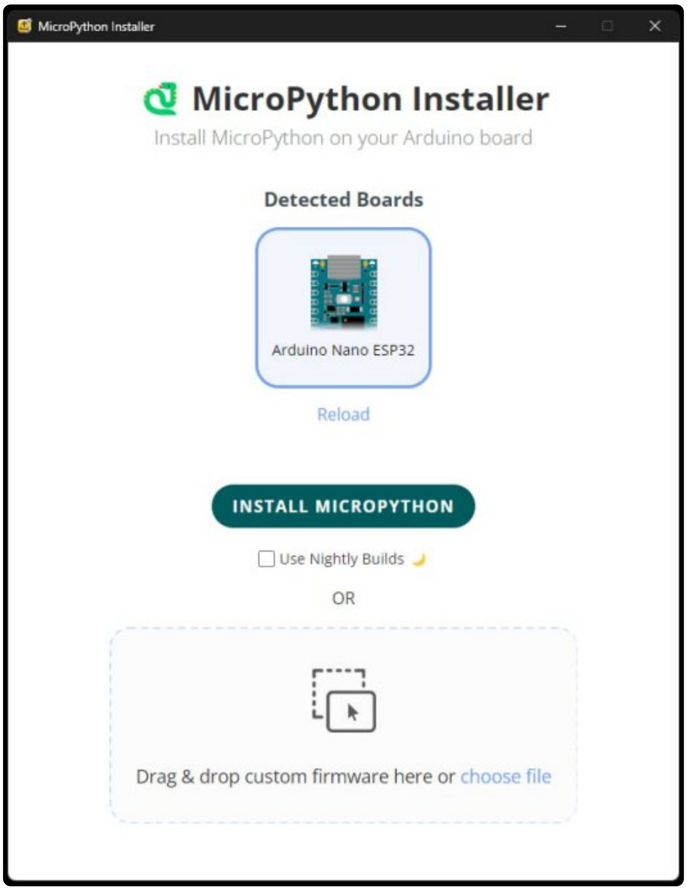


Figure 2: MicroPython Firmware tool by Arduino Labs.

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")
```

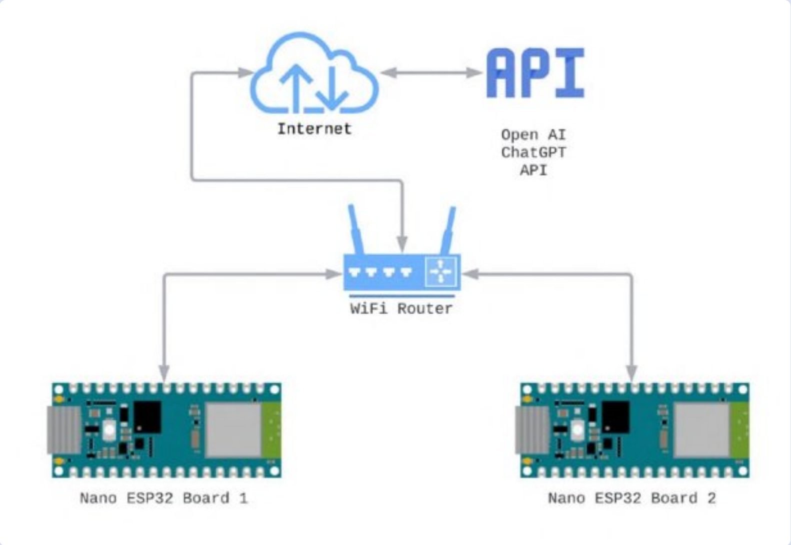
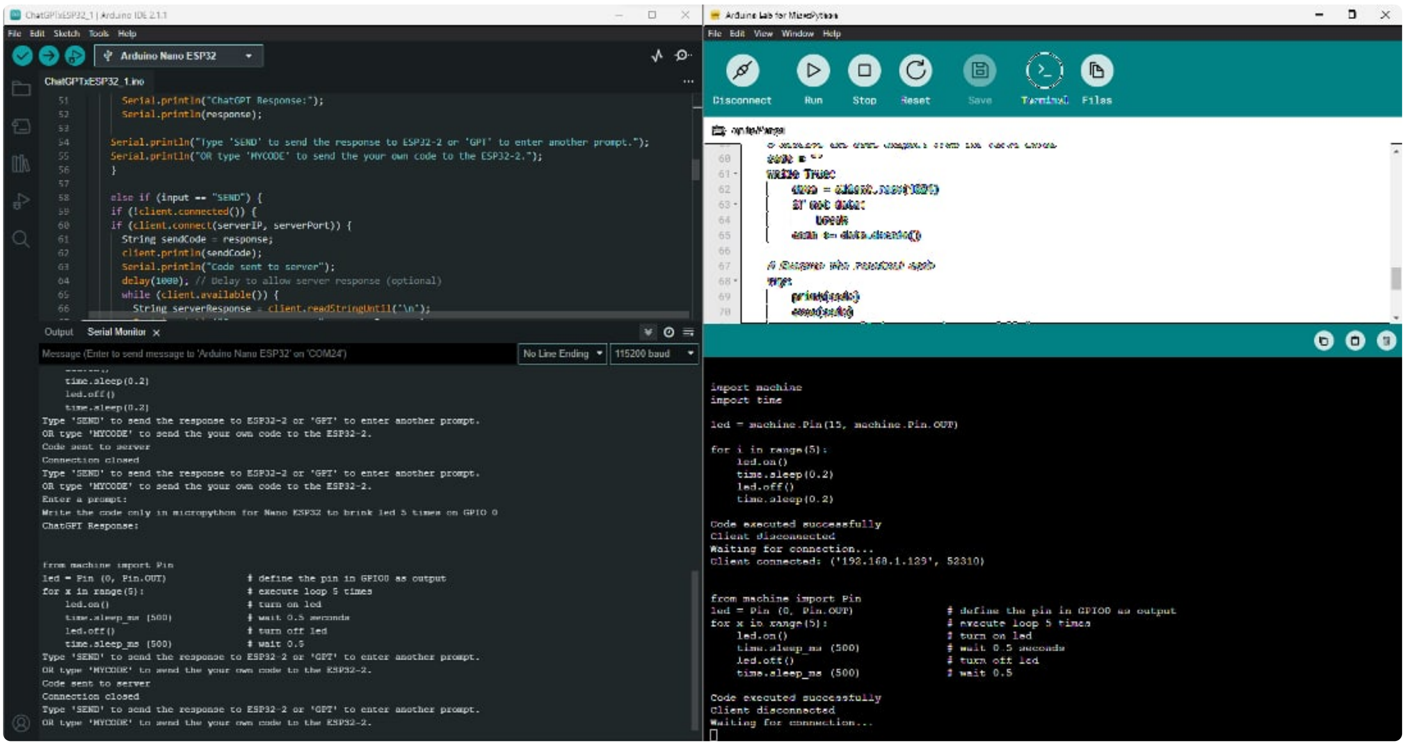


Figure 3: Communication between the two Arduino Nano ESP32 boards and ChatGPT API.


ChatGPTxESP32_1 | Arduino IDE 2.1.1
File Edit Sketch Tools Help
Arduino Nano ESP32
ChatGPTxESP32_1.ino
Output
Serial Monitor
Message (Enter to send message to 'Arduino Nano ESP32' on 'COM24')
No Line Ending
115200 baud
Arduino Lab for MicroPython
File Edit View Window Help
Disconnect
Run
Stop
Reset
Save
Terminal
Files

```
        Serial.println("ChatGPT Response:");
        Serial.println(response);

      Serial.println("Type 'SEND' to send the response to ESP32-2 or 'GPT' to enter another prompt.");
      Serial.println("OR type 'MYCODE' to send the your own code to the ESP32-2.");
      }

      else if (input == "SEND") {
      if (!client.connected()) {
      if (client.connect(serverIP, serverPort)) {
        String sendCode = response;
        client.println(sendCode);
        Serial.println("Code sent to server");
        delay(1000); // Delay to allow server response (optional)
        while (client.available()) {
          String serverResponse = client.readStringUntil('\n');
```

```
    time.sleep(0.2)
    led.off()
    time.sleep(0.2)
Type 'SEND' to send the response to ESP32-2 or 'GPT' to enter another prompt.
OR type 'MYCODE' to send the your own code to the ESP32-2.
Code sent to server
Connection closed
Type 'SEND' to send the response to ESP32-2 or 'GPT' to enter another prompt.
OR type 'MYCODE' to send the your own code to the ESP32-2.
Enter a prompt:
Write the code only in micropython for Nano ESP32 to brink led 5 times on GPIO 0
ChatGPT Response:


from machine import Pin
led = Pin (0, Pin.OUT)                # define the pin in GPIO0 as output
for x in range(5):                    # execute loop 5 times
    led.on()                          # turn on led
    time.sleep_ms (500)               # wait 0.5 seconds
    led.off()                         # turn off led
    time.sleep_ms (500)               # wait 0.5
Type 'SEND' to send the response to ESP32-2 or 'GPT' to enter another prompt.
OR type 'MYCODE' to send the your own code to the ESP32-2.
Code sent to server
Connection closed
Type 'SEND' to send the response to ESP32-2 or 'GPT' to enter another prompt.
OR type 'MYCODE' to send the your own code to the ESP32-2.
```

```
code = ""
while True:
    data = client.recv(1024)
    if not data:
        break
    code += data.decode()

# Execute the received code
try:
    print(code)
    exec(code)
```

```
import machine
import time

led = machine.Pin(15, machine.Pin.OUT)

for i in range(5):
    led.on()
    time.sleep(0.2)
    led.off()
    time.sleep(0.2)

Code executed successfully
Client disconnected
Waiting for connection...
Client connected: ('192.168.1.129', 52310)


from machine import Pin
led = Pin (0, Pin.OUT)                 # define the pin in GPIO0 as output
for x in range(5):                     # execute loop 5 times
    led.on()                           # turn on led
    time.sleep_ms (500)                # wait 0.5 seconds
    led.off()                          # turn off led
    time.sleep_ms (500)                # wait 0.5

Code executed successfully
Client disconnected
Waiting for connection...
```

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.

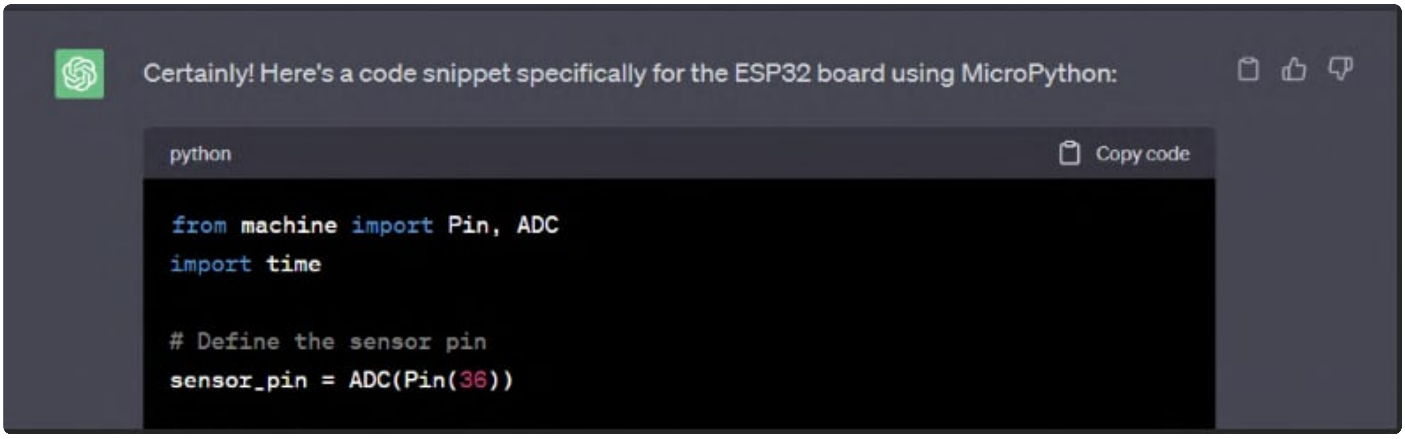
After executing the instructions, Nano ESP32-2 sends a response back to Nano ESP32-1 over the established socket connection. This response serves as confirmation that the received code snippet was successfully executed. It ensures that Nano ESP32-1 is aware of the completion of the requested task.

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.


Certainly! Here's a code snippet specifically for the ESP32 board using MicroPython:

```
from machine import Pin, ADC
import time

# Define the sensor pin
sensor_pin = ADC(Pin(36))
```

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.

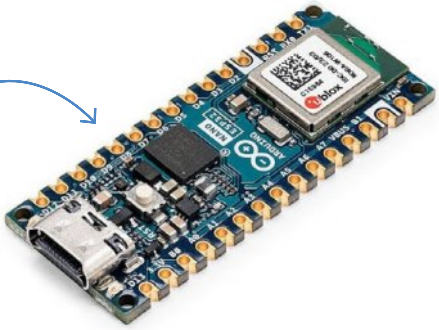
Let's explore a few more frameworks that can be used with the Nano ESP32 and their benefits:

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.

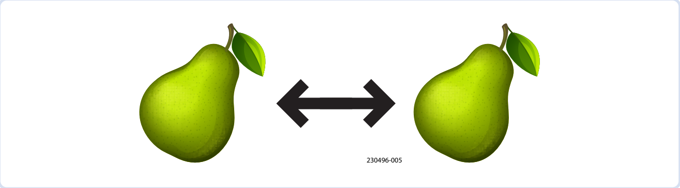


Figure 1: Pear-to-pear communication as shown here is not possible with Wi-Fi; a router is always needed between the two nodes.

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.

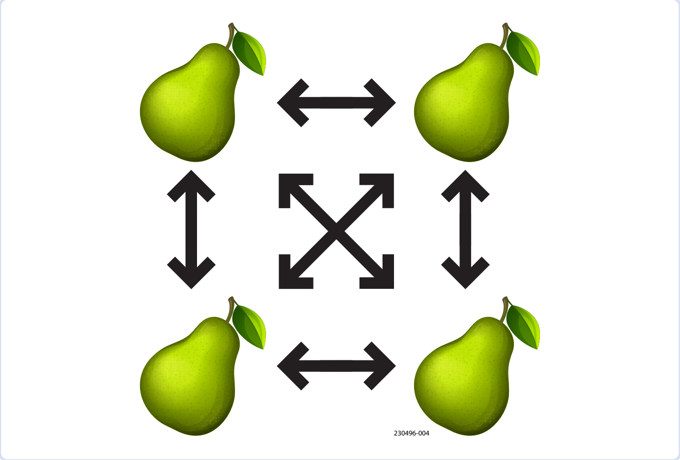


Figure 2: ESP-NOW supports many-to-many networking. In such a network, each node can talk to the other nodes directly without requiring a router.

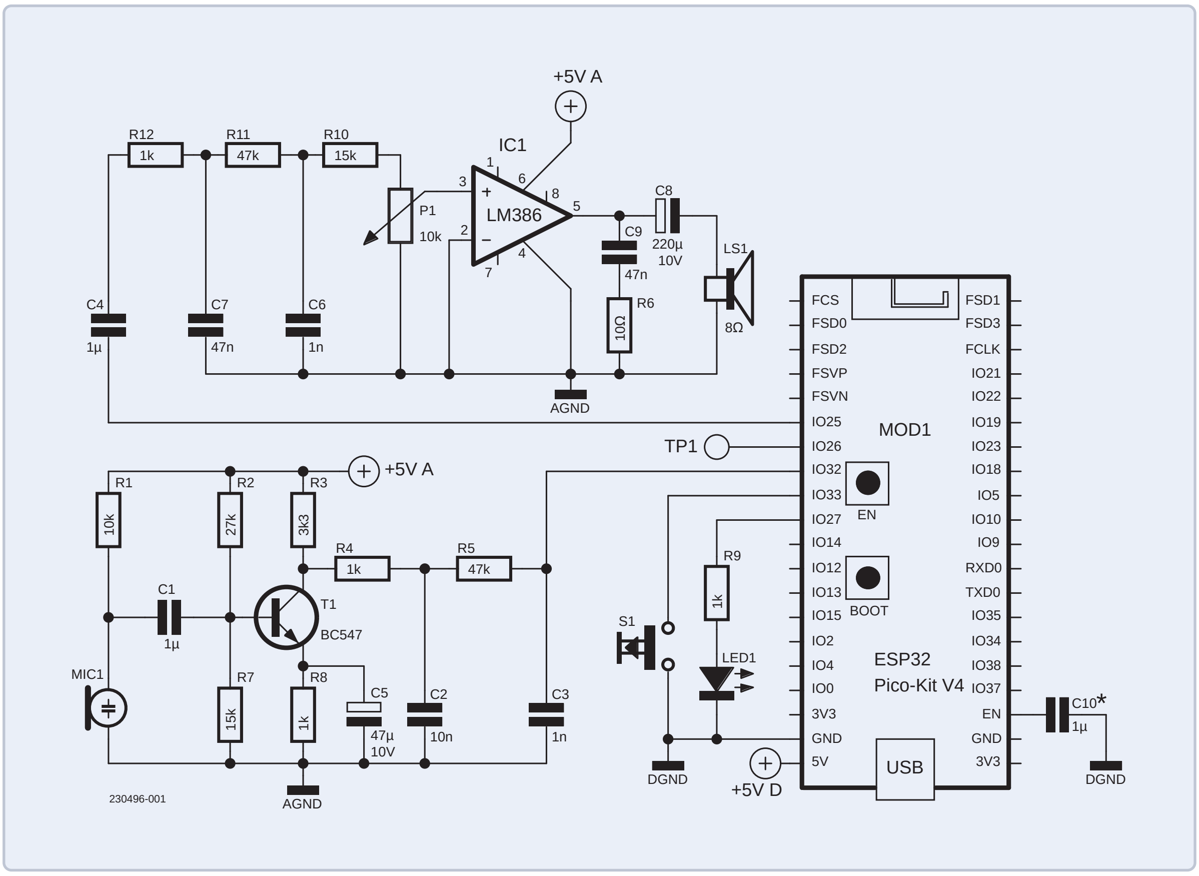


Figure 3: A simple microphone preamplifier on the input and a classic LM386 as power amplifier at the output. Note how the power supplies for the analog and digital parts are separated.

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.

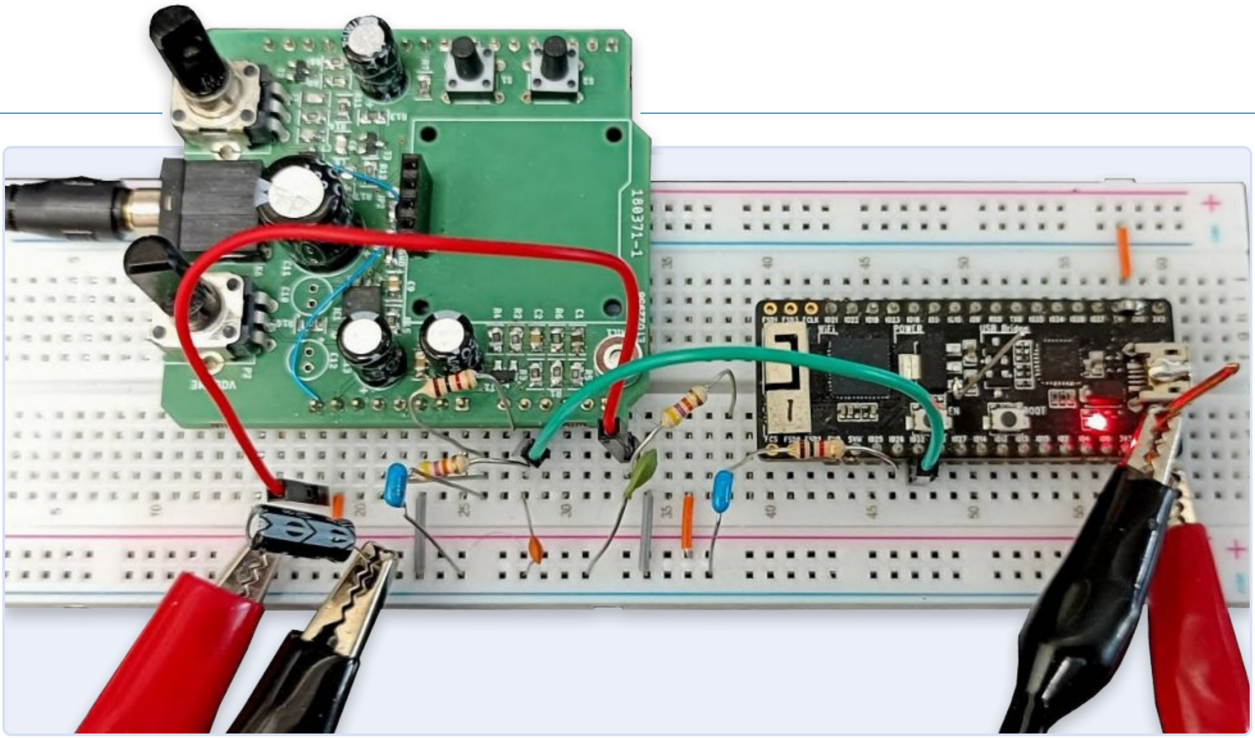


Figure 4: A proof-of-concept built on a breadboard with an ESP32-PICO-KIT and a slightly modified Elektor Snore Shield [2] for the input and output amplifiers. Note the two pairs of crocodile clips that provide the separate analog and digital power supplies.

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).

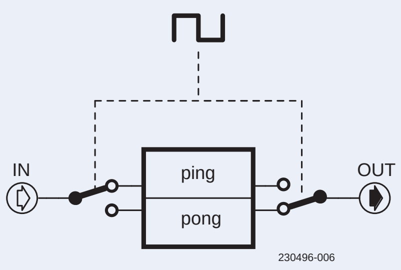


Figure 5: Double- or ping-pong buffering helps to avoid discontinuities in a data stream.

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 I²S. 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.

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

developer's zone

Tips & Tricks, Best Practices and Other Useful Information

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

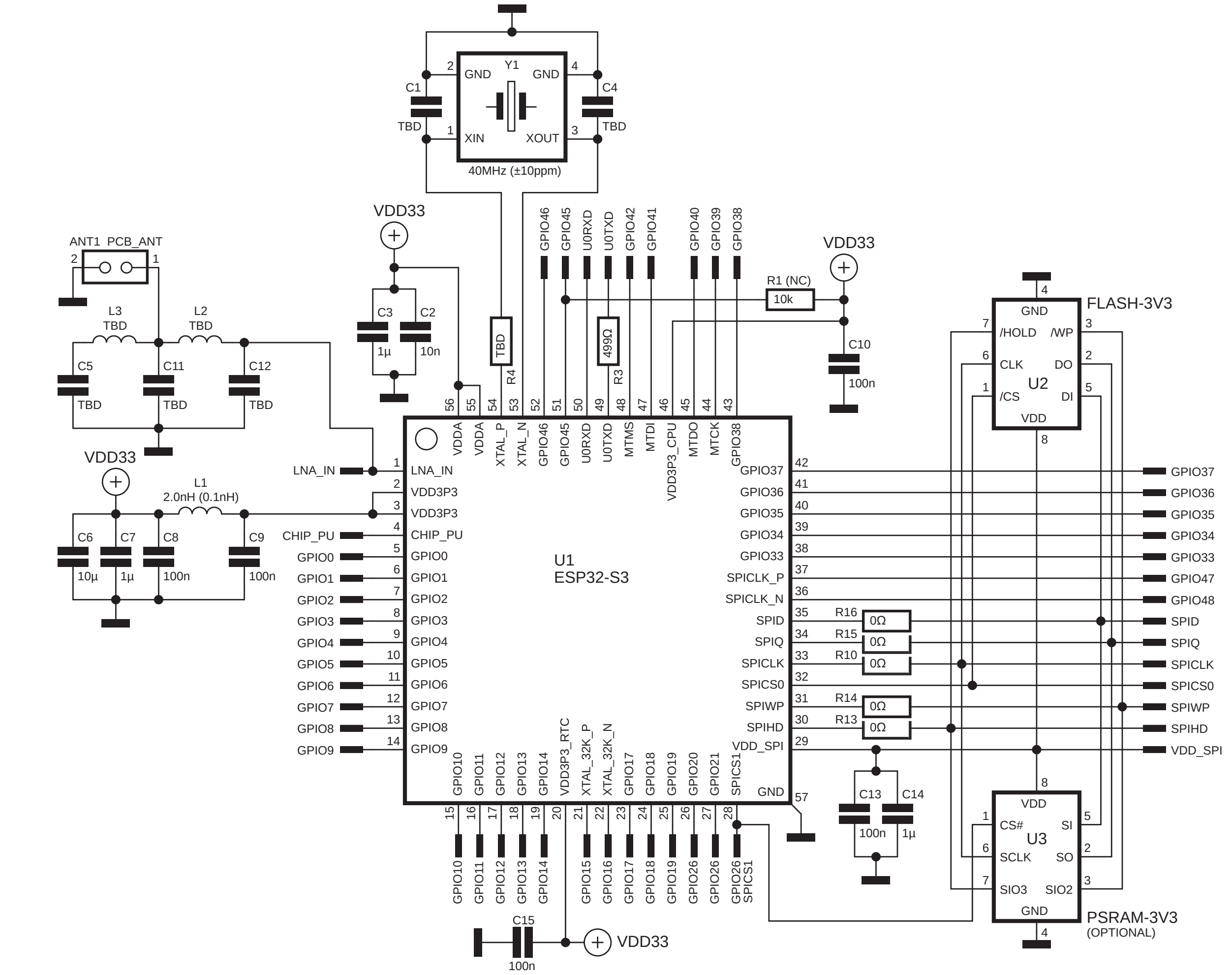


Figure 1: ESP32-S3 chip plus all the components it generally requires. Decoupling is pretty important here.

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.

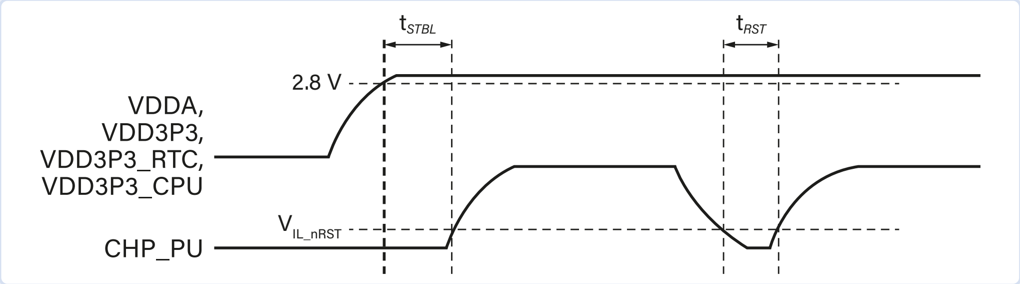
To get the best radio range and to make sure the device complies with EMC certification requirements, you need to make sure the impedances in the path from the ESP32-S3 to the antenna match. Generally, the transmission line (trace) between the ESP32 and the antenna has a 50 Ω impedance, but the ESP32-S3 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.

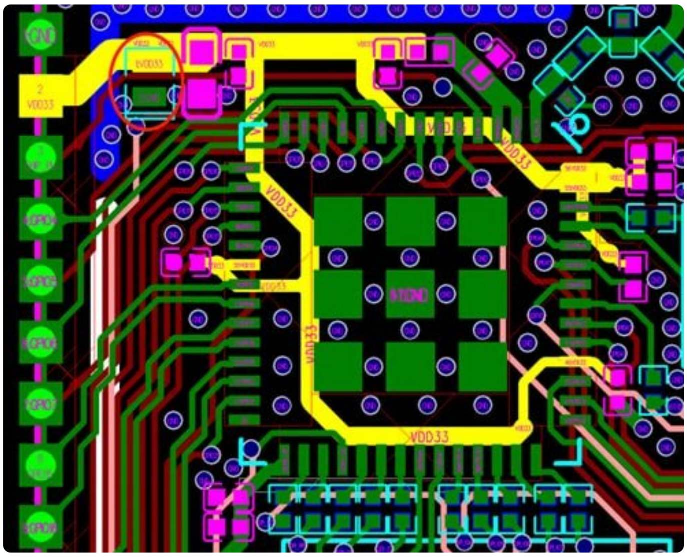


Figure 2: Each decoupling cap should be as close as possible to power pins.

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

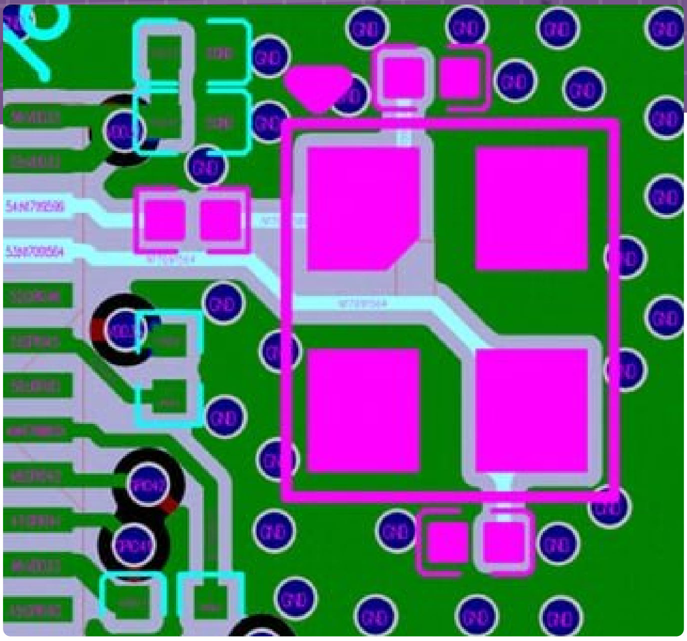


Figure 3: The two load capacitors should be placed on two sides of the crystal.

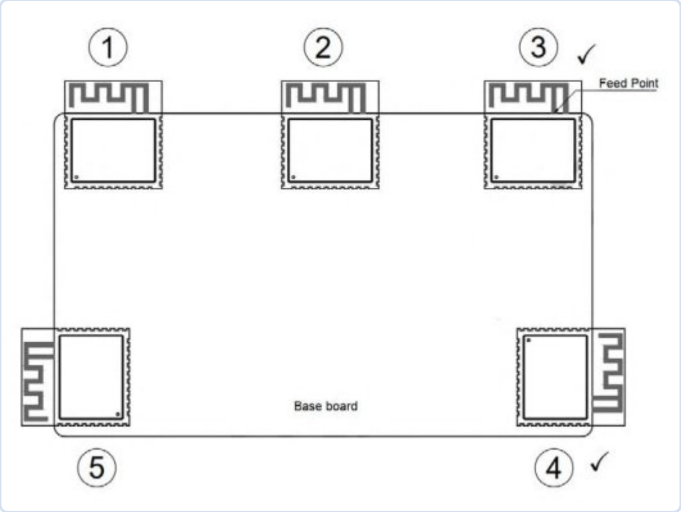


Figure 4: Positioning a module on a base board. (Source [6])

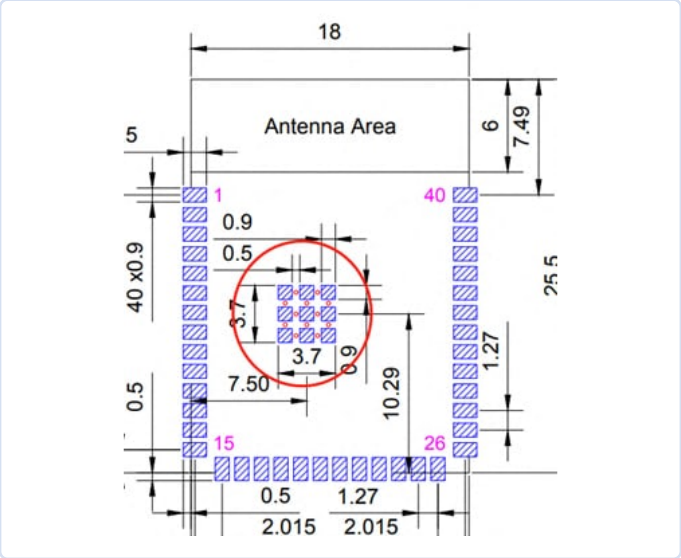
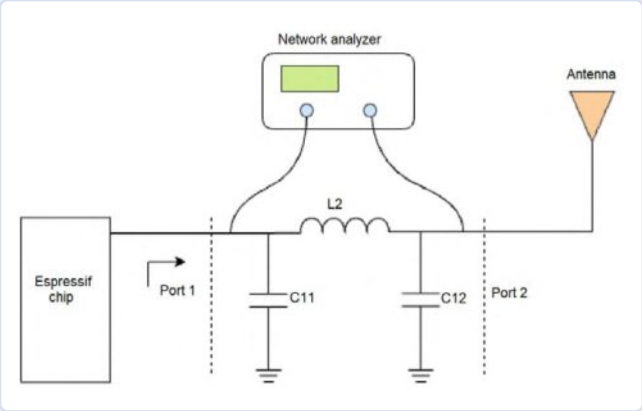


Figure 5: ESP32-S3-WROOM-1 recommended PCB land pattern. (Source: [6])

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.

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

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

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

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.

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.

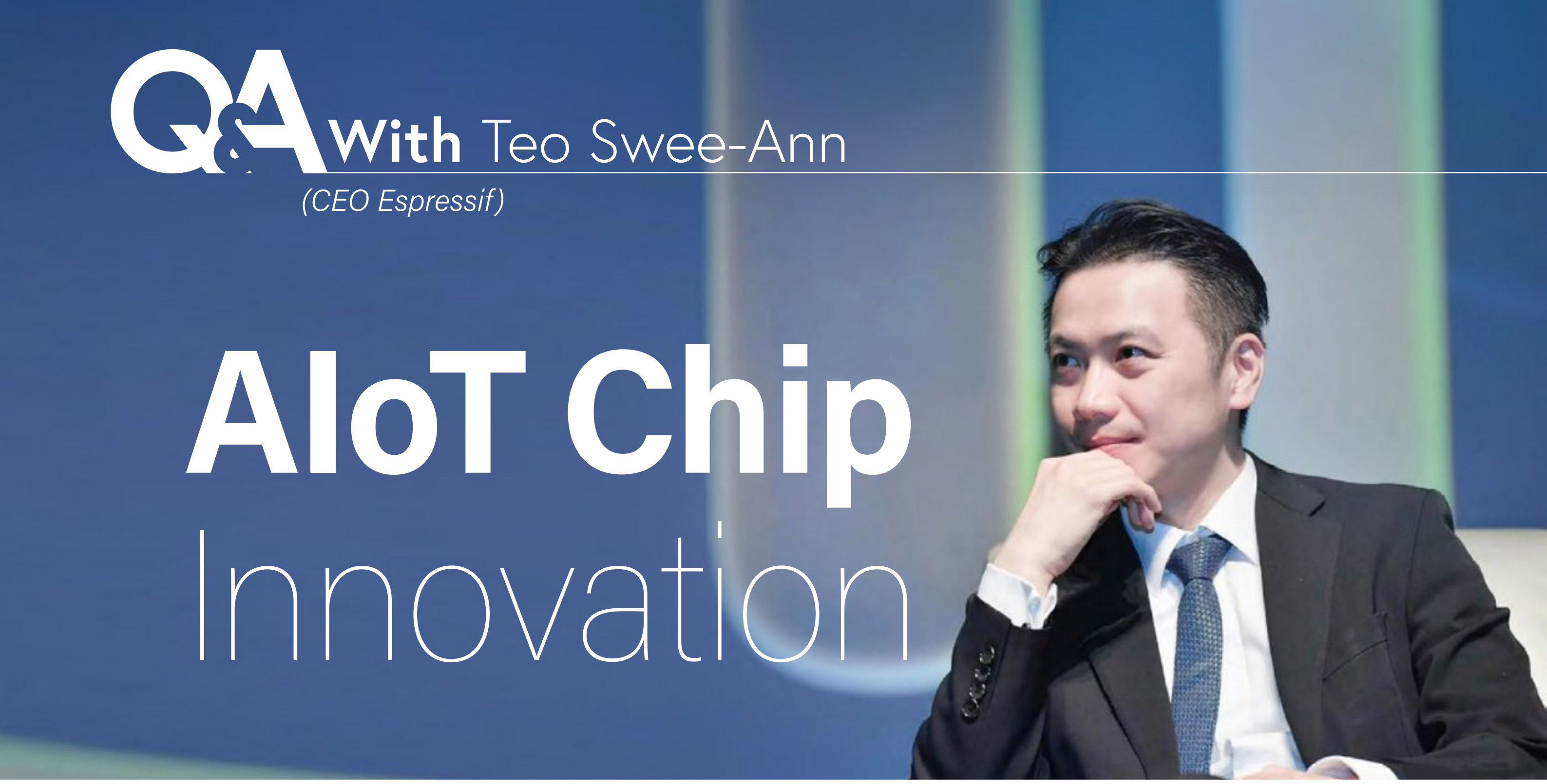
WEB LINKS

[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.

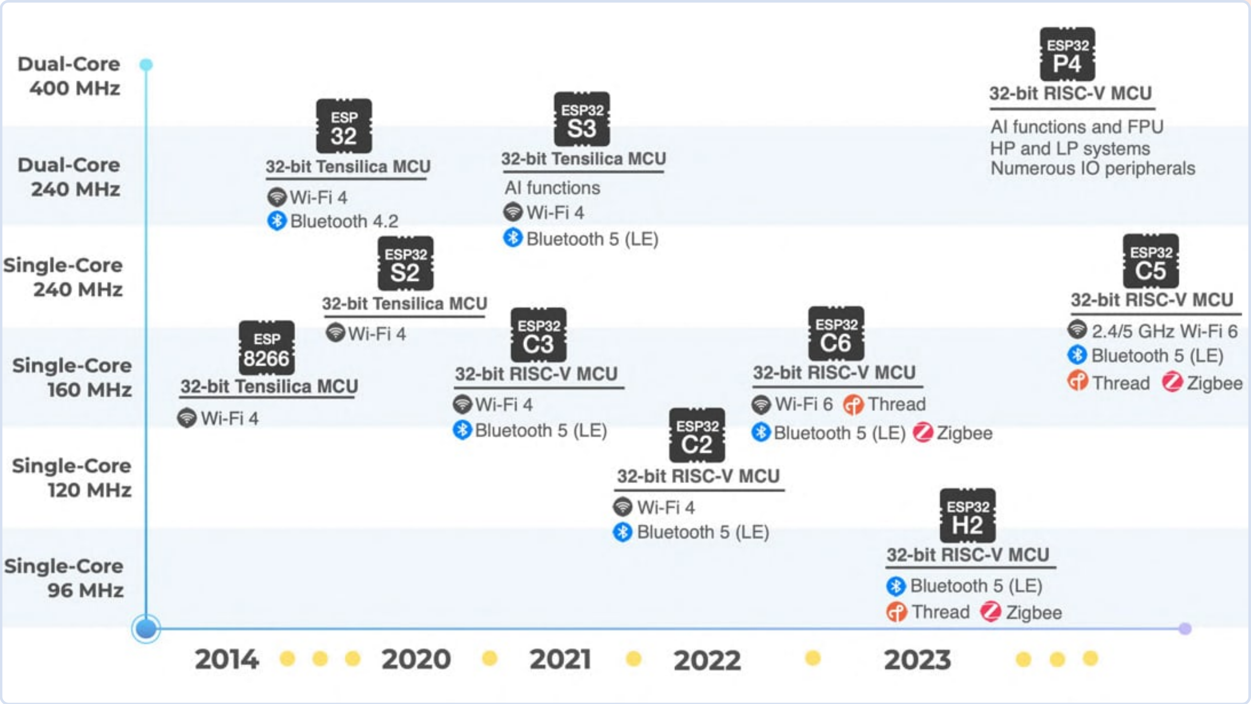
Lastly, the remaining problem was about how we could extract the singularities in the complex domain. I created an alternative technique, which I name "Cauchy Tomography" to extract these singularities, by using the Cauchy residue theorem and recovering their locations via the Generalized Pencil of Function 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."

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.

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.

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.

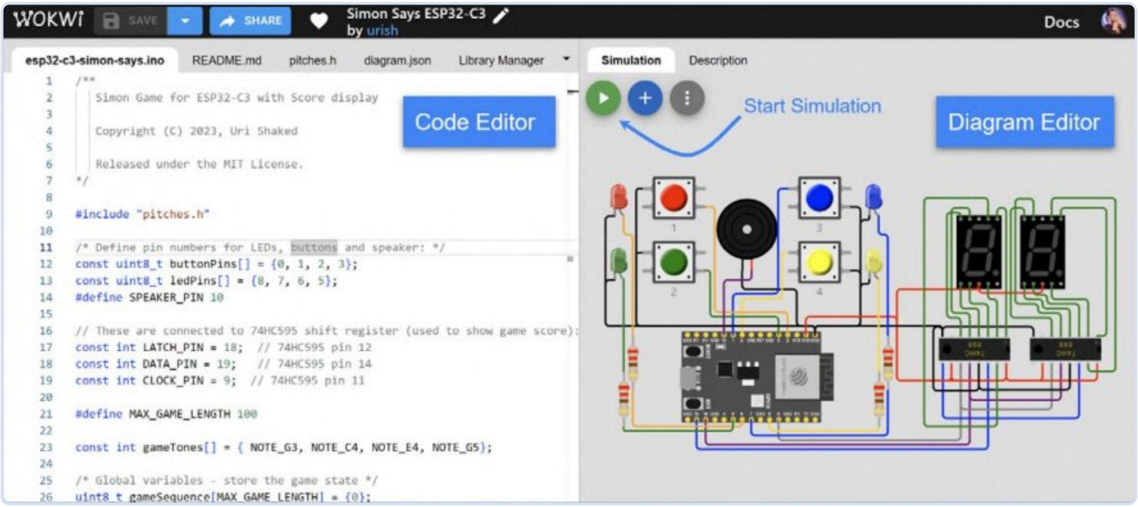
Wokwi enables you to create a digital twin of your project: You can simulate a variety of input and output devices [1], such as LCD screens, sensors, motors, LEDs, buttons, speakers, and potentiometers, and you can even create your own simulation models for new devices. The simulator allows you to iterate 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:

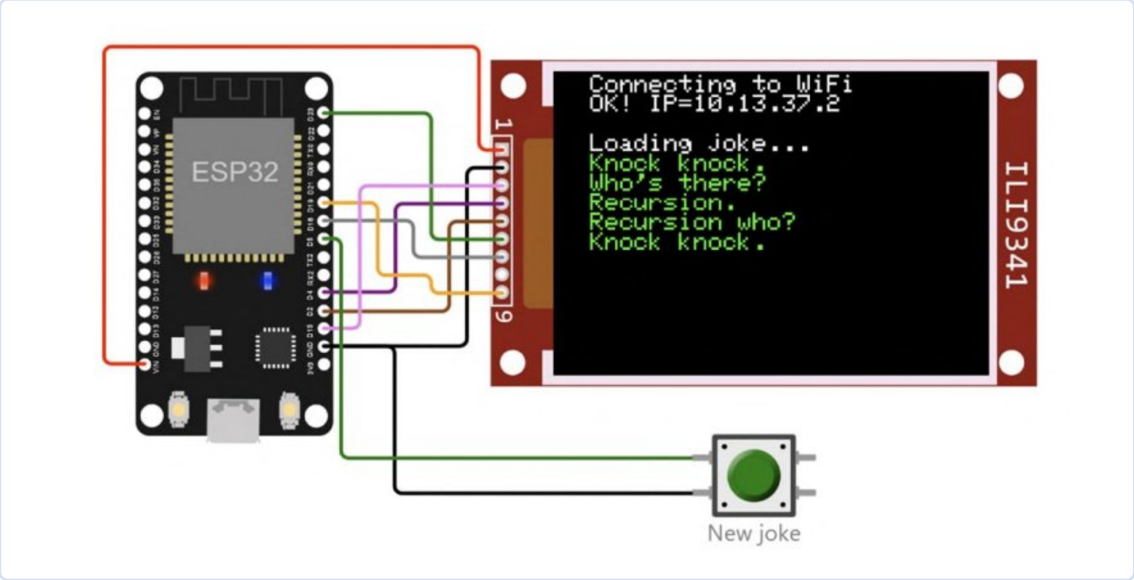
- MicroPython [2]
- ESP32 with Arduino Core [3]
- Rust [4]
- DeviceScript (TypeScript for ESP32) [5]



```
/**
  Simon Game for ESP32-C3 with Score display

  Copyright (C) 2023, Uri Shaked

  Released under the MIT License.
*/

#include "pitches.h"

/* Define pin numbers for LEDs, buttons and speaker: */
const uint8_t buttonPins[] = {0, 1, 2, 3};
const uint8_t ledPins[] = {8, 7, 6, 5};
#define SPEAKER_PIN 10

// These are connected to 74HC595 shift register (used to show game score):
const int LATCH_PIN = 18;  // 74HC595 pin 12
const int DATA_PIN = 19;   // 74HC595 pin 14
const int CLOCK_PIN = 9;  // 74HC595 pin 11

#define MAX_GAME_LENGTH 100

const int gameTones[] = { NOTE_G3, NOTE_C4, NOTE_E4, NOTE_G5};

/* Global variables - store the game state */
uint8_t gameSequence[MAX_GAME_LENGTH] = {0};
```

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

▶



```
Connecting to WiFi
OK! IP=10.13.37.2

Loading joke...
Knock knock.
Who's there?
Recursion.
Recursion who?
Knock knock.
```

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].

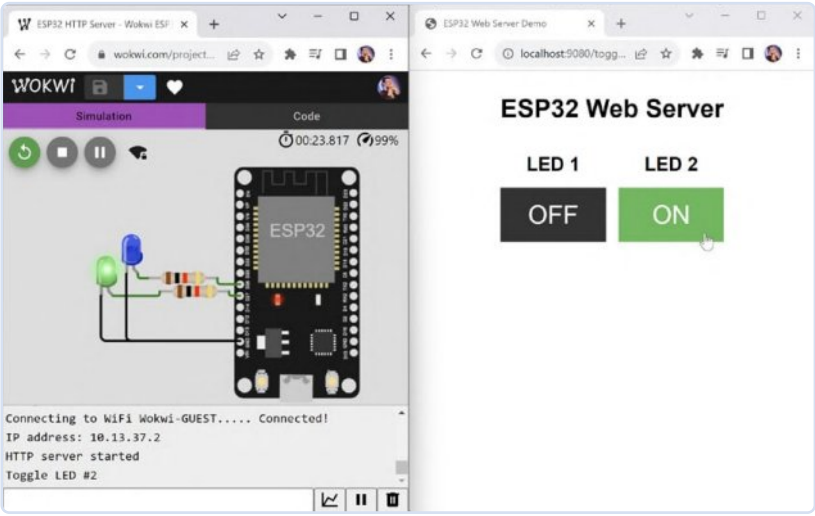
The Library Manager

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).

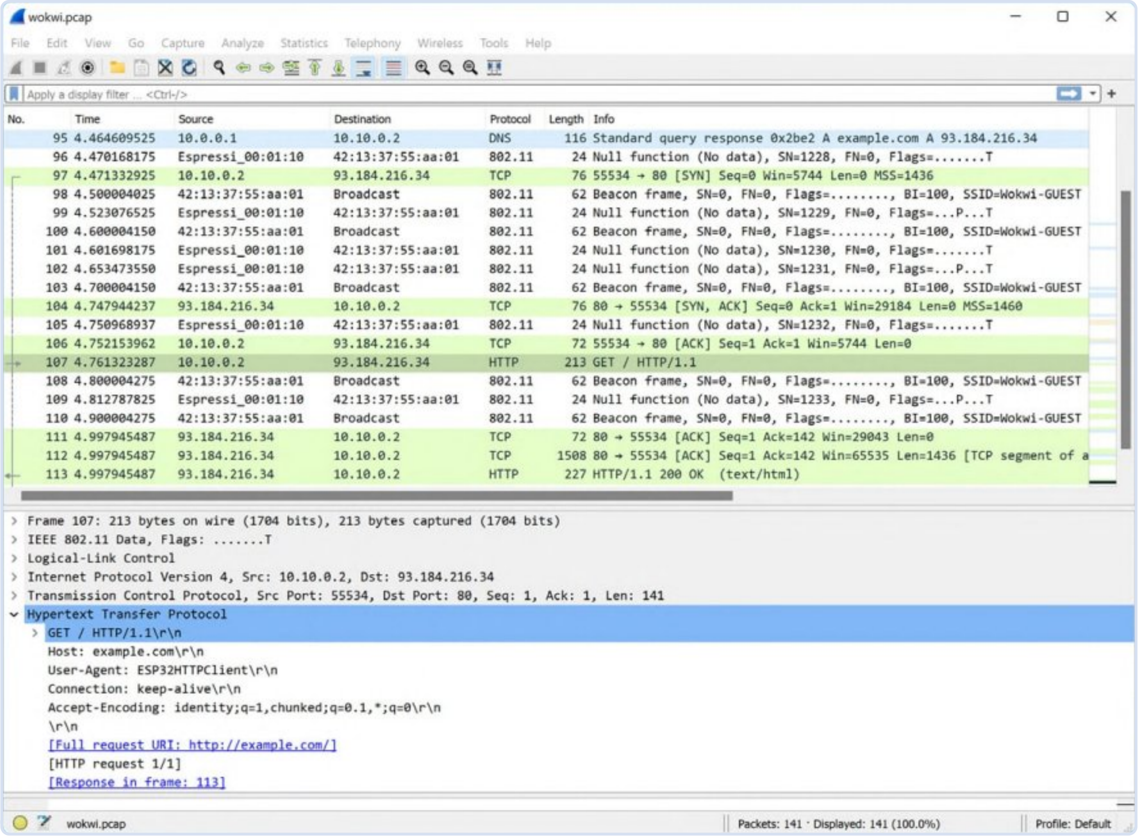
Wi-Fi Simulation

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.



```
Connecting to WiFi Wokwi-GUEST..... Connected!
IP address: 10.13.37.2
HTTP server started
Toggle LED #2
```

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



wokwi.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time         Source              Destination         Protocol Length Info
   95 4.464609525  10.0.0.1            10.10.0.2           DNS       116 Standard query response 0x2be2 A example.com A 93.184.216.34
   96 4.470168175  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1228, FN=0, Flags=.......T
   97 4.471332925  10.10.0.2           93.184.216.34       TCP        76 55534 → 80 [SYN] Seq=0 Win=5744 Len=0 MSS=1436
   98 4.500004025  42:13:37:55:aa:01   Broadcast           802.11     62 Beacon frame, SN=0, FN=0, Flags=........, BI=100, SSID=Wokwi-GUEST
   99 4.523076525  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1229, FN=0, Flags=...P...T
  100 4.600004150  42:13:37:55:aa:01   Broadcast           802.11     62 Beacon frame, SN=0, FN=0, Flags=........, BI=100, SSID=Wokwi-GUEST
  101 4.601698175  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1230, FN=0, Flags=.......T
  102 4.653473550  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1231, FN=0, Flags=...P...T
  103 4.700004150  42:13:37:55:aa:01   Broadcast           802.11     62 Beacon frame, SN=0, FN=0, Flags=........, BI=100, SSID=Wokwi-GUEST
  104 4.747944237  93.184.216.34       10.10.0.2           TCP        76 80 → 55534 [SYN, ACK] Seq=0 Ack=1 Win=29184 Len=0 MSS=1460
  105 4.750968937  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1232, FN=0, Flags=.......T
  106 4.752153962  10.10.0.2           93.184.216.34       TCP        72 55534 → 80 [ACK] Seq=1 Ack=1 Win=5744 Len=0
  107 4.761323287  10.10.0.2           93.184.216.34       HTTP      213 GET / HTTP/1.1
  108 4.800004275  42:13:37:55:aa:01   Broadcast           802.11     62 Beacon frame, SN=0, FN=0, Flags=........, BI=100, SSID=Wokwi-GUEST
  109 4.812787825  Espressi_00:01:10   42:13:37:55:aa:01   802.11     24 Null function (No data), SN=1233, FN=0, Flags=...P...T
  110 4.900004275  42:13:37:55:aa:01   Broadcast           802.11     62 Beacon frame, SN=0, FN=0, Flags=........, BI=100, SSID=Wokwi-GUEST
  111 4.997945487  93.184.216.34       10.10.0.2           TCP        72 80 → 55534 [ACK] Seq=1 Ack=142 Win=29043 Len=0
  112 4.997945487  93.184.216.34       10.10.0.2           TCP      1508 80 → 55534 [ACK] Seq=1 Ack=142 Win=65535 Len=1436 [TCP segment of a
  113 4.997945487  93.184.216.34       10.10.0.2           HTTP      227 HTTP/1.1 200 OK  (text/html)
```

```
> Frame 107: 213 bytes on wire (1704 bits), 213 bytes captured (1704 bits)
> IEEE 802.11 Data, Flags: .......T
> Logical-Link Control
> Internet Protocol Version 4, Src: 10.10.0.2, Dst: 93.184.216.34
> Transmission Control Protocol, Src Port: 55534, Dst Port: 80, Seq: 1, Ack: 1, Len: 141
v Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Host: example.com\r\n
    User-Agent: ESP32HTTPClient\r\n
    Connection: keep-alive\r\n
    Accept-Encoding: identity;q=1,chunked;q=0.1,*;q=0\r\n
    \r\n
    [Full request URI: http://example.com/]
    [HTTP request 1/1]
    [Response in frame: 113]
```

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**.

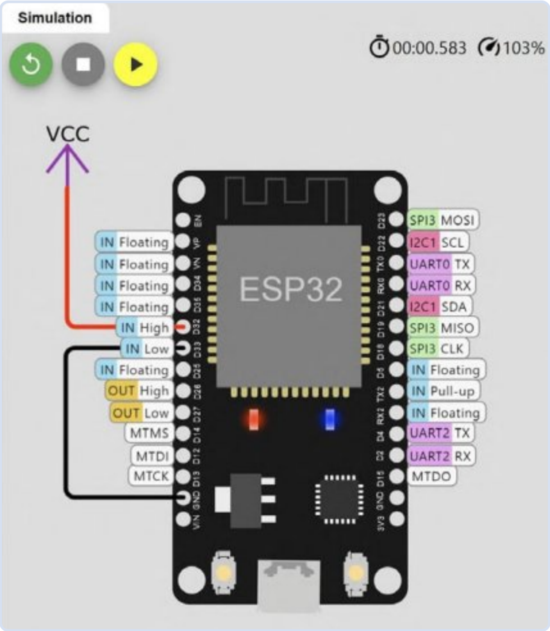


Figure 5: Simulation paused, with state indication for each pin.

Pin Function Debugger

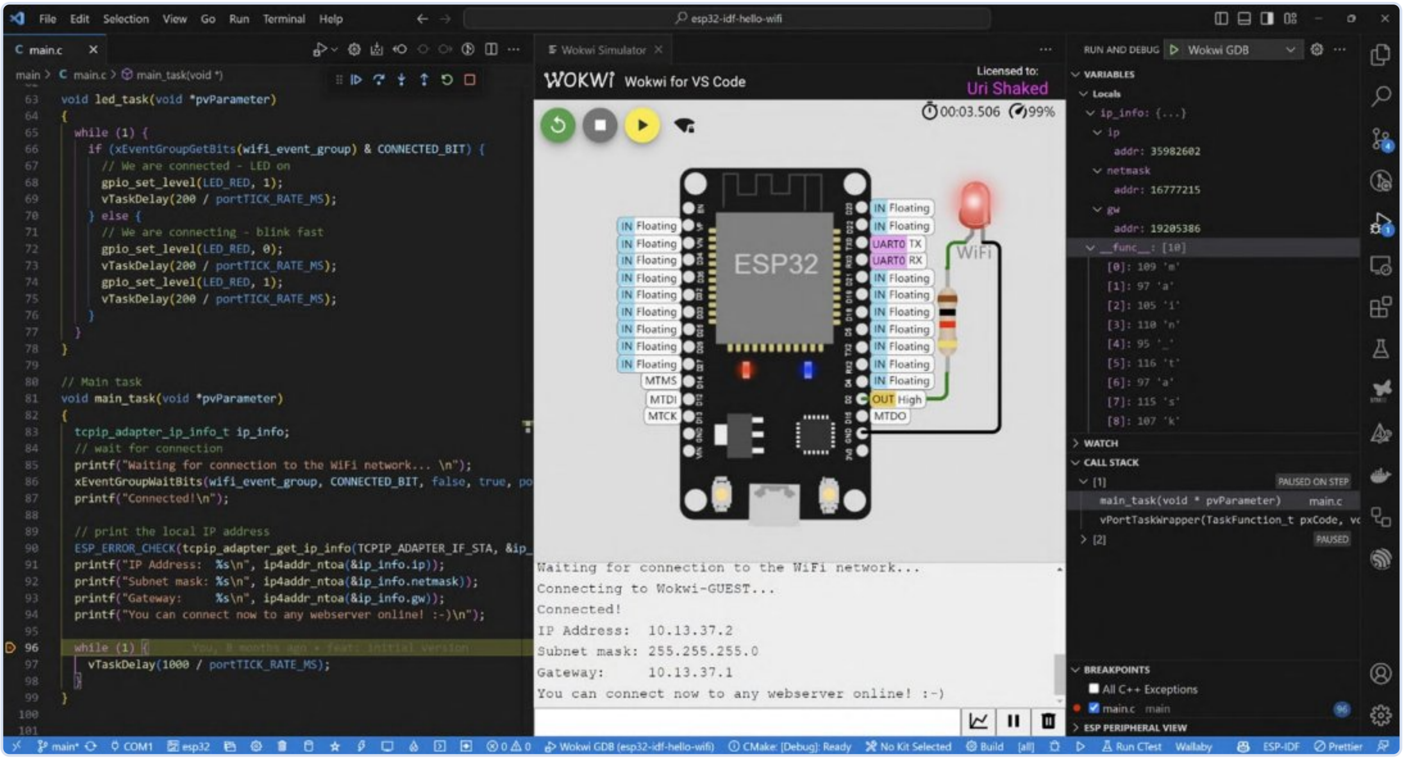
The ESP32 has a flexible peripheral pin mapping: Software can define which peripheral is connected to each of the pins by configuring IO MUX, GPIO Matrix, and the low-power/RTC peripherals. Defining the pin mapping in software can be very convenient, but it also makes it more challenging to figure out 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.



```c
void led_task(void *pvParameter)
{
  while (1) {
    if (xEventGroupGetBits(wifi_event_group) & CONNECTED_BIT) {
      // We are connected - LED on
      gpio_set_level(LED_RED, 1);
      vTaskDelay(200 / portTICK_RATE_MS);
    } else {
      // We are connecting - blink fast
      gpio_set_level(LED_RED, 0);
      vTaskDelay(200 / portTICK_RATE_MS);
      gpio_set_level(LED_RED, 1);
      vTaskDelay(200 / portTICK_RATE_MS);
    }
  }
}

// Main task
void main_task(void *pvParameter)
{
  tcpip_adapter_ip_info_t ip_info;
  // wait for connection
  printf("Waiting for connection to the WiFi network... \n");
  xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT, false, true, po
  printf("Connected!\n");

  // print the local IP address
  ESP_ERROR_CHECK(tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA, &ip_
  printf("IP Address:  %s\n", ip4addr_ntoa(&ip_info.ip));
  printf("Subnet mask: %s\n", ip4addr_ntoa(&ip_info.netmask));
  printf("Gateway:     %s\n", ip4addr_ntoa(&ip_info.gw));
  printf("You can connect now to any webserver online! :-)\n");

  while (1) {      You, 8 months ago • feat: initial version
    vTaskDelay(1000 / portTICK_RATE_MS);
  }
}
```

```
Waiting for connection to the WiFi network...
Connecting to Wokwi-GUEST...
Connected!
IP Address:  10.13.37.2
Subnet mask: 255.255.255.0
Gateway:     10.13.37.1
You can connect now to any webserver online! :-)
```

```
VARIABLES
  Locals
    ip_info: {...}
      ip
        addr: 35982602
      netmask
        addr: 16777215
      gw
        addr: 19205386
    __func__: [10]
      [0]: 109 'm'
      [1]: 97 'a'
      [2]: 105 'i'
      [3]: 110 'n'
      [4]: 95 '_'
      [5]: 116 't'
      [6]: 97 'a'
      [7]: 115 's'
      [8]: 107 'k'
CALL STACK
  [1]                                              PAUSED ON STEP
    main_task(void * pvParameter)     main.c
    vPortTaskWrapper(TaskFunction_t pxCode, vo
  [2]                                              PAUSED
```

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

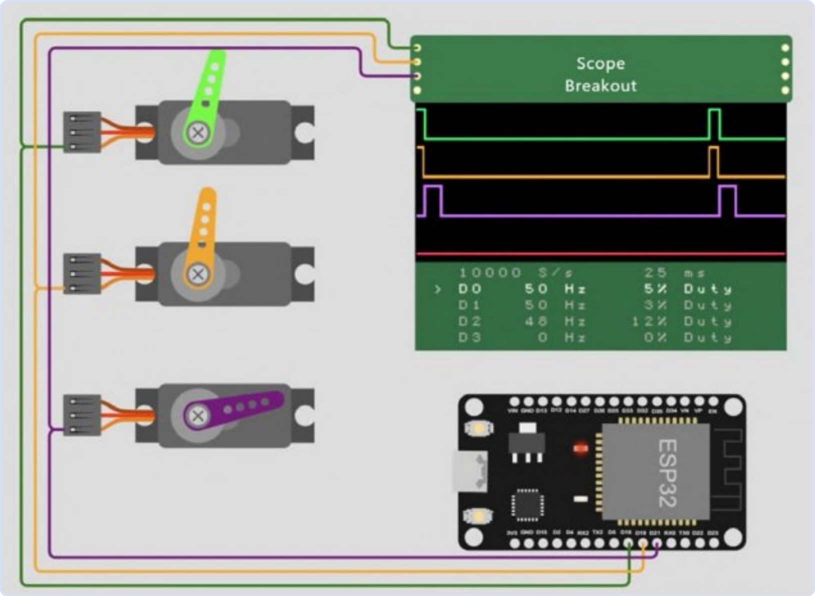
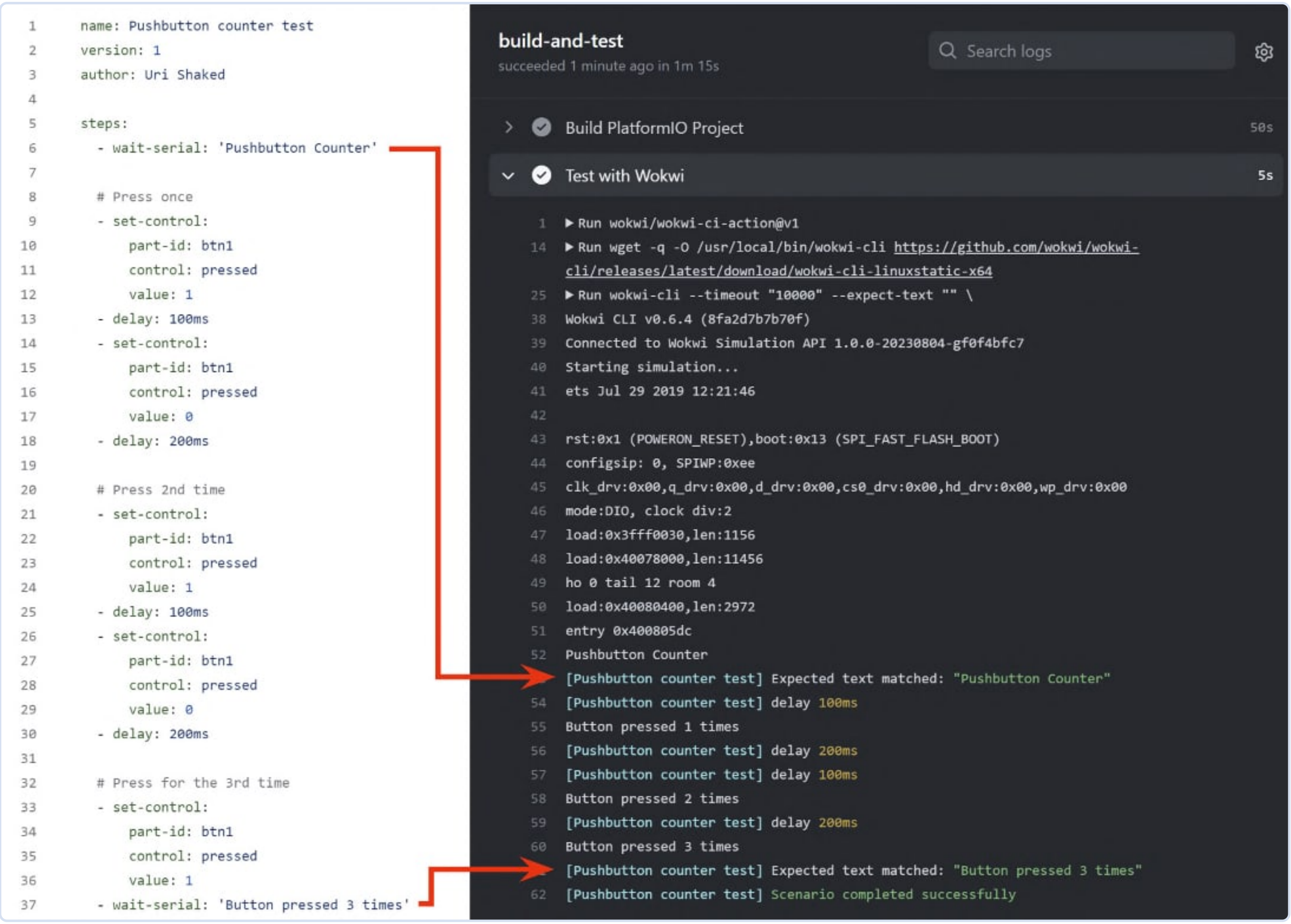


Figure 7: A digital scope one user created using the Custom Chips API.



```yaml
name: Pushbutton counter test
version: 1
author: Uri Shaked

steps:
  - wait-serial: 'Pushbutton Counter'

  # Press once
  - set-control:
      part-id: btn1
      control: pressed
      value: 1
  - delay: 100ms
  - set-control:
      part-id: btn1
      control: pressed
      value: 0
  - delay: 200ms

  # Press 2nd time
  - set-control:
      part-id: btn1
      control: pressed
      value: 1
  - delay: 100ms
  - set-control:
      part-id: btn1
      control: pressed
      value: 0
  - delay: 200ms

  # Press for the 3rd time
  - set-control:
      part-id: btn1
      control: pressed
      value: 1
  - wait-serial: 'Button pressed 3 times'
```

```
▶ Run wokwi/wokwi-ci-action@v1
▶ Run wget -q -O /usr/local/bin/wokwi-cli https://github.com/wokwi/wokwi-
cli/releases/latest/download/wokwi-cli-linuxstatic-x64
▶ Run wokwi-cli --timeout "10000" --expect-text "" \
Wokwi CLI v0.6.4 (8fa2d7b7b70f)
Connected to Wokwi Simulation API 1.0.0-20230804-gf0f4bfc7
Starting simulation...
ets Jul 29 2019 12:21:46

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1156
load:0x40078000,len:11456
ho 0 tail 12 room 4
load:0x40080400,len:2972
entry 0x400805dc
Pushbutton Counter
[Pushbutton counter test] Expected text matched: "Pushbutton Counter"
[Pushbutton counter test] delay 100ms
Button pressed 1 times
[Pushbutton counter test] delay 200ms
[Pushbutton counter test] delay 100ms
Button pressed 2 times
[Pushbutton counter test] delay 200ms
Button pressed 3 times
[Pushbutton counter test] Expected text matched: "Button pressed 3 times"
[Pushbutton counter test] Scenario completed successfully
```

▲

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.

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

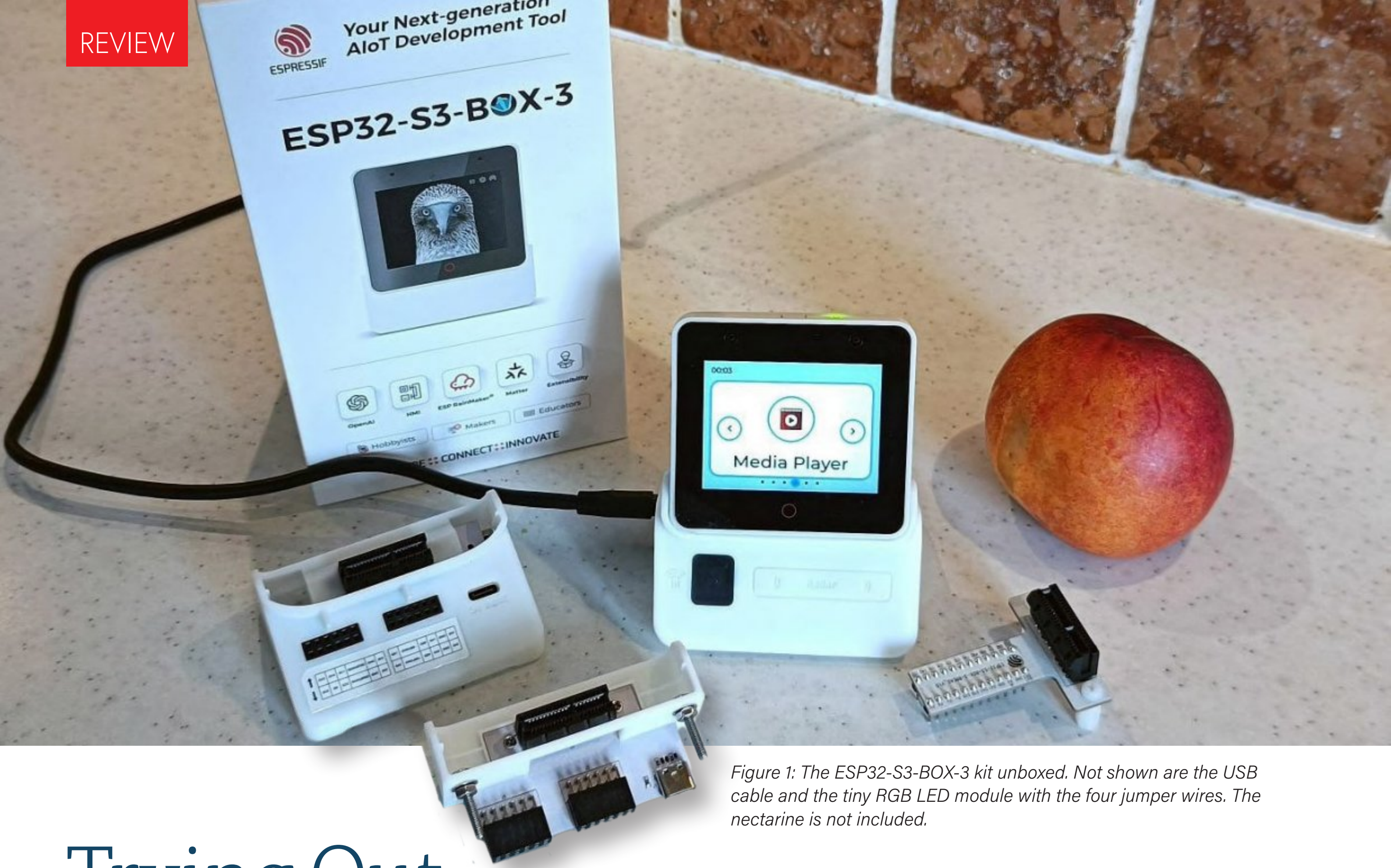


Figure 1: The ESP32-S3-BOX-3 kit unboxed. Not shown are the USB cable and the tiny RGB LED module with the four jumper wires. The nectarine is not included.

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

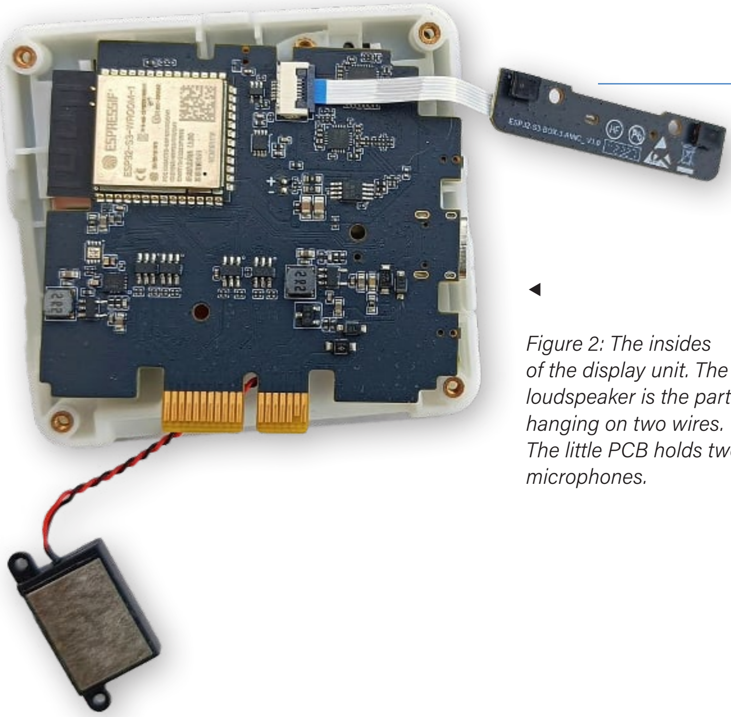
The ESP32-S3-BOX-3 (**Figure 1**) comes as one of those slow-opening boxes (to improve the Wow! effect, as I was told by someone who had worked at Apple). When opened, you see a display module (55 mm × 50 mm) with a kind of stand for it, and a short (approx. 30 cm) USB-C cable. This is only the top layer. 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).



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

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

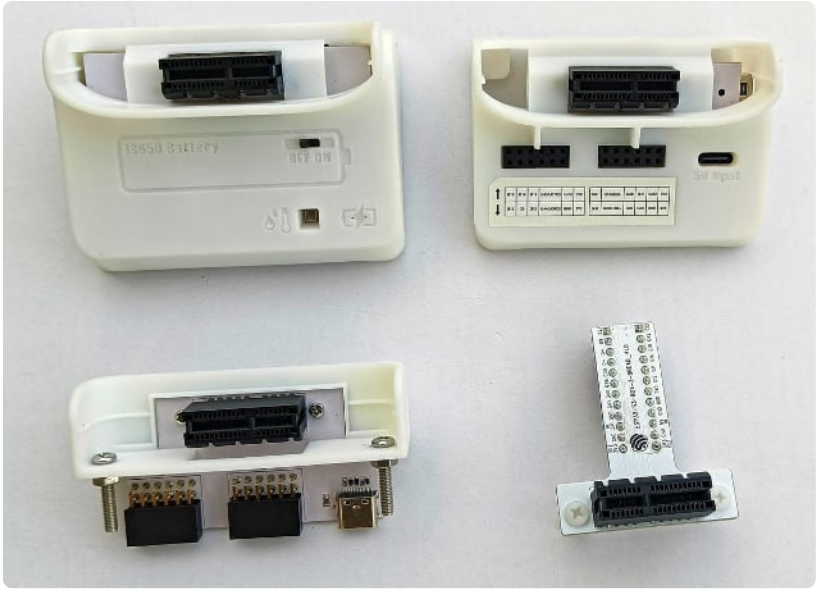
According to the help screens, you can voice activate some things by saying “Hi ee es pea” (ESP pronounced letter by letter). As nothing happened when I tried this, I downloaded the user guide [2]. For this, a QR code is printed on the bottom of the box. According to the manual, voice control works in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

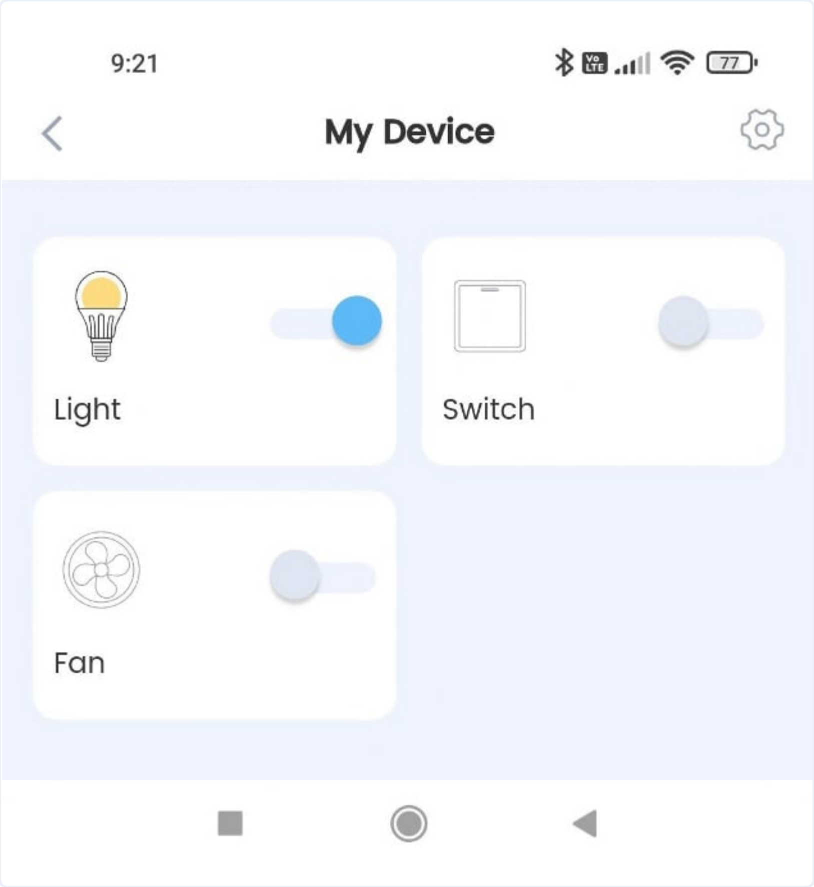
If you succeed, you will hear a ping sound and the display shows a microphone. You now have six seconds to pronounce a command. After six seconds of inactivity, the device says, “Timeout, see you next time” and returns to the mode it was in before it was activated. I found it very hard to get the device 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.

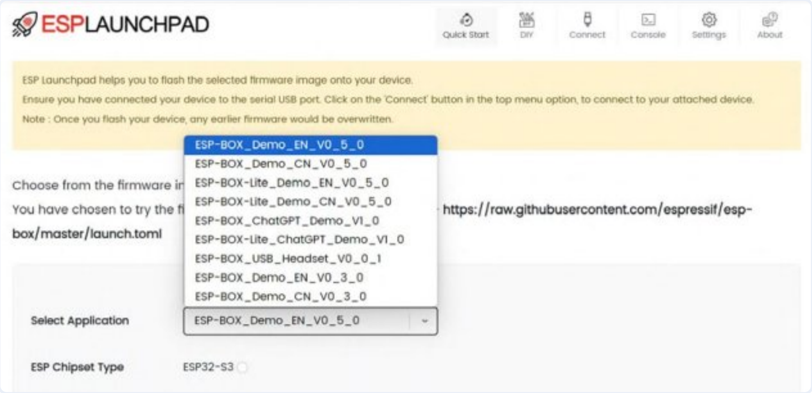
The Box in the Cloud

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.

▼



ESP Launchpad helps you to flash the selected firmware image onto your device.

Ensure you have connected your device to the serial USB port. Click on the 'Connect' button in the top menu option, to connect to your attached device.

Note : Once you flash your device, any earlier firmware would be overwritten.

Choose from the firmware ir

You have chosen to try the fi https://raw.githubusercontent.com/espressif/esp-box/master/launch.toml

Select Application ESP-BOX_Demo_EN_V0_5_0

ESP Chipset Type ESP32-S3

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.

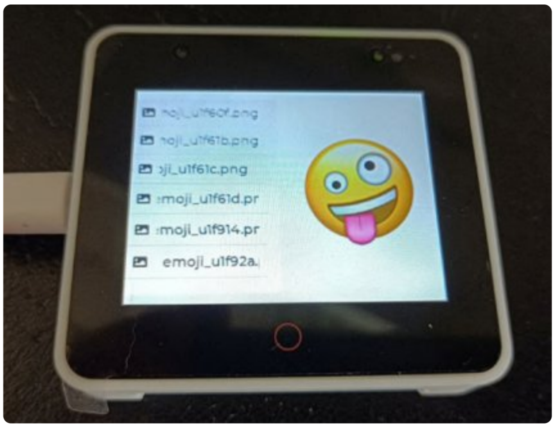


Figure 6: Artificial Intelligence of Things (AIoT), still a long way to go?

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.

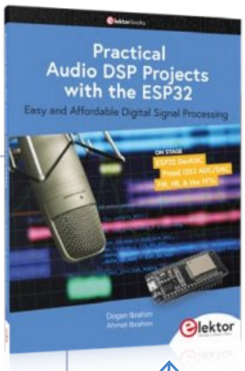
Even though there is quite a lot of information, it still feels a bit incomplete. A tutorial on how to work with the ESP-BOX app or how to create a voice-controlled application from scratch would be appreciated. There isn’t much about AIoT, yet that is what this product is all about, isn’t it?

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

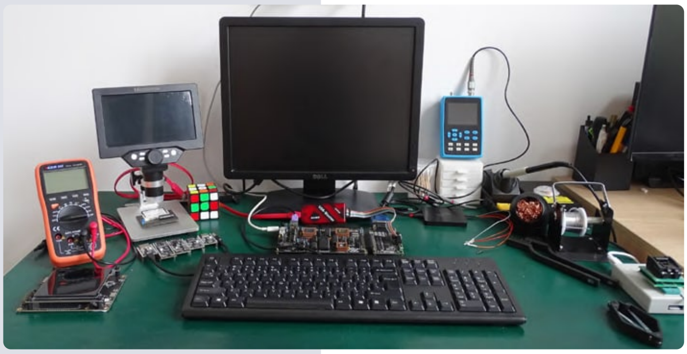
The electronics workspace: It's where the magic happens. Are you curious about what other creative engineers have in their workspaces? Looking for some tips about stocking and organizing your electronics workbench? A few Espressif engineers offer some insights.

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 I^{2}C 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

Jeroen Domburg
Location: Singapore

Organize Your Mess

My desk tends to have what's called a "chronological ordering system," which is a big pile of crap with the most recently used stuff at the top and the oldest at the bottom. It gets reorganized when I run out of desk space, when I forget what's at the bottom, or if there's stuff in the heap I need to throw away. If you're like that, don't apologize for a "messy desk:" Unless you're sharing it, your workspace can look like however you like! Seemingly, you function best by using your memory to be able to put the least effort in putting away and finding your tools again, which probably makes you more efficient than the desk-always-clean people.

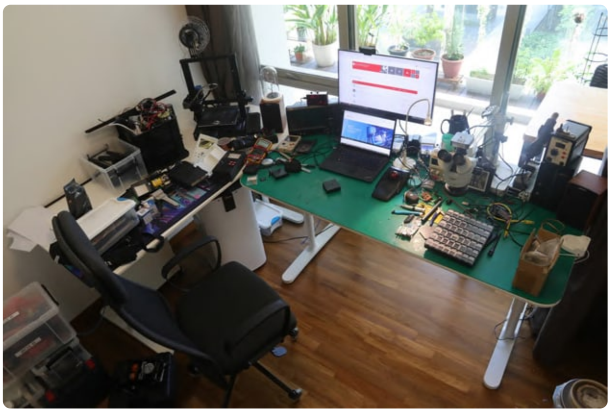
Long-term storage is a different story. I go by the motto of "if I don't know I have it, I might as well not have it," but I don't want to have to actively remember every single thing I have. As such, I store all my components, old projects, etc., in numbered boxes, with a (very-well backed up!) file on my PC that tells me what's where. That makes it easy to find a component or tool when I need it. On those tools: If you have any intention of SMD soldering, get a good microscope; if you can get a binocular one, even better! I find that having one actually stabilizes your hands, making you able to solder smaller components without issue. Obviously, having a decent soldering iron helps there as well, and good ones have been getting cheaper. Even if your dad's Weller WTCP was great at the time, a modern cheap Pinecil or TS-100 will blow it out of the water at a fraction of the price.

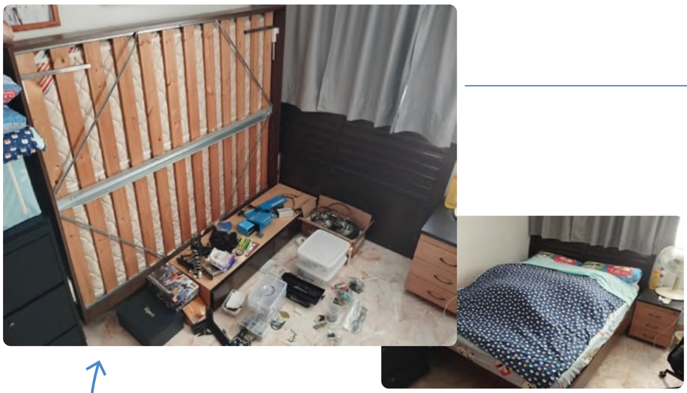
Essentials

- Computer. Can't program chips without one.
- Modern soldering station. My current favorite is an Aixun T3A with a T245 handle.
- Air purifier
- Optical binocular microscope
- Oscilloscope
- Logical analyzer
- Lab power supply

Wishlist

- Better lab PSU
- Space where I can do "dirty" stuff (e.g., resin printing, painting, etc.)
- More free time to play with electronics





Kai Jie Tan
Location: Singapore

Start Anywhere

Professional fully decked-out workspaces with equipment sitting on your desk and a pegboard full of tools within arms' reach are truly the be-all and end-all goal, but you don't need one to start. Years ago, I started with just a toolbox and electronics kit that my engineering school forced everyone to buy. They are always chucked into cabinets in my room due to space constraints. As a young adult in Singapore, living in a bedroom in my parents' apartment, this is a huge consideration. With housing prices high and long waiting times for public housing, most young adults only move out of their parents' homes when getting married. As I quickly run out of space with every project I build, especially if they are of bigger scale (think of cosplay props and electric scooters), I start to have tools in the most random places. One such example is my 3D printer sitting on top of my display cabinet, 2 meters off the ground. At the peak of COVID-19 restrictions, I lost my access to a Fab Lab and had to do everything at home. This led me to build a Murphy bed using hardware from AliExpress. It was game-changing, as it allows me to keep my work in progress safely under my bed — accessible, and little setup time as compared to having to keep everything in toolboxes. There was supposed to be a Phase 2 to this project, where I have shelves on four-bar linkages, so the shelves becomes the bed legs when folded down. That's something I have been sleeping on (pun intended).

Fast forward, and my workspace is nowhere close to the dreamy professional Fab Lab setup. I just have many more toolboxes and compartment boxes of electronics than before; however, I rarely felt that it restricted me from building anything I wanted. Some advice: As you go from project to project, you will accumulate more tools and replace worn tools along the way. So, no need to wait for a fully furnished workspace to start. Just start anywhere.

Essentials

- Fluke 115 multimeter
- Proskit 7-in-1 wire stripper
- 3D printer (very empowering to own one)

Wishlist

- Soldering fume extractor
- Air purifier
- 3D printer enclosure
- A garage to properly store tools

Pedro Minatel
Location: Braga, Portugal

Create a Sanctuary

I describe my electronics workbench as my sanctuary — a place where I can create and invent things, even though most of them will always remain works in progress. I strive to keep most of my tools organized, but, at times, a bit of mess is inevitable. I take pride in my equipment; some pieces are even older than me, like my 50s-era caliper.

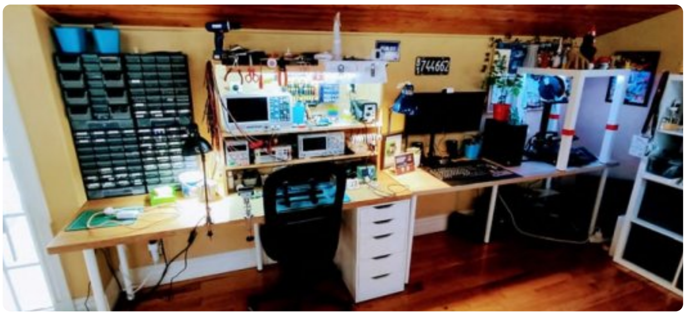
My favorite, and one of the most crucial tools on my electronic bench, is the Rohde & Schwarz RTB2002 oscilloscope. Advice: Maintain organized components! I prefer using inexpensive containers and storage boxes (similar to those for pills). However, for ESD-sensitive components, it's essential to store them in proper ESD bags. I use a labeling machine to ensure all my parts are correctly stored. Current project: I'm working on another development board, this time based on the ESP32-C6.

Essentials

- Oscilloscope (with logic analyzer)
- Reliable multimeter
- Good soldering station for SMD parts
- Nice basic tools
- 3D printer

Wishlist

- Spectrum Analyzer with VNA (such as the SVA1032X)
- Benchtop pick-and-place machine
- Reflow oven
- Digital power supply (60 V 10 A)



Jakob Hasse
Location: China, Shanghai

The Mahjong Table Workspace

My workspace is mostly used for all types of soldering, reflow-soldering PCBs, and for doing any kind of repairs of electronic products. I have a T12-clone soldering station that works really well due to the direct heating element in the tip. Soldering fumes are really not healthy, which is why I also have a fume extraction fan I bought from Taobao, a Chinese shopping platform owned by Alibaba, for around $150 — well-invested money for my health. To do PCB reflow soldering, I have a hot plate that was so cheap that I don't remember the price anymore. But, when it comes to measurement equipment and "dumb" metal tools, I buy quality. Trust me, life is too short for low-quality tools!

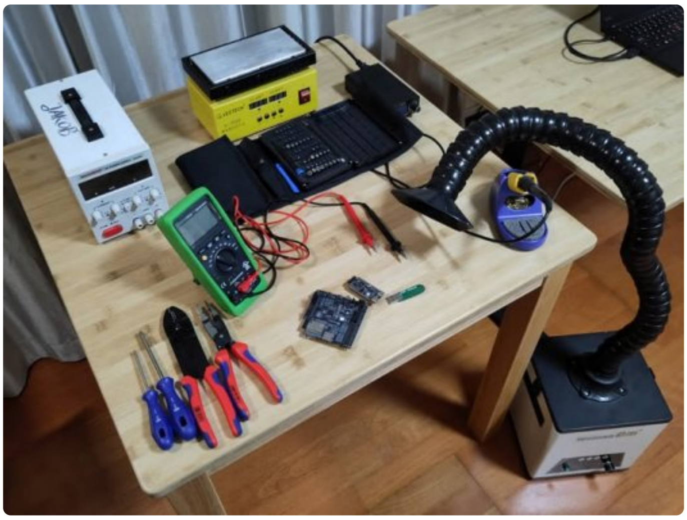
Since I currently live in Shanghai, and space here is quite a luxury, my workspace is only 80×80 cm large. It is a normal table that I ordered on Taobao. The board is made from bamboo, a really hard and sustainable material. As it turns out, however, the coating dissolves when it comes in contact with isopropyl alcohol. Hence, the next thing I will buy is definitely a real silicone mat to protect my desk not only from soldering, but also from isopropyl alcohol. There is one advantage of the small form factor of my workspace: If I have guests, I clear the desk and move it to the living room. The size is perfect for playing Mahjong.

Essentials

- T12-clone Soldering station
- Weinan fume extraction fan
- Vectech otplate
- iFixit repair set
- Maisheng MS-605D 300 W lab power supply
- Isopropyl alcohol
- Gossen-Metrawatt Metraline DM62 multimeter
- LUX screwdrivers for the common screw types
- Knippex multi-function pliers

Wishlist

- Real silicone mat
- Oscilloscope
- Hot-air rework station
- Spare tweezers



Show Off Your Electronics Workspace!

Would you like Elektor's editorial team to consider featuring your electronics workspace on Elektormagazine.com or in the pages of *Elektor Mag*? Use our online Workspace Submission Form to share details about the space where you innovate, design, debug, and program! **elektormagazine.com/workspaces**

230579-01

The ESP RainMaker Story

How We Built "Your" IoT Cloud

By Amey Inamdar, Espressif

Cloud-connected devices are one main part of the IoT. However, with full-solution cloud providers, device makers lose some control on the data, and building a cloud solution from scratch means a significant effort. At Espressif, we saw these problems and decided to offer a solution that achieves the best of both worlds.

"I" in IoT stands for internet. The internet — and hence cloud connectivity — is an integral part of connected devices to reap the benefits of connectivity. Devices connecting to the cloud to send the data and to receive commands, add value not only to end users but also device makers' business. However, having an IoT cloud poses security and privacy considerations for consumer and scalability, security, and increased engineering challenges for device makers.

It was a natural choice for many device makers to go with various full IoT solution providers who provided their own, ready cloud platform along with device-side and mobile applications. However, that soon resulted in device makers not having sufficient differentiation and not having required control of the data that is generated in the cloud. On the other hand, some of the device makers who had the ability built their own cloud from scratch, but it is a significant effort to build everything from scratch and maintain it.

At Espressif, we saw these problems and decided to provide a solution that achieves the best of both the worlds. On one side, we wanted to provide the cloud with complete functionality, and on the other, provide full ownership, control and customizability to customers so that they don't have to start from scratch.

That's where ESP RainMaker was born.

Choosing the Cloud Architecture

The first choice we had to make was what cloud architecture to use (**Figure 1**).

You may have heard it said that the cloud is just someone else's computer. This quip is indeed true. But what matters is how someone else's computer is used to host your application. With the rise of virtualization technologies, powerful servers are utilized to host multiple applications on the same hardware, completely isolated from each other. With virtual machines (think of VMWare), you can run multiple instances of operating systems on the same hardware, whereas with container technologies (Kubernetes, Docker) you can have parallel operating environments on the same hardware. While this is great,

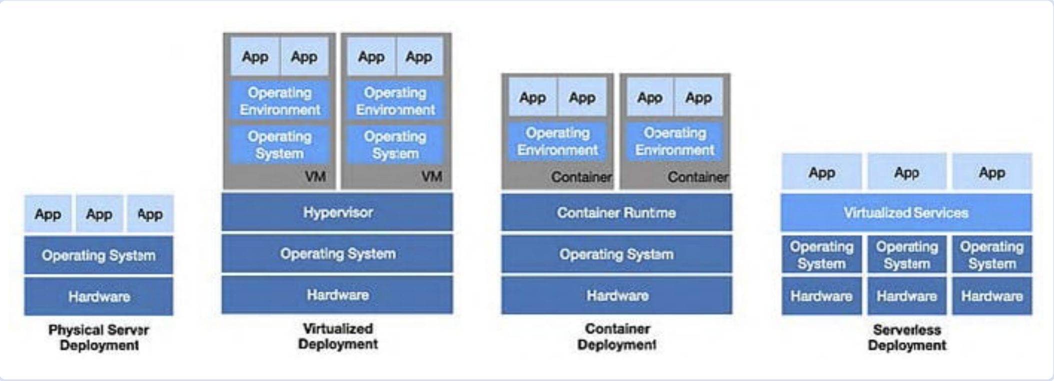


Figure 1: Evolution of cloud architectures.

both of these architectures require you to worry about what software to run and maintain, how to scale with the number of devices increasing/decreasing (and this is the job of separate DevOps engineers). When we wanted our customers to have their own IoT cloud, we couldn't burden them with these efforts. That's where a relatively new paradigm of "serverless" cloud architecture came to the rescue, offering the promise of building an easy-to-maintain IoT cloud.

Serverless architecture doesn't get rid of servers. It's just that the abstractions are provided at a higher level. For example, you get an MQTT broker without needing to worry about which MQTT software it uses, how many device connections it can support, or on which platform it runs. Amazon Web Services (AWS) provided such managed services, which allowed us to implement our IoT cloud. Among many managed services, AWS IoT Core provides a managed MQTT broker, DynamoDB provides a managed noSQL database, S3 provides storage, and, importantly, Lambda provides "function-as-a-service," where you can implement the logic without worrying about where to run it.

With this AWS managed services-based implementation, ESP RainMaker looks like a combination of configuration of various services and their interaction, and implementation of functionality through Lambda functions. Importantly, it is deployable in any AWS account, easily allowing us to create IoT cloud implementation that customers can deploy in their own AWS account and fully control the data and customization.

Dealing With Scalability

The IoT cloud typically has to worry about a large number of devices and users (through mobile apps or other clients such as voice assistants) connecting to and messaging the cloud. Depending on the industry, the number varies, and, for consumer-segment device makers, the number of devices and users can be in the millions. Hence, it is important to ensure that the cloud implementation can scale up to large numbers.

With AWS managing the different services, it is logical to assume that you don't have to worry about the scalability of the cloud backend. While this is largely true, it is also the case that each service may have certain limits. For example, the maximum number of Lambda functions concurrently executing in the AWS account is limited. This requires careful consideration in the architecture to ensure that the system is architected in such a way as to handle a lot of messaging with the appropriate use of a queuing system. It also means having the right error handling in the device firmware and mobile apps.

Today, ESP RainMaker is tested for more than 3.5 million devices and users connecting and communicating via the cloud (**Figure 2**).

Managing Operational Cost

In a traditional cloud architecture, you would typically pay for compute, storage, and memory for your virtual machines. In the serverless architecture, even the unit of billing changes. You have to pay for actual resources used such as number of devices connected and total connection time, number of MQTT messages, number of database read/writes, amount of data stored in S3, and resources consumed by Lambda functions executed. While this is pay-as-you-go pricing, it can possibly cost a lot more if your architecture and implementation is unoptimized.

Operating cost for the customers was one of the key criteria in the design of ESP RainMaker. The target was easy to set. We wanted to develop a cloud backend that would work for even a smart lightbulb — possibly the lowest cost IoT device. It took great effort to carefully choose the AWS services to ensure that ESP RainMaker could meet this goal. Also, the implementation was tuned considerably to have an optimal number of database reads/writes, choice of language of implementation to utilize minimum resources for execution, and so on.

ESP RainMaker currently has multiple customers actually selling low-cost lightbulbs meeting the operational cost requirements.

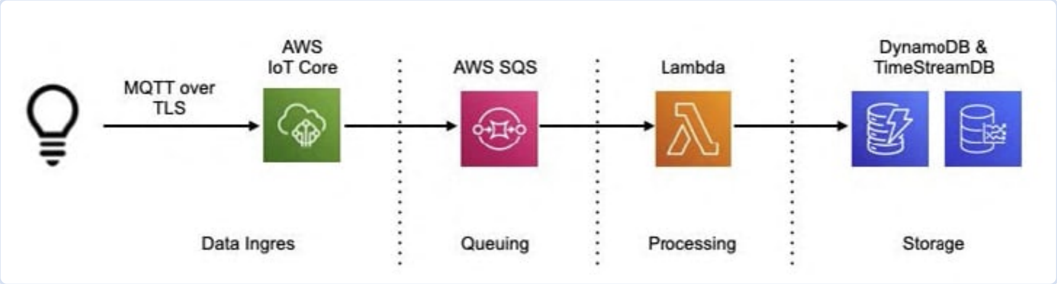


Figure 2: An example of device message processing in ESP Rainmaker.

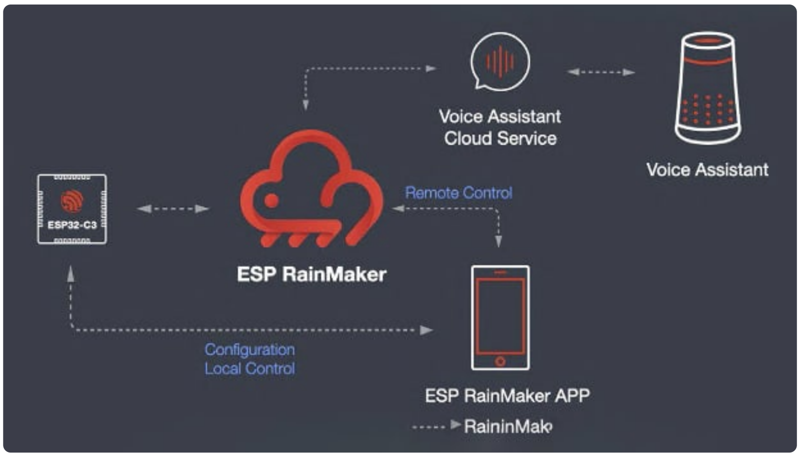


Figure 3: ESP RainMaker components.

Data Security and Privacy

Use of AWS managed services also helped in ensuring data security. They offer standard methods for authentication, authorization, encrypted data storage, application firewalls, etc. For example, AWS IoT Core MQTT broker provides X.509 certificate-based device authentication, providing a means for both a device authenticating the cloud and the cloud authenticating a device. IAM policies allow fine-grained control on authorization. Web Application Firewall (WAF) offers protection against DoS and DDoS attacks on the RESTful service endpoints.

ESP RainMaker uses all these security methods in the prescribed way, ensuring appropriate authentication and authorization using standard methods, and there is no unnecessary privilege escalation. We also worked with a third-party testing and certification company to ensure that ESP RainMaker implements all the required features to meet data privacy requirements to make it easy for customers to meet GDPR compliance when they deploy ESP RainMaker in their own account.

The IoT Cloud Alone Is Not Enough

While the cloud backend is a significant portion of the complete IoT solution, it is not enough. From the connected IoT device perspective, in addition to having the cloud to connect to, it is equally important to have device firmware, mobile phone apps to configure and use the control, and then voice assistant integrations, which have become the most natural way for users to communicate with connected devices in the smart home (**Figure 3**).

At Espressif, we created all these components. The ESP RainMaker device SDK is fully open source and easy to integrate using the many examples that it provides. The iOS and Android reference applications are also fully open source. ESP RainMaker supports Alexa and Google Assistant skills.

Additionally, the cloud is of little use if it can't be used for betterment of the product through engineering and business insights. For that, we built a remote device observability module, ESP-Insights, which provides remote device logs, crash analytics, metrics monitoring and, importantly, facilitates rich queries on the data. This is available through the ESP RainMaker web dashboard, which also allows remote OTA, device grouping, role-based access control, and business insights.

Making It Future-Ready

The Matter protocol is bringing in much-required standardization for smart home devices. Matter is a local area network protocol, and hence cloud connectivity can be considered orthogonal to Matter. Still, the cloud plays a role of providing much needed device management and remote access to the devices. ESP RainMaker offers Matter fabric support that can be used to create your own Matter ecosystem. A combination of ESP RainMaker cloud backend and phone apps implement the Matter fabric with the complete PKI infrastructure required for the Matter fabric. The device, user, and ACL databases are securely synchronized across multiple apps. Also, when you have a smart home controller device connecting to RainMaker, you can provide remote control from your mobile apps not only to your own devices, but to other Matter devices as well.

We also announced support for Mesh-Lite network topology integration with ESP RainMaker. With Mesh-Lite, you can have a dedicated Wi-Fi mesh for your IoT devices to cover a longer range and dead spots in the home. ESP32-series chip-based IoT devices act as access points to allow connection to other devices, forming a mesh (more like a tree) topology. With ESP RainMaker, users can easily create the mesh network and provision the nodes within it.

Getting Started

We've built the ESP RainMaker cloud with the same considerations that any device maker would have to undertake in implementing their own IoT cloud platform. Most importantly, Espressif provides this as a completely rebrandable and customizable platform. For developers, Espressif also provides an Espressif-managed ESP RainMaker instance along with phone apps available from the respective app stores. You can use the web dashboard for device management and perform OTA upgrades. With this, you can not only evaluate the functionality, but also create your own projects for personal use.

To get started, you can use any of the Espressif development boards and follow the steps mentioned in the "Get Started" guide [1].

230621-01

About the Author

Amey Inamdar is the Director of Technical Marketing at Espressif. He has 20 years of experience in the area of embedded systems and connected devices, with roles in engineering, product management, and technical marketing. He has worked with many customers to build successful connected devices based on Wi-Fi and Bluetooth connectivity.

WEB LINK

[1] Get Started with ESP Rainmaker: https://rainmaker.espressif.com/docs/get-started

Assembling the ELEKTOR CLOC 2.0 KIT

An Elektor Product Unboxed by Espressif

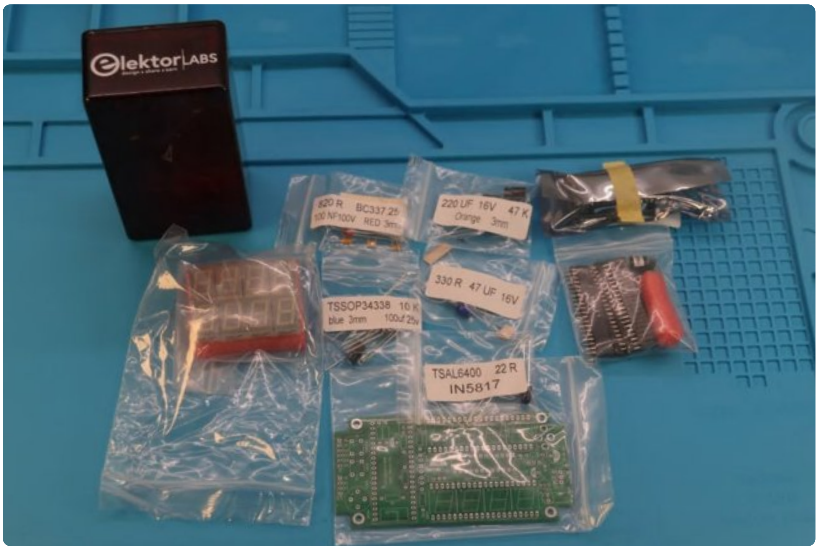
By Jeroen Domburg (Espressif)

The Elektor Cloc 2.0 is a flexible alarm clock based on an ESP32-Pico-Kit from Espressif. It comes as a kit of parts that you must assemble yourself. In this article, the people at Espressif put one together and share their thoughts.

How do you wake up in the morning? Do you have an alarm set on your phone, do you still have an old-school clock radio next to your bed, are you woken by the rooster in the morning or do you simply leave the curtains open, so the early sun rays hit your face?

You may feel that there are some downsides to all these methods. Your phone doesn't really allow you to half-open an eye to see how much sleep time you have left, the clock radio probably only has one alarm which will also wake you up in the morning on a weekend or holiday if you don't turn it off, and although the rooster and sun rays are cheap and generally reliable, it is quite hard to reconfigure the alarm on them. Generally, none of the solutions are "really" flexible.

Figure 1: The kit comes as a set of intelligently-labeled baggies, plus the Hammond case.



If you want that flexibility anyway, the obviously most configurable option is to build your own alarm clock. If you design one yourself, you can put in any functionality you might desire. Per-week alarm times, Wi-Fi connectivity, fancy alarm noises, raising the room temperature, turning on the coffee machine; anything your heart desires.

Enter Cloc 2.0

But what if you don't have the capability, time or energy to go through the entire path of architecture, hardware and software design, manufacturing etc.? Well, Elektor can help here with the Cloc 2.0. This design features a dual 7-segment LED display to view the current time and alarm on, a Wi-Fi connection, so it'll always show the correct time as well as allow for lots of configurable options. Furthermore, it has an IR remote control sender and receiver that allows you to turn on e.g. a sound system when the alarm time comes around. Best of all, it's based on an ESP32-Pico-Kit and the software is open-source, which means that it's easy to add a feature if you're missing one. Elektor was nice enough to send us a kit, so we could review it. The complete project of Cloc 2.0 is published on the Elektor Labs Website here [1].

The Kit

We got the Cloc 2.0 as a kit, and as such it arrived in parts (**Figure 1**). The most striking was the red Hammond case, with the logo of Elektor Labs printed on it. Aside from looking good, the combination of a red case and red LED display is a good one as it improves the contrast of the LEDs, making the clock easier to read. The rest of the components were in multiple baggies, labeled with the values of the components it contained. As an interesting touch, each one contains a set of recognizably different components. For instance, no bag had more than two values of resistor in it, and as such you don't have to stare at resistor bands to figure out which part is which. This is a smart idea that saves plastic but still makes finding the right component trivial; kudos to the Elektor team for thinking up this scheme! Of note is that the kit lacks buttons that are accessible from the outside of the case, the thought being that you may still have some fancy ones yourself somewhere.

Finding the Documentation

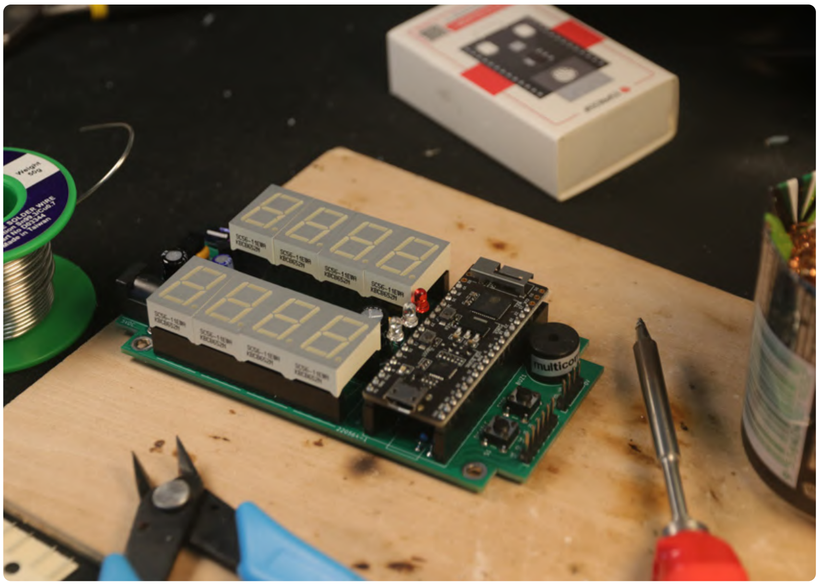
Finding the documentation to put this together could have gone a bit better. The Elektor site has pages spread out over all three domains (elektorlabs.com, elektor.com and elektormagazine.com) with different downloads and text on each page. This is a bit confusing, and it would have been helpful if there was at least a piece of paper detailing where to find all the needed downloads and documentation in the package. However, once we found everything, it turned out to be quite complete: the Elektor article that goes over the design is included as are all the downloads and even a video [2] on the best way how to put everything together.

With documentation in hand, putting together the PCB is a breeze (**Figure 2**). All the components are through-hole, so soldering is easy, and the video suggests an order of component soldering that worked very well for us.

One thing to note is that the headers which hold the 7-segment LED displays are the same type that the ESP32-Pico-Kit fits into. This is a bit of an issue as the pins for the LED displays are thinner and don't always make good contact. This is forgivable as there don't really seem to be headers for smaller pins available with the same height as these; additionally, any issue here is remedied easily by bending the pins of the displays a bit.

Programming the ESP32-Pico-Kit

Once the board assembled, the ESP32-Pico-Kit needed to be programmed. This took a few steps to get the right libraries and board support package installed, but generally wasn't too complicated either. A process like that could have been simplified a bit more by using e.g. ESP-Launchpad [3], but the current setup has the advantage that it builds the sketch from source: this means that if you find something you want to change in the code, you already have everything that is needed to rebuild and flash the firmware.



▲

Figure 2: As the PCB is all through-hole, soldering it was a breeze.

Mechanical Construction

When it comes to the mechanics of putting the PCB in the case, there is a bit more flexibility in how you want to do things: Elektor gives a drill pattern and in recent kits some bits of hardware, so you can build it the "standard" way. However, you certainly may want or even need to modify things in order to get things installed in the way you want. For one, depending on the shape and size of the buttons you want to use, you may want to make your own plan here.

The Power Supply

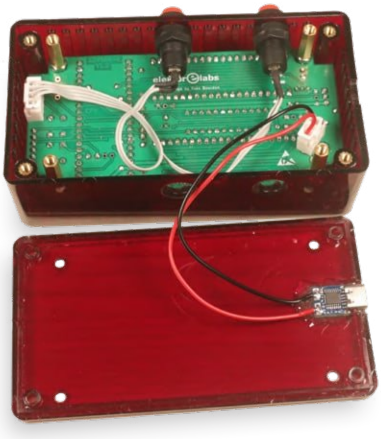
In our case, although we're sure we have dozens of them somewhere, the Big Box of Power Supplies failed to cough up a 5 V brick with the correct barrel connector. What we did find was a little PCB with a USB-C connector that we could modify to simply output 5 V at the required 500 mA or more. This has the added advantage that we don't need to keep a dedicated power supply with our Cloc; any USB-C power supply and cable will do the trick. Obviously, we did modify the case drilling plans accordingly: rather than drill a hole for the barrel jack, we cut away a bit of plastic at the back of the case that exposes the USB-C connector. The USB PCB could then be epoxy'd to the back cover to keep it in place.

Drill with Care

On that note, a word to the wise. When drilling holes and installing things, be sure to keep the order of things and the orientation of the case in mind. Obviously, with us here at Espressif being professionals, we are thoroughly aware of this and as such, we can assure you that the... drain holes... in the bottom of our case were obviously part of our original plans all along (**Figure 3**).

Figure 3: The back of the box has enough space. The buttons and cables fit in here, but you could also stash an expansion PCB there.

▼



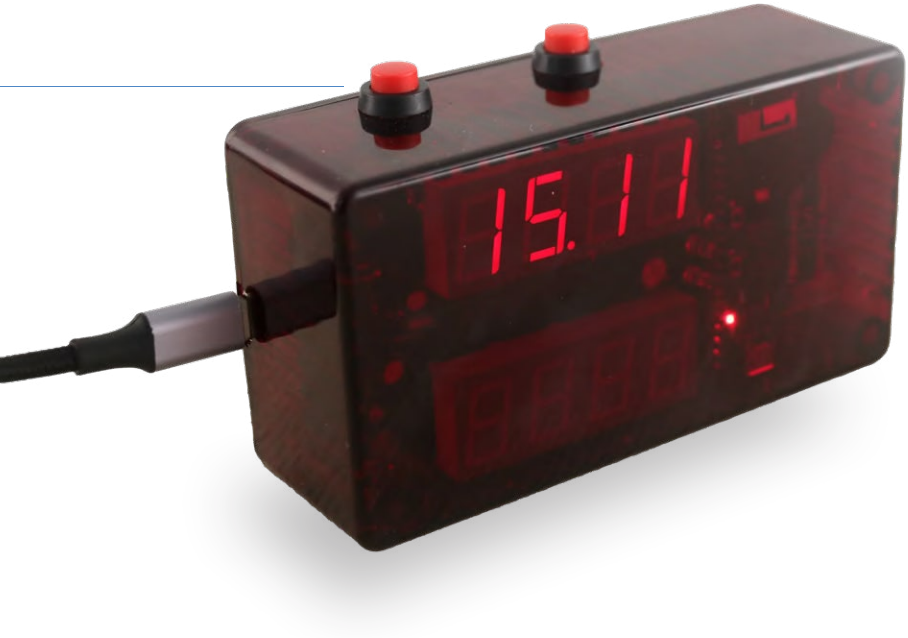


Figure 4: Our finished Cloc, powered by a USB-C adapter.

Finally, we assembled everything neatly. We still had some pretty red buttons that would go well with the case we could put in. Rather than the included headers, we used some JST connectors to connect them to the PCB. We also used a JST connector to hook up the USB power PCB. This way, we can take the main PCB fully out of the case in case it needs rework or repairs.

Configuring the Cloc

With everything put together, configuring the Cloc was trivial. Power it up, connect to it and go to the indicated IP address, and you can connect it to the local Wi-Fi network. From that point on, every time the Cloc starts up, it will show you the IP it has, which is great as you don't need to mess with network scanners or look at the router configuration to access the web interface. The web interface itself is simple enough to barely need a manual, but very complete: we had the basics like time zone and daylight saving set up in only a few moments.

All in all, we really like the Clock (**Figure 4**). The LED display is nice and readable, and the fact that it's red means it won't kill your night vision if you look at it in the dark. It is pretty configurable using the web interface and if you want to modify it to wake you up the way you prefer it, there is space to do so: both in the metaphorical sense given the low barrier of entry that comes with using the Arduino environment as well as the physical sense in that the Hammond box it is housed in has space for e.g. an extra PCB behind the Cloc PCB itself.

Suggestions for Cloc 3.0

There are some improvements we would like to see for a Cloc V3, like using a USB connector as a power supply (hint: if the design would use e.g. an ESP32-S3, the USB GPIO pins it provides could be hooked up to the USB connector as well, allowing programming and debugging the Cloc without opening the case.) Additionally, the ESP32 could perhaps drive an actual speaker for a slightly less harsh wake up sound than the current beeper. We're aware of the IR transmitter which can turn on, e.g. a radio, but we imagine fewer and fewer people will have those in their bedroom nowadays. On the other hand, if those are things you really, really need, there's nothing stopping you from integrating those in the current iteration of the Cloc, given the openness of the design and software.

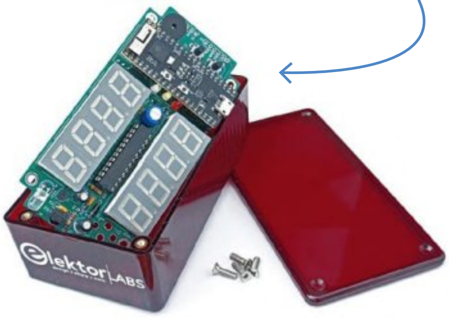
230561-01

Questions or Comments?

If you have any technical questions, you can contact the Elektor editorial team at editor@elektor.com.

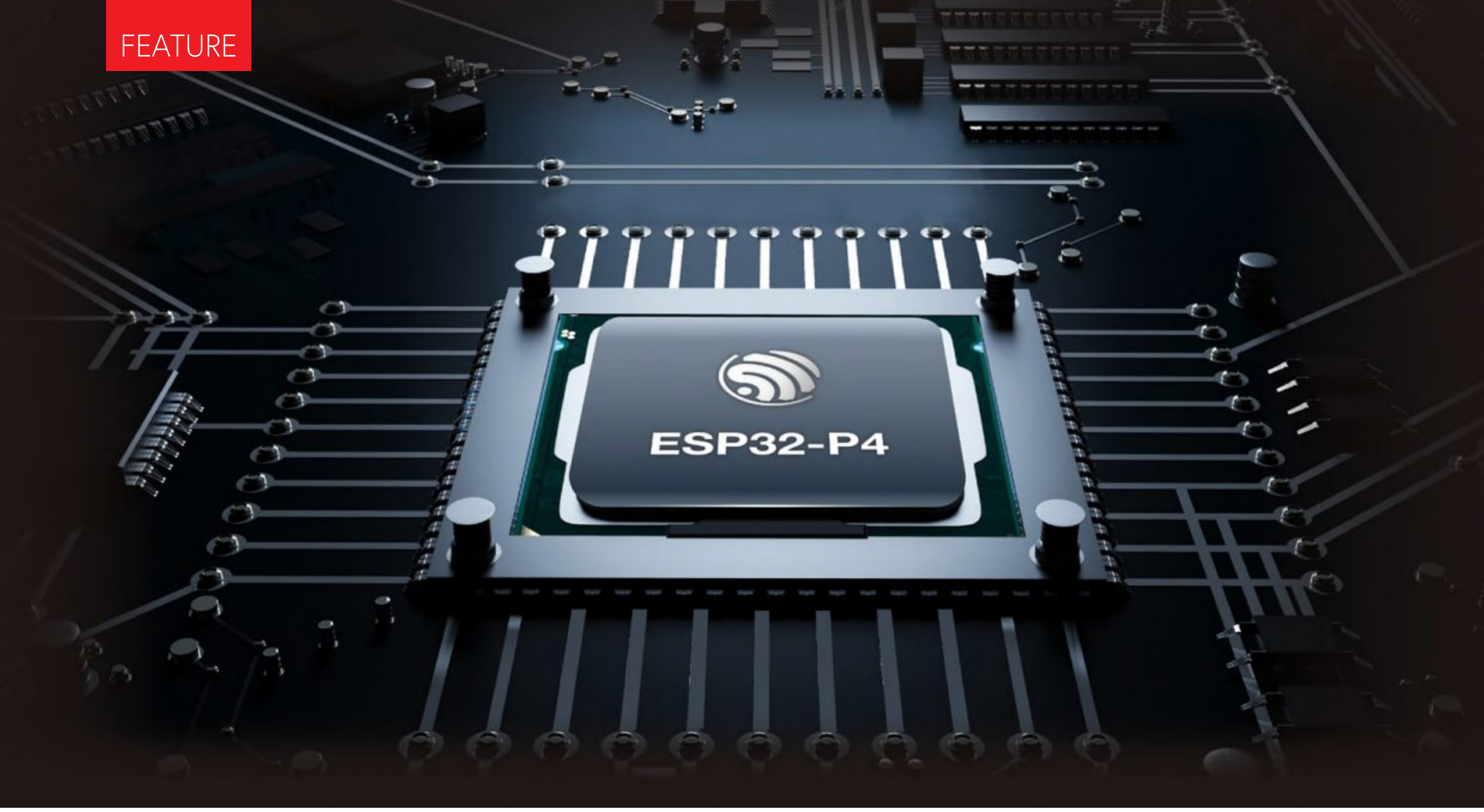
Related Products

- **Elektor Cloc 2.0 Kit**
  www.elektor.com/20438
- **ESP32-PICO-Kit V4**
  www.elektor.com/18423



WEB LINKS

[1] Cloc 2.0 Project on Elektor Labs Website: https://elektormagazine.com/labs/cloc-le-reveil-20
[2] DIY ESP32 Alarm Clock - Elektor Cloc 2.0 Assembly Guide: https://youtu.be/9VSLdFz6jyI
[3] ESP-Launchpad: https://espressif.github.io/esp-launchpad



Unleashing the ESP32-P4

The Next Era of Microcontrollers

By Anant Raj Gupta, Espressif

Welcome to the next era of microcontrollers, where affordable security, cutting-edge performance, and unparalleled connectivity converge. The high-performance microcontroller promises to reshape the world of embedded systems, opening doors to a new realm of possibilities for developers, engineers, and the entire IoT community.

At the heart of this technological leap is the ESP32-P4, a SoC (system-on-chip) meticulously engineered for high performance and fortified with top-tier security features. This powerful microcontroller supports extensive internal memory, boasts impressive image- and voice-processing capabilities, and integrates high-speed peripherals. All of these are included (**Figure 1**) to offer the capacity to meet the demanding requirements of the next era of embedded applications.

Unpacking the ESP32-P4's Computing Process

High-Performance CPU and Memory Subsystem

Fuelling the ESP32-P4's performance is a dual-core RISC-V CPU clocked at up to 400 MHz. This powerhouse is further equipped with single-precision floating-point units (FPUs) and AI extensions, providing an arsenal of computational resources. Complementing this is the integration of an LP-Core, capable of running at up to 40 MHz. This big-little architecture is pivotal for supporting ultra-low-power applications that demand occasional bursts of computing power while conserving energy.

In the pursuit of exceptional performance, the ESP32-P4 stands tall, as **Figure 2** illustrates with a performance comparison against other Espressif products such as the ESP32 and the ESP32-S3.

Memory Architecture for Unmatched Efficiency

Memory access is a critical factor in delivering seamless performance. The ESP32-P4's memory system is a master of efficiency.

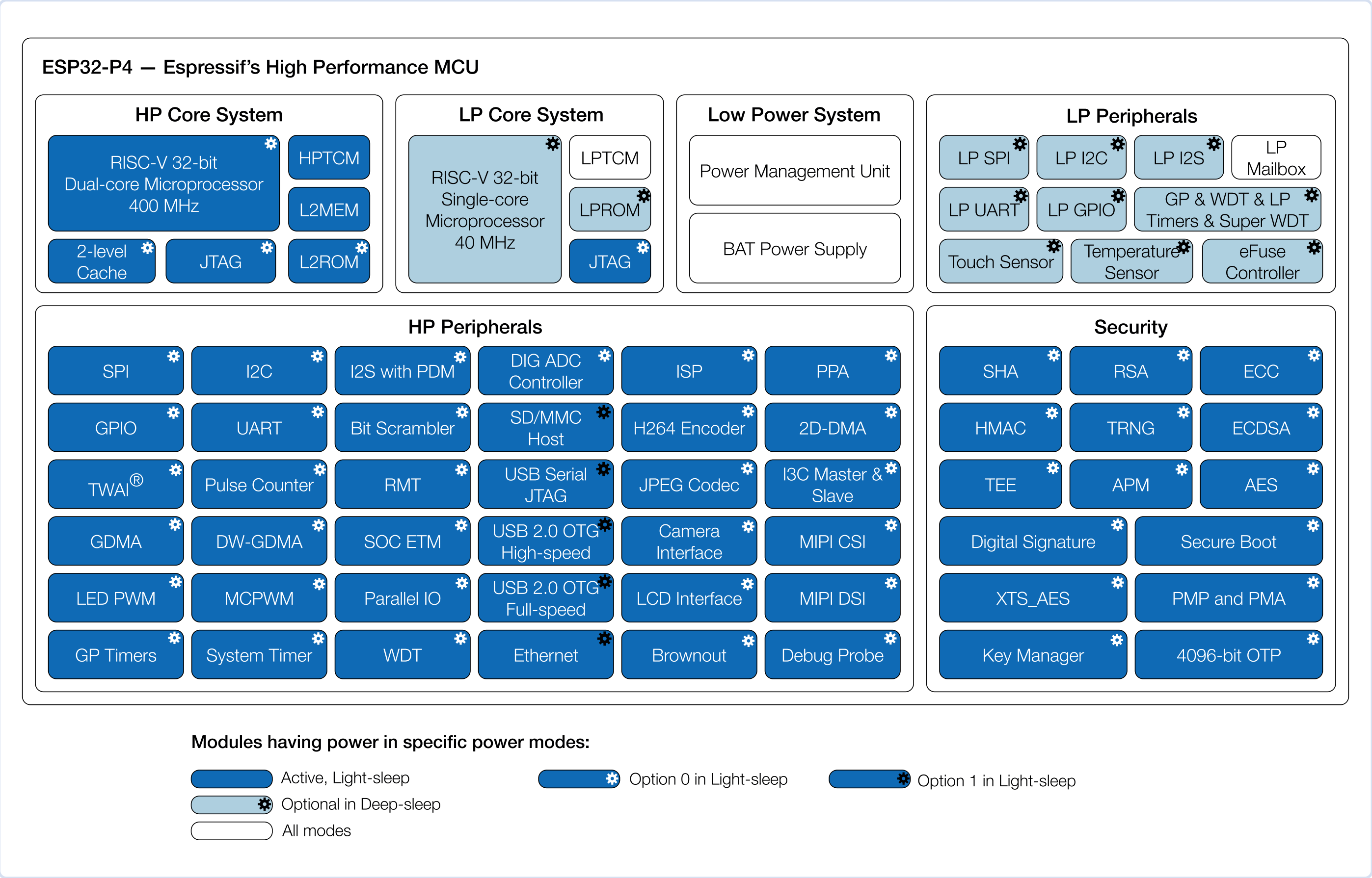


Figure 1: ESP32-P4 block diagram.

With 768 KB of on-chip SRAM, which can be configured as a cache when external PSRAM is available, and 8 KB of zero-wait TCM RAM for swift data buffering, this SoC ensures that memory access latency and capacity are never bottlenecks for your applications.

Moreover, the ESP32-P4's external memory is directly accessible, offering a whopping 64 MB of contiguous space for flash and PSRAM access via the cache. The internal 768 KB memory can be configured as L2MEM, partitioned for instructions and data, and is complemented by a dedicated 16 KB L1 instruction cache and a 64 KB L1 data cache. An additional 8 KB of tightly coupled memory with the HP core ensures single-cycle access to stored data. The PSRAM SPI offers the capability of sixteen-line DDR reads and writes, while also enhancing support for operation at a clock speed of 250 MHz, producing a maximum data throughput of 1 GB/s. This multi-stage memory architecture is engineered to make memory access almost transparent to applications, greatly benefiting high-performance use cases.

Customized Processing for AI and DSP Workloads

The ESP32-P4 boasts a 32-bit RISC-V dual-core processor that supports standard RV32IMAFCZc extensions. However, what truly sets it apart is its custom, extended instruction set. This set is designed to reduce the number of instructions in loop bodies, significantly enhancing performance. Additionally, the SoC incorporates custom AI and DSP extensions, optimizing the operation efficiency of specific AI and DSP algorithms. These custom vector instructions empower the ESP32-P4 for tasks such as neural network and deep learning workloads, enabling accelerated and efficient computation.

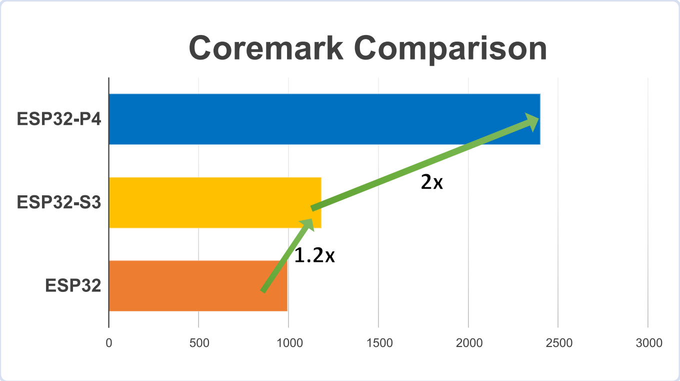


Figure 2: Coremark comparison.

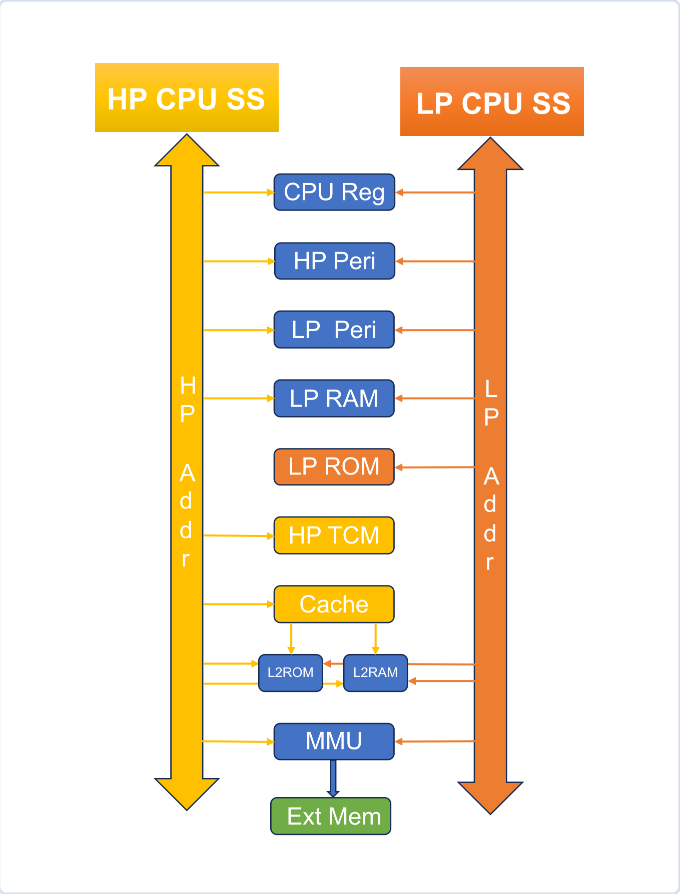


Figure 3: ESP32-P4 address space access.

Efficient Memory Access for Every Core

As depicted in **Figure 3**, the ESP32-P4's memory access architecture ensures that all peripherals and memory are accessible from both the HP and LP cores.

This setup allows the LP core to handle most functions while only waking the HP core when requiring high-performance computation. Furthermore, LP core peripherals enable critical functions even in the lowest power modes. With these features, the ESP32-P4 emerges as the ideal choice for applications requiring high-edge computing capabilities while maintaining low power consumption for extended periods.

Fortifying the Future: Best-in-Class Security

Secure Boot

Security is not an afterthought; it's the cornerstone of ESP32-P4's development. It has been meticulously engineered to deliver affordable security solutions for all. Secure Boot, a pivotal security feature, safeguards the device against running unauthorized, unsigned code. It thoroughly verifies each piece of software, including the second-stage bootloader and every application binary. As shown in **Figure 4**, the first-stage bootloader, being ROM code, remains immutable and doesn't require signing. The ESP32-P4 offers flexibility by supporting both RSA-PSS and ECDSA-based secure boot verification schemes, with ECDSA providing comparable security to RSA, but with shorter key lengths.

Flash Encryption

Flash encryption is a game-changer in protecting the contents of the ESP32-P4's off-chip flash memory. With this feature enabled, firmware is initially flashed as plaintext and subsequently encrypted during the first boot. This means that physical attempts to read the flash will be futile in recovering meaningful data. When flash encryption is activated, all memory-mapped read accesses to flash are transparently and securely decrypted at runtime. The ESP32-P4 employs the XTS-AES block cipher mode with a robust 256-bit key size for flash encryption, ensuring top-notch data protection.

Cryptographic Hardware Acceleration

The ESP32-P4 comes equipped with a comprehensive suite of hardware cryptographic accelerators, ready to tackle a range of standard algorithms used in networking and security applications. These accelerators include support for AES-128/256, SHA, RSA, ECC, and HMAC. This cryptographic muscle enhances the ESP32-P4's ability to secure data transmission, storage, and authentication, making it an exceptional choice for security-sensitive applications.

Private Key Protection

Protecting private keys is paramount, and the ESP32-P4 employs robust mechanisms to ensure their security. The SoC generates

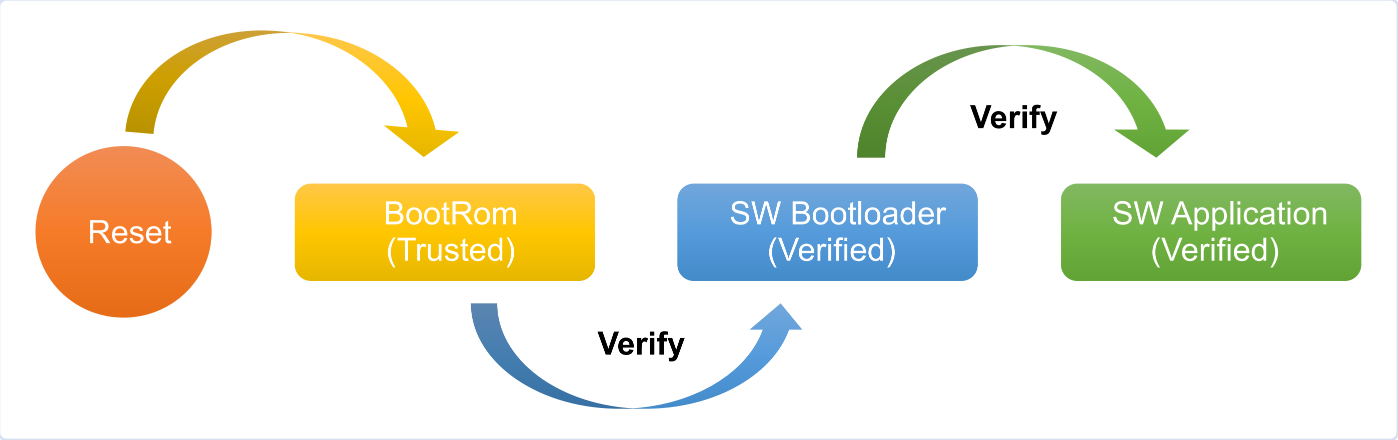


Figure 4: Secure boot flow.

private keys on-chip, inaccessible to software or physical attacks in plaintext. Hardware-accelerated digital signatures are produced, allowing applications to perform signing operations with the encrypted device's private key without exposing it in plaintext. The ESP32-P4 leverages a key manager, utilizing physically unclonable functions (PUF) unique to each chip to generate the hardware unique key (HUK). This HUK serves as the root of trust (RoT) for the chip and is automatically generated at each power-on cycle, disappearing when the chip powers off. The ESP32-P4's hardware-driven private key protection ensures unparalleled security for sensitive operations.

Access Control

The ESP32-P4 takes access control to a new level with hardware access protection, facilitating access permission management and privilege separation (**Figure 5**). This system consists of two components: physical memory protection (PMP) and access permission management (APM). PMP manages CPU access to all address spaces, with APM handling access to ROM and SRAM. PMP must grant permission for CPU access to ROM and HP SRAM, and only then does APM come into play for other address spaces. If PMP permission is denied, APM checks are not triggered. This layered approach ensures robust access control, making the ESP32-P4 a secure choice for a wide range of applications.

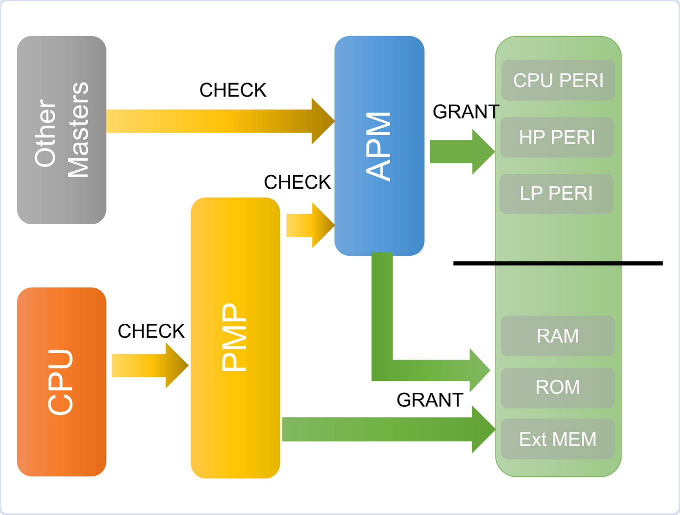


Figure 5: Access control on the ESP32-P4.

Rich Peripherals and Human-Machine Interface (HMI)

Elevated Visual and Touch Experience

The ESP32-P4 elevates the HMI experience with its support for MIPI-CSI (Camera Serial Interface) and MIPI-DSI (Display Serial Interface). Integrated with image signal processing (ISP) on the MIPI-CSI interface, this SoC can handle major input formats, such as RAW8, RAW10, and RAW12, supporting resolutions up to Full HD (1980×1080) at up to 30 frames per second (fps). This capability makes it ideal for applications such as IP cameras (**Figure 6**) and video doorbells that demand high-resolution camera inputs.

On the display side, the ESP32-P4's MIPI-DSI interface supports two lanes at 1.5 Gbps, translating to display support for 720p HD at 60 fps or 1080p full HD at 30 fps. The integration of capacitive touch inputs and speech recognition capabilities positions the ESP32-P4 perfectly for any HMI-based application, from interactive control panels to charge controllers.

Media Processing

The ESP32-P4 doesn't stop at cameras and displays; it's ready for media encoding and compression tasks. With built-in support for H.264 and other encoding formats, the ESP32-P4 enables video streaming and processing. These features can be harnessed to create budget-friendly IP camera solutions, taking advantage of the rich HMI peripherals mentioned earlier. The SoC also has an integrated hardware pixel processing accelerator (PPA), suitable for GUI development.

Expansive GPIO and Peripheral Support

With a remarkable 55 programmable GPIOs, the ESP32-P4 sets a new standard for Espressif products. It boasts support for various commonly used peripherals, such as SPI, I²S, I²C, LED PWM, MCPWM, RMT, ADC, DAC, UART, and TWAITM. The ESP32-P4 supports USB OTG 2.0 HS, Ethernet, and SDIO Host 3.0 for high-speed connectivity.

Seamless Wireless Connectivity

For applications demanding wireless connectivity, the ESP32-P4 pairs effortlessly with ESP32-C/S/H series products as a wireless companion chip. This can be achieved over SPI/SDIO/UART using ESP-Hosted or ESP-AT solutions. When utilizing the ESP-Hosted solution, application development remains consistent with working on an SoC with integrated wireless connectivity. Additionally, the ESP32-P4 can act as the host MCU for other connectivity solutions, including ACK and AWS IoT ExpressLink. Developers can rely on Espressif's mature IoT development framework (ESP-IDF) for support, leveraging their familiarity with a platform that already powers millions of connected devices.

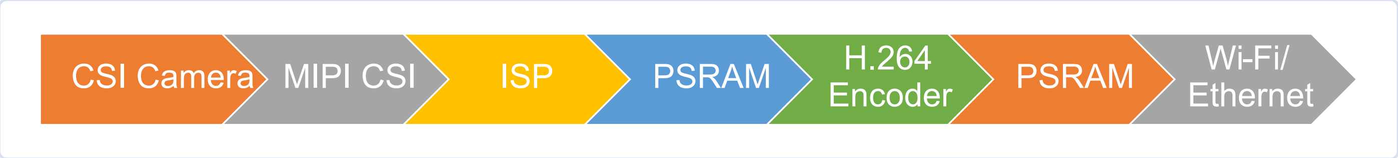


Figure 6: Example data flow for an IP camera.

Embrace the Future With ESP32-P4

The ESP32-P4 represents a new era in microcontrollers, delivering performance, security, and connectivity, which empowers developers to bring their boldest ideas to life. The ESP32-P4 is rooted as a versatile, secure, and high-performance foundation.

The future of IoT is bright, and the ESP32-P4 is poised to be at the forefront of this evolution, enabling faster and more efficient processing. Edge computing will bring intelligence closer to data sources. Artificial intelligence and machine learning will empower IoT devices to make intelligent, autonomous decisions. The ESP32-P4 is ready to embrace these shifts, armed with processing power, robust security, and a dynamic developer community.

The ESP32-P4 represents a monumental leap in the world of microcontrollers. It's not just a chip; it's an invitation to innovation. Whether you're a seasoned developer or a passionate newcomer, the ESP32-P4 empowers you to transform your ideas into reality. Take the leap, explore the potential of the ESP32-P4, and become a part of the transformative journey that this microcontroller represents.

Together, we can shape the future of IoT, creating a more connected, efficient, and secure world.

230615-01

Questions or Comments?

Do you have technical questions or comments about this article? Contact Espressif via the QR code or Elektor at editor@elektor.com.

About the Author

Anant Gupta is a Technical Marketing Manager with 15+ years of experience in system architecture, IP architecture, and SoC design. He is passionate about driving innovation and solving customer problems at Espressif Systems.

Related Product

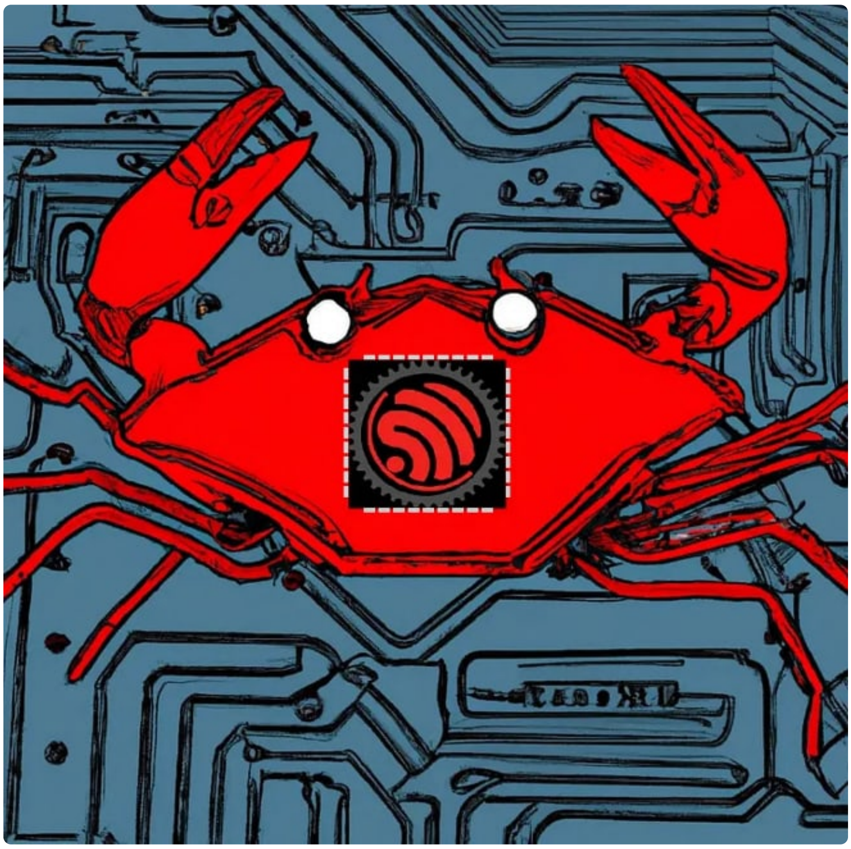
- Espressif ESP32 range
www.elektor.com/espressif

Rust + Embedded

A Development Power Duo

By Juraj Sadel (Espressif)

With its focus on memory and thread safety, Rust is a popular language for producing reliable and secure software. But is Rust a smart solution for embedded applications? As you will learn, Rust offers many advantages over traditional embedded development languages such as C and C++, including memory safety, concurrency support, and performance.



Source: Generated by DALL-E.

Rust has become the hottest new language in the world, with more and more people interested every year. It has a focus on memory and thread safety, and it comes with an intention to produce reliable and secure software. Rust's support for concurrency and parallelism is particularly relevant for embedded development, where efficient use of resources is critical.

The Beginning of Rust

The initial idea of a Rust programming language was born by accident. In 2006, in Vancouver, Mr. Graydon Hoare was returning to his apartment, but the elevator was again out of order due to a software bug. Mr. Hoare lived on the 21st floor, and as he climbed the stairs, he started thinking, "We computer people couldn't even make an elevator that works without crashing!" This accident led Mr. Hoare to work on the design of a new programming language. He hoped it would be possible to write small, fast code without memory bugs. [1] If you are interested in the more detailed and technical history of Rust, please visit [2] and [3].

Almost 18 years later, Rust has become the hottest new language in the world, with more and more people interested every year. In Q1 2020 there were around 600,000 Rust developers and in Q1 2022 the number increased to 2.2 million [4]. [4] Huge tech companies like Mozilla, Dropbox, Cloudflare, Discord, Facebook (Meta), Microsoft, and others are using Rust in their codebase. In the past six years, the Rust language remained the most "loved" programming language [5].

Embedded Development

Embedded development is not as popular as web development or desktop development, and these are a few examples of why this might be the case:

- Hardware constraints: The embedded systems will most likely have limited hardware resources, such as performance and memory. This can make it more challenging to develop software for these systems.
- Limited and niche market: The embedded market is more limited than web and desktop applications, and this can make it less financially rewarding for developers specializing in embedded programming.
- Specialized low-level knowledge: Specialized knowledge of concrete hardware and low-level programming languages is a must-have in embedded development.
- Longer development cycles: Developing software for embedded systems can take longer than developing software for web or desktop applications, due to the need for testing and optimization of the code for the specific hardware requirements.
- Low-level programming languages: These languages, such as assembly or C, do not provide much of an abstraction to the developer and provide direct access to hardware resources and memory, which will lead to memory bugs.

These are only a few examples of why and how embedded development is unique and is not as famous and lucrative for young programmers as web development. If you are used to the most common and modern programming languages like Python, JavaScript, or C# where you do not have to count every processor cycle and every kilobyte used in memory, it is a very brutal change to start with embedded development. It can be very discouraging for coming into the embedded world, not only for beginners, but also for experienced web/desktop/mobile developers. That is why it would be very interesting and necessary to have a modern programming language in embedded development.

Why Rust?

Rust is a modern and relatively young language with a focus on memory and thread safety, with the intent of producing reliable and secure software. Also, Rust's support for concurrency and parallelism is particularly relevant for embedded development, where efficient use of resources is critical. Rust's growing popularity and ecosystem make it an attractive option for developers, especially those who are looking for a modern language that is both efficient and safe. These are the main reasons why Rust is becoming an increasingly popular choice, not only in embedded development, but especially for projects that prioritize safety, security, and reliability.

> *Rust's support for concurrency and parallelism is particularly relevant for embedded development, where efficient use of resources is critical.*

Advantages (Compared with C/C++)

Let's review Rust's notable advantages.

- Memory safety: Rust offers strong memory safety guarantees through its ownership and borrowing system, which is very helpful in preventing common memory-related bugs such as null pointer dereferences or buffer overflows, for example. In other words, Rust guarantees memory safety at compile time through its ownership and borrowing system. This is especially important in embedded development, where memory/resource limitations can make such bugs more challenging to detect and resolve.
- Concurrency: Rust provides excellent support for zero-cost abstractions and safe concurrency and multi-threading, with a built-in `async/await` syntax and a powerful type system that prevents common concurrency bugs such as data races. This can make it easier to write safe and efficient concurrent code, not only in embedded systems.
- Performance: Rust is designed for high performance and can go toe-to-toe with C and C++ in performance measures while still providing strong memory safety guarantees and concurrency support.
- Readability: Rust's syntax is designed to be more readable and less error-prone than C and C++, with features like pattern matching, type inference, and functional programming constructs. This can make it easier to write and maintain code, especially for larger and more complex projects.
- Growing ecosystem: Rust has a growing ecosystem of libraries (crates), tools, and resources for (not only) embedded development, which can make it easier to get started with Rust and find necessary support and resources for a particular project.
- Package manager and build system: Rust distribution includes an official tool called Cargo, which is used to automate the build, test, and publish process together with creating a new project and managing its dependencies.

Disadvantages (Compared With C/C++)

On the other hand, Rust is not a perfect language and also has some disadvantages over other programming languages (not just over C and C++).

- Learning curve: Rust has a steeper learning curve than many programming languages, including C. Its unique features, such as already mentioned ownership and borrowing, may take some time to understand and get used to and therefore are more challenging to get started with Rust.
- Compilation time: Rust's advanced type system and borrow checker can result in longer compilation times compared to other languages, especially for large projects.
- Tooling: While Rust's ecosystem is growing rapidly, it may not yet have the same level of tooling support as more established programming languages. For example, C and C++ have been around for decades and have a vast codebase. This can make it more

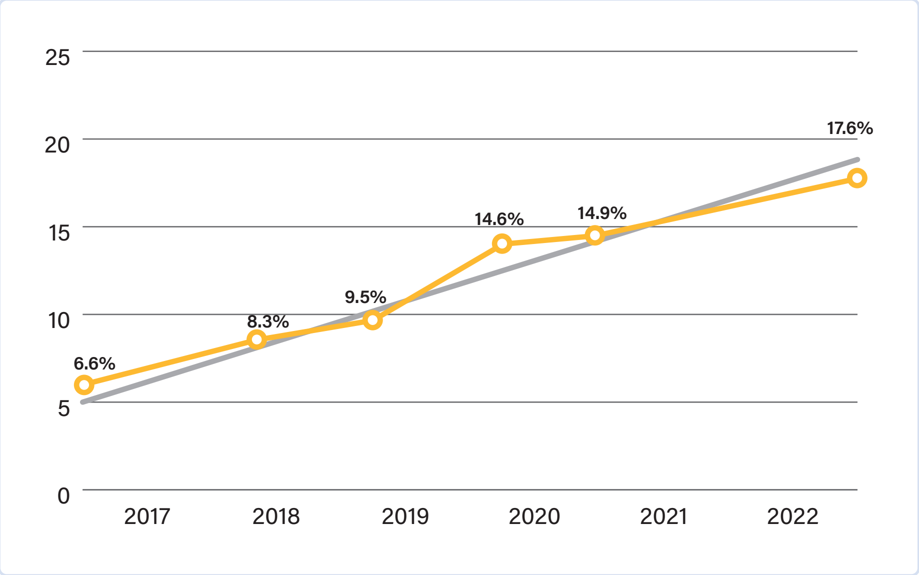


Figure 1: The growing percentage of developers that want to develop in Rust. (Source: Yalantis [4])

challenging to find and use the right tools for a particular project.

- Lack of low-level control: Rust's safety features can sometimes limit low-level control to C and C++. This can make it more challenging to perform certain low-level optimizations or interact with hardware directly, but it is possible.
- Community size: Rust is still a relatively new programming language compared to more established languages like C and C++, which means that it may have a smaller community of developers and contributors, and fewer resources, libraries, and tools.

Overall, Rust offers many advantages over traditional embedded development languages such as C and C++, including memory safety, concurrency support, performance, code readability, and a growing ecosystem. As a result, Rust is becoming an increasingly popular choice for embedded development, especially for projects that prioritize safety, security, and reliability. The disadvantages of Rust compared to C and C++ tend to be related to Rust's relative newness as a language and its unique features. However, many developers find that Rust's advantages make it a compelling choice for certain projects.

How Can Rust Run?

There are several ways to run the Rust-based firmware, depending on the environment and requirements of the application. The Rust-based firmware can typically be used in one of two modes: *hosted-environment* or *bare-metal*. Let's look at what these are.

What Is Hosted-Environment?

In Rust, the *hosted-environment* is close to a normal PC environment [6], which means that you are provided with an operating system. With the operating system, it is possible to build the Rust standard library (*std*) [7]. *std* refers to the standard library, which can be seen as a collection of modules and types that are included with every Rust installation. *It* provides a set of multiple functionalities for building Rust programs, including data structures, networking, mutexes and other synchronization primitives, input/output, and more.

With the *hosted-environment* approach, you can use the functionality from the C-based development framework called the ESP-IDF [8], because it provides a *newlib* [9] environment that is "powerful" enough to build the Rust standard library on top of. In other words, with the *hosted-environment* (sometimes called just *std*) approach, we use the ESP-IDF as an operating system and build the Rust application on top of it.. In this way, we can use all the standard library features listed above and also already implement C functionality from the ESP-IDF API.

In **Listing 1**, you can see how a blinky example [10] running on top of ESP-IDF (FreeRTOS) may look. More examples can be found in *esp-idf-hal* [11].

When You Might Want to Use Hosted Environment

- Rich functionality: If your embedded system requires lots of functionality like support for networking protocols, file I/O, or complex data structures, you will likely want to use *hosted-environment* approach because *std* libraries provide a wide range of functionality that can be used to build complex applications relatively quickly and efficiently.
- Portability: The *std* crate provides a standardized set of APIs that can be used across different platforms and architectures, making it easier to write code that is portable and reusable.
- Rapid development: The *std* crate provides a rich set of functionality that can be used to build applications quickly and efficiently, without worrying about low-level details.

What Is Bare-Metal?

Bare-metal means we do not have any operating system to work with. When a Rust program is compiled with the `no_std` attribute, it means that the program will not have access to certain features. This does not necessarily mean that you cannot use networking or complex data structures with `no_std`. You can do anything without `std` that you can do with `std` but it is more complex and challenging. `no_std` programs rely on a set of core language features that are available in all Rust environments, for example, data types, control structures or low-level memory management. This approach is useful for embedded programming where memory usage is often constrained and low-level control over hardware is required.

In **Listing 2**, you can see how a blinky example [12] running on bare-metal (no operating system) might look. More examples can be found in *esp-hal* [13].

When You Might Want to Use Bare-Metal

- Small memory footprint: If your embedded system has limited resources and needs to have a small memory footprint, you will likely want to use *bare-metal* because *std* features add a significant amount of final binary size and compilation time.

- Direct hardware control: If your embedded system requires more direct control over the hardware, such as low-level device drivers or access to specialized hardware features, you will likely want to use *bare-metal* because *std* adds abstractions that can make it harder to interact directly with the hardware.
- Real-time constraints or time-critical applications: If your embedded system requires real-time performance or low-latency response times because *std* can introduce unpredictable delays and overhead that can affect real-time performance.
- Custom requirements: *bare-metal* allows more customization and fine-grained control over the behavior of an application, which can be useful in specialized or non-standard environments.

Should You Switch From C to Rust?

If you are starting a new project or a task where memory safety or concurrency is required, it may be worth considering moving from C to Rust. However, if your project is already well-established and functional in C, the benefits of switching to Rust may not outweigh the costs of rewriting and retesting your whole codebase. In this case, you can consider keeping the current C codebase and start writing and adding new features, modules, and functionality in Rust — it is relatively simple to call C functions from Rust code. It is also possible to write ESP-IDF components in Rust [14]. In the end, the final decision to move from C to Rust should be based on a careful evaluation of your specific needs and the trade-offs involved.

230569-01

Article continues on next page.

About the Author

Juraj Sadel is an embedded software developer passionate about Rust and dedicated to enhancing embedded systems. He is also an active member of the Rust team, contributing expertise to the community, and enthusiastic about Espressif's cutting-edge technology.

Questions or Comments?

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

Listing 1: Blinky example, with hosted environment and std.

```
// Import peripherals we will use in the example
use esp_idf_hal::delay::FreeRtos;
use esp_idf_hal::gpio::*;
use esp_idf_hal::peripherals::Peripherals;

// Start of our main function i.e entry point of our example
fn main() -> anyhow::Result<()> {
    // Apply some required ESP-IDF patches
    esp_idf_sys::link_patches();

    // Initialize all required peripherals
    let peripherals = Peripherals::take().unwrap();

    // Create led object as GPIO4 output pin
    let mut led = PinDriver::output(peripherals.pins.gpio4)?;

    // Infinite loop where we are constantly turning ON and OFF the LED every 500ms
    loop {
        led.set_high()?;
        // we are sleeping here to make sure the watchdog isn't triggered
        FreeRtos::delay_ms(1000);

        led.set_low()?;
        FreeRtos::delay_ms(1000);
    }
}
```

Listing 2: Blinky example, bare metal.

```
#![no_std]
 #![no_main]

 // Import peripherals we will use in the example
 use esp32c3_hal::{
     clock::ClockControl,
     gpio::IO,
     peripherals::Peripherals,
     prelude::*,
     timer::TimerGroup,
     Delay,
     Rtc,
 };
 use esp_backtrace as _;

 // Set a starting point for program execution
 // Because this is `no_std` program, we do not have a main function
 #[entry]
 fn main() -> ! {
     // Initialize all required peripherals
     let peripherals = Peripherals::take();
     let mut system = peripherals.SYSTEM.split();
     let clocks = ClockControl::boot_defaults(system.clock_control).freeze();

     // Disable the watchdog timers. For the ESP32-C3, this includes the Super WDT,
     // the RTC WDT, and the TIMG WDTs.
     let mut rtc = Rtc::new(peripherals.RTC_CNTL);
     let timer_group0 = TimerGroup::new(
         peripherals.TIMG0,
         &clocks,
         &mut system.peripheral_clock_control,
     );
     let mut wdt0 = timer_group0.wdt;
     let timer_group1 = TimerGroup::new(
         peripherals.TIMG1,
         &clocks,
         &mut system.peripheral_clock_control,
     );
     let mut wdt1 = timer_group1.wdt;

     rtc.swd.disable();
     rtc.rwdt.disable();
     wdt0.disable();
     wdt1.disable();

     // Set GPIO4 as an output, and set its state high initially.
     let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
     // Create led object as GPIO4 output pin
     let mut led = io.pins.gpio5.into_push_pull_output();

     // Turn on LED
     led.set_high().unwrap();

     // Initialize the Delay peripheral, and use it to toggle the LED state in a
     // loop.
     let mut delay = Delay::new(&clocks);
```

```
    // Infinite loop where we are constantly turning ON and OFF the LED every 500ms
    loop {
        led.toggle().unwrap();
        delay.delay_ms(500u32);
    }
}
```

Related Products

- **J. Long, *Embedded in Embedded* (Elektor 2018)**
  Book: www.elektor.com/18876
  E-book: www.elektor.com/18877

- **A. He and L. He, *Embedded Operating System* (Elektor 2020)**
  Book: www.elektor.com/19228
  E-book: www.elektor.com/19214

WEB LINKS

[1] MIT Technology Review, "How Rust went from a side project to the world's most-loved programming language," 2023: https://technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language

[2] Rust Blog, "Announcing Rust 1.0," 2015: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html

[3] Rust Blog, "4 years of Rust," 2019: https://blog.rust-lang.org/2019/05/15/4-Years-Of-Rust.html

[4] Yalantis, "The state of the Rust market in 2023" : https://yalantis.com/blog/rust-market-overview

[5] Stack Overflow, "Stack Overflow Developer Survey," 2021: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted

[6] The Embedded Rust Book: https://docs.rust-embedded.org/book/intro/no-std.html#hosted-environments

[7] The Rust Standard Library (std): https://doc.rust-lang.org/std

[8] ESP-IDF: https://github.com/espressif/esp-idf

[9] Newlib library: https://sourceware.org/newlib

[10] Blinky example running on top of ESP-IDF: https://github.com/esp-rs/esp-idf-hal/blob/master/examples/blinky.rs

[11] ESP-IDF-HAL: https://github.com/esp-rs/esp-idf-hal/tree/master/examples

[12] Blinky example running on bare-metal: https://github.com/esp-rs/esp-hal/blob/main/esp32c3-hal/examples/blinky.rs

[13] ESP-HAL: https://github.com/esp-rs/esp-hal/tree/main

[14] ESP-IDF components in Rust: https://github.com/espressif/rust-esp32-example

Q&A With Scott Mabin

Who Are the Rust-Dacious Embedded Developers?

How Espressif is Cultivating Embedded Rust for the ESP32

Intro and Questions by Stuart Cording (Elektor)

It's just over 10 years old, but the Rust programming language has already reached position 17 in the TIOBE Programming Community index [1]. It's moved 194 places in that time to rank alongside R, Ruby, Delphi, Scratch, and MATLAB. Not only that, but Linus Torvalds, the father of Linux, accepted proposals to allow code written in Rust to be used in the kernel – something C++ has tried and failed to achieve for years. It also has a growing fan base among embedded systems developers, and Espressif has decided to leap on this.

C has been the default language for embedded systems for decades, leaving assembler for those optimizing by hand or developing real-time kernels. Developed in the 1970s by Dennis Ritchie while at Bell Labs, it is closely linked to the creation of the Unix operating system. Unix was written in assembler for the PDP-7 [2] but had to be rewritten to port it to the PDP-11 [3] (PDPs were smaller, general-purpose computers sold as alternatives to mainframes in the 1960s). C made it possible to develop code, like Unix, that was processor-independent and portable across many different architectures.

Embedded systems were initially programmed in assembler, but as code got more complex and developers looked to improve reuse, they, too, were drawn toward C. Additional features that made their lives easier included support for low-level bit manipulation, effortless variable definition, the ease with which algorithms could be coded, and simplicity of memory manipulation.

However, this last point causes most applications to get unstuck. While pointers enable programmers to access (almost) any memory location at any time, something very powerful and efficient in an embedded system, they can also be quite dangerous. Pointers can point to memory long after it has been deallocated, or manipulated to call a function that results in a security breach. In fact, Microsoft attributes around 70% of C/C++ software issues to memory corruption bugs [4].

Rust addresses this issue by enforcing memory safety. Furthermore, unlike C/C++ code, many programming issues are detected at compile time rather than at runtime. And, thanks to similarities in syntax and performance, C and C++ developers will quickly feel at home, both on their PCs and embedded systems.

So, how seriously are microcontroller manufacturers taking Rust as a language for embedded systems? To find out more, Elektor spoke to Scott Mabin, a young developer who, free from the shackles of precedent, decided to dig ever deeper into the world of Rust. As a keen user of ESP32s, he's made huge contributions to embedded Rust, which got him hired at Espressif.

Stuart Cording: Embedded systems are traditionally programmed in C. But you decided to ignore that and jump straight into Rust. Why?

Scott Mabin: I'd used Rust in my final year project at university. I built a smartwatch — not smart by today's standards, but it did more than tell the time. Part of that project explored whether Rust could replace C for firmware development and where the tradeoffs lay. And, that's when I fell in love with it. I sat there, thinking, "why isn't everyone using Rust?" It offered so much that, once compiled, there was peace of mind that a whole category of memory and race conditions had been eliminated. Of course, I understood that some projects have legacy and that you won't want to learn and deploy a new language without good reason. Still, it just seemed crazy that more people weren't turning to Rust.

Stuart Cording: Rust is already community-supported on STM32 and some Nordic devices. Why did you go for Espressif back then?

Scott Mabin: At home, I had all these ESP32s lying around and wondered, "Surely I can program these things in Rust, too?" Well, that quickly took me down a pretty deep rabbit hole. The biggest issue was that the core, Xtensa, which powers devices such the ESP8266, was only supported by the GCC compiler but not LLVM. I looked at *mrustc* [5], a tool that converts Rust to C and can then be compiled, but this cross-compilation process failed to convince. Luckily, Espressif released a fork for the LLVM toolchain to support Xtensa and, while initially hard to use, it meant that you could compile native Rust for ESP32 devices. With that, I wrote my first *blinky* LED code in Rust; well, I say "in Rust" but it basically just wrote to registers and wasn't very Rust-like in its structure. However, it was an early success, which I documented in my first blog post [6] recording my journey with Rust on ESP32.

Stuart Cording: So you were tinkering with Rust on ESP32. How did the relationship with Espressif unfold?

Scott Mabin: I started the *esp-rs* group on GitHub in my spare time to collate Rust projects for ESP32. With other community members, we'd been implementing support for peripherals, such as SPI and others. Unfortunately, Wi-Fi had still eluded us. Then, after a year of community work and blogging about our progress, Espressif inquired about hiring me to ensure Rust support for their ecosystem. The team at Espressif had always been very helpful toward us, responding to support requests on their forums or Reddit. But, when they reached out, it became pretty clear that they'd been interested in Rust for a while.

Stuart Cording: Are there any specific application spaces driving that interest in Rust, or is it a bit of crystal-ball gazing on the part of Espressif? And, what's the level of commitment like?

Scott Mabin: There is definitely customer interest. Customers are primarily using Rust in IoT projects or internet-connected applications. Most projects are pretty simple, such as data loggers. Then, there are more complex applications such as a full-body VR tracker, and an open hardware night sky sensor. There is also a certain amount of future-proofing by Espressif, preparing for when Rust plays a more significant role in the lives of embedded system developers. Around 10 or 11 of us are contributing to Espressif's Rust Enablement Team, of which about half are full-time resources. On top of that, Rust embedded has an awesome community that contributes as well.

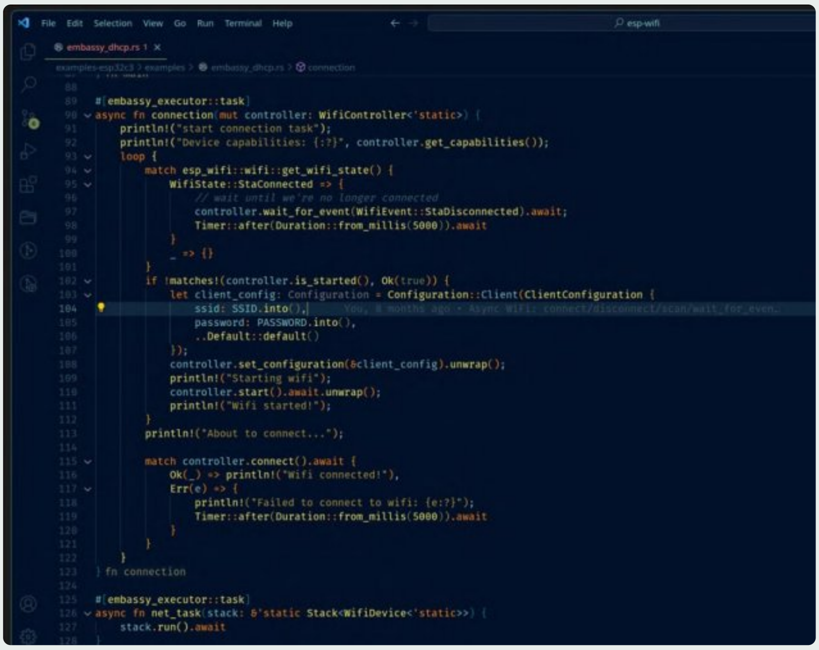
> *"In every other C project I've worked on, someone sat and wrote code for standard data structures. That time can now be spent on application development."*

Stuart Cording: You've done some embedded C development and now use Rust on microcontrollers daily. What do traditional C programmers need to consider as they transition to Rust?

Scott Mabin: If you're coming from pure C development and have no experience with C++, it's quite difficult. The most radical change is that Rust is quite type-heavy. Then, there's the resource side. Small microcontrollers with a few kilobytes of SRAM and flash will struggle, due to memory constraints. In such cases, you'll end up writing C-like Rust to keep the memory size down, thereby losing some of the benefits of this new language. You'll still get the memory safety advantages of Rust, so it's still better than C. In an apples-to-apples comparison, C and Rust applications perform about the same. In some cases, Rust is better, but the binary size is typically larger.

Stuart Cording: While microcontroller vendors offer great support with peripheral drivers, version control of all that code for the developer as the integrator is often a mess. Will Rust crates be a reason to change programming language?

Scott Mabin: Yes, I think crates are a huge benefit — a big bonus that goes beyond the language itself. Crates offer a packaging ecosystem or, at the very least, a method for pulling in dependencies without manually defining them in your build system. The *embedded_hal* [7] crate offers a trait-based interface for common peripherals, such as I²C. If you then take an I²C temperature sensor driver, it simply works on top. Then, if you go and change your microcontroller or swap to a new temperature sensor, it still works, which is really nice. But that just scratches the surface of it. Crates such as *heapless* [8] allow the creation of statically allocated data structures such as queues,


File Edit Selection View Go Run Terminal Help
esp-wifi
embassy_dhcp.rs 1
examples-esp32c3 > examples > embassy_dhcp.rs > connection
You, 8 months ago • Async WiFi: connect/disconnect/scan/wait_for_even…

```rust
#[embassy_executor::task]
async fn connection(mut controller: WifiController<'static>) {
    println!("start connection task");
    println!("Device capabilities: {:?}", controller.get_capabilities());
    loop {
        match esp_wifi::wifi::get_wifi_state() {
            WifiState::StaConnected => {
                // wait until we're no longer connected
                controller.wait_for_event(WifiEvent::StaDisconnected).await;
                Timer::after(Duration::from_millis(5000)).await
            }
            _ => {}
        }
        if !matches!(controller.is_started(), Ok(true)) {
            let client_config: Configuration = Configuration::Client(ClientConfiguration {
                ssid: SSID.into(),
                password: PASSWORD.into(),
                ..Default::default()
            });
            controller.set_configuration(&client_config).unwrap();
            println!("Starting wifi");
            controller.start().await.unwrap();
            println!("Wifi started!");
        }
        println!("About to connect...");

        match controller.connect().await {
            Ok(_) => println!("Wifi connected!"),
            Err(e) => {
                println!("Failed to connect to wifi: {e:?}");
                Timer::after(Duration::from_millis(5000)).await
            }
        }
    }
} fn connection

#[embassy_executor::task]
async fn net_task(stack: &'static Stack<WifiDevice<'static>>) {
    stack.run().await
}
```

▲

Figure 1: An example of a Wi-Fi application written in Rust for the ESP32.

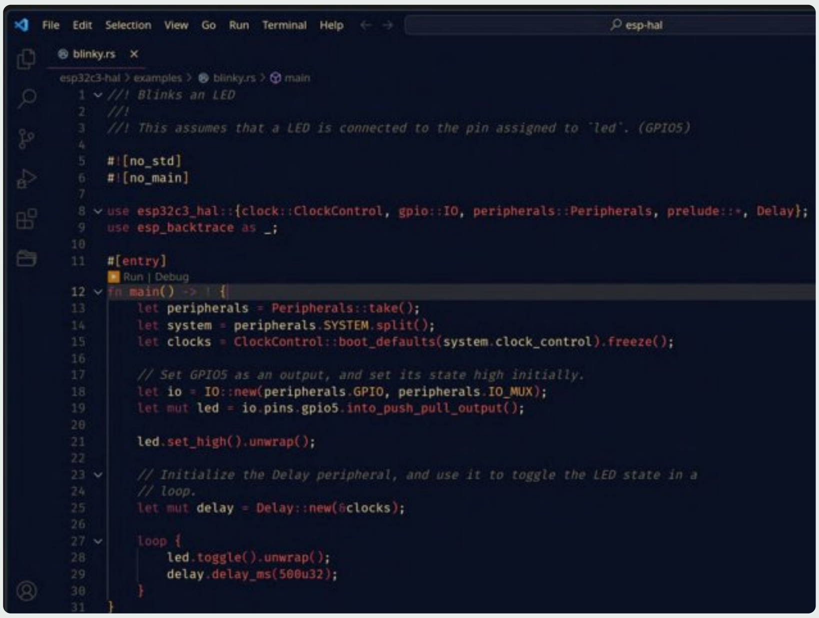
vectors, hash tables, hash sets, and strings. In every other C project I've worked on, someone sits and writes a queue, spending many hours perfecting a standard programming data structure. That time can now be spent on application development.

Stuart Cording: Much like C, Rust offers a standard library. Embedded developers abhor the standard library, so, can they also do away with Rust's?

Scott Mabin: Historically, I think we've had a hard time describing the use cases where a professional embedded programmer should use a standard library [9]. They take one look and say, "that's way too much overhead," and I would completely agree with that. However, the Rust standard library is really useful in a number of ways. Firstly, if you know Rust but not embedded, it feels familiar. You get all the threads, networking, and other things you're used to. Secondly, if you program ESP32s using the ESP-IDF [10] (Espressif IoT Development Framework), you can create Rust standard library projects and get started. If you compare it to the Python standard library, it is actually quite thin and, while there is overhead, it's not as bad as it sounds. It makes a few more assumptions than the core library (which you'll need as a minimum), such as the availability of threads and networking.

Figure 2: Scott Mabin recommends VS Code for Rust development. Here, the classic LED blinky code can be seen.

▼


File Edit Selection View Go Run Terminal Help
esp-hal
blinky.rs
esp32c3-hal > examples > blinky.rs > main
Run | Debug

```rust
//! Blinks an LED
//!
//! This assumes that a LED is connected to the pin assigned to `led`. (GPIO5)

#![no_std]
#![no_main]

use esp32c3_hal::{clock::ClockControl, gpio::IO, peripherals::Peripherals, prelude::*, Delay};
use esp_backtrace as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take();
    let system = peripherals.SYSTEM.split();
    let clocks = ClockControl::boot_defaults(system.clock_control).freeze();

    // Set GPIO5 as an output, and set its state high initially.
    let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
    let mut led = io.pins.gpio5.into_push_pull_output();

    led.set_high().unwrap();

    // Initialize the Delay peripheral, and use it to toggle the LED state in a
    // loop.
    let mut delay = Delay::new(&clocks);

    loop {
        led.toggle().unwrap();
        delay.delay_ms(500u32);
    }
}
```

Many people develop embedded Rust without the standard library (*no_std* [11]) and, if you know what you're doing, definitely go with the *no_std* approach if it makes sense for the project. However, you will have to pick and choose the pieces you want and glue them together. So, for example, the Wi-Fi crate has an example of talking to an HTTP server or setting up an access point and running a server (**Figure 1**). You can do everything in *no_std* that you can do with *std* — it's just more work. With *std*, it's a "batteries included" kind of environment.

Stuart Cording: Is part of the issue that it is just time to accept that we need microcontrollers with large memories?

Scott Mabin: I think that, no matter how big microcontrollers get, there's always going to be someone who wants to make a million of some device and needs the smallest and cheapest solution possible. There's always going to be a use case for assembly and C — and maybe even Rust — in some specific context for that small, compact application. But, I do think there is a general trend toward bigger microcontrollers and that the developer experience is taking more of a priority over hardware costs. That's just my observation in my short time in the industry, but it looks that way.

Stuart Cording: It's still early days for Rust. How will Espressif provide training to get developers up to speed?

Scott Mabin: We joined up with Ferrous Systems [12], a Rust consulting company, to provide a training course that we've made open source. There are training materials for the standard library approach, and we're also almost complete in adding the no-standard library approach.

Stuart Cording: And what about development environments and debugging tools?

Scott Mabin: I admire some of the aesthetics of environments such as vim, Neovim, or Emacs setups, but because I'm not experienced with those tools, they just get in the way. So, I use VS Code (**Figure 2**) — it does the job for me. With the newer ESPs (C3 and above), we have a module called the USB Serial JTAG. It's an extremely small, built-in peripheral that offers a serial port and a JTAG device (**Figure 3**). To use it, just connect it to your PC's USB port, and it's detected automatically by OpenOCD or *probe-rs*, a Rust debugging toolkit with a purpose similar to OpenOCD.

Stuart Cording: Supporting a new language on a processor is a huge task. Where are you currently, and what's still to do?

Scott Mabin: We pretty much have everything in place for the standard library, but we don't have everything covered for ESP-IDF. ESP-IDF is vast, has been around for years, and is written in C, so we use *bindgen* to create bindings with Rust. Basically, we need to look at the interfaces that aren't yet implemented and create a Rust API for them. We have to decide what to prioritize on a case-by-case basis. For *no_std*, we're talking programming in pure Rust, so we need to write code for all the existing hardware. We have Wi-Fi, Bluetooth, and Thread support on all the chips. So, on a scale of 0 to 100, I'd say we're around 65 to 70 percent of the way there. Unfortunately, it's a bit of a moving goalpost, as we continuously have to support new devices as they're released. We've discussed creating some elements of ESP-IDF in Rust where it makes sense. For example, Rust is very well suited for parsing byte streams. So, if a new component is needed, maybe we'll create it in Rust and provide a C API. We'll have to see whether the benefits outweigh the efforts.

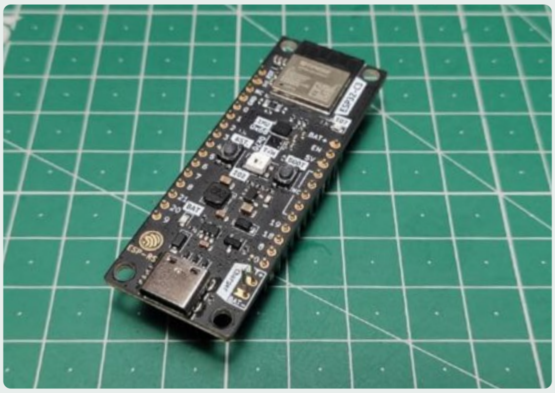


Figure 3: The ESP32-C3 integrates a USB-based debug module with a serial interface, avoiding the need for external probe hardware.

Stuart Cording: Finally, where do you, as an embedded developer, go for more information on Rust?

Scott Mabin: I mostly hang around in the ESP Rust matrix channel [13] and various other Rust embedded channels. Other than that, Google, sometimes Reddit; reading good Rust code can help me understand and learn things I didn't know.

230616-01

Questions or Comments?

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

About Scott

Scott Mabin is an embedded software engineer. Having initially explored the capabilities of embedded Rust at university, he decided to experiment with it on the ESP32 in his spare time. These efforts led to Espressif asking him to join their team to focus on improving support for Rust on their wireless MCUs and AIoT solutions.

WEB LINKS

[1] TIOBE Programming Community index: https://tinyurl.com/tioberust

[2] PDP-7 [Wikipedia]: https://tinyurl.com/pdp7wikipedia

[3] PDP-11 [Wikipedia]: https://tinyurl.com/pdp11wikipedia

[4] MSRC Team, "A proactive approach to more secure code," Microsoft, July 2019: https://tinyurl.com/msrcsecurecode

[5] mrustc Project: https://tinyurl.com/mrustcproject

[6] S. Mabin, "Rust on the ESP32," September 2019: https://tinyurl.com/rustesp32

[7] embedded_hal Crate Documentation: https://tinyurl.com/embeddedhalcrate

[8] heapless Crate Documentation: https://tinyurl.com/heaplesscrate

[9] The Rust on ESP Book, "Using the Standard Library (std)": https://tinyurl.com/rustbookstd

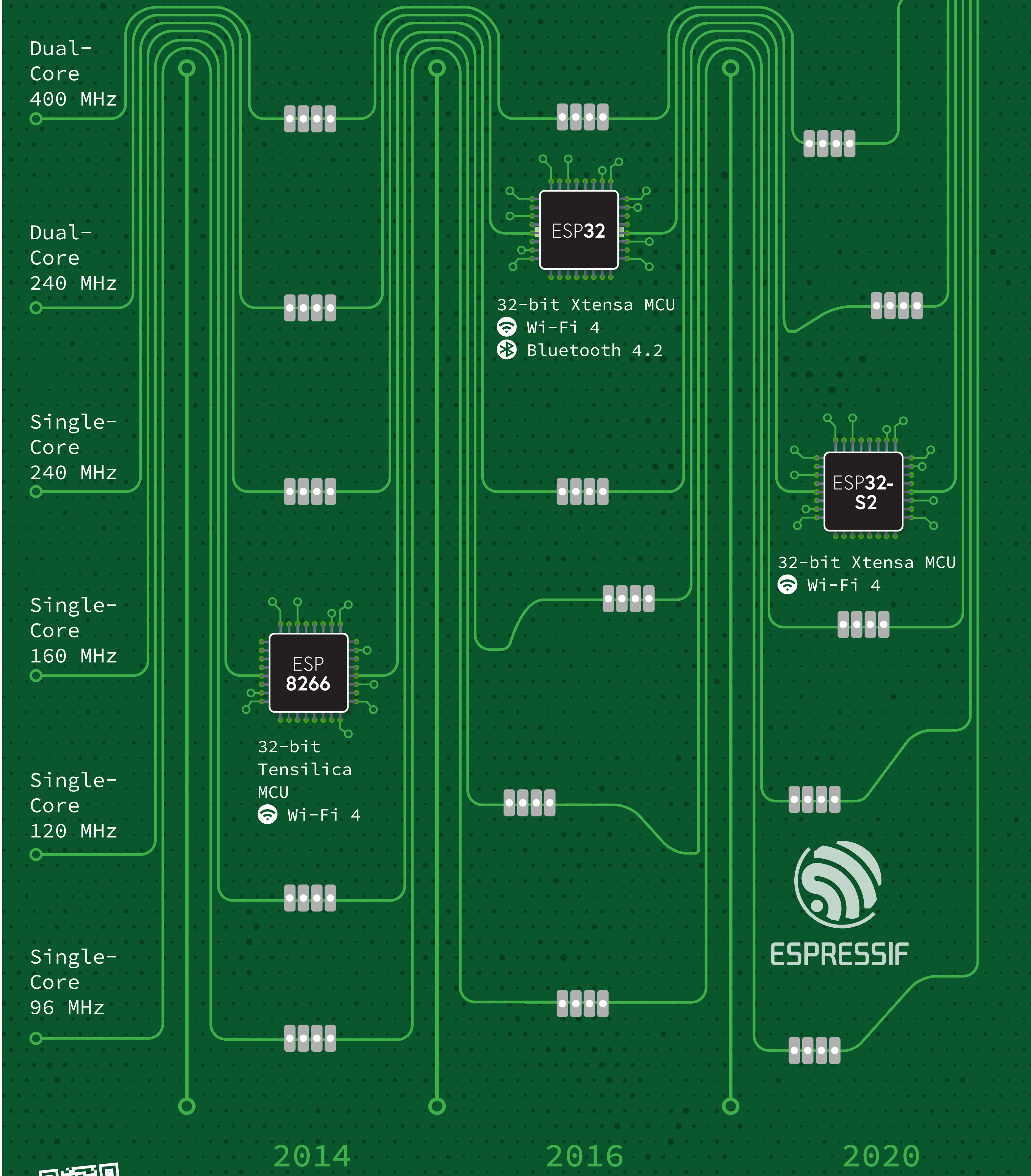
[10] ESP-IDF Programming Guide, "Get started": https://tinyurl.com/espidfgetstarted

[11] The Rust on ESP Book, "Using the Core Library (no_std)": https://tinyurl.com/rustbooknostd

[12] Training, "Embedded Rust on Espressif": https://tinyurl.com/rustonespressif

[13] Rust user group on Matrix: https://tinyurl.com/rustmatrixug

Espressif's Series of SoCs



Dual-Core 400 MHz

Dual-Core 240 MHz

Single-Core 240 MHz

Single-Core 160 MHz

Single-Core 120 MHz

Single-Core 96 MHz

ESP 8266

32-bit Tensilica MCU
Wi-Fi 4

ESP32

32-bit Xtensa MCU
Wi-Fi 4
Bluetooth 4.2

ESP32-S2

32-bit Xtensa MCU
Wi-Fi 4

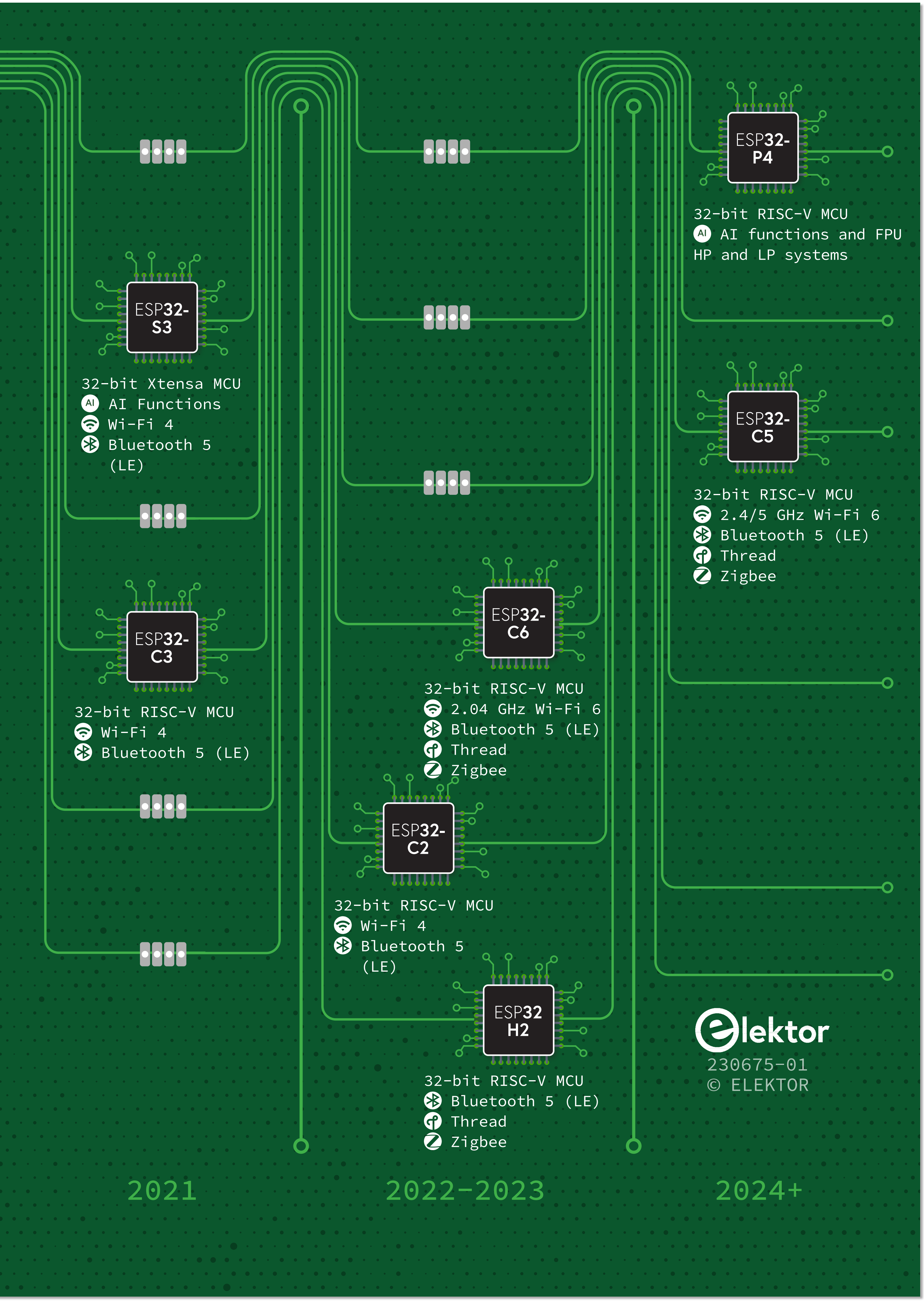
ESPRESSIF

2014

2016

2020

elektormagazine.com/posters



2021

ESP32-S3

32-bit Xtensa MCU
AI AI Functions
Wi-Fi 4
Bluetooth 5 (LE)

ESP32-C3

32-bit RISC-V MCU
Wi-Fi 4
Bluetooth 5 (LE)

2022-2023

ESP32-C6

32-bit RISC-V MCU
2.04 GHz Wi-Fi 6
Bluetooth 5 (LE)
Thread
Zigbee

ESP32-C2

32-bit RISC-V MCU
Wi-Fi 4
Bluetooth 5 (LE)

ESP32 H2

32-bit RISC-V MCU
Bluetooth 5 (LE)
Thread
Zigbee

2024+

ESP32-P4

32-bit RISC-V MCU
AI AI functions and FPU
HP and LP systems

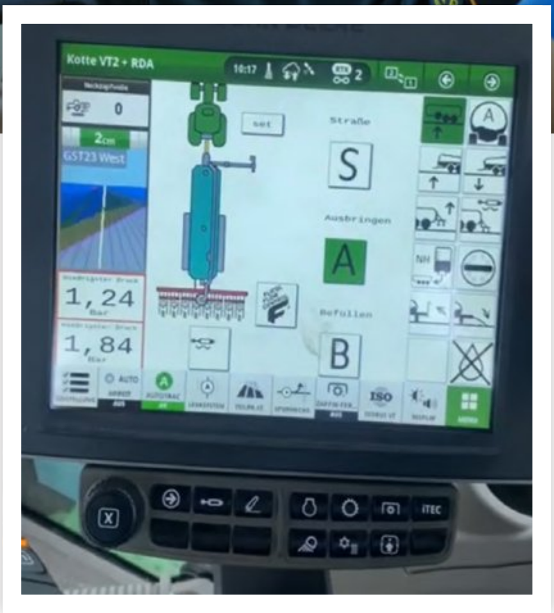
ESP32-C5

32-bit RISC-V MCU
2.4/5 GHz Wi-Fi 6
Bluetooth 5 (LE)
Thread
Zigbee

elektor
230675-01
© ELEKTOR

Building a PLC with Espressif Solutions

With the Capabilities and Functionality of the ISOBUS Protocol



▲ *Figure 1: ISOBUS (ISO 11783) user interface on a John Deere monitor. It controls a slurry tanker.*

By Franz Höpfinger, HR Agrartechnik

HR Agrartechnik GmbH needed a low-cost yet powerful PLC controller with ISOBUS (ISO11783) capabilities. The combination of an ESP32 controller from Espressif with the ESP-IDF, together with the Eclipse 4diac™ framework, resulted in the logiBUS® project.

The Challenge

Today's agricultural machines require flexible control systems. Lot sizes are usually small, and machine lifetime is long, so retrofitting control systems to existing machines in the field is quite common. In **Figure 1**, you can see the system control monitor's user interface after the retrofit.

The effort of learning C or C++ and the management of libraries and systems is complex. Debugging with JTAG is inflexible, requiring extra tools and access to the microcontroller. On the other hand, many programmers are available for PLC programming, and graphical-based programming is easier to learn.

ISOBUS, also known as ISO 11783, is a communication protocol developed for agricultural equipment to facilitate the exchange of data between different types of farm machinery and implements. "ISOBUS" consists of ISO, for the International Organization

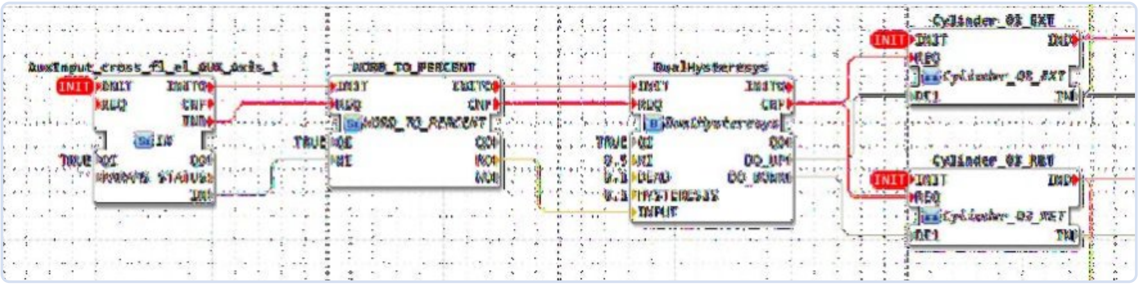


Figure 2: This is a source code listing that shows how programming in IEC 61499/Eclipse 4diac™ works. You do not write lines of source code, but use functional blocks (FBs) instead.

for Standardization, and bus, for a communication system between multiple devices.

In that context, we wanted to design a PLC system consisting of a runtime system on the device and an IDE on the PC that would provide a flexible, easy-to-use, fast automation system with model-based, low-code, graphical programming, and powerful libraries. A strict separation between a reusable, flexible runtime environment and the application was the highest priority in our requirement specification. The second point was the ability to really observe any variables and function blocks online. Further notable points were the software stack supplier's support quality, as well as resource usage. We wanted to avoid an expensive and full-blown embedded LINUX system, mainly because of boot times. Agricultural controls are switched on and off probably a dozen times a day.

We analyzed several solutions, both closed- and open-source, and finally selected Eclipse 4diac™ [1]. "Why not the OpenPLC project?" some of you Elektor readers might ask. The answer is obvious: The main two requirements were not fulfilled: OpenPLC has no strict split between application and runtime, nor an online watch functionality.

The Solution

For the logiBUS® solution [2], we combined the software side (listed below) with our basic application:

- ESP-IDF development framework (currently Version 5.1.1)
- Eclipse 4diac™ FORTE PLC Runtime environment, an open-source IEC 61499 runtime environment
- CCI ISOBUS driver, an ISO 11783 protocol stack

This gives a PLC runtime system that is independent of the desired application. The programming is done from the Eclipse 4diac™ IDE. The flashing of the application can be done over a TCP/IP Connection, so via Wi-Fi or Ethernet. The solutions bring real PLC features such as online watch and online change.

On the hardware side, an ESP32 with PSRAM is good to have because the model uses a lot of RAM, so, for bigger models, not having a PSRAM is a problem. For the ISOBUS connection, we use the ESP32 TWAI peripheral as a CAN bus controller, together with an external Infineon TLE9251VSJ CAN transceiver. Some models of the hardware, especially the ones targeted for education, are open source[3].

With the logiBUS® project still growing and being extended, we have already realized a dozen real customer projects. In **Figure 2**, you'll see an example of a source code listing for one of them.

Some demo applications, such as Bale Counter [4] and Slurry Tanker [5], were open-sourced.

The system also seems to be well-suited to other industries, such as building automation [5]. Therefore, we made an open-source version of logiBUS® without the ISOBUS stack, but with the same functionality and same power.

We hope to build a community around the topic of open-source PLC systems [6].

230610-01

Questions or Comments?

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

WEB LINKS

[1] Eclipse 4diac™: https://eclipse.dev/4diac

[2] logiBUS® Homepage: https://logibus.tech

[3] logiBUS® open-source derivatives for e.g. Building Automation (non ISOBUS): https://gitlab.com/meisterschulen-am-ostbahnhof-munchen

[4] Bale Counter: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac_EasyExampleCounter

[5] Slurry Tanker: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac-SlurryTanker-sample

[6] Resources and Wiki: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen

Import Edit Export I_Made_a_VGA_Card_

The ESP32-S3 VGA Board

Bitluni's Exciting Journey Into Product Design

Compiled by Elektor and Espressif

YouTuber Bitluni designed an ESP32-S3 VGA board to achieve remarkable resolution and color fidelity. He leveraged the Espressif ESP32-S3's LCD peripheral, which replaced the I2S of the prior version. Let's retrace his steps. By doing so, you'll learn a lot about the process of bringing a new product to life.

Years ago, well-known German YouTuber Bitluni made a VGA library and a few VGA boards for the ESP32 (**Figure 1**). Many of his devoted followers loved the design, but he admits that he's not a manufacturer and that he sold only a limited number. Things have changed since then! The ESP32 API has been updated a few times, and the ESP32-S3 hit the market (**Figure 2**). With many of his followers asking him to make a new version of the library and the board since it was incompatible with the S3, Bitluni decided to give it a go. This is how the ESP32-S3 VGA was born (**Figure 3**).

When Bitluni decided to design the new board (**Figure 4**), he wanted it to be more accessible. The earlier boards required assembly and an extra dev board. Fortunately, he had gained enough practice to design one with everything included (**Figure 5**). Plug-and-play without much soldering! One thing he wanted was for it to be breadboard-friendly, so that if you soldered headers on, you could put it on a breadboard and have an extra two rows to access the pins (**Figure 6**). The VGA connector was quite big, and the antenna needed some clearance, so this shape seemed reasonable.

Designing ensued (**Figure 7**), boards were ordered, and the results were gorgeous (**Figure 8**)! Bitluni assumed the VGA connectors he had would be compatible with the footprint of the PCBs, but he soon discovered the pins didn't match (**Figure 9**)! Luckily, he had a few narrower connectors on hand. These barely fit with the pins and the board there was an overhang (**Figure 10**). Since there were no tracks in this area, he decided to simply mill off a few millimeters and the problem was solved (**Figures 11...14**)! With the assembly complete, he was ready for some code (**Figure 15**).

Up to that point, Bitluni had not looked at the technical reference manual, but, once he did, he realized why his old library wasn't working for the S3. Espressif had removed the parallel I2S Mode and made a new peripheral for cameras and LCDs. So, some studying (**Figure 16**), thinking, and testing ensued (**Figure 17**)! Then more studying, more thinking (**Figure 18**), more testing, and then some desoldering and reconfiguring, which was followed by more reconfiguring, more soldering, and more testing (**Figure 19**). He sure was busy! Fortunately, after putting in all that effort, it worked! Not only 640×480, but also 800×600 (**Figure 20**).

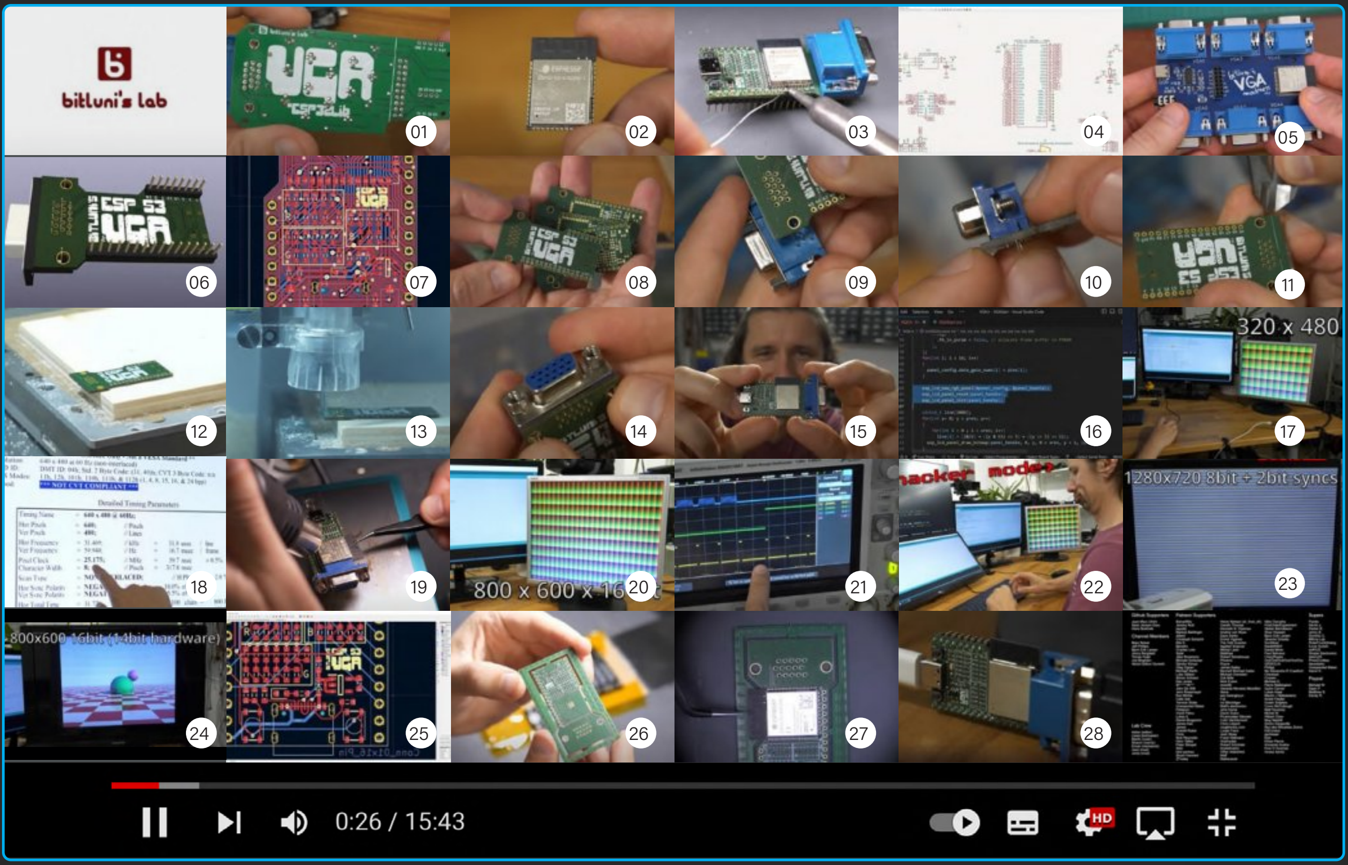
After some testing on different devices, Bitluni discovered some sync issues. Three of his other screens seemed to adjust the sync at the start of each frame. That was a showstopper, he explains: Once he zoomed in on his scope, he found a slight delay at each start of the frame (**Figure 21**). So, more studying and thinking ensued, and he activated "Hacker Mode" (**Figure 22**)!

Favorable developments followed. "I was finally able to get a clean, continuous signal," Bitluni explains. "I was finally able to share some success on the live stream." He was even able to squeeze out 1280×720p and 8-bit (**Figure 23**). It wasn't noise, but super-small squares! A day later, he was smiling again (**Figure 24**). Next, he added additional bits, reorganized things (**Figure 25**), and ordered new boards. Fast-forward one week: **Figure 26**, **Figure 27**, and **Figure 28**. Success! ⏮

230529-01

00:00:03:15 Fit

_That_Blew_My_Mind



bitluni

231K subscribers

Related Products

- **Espressif ESP32-S3-EYE**
  www.elektor.com/20626
- **D. Ibrahim, *The Complete ESP32 Projects Guide* (Elektor, 2019)**
  www.elektor.com/18860

Watch: "I Made a VGA Card That Blew My Mind"

For an in-depth video about this project, head over to Bitluni's YouTube channel:

Scan the QR-code...

...and watch the video in AR on this article.

Or

Watch it directly on YouTube
https://youtu.be/muuhgrige5Q

1/2 00:00:18:14

Acoustic Fingerprinting on ESP32

Song Recognition With Open-Source Project Olaf

By Joren Six, Ghent University (Belgium)

A few years ago, computer scientist Joren Six was tasked with enabling a wearable computing platform — that is, a microcontroller — to recognize a song. His experiments in using an ESP32 led to a fully-fledged, multi-platform, open-source project that he called Olaf, which stands for “Overly Lightweight Acoustic Fingerprinting.” Olaf is a music-recognition library that’s easy to use on embedded platforms with severely limited memory and computing power. Follow him in his journey!

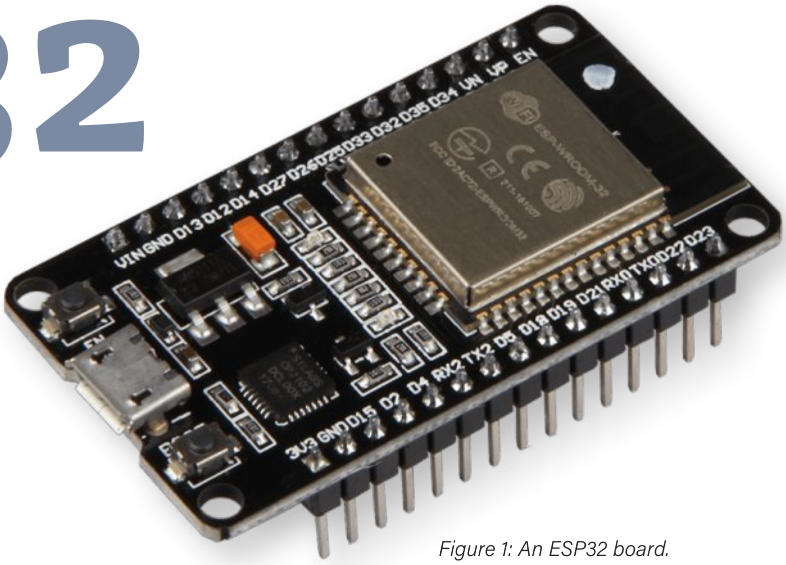

VIN GND D13 D12 D14 D27 D26 D25 D33 D32 D35 D34 VN VP EN
3V3 GND D15 D2 D4 RX2 TX2 D5 D18 D19 D21 RX0 TX0 D22 D23
ESP-WROOM-32

Figure 1: An ESP32 board.

Sometime around 2019, I was a computer scientist at Ghent University, Belgium. I was tasked with developing audio recognition technology for an electronic costume. The concept involved having lights in the costume synchronize with a specific song. Only one particular song should activate the lights; all other music should be disregarded. Typically, music recognition and synchronization are accomplished using audio fingerprinting techniques. The challenge was that this recognition needed to operate on an affordable, battery-powered microcontroller with limited processing power and memory. At the time, I successfully created a first prototype, but the idea remained in my mind.

As my daughter’s fourth birthday approached, I decided to transform the prototype into an extravagantly engineered birthday gift: an “Elsa” dress that reacts to the song “Let It Go” from the *Frozen* soundtrack [1]. Building upon the first prototype, I ordered an RGB LED strip, a robust Li-Ion battery, an I^{2}S digital microphone, and, naturally, an Elsa dress.

I had an ESP32 (**Figure 1**) microcontroller on hand and used it as the central component of the system. It supports I^{2}S, includes a floating-point unit (FPU), is easy to integrate with LED strips, and has enough RAM. The FPU simplifies the use of the same code on both conventional computers and embedded devices, eliminating the need for fixed-point math.

Building the ESP32 Version

The initial version of the project utilized an ESP32 Thing by Sparkfun — chosen for its integrated battery circuit — paired with a MEMS microphone, specifically an INMP441. This setup was complemented by a LED strip featuring individually addressable LEDs, a generic type that can be easily found on eBay or AliExpress.

From a hardware point of view, it’s a simple job. It’s a matter of connecting the I/O pins of the microphone module (SDA, SCK, WS) to suitable GPIOs on the ESP32, such as, for example, GPIO 32, 33, and 25 respectively. Of course, the microphone module needs power too.

The wiring of the LEDs (**Figure 2**) will vary depending on the type of LED strip that is actually used. For WS2812B-based strips, three wires (Power, Ground, Data) are needed. I used the *FastLED* library in the ESP32 code, so if you want to replicate the project, make sure that the LED strip you have is compatible with *FastLED*.

Figure 2: RGB LED strip.

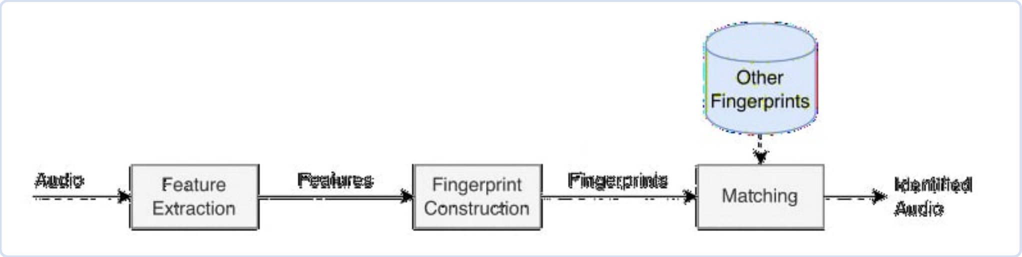


Figure 3: Sound recognition block diagram.

After soldering the components together and with assistance from my partner to sew in the LED strip, the project came to fruition. In the video embedded at [1], you can observe the outcome of our efforts. The video initially features a song that is not recognized and doesn't trigger the lights. Then, "Let It Go" is played and correctly recognized. Once the song pauses, the lights remain on for a brief period before turning off, as designed to accommodate gaps in recognition. Finally, the song resumes, and it is once again accurately identified. You can read about the latest updates of the project at [2].

Of course, as is often the case with microcontroller projects, all the magic happens in the code.

What's Under the Hood?

How can we use computers to recognize songs or music? There are a few possible ways, one common approach being the spectral-peak-based recognition. This is the technology used by Olaf, and well-known music identification services such as Shazam.

The concept behind this is quite straightforward: The app takes the audio recorded on your phone and transforms it into a format that a computer can readily compare to other songs (**Figure 3**). In the field of acoustics, this process is referred to as audio fingerprinting, where a song is condensed into a unique signature that remains identifiable even amid substantial background noise.

These signatures are based on spectrograms, which visually represent the song's frequency content over time. Spectrograms also display the signal power level, reflecting the perceived loudness through color variations. However, if Shazam were to match spectrograms like these directly, it would require an exceptionally long time to provide a result. One major breakthrough is concentrating on the peaks in the spectrogram, since these peaks contain salient information that is also processed by the human brain.

So, Olaf records a sample of sound, then computes a spectrogram. Next, the spectrogram is simplified into a scatter plot of its frequency peaks. An example of such a spectrogram with highlighted peaks is shown in **Figure 4**.

Now these scatter plots, which essentially outline the most prominent signals at various frequencies over time, must be matched to a database of many known songs. Or, in the case of Elsa's costume, to a single specific song.

Instead of searching for a match in a database for all these points in the exact sequence, a clever technique is to connect nearby peaks to create numerous pairs. Subsequently, the algorithm searches for matching pairs within a structured database containing any number of songs. If enough matches are found, and they align correctly over time, Olaf (or Shazam) can identify the song.

Evolution of the Project

The initial code for the 2019 prototype was not very well organized. However, during my second attempt, I made significant improvements, reaching a level of comfort that allowed me to share the code on GitHub. At this point the project was named *Olaf – Overly Lightweight Acoustic Fingerprinting* [3].

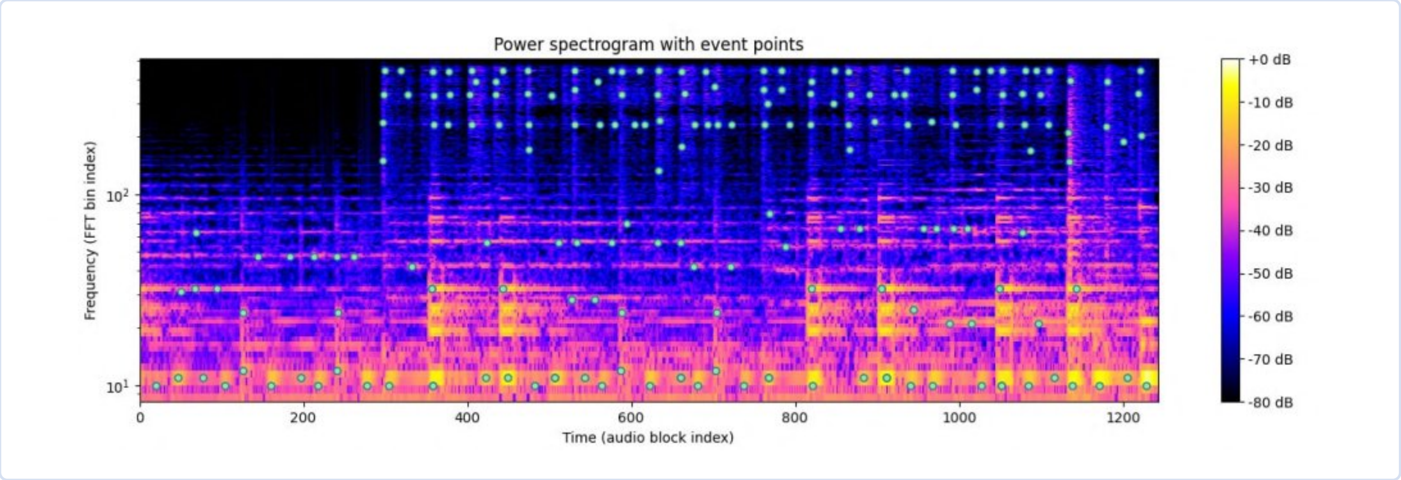


Figure 4: A spectrogram shows the how present frequencies are in music at a given time. The green dots are extracted by Olaf and show peaks in the spectrogram.

The code underwent several iterations and expanded beyond its original purpose, evolving into a versatile, general-purpose acoustic fingerprinting system with numerous potential applications. Olaf's impressive performance can be attributed, among other things, to its resource-efficient design and its utilization of the *PFFFT* library.

This acronym might not mean a lot to some readers, especially if, like Bob Pease, your favorite programming language is solder! *PFFFT* [4] stands for *Pretty Fast FFT* library, and was designed as a much lighter alternative than the well-known *FFTW*. It's indeed compact and operates very quickly, making it ideal for the ESP32.

On embedded devices, reference fingerprints are stored in memory, eliminating the need for a database. On traditional computers, fingerprints are stored in a high-performance database system — LMDB. While delving into its advantages falls beyond the scope of this article, I encourage readers to explore it further if interested.

Olaf is no longer just a cool gadget based on ESP32. It has been turned into a full application/library designed for landmark-based acoustic fingerprinting. Olaf has the capability to efficiently extract fingerprints from an audio stream, allowing for the storage of these fingerprints in a database or the identification of matches between extracted and stored fingerprints.

There seemed to be a scarcity of lightweight acoustic fingerprinting libraries that were easy to deploy on embedded platforms, which often have severe limitations in memory and computational resources.

Olaf is written in C, and I have endeavored to keep the code as portable as possible. It primarily targets 32-bit ARM devices such as certain Teensy models, specific Arduino boards, and the ESP32. Other modern embedded platforms with similar specifications may also be compatible. No doubt that Olaf, which is open source, will stimulate the creation of innovative projects combining music/sound recognition and IoT devices!

A major consequence of these efforts about portability is that Olaf also runs on personal computers, and it runs fast. The code can be compiled to run on your PC locally, but another alternative is still possible. Olaf can run in a web browser too! See the "Running It in a Web Browser" section.

The source code, with a working example for ESP32 and small debug tools that I wrote, can be found on my GitHub. There is also a PlatformIO project file for reference [6].

Build It Yourself Using an ESP32

For audio processing, the Olaf project required a bit of RAM, which often posed a problem with many microcontrollers. With a key-value store running on a PC, Olaf required about 512 KB of RAM, whereas the ESP32 version required considerably less — around 200 KB. I only work sporadically with microcontrollers, so an easy-to-use environment is key: The limited time spent with embedded devices should not be consumed by the setup and maintenance of a development environment. ESP32's IDE with PlatformIO or the Arduino IDE proved to fit that bill. Computationally, there was more headroom, and many microcontrollers worked (Teensy 3+, RP2040, any Cortex-M thing). An FPU might help to keep energy usage down, especially relevant when working on battery power.

I have also been using a couple of M5StickC's from the company M5Stack as they provided an easy-to-use package and come with an integrated microphone: I'm more into software than hardcore hardware hacking, so they were a convenient choice. I also constructed a couple of home automation devices (detailed on my website) as well as some other projects, such as controlling home ventilation or monitoring the rainwater tank's level.

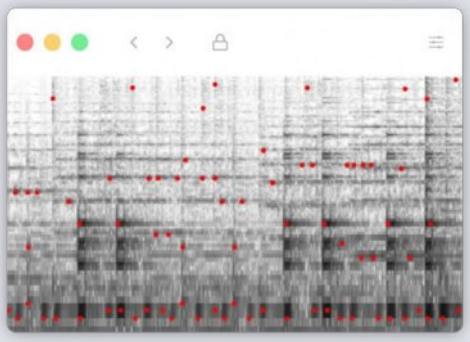
While preparing this article, I added code for an ESP32 demo, as well as additional documentation, to my GitHub repository [7]. Now, the latest version of Olaf operates reliably on ESP32 with an easily obtainable MEMS microphone. On that help page, you'll also find instructions on how to utilize I^{2}S microphones such as the INMP441.

Running It in a Web Browser

Have you heard about WebAssembly [5] (or WASM)? WASM is a portable binary code format, along with a corresponding text format, designed for executable programs. Its main objective is to facilitate the development of high-performance applications within web pages.

As it turns out, there are ways to compile C code to WASM, such as the Emscripten compiler. According to its website, Emscripten allows you to run C and C++ code on the web at nearly native speed, all without the need for plugins. Combining the Web Audio API with the WASM version of Olaf opens up possibilities for web-based acoustic fingerprinting applications.

On my blog, you can experiment with Olaf right there in your browser and read my latest posts about it. The exact same code that runs on the ESP32, as demonstrated earlier, is now running in your browser. This means that Olaf is actively listening to recognize the song "Let It Go" from the Frozen soundtrack. For your convenience, you can start playing the song from the YouTube video on the left of your screen, and on the right, you can initiate Olaf, allowing it to analyze incoming audio. Olaf calculates the Fast Fourier Transform (FFT) and visualizes it using Pixi.js. After a few seconds, the red fingerprints (see the screenshot) should transition to green, indicating a successful match. When you stop the song, the fingerprints will eventually return to red. Similar to the video demonstration mentioned earlier, transitioning from a match to no match takes a couple of seconds to account for gaps in recognition.



There's a demo program that transmits audio from the microphone over Wi-Fi to a computer, allowing one to listen to the microphone. This ensures that the microphone functions as intended and that the audio samples are accurately interpreted. It validates the I²S settings, including buffer sizes, sample rates, audio formats, and stereo or mono settings.

There's also another code example illustrating how to match incoming audio from an INMP441 microphone to fingerprints stored in a header file. Unlike the PC version, the embedded version of Olaf doesn't utilize a key-value store, but a list of hashes stored in the *src/olaf_fp_ref_mem.h* header file, which serves as the fingerprint index.

By default, *src/olaf_fp_ref_mem.h* is included in the ESP32 code. To test and debug this header file, query the mem version of Olaf on your computer: `bin/olaf query olaf_audio_your_audio_file.raw "arandomidentifier"`. The ESP32 version is essentially identical to the mem version, with the only difference that the audio in the ESP32 version comes from a MEMS microphone input and not from a file.

Once you are certain that the mem version of Olaf operates as expected and the INMP441 microphone has been tested, the ESP32 version can be deployed on the ESP32 hardware using the Arduino IDE.

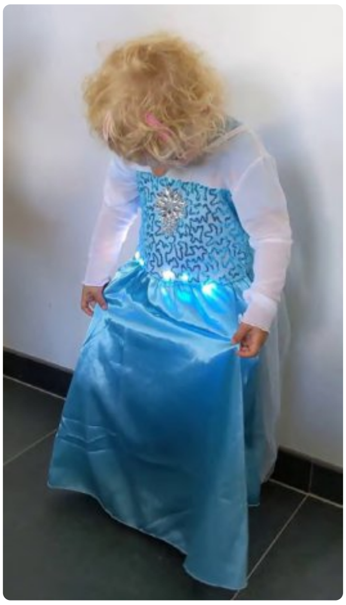


Figure 5: The ESP32-powered costume.

What's Next?

I hope this article has sparked some ideas for projects using acoustic recognition. It's always exciting to see how, by beginning with a small, playful project and refining it, you can achieve impressive results. In this instance, the project progressed from an initial prototype that didn't entirely meet my expectations to a second prototype based on an ESP32, which brought several improvements, and undoubtedly a lot of enjoyment for my daughter (**Figure 5**).

Over time and through iterative refinements, the project evolved into a comprehensive, portable, open-source, cross-platform, high-performance application and library. I have authored two academic papers on the subject to ensure that this work can be recognized and appreciated within the research community. Now, the question open to readers is: What exciting projects do you have in mind to use your ESP32 for?

230578-01

Questions or Comments?

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

About the Author

Joren Six is a computer scientist who does research in the field of Music Informatics, Music Information Retrieval, and Computational Ethnomusicology. He holds a PhD from Ghent University, Belgium, and is currently involved in several projects combining IT and acoustics.

WEB LINKS

[1] First prototype of the project: https://0110.be/posts/Olaf_-_Acoustic_fingerprinting_on_the_ESP32_and_in_the_Browser
[2] Latest updates regarding the project: https://0110.be/search?q=olaf
[3] The project's repository: https://github.com/JorenSix/Olaf
[4] Pretty Fast FFT library: https://bitbucket.org/jpommier/pffft/src/master
[5] WebAssembly on Wikipedia: https://en.wikipedia.org/wiki/WebAssembly
[6] Files for download: https://0110.be/files/attachments/475/ESP32-Olaf.zip
[7] ESP32 Olaf project on GitHub: https://github.com/JorenSix/Olaf/tree/master/ESP32

Circular Christmas Tree 2023

A High-Tech Way to Celebrate the Holiday Season

By Ton Giesberts (Elektor)

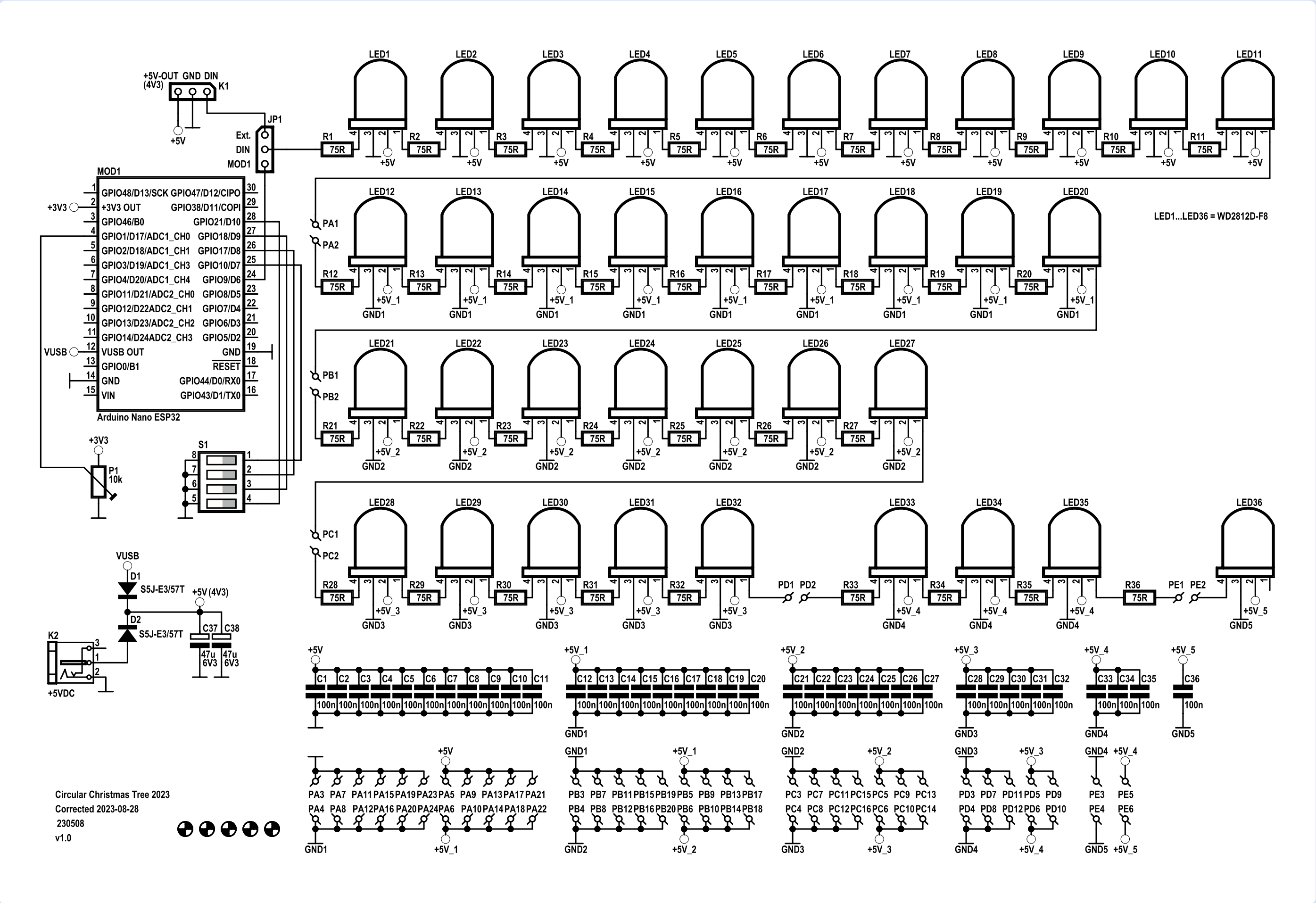
The WS2812D-F8 is a high-tech, digitally programmable 8 mm RGB LED that's individually addressable in daisy chains of up to 255 devices. In this 3D Circular Christmas Tree Project, 36 of these devices can be controlled externally or by an onboard Arduino Nano ESP32. The light effects are astonishing — you can't miss having this illuminating Elektor kit for the Holidays!

This Christmas tree is all about the shape of the construction and the use of 8 mm digital RGB LEDs. Its appearance is more like a real tree than most of the flat PCBs with a pine tree-styled outline. Instead of being just a plain 2D design, this is a real 3D one, and it's based on the two earlier versions, [1] and [2]. The new tree is realized by separating the five annular, concentric sections of the PCB supplied with the kit, and mounting them in sequence with some rigid wires that distance the boards, giving to this project the typical look of a conifer-shaped-tree. The project in [1] used a simple random number generator with two standard logic ICs and small white SMD LEDs. [2] was microcontroller-based, with the LEDs connected in a 6×6 matrix. This latest version uses 36× 8 mm digital RGB through-hole LEDs of type WS2812D-F8. This makes the circuit simple. Instead of a matrix, the inputs and outputs of the digital LEDs are connected in series, with each LED connected to a 5 V power supply. The chain of LEDs can be addressed like a NeoPixel LED strip. There are a large number of programs on the web, in many languages, that explain in detail how to manage these devices.

Connector K1 is connected to the internal +5 V (4.3 V), ground, and the input of the first LED (through jumper JP1). In this way, a variety of external microprocessors, microcontrollers, or modules can be used to control the LEDs of our Circular Christmas Tree. To make the tree independent of an external circuit, an Arduino Nano ESP32 module [3] can be mounted on the base PCB.

Schematic

The circuit consists essentially of 36 digital RGB LEDs of type WS2812D-F8, with their input (D_{in}) and output (D_{out}) pins connected in series, and each one of them powered by a 4.3 V power supply (**Figure 1**). The actual voltage on the LEDs is lower because of the drop caused by protection diodes D1 and D2. The LEDs can be controlled by an external microprocessor, microcontroller, module, or optional onboard Arduino Nano ESP32 module (MOD1).



+5V-OUT GND DIN
(4V3)
K1
JP1
Ext.
DIN
MOD1
+5V
MOD1
GPIO48/D13/SCK GPIO47/D12/CIPO
+3V3 OUT GPIO38/D11/COPI
GPIO46/B0 GPIO21/D10
GPIO1/D17/ADC1_CH0 GPIO18/D9
GPIO2/D18/ADC1_CH1 GPIO17/D8
GPIO3/D19/ADC1_CH3 GPIO10/D7
GPIO4/D20/ADC1_CH4 GPIO9/D6
GPIO11/D21/ADC2_CH0 GPIO8/D5
GPIO12/D22ADC2_CH1 GPIO7/D4
GPIO13/D23/ADC2_CH2 GPIO6/D3
GPIO14/D24ADC2_CH3 GPIO5/D2
VUSB OUT GND
GPIO0/B1 RESET
GND GPIO44/D0/RX0
VIN GPIO43/D1/TX0
Arduino Nano ESP32
+3V3
VUSB
P1
10k
S1
LED1 … LED36
R1 … R36
75R
LED1...LED36 = WD2812D-F8
PA1
PA2
PB1
PB2
PC1
PC2
PD1 PD2
PE1 PE2
GND1 GND2 GND3 GND4 GND5
+5V_1 +5V_2 +5V_3 +5V_4 +5V_5
VUSB
D1
S5J-E3/57T
+5V (4V3)
D2
S5J-E3/57T
C37 C38
47u 47u
6V3 6V3
K2
+5VDC
C1 … C36
100n
PA3 PA7 PA11 PA15 PA19 PA23 PA5 PA9 PA13 PA17 PA21
PA4 PA8 PA12 PA16 PA20 PA24 PA6 PA10 PA14 PA18 PA22
PB3 PB7 PB11 PB15 PB19 PB5 PB9 PB13 PB17
PB4 PB8 PB12 PB16 PB20 PB6 PB10 PB14 PB18
PC3 PC7 PC11 PC15 PC5 PC9 PC13
PC4 PC8 PC12 PC16 PC6 PC10 PC14
PD3 PD7 PD11 PD5 PD9
PD4 PD8 PD12 PD6 PD10
PE3 PE5
PE4 PE6
Circular Christmas Tree 2023
Corrected 2023-08-28
230508
v1.0

Figure 1: Schematic of the Circular Christmas Tree.

Jumper JP1, visible in **Figure 2** close to the Arduino Nano board, selects LED1's control signal (which is the first receiving device of the 36 daisy-chained LEDs). The use of an Arduino Nano ESP32 makes the tree more versatile. Through software, it is possible to change the tree's lighting patterns remotely using the onboard Wi-Fi and/or Bluetooth LE. All of the LED inputs have a 75 Ω resistor in series (R1…R36, according to the requirements indicated in the WS2812D-F8's datasheet [4]). Each LED is decoupled by a 100 nF capacitor (C1…C36). The power supply for the LEDs on the base PCB is decoupled additionally by 47 µF tantalum capacitors C37 and C38.

All of these components are SMDs and mounted on the bottom of the PCBs, so they're kept out of sight. The LEDs can be powered by a 5 VDC adapter, preferably rated at least 1.5 A, through DC power connector K2, or by the VBUS pin on the Arduino Nano board (if present) in the MOD1 PCB area. Diodes D1 and D2 (5 A, 50 VDC, S5J-E3/57T, SMD case, SMC size), also mounted on the bottom side of the PCB, prevent damage if the two power supplies are connected simultaneously. It's possible, anyhow, to connect a power supply to both MOD1 and K2, but it is not necessary. The module itself is powered by its USB-C connector, and the maximum current of the VBUS pin is sufficient (assuming that the AC adapter connected is powerful enough, of course).

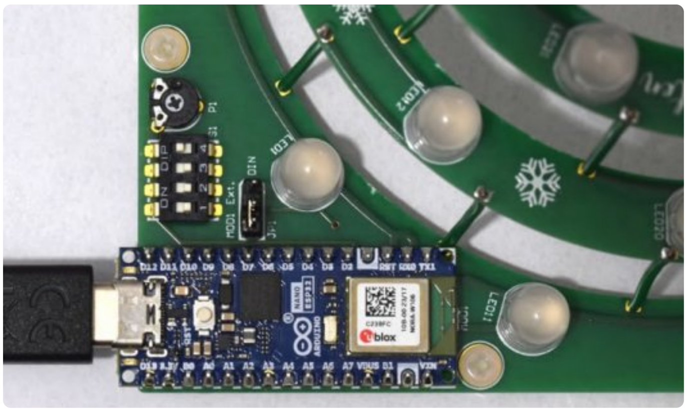


Figure 2: Detail of MOD1, S1, and P1 mounted on the base PCB.

The diodes are intentionally not Schottky type. Voltage drop across the diodes is higher and we must ensure that 3.3 V logic signals are well within the WS2812D-F8 specification. For this reason, if the LED chain is controlled by an external circuit, the best option is always to power this circuit through the internal power source of K1. The actual voltage on K1 will be lower than the 5 V of K2 — around 4.3 V (or even lower, at high current/brightness setting) — because of the voltage drop across D2.

To connect the PCBs, 39 wire segments connect the +5 V and ground from a PCB to the one above it (**Figure 3**). In the schematic, these are connections PA1–PA2 through PE5–PE6. One extra wire connects each LED D_{out} pin on a lower PCB to the first LED on the PCB above it.

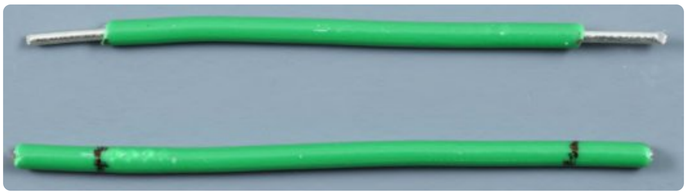


Figure 3: One wire marked and another one stripped. The remaining insulation should be 3 cm long.

If an onboard Arduino Nano ESP32 is used to control the LEDs, DIP switch S1 and small trimmer P1, both visible in Figure 2, can be mounted on the PCB as well. Software can read these components to select different patterns or adjust brightness. Without the onboard module, these components have no purpose.

Power Supply

The power supply for the LEDs is an AC-to-+5 V DC adapter with a barrel connector, connected to K2. The kit available in our shop only contains the parts for the PCB, including the small trimmer and the DIP switches. The AC adapter and optional Arduino Nano ESP32 module are not included and must be purchased separately.

This module has a USB-C connector that also powers the LEDs through its VBUS pin. As mentioned above, two diodes (D1, D2) prevent the two power supplies being connected directly to each other. VBUS is internally connected to the USB-C connector's 5 V through a Schottky diode of type PMEG6020AELR (Nexperia). This diode is rated at 2 A. Currently, the maximum current from the VBUS pin is not specified in its datasheet [5]. The current of the WS2812D-F8 is 12 mA, according to an overview of this family on the manufacturer's website.

However, the datasheet reports 15 mA in its specs, and many parameters are rated at the condition of $I_F = 20$ mA. In our prototype, the maximum current at maximum setting (3×255 decimal per LED) was just under 11 mA per color — far from 20 mA. These differences from the specifications are a bit confusing. Apparently, a more realistic value for the current is around 12 mA. Assuming this is the correct value, the maximum current at maximum LED brightness:

36 (LEDs) × 3 (colors) × 12 mA = 1.3 A.

In our Christmas Tree prototype, the total maximum current measured is 1.156 mA. The VBUS pin on MOD1 shouldn't have a problem with this current. But, in general, we strongly advise you never to use the maximum rated current of an LED, as it reduces the LED's lifespan significantly! The brightness between, for instance, one-third and maximum rated current is not that much lower. Consider taking this into account when writing the software. Setting all the LEDs to give white light, for very bright lighting in a well-lit room, will require an overall current of around 200 mA. This will be more than enough in most of the cases, and will allow you to utilize any standard USB type-A port to power the module.

PCB

As anticipated, the 136×136 mm PCB is a panel, with the base PCB holding the five circular-shaped PCBs with 24 breakoff bridges.

The figures in the **Component List** frame show the integer board, whilst **Figure 4** illustrates all the smaller components of it, after breaking the bridges. Please be aware that it takes some force to break them. A construction manual, available at Elektor Labs [6], gives all the details on how to build the tree. The copper of the 39 wire segments is 0.8 mm thick. These wires also help shape the tree, and that's why we have chosen ***green*** insulated wire. But, if you think a bare tin-plated wire is better looking, you can, of course, strip off all the insulation.

Closely observe that all PCBs are mounted in parallel, with a distance of 3 cm to the next board. The large number of wires makes the construction quite rigid, as you can see in **Figure 5**. That shows the completed prototype without the additional module. To keep the voltage drop from the bottom to the top minimal, all wires, except the signal wire, are connected to +5 V and ground alternately. When stripping all insulation, beware of short circuits between the bare wires!

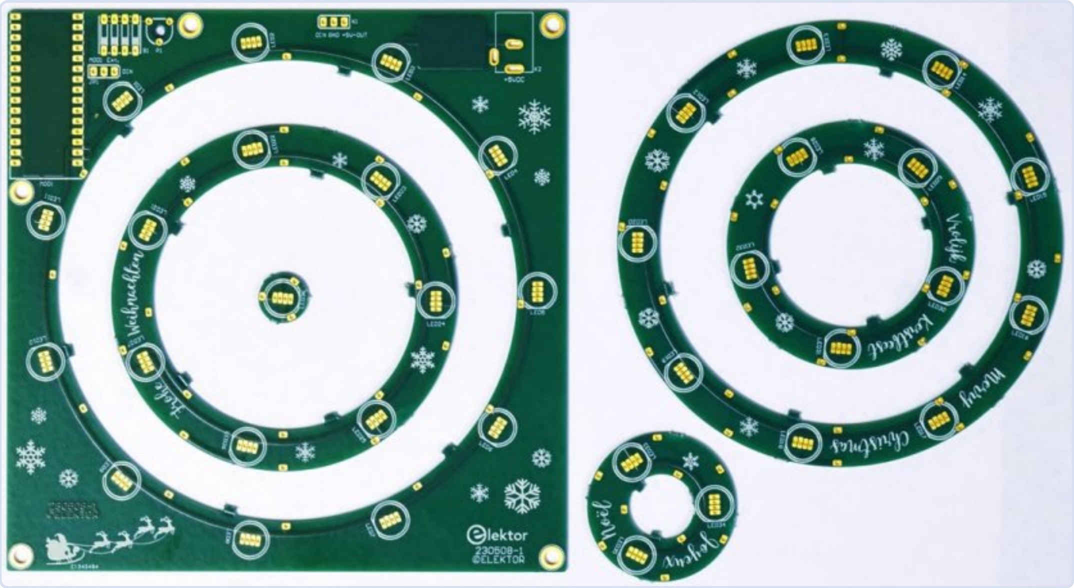
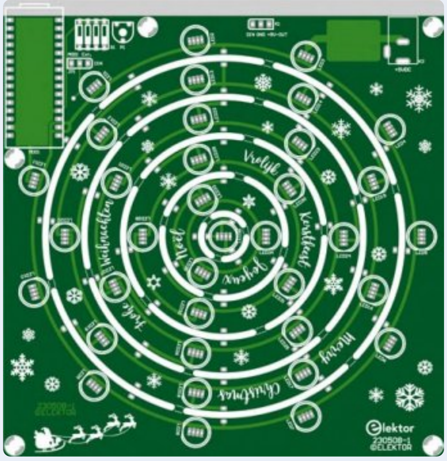
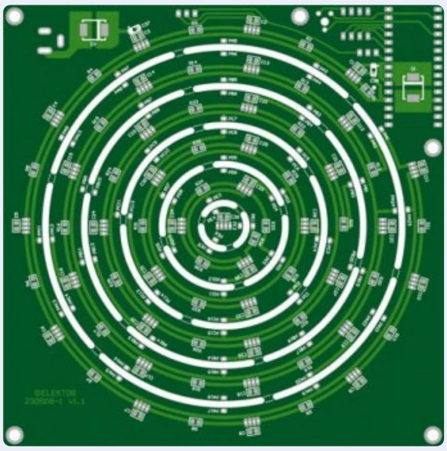


Figure 4: All PCBs separated. The bridging segments that connected the 6 PCBs can now be removed with a pair of wire cutters.

Component List





Resistors
R1...R36 = 75 Ω, 0W125, 5 %, SMD 0805
P1 = 10 kΩ, 0W1, 20 %, trimmer, top adjust, 6 mm round (Piher PT6KV-103A2020)

Capacitors
C1...C36 = 100 n, 50 V, 5 %, X7R, SMD 0805
C37, C38 = 47u, 6V3, 10 %, tantalum, case size A (1206)

Semiconductors
D1, D2 = S5J-E3/57T, SMD case size SMC
LED1-LED36 = WS2812D-F8, 8 mm, THT
MOD1 (optional) = Arduino Nano ESP32 with headers

Others
K1, JP1 = Pin header, 3x1, vertical, 2.54 mm pitch
Shunt jumper for JP1, 2.54 mm pitch
K2 = MJ-179PH (Multicomp Pro), DC power connector, 4 A, pin diam. 1.95 mm
S1 = DIP switch, 4-way
PA1...PE6 = 2 m wire, 0.81 mm solid, 0.52 mm2 / 20AWG, insulated green (Alpha Wire 3053/1 GR005)
H1...H5 = Nylon standoff, female-female, M3, 5 mm
H1...H5 = Nylon screw, M3, 5 mm

Misc.
PCB 230508-1 (136 × 136 mm)

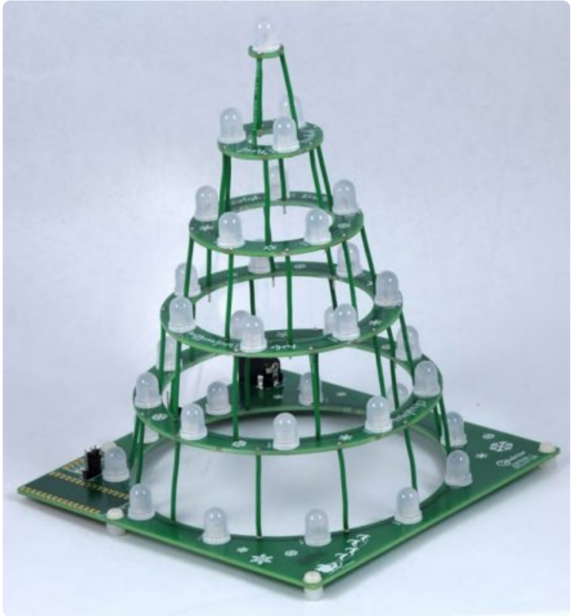


Figure 5: The completed prototype, without the optional Arduino module.

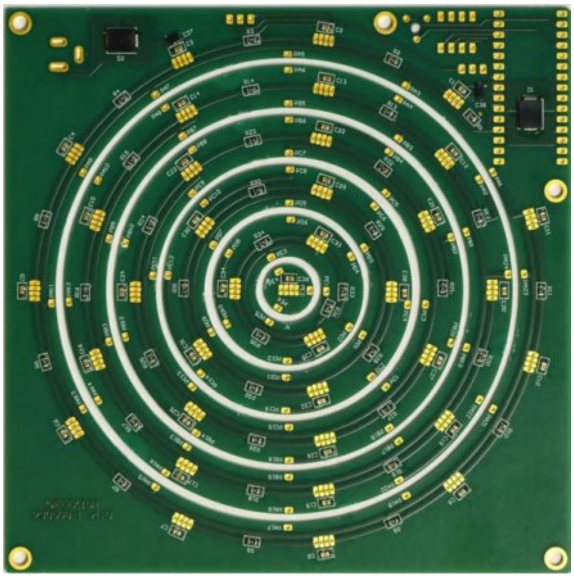


Figure 6: All SMDs are soldered on the back sides of the PCBs: R1...R36, C1...C38, and D1 and D2.

The tracks for the power supply are 1 mm wide. All SMD components are mounted on the bottom side of the PCBs, as visible in **Figure 6**. Make sure to separate the PCBs before starting to solder. No other components are on top of the circular-shaped PCBs, except for the LEDs, whose leads have to be cut as shown in **Figure 7**. Through-hole components JP1, K1, K2, P1, S1, and MOD1 are placed on top of the square base PCB. Since SMD components are used in this project, having some experience with their soldering is a recommended prerequisite.

Using a soldering iron with a fine tip and thin solder, the 0805 resistors and capacitors are not that difficult to solder. It is recommended to use very thin solder, 0.35 mm diameter, ensuring not too much tin will be used for the solder joints. The tree is placed on five

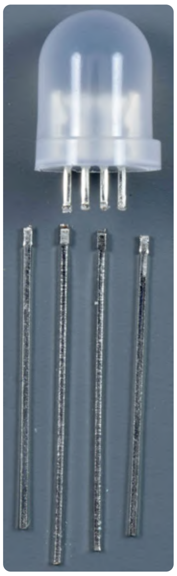


Figure 7: The leads of RGB LED WS2812D-F8 cut just above the widening.

5 mm white nylon standoffs and fastened with 5 white nylon screws. These are less noticeable. Optional module MOD1 is placed in a corner of the PCB to keep the base as small as possible. P1 and S1 are placed next to the module.

Optional Arduino Nano ESP32 Module

To make this Christmas Tree work independently of any external circuit, the Arduino Nano ESP32 module (MOD1) can be placed on the base PCB along with small trimmer P1 and DIP switches S1. Set jumper JP1 to position MOD1. The two extra components can be used to adjust and/or select brightness/speed and different patterns/modes.

Programming of the module can be done with the latest version of the Arduino IDE and the C programming language. Or, if you are looking for a MicroPython project, this Christmas Tree is also a good choice. A dedicated IDE called Arduino Lab for MicroPython [7] can be downloaded from GitHub. Also, a MicroPython installer [8] is necessary, and likewise available from GitHub. It's all well-documented on the Arduino website and gives beginners good resources to get it working [9].

Elektor developed basic software that can be downloaded from this project's Elektor Labs webpage [6], where you'll also find some details on its functionality.

During software development, you should be able to power the project from a PC's USB connector, but an AC-DC adapter with a UCB-C connector is needed to power the tree once done. The maximum current available depends, of course, on the specific port. Through a USB-C-to-USB-A adapter or adapter cable, the Arduino Nano can also be connected to a "legacy" USB port if the brightness is kept low. Current is limited to 500 mA then, something to keep in mind when testing the software! Or, connect a power supply to K2 as well and set the voltage a little higher than 5 V, so it will power the LEDs. **Figure 8** doesn't really show the LEDs' full, bright color, but still gives a good impression of what could be achieved through different settings.

230665-01

About the Author

Ton Giesberts started working at Elektuur (now Elektor) after his studies, when we were looking for someone with an affinity for audio. Over the years, he has worked mainly on audio projects. Analog design has always been his preference. Of course, projects in other fields of electronics are also part of the job. One of Ton's mottos is: "If you want to have it done better, do it yourself." For example, for a PCB design for an audio project with distortion figures on the order of 0.001%, a good layout is crucial!

Questions or Comments?

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

Related Products

- **Circular Christmas Tree Kit**
  www.elektor.com/20672
- **Arduino Nano ESP32 with Headers**
  www.elektor.com/20529

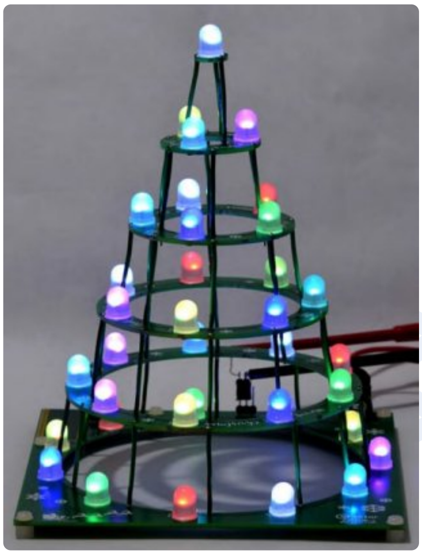


Figure 8: Test setup of the Christmas Tree.

WEB LINKS

[1] Christmas Tree V1: https://elektormagazine.com/labs/130478-xmas-tree-2014

[2] Christmas Tree V2: https://elektormagazine.com/labs/circular-christmas-tree-150453

[3] Documentation about the Arduino Nano ESP32: https://docs.arduino.cc/hardware/nano-esp32

[4] WS2812D-F8 Datasheet [PDF]: https://soldered.com/productdata/2021/03/Soldered_WS2812D-F8_datasheet.pdf

[5] Arduino Nano ESP32 Datasheet [PDF]: https://docs.arduino.cc/resources/datasheets/ABX00083-datasheet.pdf

[6] This project on Elektor Labs: https://elektormagazine.com/labs/circular-christmas-tree-2023-230508

[7] Arduino Lab for Windows [Zip]: https://tinyurl.com/arduinolab4win

[8] Arduino MicroPython Installer: https://github.com/arduino/lab-micropython-installer/releases/tag/v1.2.1

[9] MicroPython course MicroPython 101 by Arduino: https://docs.arduino.cc/micropython-course/

Borislav Milicevic (mki407)
How to make old devices smarter with a ESP8266?
ESP8266 Smart remote
How to make old devices smarter with a ESP8266?
4.7
Rate the project
TECH MASTER EVENT

A Simpler and More Convenient Life

An Amateur Project Based on the Espressif ESP8266 Module

Contributed by Transfer Multisort Elektronik Sp. z.o.o.

With each passing year, the concept of SmartHome is becoming increasingly popular, and the availability of solutions that help us manage our living space more efficiently is growing. In addition, some products that appear on the market offer compatibility with older devices, thanks to which we can use the existing equipment together with the latest technological advancements. Remote control of home appliances and the automation of various processes helps improve energy efficiency, protect the environment, increase our comfort and save money. The Smart ESP8266 remote project developed for a contest held by TechMasterEvent combines all these advantages.

Espressif is a manufacturer of well-received SoC integrated circuits and wireless transmission modules, many of which are available at TME. Thanks to their compact size and low-energy consumption, the products from Espressif can be successfully used both in consumer and industrial electronics.

Below, you can read about a device based on the ESP8266 module. It is an amateur project created by a participant of a TechMasterEvent contest. The entrants were asked to design an electronics project which seeks to make life easier.

You can find the ESP8266 module as well as several other components which may come in handy when you build your IoT projects (single-board computers, communication and memory modules, displays and much more) at [1].

ESP8266, IR LED and IR receiver

Smart ESP8266 remote is a project that aims to make controlling your home devices a breeze. With the use of an ESP8266, IR LED and IR receiver, this project eliminates the need for multiple remotes for different devices such as air conditioners or televisions. The project connects to a phone app, allowing users to easily send commands to their devices and even save the signals sent by their current remotes for future use.

In addition to its convenience and ease of use, the Smart ESP8266 remote is also a great solution for older devices that may not be compatible with traditional smart home technology. With the ability to read and save signals from traditional remotes, the Smart ESP8266 remote allows you to control older devices that may not have the capability to connect to the Internet or other smart home systems. This makes it a cost-effective alternative to upgrading your devices or purchasing expensive smart home devices.

The IR LED and IR receiver are used to transmit and receive IR signals respectively, which are used to control the household devices. The project can read and save signals from traditional remotes, allowing the user to control older devices that may not have the capability to connect to the internet or other smart home systems.

In addition to the hardware components, the Smart ESP8266 remote project also requires software to function, which you will find at [2].

The Smart ESP8266 remote offers several benefits such as convenience and ease of use, cost-effectiveness and flexibility. It eliminates the need to purchase expensive smart home devices or upgrading older devices, making it a cost-effective alternative. The project is also flexible enough to be easily modified or customized to work with different devices and different IR protocols, making it a versatile solution for controlling different devices with a single app.

230656-01

WEB LINKS

[1] TME shop: https://tme.eu
[2] Source code for this project: https://techmasterevent.com/project/how-to-make-old-devices-smarter-with-a-esp8266

How to Build IoT Apps without Software Expertise

With Blynk IoT Platform and Espressif Hardware

Contributed by Blynk Inc.

What if you could develop a mobile app without writing a line of code, brand it, and publish to app stores within a month? Launch production-grade IoT software without hiring software engineers? With Blynk IoT it's possible within a month, not years!

Blynk firmware library supports:
- ESP32
- ESP32-S2
- ESP32-S3
- ESP32-C3
- ESP8266
- and others

What is included in Blynk IoT?

Blynk is a low-code IoT software platform featuring cloud, firmware libraries, no-code native mobile app builder, and a web console to manage it all. You get Wi-Fi device provisioning, data visualization, automations, notifications, OTA updates, and a robust user and device management system [1].

Blynk App Builder for iOS and Android

Enables the creation of quick prototypes and fully functional standalone apps without coding skills. In constructor mode, you can choose from 50+ customizable UI elements like buttons, sliders, charts, maps, gauges, etc., and drag and drop them to the canvas to create a custom UI for your connected product. You can set up multiple app pages, use various interactions, customize images, fonts, colors, and icons to make your app unique.

Web Dashboard Builder

It has similar architecture for creating historical and real-time data visualizations, and for controlling and monitoring devices using pre-built UI elements. The cool thing is you can build independent interfaces for mobile and web, based on your needs.

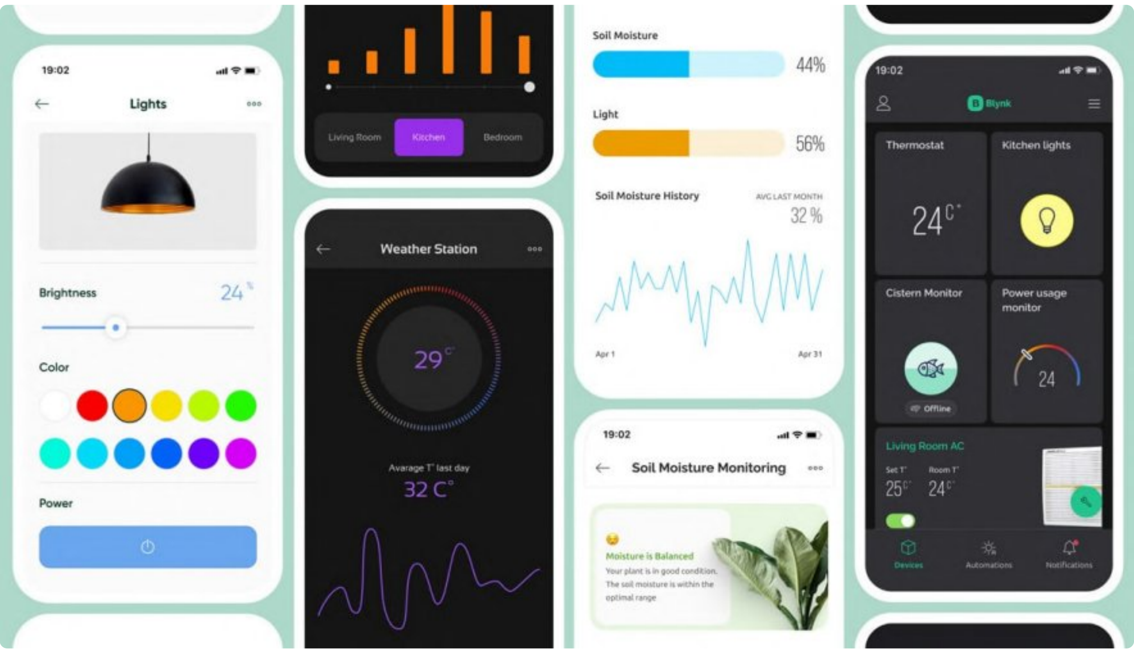


Figure 1: No-Code Interfaces created with Blynk.

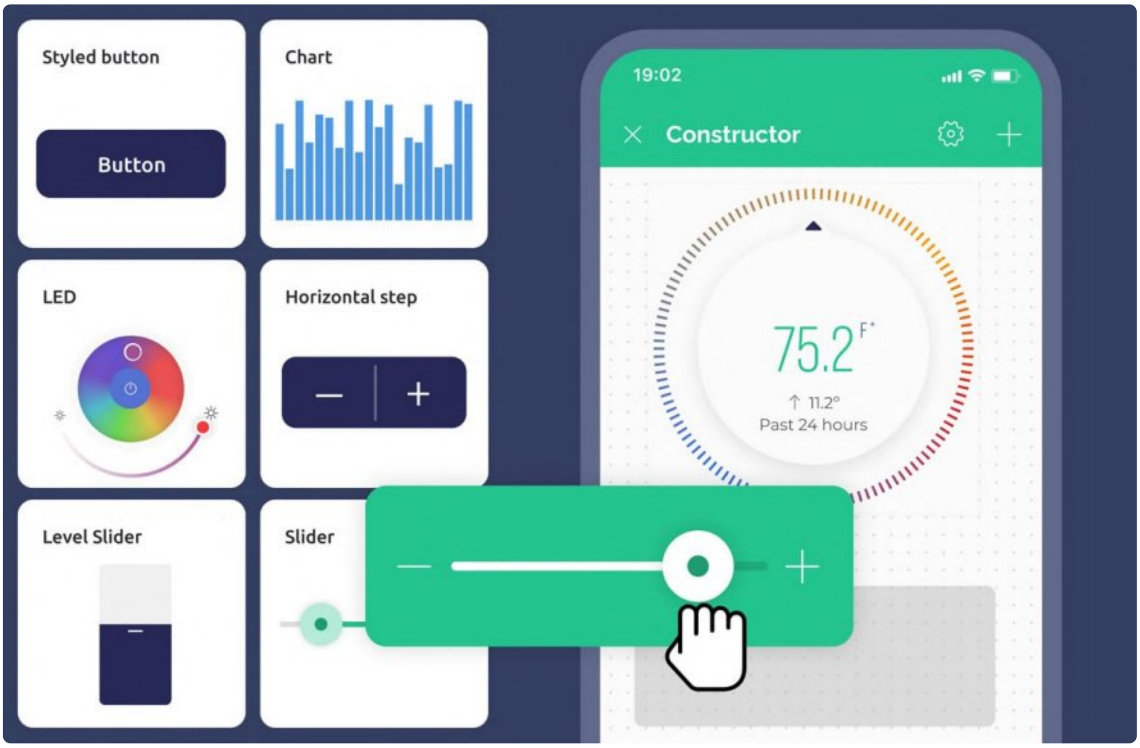


Figure 2: Blynk Drag-n-Drop App Builder.

Advanced User Management System

It helps keep everything structured, even at enterprise scale. You can create a multi-level organization structure and manage devices and user's roles, permissions, passwords, and much more.

Built-in Device Lifecycle Management

The functionality covers all needs related to token management, Wi-Fi provisioning with dynamic token generation, adding devices, and assigning them to users. It offers reliable and secure OTA firmware updates managed in a simple interface.

No-Code Automation Scenarios

Can be set based on date, time of the day, user actions, or device state. You can notify users about important events on the hardware via pushes, in-apps, emails, or SMS.

How to Connect Your ESP to Blynk? What's the Integration Effort?

Depending on your hardware setup, you can go one of the two routes for connecting your Espressif device. Both enable all the Blynk IoT features out of the box, including Wi-Fi provisioning, OTA, and secure connection to Blynk Cloud. Choose *Blynk.Edgent* [3] for single-MCU devices. If you're offloading connectivity to a secondary MCU, go with *Blynk.NCP* [4][5].

Both routes require minimal implementation effort, with code examples provided by Blynk. For the dual-MCU setup, it's a ready binary for the NCP and a lightweight library for the primary MCU communicating with the Network Co-Processor over the UART interface.

Your journey from device setup to full-scale IoT infrastructure and app launch can take just weeks [6].

230659-01

WEB LINKS

[1] Official website: https://bit.ly/blynk-iot
[2] Blynk.Console — create your free account: https://bit.ly/blynk-cloud
[3] Blynk.Edgent documentation: https://bit.ly/doc-edgent
[4] Blynk.NCP documentation: https://bit.ly/doc-ncp
[5] What is Blynk.NCP: https://bit.ly/blynk-ncp
[6] Ready-made weather station project to play around: https://bit.ly/weather-blueprint

Building a Smart User Interface on ESP32

Contributed by Slint

Smartphones have redefined the user experience of touch-based user interfaces (UIs). Building a modern smart UI necessitates the use of modern graphical libraries and tools. In this article, we'll share tips and showcase Slint, a toolkit for creating interactive UIs that meet and exceed user expectations.

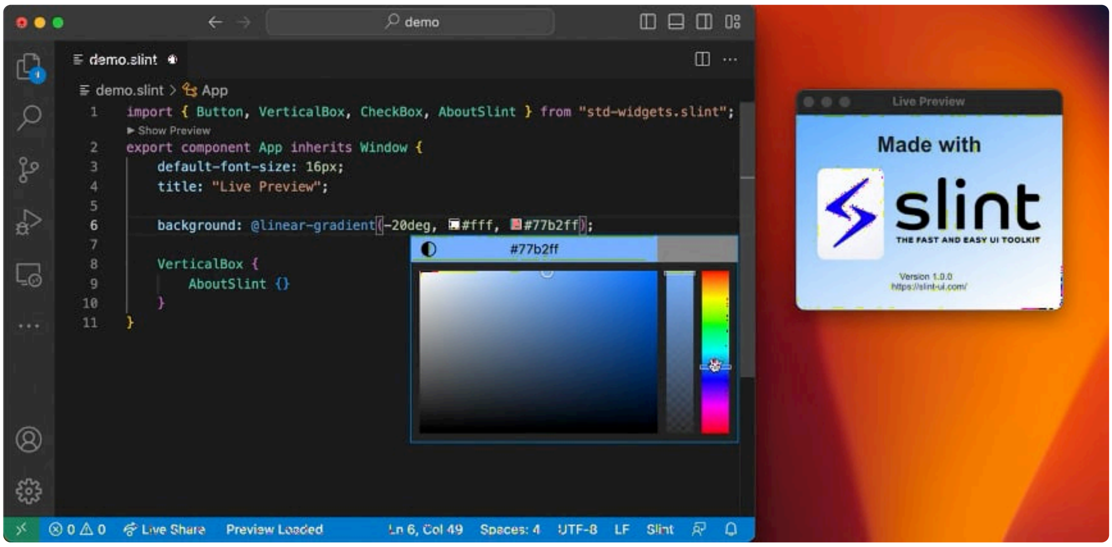
Figure 1: C++ and Rust logos.

Slint is a next-generation toolkit for building native graphical UIs in C++, Rust, and JavaScript, with a broad cross-platform support, including bare metal, RTOSs, and embedded Linux. On GitHub, Slint has more than 10.000 stars.

Choose a Programming Language — C/C++ or Rust

In embedded programming, C/C++ have been the favorite programming languages for a long time. But Rust, known for its memory safety and performance, is becoming popular among embedded developers.

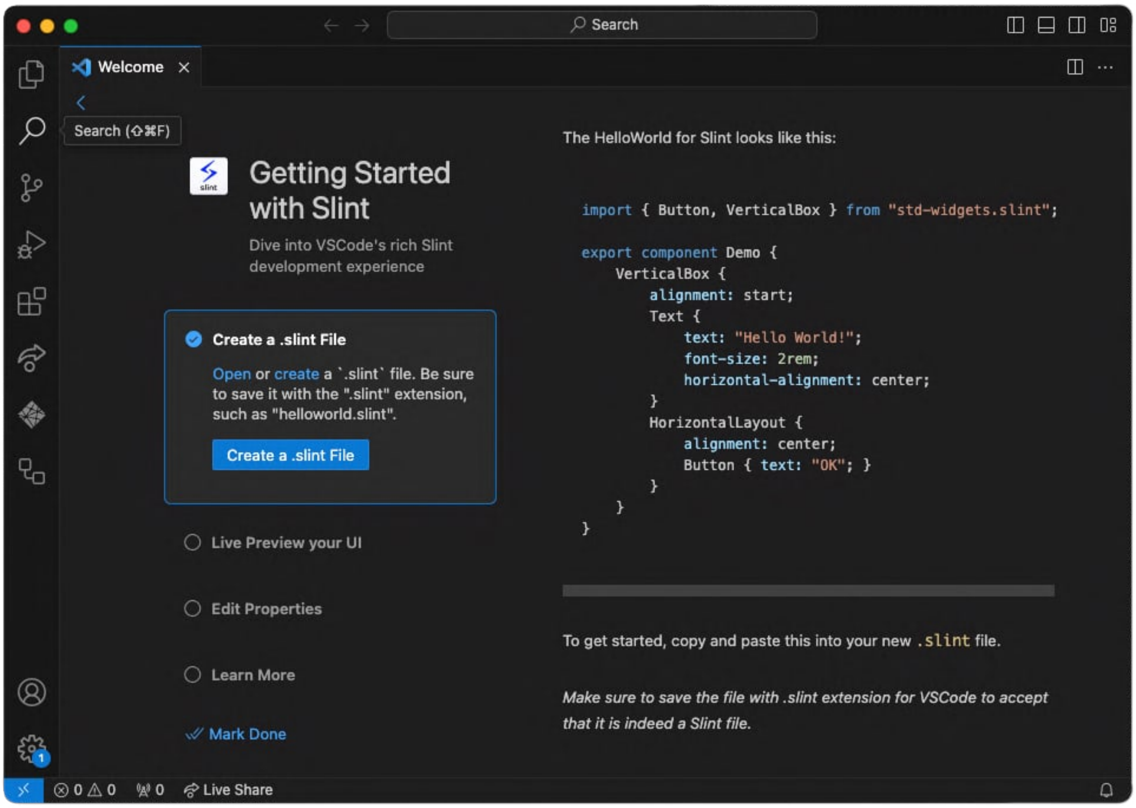
Slint, the only toolkit to provide native APIs for both C++ and Rust (**Figure 1**), offers developers the choice: Write your business logic in either language. Furthermore, it provides a transition path for those interested in moving from their code from C/C++ to Rust.



```
import { Button, VerticalBox, CheckBox, AboutSlint } from "std-widgets.slint";
export component App inherits Window {
    default-font-size: 16px;
    title: "Live Preview";

    background: @linear-gradient(-20deg, #fff, #77b2ff);

    VerticalBox {
        AboutSlint {}
    }
}
```

Figure 2: Quick iterations with Slint's Live-Preview.

slint
GUI toolkit



Getting Started with Slint

Dive into VSCode's rich Slint development experience

Create a .slint File

Open or create a `.slint` file. Be sure to save it with the ".slint" extension, such as "helloworld.slint".

Create a .slint File

○ Live Preview your UI

○ Edit Properties

○ Learn More

Mark Done

The HelloWorld for Slint looks like this:

```
import { Button, VerticalBox } from "std-widgets.slint";

export component Demo {
    VerticalBox {
        alignment: start;
        Text {
            text: "Hello World!";
            font-size: 2rem;
            horizontal-alignment: center;
        }
        HorizontalLayout {
            alignment: center;
            Button { text: "OK"; }
        }
    }
}
```

To get started, copy and paste this into your new .slint file.

Make sure to save the file with .slint extension for VSCode to accept that it is indeed a Slint file.

Figure 3: Getting started with Slint.

Separate UI from Business Logic

Common patterns like MVC or MVVM promote separating business logic from the UI to enhance efficiency and code quality.

In Slint, the UI is defined using a language akin to HTML/CSS, promoting a strict division between presentation and business logic. Complete your UI design through quick iterations with Slint's Live-Preview (**Figure 2**).

Enjoy a Good Developer Experience (DX)

Today's complexity in software development requires a good DX: Developers build with confidence, drive greater impact, and feel satisfied.

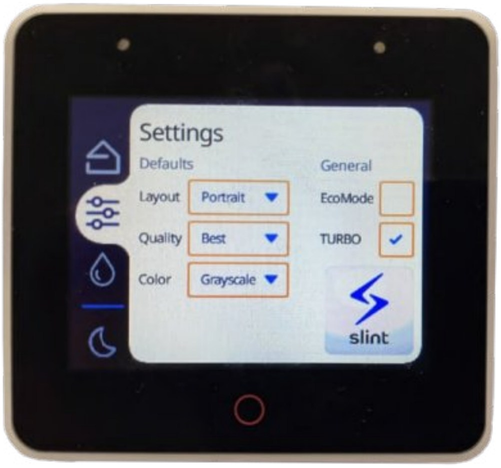
You can keep using your favorite IDE. Choose between Slint's VS code extension (**Figure 3**) and the generic language server: Enjoy code completion, syntax highlighting, diagnostics, live-preview, and more. Additionally, Slint offers an ESP-IDF component, simplifying its integration with the Espressif IoT Development Framework (IDF).

Deliver an Exceptional User Experience (UX)

UI performance is critical for an exceptional UX. Enjoy flexibility in hardware design with Slint's line-by-line and framebuffer rendering capabilities on the ESP32 platform, ensuring a more versatile approach to device development (**Figure 4**).

To get started with Slint on ESP32, visit [1].

230670-01



Settings

Defaults

Layout: Portrait

Quality: Best

Color: Grayscale

General

EcoMode □

TURBO ☑

Figure 4: A Slint demo on ESP32.

WEB LINK

[1] Slint on ESP32: https://slint.dev/esp32

Quick & Easy IoT Development with M5Stack

Contributed by M5Stack

As a world-renowned modular IoT development platform based on ESP32, M5Stack builds hundreds of controllers, sensors, actuators and communication modules in modularized style that can be connected via standard interfaces. By stacking modules with different functionalities, users can accelerate product verification and development.

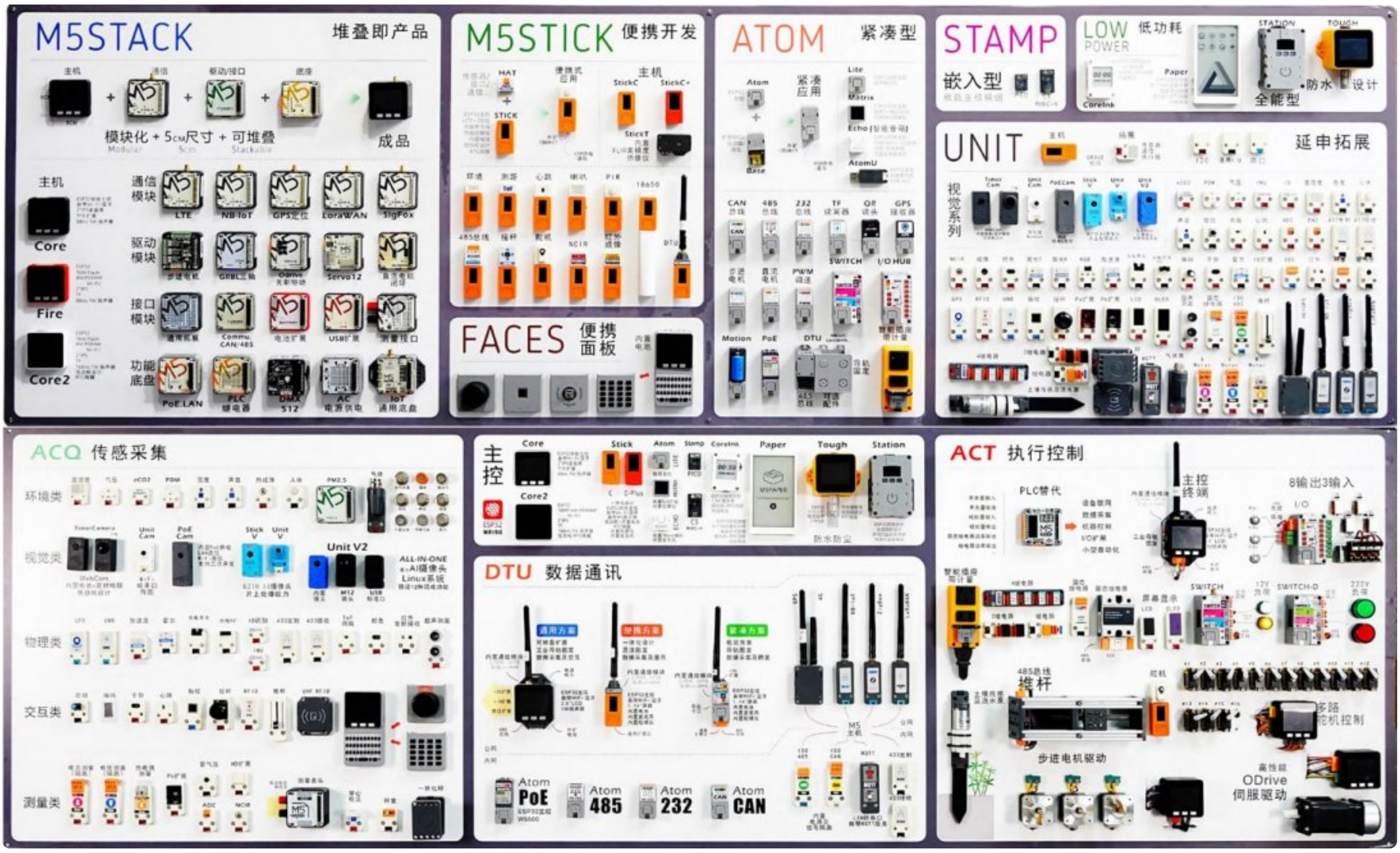


Figure 1: M5Stack Eco-system Family.

Figure 2: M5Dial is suitable for Smart Home.

> *If you are a fan of ESP32, then M5Stack is a must-have!*

The M5Stack modules (**Figure 1**) [1] can be plugged and played with the UIFlow low-code graphical programming IDE to provide the best experience for prototyping IoT projects, from entry-level hobbyists to professional developers.

With Stackable hardware modules and a user-friendly graphical programming platform, M5Stack provides clients in Industrial IoT, Home Automation, Smart Retail, Smart Agriculture and STEM education, with efficient and reliable Quick & Easy IoT Development experience.

New: The M5Dial

The recently launched M5Dial [2] is a highly suitable product for Smart Home. As a versatile embedded development board, M5Dial integrates various functionalities and sensors required for Smart Home control (**Figure 2**).

The main controller of M5Dial is M5StampS3, a microcontroller based on the ESP32-S3 chip, known for its high performance and low-power consumption. It supports Wi-Fi and Bluetooth communication, as well as multiple peripheral interfaces such as SPI, I2C, UART, ADC, and more. M5StampS3 also comes with 8 MB of built-in flash, providing sufficient storage space for users.

M5Dial features a 1.28-inch circular TFT touch screen, a rotary encoder, an RFID detection module, an RTC circuit, a buzzer, physical buttons, and other functionalities, enabling users to easily implement various projects.

The standout feature of M5Dial is its rotary encoder, which accurately records the position and direction of the knob, providing users with an enhanced interactive experience. Users can adjust settings such as volume, brightness, menus, or control home appliances like lights, air conditioning, curtains, etc., using the rotary knob. The built-in display screen of the device can also show different interactive colors and effects. With its compact size of 45 mm × 45 mm × 32.2 mm and light weight of 46.6 g, M5Dial is easy to implement.

Whether it's used to control household appliances in Smart Home or to monitor and control systems in industrial automation, M5Dial can be easily integrated to provide smart control and interactive functionalities.

230662-01

WEB LINKS

[1] The Innovator of Modular IoT Development Platform | M5Stack: https://m5stack.com/
[2] ESP32-S3 Smart Rotary Knob w/ 1.28" Round Touch Screen: https://shop.m5stack.com/products/m5stack-dial-esp32-s3-smart-rotary-knob-w-1-28-round-touch-screen

Prototyping an ESP32-Based Energy Meter

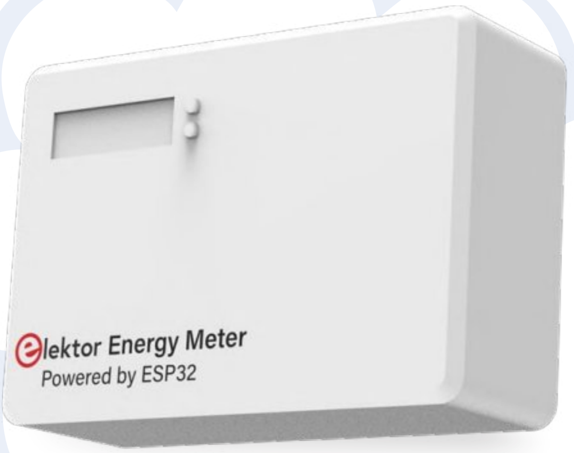


Figure 1: Test rendering of how our Single-Phase Energy Meter might look.

By Saad Imtiaz (Elektor)

This article presents the journey of developing an energy meter using an Espressif ESP32, emphasizing real-time power consumption monitoring and safety. It highlights the initial steps, requirements, and considerations that take place when embarking on an embedded project. As the project progresses, future achievements will be shared in upcoming editions of *Elektor Mag*.

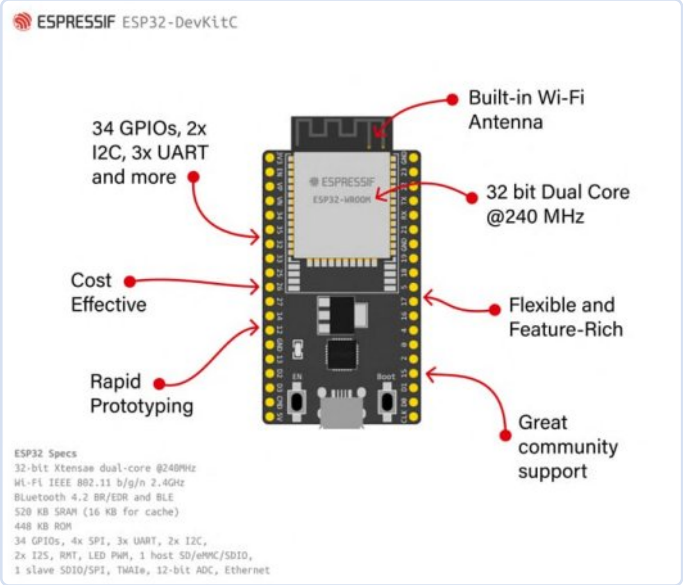
In the field of engineering, combining the right technologies can lead to significant advancements. This project aims to develop an energy meter using the Espressif ESP32 microcontroller and Microchip's ATM90E32AS energy metering IC. In this article, the beginning of this project's journey is briefly shared, from component selection to prototyping. The goal is straightforward: to create a reliable system for accurate energy measurement from your home or workshop's main circuit box. This meter will enable users to track their power consumption in real time, offering insights that can lead to more efficient energy use.

Design and Requirements

The project has clear goals and design requirements: real-time monitor single-phase power using three current transformers (CTs), keep it affordable, and make it user-friendly. The choice of ESP32 and ATM90E32AS IC components was guided by these aims, offering both cost-effectiveness and reliable performance. Another target was to keep the size smaller than 100×80×30 mm (L×W×H) to ensure that it can be accommodated in a circuit breaker box. To enhance the user experience, a mobile interface is also included for remote monitoring, as well as an OLED display with buttons for direct interaction. The design also allows for future software updates, ensuring long-term utility for the consumer. In **Figure 1**, the rendering of the current prototype enclosure is shown.

Microcontroller Selection

The choice of the ESP32 microcontroller was predicated on a detailed analysis of its capabilities. The chip excels in several areas crucial to the success of this project. First, its ease of integration into varied circuit designs provides flexibility during the engineering phase. Second, its cost-effectiveness makes it an attractive choice for a prototype that aims to balance performance and budget. Third, the compatibility with a wide range of sensors and ICs offers significant advantages. Lastly, the extensive community support for ESP32 chip augments its suitability for this project. **Figure 2** highlights the ESP32-D0WD-V3's



ESP32 Specs
32-bit Xtensae dual-core @240MHz
Wi-Fi IEEE 802.11 b/g/n 2.4GHz
BLuetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C,
2x I2S, RMT, LED PWM, 1 host SD/eMMC/SDIO,
1 slave SDIO/SPI, TWAIe, 12-bit ADC, Ethernet

Figure 2: Main features and advantages of the ESP32.

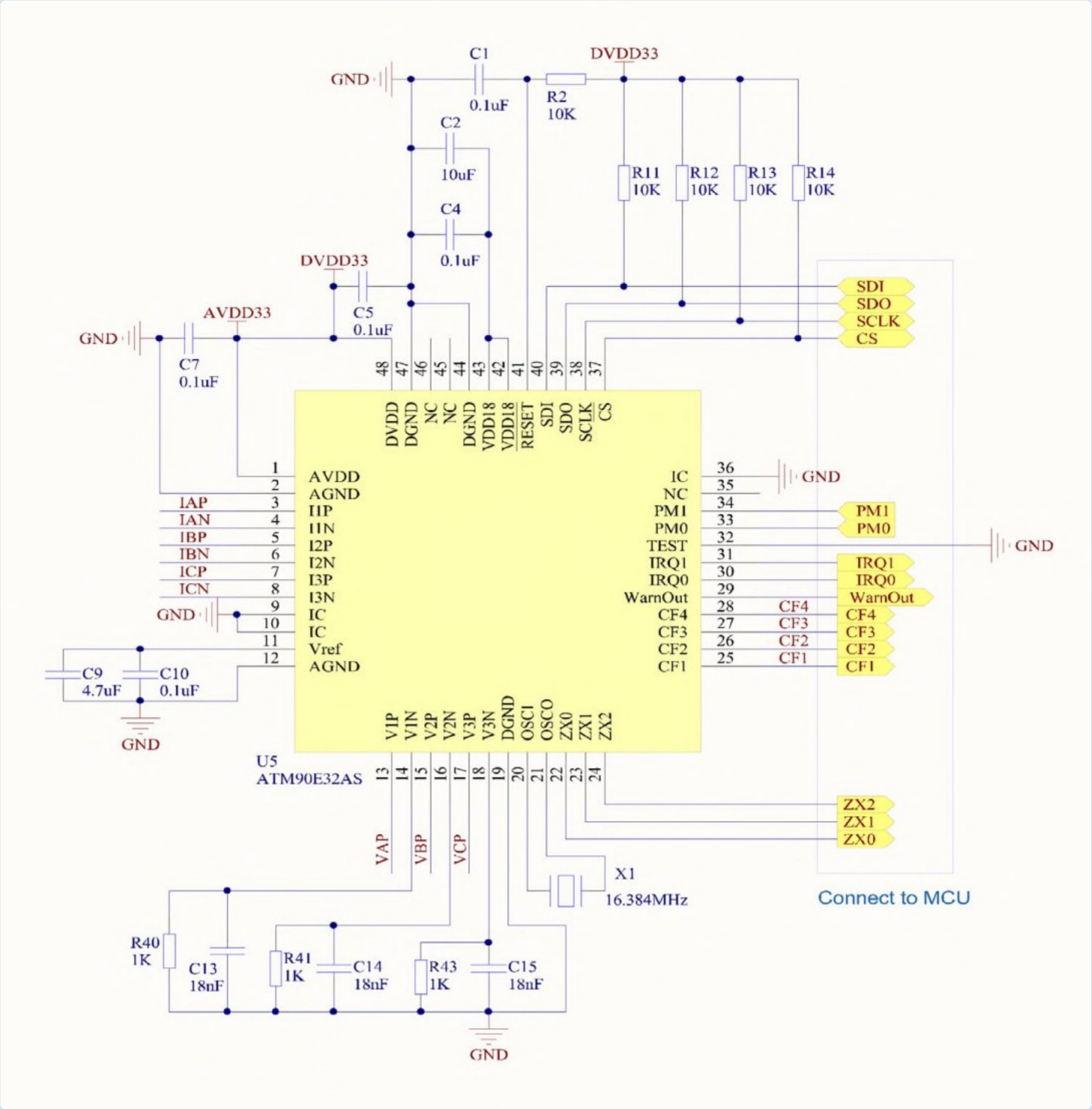


Figure 3: The energy meter is based on an application note by Atmel [2]. Here, you can see the circuitry around the metering IC.

main features and advantages resulting in its being selected for this project.

Metering IC Integration

The ATM90E32AS IC from Microchip was integrated according to the manufacturer's application note; the document served as a cornerstone in ensuring that the energy metering IC communicated seamlessly with the ESP32 microcontroller. However, this phase was not devoid of challenges. The procurement of the correct components within budget constraints required meticulous planning, given the constraints on availability. In **Figure 3**, the application note provided by Atmel (now Microchip) in shown.

Design Phase and Electrical Safety Standards

The design phase is indeed a pivotal part of the engineering process, particularly when safety is an indispensable consideration. In a device designed to interact with mains AC voltages, meticulous attention must be paid to conformance with established safety standards. In **Figure 4**, the project's block diagram is shown.

To ensure safety, several specialized electrical components were integrated into the design. Metal oxide varistors (MOVs) were used for transient voltage suppression to protect the circuitry from voltage spikes. Furthermore, fuse components were included as an essential failsafe to prevent overcurrent conditions.

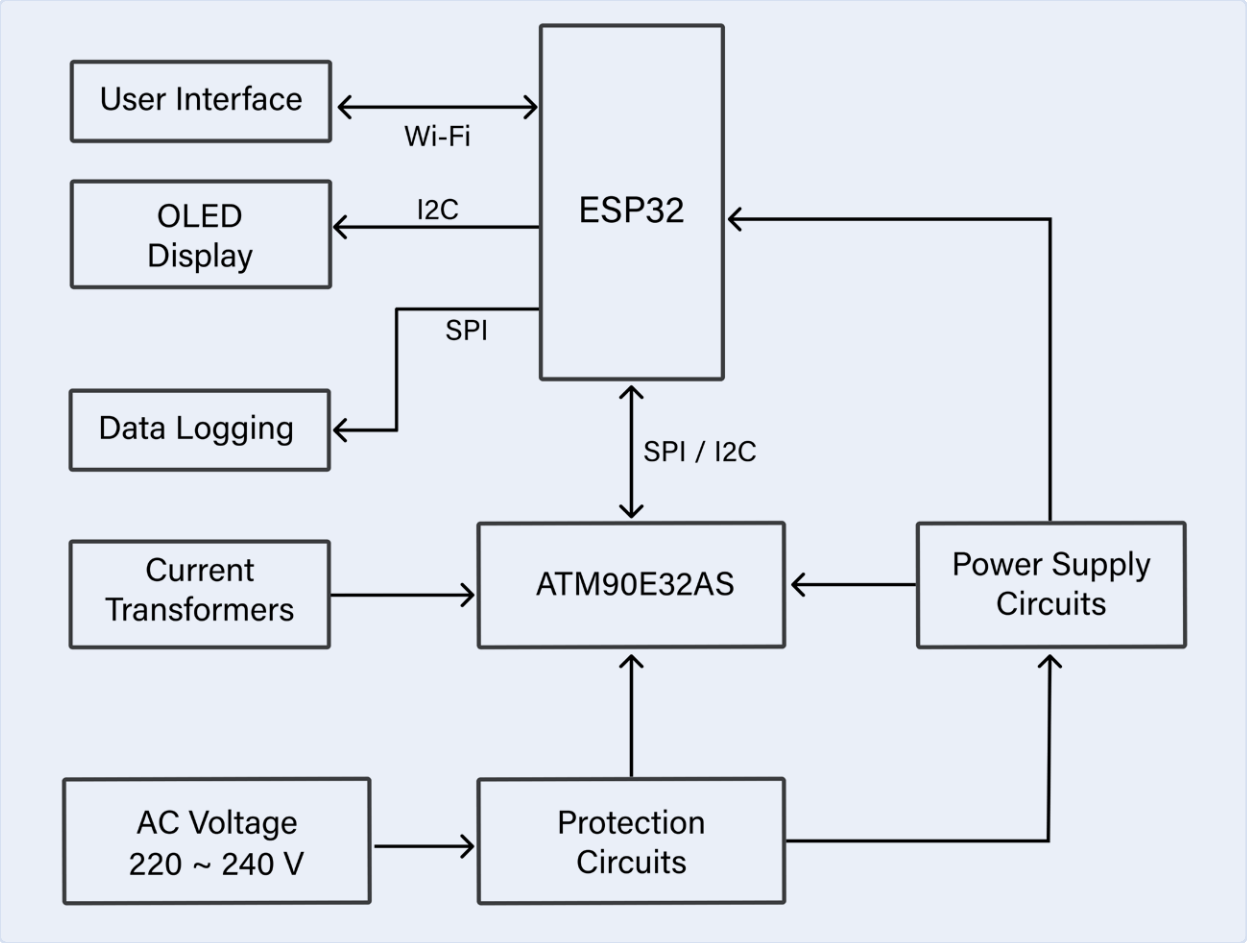


Figure 4: Block diagram of our Energy Meter project.

Beyond component selection, circuit design also focused on layout considerations that would abide by safety norms. Adequate creepage and clearance distances were maintained between the conductive elements on the PCB to prevent electrical arcing. Trace widths for AC voltage lines were calculated carefully to handle the current ratings, adhering to IPC-2221 standards [1]. This was critical in ensuring the thermal performance of the board under full-load conditions. To ensure ground integrity, a solid ground plane was used. Special attention was given to the design of differential pairs for signal integrity, making sure that the routing followed precise geometry to minimize electromagnetic interference.

Manufacturing Selection: Opting for JLC PCB

After scrutinizing various PCB assembly services, JLC PCB was selected. The principal reason for this choice was the balance of cost-effectiveness and reliability that they offer. This decision was important in keeping the project within budget without compromising on the quality of the assembled board. Currently, the prototype schematic and PCB designs are being finalized, and they will soon be sent for production.

Reflecting on the Journey and Looking Forward

In retrospect, this project shows what can be achieved when careful planning meets good engineering. The hurdles we faced helped us improve our design. As we move from making a prototype to possibly mass-producing it, we expect it to make a real difference in how people manage energy. This project will be detailed in upcoming editions of this magazine — we're still in the process of getting the prototype made, tested, and working, on the software that will run it. There's more to come, so stay tuned for updates on this project. We will near completion and share updates in the January/February 2024 edition of Elektor, which is dedicated to the topic of *Power & Energy*.

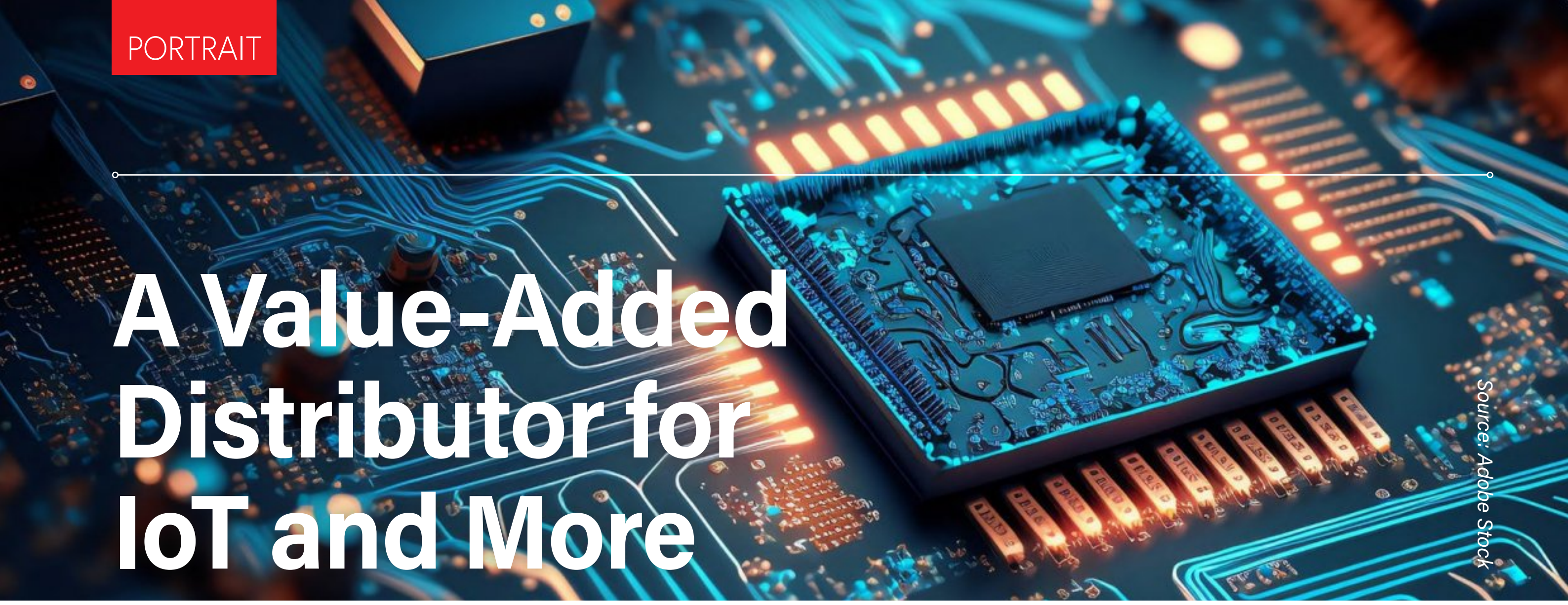
230646-01

Questions or Comments?

If you have questions about this article, feel free to email the Elektor editorial team at editor@elektor.com.

Related Products

- **ESP32-DevKitC-32E**
  www.elektor.com/20518
- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324



A Value-Added Distributor for IoT and More

Source: Adobe Stock

Contributed by Steliau Technology

Steliau Technology is an innovative company specializing in electronic solutions. The company stands out for its engineering expertise and passion for innovation. Steliau Technology, through its partner Espressif, provides essential electronic components for wireless connectivity and IoT, such as the ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6 and many more products.

Founded in 2018, Steliau Technology [1] defines itself as a value-added distributor of electronic solutions. Human-Machine Interfaces, screens and touch solutions, connectivity & IoT are all areas of expertise largely mastered by Steliau and which give it an already well-established reputation in the electronics market.

Steliau Technology is well-known for its strategic partnerships with leading electronics companies, enabling it to strengthen its position in the fields of IoT and connectivity. Espressif and Steliau Technology share a long-standing partnership, as the first distributor, consolidated over the years. As the official distributor for France and Italy, Steliau Technology is the one-stop shop for Espressif solutions.

Steliau Technology is perfectly equipped to offer full support for the entire range of Espressif products, both in terms of hardware and embedded software. The team has in-depth technical expertise covering all aspects of connectivity, from hardware design to software programming. This means that Steliau Technology is able to provide full support to customers, ensuring robust and high-performance connectivity solutions.

This strong partnership guarantees our customers privileged access to the best connectivity solutions on the market, such as the latest Espressif generations: ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6.

Steliau Technology through its partner Espressif provides essential electronic components for wireless connectivity and IoT in a variety of markets and sectors, contributing to the constant evolution of the technology.

IoT Solutions and More

First of all, in the field of the Internet of Things (IoT), Espressif's Wi-Fi and Bluetooth microcontrollers are widely used. These components are vital for smart home applications such as connected thermostats, security cameras and lighting control devices. They play a leading role in interoperability in connected home products with compatible Matter solutions (through WiFi and Thread in particular). Espressif's solutions are also used in the industrial sector for remote monitoring, data collection, and machine control. In addition, Espressif's products are present in the healthcare sector, where they power connected medical devices and fitness tracking devices.

As a global electronics partner, Steliau has the ability to offer solutions integrating Espressif's products for touchscreen control. Thanks to its expertise, Steliau Technology is able to design integrated solutions in which Espressif's products control the touch screen, with a number of success stories to our credit, particularly for screen sizes up to 7 inches. Our ability to offer a global solution is reinforced by specific technical support covering all these areas. ⏮

230661-01

Any requests?

Steliau is available to assist customers with any queries they may have. Please contact remi.krief@steliau-technology.com for any request about Espressif solutions.

WEB LINK

[1] Steliau Technology: https://steliau-technology.com/en

In-Depth Insights: Interview With Arduino on the Nano ESP32

Alessandro Ranellucci and Martino Facchin Discuss Espressif Collaboration

Questions by Saad Imtiaz and Clemens Valens (Elektor)

In summer 2023, Arduino launched the Nano ESP32. Based on the ESP32-S3 from Espressif, the new board features 2.4 GHz, 802.11 b/g/n Wi-Fi, and long-range Bluetooth 5 (LE) connectivity in a Nano form factor. The Nano ESP32 is not the first Arduino board sporting a processor from Espressif, but this time it's the main act instead of just a wireless communication module supporting another MCU. Elektor interviewed Alessandro Ranellucci (Head, Arduino Makers Division) and Martino Facchin (Hardware & Firmware Manager, Arduino) about this collaboration between Espressif and Arduino.

Elektor: Can you explain why you chose the ESP32 for the new Arduino Nano series instead of another microcontroller?

Alessandro Ranellucci: The Arduino Nano series represents a group of boards designed to be consistent with each other in terms of shape, pin layout, and core technical features. This consistency is something our community of makers appreciates because it allows for seamless interchangeability within the Nano series. However, we hadn't introduced a board based on the popular ESP32 architecture yet. We felt it was necessary to provide a powerful option within the Nano series that utilized the ESP32, hence the decision.

Martino Facchin: Yes, in the last four years or so, we've been growing the Nano series by including microcontrollers from various silicon vendors that we hadn't worked with before. We began by integrating microcontrollers from Nordic, followed by Raspberry Pi. Essentially, it's our testing ground to try out both new and widespread architectures, while adding unique value that isn't available elsewhere. For example, on the ESP32 bare chip, there was no USB. In the C3, there is USB, but it has a fixed vendor and product ID, so you couldn't really distinguish between individual boards. The S3 is the first one that has this capability, so we used it, rather than anything prior.

Elektor: What unique technical advantages does the ESP32 have over other options?

Martino Facchin: Yes, indeed, we had the opportunity to collaborate directly with u-blox to acquire a specialized chip that isn't available for general purchase, which includes PSRAM integrated within the chip itself. We're offering a chip that comes with PSRAM and the largest possible amount of external flash memory. These details are the technical deep dive, you could say, tailored for the enthusiasts and geeks. The board's features push the ESP32 to its highest potential currently available. Then, naturally, it includes Wi-Fi, BLE, dual-core processing, and all the other functionalities that come with the ESP32. We chose not to include additional features on the Arduino Nano ESP32 like we have with other boards. This one is designed as a building block. Unlike the Arduino Nano 33 BLE Sense, it isn't a device you can use right out of the box for

Martino Facchin

Alessandro Ranellucci

substantial applications. You need to attach additional components, such as sensors or modules, to the board. It comes equipped with an RGB LED, but, beyond that, the maker has the freedom to add to it with other components, using it as a well-calibrated base for their projects.

Elektor: Did you face any difficulties fitting the ESP32 chip into Arduino Nano form factor, and if so, how did you address these issues?

Alessandro Ranellucci: Indeed, the initial challenge involved forming a partnership with Espressif. They've been crafting an excellent Arduino core for the ESP32 boards for many years, one that's deeply embedded within the Arduino framework, yet it also adhered to their distinct technical preferences. The challenge was figuring out how to combine our efforts to create a more cohesive and improved user experience. So, we established this collaboration. I think Martino can also mention other challenges, such as, for example, the pin numbering.

Martino Facchin: Definitely not much, from a hardware standpoint, there were no issues. Of course, that is easy compared to other boards we've madein the last couple of years. From a software standpoint, that's a very different matter because pin numbering was very tied to the ESP world. They numbered pins exactly as they are on the CPU socket. On the underside of the board, you might see labels from other manufacturers with numbers like 31, then next to it a 4, and then a 55, and so on. That doesn't work out for the Nano — not the solution we were looking for. So, we developed a way to translate logical pin numbers into internal pin numbers. We had this accepted into the ESP32 core community core. Right now, every board with an ESP32, from any other manufacturer, can use the same logic if they want to adopt our philosophy, and this is a direct contribution to one core that we're not maintaining. It was tough because the release time of this feature had to match the release time of the board perfectly. However, it succeeded, eventually.

It was a challenge because it was the first time we released something not fully controlled by us, and that was difficult, but we did it.

Alessandro Ranellucci: We gained significant insights from the whole iteration process because, as Martino mentioned, any manufacturer can now opt to use logical PINs. It's more user-friendly to number pins sequentially than to use the controller's pin numbering, such as PA1 and PD1.

Elektor: Did the ESP32 software development, ecosystem, and community support affect your decision to use it in the new Arduino Nano?

Martino Facchin: No. We would have chosen to use it even if there was no community support. Certainly, with the community doing extensive work and our projects flourishing, we benefit greatly from their contributions, especially to the library manager. Some libraries were already compatible with our board, which was beneficial. Others were exclusive to the ESP32, which was a drawback, but now we can utilize those as well. This compatibility was a factor in our decision, but we would have proceeded with the ESP32 regardless.

> *Pin numbering was a challenge because the numbers were very tied to the ESP32 world. But we found a solution – logical pin numbers.*
>
> Martino Facchin

Alessandro Ranellucci: On the software front, we had the option to create a new core, just as we did with other products, such as the RP2040, where we developed our own to have full control over the software. But Espressif has done excellent work over the years, and they have a robust community. That's why we chose to collaborate with them — a decision influenced by the strength of the ecosystem.

We aim to remain neutral regarding technology, avoiding exclusive commitment to just one manufacturer's line of microprocessors. Our goal is to offer a versatile and interoperable Arduino platform, as that is how users perceive and expect Arduino to be. Thus, we're continuously experimenting with and researching new products to develop.

Martino Facchin: Individuals who switch to Arduino from other backgrounds, such as BSP or integrated environments that involve time-consuming setup and configuration processes, often say Arduino just works. This is, I believe, our greatest value — that a user can begin using Arduino in the simplest and fastest way possible. And as Alessandro mentioned, we are agnostic. Six years ago, I couldn't make this claim because Atmel was essentially our main sponsor, but now we are completely impartial.

Elektor: Could you talk about any improvements or changes you've made to the ESP32 platform to make it more compatible with the objectives of the Arduino Nano ESP32?

Martino Facchin: We modified the upload process typically used for ESP32, which traditionally required users to switch to the native USB module to operate the ESP tool — a process not seamlessly integrated with our IDE. Now, when uploading a sketch, an off-the-shelf (OTS) method is employed. The sketch is uploaded over Device Firmware Update (DFU), pushing it to the second partition through an over-the-air (OTA) update. The bootloader is then instructed to attempt a reboot from this second partition. If successful, the new sketch is loaded; if not, the original sketch remains. This implementation is a significant improvement, as Espressif had already considered various use cases. We adapted their approach to develop a faster, more reliable system with a recovery mode.

We have double tap support (you can enter recovery mode by double pressing the reset button) which was not there on the ESP board. These are some modifications that were all well-accepted by the community because of the effort. Even though the community core is backed by Espressif, it is a community effort, so everyone was happy about this.

Alessandro Ranellucci: There are two more contributions that we made to the ecosystem at large, so not specifically to the core. Some of the work we did was debugging, and we also added MicroPython support for ESP32.

Elektor: As for debugging, is it on this board that there's some access to the ESP through solder points, or is it another board?

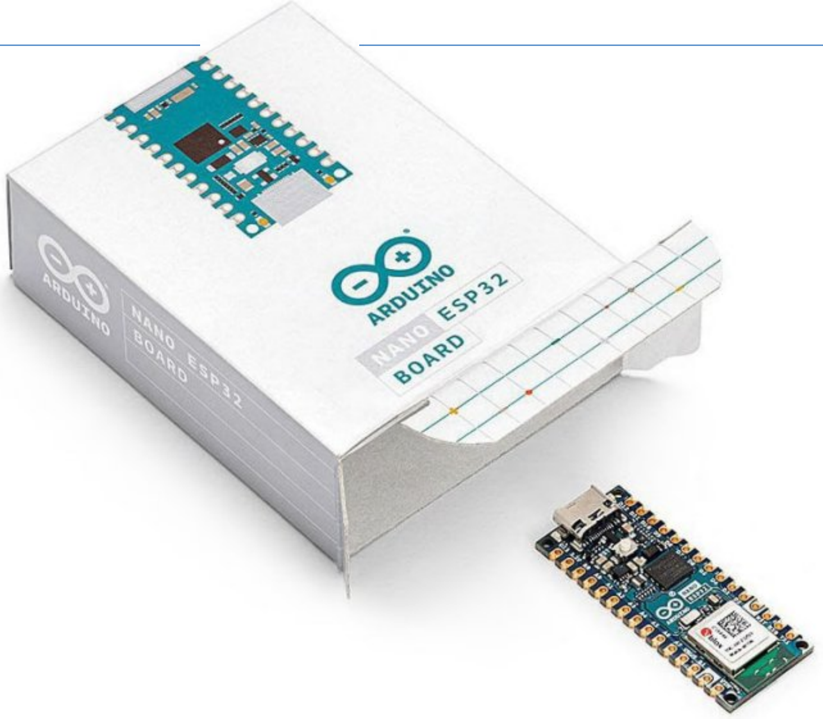
Martino Facchin: No, you can debug this board directly via USB — Espressif was kind enough to provide this. Over USB, you have one serial port, CDC, the usual stuff, and also a JTAG interface. You can interact with the JTAG interface using just the normal USB alongside normal communication. It doesn't really hurt to connect an external debugger, but you don't need to. And, in fact, it's already integrated in the IDE. You just need to connect the board, select "Debug mode (Hardware CDC)" from the menu and press the big *Debug* button. Under the hood, a bunch of things involving OpenOCD and GDB will run, but this is transparent from the user's point of view.

Alessandro Ranellucci: Our contribution has been to design a user experience for this. With Arduino, it's much easier than using professional full-fledged debugging tools, testing, documenting this, raising awareness, disseminating it to users, and then also interacting with Espressif.

Elektor: The minutiae of implementing this reminds me of questions online. There was confusion about the ports when you plugged in the board — there were two ports and people didn't know which port to use.

Martino Facchin: Yeah, it is not easy to explain because the main way we've always told people to do stuff has been to start from the basics, for example; you do it this way and there is one serial port and so on and so forth, and then you move on to an advanced topic. Advanced topics need to be explained very well. For this reason, we have documentation on how to do debugging properly. We don't just leave the feature there and let people go and experiment. Besides, people don't usually read documentation, so we're doing our best to avoid this friction and make everything available. The documentation should still be read, and everything is explained very well in it.

Alessandro Ranellucci: And the most recent version of the Arduino IDE, released one month ago, has many improvements



to port selection and board selection; the confusion about multiple ports or ports changing after you do operations and so on is handled in a friendlier way.

Martino Facchin: Also, we added this DFU menu, so we now have pluggable discovery. Pluggable discovery is a very interesting concept because you can discover "things" in many ways besides the serial port or over Wi-Fi via MDNS. Paul Stoffregen added discovery via HID to Version 1 of Arduino IDE because his bootloader was HID-based, but this was always an external patch for the Arduino installer. We decided to make this available to everyone. So, he ported his discoverer to Arduino IDE Version 2. Then, we had another problem with the Minima — when you went in bootloader mode, it didn't expose a serial port. It was confusing not seeing it on the menu. Therefore, we decided to add our discoverer to the DFU port. ESP32 has a DFU during runtime, due to the way you upload code. Occasionally, it might appear on your computer, but, as I mentioned earlier, everything is explained in the documentation. This port is meant for your uploading and you can select either one. It does not change the fact that it will accept uploads.

Elektor: Were there any compatibility changes in terms of libraries of existing code when transitioning to the ESP32 chip for the new Nano model?

Alessandro Ranellucci: Indeed — every time we add a new architecture to the family. There are the basic official libraries that need to be ported, especially when they are highly optimized, so they run with low-level code. In this case, we had to extend some basic libraries. But the ecosystem of libraries for the ESP32 was much more mature.

Elektor: Can you highlight any security features or considerations that led you to choose the ESP32 for the Arduino Nano ESP32, especially for IoT applications?

Alessandro Ranellucci: I would not say that we chose the ESP32 especially for security considerations. Of course, ESP32 has some capabilities there, which we use partially at the moment.

Martino Facchin: Really partially — we aren't using any encryption or what they call "secure boot" because it makes the user's life extremely difficult. You have to sign every binary you produce, and then you must change it for every board, or it's meaningless. So, we're not implementing it, but we're allowing it, of course. From an integrator standpoint, when you take this product and want to make it really secure, it has all the capabilities for doing so. But, this is not the reason we selected it.

Alessandro Ranellucci: Usually, on all our boards, we had a separate hardware secure element. So that's the method we chose for all our products. In this case, we did not implement it because the main microcontroller, in theory, has these capabilities. For now, though, we decided to use this chip as-is, to keep it simple. Nothing prevents us from further development, of course.

Martino Facchin: The same happened with the Portenta H7, for example, when we released the secure bootloader and the MCUboot infrastructure to provide secure OTA and secure updates; this was an opt-in, and not something we provide straight out the box. Eventually, I am pretty sure we will also provide some sort of documentation via a menu.

Elektor: Are there any plans to use the Espressif chip in more Arduino boards, such as the PRO line?

Alessandro Ranellucci: I cannot answer for the PRO line, but, in general, yes.

Martino Facchin: We love the u-blox form factor because it really fits well with the Nano's. It's very small. We've been using their modules throughout almost all of the Nano boards, either as a companion chip for Wi-Fi or, for the Nano BLE, as the main microcontroller. So, we're quite happy with them as manufacturers. At the same time, of course, on the UNO R4, we had the normal Espressif module. And yes, we are planning to do other stuff.

Elektor: Can you go into more detail about collaboration or contact you had with the ESP32 development team to guarantee seamless integration into the Arduino ecosystem?

Alessandro Ranellucci: Well, there's a good development team at Espressif. They do a great job involving the community in the development process. So, we had in-person meetings with them and recurrent calls as well. We also shared a communication channel on Slack or other communication platforms so that we had a direct, daily way to update each other, informing each other of issues and raising our hands before going public on GitHub. It was a very close collaboration. So, we can name many people from Espressif who helped us in this collaboration.

Alessandro Ranellucci: Actually, at a higher level, Ivan Grokhotkov (VP of Software Platforms, Espressif) helped us to facilitate all the communication, and Pedro Minatel (Developer Advocate) was super helpful in helping us to talk to the right person.

Elektor: Looking ahead, do you envision the partnership between Arduino and Espressif evolving and what opportunities do you foresee for further innovation and collaboration?

Alessandro Ranellucci: I would say that beyond the hardware products, we agree a lot across the two teams on the need to work closely on API standardization. The API is now more interoperable between the Arduino and ESP32 worlds. That's where we would like to keep the collaboration going.

Martino Facchin: Also, Espressif has been branching out into a lot of different topics, such as Rust. They have many developers working on Rust, which is not a priority for us, but we could gather some very interesting ideas from the work they do there. Then there's the Zephyr operating system, in which we're both partners. In the board technical steering committee, we can decide stuff, and so can Espressif. All the development we're both doing is quite focused on making future tools for future generations better. In the end, we're working toward the same goal.

Elektor: When comparing ESP32 to other chips, could you elaborate on the specific benchmarks or performance metrics that led you to conclude that it was the ideal choice for the Arduino Nano ESP32?

Martino Facchin: The ESP32-S3 is a good chip. Its power consumption is not amazingly low, but it's not bad. Ultra-low power capabilities are there if you really want to push it, but we're not going around telling our users to go and put everything into deep sleep and use the ultra-low power, even though it's nice to see these things exist. I would say that performance is not the main reason the Nano family exists. Rather, it's easy to use. We had the chance to benchmark against Portenta H7, for example, on machine learning capabilities. The Portenta H7 is about seven times faster than the ESP32, even at comparable clock speeds. The ESP32 has half the clock speed of the Portenta H7. The Nano exists for ease of use, environment, library availability, form factor, etc.

Elektor: Considering the ESP32's real-time operating system capabilities, how does this factor into the design of the ESP32 and its potential for multitasking?

Alessandro Ranellucci: This is, I would say, a growing need — how to use multiple cores, multitasking, and so on. This need is not currently that in demand among makers, but an ecosystem will need to be provided.

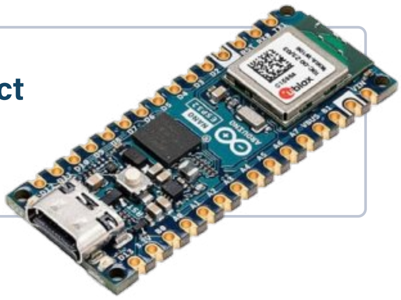
Martino Facchin: We tried a standardization effort a little over a year ago, with something called Arduino threads. If you look at GitHub, we've tried to push this idea of having threads hidden behind different tabs in your IDE. So, you have a tab with `setup()` and `loop()`, as usual. Then, you have another tab with another `setup()` and `loop()`, and this represents your second thread. Then there's a third, a fourth, whatever is required. There's still the issue of variable synchronization, where you have the usual RTOS restriction requiring you to use different variable names between two threads. We solved it on a higher level. So, we hid this complexity using Mbed, but it wasn't actually an Mbed-only thing. We wanted to port this to multiple operating systems.

If it succeeded, the maker community didn't respond at all to this thing. So, it's probably not something that's really required by makers. Threading is nice, but it also makes your life difficult. And at the same time, of course, you can use FreeRTOS; you can do whatever you want if you're skilled enough, but we're not pushing for it — ease of use is what we've always been about.

230524-01

Related Product

› Arduino Nano ESP32
www.elektor.com/20562



ARDUINO CLOUD

What Arduino Cloud is

Develop from anywhere

- NO CODE
  With ready-to-use templates
- LOW-CODE
  Automatically generated sketches
- FULL ARDUINO EXPERIENCE
  Either offline with the UDE2 or online with the Cloud Editor
- STORE YOUR SKETCHES ONLINE
  Use your code in your favourite Arduino development environment

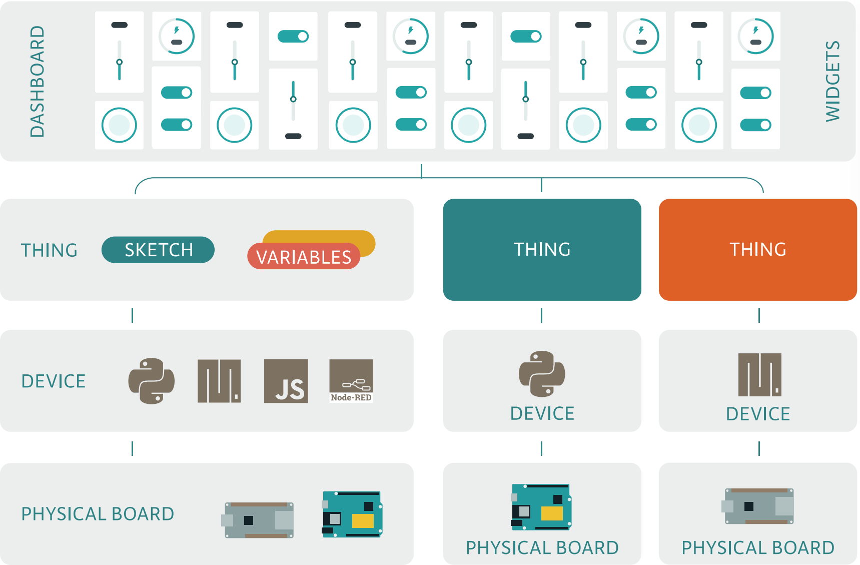
Program/Deploy

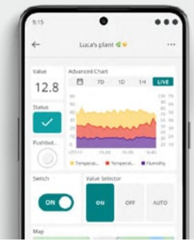
- CABLE
  Traditional USB programming
- OVER-THE-AIR (OTA) UPDATES
  Deploy your firmware wirelessly to your devices
- MASS SCALE & AUTOMATION
  With the Arduino Cloud CLI

Monitor & Control

- CUSTOM DASHBOARDS
  Using drag and drop widget
- INSIGHTFUL WIDGETS
  Interact with the devices and get real-time and historical data with dozens of widgets
- MOBILE APP
  Visualise your data in real-time from your phone with the IoT remote app

How does it work?





FULL CONTROL IN YOUR HANDS

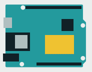
Use your dashboards on the go, and control projects from anywhere in the world, using the free IoT Cloud Remote app.

GET IT ON Google Play

Download on the App Store

Compatible hardware

WITHIN ARDUINO DEVELOPMENT ENVIRONMENTS



ARDUINO

Cloud Applications can be developed using the Arduino Cloud Editor or Arduino IDE 2.

ESP32/ESP8266

+70% Of Arduino Cloud active users use ESP-based boards.

OUTSIDE ARDUINO DEVELOPMENT ENVIRONMENTS

Node-RED

Use your favourite programming environment and language to connect your devices to the Cloud.

Third party platform integration

IFTTT · alexa · Node-RED · zapier

TRIGGER ACTIONS ON THIRD PARTY PLATFORMS

Connect your Arduino Cloud devices to external platforms such as IFTTT, Zapier and Google Services using webhooks and unlock endless possibilities.

Seamlessly integrate your IoT devices with over 2 000 apps, enabling tasks like recieving phone notifications, automating social media updates, streamlining data logging to external files, creating calendar events, or sending e-mail alerts.

Get 30% off on the yearly Maker plan with code ELEKTOR30*

cloud.arduino.cc/elektor

Questions Compiled by Elektor's Editorial Team

Your AIoT Solution Provider

Insights From ESPRESSIF

Elektor's global community comprises electrical engineers, makers, and students interested in a wide range of electronics-related topics. Several of these community members recently crafted some thought-provoking questions for the engineers and leaders at Espressif.

Teo Swee Ann founded Espressif in 2008. What led him to do so? What was his vision for Espressif in 2008, and how has his vision for the company changed since then? — *Margriet Debeij (The Netherlands)*

Upon establishing Espressif, Teo's initial goal was to create a product that automated analog design. However, the company discovered its true calling in designing proprietary internet-of-things (IoT) chips. Teo established a clear vision and strategy that emphasized creating innovative, affordable, and energy-efficient IoT solutions. Teo's unwavering emphasis on innovation has been instrumental in Espressif Systems' success. He also understood the value of engaging with the developer community and actively supported the growth of the Espressif developers' community. By fostering an ecosystem of collaboration and knowledge sharing, he ensured that Espressif remained at the forefront of technological advancements. As the technology landscape evolved, Teo led Espressif Systems in embracing emerging technologies such as artificial intelligence, machine learning, and edge computing, ensuring that the company remained at the forefront of innovation.

How does ESP-NOW compare to ZigBee or to Matter? — *Clemens Valens (France)*

Both ESP-NOW and ZigBee can be used to build low-power IoT devices, but the co-existence of ESP-Now with Wi-Fi can enable a wide range of possibilities that the applications can leverage to provide unique solutions. ESP-NOW can also provide a longer range and higher bandwidth, especially in outdoor scenarios. ESP-NOW operates even in the absence of a Wi-Fi infrastructure network and is a proprietary protocol from Espressif, whereas Matter and ZigBee are standards defined by the Connectivity Standards Alliance. Matter is an application layer protocol that supports Thread and Wi-Fi as protocols for wireless communication. Both ZigBee and ESP-NOW devices can become part of the Matter network with the use of a Matter Bridge. We have solutions for the ESP-NOW Matter bridge and Zigbee Matter bridge.

Espressif has a collaboration with Arduino which has resulted in the Arduino Nano ESP32 board. Is it possible that you will partner with the Raspberry Pi ecosystem and come up with a joint product? — Ferdinand te Walvaart (The Netherlands)

We value our collaboration with Arduino. It includes official support for the Arduino IDE (integrated development environment) for the ESP-IDF across various chipsets from Espressif. Taking this further, we have announced the Arduino UNO R4 Wi-Fi and the Arduino Nano ESP32, both including the ESP32-S3 module. We believe all of us, Arduino, Raspberry Pi, and Espressif, share a common community and open-source-driven mindset which makes collaboration a natural path. We feel Raspberry Pi has a primary focus on the MPU segment as of now, and we do not have a product to offer in that segment. Having said that, with the launch of the Pico series, there might be the possibility of collaborating and working together in the future.

What are the challenges of marketing to engineers as opposed to a more general audience? — Erik Jansen (The Netherlands)

Engineers are a group that really digs deep into the technical details, so we must find that sweet spot between being accurate and keeping things understandable. Our goal is to show them how our products fit right into their specific needs and challenges. They are all about research, so we make sure we provide them with all the technical info they crave to help them make informed decisions. We also know they spend time on technical forums and platforms, so we want to be where they are. Our marketing needs to be like a "beacon" in those spaces. Engineers value logical decisions, so our messaging focuses on how our products bring value, benefits, and technical innovation to their projects. It's about speaking their language and showing them how we are all about making their engineering journey smoother and more exciting.

How many engineers work for Espressif? Tell us about the company culture. What is the work environment like for engineers working at Espressif? — C. J. Abate (United States)

Espressif is in a true sense a technology-driven company. More than 75% of our workforce is engineering staff, totaling around 450 employees in the various R&D departments. Our employees come from diverse backgrounds and are from 30 different nationalities across our offices in 5 different countries, including China, India, Czechia, Singapore, and Brazil. We are proud to have a company culture that encourages employees to explore innovative ideas, take calculated risks, and continuously improve our products and services. Everyone is approachable, a mindset of innovation and collaboration is instilled, and employees are encouraged to pursue what they believe in and are passionate about. This helps to create differentiated and groundbreaking solutions.

Special skills are required for developing and managing AI/IoT technologies. As there is a shortage in qualified professionals for all technical markets, does this limit the growth of Espressif products and revenue? — Ruud Schaay (The Netherlands)

The shortage of skilled professionals in AI/IoT does indeed pose challenges. Our products cater to developers, offering user-friendly platforms supported by extensive resources. This enables enthusiasts from diverse backgrounds to engage with AI/IoT technologies effectively. We also actively collaborate with educational institutions worldwide through our global teams and online communities, providing workshops and resources to bridge the skills gap. This approach not only supports the growth of Espressif products, but contributes to the larger tech landscape. Our focus on accessibility and education empowers aspiring professionals, fostering a culture of innovation and skill development. While the shortage of talent is a concern, we are committed to equipping individuals with the tools and knowledge necessary to thrive in the AI/IoT arena, propelling both Espressif and the industry forward.

Espressif seems to have one of the strongest RISC-V offerings for MCUs. Will all future devices use RISC-V? — Stuart Cording (Germany)

We ensure that our hardware is widely accessible, and our software is available in the open-source community. This is a core philosophy for Espressif. The adoption of RISC-V processors into our MCU offerings was a natural progression, and we're proud of what we've been able to achieve by providing a hassle-free transition for customers between the various products under the same development framework. We are committed to the further adoption of RISC-V into our portfolio and will have our first dual-core RISC-V product (aka ESP32-P4) on the market soon. The adoption of RISC-V provides us with flexibility in implementation and enhances our rich intellectual property (IP) portfolio, which is critical to providing more advanced and affordable solutions to our customers.

While 5G is moving the market toward deterministic networks, with high computational speed and real-time response, the market is likewise becoming very sensitive to energy consumption. How is Espressif addressing this seemingly impossible equation? — Roberto Armani (Italy)

It is true that energy savings and carbon footprints are becoming increasingly important goals for our customers as well as their end consumers. We do recognize this and continue to improve our products and solutions to cater to this growing need. For example, we have re-architected our approach for light sleep mode, which, in the newer products such as the ESP32-C6 and the ESP32-H2, allows the application to power down most of the peripherals, which can provide a reduction of up to 80% in current consumption. We also provide product solutions to cater to growing low-power needs. Like the ESP32-C2-based ESP-NOW switch, which allows 10,000 presses on the button on a single coin cell. We will continue to innovate and focus on reducing the overall current consumption of our products.

Espressif solutions are used in thousands of electronics designs, from pro applications to DIY maker projects. Your engineering team must have a short list of favorite projects you've seen developed in the community. Can you share with us three of the most exciting or innovative Espressif-based projects you've seen come through the community? — C. J. Abate (United States)

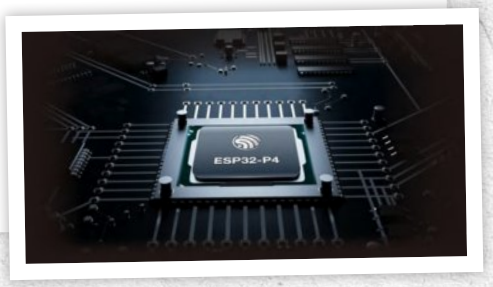
We continually come across numerous captivating projects within our inventive community and the industry. Indeed, this edition features a selection of these remarkable endeavors, spanning both hardware (HW) and firmware (FW). Among them are ESP32-based evaluation boards, no larger than coin cell batteries, as well as ingenious endeavors running Linux on ESP32-S3. Additionally, we've encountered projects simulating game consoles on ESP products and crafting synchronized digital clocks with multiple ESP boards. Some have even delved into creating VGA cards with Espressif products or porting sound drivers to the ESP32. It would be unfair to narrow it down to just three standout projects.

How many boards/modules have you sold so far? — Muhammed Söküt (Germany)

We launched our first IoC SoC product in 2014 with the release of the popular ESP8266. This was an IoT gamechanger product that integrated wireless connectivity and the microcontroller on the same die. The ESP8266 and the flagship ESP32, released in 2016, revolutionized the industry, leading to a cumulative sale of 100 million units by 2017. We have continued to receive the love of our customers ever since, and evolved as a company with new and innovative products and solutions. We shipped more than 200 million units in 2021 alone. Cumulatively, we have surpassed one billion unit shipments, which was announced recently.

Espressif's success started with the affordable and easy-to-use ESP8266, which was used mainly to establish Wi-Fi connections. Today, the Espressif chips and SoCs are often used for Wi-Fi connection, together with other controllers on the same board hosting the main application. The ESP32-P4 breaks out, as it is a high-performance CPU on its own. Will Espressif go even further in this direction? Will we see, for example, an ESP64? — Jens Nickel (Germany)

We created the ESP32-P4 after seeing that many designs use the ESP32 without any connectivity, and we believed we could provide a more powerful, yet optimized, solution to address this need. Our optimizations on RISC-V implementation have allowed us to create a high-performance, multi-core MCU with an AI (artificial intelligence) extension. We have continuously improved the set of peripherals. With all this, we are well-equipped to create new system-on-a-chip (SoC) definitions quickly. However, it may be too early to comment on what specific MCU-only offerings we will have in the future.



What are your plans for future microcontrollers (e.g., successors of the ESP32)? Can we expect things such as integrated USB, 5 GHz Wi-Fi, or Bluetooth 5.x? — *Dr. Thomas Scherer (Germany)*

Since the launch of ESP32 in 2016, we have released a series of products on the ESP32-C series, the ESP32-S series, and, lately, the ESP32-H and the ESP32-P series. These products are aimed at catering to the diverse needs of different applications and industries. Most of these newly launched products support Bluetooth Low Energy 5. The ESP32-S2/S3 and ESP32-C6 support USB OTG. The ESP32-H2, which went to mass production earlier this year, supports 802.15.4 connectivity, which enables it to be used in Zigbee- and Thread-connected applications. We recently also announced the ESP32-C5, which will support dual-band (2.4 GHz and 5 GHz) Wi-Fi 6 and will be available soon. The upcoming ESP32-P4 shall support some advanced human-machine interface (HMI) and media peripherals such as MIPI-DSI and MIPI-CSI with integrated image signal processor (ISP), H.264 encoder, etc., as well as enable a plethora of new and diverse applications.

Espressif products are widely deployed in such products as home appliances, light bulbs, smart speakers, consumer electronics, and payment terminals. Are there any upcoming plans for Espressif to expand the use of its products beyond their current deployments? — *Alina Neacşu (Germany)*

At Espressif, our goal and focus are democratizing access to IoT technologies and segmenting with innovative, developer-centric, and affordable wireless connectivity solutions. The ESP32 series of chips will continue to evolve and provide better connectivity, higher computing power, stronger security, an increasingly improved set of peripherals, and lower power consumption. In addition to this, Espressif has also evolved as a complete solution provider, identifying customer pain points and addressing them effectively with solutions that go beyond typical hardware and software development kits (SDK). ESP RainMaker, ESP Insights, and ESP ZeroCode modules are good examples of this. These solutions and products do not limit us to only a particular segment or industry.

Many Espressif products contain a wireless transmitter device of some type, often for the ISM bands and incorporating an on-board antenna. How does Espressif ensure that RF power, bandwidth, and spurious emissions are within specifications and legal limits? — *Jan Buiting (The Netherlands)*

When we architect and design our products, we consider and strictly follow the guidelines and specifications for the different communication protocols and mediums defined by the governing bodies and alliances. Our products and modules go through third-party labs for regulatory tests and certifications, which ensures that they're compliant with the specifications and limits defined by different geographical regions. You can find these certifications on our website. They can be used by our customers to accelerate their products' certification processes.

The industrial sector demands reliable and robust microcontrollers. How does Espressif plan to leverage its advancements and capabilities to stay ahead of competitors in addressing the specific needs of the industrial market? — *Saad Imtiaz (Pakistan)*

The industrial market does require a few specific requirements that aren't common in the consumer segment. They would be categorized as more stringent operating conditions, such as higher temperature, greater reliability (low failure rates), and longevity, supporting the product for multiple years. Our modules and SoCs support temperatures of up to 105°C, which makes them qualified for use in most industrial applications. Our products use mature process nodes and go through extensive reliability tests to ensure the lowest possible failure rates for our customers.

230604-01

Streamlining MCU Development With ESP-IDF Privilege Separation

By Harshal Patil, Espressif

This article introduces privilege separation in microcontroller (MCU) applications using ESP-IDF by Espressif Systems. It splits firmware into protected core and user application, simplifying development. It covers steps for getting started, making MCU development more efficient.

Typically, applications on microcontrollers (MCU) are developed as monolithic firmware. But, in a general-purpose operating system, there are two modes of operation, the kernel and the user mode. In kernel mode, the program has direct and unrestricted access to system resources, whereas in the user mode, the application program does not have direct access to system resources. In order to access the resources, a system call must be made.

The main idea behind achieving this separation is to make the end application development easier without worrying about the underlying changes in the system, just like how applications on desktop/mobile phones are developed: The underlying operating system handles the critical functionality, and the end application can use the interface exposed by the operating system.

ESP Privilege Separation

Traditionally, any ESP-IDF application on an Espressif SoC is built as a single monolithic firmware without any separation between the "core" components (operating system, networking, etc.) and the "application" or "business" logic. In the ESP Privilege Separation framework, we split the firmware image into two separate and independent binaries: protected and user application.

Getting Started

Let's start building our first project, *Blink*. The blinking LED is the "Hello World" of the embedded systems world, demonstrating the simplest thing you can do with an MCU to see a physical output. So, here we go!

Step 0: Hardware Requirements for the Project

- An ESP32-C3 or ESP32-S3-based development board with a built-in LED. Some devkits that can be used are the ESP32-C3-DevKitM-1 or the ESP32-S3-DevKitC-1. We will choose the ESP32-C3- DevKitM-1 for the following hands-on.
- A USB 2.0 cable (Standard-A to Micro-B)
- A computer running Windows, Linux, or macOS

Step 1: Getting the ESP Privilege Separation Project

- Clone the ESP Privilege Separation project repository.

```
git clone https://github.com/espressif/esp-privilege-
separation.git
```

Step 2: Setting Up the ESP-IDF Environment

- Clone the ESP-IDF project repository.

```
git clone -b v4.4.3 --recursive https://github.com/
espressif/esp-idf.git
```

- Set up the tools.

```
cd esp-idf
./install.sh
```

- Set up the environment variables.

```
source ./export.sh
```

- Apply the ESP Privilege Separation-specific patch on ESP-IDF.

```
git apply -v /idf-patches/
privilege-separation_support_v4.4.3.patch
```

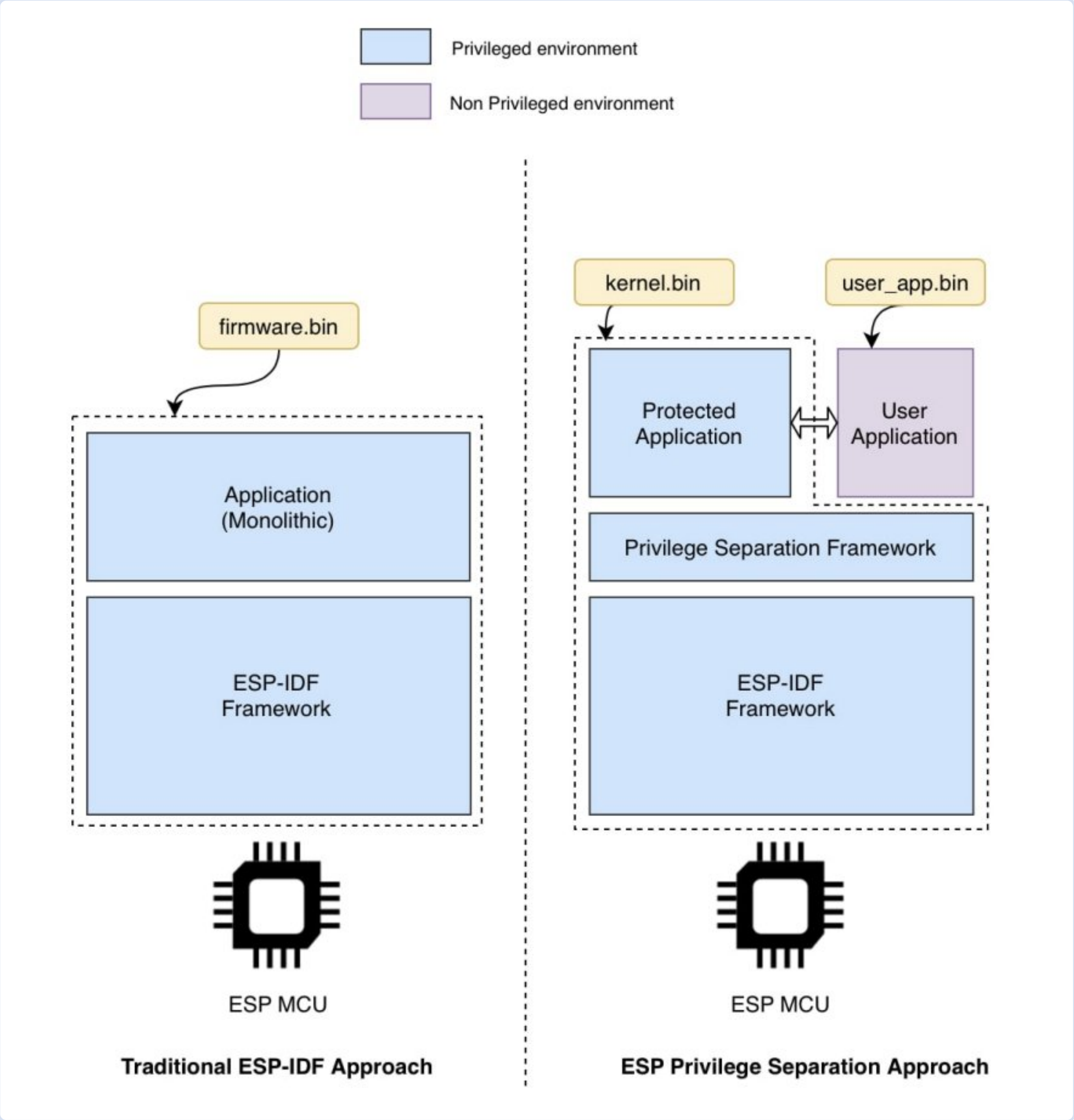


Figure 1: Traditional ESP-IDF built as a single monolithic firmware versus the ESP Privilege Separation framework, where the firmware image is split. (Source: Espressif)

Step 3: Running the Blink example

- Build the example.

```
cd /examples/blink
idf.py set-target esp32c3
idf.py build
```

- Flash and run the example.

```
idf.py flash monitor
```

Diving Into the Implementation

Now we have our LED blinking, but why would this stand out from any other blink project? Answer: Privilege Separation. Let's briefly understand the implementation:

- The example is split into two applications: the protected app and the user app.
- The protected app searches a user partition in the partition table and loads the app in the user-defined sections.
- It then configures the memory sections for the lower privilege region (`WORLD1`) and grants access as configured by the developer.
- Finally, it spawns a separate low-privilege task with the user entry point.
- Once, the user app is loaded, it spawns a task that toggles the LED connected to GPIO8 (ESP32-C3-DevKitM-1).

Integration with ESP RainMaker

Now we have our LED blinking devkit, but we are still not able to toggle the LED on our command. To qualify it as being a real "connected" device and control the LED on demand, we would need to integrate it with ESP RainMaker. ESP RainMaker is a lightweight AIoT Cloud software, which allows users to build, develop and deploy customized AIoT solutions with a minimum amount of code and maximum security. Let's jump into building the `rmaker_switch` example present in the ESP Privilege Separation project.

Step 0: Prerequisites

- ESP-IDF for ESP Privilege Separation has already been set up before building any example.
- ESP RainMaker mobile application (Android/iOS).
- Wi-Fi network.

Step 1: Setting Up ESP RainMaker

- Clone the ESP RainMaker project repository.

```
git clone --recursive https://github.com/espressif/
esp-rainmaker
git checkout 00bcf4c0c30d96b8954660fb396ba313fb6c886f
export RMAKER_PATH=
```

Step 2: Running the `rmaker_switch` Example

- Build the example.

```
cd /examples/rmaker_switch
idf.py set-target esp32c3
idf.py build
```

- Flash and run the example.

```
idf.py flash monitor
```

Step 3: Provisioning the Device

- Once the example is running, a QR code shows up in the terminal. The QR code can be used for Wi-Fi provisioning.
- Sign in to the ESP RainMaker mobile application and click on *Add Device*. This will open the camera for QR code scanning.
- Follow the *Provision* workflow so that your device can connect to your Wi-Fi network.

Step 4: Controlling the LED Switch

- Finally, you will see a *Switch* device added to the home screen of the mobile app.
- You can now try toggling the switch icon to control your LED.

But, how did we achieve this seamless ESP RainMaker integration? We introduce an `rmaker_syscall` component into our blink example that exposes system calls for the ESP Rainmaker framework, and, when the user application boots up, it initializes the ESP Rainmaker service in the protected application, and creates an ESP RainMaker switch device. As the ESP RainMaker component is placed in the protected app, system calls are needed for all public APIs exposed by it. ESP Privilege Separation's simple extensibility features allow you to add application-specific custom system calls, and this layer uses the data stored in the device context to execute user-space read and write callbacks.

OTA Firmware Updates With ESP Privilege Separation

Over-the-air (OTA) firmware updating is one of the most important features for any connected device. It enables the developers to ship out new features and bug fixes by remotely updating the application. In ESP Privilege Separation, there are two applications — *protected_app* and *user_app*, for which the framework provides the ability to independently update both binaries. The protected app, being higher-privileged, can be updated by just following the normal OTA interface provided by ESP-IDF, whereas the lower privileged user app OTA update is made possible as ESP Privilege Separation exposes a system call for the user application, `usr_esp_ota_user_app()`, to execute the OTA process.

Let's try out the user app OTA example present in the ESP-Privilege Separation project.

Step 0: Prerequisites

- ESP-IDF for ESP Privilege Separation has already been set up before building any example.
- A public URL of the hosted new user app image.

Step 1: Running the *rmaker_switch* Example

- Build the example.

```
cd /examples/esp_user_ota
idf.py set-target esp32c3
idf.py build
```

- Flash and run the example.

```
idf.py flash monitor
```

Step 2: Initiating the OTA Update

Enter the OTA URL as the user app accepts the OTA URL using the `user-ota` command through the console. This initiates an OTA, and the new user app boots up once the OTA is completed. Now, we have our connected device with the most important features ready to deploy.

You could scan the QR code to watch the ESP Privilege Separation example, demonstrating a real-world use case of user app OTA update using ESP Rainmaker and ESP Privilege Separation.

As it is said, the true measure of success for any product lies in the hands of its users, and that's you! We are really keen to hear your thoughts, so please share your experiences, suggestions, and reviews with us. Your feedback fuels our innovation and helps us refine our product to better serve your needs. Visit our website [1], drop us an email, connect with us on social media to share your insights, or open a new issue in the project's GitHub repository [2]. If you have a specific requirement that you believe fits well in this framework, or if solving such problems excites you, we would surely love to talk to you for collaboration.

230598-01

Questions or Comments?

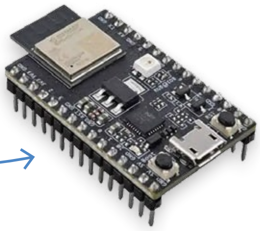
If you have technical questions or comments about this article, feel free to contact the author at harshal.patil@espressif.com or the Elektor editorial team at editor@elektor.com.

About the Author

Harshal Patil has been working as a software engineer at Espressif Systems for the past year. He is passionate about solving real-world problems by building secure, connected systems. Being a tech and sports enthusiast, outside of work he likes to explore new innovative gadgets and play football.

Related Products

- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324
- **ESP32-S3-DevKitC-1**
  www.elektor.com/20697



WEB LINKS

[1] Espressif: http://www.espressif.com
[2] The project's repository: https://github.com/espressif/esp-privilege-separation/issues

An Open-Source Speech Recognition Server...

...and the ESP-BOX

By Kristian Kielhofner (United States)

Many of us like Amazon Echo and similar devices, but there have always been concerns about privacy and data usage. The Willow Platform is a free and open-source alternative. In addition to the Willow Inference server, a high-quality voice user interface is needed that captures the audio and refines it. The ESP-BOX from Espressif brings not only the microphones and audio processing power at an attractive price point, it is also accompanied by a powerful software ecosystem.

Since releasing the Alexa platform eight years ago, Amazon has sold over 500 million Echo devices. However, there has always been concern and controversy surrounding privacy, data usage, and increasingly annoying attempts by Amazon to further monetize Echo users. Willow [1] is a platform that provides a free and open source, maker-friendly Alexa competitive voice user interface without sacrificing quality or breaking the bank.

Hardware

Raspberry Pi

A high-quality voice user interface needs to exist in and interact with the physical world. For many years, the open-source ecosystem has attempted voice user interfaces by utilizing the Raspberry Pi, various microphones, etc. (**Figure 1**). This approach comes with significant issues:

- Price point. By the time you get a Raspberry Pi, touch LCD, high-quality microphone array, speaker, suitable enclosure, etc you are looking at a price point at least 3× the cost of an Echo device ($50 USD or less). This approach requires sourcing of components, careful assembly, creating a suitable enclosure, software development, etc. Repeat this for several

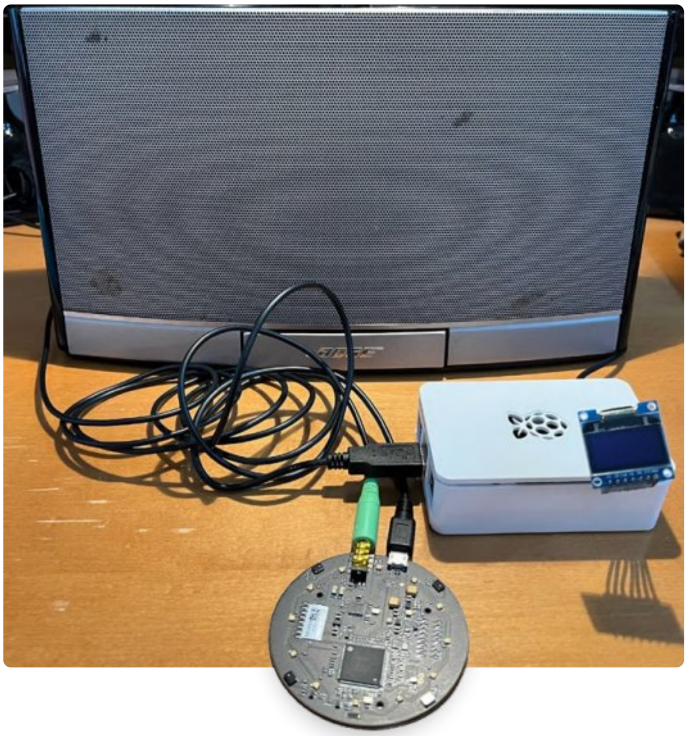


Figure 1: From the author’s prior attempts with Raspberry Pi.

devices in your environment and the required time and cost balloon significantly.

- Availability. It's been getting better recently but starting with the Raspberry Pi there have been supply chain issues with these components. For the past several years, obtaining these components has ranged from impossible to unnecessarily expensive due to secondary resale.
- Management. Raspberry Pi-based solutions often require a full Linux distribution with included support for hardware components and the bewildering array of software required for a voice user interface. Managing half a dozen Linux computers can prove challenging for many users.
- Quality. Amazon (and others) have invested significant resources in designing a voice user interface that can work in acoustically challenging environments with a wide variety of human speakers. It's not quite as simple as slapping a microphone and speaker on a Pi.
- Ecosystem. Once you get an accurate transcript of a voice command, you need to actually do something with it and then provide feedback to the user.
- Performance. With the most optimized speech recognition implementations, a Raspberry Pi 4 can approximately perform real-time speech recognition with the lowest quality model. The accuracy leaves a lot to be desired and "real time" just isn't fast enough — especially when you consider an error in transcription will require you to repeat the command, eliminating the convenience aspect of a voice user interface.

In **Table 1** you can see that a Raspberry Pi is slightly faster than real-time speech recognition with the lowest quality speech recognition model available.

Interestingly, speech recognition models have higher real time multiple performance with longer speech segments. Unfortunately, speech commands for voice assistants are very short and incur significant performance penalties as a result. This benchmark actually favors the real-time speech recognition performance of the Raspberry Pi.

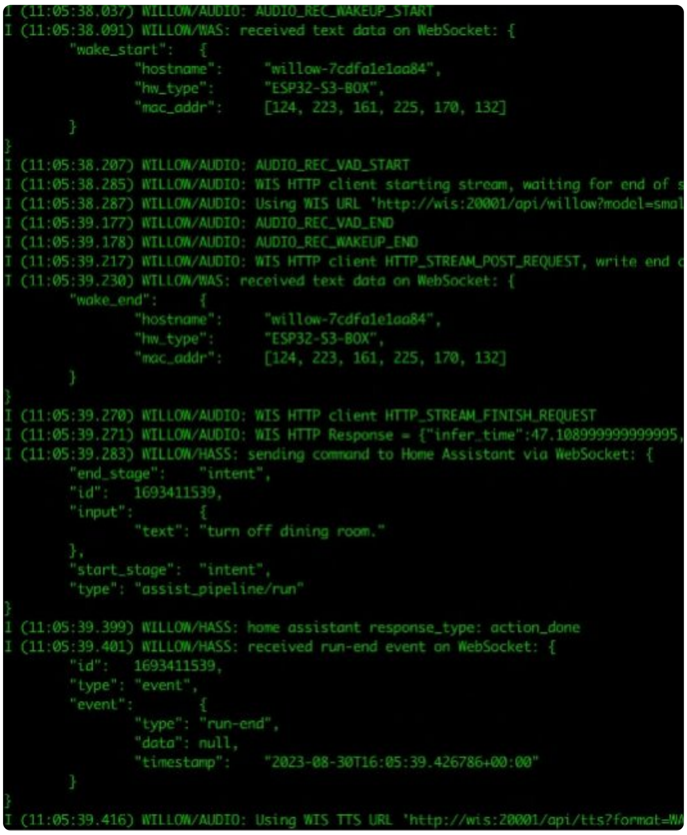
We all know and love the Raspberry Pi — it's fantastic for a wide range of applications and use cases. Unfortunately, a voice assistant with speech recognition and speech synthesis just isn't one of them.

As we can see from an example screenshot (**Figure 2**), Willow speech recognition with the Willow Inference Server went from the end of speech to confirmation of action completed by the Home Assistant home automation system in 222 milliseconds. This is roughly 25% of the time it takes the Raspberry Pi 4 to do speech recognition alone — while utilizing a speech recognition model that's substantially more accurate than the "tiny" model that is barely real time on the Raspberry Pi.

Like many others, I've made a few attempts at creating a DIY voice user interface with various approaches over the years.

Table 1: Raspberry Pi 4 speech recognition performance.

Device	Model	Beam Size	Speech Duration (ms)	Inference Time (ms)	Realtime Multiple
Pi	tiny	1	3,840	3,333	1.15×
Pi	base	1	3,840	6,207	0.62×
Pi	medium	1	3,840	50,807	0.08×
Pi	large-v2	1	3,840	91,036	0.04×



```
I (11:05:38.037) WILLOW/AUDIO: AUDIO_REC_WAKEUP_START
I (11:05:38.091) WILLOW/WAS: received text data on WebSocket: {
        "wake_start":   {
                "hostname":     "willow-7cdfa1e1aa84",
                "hw_type":      "ESP32-S3-BOX",
                "mac_addr":     [124, 223, 161, 225, 170, 132]
        }
}
I (11:05:38.207) WILLOW/AUDIO: AUDIO_REC_VAD_START
I (11:05:38.285) WILLOW/AUDIO: WIS HTTP client starting stream, waiting for end of s
I (11:05:38.287) WILLOW/AUDIO: Using WIS URL 'http://wis:20001/api/willow?model=smal
I (11:05:39.177) WILLOW/AUDIO: AUDIO_REC_VAD_END
I (11:05:39.178) WILLOW/AUDIO: AUDIO_REC_WAKEUP_END
I (11:05:39.217) WILLOW/AUDIO: WIS HTTP client HTTP_STREAM_POST_REQUEST, write end c
I (11:05:39.230) WILLOW/WAS: received text data on WebSocket: {
        "wake_end":     {
                "hostname":     "willow-7cdfa1e1aa84",
                "hw_type":      "ESP32-S3-BOX",
                "mac_addr":     [124, 223, 161, 225, 170, 132]
        }
}
I (11:05:39.270) WILLOW/AUDIO: WIS HTTP client HTTP_STREAM_FINISH_REQUEST
I (11:05:39.271) WILLOW/AUDIO: WIS HTTP Response = {"infer_time":47.108999999999995,
I (11:05:39.283) WILLOW/HASS: sending command to Home Assistant via WebSocket: {
        "end_stage":    "intent",
        "id":   1693411539,
        "input":        {
                "text": "turn off dining room."
        },
        "start_stage":  "intent",
        "type": "assist_pipeline/run"
}
I (11:05:39.399) WILLOW/HASS: home assistant response_type: action_done
I (11:05:39.401) WILLOW/HASS: received run-end event on WebSocket: {
        "id":   1693411539,
        "type": "event",
        "event":        {
                "type": "run-end",
                "data": null,
                "timestamp":    "2023-08-30T16:05:39.426786+00:00"
        }
}
I (11:05:39.416) WILLOW/AUDIO: Using WIS TTS URL 'http://wis:20001/api/tts?format=WA
```

Figure 2: Willow speech recognition performance.

Unfortunately, the result has always been the same — lots of work and cost that can make for an interesting demo but is completely impractical and unusable in the real world because of the issues noted above (and more).

ESP-BOX

Then I discovered the ESP-BOX development platform from Espressif (**Figure 3**). Like many of you, I've been using Espressif devices for a variety of applications for a decade and I know how robust, featureful, cost-effective, and actually available the hardware is.

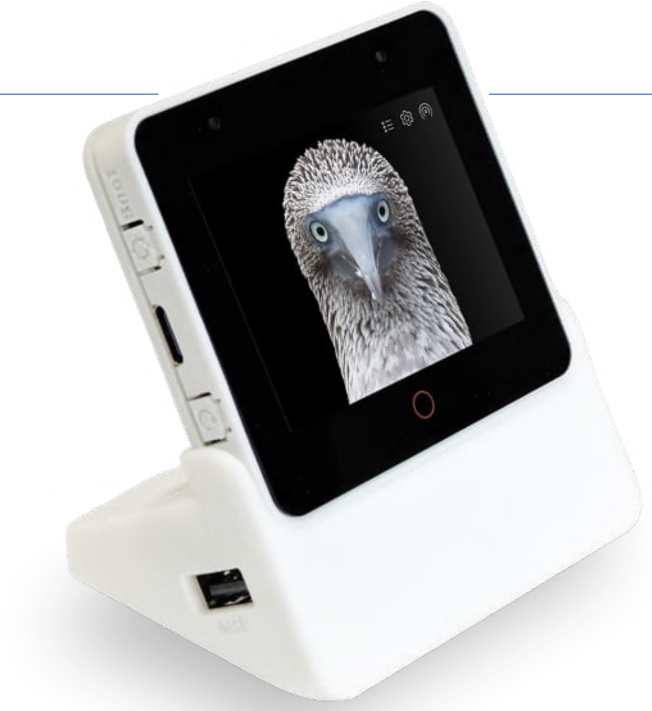


Figure 3: ESP32-S3-BOX-3 comes with a 2.4-inch display, dual microphones, and speaker.

I also know from previous projects that Espressif provides a lot of high quality supporting software and documentation for their hard- and software.

The ESP-BOX from Espressif is a development platform that is purpose built for this use case. For roughly $50 USD from your favorite DIY electronics retailer or distributor, you get:

- An ESP32-S3 with 16 MB of flash and 16 MB of high speed SPI connected RAM
- A 2.4" display with capacitive touch screen
- Dual microphones (very important — more on this later)
- Speaker for audio output
- Hardware mute button
- Many expansion options with the new ESP32-S3-BOX-3 assemblies and components

All ready to go in an aesthetically pleasing, acoustically optimized enclosure. Theoretically, with the ESP-BOX and $50, I could have a device that I could take out of the box, flash, and put in my kitchen, bedroom, office, etc.

Software

In addition to making this near perfect hardware, Espressif has long championed open source. I was thrilled to see they support the ESP-BOX with a variety of free and open-source libraries:

- ESP IDF as the fundamental SDK
- ESP ADF for audio applications
- ESP SR for speech recognition
- ESP DSP for highly optimized signal processing routines (FFT, vector math, etc.)
- An LVGL component to drive LCD displays (with touch)

With the ESP-BOX and these libraries within a week I developed a very rough proof of concept that could capture speech, send it to my speech recognition implementation, and provide the transcript to a platform to take action.

Audio

As I had learned from my prior failed attempts, a voice user interface begins at wake word. You should be able to address a voice interface with a specific word, have it wake up, and start capturing speech. Wake word is the equivalent of a power button — it shouldn't turn on randomly, and when you press the button it should work every time. For voice interface applications, if you have to repeat yourself several times for the device to activate, you pretty quickly get into a scenario where it's faster, easier, and far less frustrating to just take your phone out of your pocket and do whatever you're trying to do there.

Fortunately, the ESP Speech Recognition framework not only provides a wake word engine, it also includes several wake words. I found the wake implementation to be extremely reliable — waking consistently while also minimizing false wake activation. Thanks to ESP SR, I had a reliable voice "power button".

Then the next challenge — getting clean audio. Once again, ESP-SR to the rescue. ESP-SR includes the Espressif AFE (audio front end). A lot of ink could be spilled on the AFE alone but in a nutshell, it provides an audio processing layer between the microphone input and audio capture that performs:

Acoustic echo cancellation (AEC). One of the many challenges with far-field speech recognition is the acoustic properties of the physical environment. Distance, hard reflective surfaces, convoluted audio paths (corners, objects in the way), etc can capture a lot of echo from the environment. The AEC implementation in ESP SR eliminates much of this echo in addition to removing echo in scenarios where there is bi-directional audio (such as a speakerphone application).

Blind source separation (BSS). In noisy environments, it's important to eliminate non-speech noise that can contribute to poor quality speech recognition. The ESP-SR BSS implementation, utilizing multiple microphones, can essentially "focus" audio capture on the direction of incoming audio to significantly reduce captured background noise.

Noise suppression (NS). In cases where there is only one microphone (or only one microphone is active) noise suppression can significantly reduce the capture of non-human audio. For applications where custom hardware is used single microphones can significantly reduce bill of materials costs and design complexity.

Voice activity detection (VAD). Reliable wake word is only one piece of the puzzle. When we wake and start capturing audio, we need to stop capturing audio when the person has finished speaking.

VAD is able to detect the start and end of speech, so the user doesn't have to manually end recording.

Willow Inference Server

Now that we can wake, capture clean speech, and stop at the end of speech, we need to send it somewhere. Fortunately, Espressif provides their Audio Development Framework (ADF) for this task. In my quick and dirty proof of concept, I used the ESP ADF HTTP stream pipeline example to provide post-AFE captured audio in chunks via HTTP POST to an HTTP endpoint. I was able to quickly adapt my speech recognition inference server implementation to buffer incoming audio frames from the ESP-BOX, wait for an end marker (thanks to VAD), and immediately pass this buffer to the underlying speech recognition model. When the speech recognition inference server returns with the speech transcript it is provided as an HTTP JSON response to the ESP-BOX for further execution. This speech recognition inference server implementation became known as the *Willow Inference Server (WIS)*.

The ESP-BOX with Willow is able to take the speech transcript and send it to a user-configured Home Assistant, OpenHAB, or custom HTTP REST endpoint. Willow will display the speech recognition transcript and response from the command endpoint on the display. Depending on user configuration it will also play a success/failure tone or use text to speech from the Willow Inference Server to speak the output text resulting from the voice command.

Today, with the ESP-BOX, Willow, and the Willow Inference Server I'm able to say a wake word, capture a command, and send it to Home Assistant — with the latency from end of speech to the action being completed in well under 500 ms (**Figure 4**), depending on WIS hardware and configuration.

Multinet: No Extra Servers or Hardware

However, the magic of ESP-SR isn't over yet. In addition to the provided wake word models, ESP-SR includes an on-device speech recognition model called *MultiNet* for completely on-device voice command recognition. ESP-SR includes the ability to recognize up to 400 pre-defined voice commands completely on the device (without a Willow Inference Server). Willow includes support for MultiNet and when used with Home Assistant it can even pull the names of configured entities to automatically generate this grammar (**Figure 5**) — without any additional components and with performance and accuracy comparable to that of the Willow Inference Server for these predefined commands.

Still Maker- and Hacker-Friendly

While the ESP-BOX with Willow can replace Alexa devices in minutes, it still keeps true to its Espressif maker roots.

The ESP-BOX supports a wide range [2] of components with various sensors, GPIO, USB host interface and more via the expansion interface. Serial and JTAG are of course available via the USB C port on the main unit.

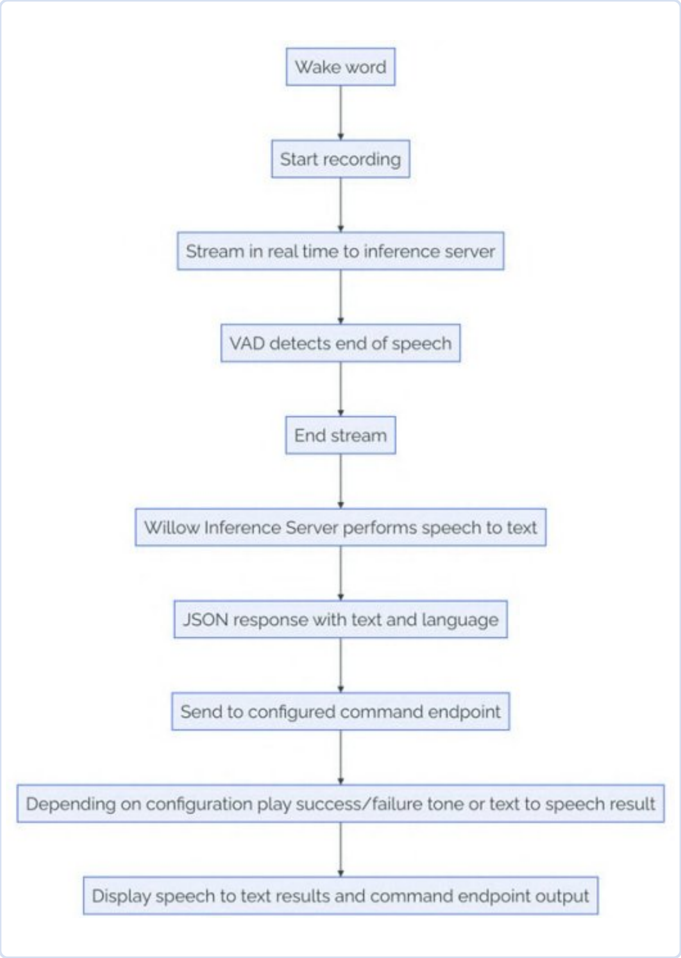

Wake word
Start recording
Stream in real time to inference server
VAD detects end of speech
End stream
Willow Inference Server performs speech to text
JSON response with text and language
Send to configured command endpoint
Depending on configuration play success/failure tone or text to speech result
Display speech to text results and command endpoint output

Figure 4: Willow inference server flow.

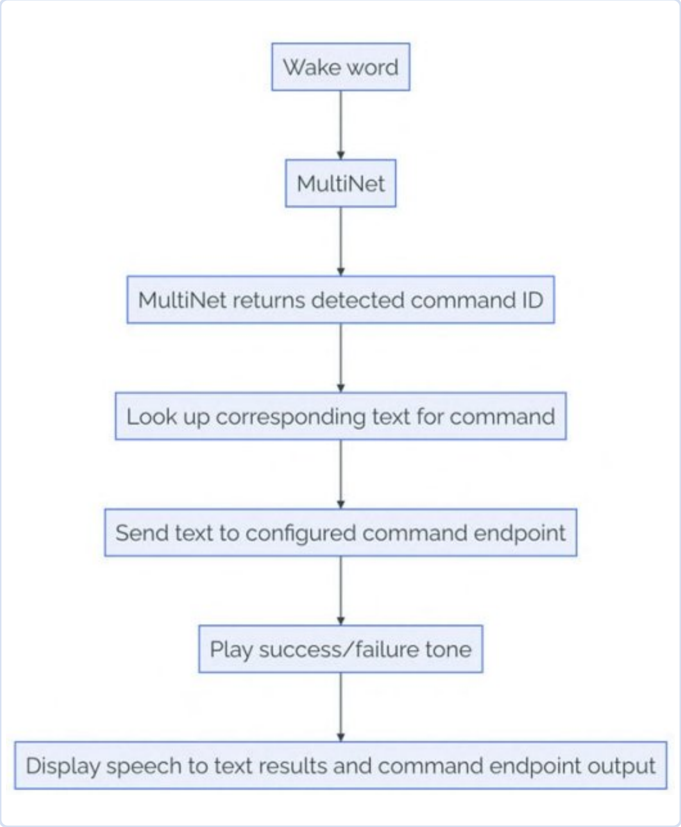

Wake word
MultiNet
MultiNet returns detected command ID
Look up corresponding text for command
Send text to configured command endpoint
Play success/failure tone
Display speech to text results and command endpoint output

Figure 5: MultiNet mode flow.

Willow is written in C with ESP-IDF but the ESP-BOX also supports platforms like Arduino, PlatformIO, and CircuitPython.

With Willow and the ESP-BOX we now have what has long been the “holy grail” of open-source voice user interfaces: an Alexa like experience at a similar price point with more flexibility, control, and complete privacy that also provides the open-source software and maker hardware interfaces we all enjoy!

230564-01

About the Author

Kristian Kielhofner is the founder of Willow — an open-source project to create a local, self-hosted Amazon Echo/Google Home competitive Voice Assistant. Since playing with the Apple IIe as a child, Kristian has been a lifelong technology enthusiast and has gone on to found multiple open-source projects and startups in the voice and machine learning spaces. Kristian now spends his time working on Willow, other open-source projects, and advising early-stage technology companies.

Questions or Comments?

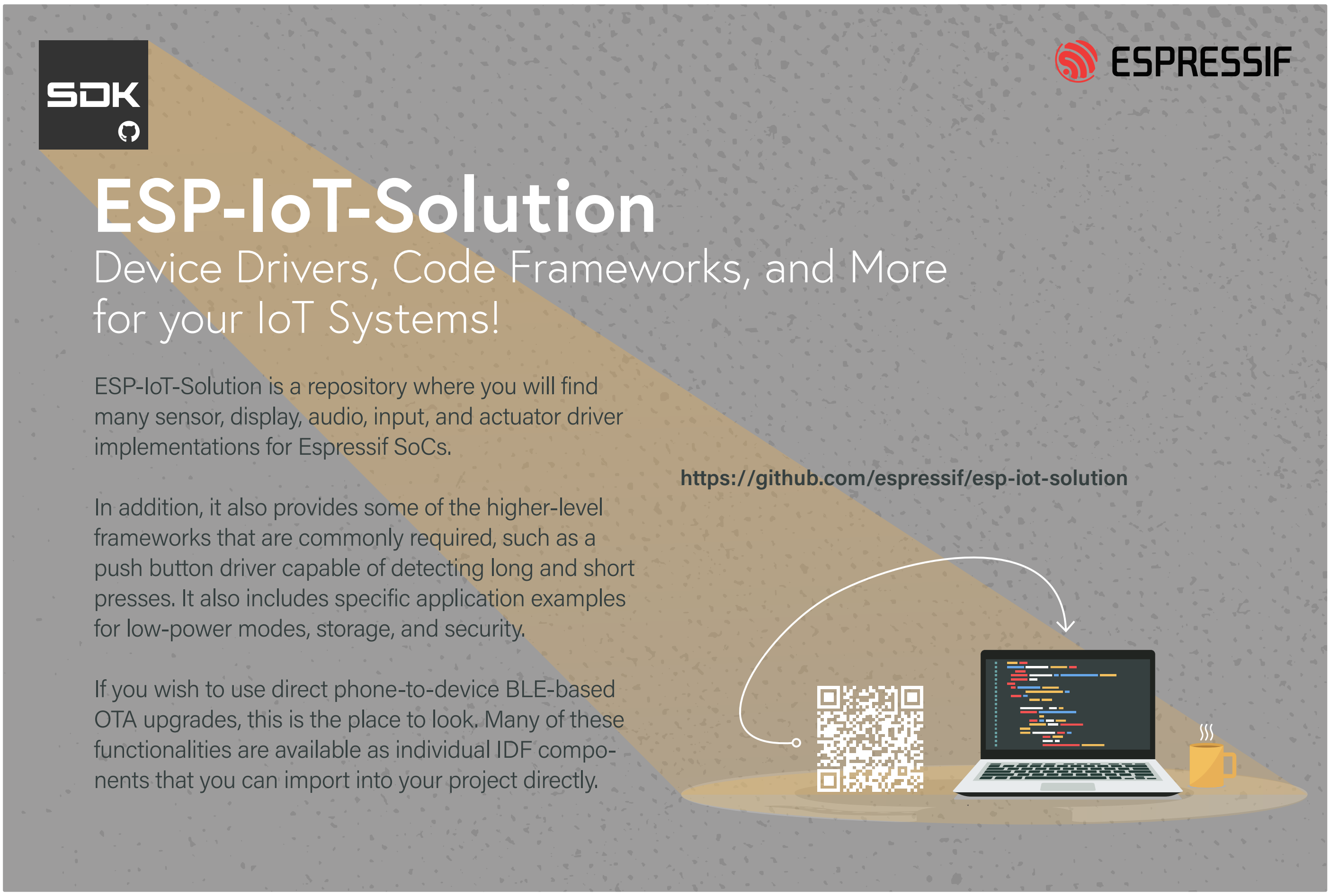
If you have technical questions or comments about this article, feel free to contact the author at kris@tovera.com or the Elektor editorial team at editor@elektor.com.

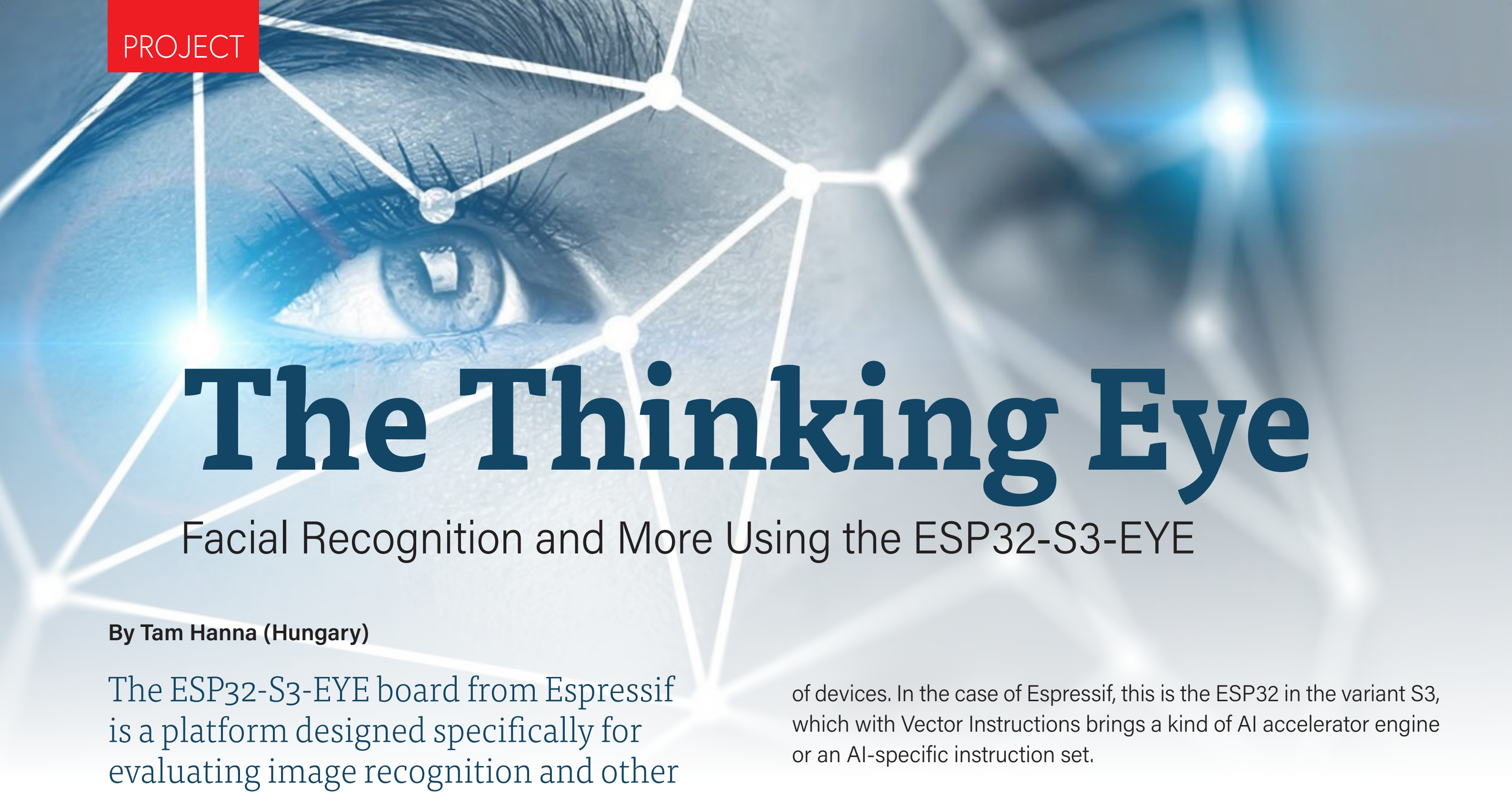
Related Products

- **ESP32-S3-BOX-3**
  www.elektor.com/20627

WEB LINKS

[1] Willow platform: https://heywillow.io
[2] ESP32-S3-BOX-3: https://www.espressif.com/en/news/ESP32-S3-BOX-3





The Thinking Eye

Facial Recognition and More Using the ESP32-S3-EYE

By Tam Hanna (Hungary)

The ESP32-S3-EYE board from Espressif is a platform designed specifically for evaluating image recognition and other AI applications. It comes with a software ecosystem and numerous examples, both for the manufacturer's ESP-WHO framework for facial recognition and for manual operation using TensorFlow. Let's get started!

The progress and increased sophistication of artificial intelligence algorithms has been breathtaking in recent years. There are now high-performance systems that deliver impressive human-like performance (especially in the case of AI hosted in the Cloud). Moore's law and the ever-increasing demands of end-users ensure that many microcontroller manufacturers include "AI-optimized" chips into their range of devices. In the case of Espressif, this is the ESP32 in the variant S3, which with Vector Instructions brings a kind of AI accelerator engine or an AI-specific instruction set.

A robust and active developer ecosystem has evolved around this ESP32-S3 over the past few months, making it easier for developers to implement various artificial intelligence applications. In this article, we will take a closer look at its capabilities and experiment with the technology.

Stress-Free Startup

Espressif is well aware of the needs of developers. Since the first release a few years ago of its ESP32-LyraT board, targeted at audio applications, the company has consistently introduced evaluation boards optimized for "special-purpose" applications.

I will use the ESP32-S3-Eye to run all the examples described in the following steps. **Figure 1** shows a block diagram of all the components included on the PCB.

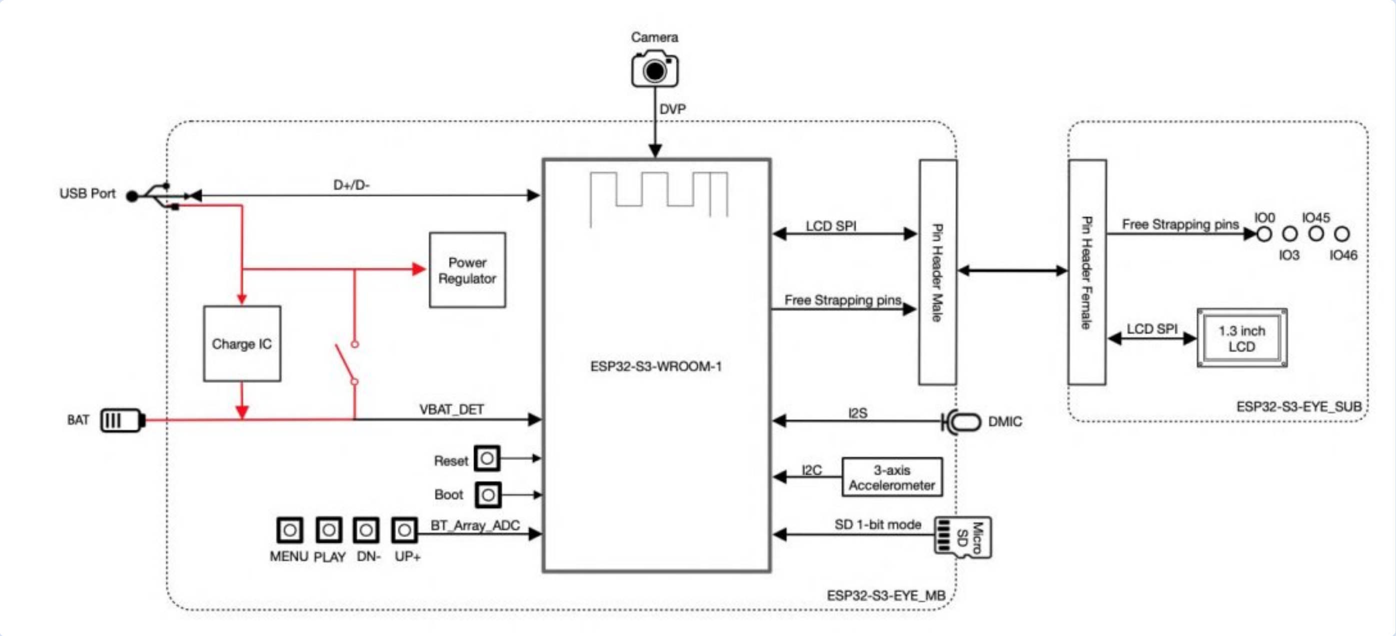


Figure 1: The ESP32-S3-EYE block diagram (Source: [10]).

One compelling reason to use this particular evaluation board, priced at around $45 in the US, is the inclusion of both a camera and a small LCD on the PCB. [1] This color screen proved to be very helpful, allowing the results of the machine learning process to be seen directly, without the need for a PC.

A digital microphone is fitted to the front of the PCB surface just below the screen. It can be used to test various speech recognition engines.

It's worth noting that the experiments conducted could also be carried out using a different hardware platform. The ESP32-S3-EYE board schematic can be found at [2], and it serves as a recommended basis for custom circuit designs. The components used by Espressif are generally readily available from many distributors.

Component Availability and the ES8388

Sourcing the semiconductor components preferred by Espressif sometimes requires a different approach. In the case of the ES8388 audio codec, I couldn't find any of the traditional distributors based in the US or Europe that stocked this item. A direct inquiry to the manufacturer, however, pointed me in the direction of Newtech Component Ltd., a distributor who were kind enough to send 20 samples free of charge. For this, it is advisable to make use of a freight or parcel forwarding service which can supply an address (local to the distributor) where the samples can be sent. I often use TipTrans for this purpose.

The board is supplied complete with the basic firmware installed which avoids any possible hiccup in the system setup procedure. If the ESP32-S3-EYE board is in its factory state, you only need to supply power through the micro-USB port and wait a few seconds. The facial recognition application will now launch as shown in **Figure 2**.

It's worth noting at this point that AI and ML algorithms exhibit remarkable resilience when it comes to the quality of the captured input data. The photo shown in Figure 2 was taken with the protective plastic film still over the camera lens. The resulting reduction in contrast (and the state of my unshaven face) were not enough to fool the AI facial recognition process.

Back to the Future

You may at some point wish to reset your ESP32-S3-EYE to its "factory default" state. Pre-built .bin files can be found at the URL *https://github.com/espressif/esp-who/tree/master/default_bin*. These can be flashed to the board in the same way as any other binary file.

A First Experiment

The, admittedly, almost unlimited availability of venture capital has led to a flood of AI frameworks. Logic dictates that such companies would not only target PCs but also include microcontrollers in their list of supported platforms.

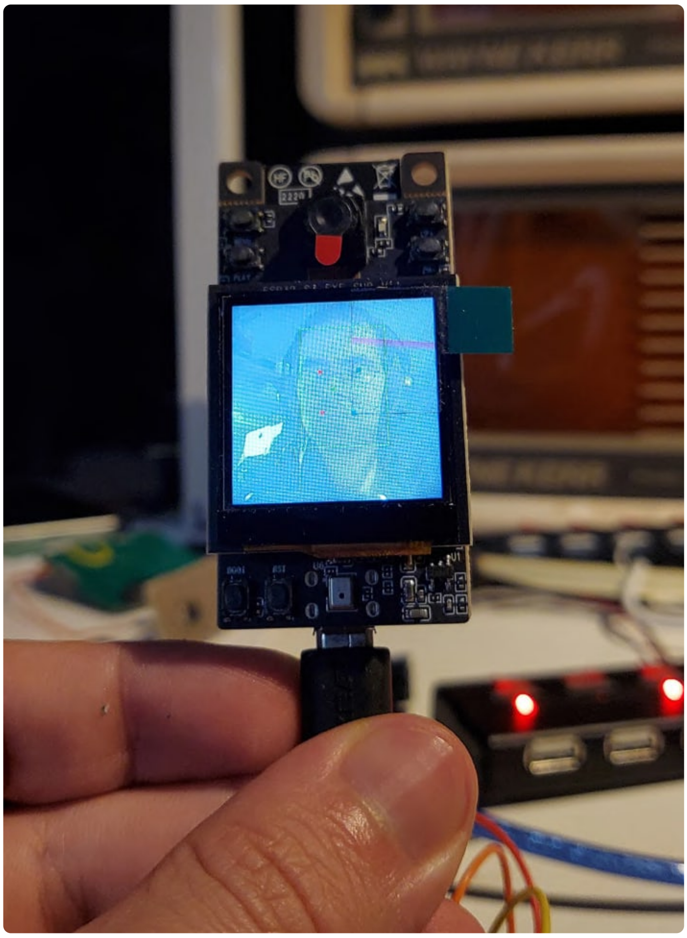


Figure 2: The camera worked, even in my dingy lab with the lens tape still in place!

A comprehensive overview of the current state of the market would be beyond the scope of this article, so for our first step, we will work with the image processing development platform *ESP-WHO* managed by Espressif. This is a framework "optimized" for certain types of image recognition, and its structural layout is shown in **Figure 3**.

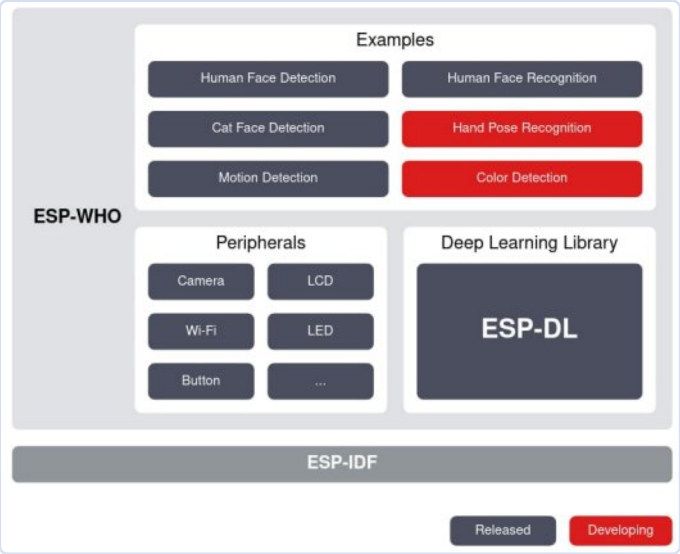
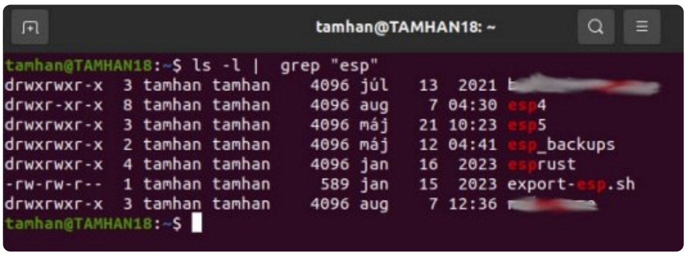


Figure 3: ESP-WHO uses various other parts of the Espressif ecosystem (Source: [11]).



```
tamhan@TAMHAN18: ~
tamhan@TAMHAN18:~$ ls -l |  grep "esp"
drwxrwxr-x  3 tamhan tamhan    4096 júl   13  2021 [redacted]
drwxr-xr-x  8 tamhan tamhan    4096 aug    7 04:30 esp4
drwxrwxr-x  3 tamhan tamhan    4096 máj   21 10:23 esp5
drwxrwxr-x  2 tamhan tamhan    4096 máj   12 04:41 esp_backups
drwxrwxr-x  4 tamhan tamhan    4096 jan   16  2023 esprust
-rw-rw-r--  1 tamhan tamhan     589 jan   15  2023 export-esp.sh
drwxrwxr-x  3 tamhan tamhan    4096 aug    7 12:36 [redacted]
tamhan@TAMHAN18:~$
```

Figure 4: Different versions of the ESP-IDF can generally coexist on a workstation.

The *ESP Deep Learning Library* (ESP-DL), available at [3], is an important resource. This is essentially an optimized library that provides various AI techniques and achieves significant latency reduction when used in combination with a hardware accelerator engine.

One point of note is that, out-of-the-box, the WHO component runs under version 4.4 of the *ESP-IDF* and is not yet supported in version 5.0. In my consultancy work I generally use version 4.4 for projects, but also have multiple variants of the ESP32 development environment installed on my computer, as you can see in **Figure 4**.

The IDF version used in the following steps is identified as follows:

```
tamhan@TAMHAN18:~/esp4/esp-idf$ idf.py --version
ESP-IDF v4.4.4-dirty
```

Next, we are ready to download the *ESP-WHO* code. This is done by using the following `git`-command in the command line:

```
tamhan@TAMHAN18:~/esp4$ git clone --recursive https://
github.com/espressif/esp-who.git
. . .
```

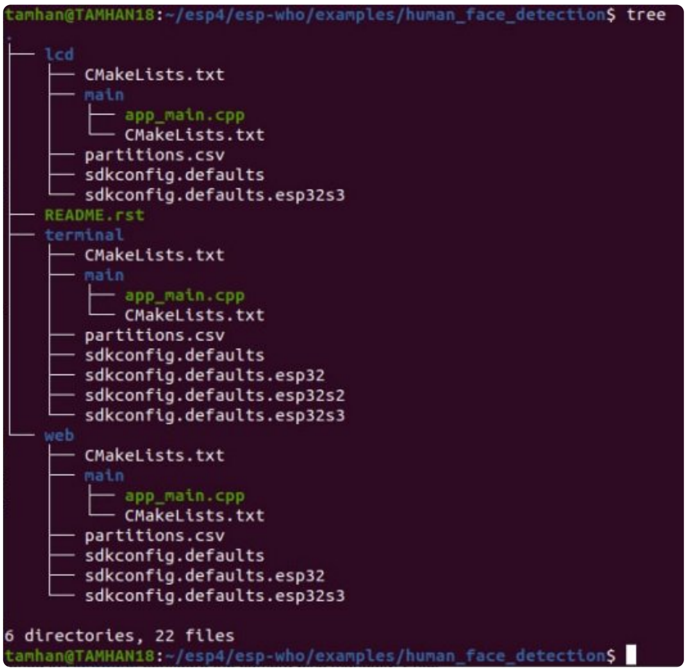
The execution of the `Compressing objects:` command step can take some time; repositories are sometimes quite large and slow to work with. After completing the work, it is advisable to perform submodule initialization according to the following scheme:

```
tamhan@TAMHAN18:~/esp4$ cd esp-who/
tamhan@TAMHAN18:~/esp4/esp-who$ git submodule update
--recursive --init
```

In most cases, the command `git submodule update --recursive --init` will not display any content on the screen — this informs you that the current image is "complete".

Some Examples

The *ESP-WHO* face recognition software provided by Espressif allows for three different user interface options. Example projects can be seen in **Figure 5**.



```
tamhan@TAMHAN18:~/esp4/esp-who/examples/human_face_detection$ tree
.
├── lcd
│   ├── CMakeLists.txt
│   ├── main
│   │   ├── app_main.cpp
│   │   └── CMakeLists.txt
│   ├── partitions.csv
│   ├── sdkconfig.defaults
│   └── sdkconfig.defaults.esp32s3
├── README.rst
├── terminal
│   ├── CMakeLists.txt
│   ├── main
│   │   ├── app_main.cpp
│   │   └── CMakeLists.txt
│   ├── partitions.csv
│   ├── sdkconfig.defaults
│   ├── sdkconfig.defaults.esp32
│   ├── sdkconfig.defaults.esp32s2
│   └── sdkconfig.defaults.esp32s3
└── web
    ├── CMakeLists.txt
    ├── main
    │   ├── app_main.cpp
    │   └── CMakeLists.txt
    ├── partitions.csv
    ├── sdkconfig.defaults
    ├── sdkconfig.defaults.esp32
    └── sdkconfig.defaults.esp32s3

6 directories, 22 files
tamhan@TAMHAN18:~/esp4/esp-who/examples/human_face_detection$
```

Figure 5: ESP-IDF variants in the house!

In the following steps, we will use the *lcd* variant of the examples `~/esp4/esp-who/examples/human_face_detection.` The calculated results will then be displayed directly on the tiny screen mounted on the ESP32. The terminal variant uses the *idf.py* monitor for communication, while the *web* variant sets up a web server.

First, you need to specify the type of ESP32 controller you want to target using the following scheme. If, like me, you are using an ESP32-S3-EYE board, the string `esp32s3` is passed using:

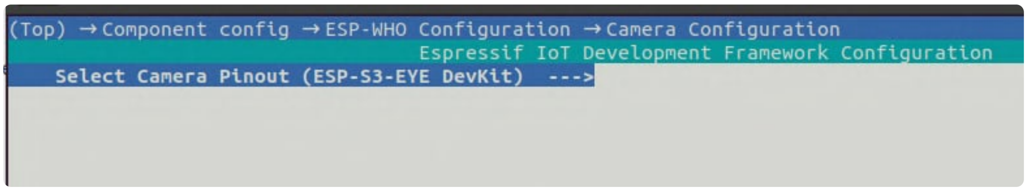
```
tamhan@TAMHAN18:~/esp4/esp-who/examples/human_face_
detection/lcd$ idf.py set-target esp32s3
```

The actual source code of the example, located in the file *main/app_main.cpp*, is impressively simple. To gain a better insight, I will first print it in its entirety before adding comments (see **Listing 1**).

The core of the *ESP-WHO* engine uses the *queue* kernel object implemented in FreeRTOS, detailed in [4]. The two static declarations provide an input and an output queue, which are later used to handle data flow through the recognition pipeline.

The rest of the code contributed by the developer is mainly focused on building a pipeline via calls to three functions. Analogies to the pipeline model used by the ESP-ADF audio framework [5] are evident.

To continue setup, you will need to enter `idf.py menuconfig` in the first step to load the familiar *menuconfig* configuration environment



```
(Top) → Component config → ESP-WHO Configuration → Camera Configuration
                             Espressif IoT Development Framework Configuration
    Select Camera Pinout (ESP-S3-EYE DevKit)  --->
```

Figure 6: If you use an in-house camera module, you need to reconfigure settings here!

Listing 1: Get in the queue.

```
#include "who_camera.h"
#include "who_human_face_detection.hpp"
#include "who_lcd.h"

static QueueHandle_t xQueueAIFrame = NULL;
static QueueHandle_t xQueueLCDFrame = NULL;

extern "C" void app_main()
{
    xQueueAIFrame = xQueueCreate(2, sizeof(camera_fb_t *));
    xQueueLCDFrame = xQueueCreate(2, sizeof(camera_fb_t *));

    register_camera(PIXFORMAT_RGB565, FRAMESIZE_240X240, 2, xQueueAIFrame);
    register_human_face_detection(xQueueAIFrame, NULL, NULL, xQueueLCDFrame, false);
    register_lcd(xQueueLCDFrame, NULL, true);
}
```

used in other ESP32 projects (and the Linux kernel). After that, navigate to the *Component config* ➜ *ESP-WHO Configuration* section. In the *Camera Configuration* field, make sure that the camera of your evaluation board is preconfigured, as shown in **Figure 6**.

Save the build configuration created in *menuconfig*, and then issue the usual `idf.py build` command to start compiling. The initial image creation will take a bit more time, as the framework needs to compile around 1,200 code files.

Before flashing the board using `idf.py flash` and the port number, you will need to put the board into bootloader mode. Use the two buttons near the USB connector: Hold down BOOT while momentarily pressing RST, then release both. Now you can go ahead and flash the program onto the ESP32 as usual. You can recognize the successful bootloader mode entry in *dmesg*, as shown in **Figure 7**. Now press the RST button again to get the new variant of the familiar facial recognition (Figure 2) running.

A Quick Look at the Code

The use of *ESP-WHO* provides a relatively simple introduction for developers to begin experimenting with AI applications without too much effort. Espressif also provides the source code for some of the components [6], which allows us to take a look at the facial recognition routine.

It uses the preexisting `HumanFaceDetectMSR01` detector, which receives some weights passed as part of the parameterization procedure:

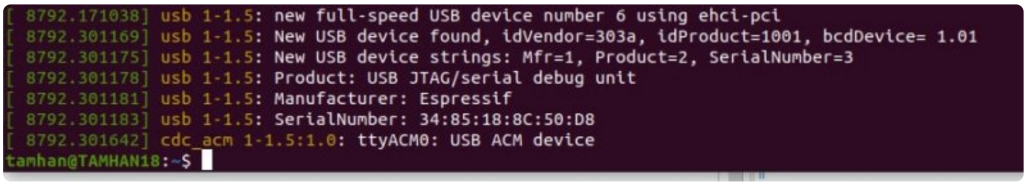
```
static void task_process_handler(void *arg) {
  camera_fb_t *frame = NULL;
  HumanFaceDetectMSR01 detector
      (0.3F, 0.3F, 10, 0.3F);
```

The actual work is done by the ML model in an endless loop, which retrieves individual frames to be processed from the queue created earlier:

```
while (true) {
   bool is_detected = false;
   if (xQueueReceive(xQueueFrameI,
     &frame, portMAX_DELAY)) {
     std::list<dl::detect::result_t> &detect_results =
       detector.infer((uint16_t *)frame->buf,
      {(int)frame->height, (int)frame->width, 3});
```

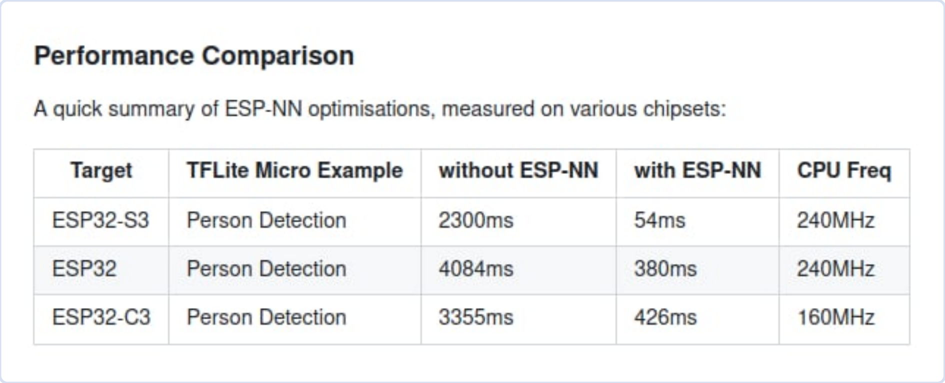
Calls to the `detector.infer()` function ensure the actual inference process is executed. If the returned `results` object has detections, the program invokes two functions for output:

```
    if (detect_results.size() > 0) {
     draw_detection_result((uint16_t *)frame->buf,
    frame->height, frame->width, detect_results);
     print_detection_result(detect_results);
     is_detected = true;
    }
   }
```



```
[ 8792.171038] usb 1-1.5: new full-speed USB device number 6 using ehci-pci
[ 8792.301169] usb 1-1.5: New USB device found, idVendor=303a, idProduct=1001, bcdDevice= 1.01
[ 8792.301175] usb 1-1.5: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 8792.301178] usb 1-1.5: Product: USB JTAG/serial debug unit
[ 8792.301181] usb 1-1.5: Manufacturer: Espressif
[ 8792.301183] usb 1-1.5: SerialNumber: 34:85:18:8C:50:D8
[ 8792.301642] cdc_acm 1-1.5:1.0: ttyACM0: USB ACM device
tamhan@TAMHAN18:~$ 
```

Figure 7: This status message indicates a successful connection.



Performance Comparison

A quick summary of ESP-NN optimisations, measured on various chipsets:

Target	TFLite Micro Example	without ESP-NN	with ESP-NN	CPU Freq
ESP32-S3	Person Detection	2300ms	54ms	240MHz
ESP32	Person Detection	4084ms	380ms	240MHz
ESP32-C3	Person Detection	3355ms	426ms	160MHz

Figure 8: Activating the AI accelerator gets the job done quicker (Source: [9]).

The rest of *ESP-WHO* generally consists of preexisting software components, such as *ESP-Cam* [7] for example, to handle access to the camera. I will not venture further into the *ESP-WHO* software because the examples provided are self-explanatory.

Delve Deeper Using TensorFlow

Those seeking better control over their system's behavior might be tempted to code everything from scratch, but that is not a particularly practical solution. Unless a great deal of care is taken, the system can quickly become unmanageable and gives rise to higher maintenance costs over the software lifespan.

Google's *TensorFlow* library, available at [8], has become something of a quasi-standard and has been available in optimized versions for various microcontrollers for some time now. It's important to note that on GitHub, it actually consists of two repositories: the reason for this quirk is because Google shifted the distribution of generic and vendor-specific code in the *TensorFlow* library part way through the framework's development. The version of the library we need can be found at [9].

Since TensorFlow also accesses various components of the ESP-IDF framework in the background, deployment from GitHub should also be made including the `--recursive` parameter which points out to the command line tool the need to provide related code locations and ensures we get the main repository and all submodules:

```
tamhan@TAMHAN18:~/esp4$ git clone --recursive https://
github.com/espressif/tflite-micro-esp-examples.git
```

For an initial smoke test, we can use the example *~/esp4/tflite-micro-esp-examples/examples/hello_world*. It uses an ML model optimized by "predicting" sine function values — an alternative to the more conventional CORDIC algorithm and a model that executes with minimal input data requirements. In the next step, once again, enter `idf.py set-target esp32s3` to optimize the project skeleton for the ESP32-S3 target.

Depending on the version of ESP-IDF available on your workstation or PC, you may encounter error messages during the parameterization of the code downloaded from the repository (*Invalid manifest...*). These errors point to partially outdated components in the compilation toolchain. To resolve this, simply execute the following program using the command line:

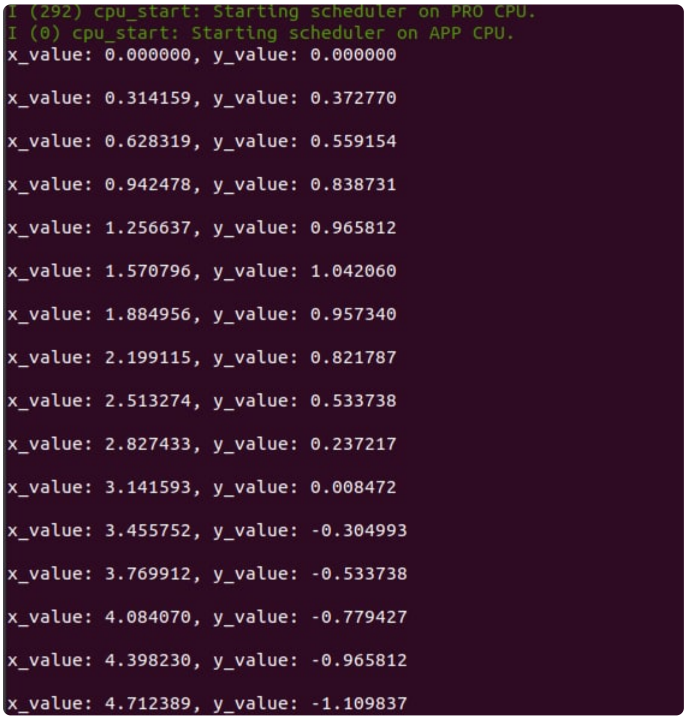
```
/home/tamhan/.espressif/python_env/idf4.4_py3.8_env/bin/
python -m pip install --upgrade idf-component-manager
```

Next, open a *Nautilus* window pointing to the root folder of the project skeleton by entering the following command. Deletion of the *build* folder must be done manually so that the `set-target` command can create the compilation support files again:

```
tamhan@TAMHAN18:~/esp4/tflite-micro-esp-examples/examples/
hello_world$ nautilus .
tamhan@TAMHAN18:~/esp4/tflite-micro-esp-examples/examples/
hello_world$ idf.py set-target esp32s3
tamhan@TAMHAN18:~/esp4/tflite-micro-esp-examples/examples/
hello_world$ idf.py menuconfig
```

Note the *ESP-NN* entry in *Menuconfig*; it allows the DL library configuration behavior. Selecting the *Optimized versions* option in the *Optimization for neural network functions* section is highly advisable because it activates the accelerator. The performance figures given in **Figure 8**, show the significant increase that can be expected with this enabled.

The compilation and execution will then proceed as expected. In **Figure 9**, you can see the output of the computed results.



```
I (292) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
x_value: 0.000000, y_value: 0.000000

x_value: 0.314159, y_value: 0.372770

x_value: 0.628319, y_value: 0.559154

x_value: 0.942478, y_value: 0.838731

x_value: 1.256637, y_value: 0.965812

x_value: 1.570796, y_value: 1.042060

x_value: 1.884956, y_value: 0.957340

x_value: 2.199115, y_value: 0.821787

x_value: 2.513274, y_value: 0.533738

x_value: 2.827433, y_value: 0.237217

x_value: 3.141593, y_value: 0.008472

x_value: 3.455752, y_value: -0.304993

x_value: 3.769912, y_value: -0.533738

x_value: 4.084070, y_value: -0.779427

x_value: 4.398230, y_value: -0.965812

x_value: 4.712389, y_value: -1.109837
```

Figure 9: Neural networks work with sine waves too.

Analysis of the TensorFlow Code

Don't be surprised if you see chaotic images on the ESP-EYE display — the built-in LCD has its own framebuffer controller, which just displays the last image stored until there is a new update.

If you open the *main.cc* file in the text editor of your choice, you'll find the following snippet:

```
#include "main_functions.h"

extern "C" void app_main(void) {
  setup();
  while (true) {
   loop();
  }
}
```

First impressions are not misleading here. TensorFlow is closely aligned with the Arduino environment. The actual implementation of the `setup()` and `loop()` functions can be found in the *main_functions.cc* file.

The `loop()` function is of particular interest because it is responsible for executing the payloads. Its first task is to generate input data that will be fed to the sine predictor:

```
void loop() {
  float position =
    static_cast<float>(inference_count) /
    static_cast<float>(kInferencesPerCycle);
  float x = position * kXrange;
```

In the next step, the information is quantified to make it more "digestible" for the neural network. This is a popular approach in the field of machine learning. Normalizing all input values to the range between 0 and 1 helps keep the resulting algorithmic complexity in check:

```
  int8_t x_quantized =
    x / input->params.scale +
    input->params.zero_point;
  input->data.int8[0] = x_quantized;
```

The actual calculation and quantification then occur in the following block:

```
  TfLiteStatus invoke_status =
      interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
   MicroPrintf("Invoke failed on x: %f\n",
    static_cast<double>(x));
   return;
  }
  int8_t y_quantized = output->data.int8[0];
  float y = (y_quantized -
        output->params.zero_point) *
        output->params.scale;
```

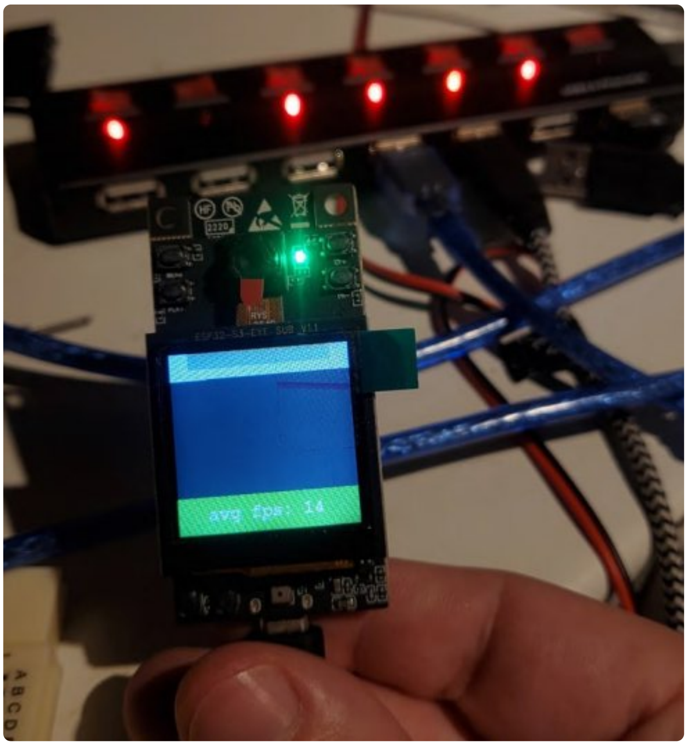


Figure 10: The green footer indicates successful recognition.

Finally, some housekeeping is necessary. It's interesting to note that according to the TensorFlow Micro documentation, the function responsible for the actual data output, `HandleOutput()`, needs to be provided by the developer:

```
  HandleOutput(x, y);
  inference_count += 1;
  if (inference_count >= kInferencesPerCycle)
      inference_count = 0;
}
```

Experiments Using More Advanced Modules

If you carefully analyze the *~/esp4/tflite-micro-esp-examples/examples* examples folder provided by Google, you'll notice that it includes a speech recognition example called *micro_speech* and even a facial recognition example called *person_detection*. To get these examples up and running, it's necessary to follow the familiar three-step procedure of parameterization, checking *menuconfig* settings, compilation, followed by uploading the machine code to the ESP32.

It's important to note that these advanced examples typically work using command-line communications. If you want to add screen output capability to the person detector, you need to adapt the *esp_main.h* file as follows:

```
// Enable this to do inference on embedded images
//#define CLI_ONLY_INFERENCE 1
#define DISPLAY_SUPPORT 1
```

After recompiling, the screen shown in **Figure 10** appears. Addressing the display issue would be the subject of another article. When the displayed footer turns green, it indicates a successful recognition.

Two Methodologies

The experiments conducted here demonstrate that the ESP32-S3 platform is capable of executing object and facial recognition payloads in various ways. While the ESP-WHO path produces quick and impressive results, the method of integrating it into the TensorFlow ecosystem enables the utilization of advanced techniques from the field of machine learning.

Translated by Martin Cooke — 230556-01

Related Products

- ESP32-S3-EYE
  www.elektor.com/20626

Questions or Comments?

If you have any technical questions, please feel free to contact the author using tamhan@tamoggemon.com or get in touch with the Elektor editorial team here at editor@elektor.com.

WEB LINKS

[1] ESP32-S3-EYE distributor search: https://oemsecrets.com/compare/ESP32-S3-EYE
[2] Schematic of the ESP32-S3-EYE: https://tinyurl.com/esp32eyeschematics
[3] ESP Deep Learning Library: https://github.com/espressif/esp-dl
[4] FreeRTOS Queues: https://freertos.org/a00018.html
[5] T. Hanna, "Audio Signals and the ESP32," Elektor 1-2/2023: https://elektormagazine.com/magazine/elektor-288/61449
[6] AI example ESP-WHO: https://github.com/espressif/esp-who/tree/master/components/modules/ai
[7] ESP32 camera control: https://github.com/espressif/esp32-camera
[8] TensorFlow: https://tensorflow.org
[9] TensorFlow Lite Micro: https://github.com/espressif/tflite-micro-esp-examples
[10] ESP32-S3-EYE Block diagram:
https://github.com/espressif/esp-who/blob/master/docs/en/get-started/ESP32-S3-EYE_Getting_Started_Guide.md
[11] ESP-WHO on GitHub: https://github.com/espressif/esp-who

ESP32-C2-Based Coin Cell Switch

Design and Performance Evaluation

By Li Junru and Zhang Wei, Espressif

The ESP32-C2-based coin cell switch is a solution for the smart home market. It offers direct communication with ESP-equipped devices, stable bidirectional communication, compact size, and up to five years of battery life. This article discusses software and hardware implementation, highlighting its benefits for modern smart homes and IoT applications.

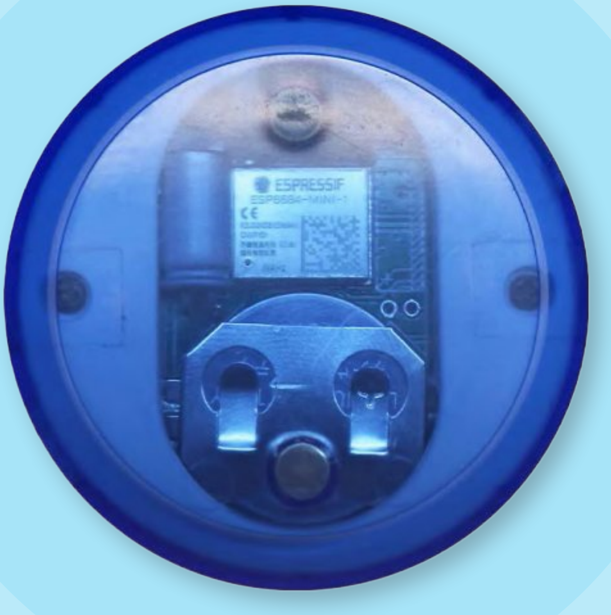


Figure 1: The ESP32-C2-based Coin Cell Switch in its enclosure.

With the rapid development of the smart home market, there is an increasing demand for low-power yet high-efficiency wireless switches. This article presents a cutting-edge solution, an ESP32-C2-based coin cell switch (**Figure 1**), aiming to tackle challenges such as delayed response and the need for additional gateways, which are commonly faced by other wireless switch solutions based on technologies like Bluetooth LE and ZigBee.

The ESP32-C2-based coin cell switch solution boasts several advantages over other wireless switch alternatives:

- Directly communicates with smart lights or wall switches equipped with ESP chips, eliminating the need for an extra gateway.
- Ensures stable bidirectional communication, guaranteeing a high success rate for Wi-Fi packet transmission.
- Powered by button batteries, the device's size is minimized, supporting versatile product form factors like adhesive switches, multi-key switches, touch switches, and rotary switches.
- The ESP32-C2 remains in a complete power-off state when not in use, enabling a single CR2032 button battery to last up to 5 years (assuming 10 presses per day).

Throughout this article, we delve into the comprehensive implementation details of the ESP button battery switch, showcasing its prowess in addressing the demands of modern smart home technology.

Button batteries, such as the commonly used CR2032, serve as a prevalent power source for IoT devices. Known for their compact design and lightweight, button batteries are well-suited for small electronic devices, without adding bulk or weight. Their relatively high-energy density allows them to store more energy in a smaller volume, resulting in an extended usage lifespan. Additionally, button batteries exhibit a lower self-discharge rate, meaning they retain their charge even during prolonged periods of non-use. Providing a relatively stable voltage output during discharge, button batteries play a vital role in ensuring the normal operation of devices. However, due to their design, they typically offer lower current output, making them unsuitable for high-power devices.

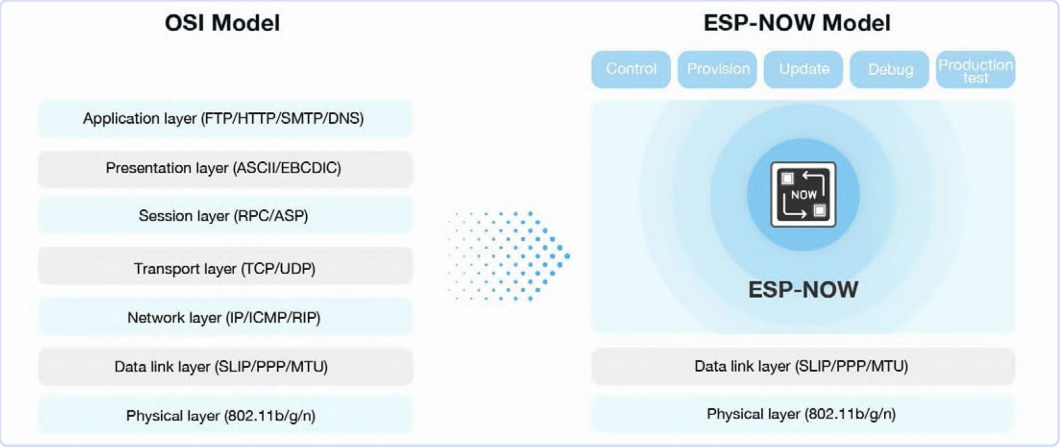

OSI Model
Application layer (FTP/HTTP/SMTP/DNS)
Presentation layer (ASCII/EBCDIC)
Session layer (RPC/ASP)
Transport layer (TCP/UDP)
Network layer (IP/ICMP/RIP)
Data link layer (SLIP/PPP/MTU)
Physical layer (802.11b/g/n)
ESP-NOW Model
Control
Provision
Update
Debug
Production test
NOW
ESP-NOW
Data link layer (SLIP/PPP/MTU)
Physical layer (802.11b/g/n)

Figure 2: The ESP-NOW Model simplifies the top five layers of an OSI Model into one layer.

In the diverse fields of application, Wi-Fi IoT devices have seen extensive use, benefiting from the widespread coverage of Wi-Fi networks. By eliminating the need for traditional wired connections between devices, Wi-Fi technology enables a more flexible and convenient deployment and management of IoT devices. Moreover, Wi-Fi technology supports simultaneous connections to multiple devices, making it particularly valuable for IoT scenarios involving numerous interconnected devices. The market offers a plethora of devices and components that support Wi-Fi, significantly reducing the development and production costs of IoT devices. Nonetheless, it is important to consider that Wi-Fi devices consume relatively higher energy compared to other low-power wireless technologies. Consequently, certain IoT devices relying on battery power might find Wi-Fi less suitable. For instance, the ESP32-C2 exhibits a maximum transmission current of 370 mA and a maximum reception current of 65 mA during RF operation.

The power consumption characteristics of Wi-Fi devices pose two main challenges when applied in low-power domains. On one hand, the significant reception current makes it difficult for the chip to sustain continuous operation in the receiving state. On the other hand, the instantaneous high current during packet transmission can impact the voltage stability of the chip's power supply, potentially leading to chip resets. In this article, we present a button battery switch based on the ESP32-C2 that tackles these challenges by employing a well-balanced combination of software and hardware, resulting in impressive battery life.

The article is divided into three main sections. Firstly, we delve into the software implementation, providing detailed insight into the program's methodology and its corresponding outcomes. Secondly, we explore the matched hardware implementation, offering guidance on device selection. Lastly, in the experimental section, we evaluate the actual power performance and packet latency of the device, providing readers with a comprehensive assessment of this solution.

Program Design

Protocol Layer Selection: The first crucial decision was choosing the appropriate protocol layer. Conventionally, Wi-Fi operates in a connected mode, where devices maintain a constant connection to the network unless intentionally disconnected or out of range. This continuous connection enables devices to be readily available for communication without the need to reconnect each time they are used. However, maintaining this connection consumes significant energy.

To address this challenge, we opted for a more lightweight communication solution. In the software aspect, we achieved this by broadcasting messages at the data link layer, allowing the receiving devices to easily retrieve and forward the information. Additionally, we implemented a predetermined reception time to minimize power consumption as much as possible. For this purpose, the ESP-NOW protocol developed by Espressif Systems proved to be an ideal fit. ESP-NOW is a wireless communication protocol based on the data link layer. It simplifies the upper five layers of the OSI model into one layer (**Figure 2**).

Device identification is determined through a unique identifier (MAC address), eliminating the need for data to pass through complex layers such as the network layer, transport layer, session layer, presentation layer, and application layer in sequence. It also eliminates the need to add and remove headers at each layer, greatly alleviating network congestion-related lag and delays caused by packet loss, resulting in higher response speeds. Additionally, ESP-NOW can coexist with Wi-Fi and Bluetooth LE, and it supports Espressif's multiple series of SoCs that have Wi-Fi functionality.

Workflow: To ensure reliable communication, we implemented an acknowledgment mechanism at the application layer (**Figure 3**). The communication process begins with the

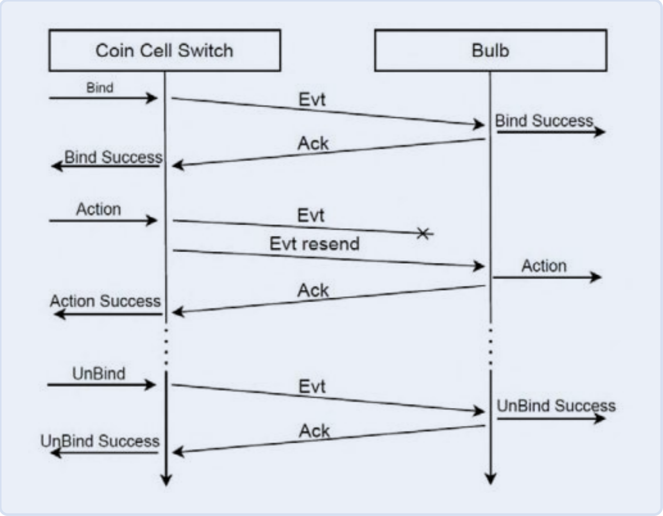


Figure 3: The acknowledgment mechanism implemented at the application layer.

Table 1: Energy consumption before initialization process optimization.

Action	Duration (ms)	Average power (mW)	Energy consumption (mJ)
Boot	364.7	58	21.16
Wi-Fi Init	115.2	69.8	8.03
Wi-Fi Start	56.6	303.9	17.2

Table 2: Energy consumption after initialization process optimization.

Action	Duration (ms)	Average power (mW)	Energy consumption (mJ)
Boot	37.52	53.14	2.0
Wi-Fi Init	6.55	72.62	0.48
Wi-Fi Start	19.12	164.0	3.13

Coin Cell Switch initiating the communication. Upon receiving a message, the receiving end responds with an ACK, indicating the successful reception of the message, and the communication is concluded. If the acknowledgment is not received within a specified timeout period, the switch will attempt to retransmit the message several times. To maintain a streamlined protocol, all packets are broadcasted using the ESP-NOW protocol at the data link layer.
Device binding and unbinding are accomplished through bidirectional MAC address verification, ensuring the security and simplicity of the protocol design.

Channel Switching Strategy: The channels for ESP-NOW reception and transmission follow the channel of the currently connected AP (Access Point) of the device. The Coin Cell Switch can only communicate with the controlled device when they are on the same channel. When the controlled device is already connected to the specified Wi-Fi AP, its operating channel follows changes in the AP's channel. The Coin Cell Switch needs to determine the current operating channel of the controlled end. In an environment where the surrounding network conditions are relatively stable, the AP's channel does not frequently switch.

Therefore, the device first sends on the channel where it successfully sent data the last time. If multiple send attempts fail and it is determined that the receiving end has switched to a new channel for operation, then it will iterate through all the remaining Wi-Fi channels.

Packet Transmission Current Modulation: During Wi-Fi packet transmission, the instantaneous current can exceed 300 mA, far beyond the maximum current that button batteries can handle. To address this, we implemented an intermittent packet transmission strategy. After each packet transmission, a certain interval is introduced, during which the chip enters a light sleep state. In this state, the chip consumes only a few tens of μA of current, and the button battery discharge is mainly used to charge the capacitors before the next packet transmission.

Let's assume that the average current during packet transmission is I_0, with a duration of t_0, and the average current during sleep is I_1, with a duration of t_1. The overall average current during the packet transmission process can be calculated as:

$$I = \frac{I_0 t_0 + I_1 t_1}{t_0 + t_1}$$

By carefully modulating the timing of I_0 and I_1, we can ensure that the average current remains below the safe output current of the button battery. This strategy helps to achieve a more efficient and energy-saving operation of the button battery switch.

Optimization of the Initialization Process: The initialization process of a Wi-Fi chip involves several stages, from power-up to the completion of signal transmission. We conducted a thorough analysis of each stage and its respective duration. By default, the chip startup includes the boot, Wi-Fi initialization, and Wi-Fi start processes. Among these, the boot initialization takes the longest time, and Wi-Fi start results in the highest short-term power consumption. To reduce the current consumption, we implemented the following optimizations:

> *The ESP32-C2 coin cell switch provides a convenient and versatile way to control smart devices.*

1. Disabling Non-Essential Logging: We turned off unnecessary logging outputs to minimize power consumption during the boot process.
2. Flash Verification: We disabled flash verification, as it was not essential for our operation.
3. Wi-Fi Calibration Information: To avoid frequent Wi-Fi calibration, we stored the Wi-Fi calibration information in NVS (nonvolatile storage).

Take a look at **Table 1** and **Table 2**. By implementing these optimizations, we managed to reduce the overall energy consumption (Boot + Wi-Fi Init + Wi-Fi Start) during the initialization process from 46.39 mJ to 5.61 mJ. Additionally, the initialization time decreased from 536.5 ms to 63.19 ms. For more detailed configuration information, please refer to the ESP-NOW coin_cell_demo [1].

Circuit Design

The ESP32-C2 requires a working voltage of 3.3 V, which is higher than the voltage provided by the button battery. Therefore, a boost circuit needs to be designed to step up the voltage. The stability of the power supply directly impacts the packet transmission performance and overall stability of the device. A well-designed voltage regulator circuit can enhance the RF performance and battery life of the device. It is essential to consider both production costs and the required performance when designing the circuit.

The voltage boost circuit should be carefully designed to ensure efficient power conversion with minimal power loss. Additionally, it should provide a stable and reliable power supply to the ESP32-C2, enabling it to perform optimally during both active and sleep modes. Moreover, the circuit design should consider factors such as current consumption, heat dissipation, and efficiency to strike the right balance between performance and power consumption.

The overall goal of the circuit design is to achieve a robust and cost-effective solution that meets the power requirements of the ESP32-C2, while also optimizing the device's performance and battery life. Collaborating with experienced hardware engineers and utilizing appropriate components and techniques can lead to a successful circuit design that meets the specific needs of the button battery-powered Wi-Fi switch.

Overall Circuit Design

Power Converter Selection: The button battery can be considered as a voltage source with rapidly increasing internal resistance as it discharges. Initially, its internal resistance is approximately 10 Ω, but it can increase to even hundreds of ohms as it nears the end of its discharge cycle. The ESP32-C2, as the output device, requires a power source capable of providing a current output of 500 mA or higher. The power supply ripple can significantly impact the RF (radio frequency) TX performance. When measuring power supply ripple, it is crucial to test it under normal packet transmission conditions.

The power supply ripple can vary depending on the power mode changes. Higher packet transmission power can lead to larger ripple effects. To mitigate the impact of the high internal resistance of the button battery, the boost converter's minimum input voltage requirement should be as low as possible, while maintaining high efficiency.

For this design, as you can see in the overall schematics in **Figure 4**, the SGM6603 was selected as the boost chip. It offers a minimum input voltage as low as 0.9 V and a maximum switch current of 1.1 A, making it suitable for efficiently boosting the voltage from the button battery to meet the ESP32-C2's power requirements.

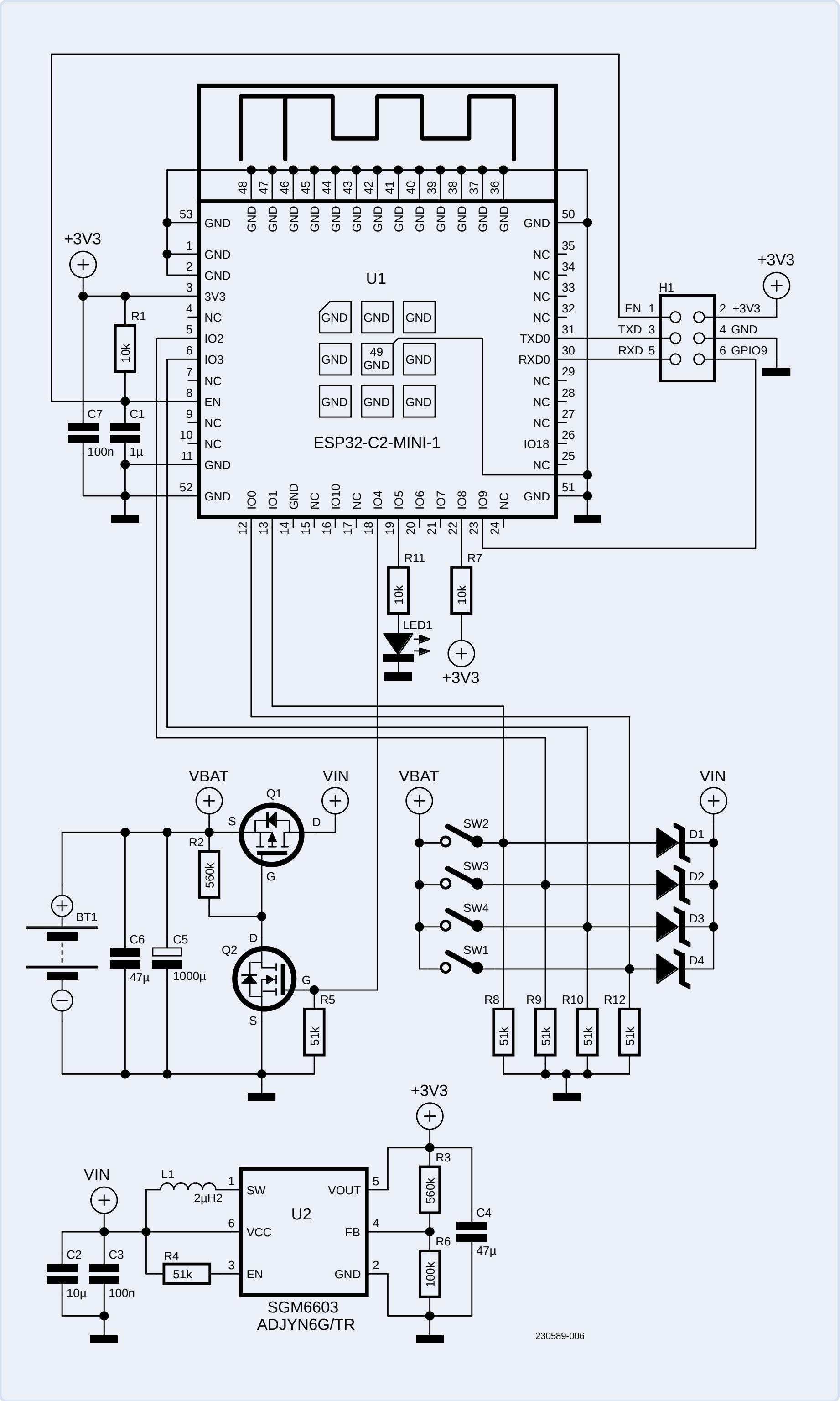


Figure 4: Schematic diagram of the coin cell switch.

Capacitor Selection: There are two sets of capacitors related to the power supply: the capacitors before and after the boost converter. The capacitors after the boost converter are typically connected in parallel with the Wi-Fi module's power input, providing the function of stabilizing the output voltage and reducing the voltage drop during packet transmission. Larger capacitors result in a smoother voltage change for the chip's power supply. On the other hand, the voltage across these capacitors follows the chip's power supply voltage. When the Wi-Fi chip is powered, the capacitors are charged, and when the chip is powered off, this set of capacitors is completely discharged. Therefore, using excessively large capacitors can lead to reduced system efficiency. It is essential to strike a balance between capacitor size and system efficiency.

The capacitors before the boost converter are connected in parallel with the battery. Their main function is to reduce the instantaneous current from the battery. During periods of high current, the capacitors act as the primary power source, while during periods of low

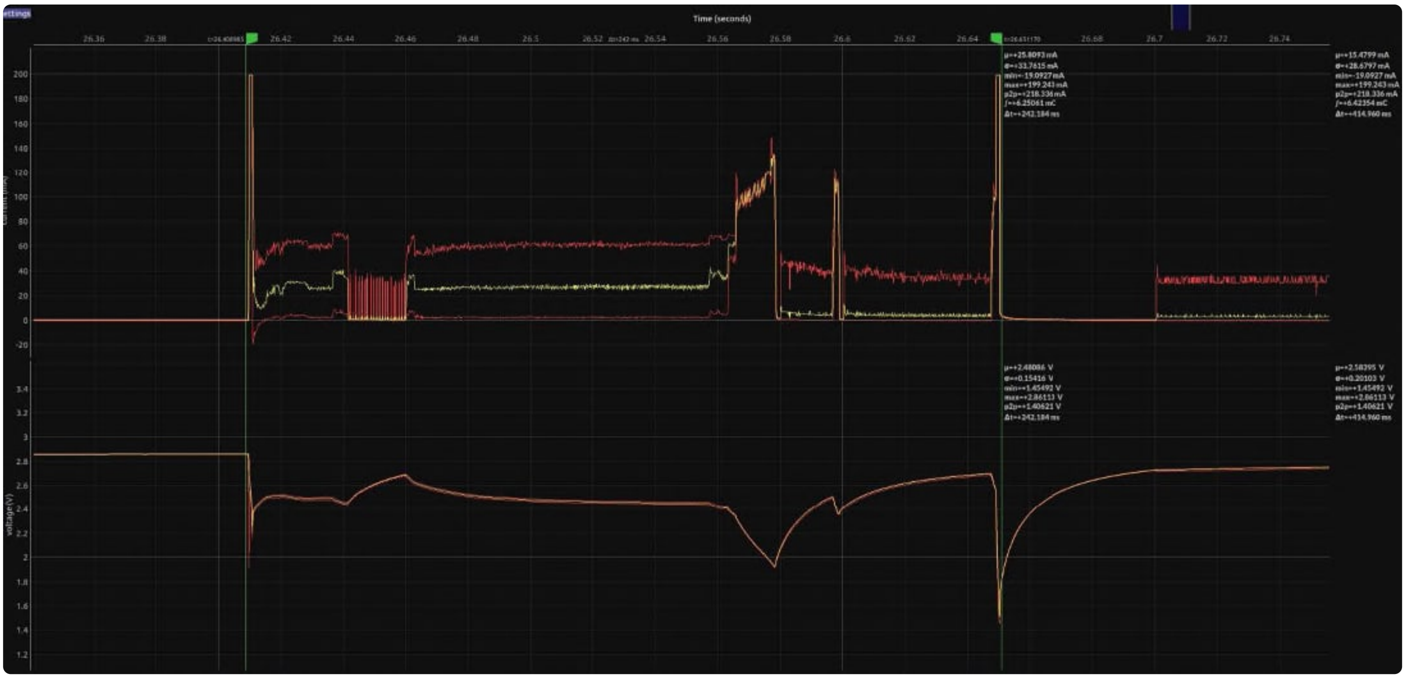


Figure 5: Scope screenshot of voltage and current values during the transmission of a standard data packet.

current, the battery becomes the main power source and charges the capacitors. When the boost converter is not operational, the only power consumption in the circuit comes from the capacitor's leakage current. Considering volume and leakage current, solid-state electrolytic capacitors and aluminum electrolytic capacitors are ideal choices. For instance, a 1000 μF solid-state electrolytic capacitor typically has a leakage current of around 1 μA at 3 V voltage.

Controlled Power Switch Design: In applications where the device's operational lifetime is measured in years, standby current (leakage current) during non-working states becomes a critical factor affecting the device's overall lifespan. To address this concern, the present design incorporates a controlled power switch consisting of two MOSFETs Q1 and Q2 in the schematic of Figure 4. This switch allows the chip to actively close or open the connection with the battery, effectively disconnecting the power supply module and RF module from the battery when the device is not in use. When the device needs to be turned on, a button is pressed, providing a conductive pathway for the chip to be powered on. At the same time, the chip utilizes ADC voltage sampling to identify which button has been pressed.

The utilization of this controlled power switch ensures efficient power management, reducing unnecessary energy consumption during idle periods and extending the overall battery life of the device. By completely disconnecting the power supply and RF modules when not in use, the device's standby current is minimized, optimizing its longevity and usability in various consumer electronic applications. In addition to the above-mentioned components, the circuit also includes the ESP32-C2 minimum system and an LED indicator light.

Battery Life Evaluation

After the final optimizations, a typical packet transmission process was analyzed, and the input-side voltage and current of the boost converter were recorded (**Figure 5**). Based on the provided information, the entire packet transmission time for the device is 240 ms, and the average current during operation is 25.8 mA.

To accurately assess the device's battery life, considering the uncertainty in the voltage converter's efficiency under dynamic load, a practical evaluation was conducted. In this evaluation, the device's overall packet transmission count was tested in the specified working mode. Each cycle was measured from the moment the device is triggered (button pressed) to the successful transmission of a signal and receiving an acknowledgment, encompassing the entire operation period.

After testing, it was found that a single CR2032 button cell battery could support approximately 65,000 packet transmission cycles. Additionally, the device's standby current was measured to be as low as 1 μA. Assuming the device transmits packets 10 times a day, the practical battery life for the CR2032 button cell is approximately 5 years.

This evaluation demonstrates the efficient power management and low-power consumption of the device, making it suitable for long-term use in consumer electronic applications. With the optimized power control and advanced circuit design, the device achieves an impressive battery life, ensuring reliable and extended operation in various IoT and smart home applications.

Final Considerations

The ESP32-C2 coin cell switch provides a convenient and versatile way to control smart devices. Leveraging the ESP32-C2's flexible power management and protocols, this solution realizes a coin cell switch based on the Wi-Fi protocol, allowing easy communication with other devices equipped with ESP chips. As part of our future plans, we aim to integrate this technology into Matter (formerly known as Project CHIP) standards, enabling flexible multi-device control and collaboration with conventional power-supplied devices.

Don't forget to check out Espressif's GitHub repository [2] for more open-source demos and information about ESP-IoT-Solution and ESP-NOW.

The coin cell switch solution enhances convenience and efficiency in the realm of smart homes and the Internet of Things. With its optimized design and efficient power management, it ensures long-lasting battery life, providing users with a reliable and enduring smart device control experience.

230589-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the authors (zhang.wei@espressif.com or lijunru@espressif.com) or the Elektor editorial team at editor@elektor.com.

Related Products

- **Espressif ESP32 range**
  www.elektor.com/espressif
- **Getting Started with ESPHome**
  www.elektor.com/19738

Getting Started with ESPHome
Develop your own custom home automation devices
Koen Vervloesem
elektor

About the Authors

Zhang Wei is currently a senior staff application engineer in Espressif Systems. A seasoned software engineer with over two decades of experience in embedded systems, wireless networking, and IoT development, he enjoys finding simple and clean solutions for solving problems. Holding a bachelor's degree in electrical and computer engineering and a master's degree in knowledge engineering from the National University of Singapore, he has enriched his expertise through roles at tech companies like STMicroelectronics, Greenwave Systems, and dormakaba digital department. Outside of work, Zhang Wei likes football and travel.

Li Junru works as an AE engineer at Espressif Shanghai. Embedded systems have always been his passion. He is on a mission to help enthusiasts unleash their creativity with ESP32. Li Junru's inspiring message: "Let's collaborate and make embedded systems development exciting together!"

WEB LINKS

[1] ESP Now coin_cell_demo: https://tinyurl.com/espnowcoincell
[2] Espressif's GitHub repository: https://github.com/orgs/espressif/repositories

The Smart Home Leaps Forward with matter

Unlocking Smart Home IoT Potential

By Kedar Sovani, Espressif

Consumers and developers often experienced frustration that came associated with connected products — in most cases siloed devices that require their own apps to configure and operate, offering neither a consistent user experience nor fully interoperable inter-device communication. The Matter protocol offers a secure way in which connected devices can be configured, discovered and operated by the end-user. Since its launch in October 2022, more than 1,000 products from many different companies are now Matter-certified.

Toward the end of 2022, the Matter smart home standard was launched — a result of years of collaborative development through major participants in the industry. Over the past several months, we have seen numerous devices being launched that are powered with this universal interoperable standard. In this article, let's take a look at the benefits that Matter offers, how far it has come, and where it's going.

The Pre-Matter Era

As consumers — as well as developers — of connected products, you might have experienced the frustration that came associated with connected products. This has often been characterized by siloed devices that require their own apps to configure and operate, offering neither a consistent user experience nor fully interoperable inter-device communication. The only unifying feature was the voice assistant skills from various ecosystems that had built-in support for these proprietary protocols.

On the device developer/manufacturer side, all this implied that the overall cost to implement a great product was much higher, since developers had to cater to compliance and certifications from various organizations.

Enter Matter

Various stakeholders in the smart home industry realized that these problems were holding the value and growth of the smart home back. This industrywide collaborative effort culminated in the launch of the Matter standard in late 2022.

So, what exactly does Matter offer? Matter offers a secure means by which connected devices can be configured, discovered, and operated by the end user.

Since its launch in October 2022, 1,000+ products have been Matter-certified, while the Connectivity Standards Alliance (CSA) boasts 300+ corporations that are now part of the collaborative effort toward building and adopting it. The large scale of the drive for Matter adoption has led to multiple connected devices having Matter support out of the box.

Matter-enabled devices can be configured and operated by entities that are commonly called Matter controllers (phones, speakers, displays). Multiple ecosystems, such as iOS, Android, Alexa, and SmartThings, already have incorporated Matter controller support within their baseline offerings. This allows consumers to start configuring and operating Matter-enabled connected devices without even having to purchase a separate Matter controller or install a separate phone app (**Figure 1**).

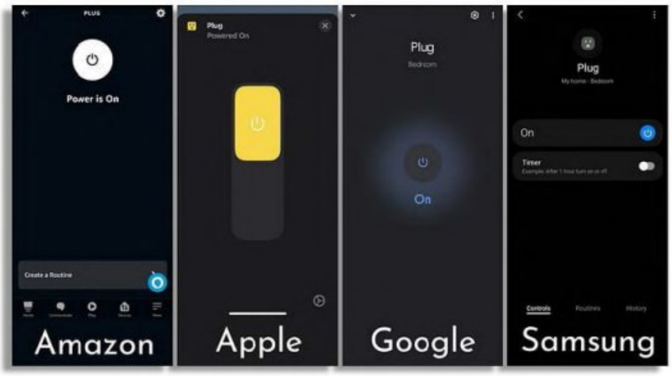


Figure 1: A Matter Plug as seen in various ecosystem apps.

Security and Privacy

For many connected devices we use today, as consumers we are left wanting for information about the security practices and principles deployed during development, and subsequently management, of the connected product.

Security and privacy were the backbone in building the Matter specification. As a large number of organizations (with high volumes and years of experience in the smart home industry) collaborated on these specifications, they ensured that no stone was left unturned to provide the best of this to users. Users of Matter products can rest assured that every aspect of the product's behavior, from initial configuration to subsequent operation and management, is built using the best security standards and practices.

Local Network

The architecture of Matter differs in one fundamental way from that of the connected devices before Matter. Previously any connected device was typically controlled over the local network by a device, such as a phone, on the same network. For all other scenarios, the devices connected to a cloud platform, and an app running on a phone not on the same network communicated with these devices via the cloud. Integrations with voice assistants also happened through cloud-to-cloud communication, as is shown in **Figure 2**.

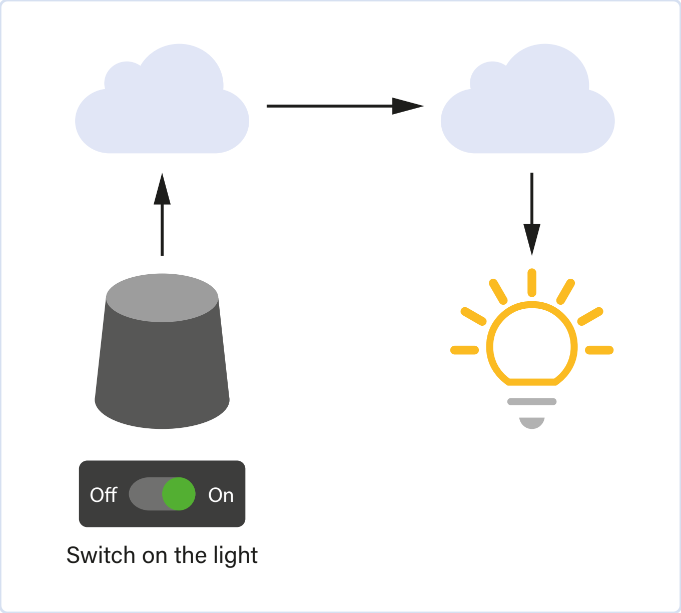


Figure 2: Sample Communication (Non-Matter).

Matter is a local-network-only protocol. It allows for the initial configuration, discovery, and operation of connected devices on the local network itself. The connected devices themselves are not required to talk to any cloud service, as was the case with earlier devices. Voice assistant integrations, therefore, work directly by sending the appropriate Matter commands over the local network (**Figure 3**). So, it's also left up to the Matter controllers to decide how to provide remote control for these devices.

This provides the end user with a choice in selecting the Matter devices and ecosystems that they trust with their privacy and security.

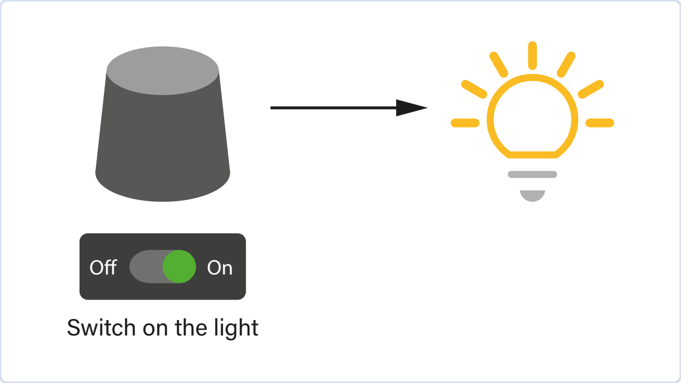


Figure 3: Sample Communication (Matter).

Transports

Matter offers its features over Wi-Fi and Thread (802.15.4) transport protocols, which are pervasive in the smart home. Each of these transports have their own advantages.

The Wi-Fi network, being ubiquitous, allows Matter Wi-Fi devices to be easily integrated into existing networks.

Thread-based (802.15.4) devices are better suited for low-power applications such as sensors, where the data duty cycles are much lower. Thread-based Matter devices do need a Thread border router to be part of the network. Most of the existing smart speakers/ecosystem devices have Thread support and are upgrading their software to support the border router feature. This makes integrating Matter devices in the home network much easier.

Commissioning

All connected devices need to be initially on-boarded to the user's network using some provisioning mechanism. This is a process often fraught with errors, misconfiguration, and resultant frustration on the user's part.

Matter devices are on-boarded using a process called Matter commissioning. Given the out-of-the-box support for Matter in most ecosystems, this process has been robust and refined, providing a smooth experience for any user. Under the hood, most Matter devices expose the commissioning functionality over Bluetooth Low Energy (BLE). Matter controllers will transfer the target network credentials (Wi-Fi or Thread) over a secure BLE link.

Devices may also offer Matter commissioning over Ethernet or Wi-Fi if they are already part of the user's network. This mechanism is crucial in allowing existing devices to be upgraded to expose Matter support to their end users.

Attestation

Device Attestation is an important Matter security feature. It's a process that happens during the Matter device's commissioning, where the Matter controller ascertains the authenticity of the device. Every Matter device is programmed with a unique set of manufacturer-specific security certificates. The device attestation process ensures that the device really came from the manufacturer it claims to be from. In case of a discrepancy, the end user is warned about the device authentication issue. This makes it difficult to create and deploy fakes of Matter products, thus protecting users and their data.

Device-to-Device Communication

Before Matter, most device-device communication was orchestrated through the cloud or by voice assistants. This required that all devices talk to the same cloud or provide some cloud-to-cloud integrations to allow for creation of value-added ecosystems that automatically do the correct thing.

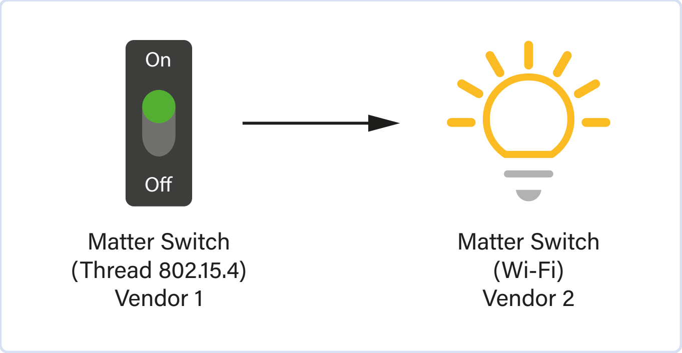


Figure 4: Matter Binding: Device-to-Device Communication.

One major advantage of Matter-enabled devices is inter-device communication completely within the local network, so no cloud communication is involved. A Matter device from one vendor can be set up to communicate with a Matter device from another. For example, you can set up automations to switch a group of lights (or an airconditioner) based on the state of a physical switch or a sensor. What's more, this can work even though the devices have different transports (one may have Wi-Fi while the other has Thread). Being able to set these automations (called "bindings" in Matter parlance) from any Matter-supported ecosystem or app makes it even easier to really deliver the value of the smart home (**Figure 4**).

Coexistence of Ecosystems

Another problem often faced in most homes is the heterogeneity of ecosystems used by occupants of a home. Most users have their own preferences in ecosystems, such as Android, iOS, Alexa, GVA, SmartThings, or Home Assistant. This leads to a fragmented user experience, where some connected devices can only be a part of one ecosystem or can only provide second-grade features to other ecosystems. In contrast, a Matter device can be part of multiple ecosystems at once. You can add a Matter device to up to five separate ecosystems. All ecosystem users can simultaneously enjoy the same benefits that Matter devices offer. Even notification of a change on one ecosystem is received by another.

Over-the-Air (OTA) Firmware Upgrade

Over-the-air (OTA) firmware upgrade allows a product to not only include the latest security or functionality fixes, but also to enable newer features over time. The Matter specification provides for a distributed ledger where manufacturers can upload and maintain OTA firmware upgrade images for their products. The distributed ledger is accessible to all Matter controllers, which can download the necessary firmware upgrades and deploy these upgrades on the Matter devices, with users' consent.

Certification

Matter has a strict device certification process. The presence of the Matter badge on a connected device indicates that the device has undergone and passed all the tests necessary to get the device Matter-certified. Matter certification helps guarantee that the device is interoperable with other Matter devices and controllers, and meets a minimum bar of robustness and functionality desired by consumers. Consumers of Matter products need no longer rely on the word of the manufacturer for these baseline requirements. All Matter manufacturers are held to the same standard of robustness and interoperability.

What's New With Matter 1.2?

Recently, Matter version 1.2 was released. The first version of Matter started with support for the most commonly available devices, such as lights, sockets, plugs, blinds, and thermostats. The latest Matter 1.2 standard incorporates support for appliances (white goods), including washing machines, refrigerators, airconditioners, dishwashers, and robot vacuum cleaners. Support for smoke/CO alarms, air-quality sensors, air purifiers, and fans is also included. This support includes additional specific commands for these devices beyond the typical on/off control.

The core Matter spec also includes upgrades for a more flexible composition of appliances from its sub-parts. This allows for more accurate modelling of a variety of appliances. The newly added support for semantic tags allows for an interoperable way to describe a device's location or semantic functions.

What About Developers?

The Matter standard is publicly available for download (after filling a form). The Matter C++ SDK is hosted on GitHub [1], with all the development happening on GitHub itself. The SDK implements both firmware-side (device) as well as controller-side functionality. Experimental implementations for JavaScript [2] and Rust [3] are also in the works. This allows developers to access the specifications easily and contribute any enhancements to the repository of interest.

Solutions such as ESP-Launchpad [4] and ESP-ZeroCode [5] allow developers to try out and get a feel for Matter's user experience easily by flashing the firmware to any of a variety of hardware development kits.

This allows developers to build and experience Matter for the vision of the device that they have easily. Once device firmware and hardware is available, all Matter ecosystems allow developers to evaluate these devices without requiring production firmware or production certificates. This makes iterating through Matter device development much easier.

For phone app developers, the latest version of iOS [6] and Android [7] include APIs that allow these apps to access the Matter APIs. App developers can use this feature to provide richer features and functionality than that offered by the base phone operating systems.

I hope this intrigues you, the developers, to experience Matter products yourself, and get started with Matter product development for your devices. ⏮

230617-01

Questions or Comments?

If you have questions about this interview, feel free to email the author at kedar.sovani@espressif.com or the Elektor editorial team at editor@elektor.com.

About the Author

Kedar Sovani has over 21 years of experience in the fields of systems software, security, and IoT. For the past six years, he's been working at Espressif. He's the Senior Director of Engineering, focusing on IoT Ecosystems, where his work helps in identifying and building platforms and solutions that help developers build IoT products faster, and in a more robust and secure manner. A hands-on developer, he is currently tinkering with Matter and Rust.

WEB LINKS

[1] Matter C++ SDK on GitHub: https://github.com/project-chip/connectedhomeip
[2] Implementation for JavaScript: https://github.com/project-chip/matter.js
[3] Implementation for Rust: https://github.com/project-chip/rs-matter
[4] ESP-Launchpad: https://espressif.github.io/esp-launchpad
[5] ESP-ZeroCode: https://zerocode.espressif.com
[6] Home Mobile SDK for iOS: https://developer.apple.com/documentation/matter
[7] Home Mobile SDK for Android: https://developers.home.google.com/matter/apis/home

Get your hands on new ESPRESSIF Hardware!

There is nothing that excites us more than getting our hands on new hardware, and so this collaboration with Espressif has been a treat! Want to experience the real deal yourself? Elektor has stocked up the stores to accommodate all products that are featured in this edition!

ESP32-S3-Box-3

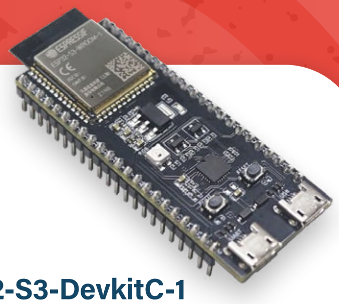
ESP32-S3-BOX-3 is a fully open-source AIoT development kit based on the powerful ESP32-S3 AI SoC, and is designed to revolutionize the field of traditional development boards. ESP32-S3-BOX-3 comes packed with a rich set of add-ons, empowering developers to easily customize and expand this kit's functionality.

www.elektor.com/20627

ESP32-S3-Eye

The ESP32-S3-EYE is a small-sized AI development board. It is based on the ESP32-S3 SoC and ESP-WHO, Espressif's AI development framework. It features a 2-Megapixel camera, an LCD display, and a microphone, which are used for image recognition and audio processing.

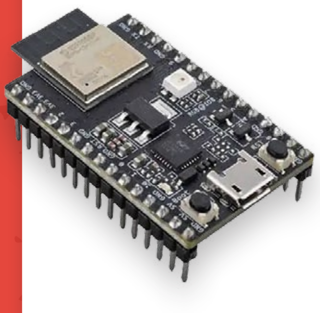
www.elektor.com/20626



ESP32-S3-DevkitC-1

The ESP32-S3-DevKitC-1 is an entry-level development board equipped with ESP32-S3-WROOM-1, ESP32-S3-WROOM-1U, or ESP32-S3-WROOM-2, a general-purpose Wi-Fi + Bluetooth Low Energy MCU module that integrates complete Wi-Fi and Bluetooth LE functions.

www.elektor.com/20697



ESP32-C3-DevKitM-1

ESP32-C3-DevKitM-1 is an entry-level development board based on ESP32-C3-MINI-1, a module named for its small size. This board integrates complete Wi-Fi and Bluetooth LE functions. Most of the I/O pins on the ESP32-C3-MINI-1 module are broken out to the pin headers on both sides of this board for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-C3-DevKitM-1 on a breadboard.

www.elektor.com/20324

ESP32-Cam-CH340

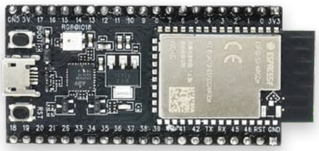
The ESP32-Cam-CH340 development board can be widely used in various Internet of Things applications, such as home intelligent devices, industrial wireless control, wireless monitoring, QR wireless identification, wireless positioning system signals and other Internet of Things applications.

www.elektor.com/19333

Elektor Cloc 2.0 Kit

Cloc is an is an easy to build ESP32-based alarm clock that connects to a timeserver and controls radio & TV. It has a double 7-segment retro display with variable brightness. One display shows the current time, the other the alarm time.

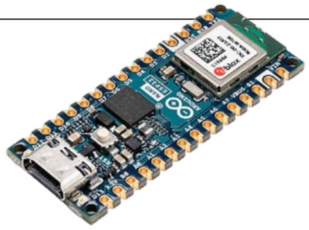
www.elektor.com/20438



ESP32-S2-Saola-1M

ESP32-S2-Saola-1M is a small-sized ESP32-S2 based development board. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-S2-Saola-1M on a breadboard. ESP32-S2-Saola-1M is equipped with the ESP32-S2-WROOM module, a powerful, generic Wi-Fi MCU module that has a rich set of peripherals.

www.elektor.com/19694



Arduino Nano ESP32

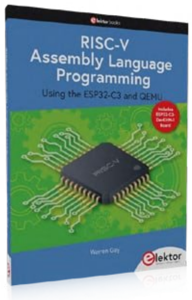
The Arduino Nano ESP32 is a Nano form factor board based on the ESP32-S3 (embedded in the NORA-W106-10B from u-blox). This is the first Arduino board to be based fully on an ESP32, and features Wi-Fi, Bluetooth LE, debugging via native USB in the Arduino IDE as well as low power.

www.elektor.com/20562

MakePython ESP32 Development Kit

The MakePython ESP32 Kit is an indispensable development kit for ESP32 MicroPython programming. Along with the MakePython ESP32 development board, the kit includes the basic electronic components and modules you need to begin programming. With the 46 projects in the enclosed book, you can tackle simple electronic projects with MicroPython on ESP32 and set up your own IoT projects.

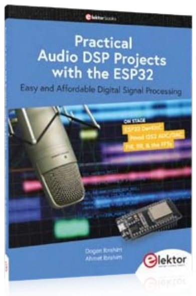
www.elektor.com/20137



RISC-V Assembly Language Programming using ESP32-C3 and QEMU (+ FREE ESP32 RISC-V Board)

The availability of the Espressif ESP32-C3 chip provides a way to get hands-on experience with RISC-V. The open sourced QEMU emulator adds a 64-bit experience in RISC-V under Linux. These are just two ways for the student and enthusiast alike to explore RISC-V in this book. The projects in this book are boiled down to the barest essentials to keep the assembly language concepts clear and simple.

www.elektor.com/20296



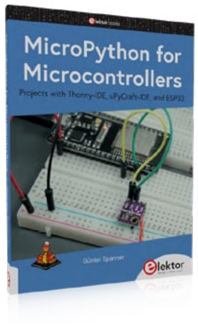
Practical Audio DSP Projects with the ESP32

The aim of this book is to teach the basic principles of Digital Signal Processing (DSP) and to introduce it from a practical point of view using the bare minimum of mathematics. Only the basic level of discrete-time systems theory is given, sufficient to implement DSP applications in real time. The practical implementations are described in real-time using the highly popular ESP32 DevKitC microcontroller development board.

www.elektor.com/20558

MicroPython for Microcontrollers

Powerful controllers such as the ESP32 offer excellent performance as well as Wi-Fi and Bluetooth functionality at an affordable price. With these features, the Maker scene has been taken by storm. Compared to other controllers, the ESP32 has a significantly larger flash and SRAM memory, as well as a much higher CPU speed. Due to these characteristics, the chip is not only suitable for classic C applications, but also for programming with MicroPython. This book introduces the application of modern one-chip systems.



www.elektor.com/19736

Where Is **Smart Home IoT** Headed?

By Amey Inamdar, Espressif

While connectivity technology has evolved and has become more accessible, secure, and cost-effective, for the vision that was painted over the last decade, the smart home is just at the starting line. We are seeing a proliferation of many devices, including smart thermostats, security systems, smart appliances, intelligent lighting, and voice assistants. Still, in terms of adoption, the technology has a long way to go. Recently, there have been two main advancements that hold the potential to accelerate adoption, and we are going to discuss those and the effect in this article.

Power consumption, cost, ease of development, and connectivity options have been important considerations when building any smart home device. If we look around, many of these problems have been solved to a great extent. The question then is, what's holding up the wide adoption of these devices? Among multiple answers to this question, probably the most important is the perceived value of smart home devices — what use cases they unlock. Are these devices smart, or do they just provide the ability for remote control? Another key reason could be privacy and security concerns. Voice has become a natural interface to interact in the smart home, but it comes at a price of perceived loss of privacy.

That's where the two key advancements that happened recently offer a glimmer of hope. We are seeing the great transformation that generative AI and large language models (LLMs) are bringing to different fields. Large language models have the potential to improve the efficiency and autonomy of smart home devices, decentralizing decision-making and bringing it to the edge. These LLMs also have the potential to contribute to better privacy. As an example, if we look at the current voice interface, it is either driven by cloud-based inferencing or a fixed command-based voice interface requiring the user to remember the exact commands on which it can operate. But, with LLMs providing offline models such as Whisper.ai to understand different languages and inference locally, they hold the potential to provide natural voice interfaces for smart homes that can work completely offline. We already see a few open-source projects moving in this direction.

This advancement in generative AI is being complemented by having tiny microcontrollers becoming better at running ML models on the edge. It is not uncommon now to see microcontrollers having power-efficient AI acceleration engines that can accelerate ML models' execution. This enables low-power sensors that can not only sense the data but also process it to extract meaning from the data.

However, just having this intelligence available for smart home IoT devices is not going to help if all the devices don't speak the same language. That's where standardization is very much needed. Lack of standardization is one of the key issues for the mass adoption of smart home IoT devices currently. Having no way for different vendor devices to talk to each other limits the use cases for the consumer, making it painful to configure and operate smart home devices. This is where the recent standardization efforts become important. A protocol such as Matter, when successful, can provide the ability for devices to talk the same language, enabling a more consumer-centric smart home with better use cases and user experience. One more indirect impact of standardization is that it unlocks developers to create value at a higher level than just what device manufacturers provide.

Thus, AI and standardization are both important in solving the key concerns of the value and privacy of smart home devices. This must be assisted with innovation in connectivity, sensing, and computing, to enable the proliferation of cost-effective and secure smart home devices. We've witnessed many such advancements. For example, the Wi-Fi 6 standard has the ability to create battery-operated connected devices without compromising on bandwidth. The rise of mmWave and UWB sensors provides much better occupancy sensing without requiring privacy-intruding cameras in all places. Cybersecurity labeling schemes are also gaining traction and various regions and countries are coming up with a clear way to label smart home devices with a security star rating that can help consumers make an informed choice.

There is no magic bullet for making a pervasive smart home a reality. But, we are most likely heading in a better direction now than we were before. It is certainly an interesting time and domain to watch how it evolves and play our respective parts to make the world better.

230655-01

empowered
connectivity
everywhere

The Next Era of
Microcontrollers

ESPRESSIF

High Performance MCU

With RISC-V Dual-Core Upto 400MHz

AI
Acceleation

High-Speed
MEMORY

Powerful
IMAGE & VOICE
Processing Capabilities

HMI Capabilities

- MIPI-CSI with ISP
- MIPI-DSI - 1080P
- Capacitive Touch
- H.264 Encoding - 1080P@30fps
- Pixel Processing Accellerator

Best-in-Class Security

- Cryptographic Accelerators
- Secure Boot, Flash Encryption
- Private Key protection
- Access Controls

Connectivity

- USB2.0 High Speed
- Ethernet
- SPI
- SDIO3.0
- UART
- I2C, I2S

......

IP Camera

Touch Panel

Video Door bell

Robotic Control

Industrial Robot

www.espressif.com

Learn More About
ESP SoCs