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?

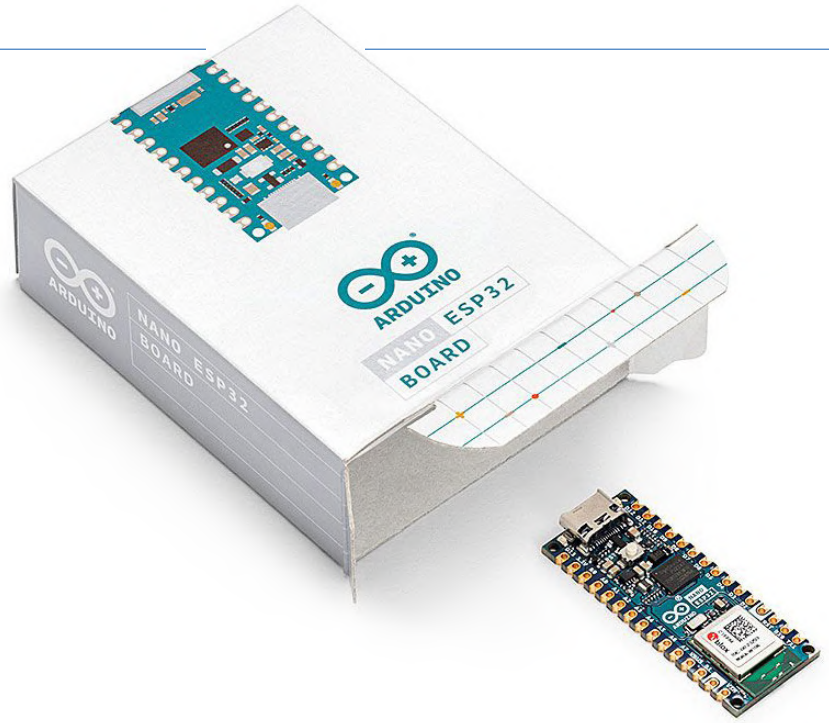
Martino Facchin: No, you can debug this board directly via USB — Espressif was kind enough to provide this. Over USB, you have one serial port, CDC, the usual stuff, and also a JTAG interface. You can interact with the JTAG interface using just the normal USB alongside normal communication. It doesn't 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

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

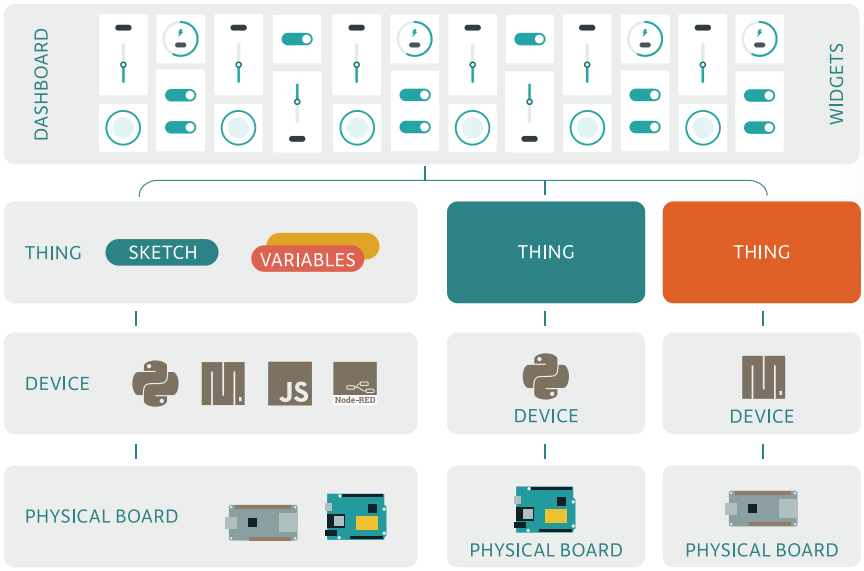
Program/Deploy

- **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

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

Third party platform integration

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.

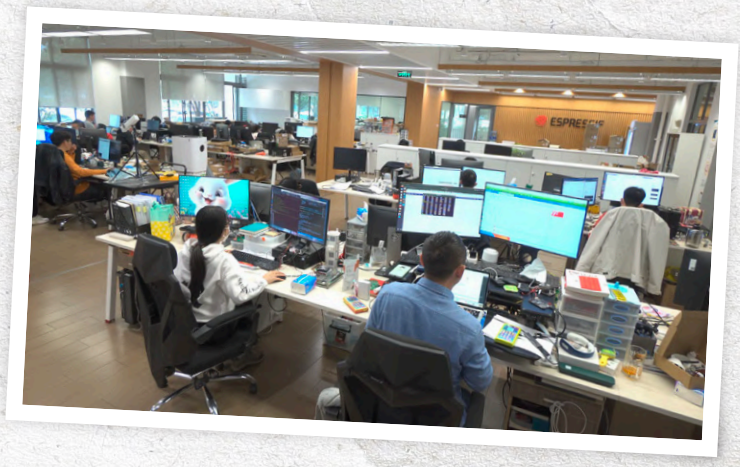
Seamlessly integrate your IoT devices with over 2 000 apps, enabling tasks like recieving phone notifications, automating social media updates, streamlining data logging to external files, creating calendar events, or sending e-mail alerts.

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

cloud.arduino.cc/elektor

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)

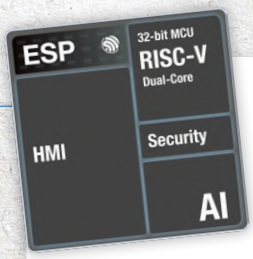
Engineers are a group that really digs deep into the technical details, so we must find that sweet spot between being accurate and keeping things understandable. Our goal is to show them how our products fit right into their specific needs and challenges. They are all about research, so we make sure 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.

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.

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.



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)

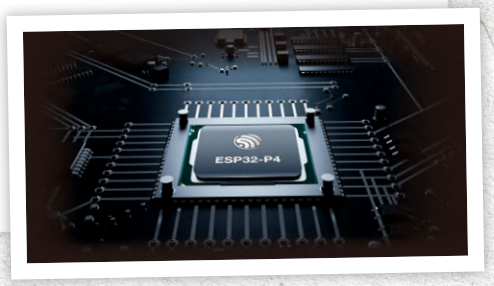
We continually come across numerous captivating projects within our inventive community and the industry. Indeed, this edition features a selection of these remarkable endeavors, spanning both hardware (HW) and firmware (FW). Among them are ESP32-based evaluation boards, no larger than coin cell batteries, 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.

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.

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.

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
```

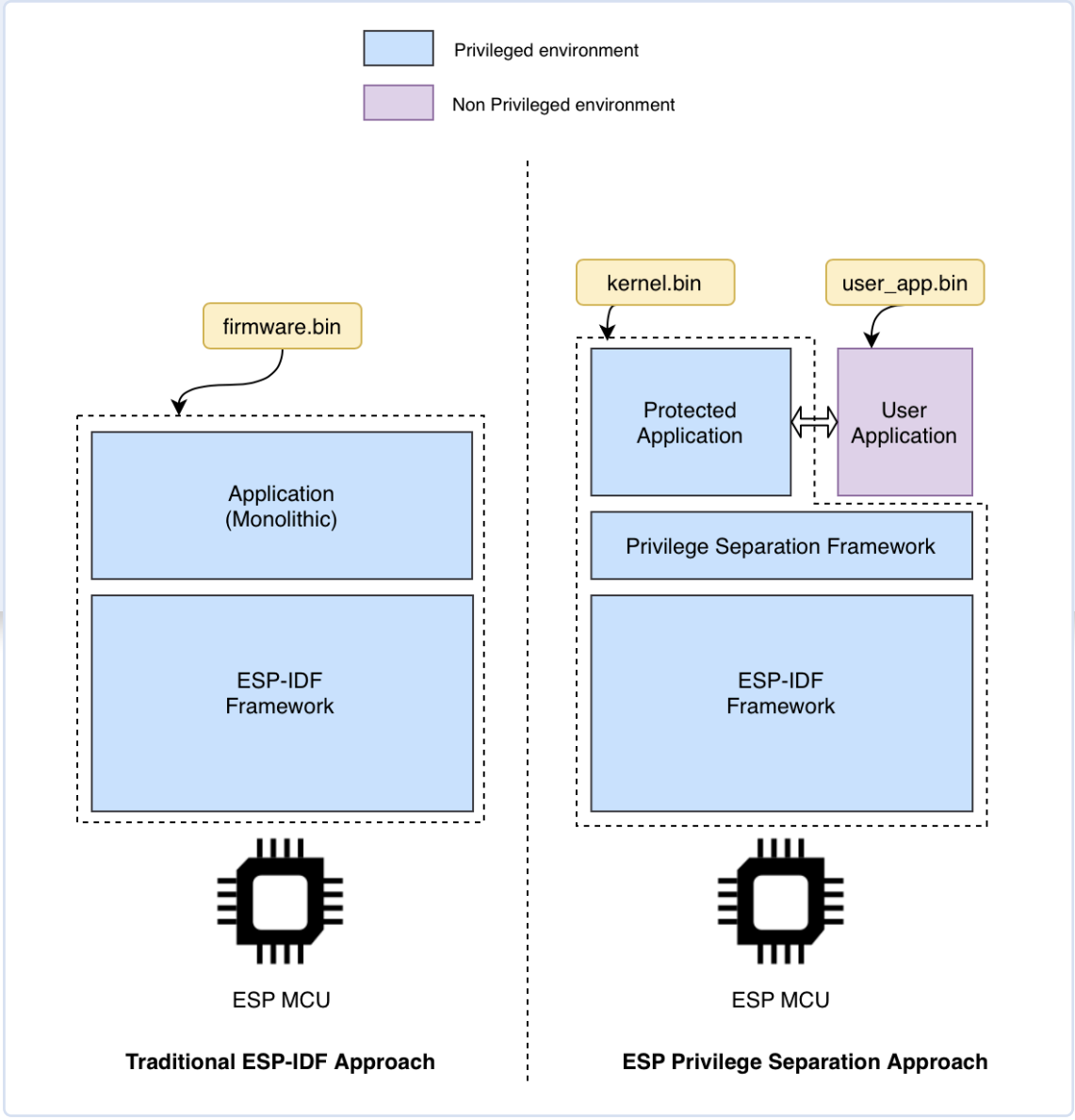


Figure 1: Traditional ESP-IDF built as a single monolithic firmware versus the ESP Privilege Separation framework, where the firmware image is split. (Source: Espressif)

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?

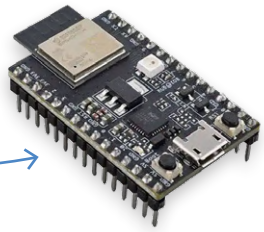
If you have technical questions or comments about this article, feel free to contact the author at harshal.patil@espressif.com or the Elektor editorial team at editor@elektor.com.

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

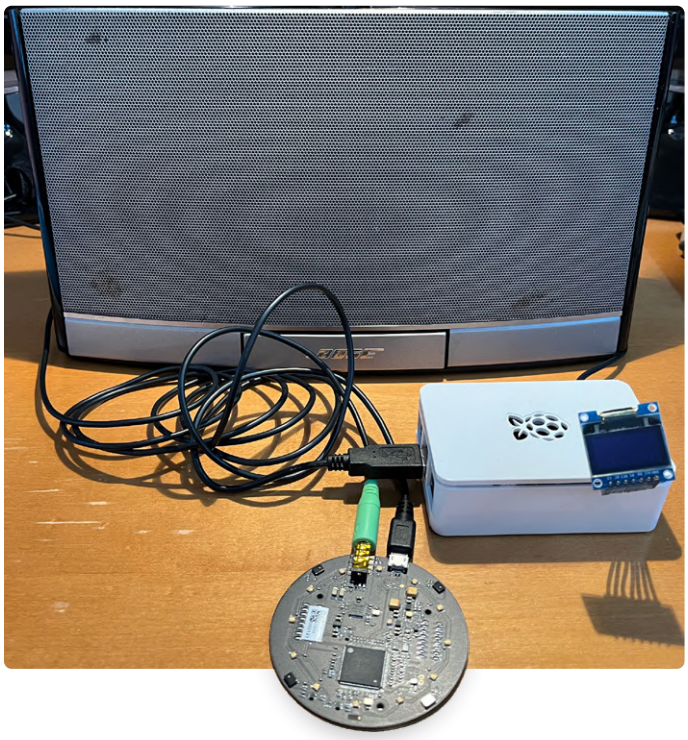


Figure 1: From the author's prior attempts with Raspberry Pi.

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×

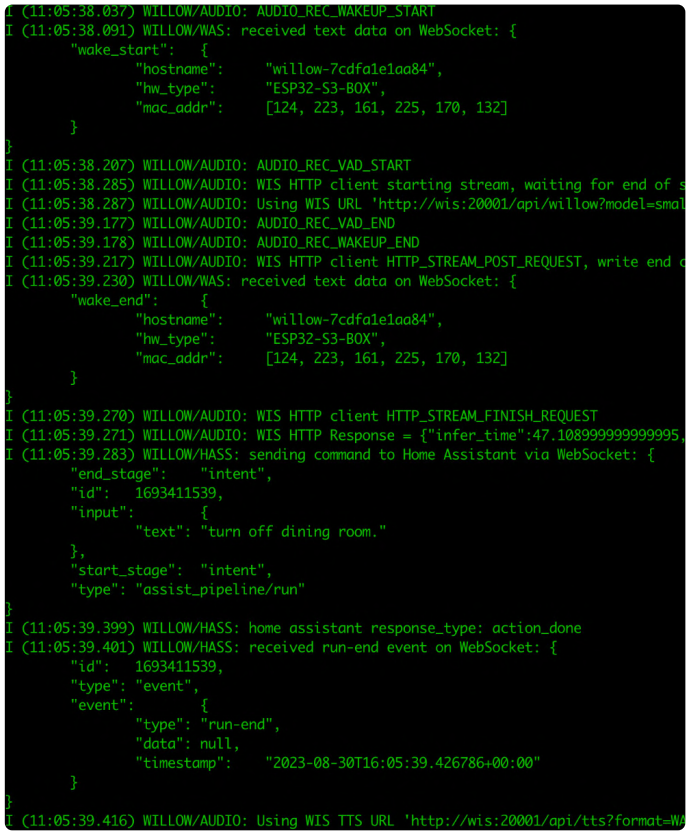


Figure 2: Willow speech recognition performance.

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.

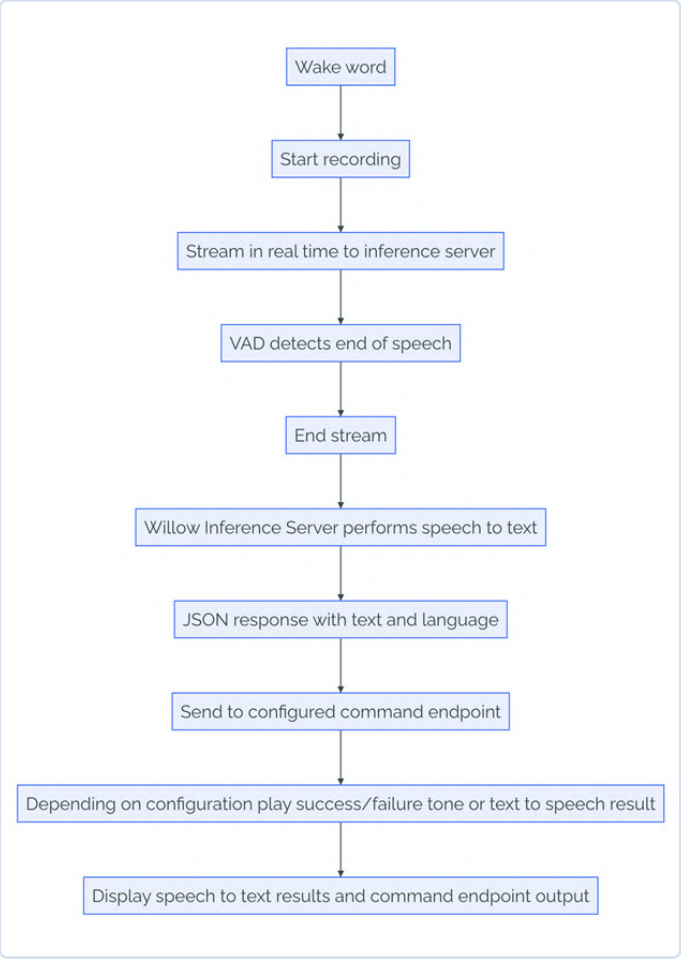


Figure 4: Willow inference server flow.

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.

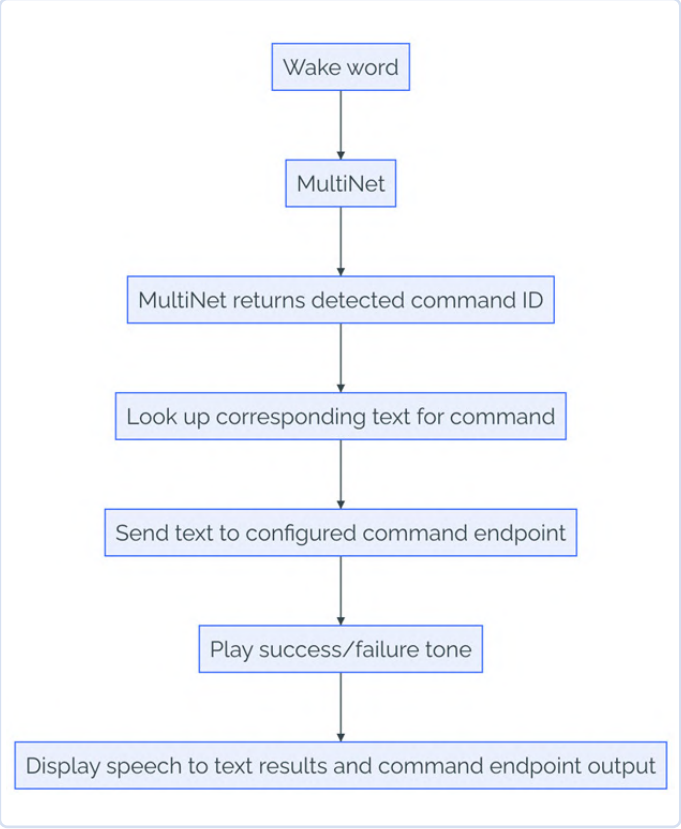


Figure 5: MultiNet mode flow.

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.

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.

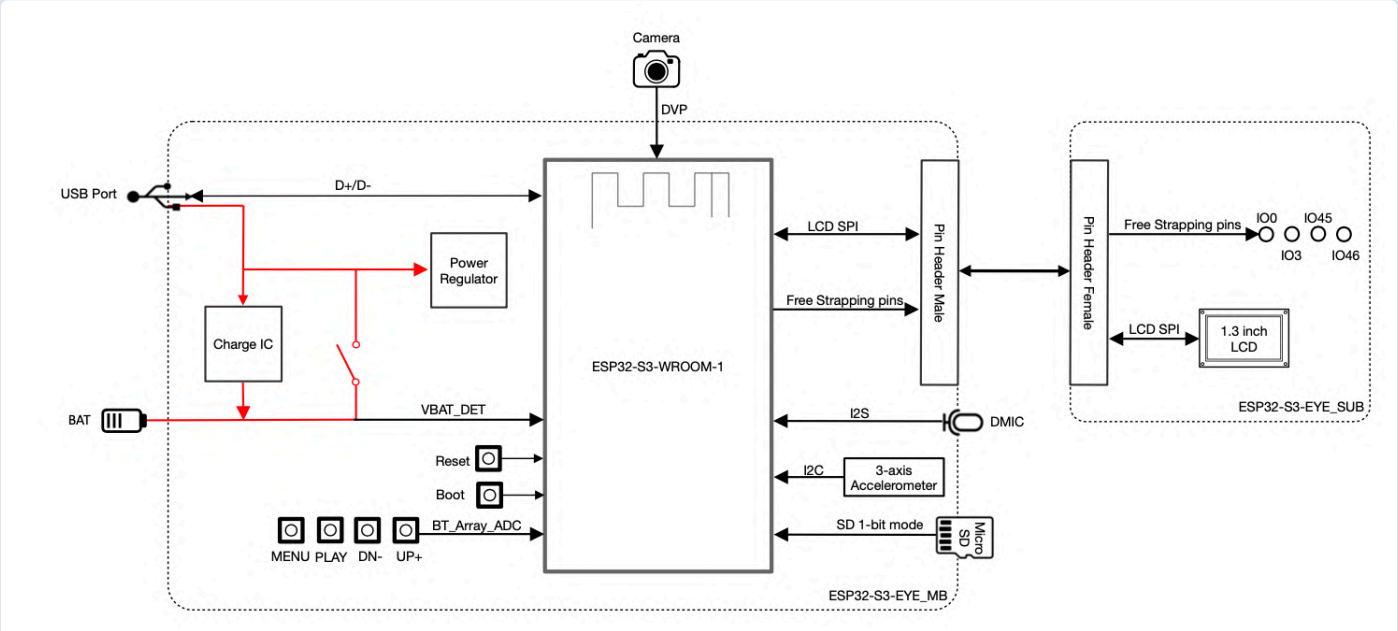


Figure 1: The ESP32-S3-EYE block diagram (Source: [10]).

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.

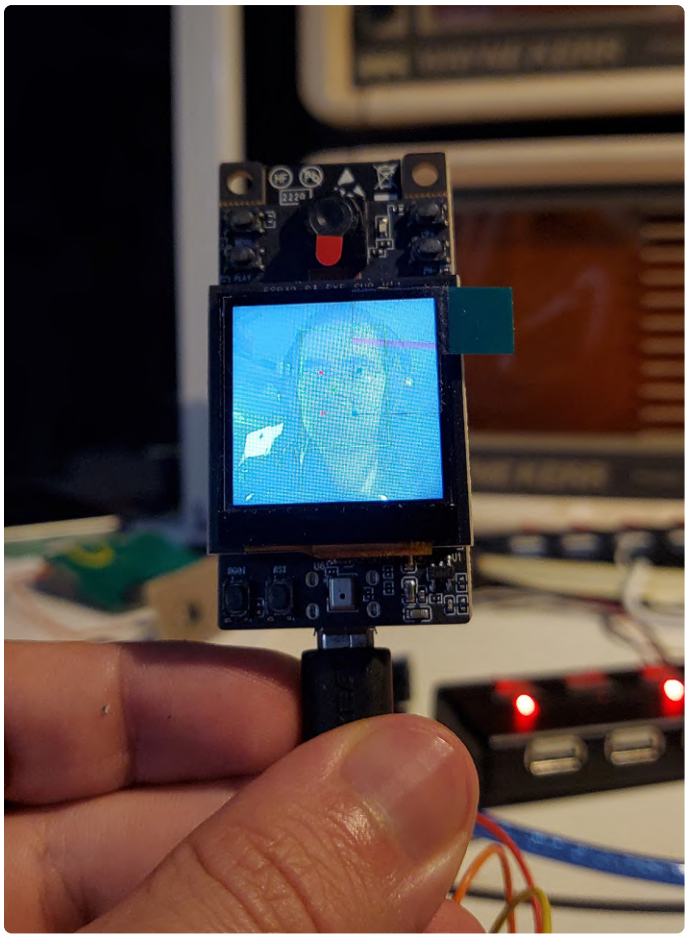


Figure 2: The camera worked, even in my dingy lab with the lens tape still in place!

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**.

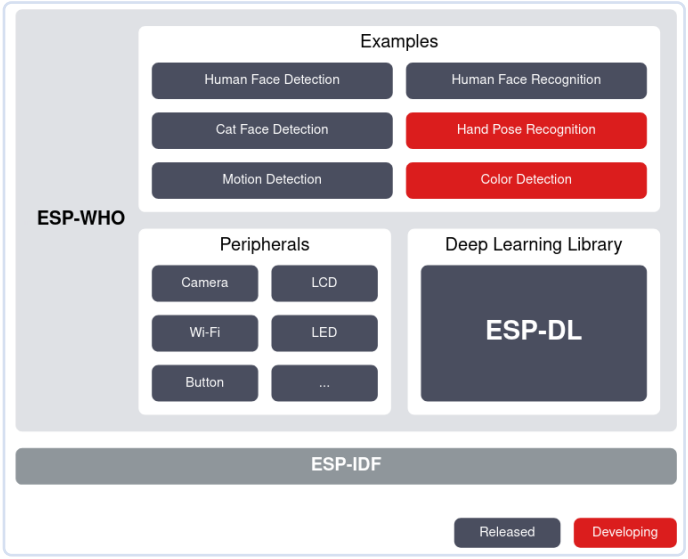


Figure 3: ESP-WHO uses various other parts of the Espressif ecosystem (Source: [11]).

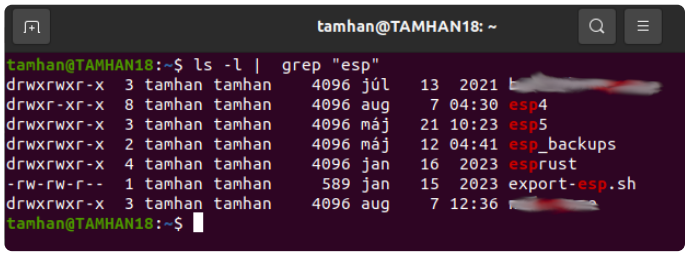


Figure 4: Different versions of the ESP-IDF can generally coexist on a workstation.

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**.

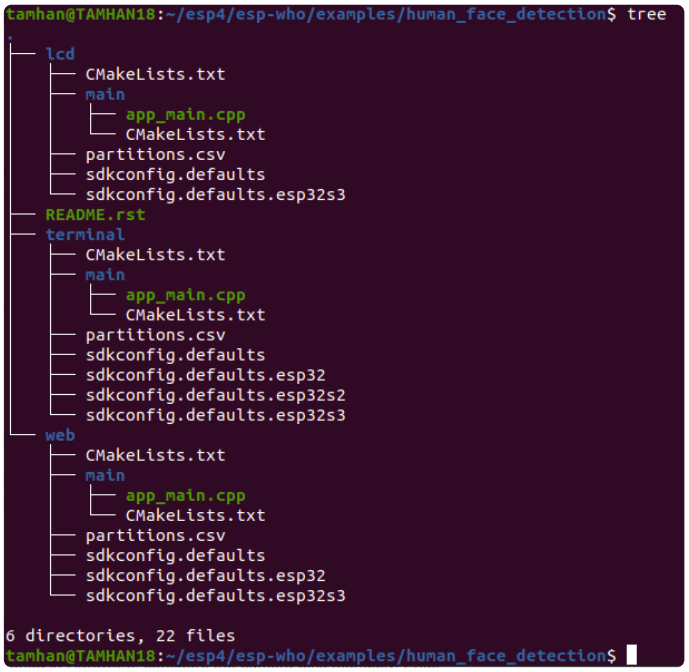


Figure 5: ESP-IDF variants in the house!

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

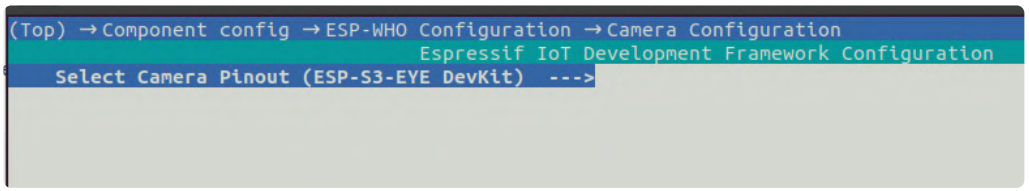


Figure 6: If you use an in-house camera module, you need to reconfigure settings here!

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;
```

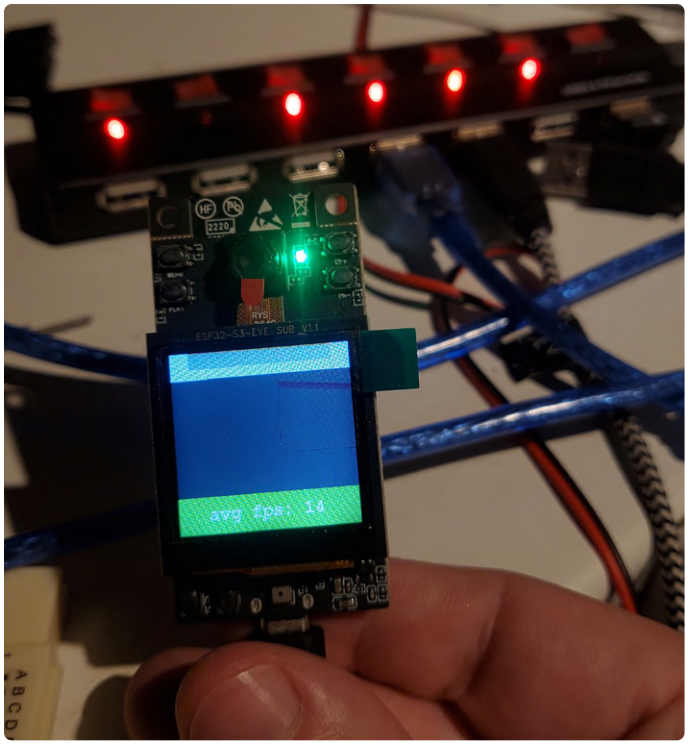


Figure 10: The green footer indicates successful recognition.

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.

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

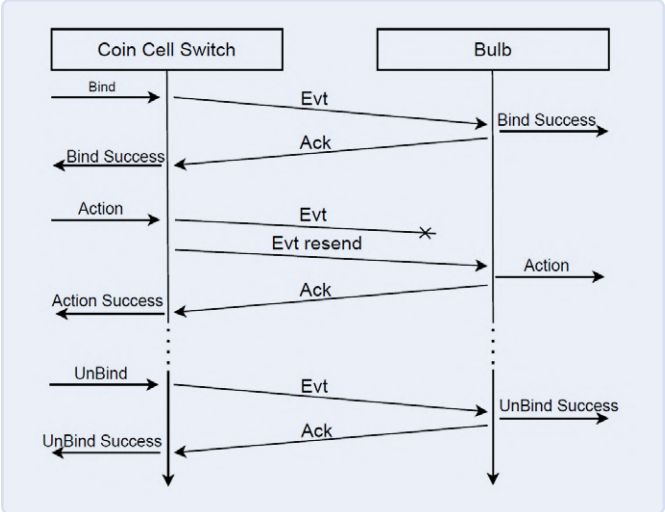


Figure 3: The acknowledgment mechanism implemented at the application layer.

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:

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 ESP32-C2 coin cell switch provides a convenient and versatile way to control smart devices.

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.

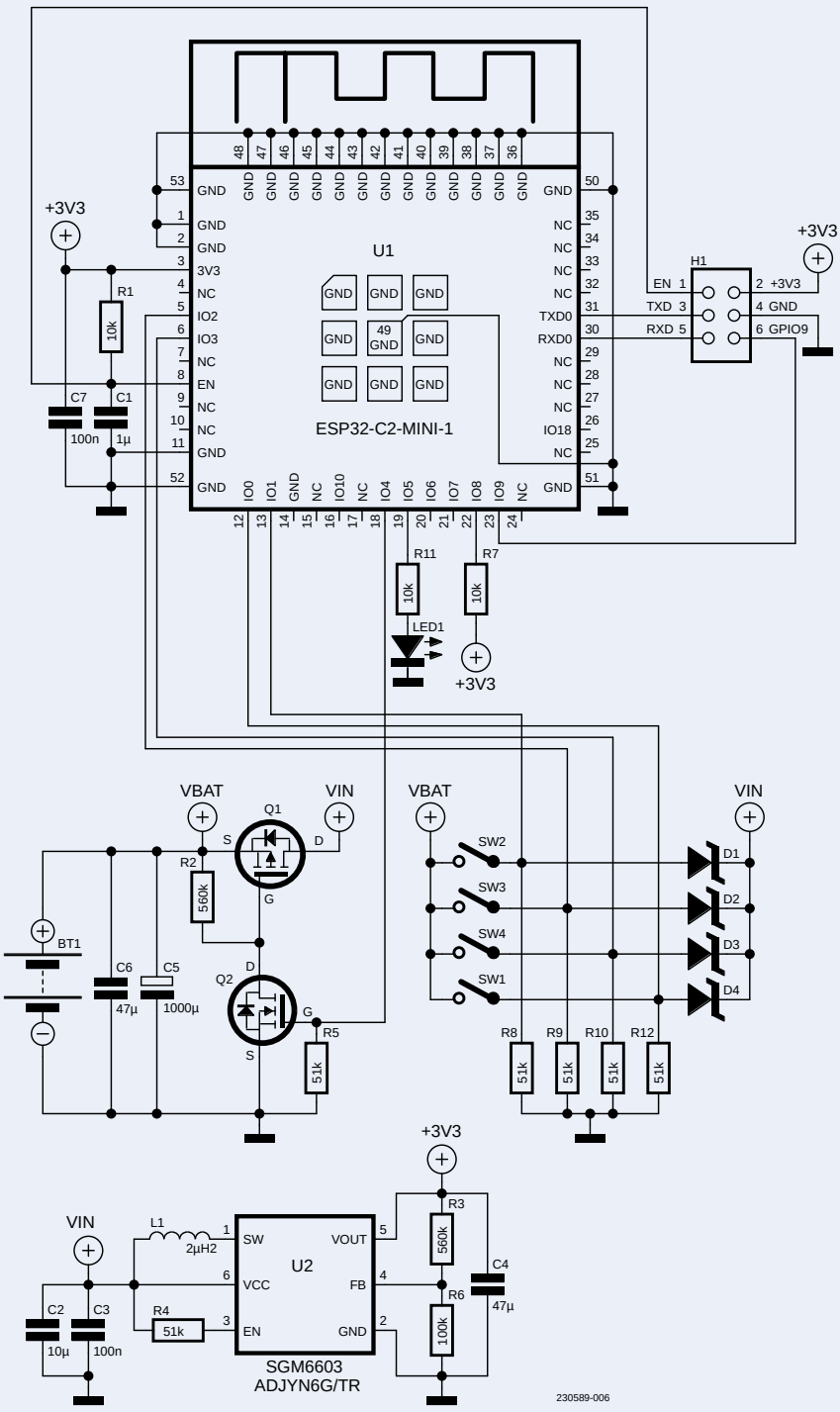


Figure 4: Schematic diagram of the coin cell switch.

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

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

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**).

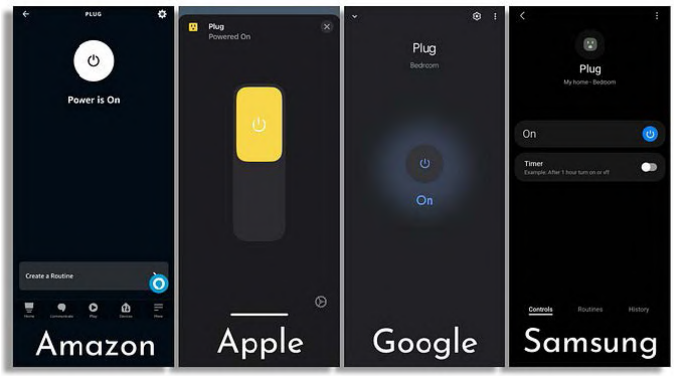


Figure 1: A Matter Plug as seen in various ecosystem apps.

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**.

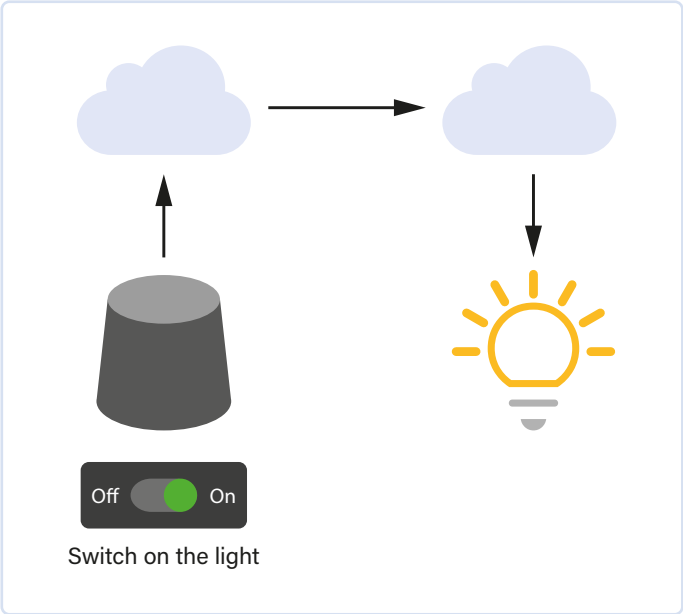


Figure 2: Sample Communication (Non-Matter).

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.

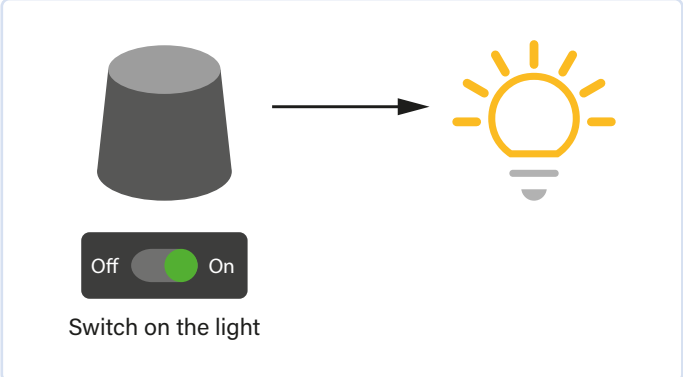


Figure 3: Sample Communication (Matter).

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.

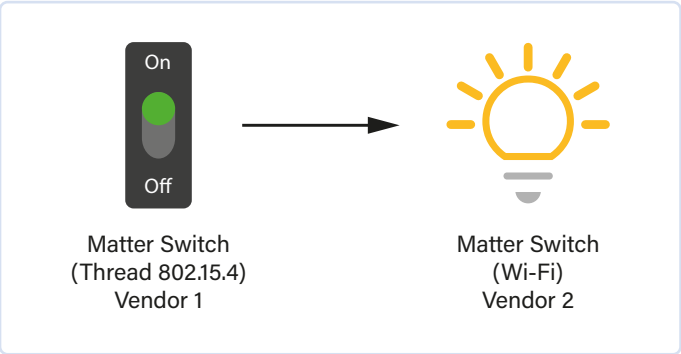


Figure 4: Matter Binding: Device-to-Device Communication.

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.

Related Products

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

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

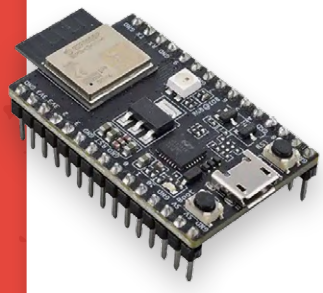
The ESP32-S3-EYE is a small-sized AI development board. It is based on the ESP32-S3 SoC and ESP-WHO, Espressif's AI development framework. It features a 2-Megapixel camera, an LCD display, and a microphone, which are used for image recognition and audio processing.

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

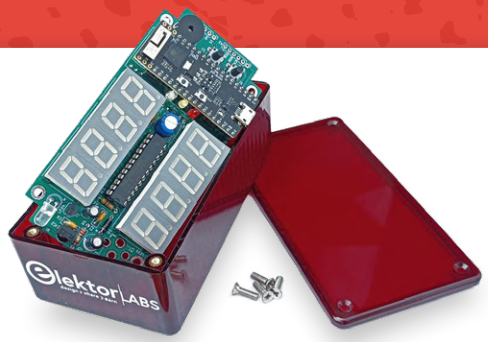
ESP32-C3-DevKitM-1 is an entry-level development board based on ESP32-C3-MINI-1, a module named for its small size. This board integrates complete Wi-Fi and Bluetooth LE functions. Most of the I/O pins on the ESP32-C3-MINI-1 module are broken out to the pin headers on both sides of this board for easy 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.

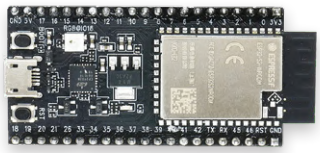
www.elektor.com/19333



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.

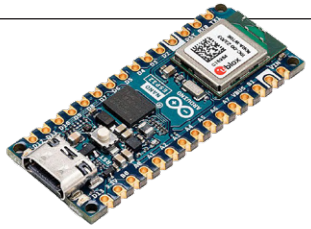
www.elektor.com/20438



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

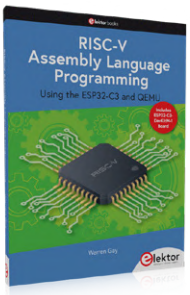
The Arduino Nano ESP32 is a Nano form factor board based on the ESP32-S3 (embedded in the NORA-W106-10B from u-blox). This is the first Arduino board to be based fully on an ESP32, and features Wi-Fi, Bluetooth LE, debugging via native USB in the Arduino IDE as well as low power.

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.

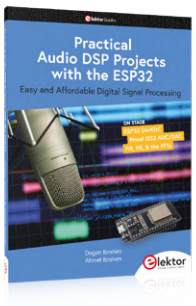
www.elektor.com/20137



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

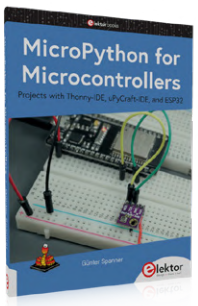
The aim of this book is to teach the basic principles of Digital Signal Processing (DSP) and to introduce it from a practical point of view using the bare minimum of mathematics. Only the basic level of discrete-time systems theory is given, sufficient to implement DSP applications in real time. The 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

MACNICA
ATD EUROPE

Your official authorized distributor
in Europe for Espressif Systems

ESPRESSIF

Macnica ATD Europe
+49 (0)89 899 143-11
sales.mae@macnica.com
www.macnica-atd-europe.com

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

ESP32-P4

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**