Jeroen Domburg
Location: Singapore

Organize Your Mess

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

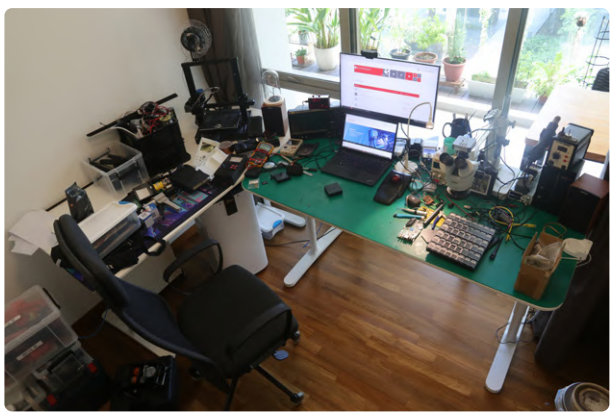
Long-term storage is a different story. I go by the motto of "if I don't know I have it, I might as well not have it," but I don't want to have to actively remember every single thing I have. As such, I store all my components, old projects, etc., in numbered boxes, with a (very-well backed up!) file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

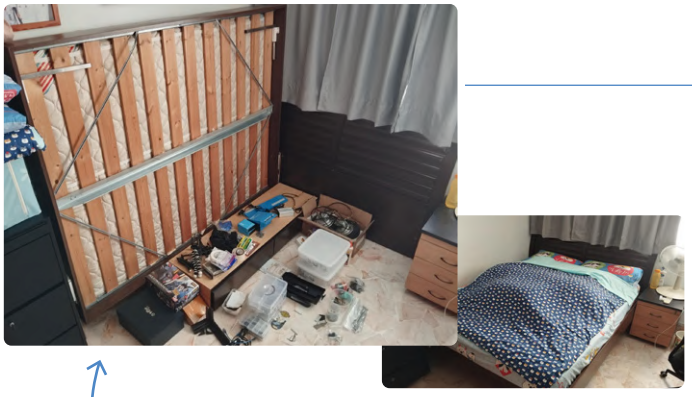
Essentials

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

Wishlist

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





Kai Jie Tan
Location: Singapore

Start Anywhere

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

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

Essentials

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

Wishlist

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

Pedro Minatel
Location: Braga, Portugal

Create a Sanctuary

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

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

Essentials

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

Wishlist

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

Jakob Hasse
Location: China, Shanghai

The Mahjong Table Workspace

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

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

Essentials

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

Wishlist

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

Show Off Your Electronics Workspace!

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

230579-01

CASE STUDY

The ESP RainMaker Story

How We Built "Your" IoT Cloud

By Amey Inamdar, Espressif

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

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

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

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

That's where ESP RainMaker was born.

Choosing the Cloud Architecture

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

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

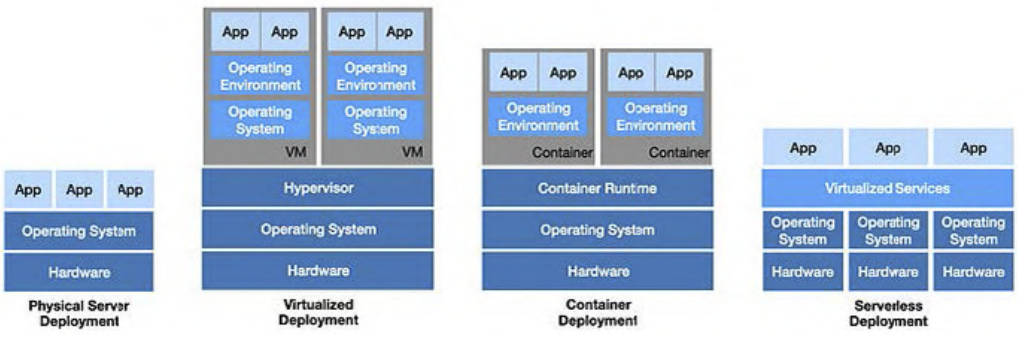


Figure 1: Evolution of cloud architectures.

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

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

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

Dealing With Scalability

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

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

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

Managing Operational Cost

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

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

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

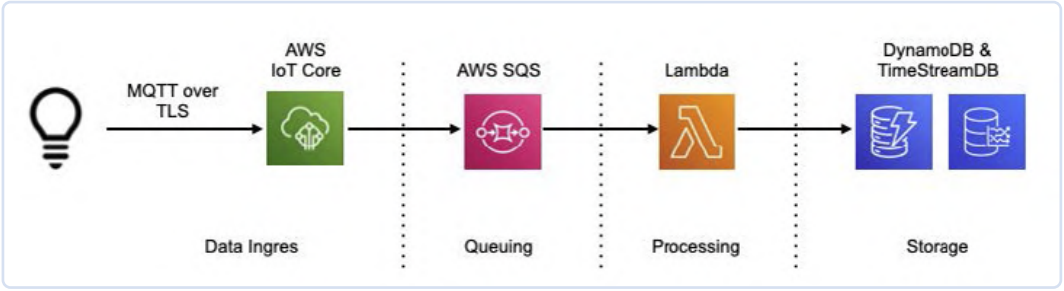


Figure 2: An example of device message processing in ESP Rainmaker.

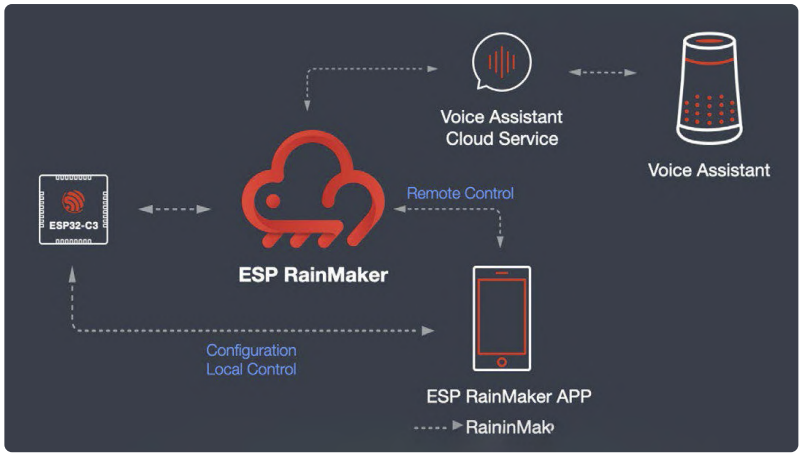


Figure 3: ESP RainMaker components.

Data Security and Privacy

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

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

The IoT Cloud Alone Is Not Enough

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

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

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

Making It Future-Ready

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

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

Getting Started

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

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

230621-01

About the Author

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

WEB LINK

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

Assembling the ELEKTOR CLOC 2.0 KIT

An Elektor Product Unboxed by Espressif

By Jeroen Domburg (Espressif)

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

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

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

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

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

Enter Cloc 2.0

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

The Kit

We got the Cloc 2.0 as a kit, and as such it arrived in parts (**Figure 1**). The most striking was the red Hammond case, with the logo of Elektor Labs printed on it. Aside from looking good, the combination of a red case and red LED display is a good one as it improves the contrast of the LEDs, making the clock easier to read. The rest of the components were in multiple baggies, labeled with the values of the components it contained. As an interesting touch, each one

contains a set of recognizably different components. For instance, no bag had more than two values of resistor in it, and as such you don't have to stare at resistor bands to figure out which part is which. This is a smart idea that saves plastic but still makes finding the right component trivial; kudos to the Elektor team for thinking up this scheme! Of note is that the kit lacks buttons that are accessible from the outside of the case, the thought being that you may still have some fancy ones yourself somewhere.

Finding the Documentation

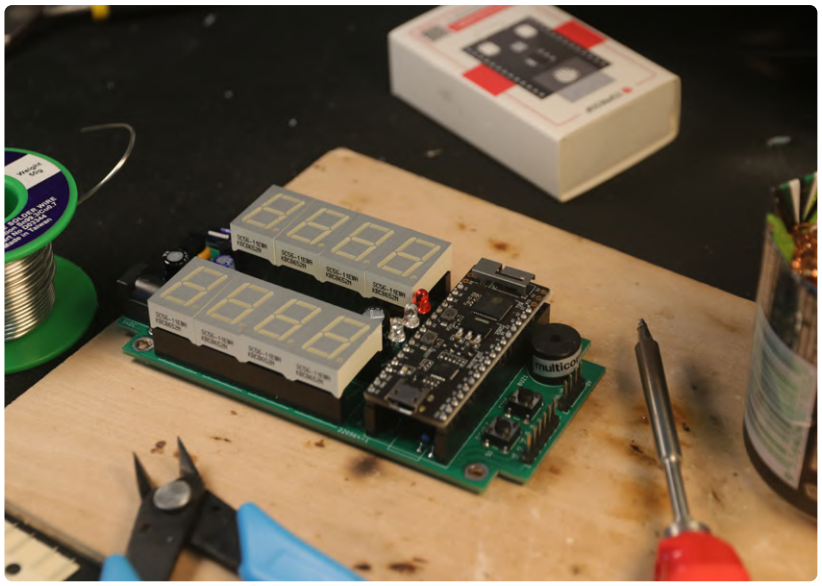
Finding the documentation to put this together could have gone a bit better. The Elektor site has pages spread out over all three domains (elektorlabs.com, elektor.com and elektormagazine.com) with different downloads and text on each page. This is a bit confusing, and it would have been helpful if there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

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

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

Programming the ESP32-Pico-Kit

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



▲

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

Mechanical Construction

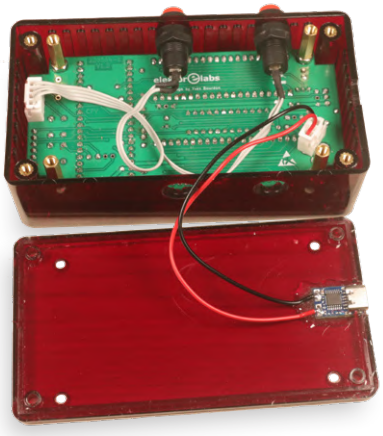
When it comes to the mechanics of putting the PCB in the case, there is a bit more flexibility in how you want to do things: Elektor gives a drill pattern and in recent kits some bits of hardware, so you can build it the "standard" way. However, you certainly may want or even need to modify things in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

The Power Supply

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

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

▼



Drill with Care

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

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

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

Configuring the Cloc

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

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

Suggestions for Cloc 3.0

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

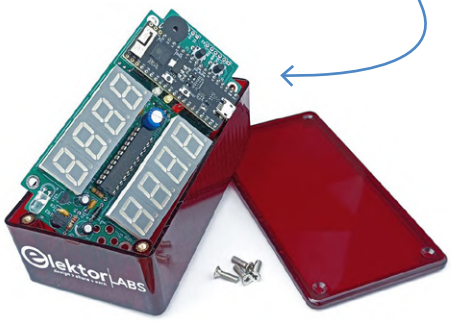
230561-01

Questions or Comments?

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

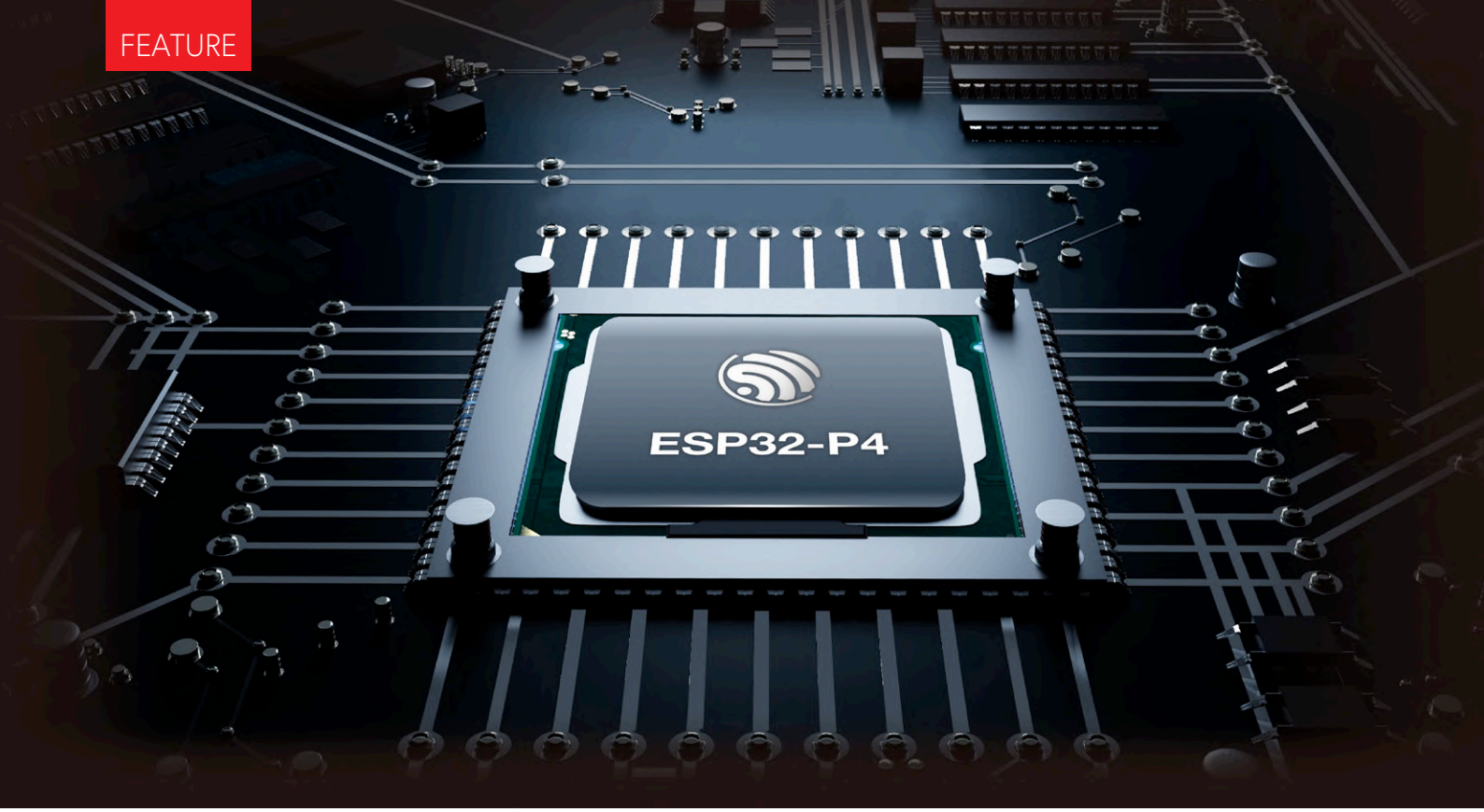
Related Products

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



WEB LINKS

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



Unleashing the **ESP32-P4**

The Next Era of Microcontrollers

By Anant Raj Gupta, Espressif

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

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

Unpacking the ESP32-P4's Computing Process

High-Performance CPU and Memory Subsystem

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

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

Memory Architecture for Unmatched Efficiency

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

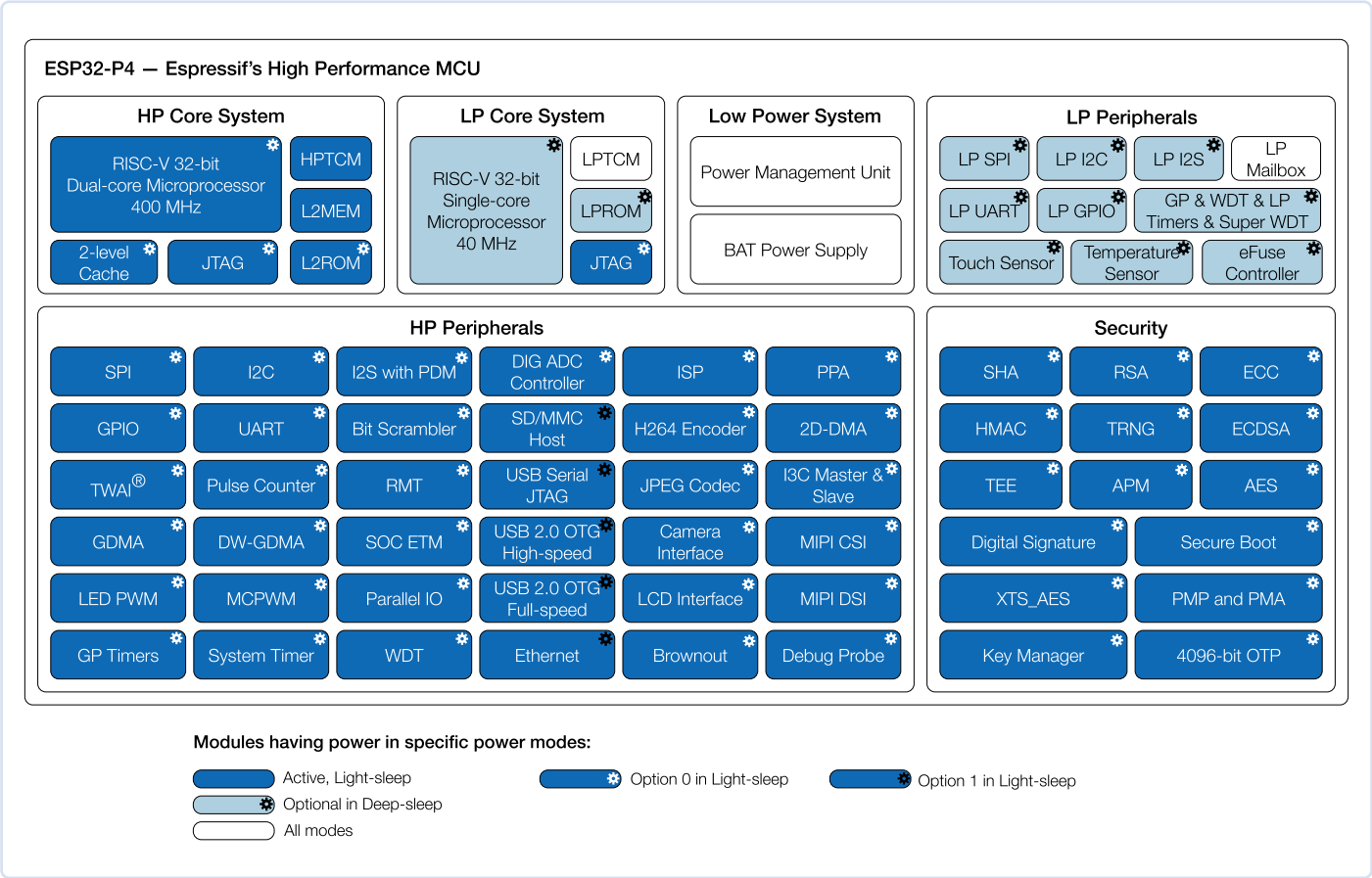


Figure 1: ESP32-P4 block diagram.

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

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

Customized Processing for AI and DSP Workloads

The ESP32-P4 boasts a 32-bit RISC-V dual-core processor that supports standard RV32IMAFCZc extensions. However, what truly sets it apart is its custom, extended instruction set. This set

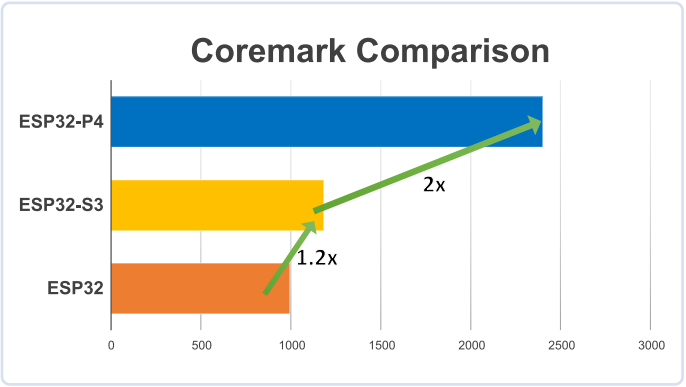


Figure 2: Coremark comparison.

is designed to reduce the number of instructions in loop bodies, significantly enhancing performance. Additionally, the SoC incorporates custom AI and DSP extensions, optimizing the operation efficiency of specific AI and DSP algorithms. These custom vector instructions empower the ESP32-P4 for tasks such as neural network and deep learning workloads, enabling accelerated and efficient computation.

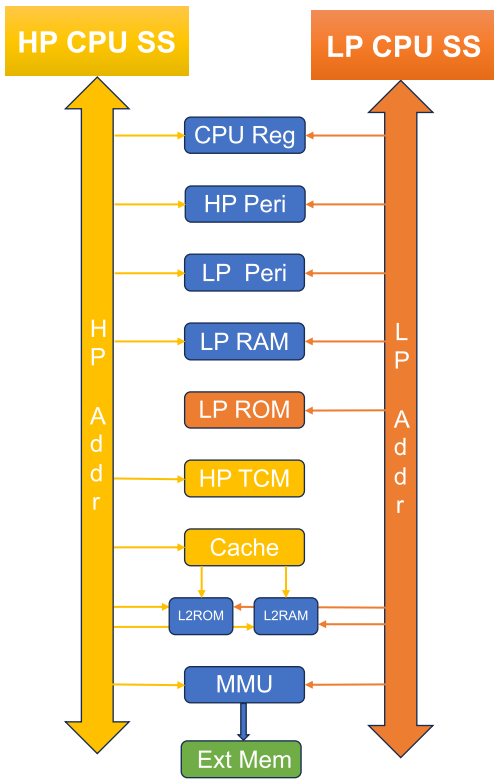


Figure 3: ESP32-P4 address space access.

Efficient Memory Access for Every Core

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

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

Fortifying the Future: Best-in-Class Security

Secure Boot

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

Flash Encryption

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

Cryptographic Hardware Acceleration

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

Private Key Protection

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

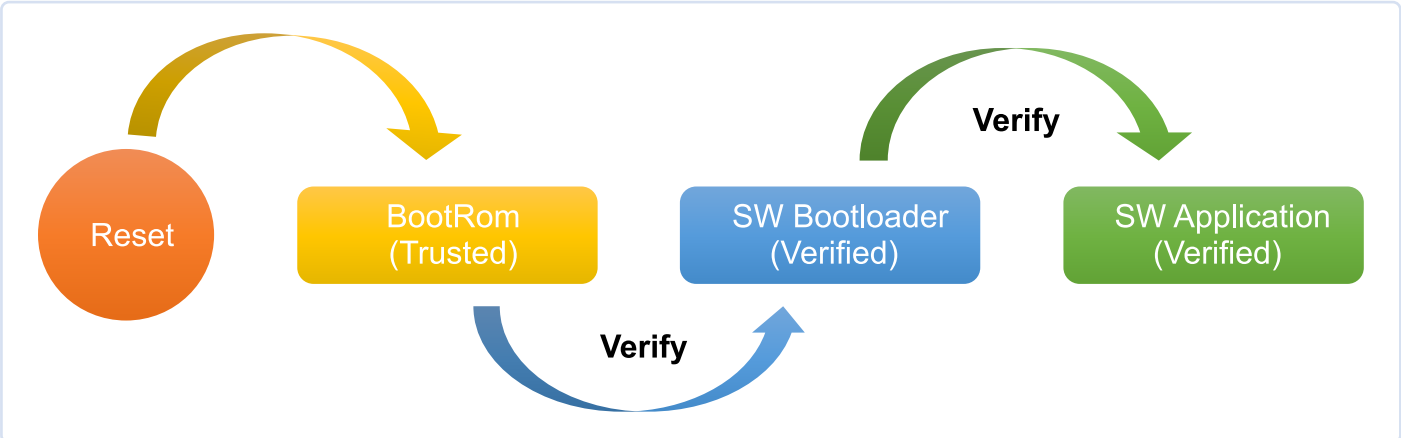


Figure 4: Secure boot flow.

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

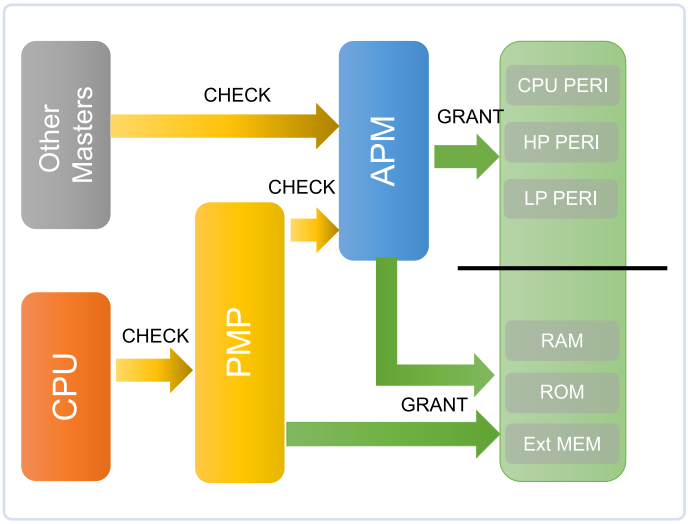


Figure 5: Access control on the ESP32-P4.

Access Control

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

Rich Peripherals and Human-Machine Interface (HMI)

Elevated Visual and Touch Experience

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

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

Media Processing

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

Expansive GPIO and Peripheral Support

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

Seamless Wireless Connectivity

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

Figure 6: Example data flow for an IP camera.

Embrace the Future With ESP32-P4

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

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

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

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

230615-01

Questions or Comments?

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

About the Author

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

Related Product

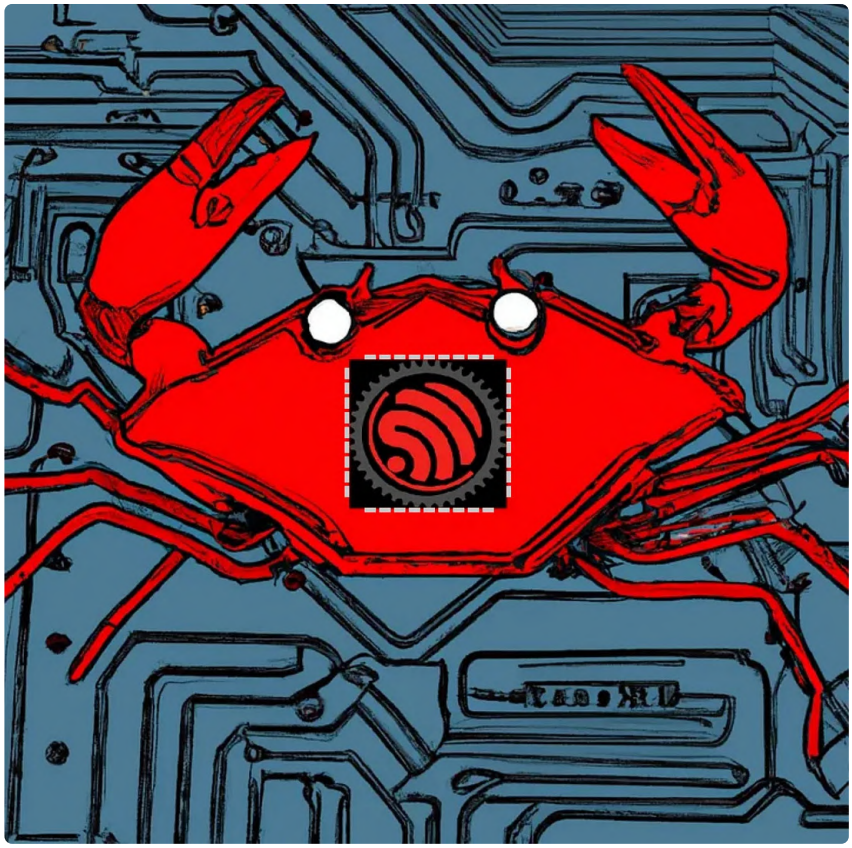
› **Espressif ESP32 range**
www.elektor.com/espressif

Rust + Embedded

A Development Power Duo

By Juraj Sadel (Espressif)

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



Source: Generated by DALL-E.

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

The Beginning of Rust

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

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

Embedded Development

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

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

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

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

Why Rust?

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

Advantages (Compared with C/C++)

Let's review Rust's notable advantages.

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

Disadvantages (Compared With C/C++)

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

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

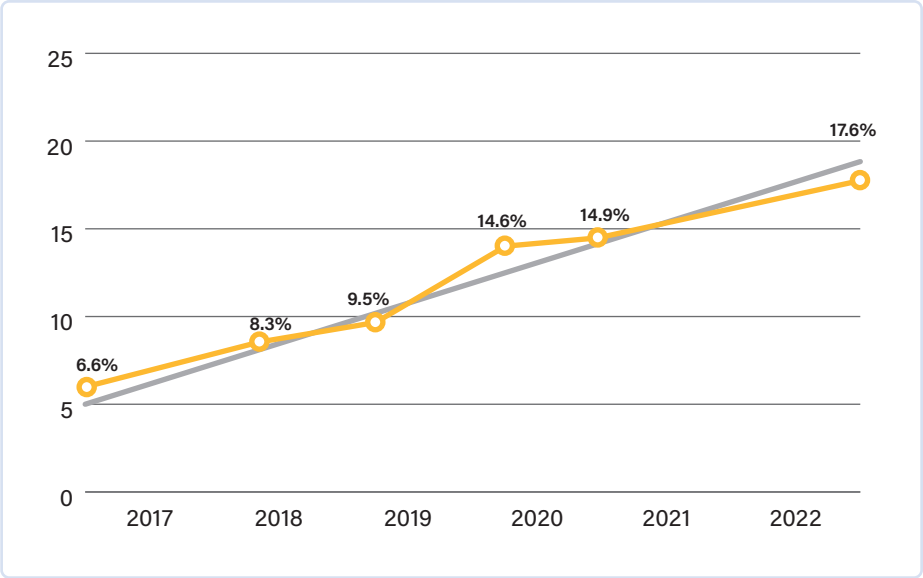


Figure 1: The growing percentage of developers that want to develop in Rust. (Source: Yalantis [4])

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

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

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

How Can Rust Run?

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

What Is Hosted-Environment?

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

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

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

When You Might Want to Use Hosted Environment

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

What Is Bare-Metal?

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

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

When You Might Want to Use Bare-Metal

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

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

Should You Switch From C to Rust?

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

230569-01

Article continues on next page.

About the Author

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

Questions or Comments?

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

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

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

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

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

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

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

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

Listing 2: Blinky example, bare metal.

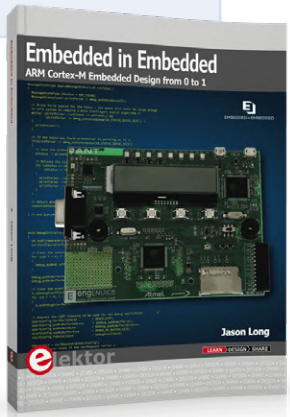
```
#![no_std]
 #![no_main]

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

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

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

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

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

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

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

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

Related Products

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

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



WEB LINKS

[1] MIT Technology Review, "How Rust went from a side project to the world's most-loved programming language," 2023: https://technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language
[2] Rust Blog, "Announcing Rust 1.0," 2015: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html
[3] Rust Blog, "4 years of Rust," 2019: https://blog.rust-lang.org/2019/05/15/4-Years-Of-Rust.html
[4] Yalantis, "The state of the Rust market in 2023" : https://yalantis.com/blog/rust-market-overview
[5] Stack Overflow, "Stack Overflow Developer Survey," 2021: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted
[6] The Embedded Rust Book: https://docs.rust-embedded.org/book/intro/no-std.html#hosted-environments
[7] The Rust Standard Library (std): https://doc.rust-lang.org/std
[8] ESP-IDF: https://github.com/espressif/esp-idf
[9] Newlib library: https://sourceware.org/newlib
[10] Blinky example running on top of ESP-IDF: https://github.com/esp-rs/esp-idf-hal/blob/master/examples/blinky.rs
[11] ESP-IDF-HAL: https://github.com/esp-rs/esp-idf-hal/tree/master/examples
[12] Blinky example running on bare-metal: https://github.com/esp-rs/esp-hal/blob/main/esp32c3-hal/examples/blinky.rs
[13] ESP-HAL: https://github.com/esp-rs/esp-hal/tree/main
[14] ESP-IDF components in Rust: https://github.com/espressif/rust-esp32-example

Q&A With Scott Mabin

Who Are the Rust-Dacious Embedded Developers?

How Espressif is Cultivating Embedded Rust for the ESP32

Intro and Questions by Stuart Cording (Elektor)

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

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

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

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

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

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

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

Scott Mabin: I'd used Rust in my final year project at university. I built a smartwatch — not smart by today's standards, but it did more than tell the time. Part of that project explored whether Rust could replace C for firmware development and where the tradeoffs lay. And, that's when I fell in love with it. I sat there, thinking, "why isn't everyone using Rust?" It offered so much that, once compiled, there was peace of mind that a whole category of memory and race conditions had been eliminated. Of course, I understood that

some projects have legacy and that you won't want to learn and deploy a new language without good reason. Still, it just seemed crazy that more people weren't turning to Rust.

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

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

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

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

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

Scott Mabin: There is definitely customer interest. Customers are primarily using Rust in IoT projects or internet-connected applications. Most projects are pretty simple, such as data loggers. Then, there are more complex applications such as a full-body VR tracker, and an open hardware night sky sensor. There is also a certain amount of future-proofing by

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

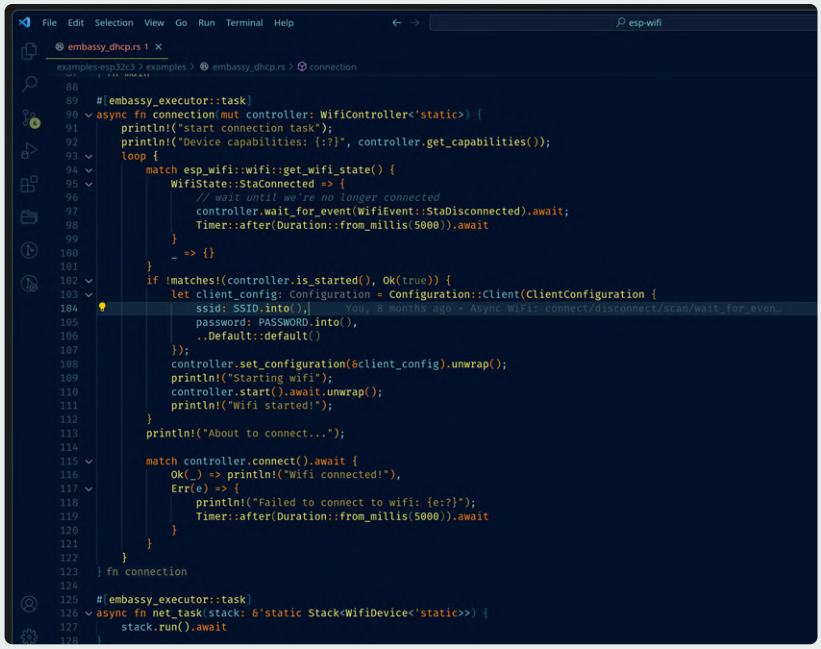
Espressif, preparing for when Rust plays a more significant role in the lives of embedded system developers. Around 10 or 11 of us are contributing to Espressif's Rust Enablement Team, of which about half are full-time resources. On top of that, Rust embedded has an awesome community that contributes as well.

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

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

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

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



▲

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

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

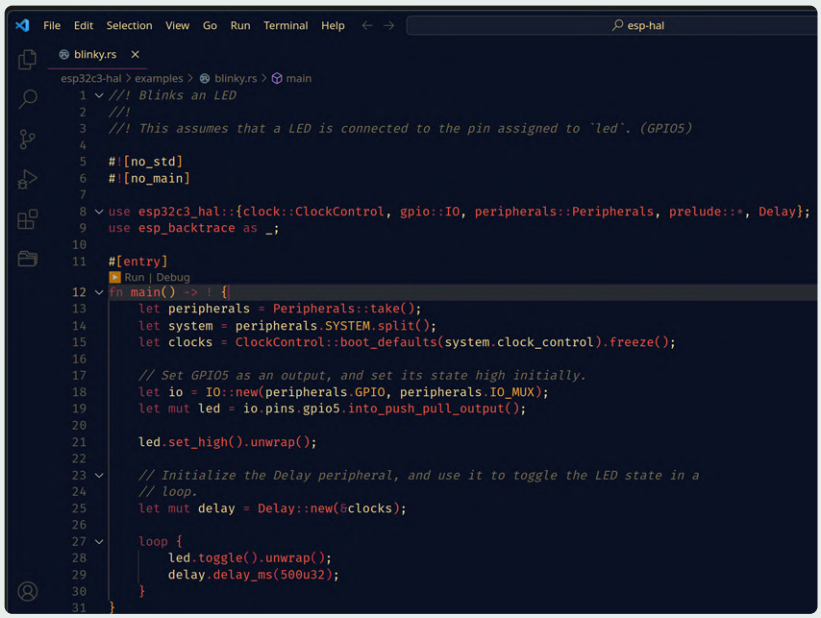
Stuart Cording: Much like C, Rust offers a standard library. Embedded developers abhor the standard library, so, can they also do away with Rust's?

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

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

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

▼



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

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

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

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

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

Scott Mabin: I admire some of the aesthetics of environments such as vim, Neovim, or Emacs setups,

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

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

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

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

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

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

230616-01

Questions or Comments?

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

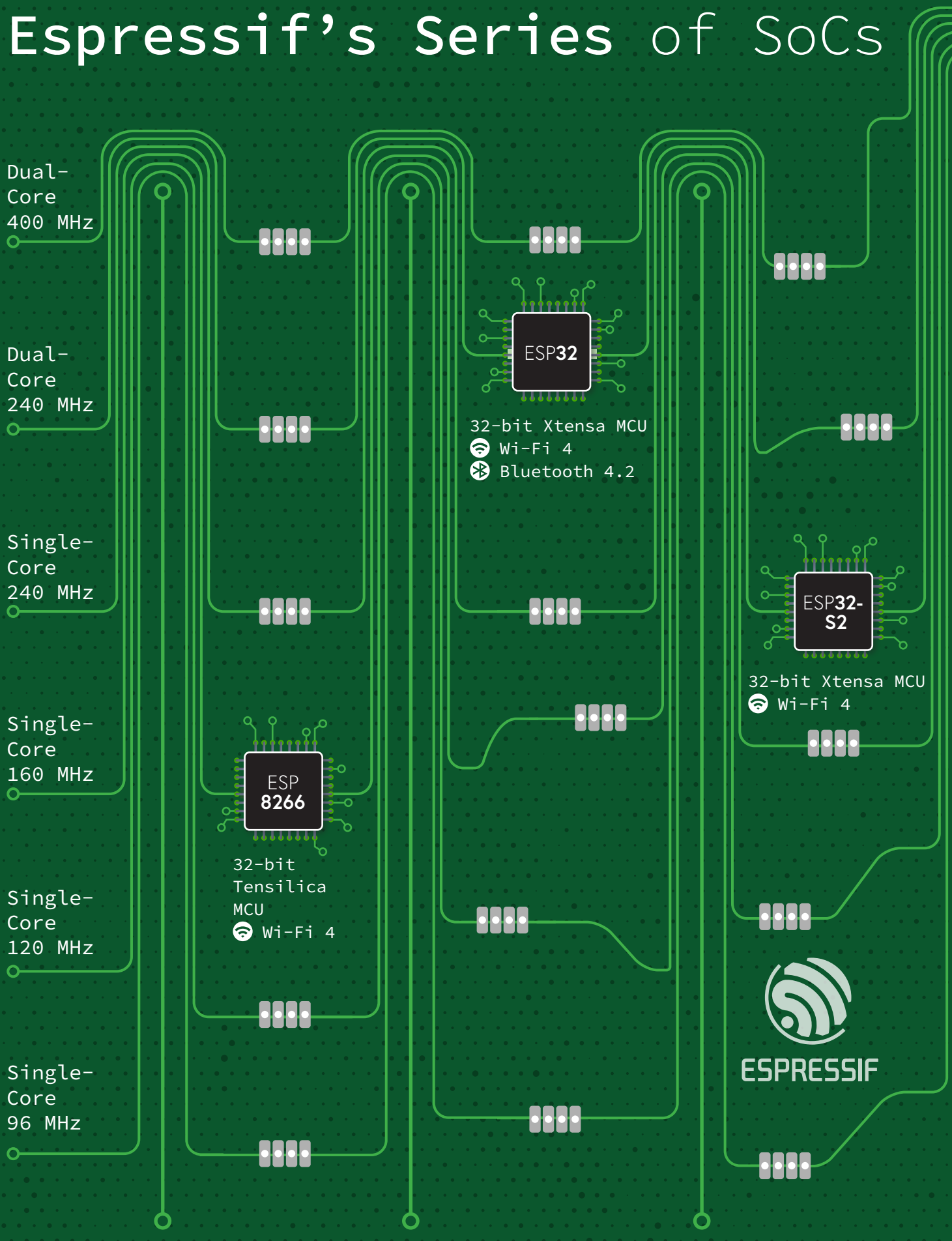
About Scott

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

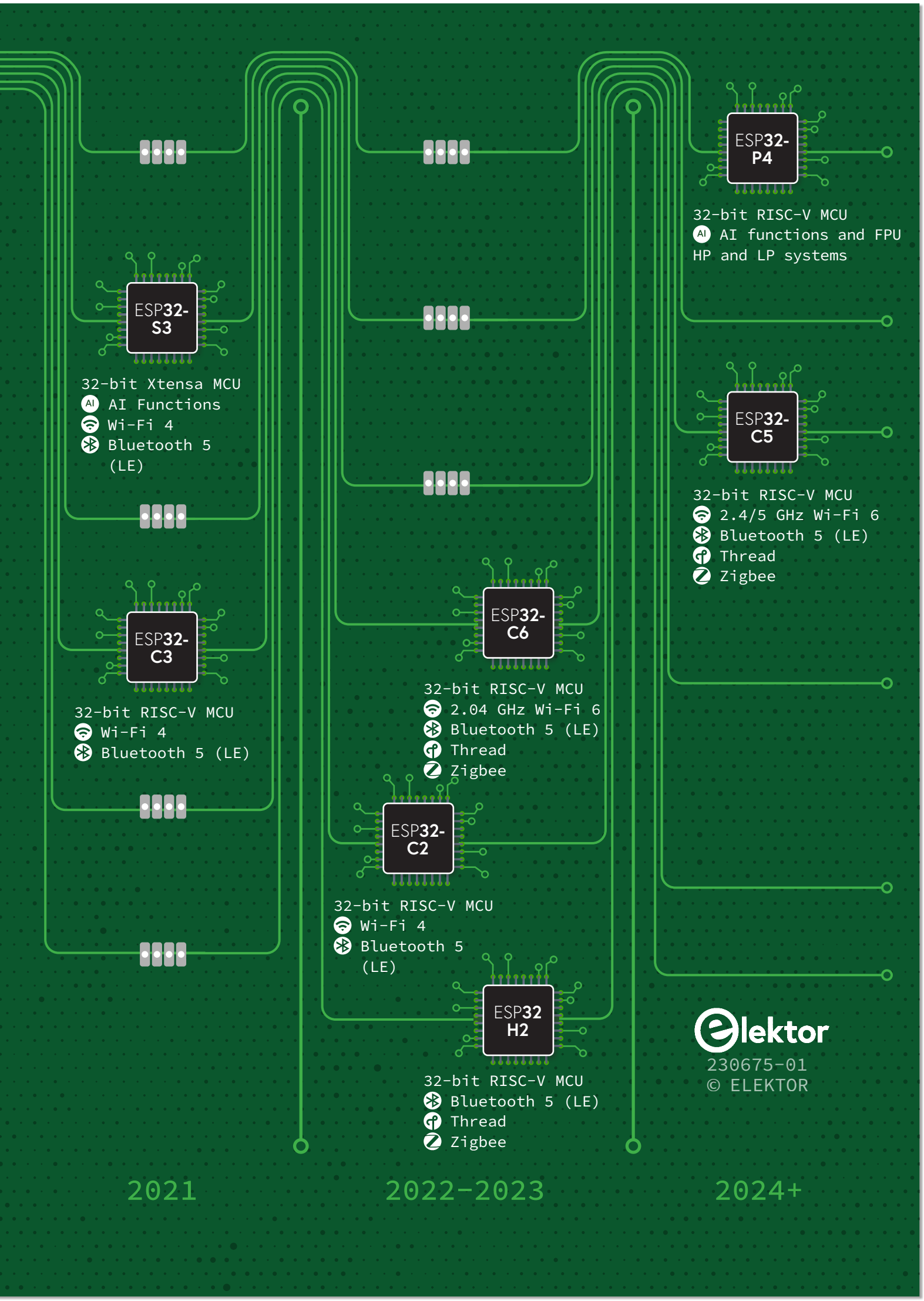
WEB LINKS

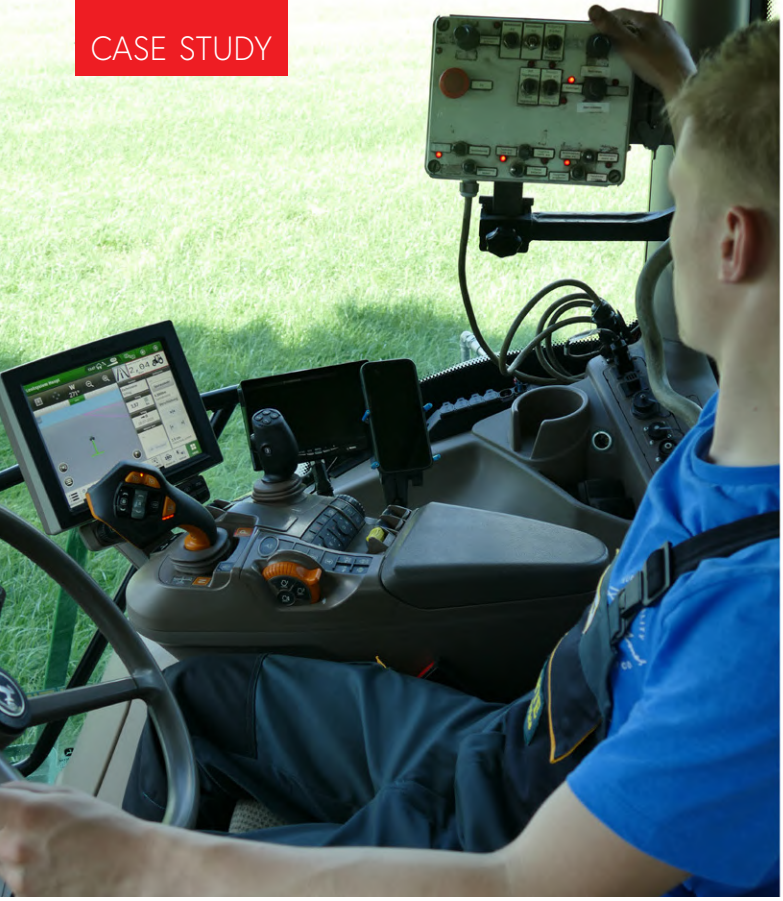
[1] TIOBE Programming Community index: https://tinyurl.com/tioberust
[2] PDP-7 [Wikipedia]: https://tinyurl.com/pdp7wikipedia
[3] PDP-11 [Wikipedia]: https://tinyurl.com/pdp11wikipedia
[4] MSRC Team, "A proactive approach to more secure code," Microsoft, July 2019: https://tinyurl.com/msrcsecurecode
[5] mrustc Project: https://tinyurl.com/mrustcproject
[6] S. Mabin, "Rust on the ESP32," September 2019: https://tinyurl.com/rustesp32
[7] embedded_hal Crate Documentation: https://tinyurl.com/embeddedhalcrate
[8] heapless Crate Documentation: https://tinyurl.com/heaplesscrate
[9] The Rust on ESP Book, "Using the Standard Library (std)": https://tinyurl.com/rustbookstd
[10] ESP-IDF Programming Guide, "Get started": https://tinyurl.com/espidfgetstarted
[11] The Rust on ESP Book, "Using the Core Library (no_std)": https://tinyurl.com/rustbooknostd
[12] Training, "Embedded Rust on Espressif": https://tinyurl.com/rustonespressif
[13] Rust user group on Matrix: https://tinyurl.com/rustmatrixug

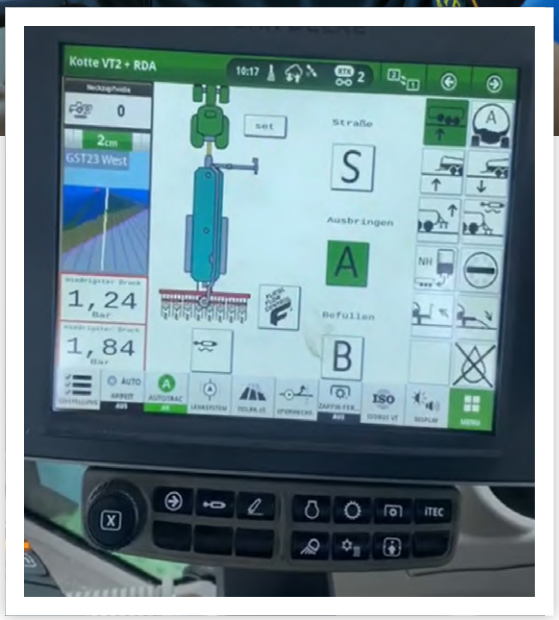
Espressif's Series of SoCs



elektormagazine.com/posters


ESP32-S3
32-bit Xtensa MCU
AI AI Functions
Wi-Fi 4
Bluetooth 5 (LE)
ESP32-C3
32-bit RISC-V MCU
Wi-Fi 4
Bluetooth 5 (LE)
ESP32-C6
32-bit RISC-V MCU
2.04 GHz Wi-Fi 6
Bluetooth 5 (LE)
Thread
Zigbee
ESP32-C2
32-bit RISC-V MCU
Wi-Fi 4
Bluetooth 5 (LE)
ESP32 H2
32-bit RISC-V MCU
Bluetooth 5 (LE)
Thread
Zigbee
ESP32-P4
32-bit RISC-V MCU
AI AI functions and FPU
HP and LP systems
ESP32-C5
32-bit RISC-V MCU
2.4/5 GHz Wi-Fi 6
Bluetooth 5 (LE)
Thread
Zigbee
elektor
230675-01
© ELEKTOR
2021
2022-2023
2024+





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

Building a PLC with Espressif Solutions

With the Capabilities and Functionality of the ISOBUS Protocol

By Franz Höpfinger, HR Agrartechnik

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

The Challenge

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

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

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

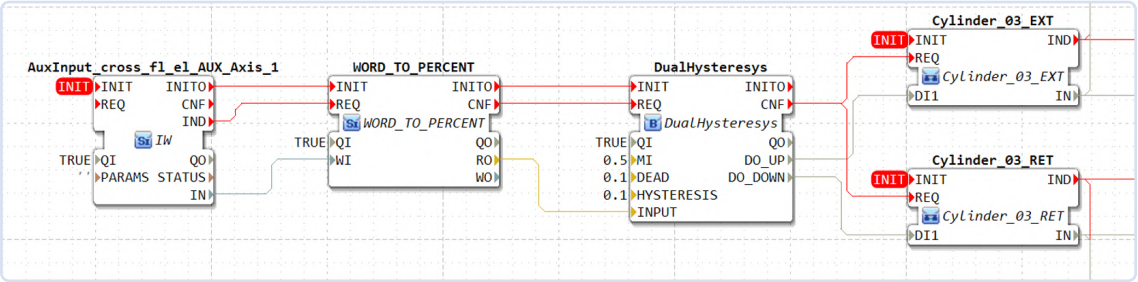


Figure 2: This is a source code listing that shows how programming in IEC 61499/Eclipse 4diac™ works. You do not write lines of source code, but use functional blocks (FBs) instead.

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

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

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

The Solution

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

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

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

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

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

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

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

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

230610-01

Questions or Comments?

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

WEB LINKS

[1] Eclipse 4diac™: https://eclipse.dev/4diac
[2] logiBUS® Homepage: https://logibus.tech
[3] logiBUS® open-source derivatives for e.g. Building Automation (non ISOBUS): https://gitlab.com/meisterschulen-am-ostbahnhof-munchen
[4] Bale Counter: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac_EasyExampleCounter
[5] Slurry Tanker: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac-SlurryTanker-sample
[6] Resources and Wiki: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen

The ESP32-S3 VGA Board

Bitluni's Exciting Journey Into Product Design

Compiled by Elektor and Espressif

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

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

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

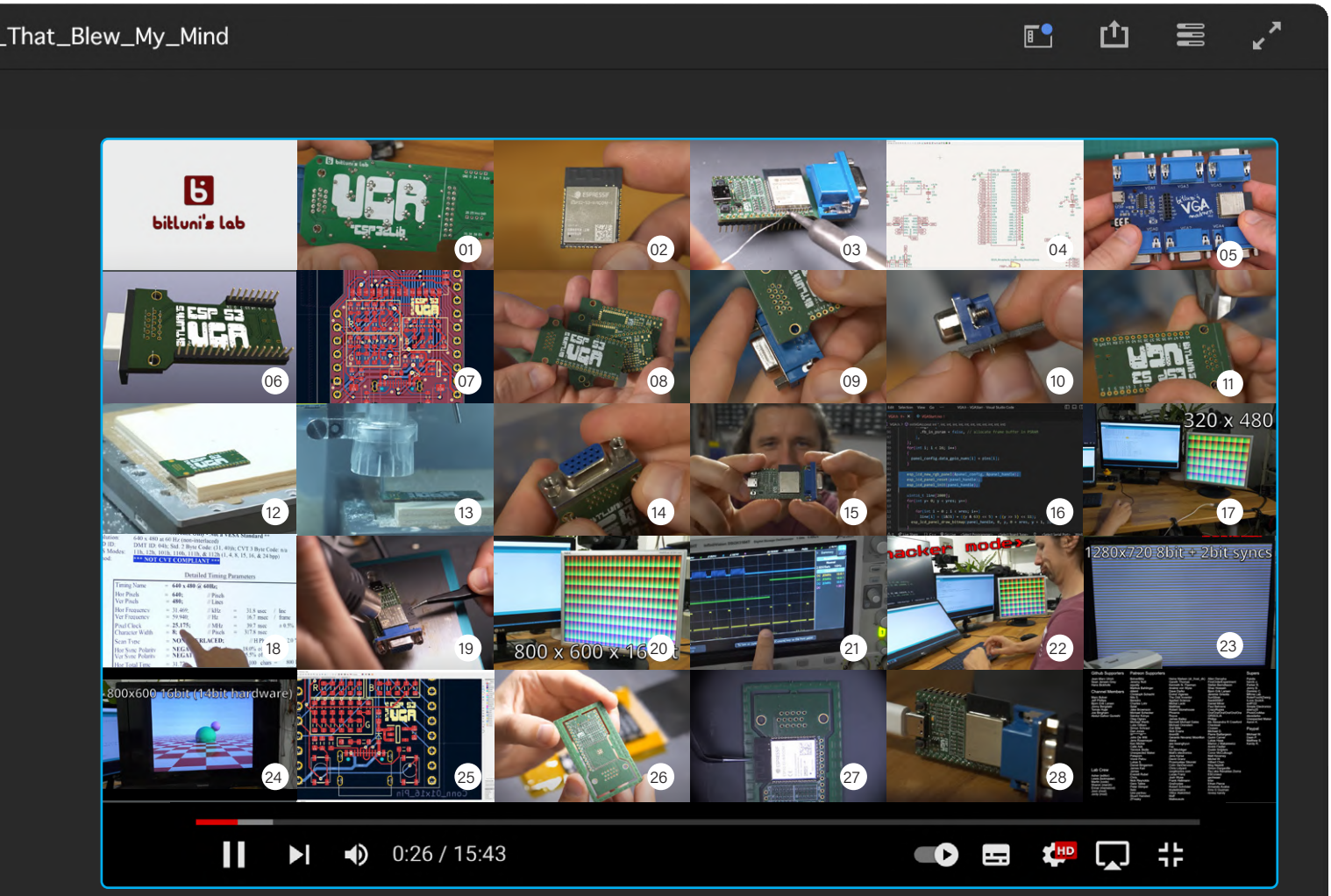
Designing ensued (**Figure 7**), boards were ordered, and the results were gorgeous (**Figure 8**)! Bitluni assumed the VGA connectors he had would be compatible with the footprint of the PCBs, but he soon discovered the pins didn't match (**Figure 9**)! Luckily, he had a few narrower connectors on hand.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

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

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

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

230529-01



bitluni
231K subscribers

Related Products

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

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

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

Scan the QR-code...

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

or

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

Acoustic Fingerprinting on ESP32

Song Recognition With Open-Source Project Olaf

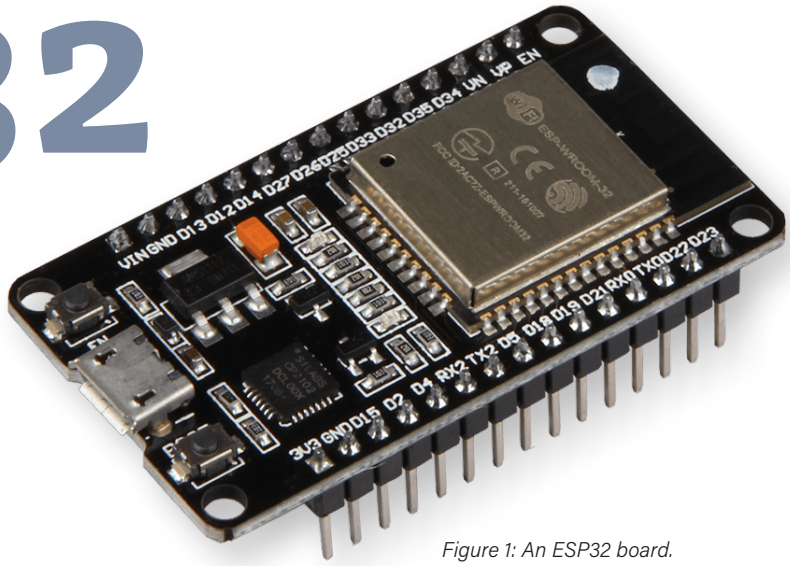


Figure 1: An ESP32 board.

By Joren Six, Ghent University (Belgium)

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

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

As my daughter's fourth birthday approached, I decided to transform the prototype into an extravagantly engineered birthday gift: an "Elsa" dress that reacts to the song "Let It Go" from the *Frozen* soundtrack [1]. Building upon the first prototype, I ordered an RGB LED strip, a robust Li-Ion battery, an I²S digital microphone, and, naturally, an Elsa dress.

I had an ESP32 (**Figure 1**) microcontroller on hand and used it as the central component of the system. It supports I²S, includes a floating-point unit (FPU), is easy to integrate with LED strips, and has enough RAM. The FPU simplifies the use of the same code on both conventional computers and embedded devices, eliminating the need for fixed-point math.

Building the ESP32 Version

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

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

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

Figure 2: RGB LED strip.

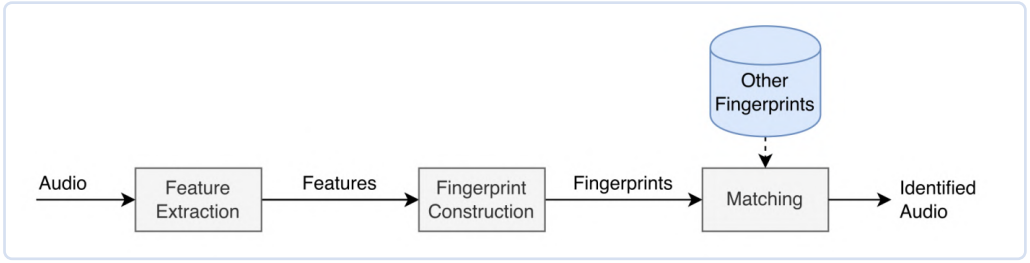


Figure 3: Sound recognition block diagram.

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

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

What's Under the Hood?

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

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

These signatures are based on spectrograms, which visually represent the song's frequency content over time. Spectrograms also display the signal power level, reflecting the perceived loudness through color variations.

However, if Shazam were to match spectrograms like these directly, it would require an exceptionally long time to provide a result. One major breakthrough is concentrating on the peaks in the spectrogram, since these peaks contain salient information that is also processed by the human brain.

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

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

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

Evolution of the Project

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

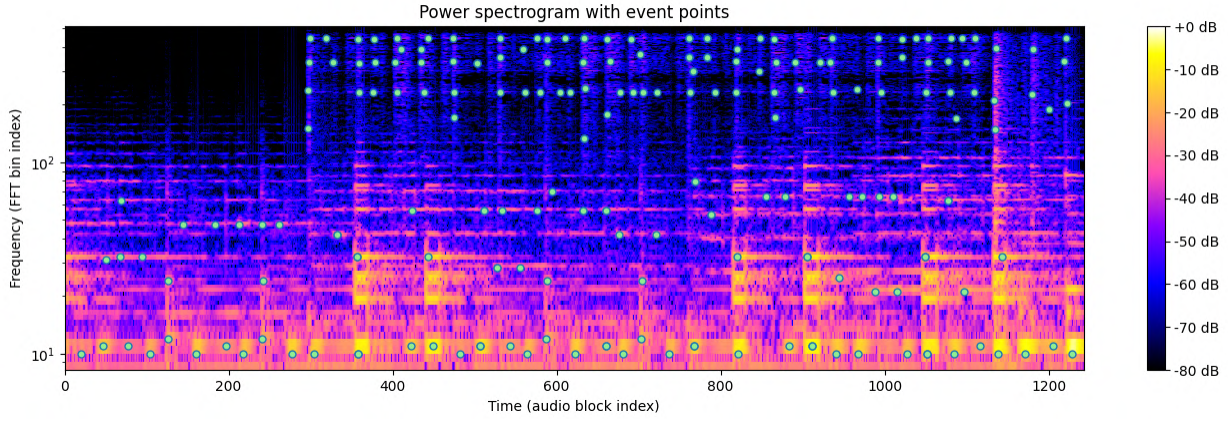


Figure 4: A spectrogram shows the how present frequencies are in music at a given time. The green dots are extracted by Olaf and show peaks in the spectrogram.

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

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

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

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

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

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

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

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

Build It Yourself Using an ESP32

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

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

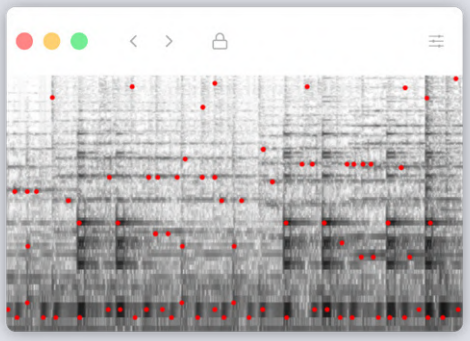
While preparing this article, I added code for an ESP32 demo, as well as additional documentation, to my GitHub repository [7]. Now, the latest version of Olaf operates reliably on ESP32 with an easily obtainable MEMS microphone. On that help page, you'll also find instructions on how to utilize I²S microphones such as the INMP441.

Running It in a Web Browser

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

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

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



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

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

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

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

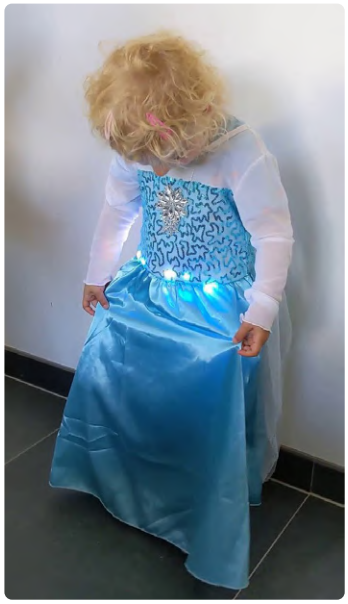


Figure 5: The ESP32-powered costume.

What's Next?

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

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

230578-01

Questions or Comments?

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

About the Author

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

Related Products

- **JOY-iT NodeMCU ESP32 Development Board**
  www.elektor.com/19973
- **ESP32-DevKitC-32E**
  www.elektor.com/20518
- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324

WEB LINKS

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

Circular Christmas Tree 2023

A High-Tech Way to Celebrate the Holiday Season

By Ton Giesberts (Elektor)

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

This Christmas tree is all about the shape of the construction and the use of 8 mm digital RGB LEDs. Its appearance is more like a real tree than most of the flat PCBs with a pine tree-styled outline. Instead of being just a plain 2D design, this is a real 3D one, and it's based on the two earlier versions, [1] and [2]. The new tree is realized by separating the five annular, concentric sections of the PCB supplied with the kit, and mounting them in sequence with some rigid wires that distance the boards, giving to this project the typical look of a conifer-shaped-tree.

The project in [1] used a simple random number generator with two standard logic ICs and small white SMD LEDs. [2] was microcontroller-based, with the LEDs connected in a 6×6 matrix. This latest version uses 36× 8 mm digital RGB through-hole LEDs of type WS2812D-F8. This makes the circuit simple. Instead of a matrix, the inputs and outputs of the digital LEDs are connected in series, with each LED connected to a 5 V power supply. The chain of LEDs can be addressed like a NeoPixel LED strip. There are a large number of programs on the web, in many languages, that explain in detail how to manage these devices.

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

Schematic

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

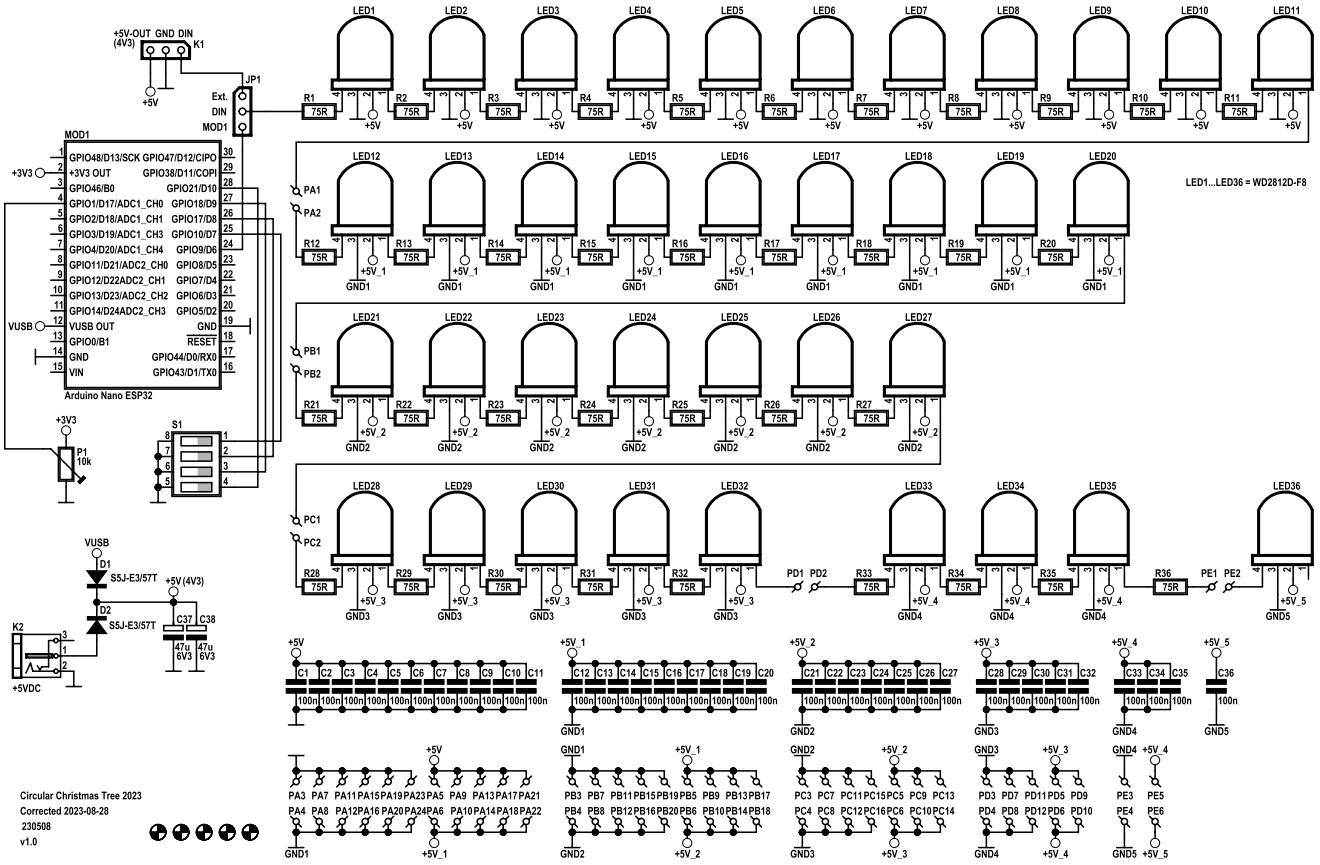


Figure 1: Schematic of the Circular Christmas Tree.

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

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

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

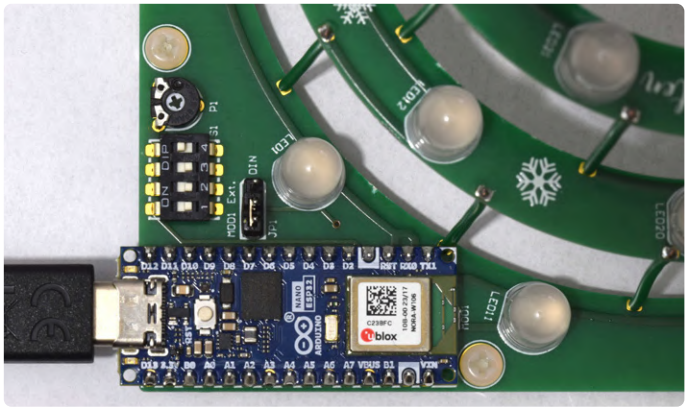


Figure 2: Detail of MOD1, S1, and P1 mounted on the base PCB.

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

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

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

Power Supply

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

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

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

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

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

PCB

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

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

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

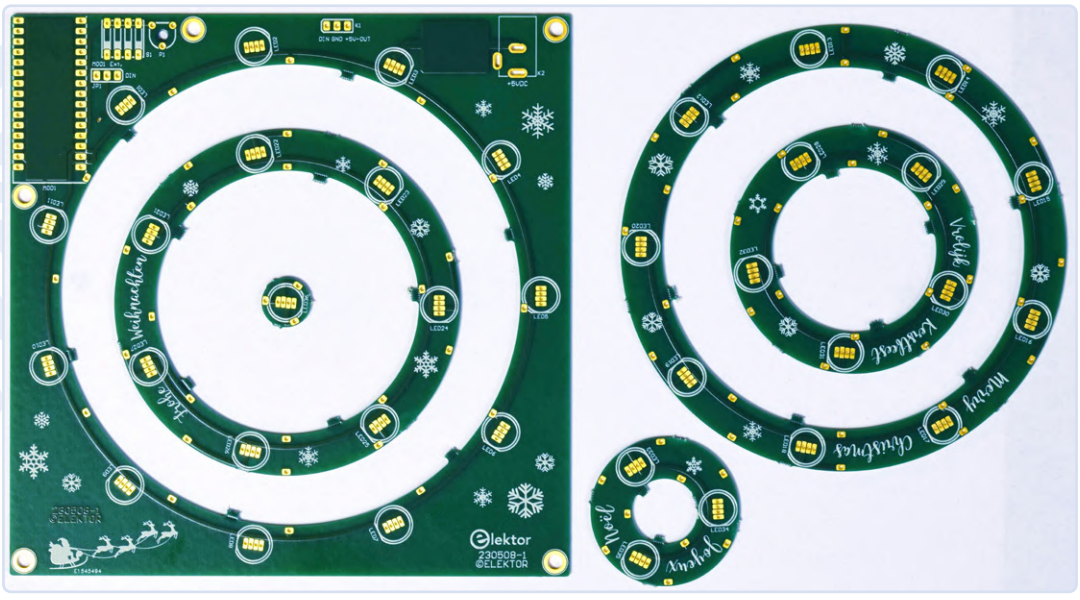
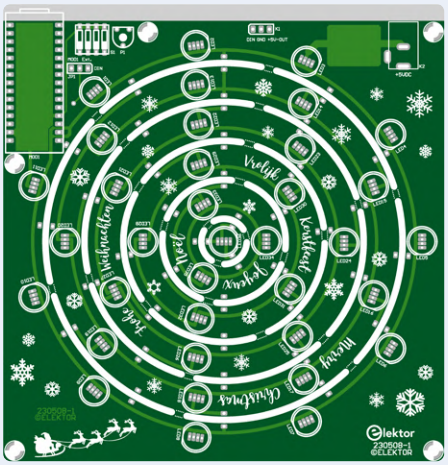
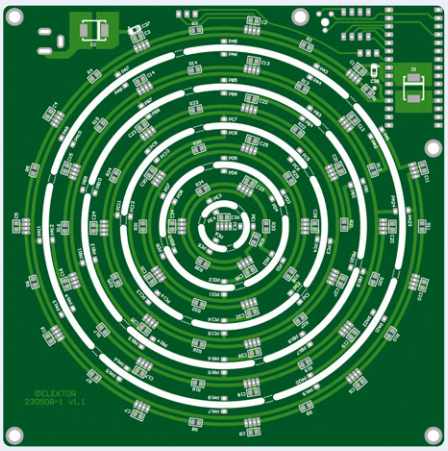


Figure 4: All PCBs separated. The bridging segments that connected the 6 PCBs can now be removed with a pair of wire cutters.

Component List





Resistors

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

Capacitors

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

Semiconductors

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

Others

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

Misc.

PCB 230508-1 (136 × 136 mm)

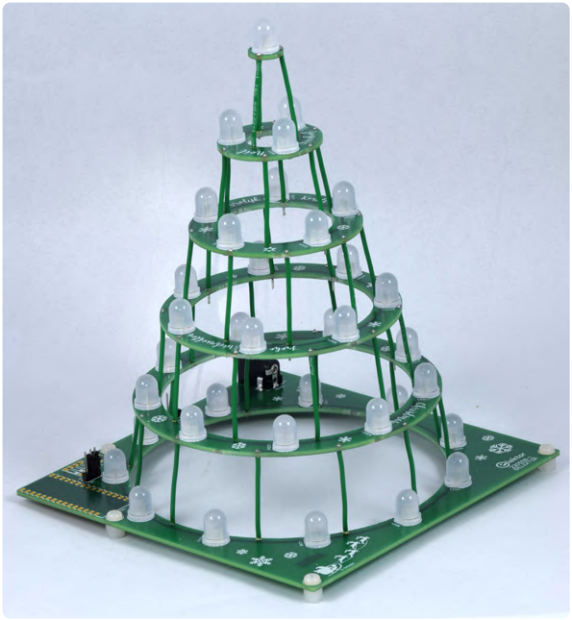


Figure 5: The completed prototype, without the optional Arduino module.

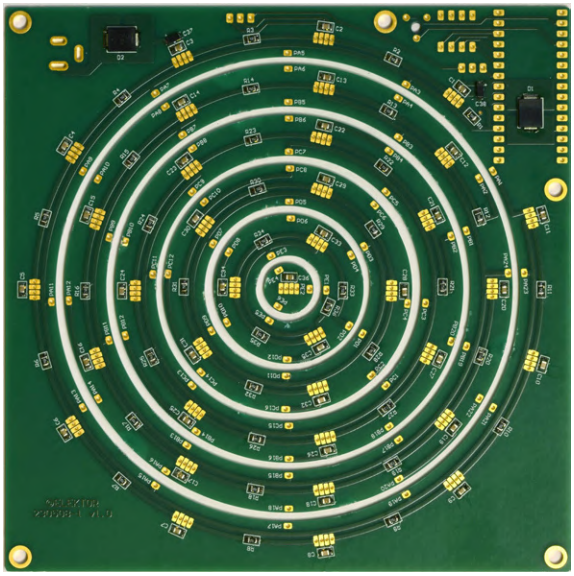


Figure 6: All SMDs are soldered on the back sides of the PCBs: R1...R36, C1...C38, and D1 and D2.

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

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

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

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

Optional Arduino Nano ESP32 Module

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

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

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

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

230665-01

About the Author

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

Questions or Comments?

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

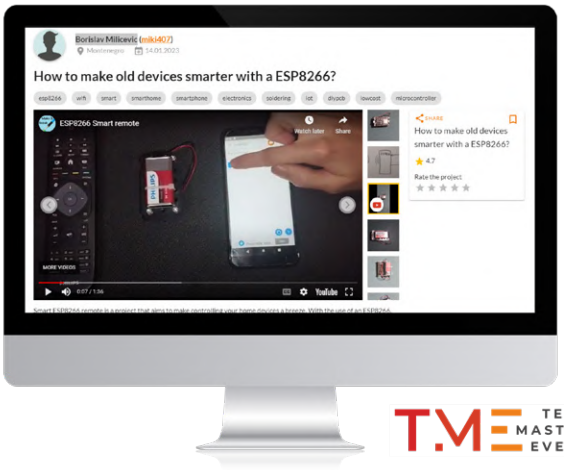
Related Products

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

Figure 8: Test setup of the Christmas Tree.

WEB LINKS

[1] Christmas Tree V1: https://elektormagazine.com/labs/130478-xmas-tree-2014
[2] Christmas Tree V2: https://elektormagazine.com/labs/circular-christmas-tree-150453
[3] Documentation about the Arduino Nano ESP32: https://docs.arduino.cc/hardware/nano-esp32
[4] WS2812D-F8 Datasheet [PDF]: https://soldered.com/productdata/2021/03/Soldered_WS2812D-F8_datasheet.pdf
[5] Arduino Nano ESP32 Datasheet [PDF]: https://docs.arduino.cc/resources/datasheets/ABX00083-datasheet.pdf
[6] This project on Elektor Labs: https://elektormagazine.com/labs/circular-christmas-tree-2023-230508
[7] Arduino Lab for Windows [Zip]: https://tinyurl.com/arduinolab4win
[8] Arduino MicroPython Installer: https://github.com/arduino/lab-micropython-installer/releases/tag/v1.2.1
[9] MicroPython course MicroPython 101 by Arduino: https://docs.arduino.cc/micropython-course/



A Simpler and More Convenient Life

An Amateur Project Based on the Espressif ESP8266 Module

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

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

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

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

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

ESP8266, IR LED and IR receiver

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

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

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

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

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

230656-01

WEB LINKS

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

Blynk

How to Build IoT Apps without Software Expertise

With Blynk IoT Platform and Espressif Hardware

Contributed by Blynk Inc.

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

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

What is included in Blynk IoT?

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

Blynk App Builder for iOS and Android

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

Web Dashboard Builder

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

Figure 1: No-Code Interfaces created with Blynk.

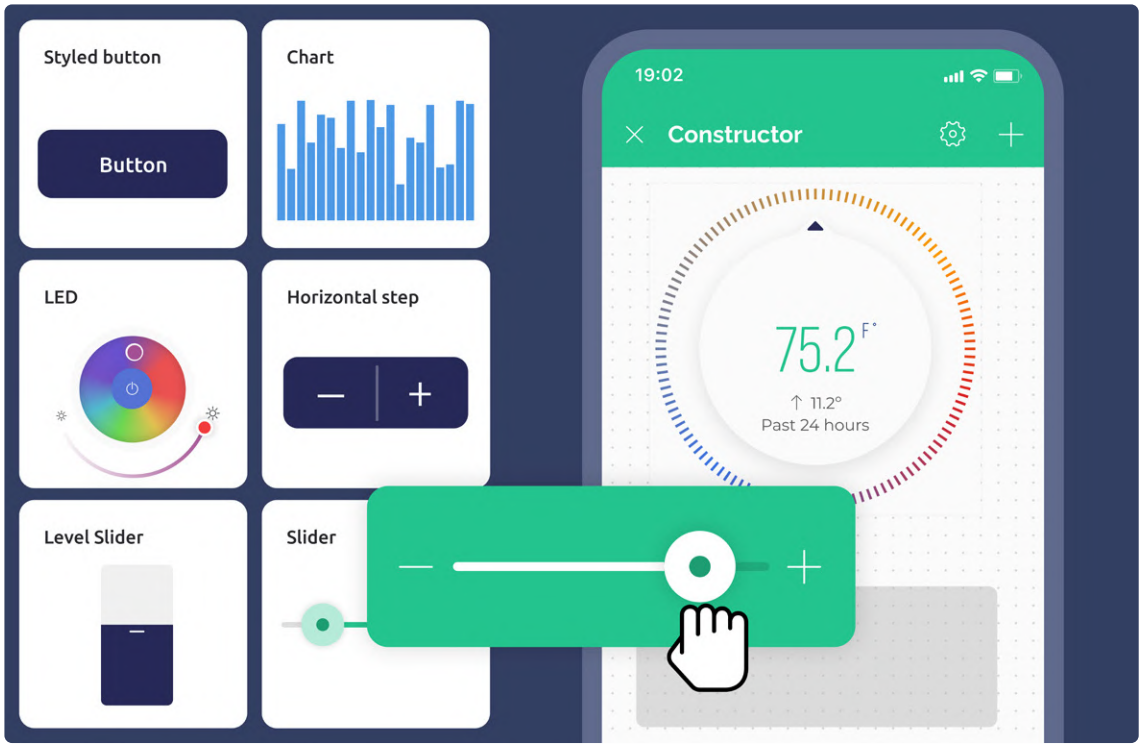


Figure 2: Blynk Drag-n-Drop App Builder.

Advanced User Management System

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

Built-in Device Lifecycle Management

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

No-Code Automation Scenarios

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

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

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

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

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

230659-01

WEB LINKS

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

Building a Smart User Interface on ESP32

Contributed by Slint

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

Figure 1: C++ and Rust logos.

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

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

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

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

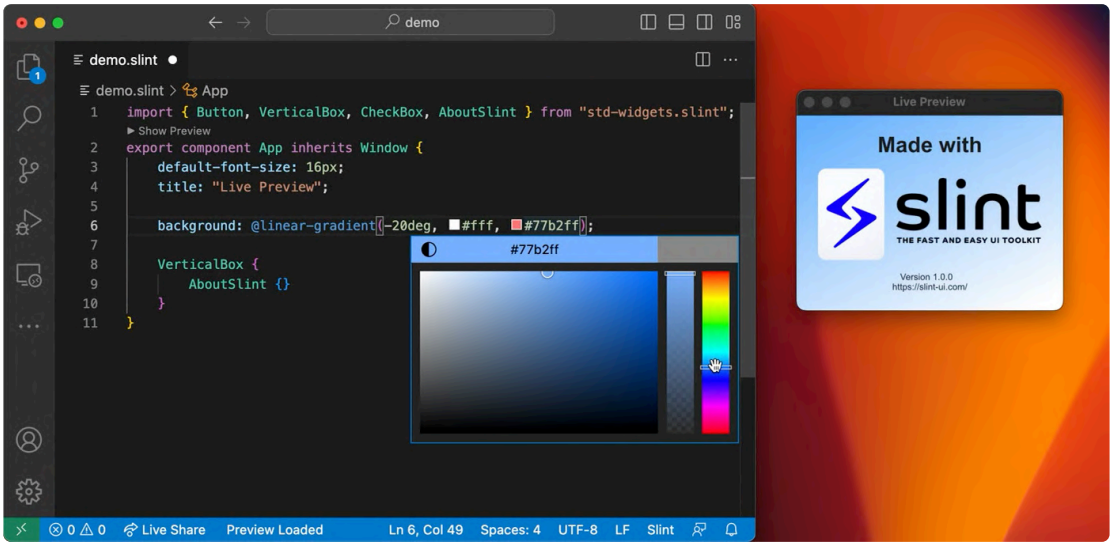


Figure 2: Quick iterations with Slint's Live-Preview.

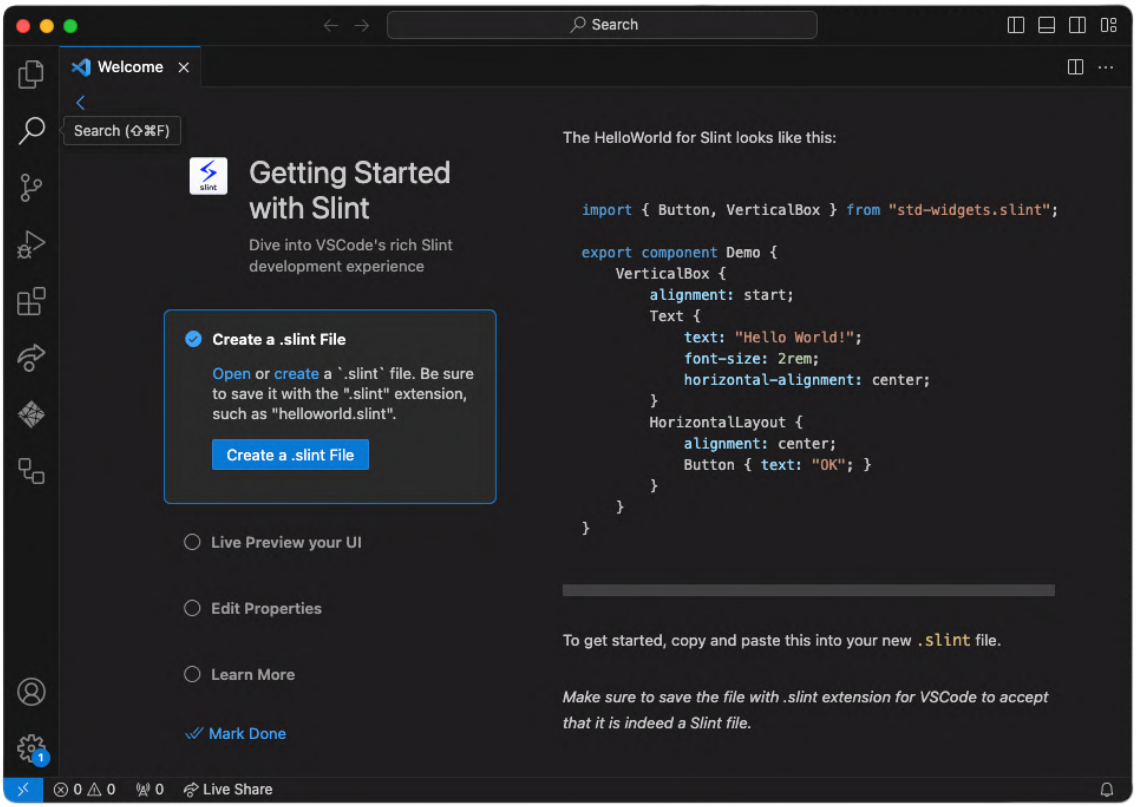


Figure 3: Getting started with Slint.

Separate UI from Business Logic

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

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

Enjoy a Good Developer Experience (DX)

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

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

Deliver an Exceptional User Experience (UX)

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

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

230670-01

Figure 4: A Slint demo on ESP32.

WEB LINK

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

Quick & Easy IoT Development with M5Stack

Contributed by M5Stack

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

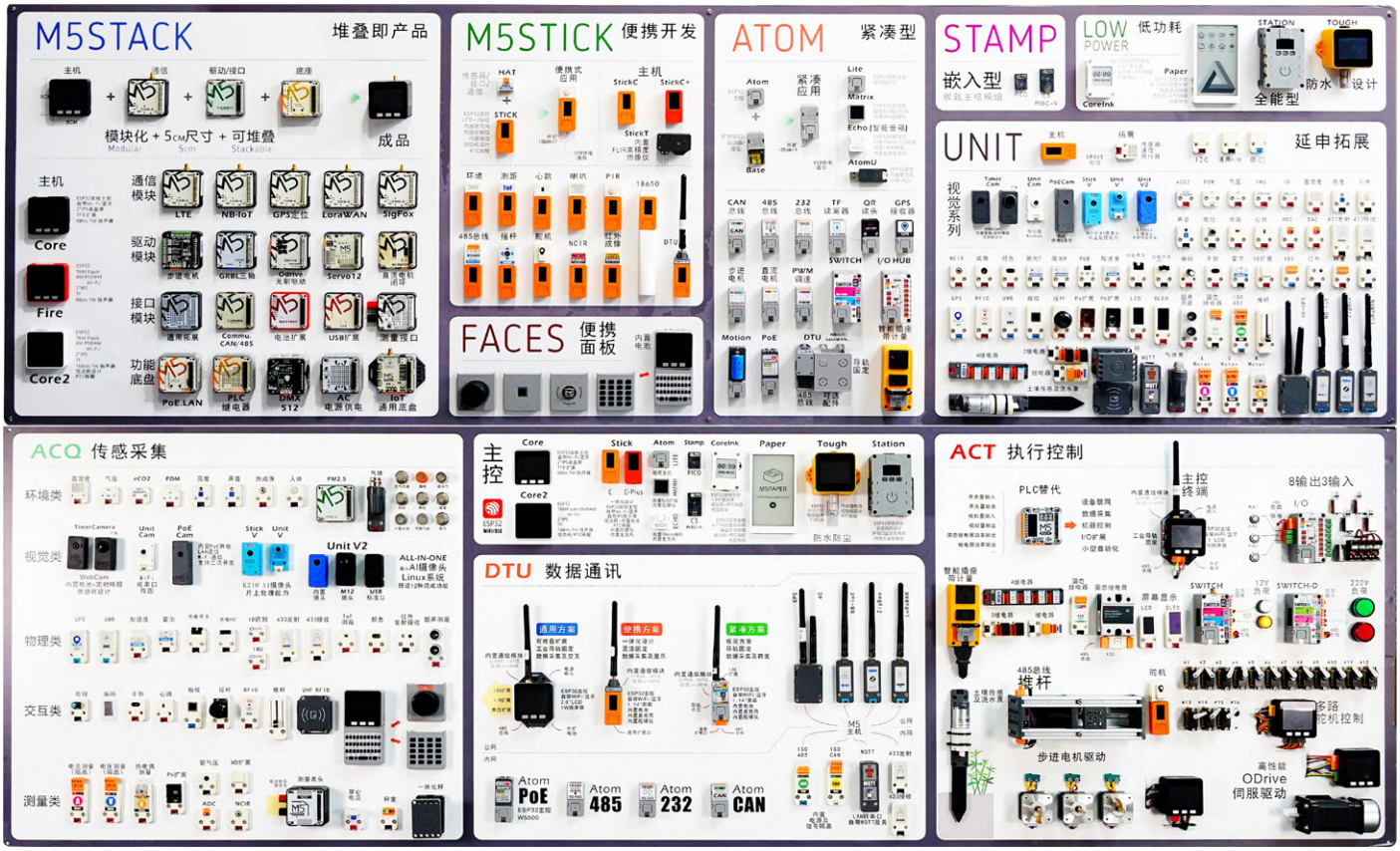


Figure 1: M5Stack Eco-system Family.

Figure 2: M5Dial is suitable for Smart Home.

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

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

New: The M5Dial

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

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

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

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

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

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

230662-01

WEB LINKS

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

Prototyping an ESP32-Based Energy Meter

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

By Saad Imtiaz (Elektor)

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

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

Design and Requirements

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

Microcontroller Selection

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

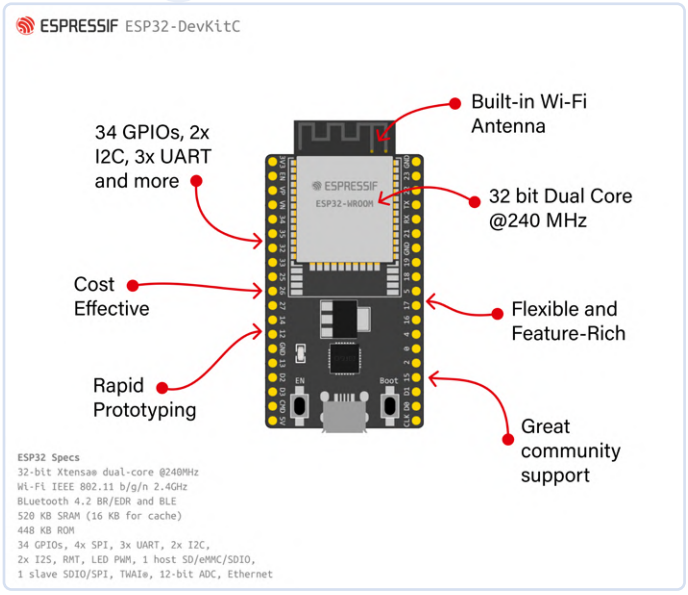


Figure 2: Main features and advantages of the ESP32.

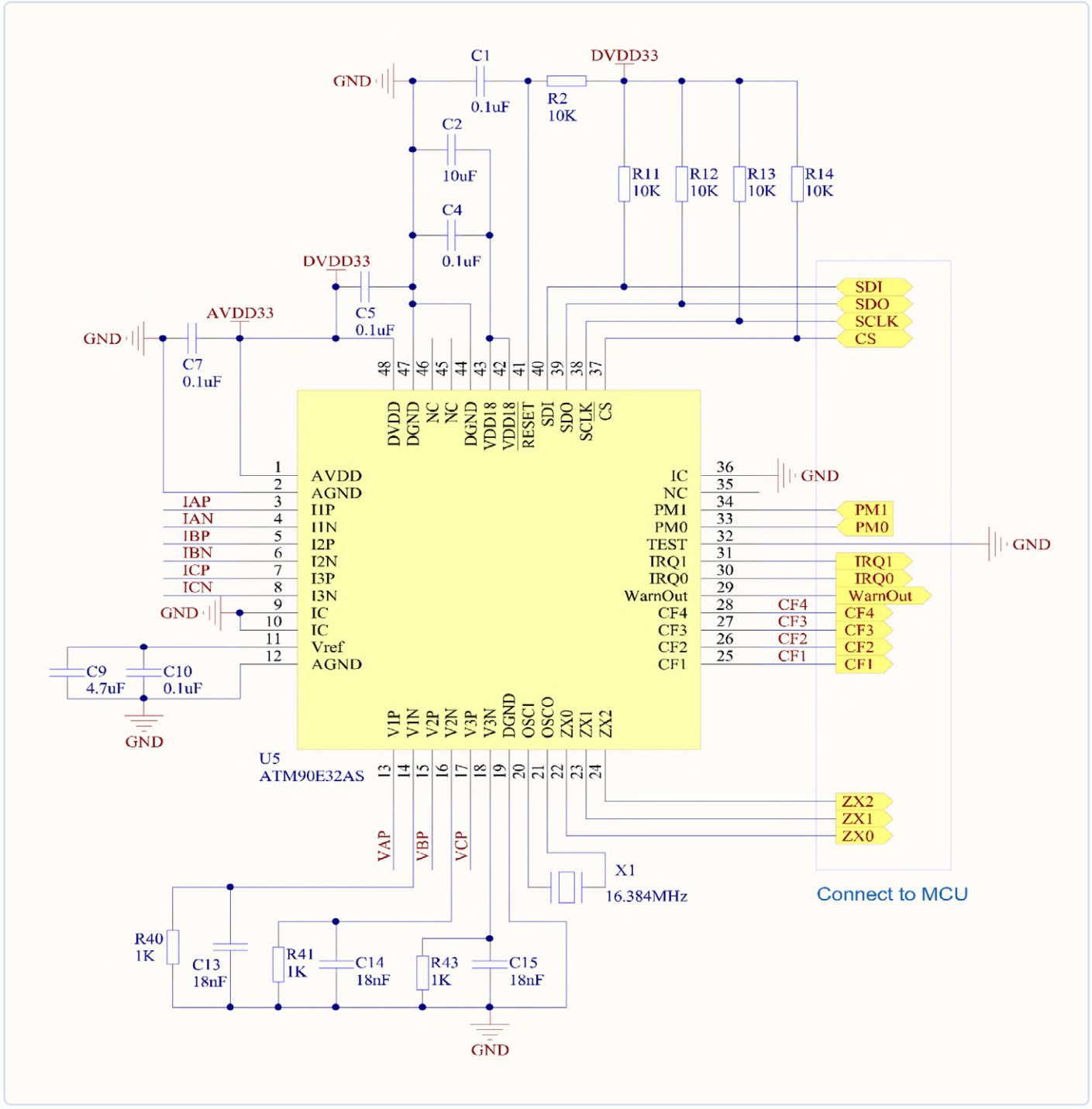


Figure 3: The energy meter is based on an application note by Atmel [2]. Here, you can see the circuitry around the metering IC.

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

Metering IC Integration

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

Design Phase and Electrical Safety Standards

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

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

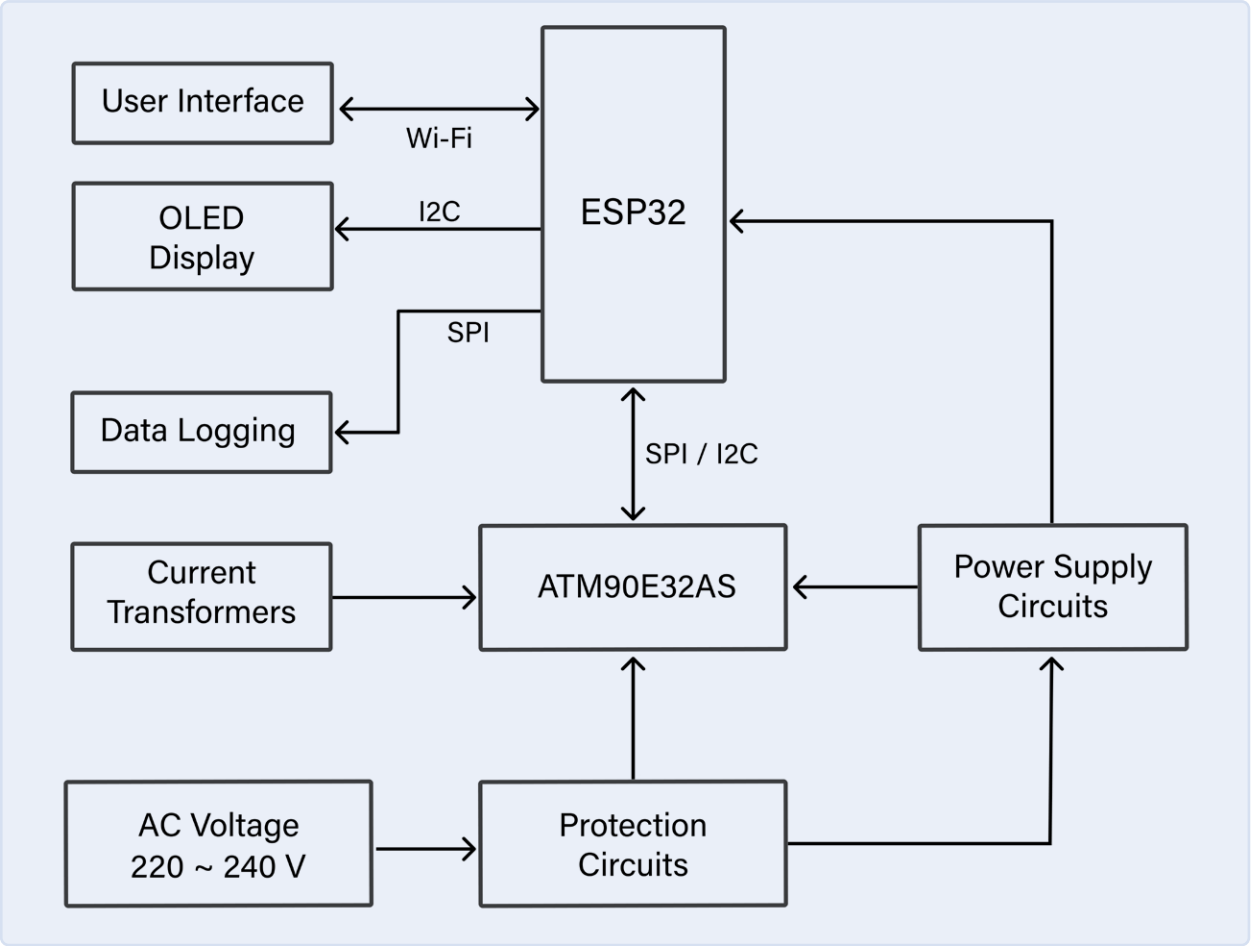


Figure 4: Block diagram of our Energy Meter project.

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

Manufacturing Selection: Opting for JLC PCB

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

Reflecting on the Journey and Looking Forward

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

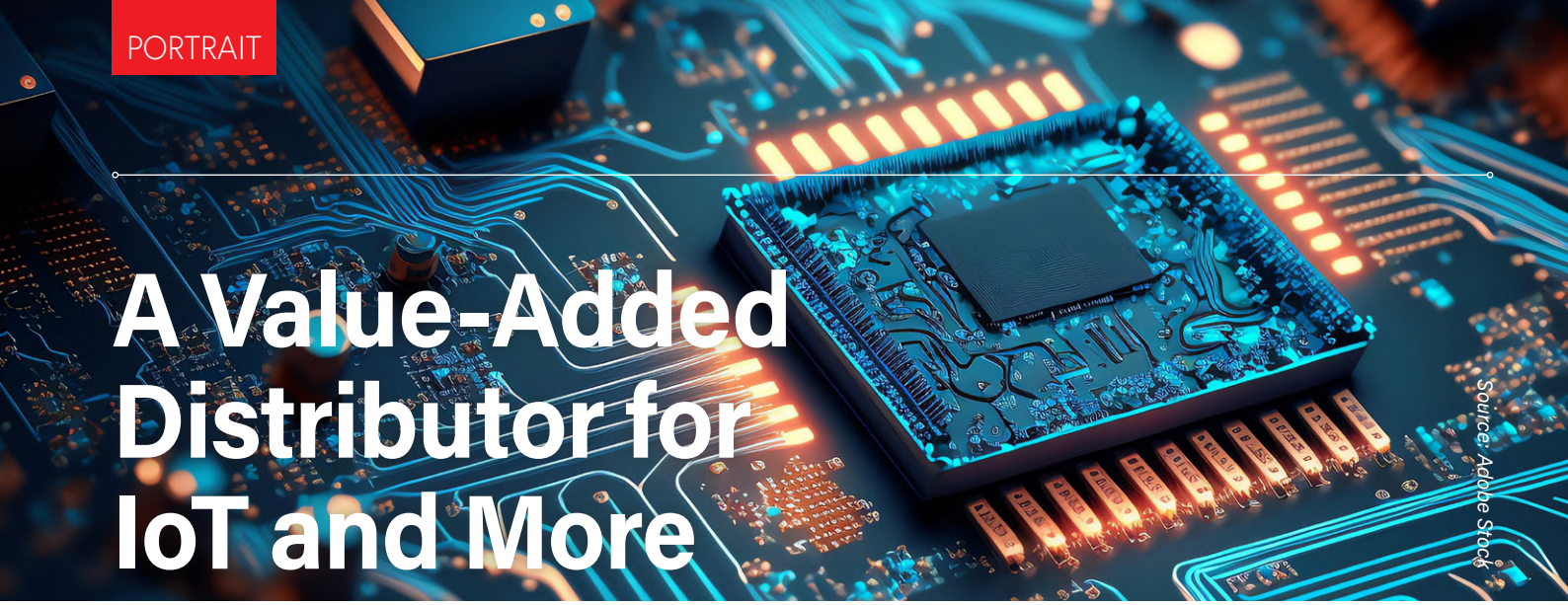
230646-01

Questions or Comments?

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

Related Products

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



A Value-Added Distributor for IoT and More

Source: Adobe Stock

Contributed by Steliau Technology

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

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

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

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

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

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

IoT Solutions and More

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

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

230661-01

Any requests?

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

WEB LINK

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

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

Alessandro Ranellucci and Martino Facchin Discuss Espressif Collaboration

Questions by Saad Imtiaz and Clemens Valens (Elektor)

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

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

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

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

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

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