



Jeroen Domburg
Standort: Singapur

Organisieren Sie Ihr Chaos

Auf meinem Schreibtisch herrscht ein so genanntes „chronologisches Ordnungssystem“. Also ein großer Haufen Mist, bei dem die zuletzt verwendeten Dinge oben und die ältesten unten liegen. Er wird neu geordnet, wenn ich keinen Platz mehr auf dem Schreibtisch finde, wenn ich vergesse, was unten liegt, oder wenn sich in dem Haufen Dinge befinden, die ich wegwerfen muss. Wenn Sie so sind, brauchen Sie sich nicht für einen „unordentlichen Schreibtisch“ zu entschuldigen: Solange Sie ihn nicht mit anderen teilen, kann Ihr Arbeitsplatz so aussehen, wie Sie wollen! Es scheint, dass Sie am besten funktionieren, wenn Sie Ihr Gedächtnis nutzen, um möglichst wenig Aufwand mit dem Weglegen und Wiederfinden Ihrer Werkzeuge zu haben, was Sie wahrscheinlich effizienter macht als die Leute, deren Schreibtisch immer sauber ist.

Langfristige Aufbewahrung ist eine andere Geschichte. Ich halte mich an das Motto „Wenn ich nicht weiß, dass ich es habe, kann ich es genauso gut darauf verzichten“, aber ich möchte mir nicht jedes einzelne Teil, das ich habe, aktiv merken müssen. Deshalb bewahre ich alle meine Bauteile, alte Projekte und so weiter in nummerierten Kisten auf, mit einer (sehr gut gesicherten!) Datei auf meinem PC, die

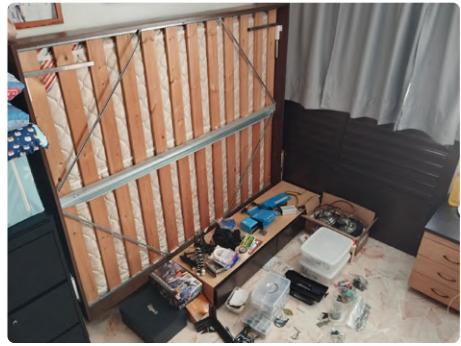
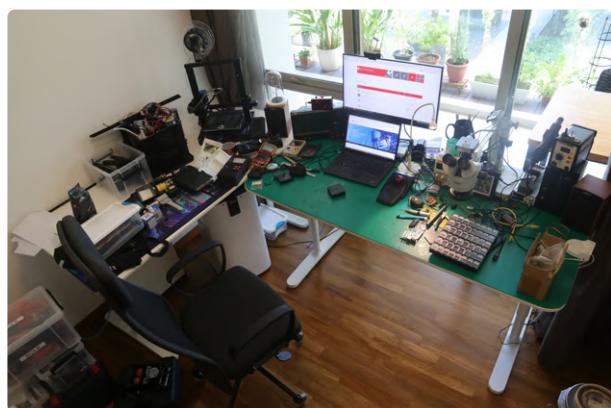
mir sagt, was wo ist. So kann ich ein Bauteil oder ein Tool leicht wiederfinden, wenn ich es brauche. Zu diesen Werkzeugen: Wenn Sie vorhaben, SMDs zu löten, sollten Sie sich ein gutes Mikroskop zulegen; wenn Sie ein binokulares bekommen können, umso besser! Ich finde, dass ein solches Mikroskop die Hände stabilisiert, so dass man auch kleinere Bauteile problemlos löten kann. Ein guter Lötkolben ist natürlich auch hilfreich, und gute Lötkolben sind immer billiger geworden. Selbst wenn der Weller WTCP 50 Ihres Vaters damals großartig war, ein moderner Pinecil oder TS-100 ist billiger und besser.

Unverzichtbar

- Computer. Ohne einen solchen kann man keine Chips programmieren.
- Moderne Lötzation. Mein derzeitiger Favorit ist Aixun T3A mit einem T245-Lötkolben.
- Luftreiniger
- Optisches binokulares Mikroskop
- Oszilloskop
- Logik-Analysator
- Labor-Netzteil

Wunschzettel

- Besseres Labor-Netzteil
- Platz, wo ich „schmutzige“ Sachen machen kann (zum Beispiel Harzdruck oder „Malerei“)
- Mehr freie Zeit, um mit Elektronik zu spielen



Kai Jie Tan
Standort: Singapur

Einfach irgendwo anfangen

Professionelle, voll ausgestattete Arbeitsplätze mit Geräten auf dem Schreibtisch und einer Stecktafel voller Werkzeuge in Reichweite sind das Nonplus-ultra, aber für den Anfang braucht man so etwas nicht. Vor Jahren habe ich nur mit einem Werkzeugkasten und einem Elektronikset angefangen, das jeder an meiner Ingenieurschule kaufen musste. Alles ist aus Platzgründen immer in den Schränken in meinem Zimmer verstaut. Als junger Erwachsener in Singapur, der in einem Schlafzimmer in der elterlichen Wohnung lebt, ist dies ein wichtiger Faktor. Angesichts der hohen Wohnungspreise und der langen Wartezeiten für Sozialwohnungen ziehen die meisten jungen Erwachsenen erst dann aus der Wohnung ihrer Eltern aus, wenn sie heiraten. Da mir bei jedem Projekt, das ich baue, schnell der Platz ausgeht, vor allem wenn es sich um größere Projekte handelt (zum Beispiel Cosplay-Requisiten und Elektroroller), habe ich meine Werkzeuge an den unmöglichsten Orten. Ein Beispiel dafür ist mein 3D-Drucker, der oben auf meiner Vitrine steht, zwei Meter über dem Boden. Auf dem Höhepunkt der COVID-19-Beschränkungen verlor ich meinen Zugang zu einem Fab Lab und musste alles zu Hause machen. Das führte dazu, dass ich ein Wand-Klapptbett mit Hardware von AliExpress baute. Das war eine große Umstellung, denn so kann ich meine laufenden Arbeiten sicher unter meinem Bett aufbewahren - zugänglich und mit geringem Zeitaufwand im Vergleich zu den bisherigen Kisten, in denen ich alles aufbewahren muss. Eine zweite Phase des Umbaus meines Zimmers sind Regale auf Gelenkviererecken, so dass die Regale zu den Bettbeinen werden, wenn sie heruntergeklappt werden. Das ist etwas, worüber ich noch schlafen muss (Wortspiel beabsichtigt). Mein Arbeitsbereich ist bei weitem nicht so groß wie der eines professionellen Fab Labs. Ich habe nur viel mehr Werkzeugkästen und Elektronikfächer als früher, aber ich hatte selten das Gefühl, dass mich das daran hinderte, zu bauen, was ich wollte. Ein Ratschlag: Wenn Sie Projekt um Projekt durchführen, werden Sie immer mehr Werkzeuge sammeln und verschlissene Werkzeuge ersetzen. Sie brauchen also nicht darauf zu warten, dass der Arbeitsbereich vollständig eingerichtet ist, um anzufangen. Fangen Sie einfach irgendwo an.

Unverzichtbar

- Multimeter Fluke 115
- Proskit 7-in-1-Abisolierzange
- 3D-Drucker (sehr fördernd, einen zu besitzen)

Wunschzettel

- Lötdampfabsaugung
- Luftreiniger
- Gehäuse für den 3D-Drucker
- Eine Garage, um Werkzeuge besser aufzubewahren

**Pedro Minatel**

Standort: Braga, Portugal

Einen Zufluchtsort schaffen

Ich bezeichne meinen Elektronik-Arbeitsplatz als mein Heiligtum - ein Ort, an dem ich Dinge erschaffen und erfinden kann (auch wenn die meisten Erfindungen immer noch in Arbeit sind). Ich bemühe mich, die meisten meiner Werkzeuge geordnet aufzubewahren, aber manchmal ist ein bisschen Unordnung unvermeidlich. Ich bin stolz auf meine Ausrüstung; einige Teile sind sogar älter als ich, wie mein Messschieber aus den 1950er Jahren.

Mein Lieblingsgerät und eines der wichtigsten Werkzeuge an meinem Elektronik-Arbeitsplatz

ist das Oszilloskop RTB2002 von Rohde & Schwarz. Ratschlag: Sorgen Sie für Ordnung bei den Komponenten! Ich verwende am liebsten preiswerte Behälter und Aufbewahrungsboxen (ähnlich wie bei Tabletten). Für ESD-empfindliche Bauteile ist es jedoch unerlässlich, sie in geeigneten ESD-Beuteln aufzubewahren. Ich verwende eine Etikettiermaschine, um dafür zu sorgen, dass alle meine Teile richtig gelagert sind. Aktuelles Projekt: Ich arbeite an einem weiteren Entwicklungsbrett, diesmal auf Basis des ESP32-C6.

Unverzichtbar

- Oszilloskop (mit Logik-Analysator)
- Zuverlässiges Multimeter
- Gute Lötkugelstation für SMD-Bauteile
- Gutes Basiswerkzeug
- 3D-Drucker

Wunschzettel

- Spektrum-Analysator mit Vektornetzwerkanalyse (z. B. SVA1032X)
- Benchtop-Bestückungsautomat (Pick-and-Place)
- Reflow-Ofen
- Digitales Netzgerät (60 V, 10 A)



Zeigen Sie Ihren Elektronik-Arbeitsplatz!

Möchten Sie, dass die Elektor-Redaktion Ihren Elektronik-Arbeitsplatz auf elektormagazine.de und/oder den Seiten der Zeitschrift Elektor vorstellt? Nutzen Sie unser Online-Formular und teilen Sie uns Details über den Raum mit, in dem Sie innovieren, entwerfen, debuggen und programmieren! www.elektormagazine.com/workspaces ↗

RG – 230579-02

**Jakob Hasse**

Standort: China, Shanghai



Der Mahjong-Tisch-Arbeitsbereich

Mein Arbeitsplatz wird hauptsächlich für alle Arten von Lötarbeiten, das Reflow-Löten von Leiterplatten und für alle Arten von Reparaturen an elektronischen Produkten verwendet. Ich habe einen Klon einer T12- Lötkugelstation, die aufgrund des direkten Heizelements in der Spitze sehr gut funktioniert. Lötdämpfe sind wirklich ungesund, deshalb habe ich auch einen Dunstabzug, den ich bei Taobao, einer chinesischen Einkaufsplattform von Alibaba für etwa 150 Dollar gekauft habe - gut investiertes Geld für meine Gesundheit. Für das Reflow-Löten von Leiterplatten habe ich eine Heizplatte, die so billig war, dass ich den Preis nicht mehr weiß. Aber wenn es um Messgeräte und „dumme“ Metallwerkzeuge geht, kaufe ich Qualität. Glauben Sie mir, das Leben ist zu kurz für minderwertige Werkzeuge!

Da ich derzeit in Shanghai wohne und Platz hier ein ziemlicher Luxus ist, ist mein Arbeitsplatz nur 80x80 cm² groß. Es ist ein normaler Tisch von Taobao mit einer Platte aus Bambus, einem wirklich harten und nachhaltigen Material. Wie sich jedoch herausstellte, löst sich

die Beschichtung auf, wenn sie mit IPA (nein, kein Indian Pale Ale, sondern Isopropylalkohol) in Kontakt kommt. Daher werde ich als Nächstes definitiv eine richtige Silikonmatte kaufen, um meinen Schreibtisch nicht nur vor Lötarbeiten, sondern auch vor IPA zu schützen. Auf diese Weise genieße ich den Vorteil des kleinen Formfaktors meines Arbeitsbereichs: Wenn ich Gäste habe, räume ich den Schreibtisch weg und stelle ihn ins Wohnzimmer: Die Größe ist perfekt, um Mahjong zu spielen.

Unverzichtbar

- T12-Lötkugelstation-Klon
- Weinan-Dunstabzugs-haube
- Vectech-Heizplatte
- iFixit-Reparaturset
- Maisheng MS-605D 300 W Labor-Netzteil
- Isopropylalkohol
- Metraline-DM62-Multimeter von Gossen-Metrawatt
- LUX-Schraubendreh-her für die gängigen Schraubentypen
- Knippex-Multifunktions-zange

Wunschzettel

- Echte Silikonmatte
- Oszilloskop
- Heißluft-Rework-Station
- Ersatz-Pinzette



Die ESP-RainMaker-Story

Wie wir „Ihre“ IoT-Cloud entwickelten

Von Amey Inamdar, Espressif

Mit einer Cloud verbundene Geräte machen eigentlich das IoT aus. Bei Anbietern von Cloud-Komplettlösungen verliert man aber schnell einen Teil der Kontrolle über die eigenen Daten. Allerdings bedeutet der Aufbau einer eigenen Cloud-Lösung von Grund auf einen erheblichen Aufwand. Als wir bei Espressif auf diese Probleme stießen, beschlossen wir, dafür eine Lösung anzubieten, die das Beste aus beiden Welten vereint.

Das „I“ in IoT steht bekanntlich für das Internet. Das Internet - und damit die Cloud-Konnektivität - ist ein integraler Bestandteil von vernetzten Geräten, die von den Vorteilen der Konnektivität profitieren. Per Cloud verbundene Geräte, die Daten senden und Befehle empfangen, schaffen nicht nur für die Anwender, sondern auch für die Gerätshersteller einen Mehrwert. Der Einsatz einer IoT-Cloud bringt jedoch Sicherheits- und Datenschutzaspekte für die Verbraucher sowie Skalierbarkeit, Sicherheit und größere technische Herausforderungen für Entwickler mit sich.
Es ist naheliegend, sich an verschiedene Anbieter von IoT-Komplettlösungen zu wenden, die ihre eigene, fertige Cloud-Plattform zusammen mit geräteseitigen und mobilen Anwendungen bereitstellten.

Dieses Vorgehen führte jedoch bald dazu, dass sich Gerätshersteller nicht ausreichend differenzieren konnten und auch nicht die erforderliche Kontrolle über die in der Cloud befindlichen Daten hatten. Einige Hersteller zogen eine ihre eigene, von Grund auf entwickelte Cloud vor, doch dies und die erforderliche Wartung ist ein erheblicher Aufwand. Als wir bei Espressif auf diese Probleme stießen, beschlossen wir, dafür eine Lösung anzubieten, die das Beste aus beiden Welten vereint. Wir wollten diese Cloud mit kompletter Funktionalität ausstatten und den Kunden die volle Verantwortung, Kontrolle und Anpassungsmöglichkeit geben, damit sie nicht bei null anfangen müssen.
Das war die Geburtsstunde von ESP RainMaker.

Die Wahl der Cloud-Architektur

Unsere erste Entscheidung bestand in der Wahl der Cloud-Architektur (**Bild 1**). Sie kennen vielleicht den Spruch, dass eine Cloud nur der Computer eines anderen ist. Dies ist tatsächlich wahr. Es kommt jedoch darauf an, wie der Computer eines anderen Anbieters zum Hosten Ihrer Anwendung verwendet wird. Mit dem Aufkommen von Virtualisierungstechnologien wurden leistungsstarke Server dazu genutzt, um mehrere Anwendungen auf derselben Hardware zu hosten, und zwar völlig isoliert voneinander. Mit virtuellen Maschinen wie etwa VMware kann man mehrere Instanzen von Betriebssystemen auf derselben Hardware ausführen, während mit Container-Technologien (Kubernetes, Docker) parallele Betriebsumgebungen auf derselben Hardware möglich sind. Das ist im Prinzip zwar großartig, aber bei beiden Architekturen muss man sich Gedanken darüber machen, welche Software ausgeführt und gewartet werden sowie wie die Skalierung bei steigender oder sinkender Gerätzahl erfolgen soll (das ist die Aufgabe von separaten DevOps-Ingenieuren). Nur um unseren Kunden ihre eigene

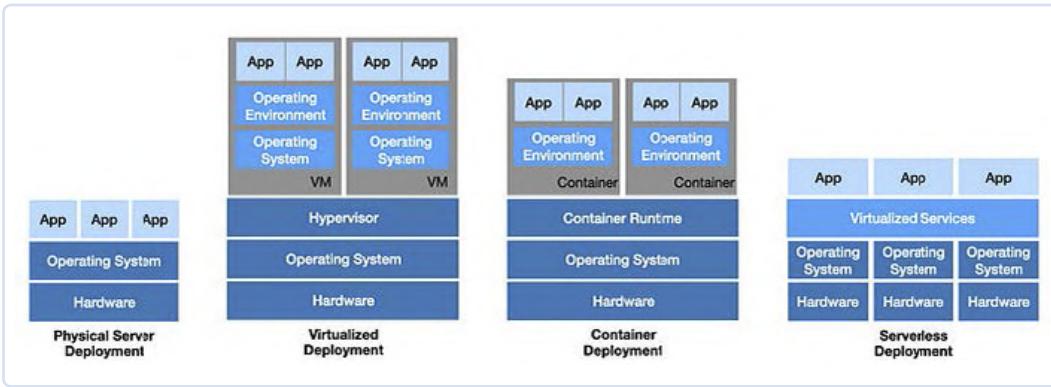


Bild 1: Evolution von Cloud-Architekturen.

IoT-Cloud anzubieten, konnten wir sie nicht mit diesem Aufwand beladen. An dieser Stelle war das recht neue Paradigma einer „serverlosen“ Cloud-Architektur hilfreich, das den Aufbau einer einfach zu wartenen IoT-Cloud erlaubt.

Diese serverlose Architektur schafft die Server aber nicht ab. Es werden lediglich Abstraktionen auf einer höheren Ebene bereitgestellt. So erhalten Sie beispielsweise einen MQTT-Broker, ohne sich Gedanken darüber machen zu müssen, welche MQTT-Software er verwendet, wie viele Geräteverbindungen er unterstützen kann oder auf welcher Plattform er läuft. Amazon Web Services (AWS) bietet solche verwalteten Dienste an, mit denen wir unsere IoT-Cloud implementieren konnten. Neben vielen anderen verwalteten Services bietet AWS IoT Core einen verwalteten MQTT-Broker, mit DynamoDB eine verwaltete noSQL-Datenbank, mit S3 Speicherplatz und - besonders wichtig: Lambda bietet „Function-as-a-Service“, bei dem man die Logik implementieren kann, ohne sich Gedanken darüber zu machen, wo sie ausgeführt werden soll.

Mit dieser auf AWS Managed Services basierenden Implementierung sieht ESP RainMaker aus wie eine Kombination aus der Konfiguration verschiedener Services und deren Interaktion sowie der Implementierung von Funktionen durch Lambda-Funktionen. Wichtig ist, dass es in jedem AWS-Konto bereitgestellt werden kann, was es uns leicht macht, eine IoT-Cloud-Implementierung zu erstellen, die unsere Kunden in ihrem eigenen AWS-Konto bereitstellen und die Daten und Anpassungen vollständig kontrollieren können.

Skalierbarkeit

Die IoT-Cloud muss sich in der Regel um eine große Anzahl von Geräten und Benutzern (über mobile Apps oder andere Clients wie Sprachassistenten) kümmern, die sich mit der Cloud verbinden und Nachrichten austauschen. Je nach Branche variiert die Anzahl. Bei Herstellern von Geräten für Verbraucher kann die Zahl der Geräte und Nutzer in die Millionen gehen. Daher muss sichergestellt werden, dass die Cloud-Implementierung auf eine große Anzahl skaliert werden kann. Da AWS die verschiedenen Services verwaltet, muss man sich keine Gedanken über die Skalierbarkeit des Cloud-Backends machen. Das

gilt zwar im Prinzip, doch jeder Service hat bestimmte Grenzen. So ist beispielsweise die maximale Anzahl von gleichzeitig im AWS-Konto ausgeführten Lambda-Funktionen begrenzt. Dies erfordert eine sorgfältige Überlegung in der Architektur, damit das System so aufgebaut ist, dass es viele Nachrichten via einer angemessenen Warteschlangen-Lösung verarbeiten kann. In der Geräteware und den mobilen Anwendungen muss es daher eine richtige Fehlerbehandlung geben. Heute ist ESP RainMaker schon für mehr als 3,5 Millionen Geräte und Nutzer getestet, die sich über die Cloud verbinden und kommunizieren (**Bild 2**).

Betriebskosten

In einer herkömmlichen Cloud-Architektur zahlt man in der Regel für Rechenleistung, Massen- und Arbeitsspeicher für die genutzten virtuellen Maschinen. In der serverlosen Architektur ändert sich sogar die Einheit der Abrechnung. Man muss hier für die tatsächlich genutzten Ressourcen zahlen, zum Beispiel für die Anzahl verbundener Geräte und die Gesamtverbindungszeit, die Anzahl an MQTT-Nachrichten, die Anzahl von Lese-/Schreibvorgängen in der Datenbank, die in S3 gespeicherte Datenmenge und die von ausgeführten Lambda-Funktionen verbrauchten Ressourcen. Auch wenn es sich hierbei um ein Pay-as-you-go-Preismodell handelt, kann das unter Umständen viel Geld kosten, wenn die Architektur und Implementierung nicht optimiert ist.

Die Betriebskosten für unsere Kunden waren eines der wichtigsten Kriterien bei der Entwicklung von ESP RainMaker. Das war einfach zu setzen: Wir wollten ein Cloud-Backend entwickeln, das sogar für eine intelligente Glühbirne funktioniert - wohl das günstigste IoT-Gerät. Die sorgfältige Auswahl der AWS-Dienste, damit ESP RainMaker dieses Ziel erreichen kann, war sehr aufwendig. Außerdem wurde die Implementierung erheblich optimiert, um eine optimale Anzahl von Datenbank-Lese- und Schreibvorgängen zu erreichen sowie die Implementierungssprache so zu wählen, dass möglichst wenig Ressourcen für die Ausführung benötigt werden.

ESP RainMaker hat derzeit mehrere Kunden, die kostengünstige Glühbirnen verkaufen. Auch hier stimmen die Betriebskosten.

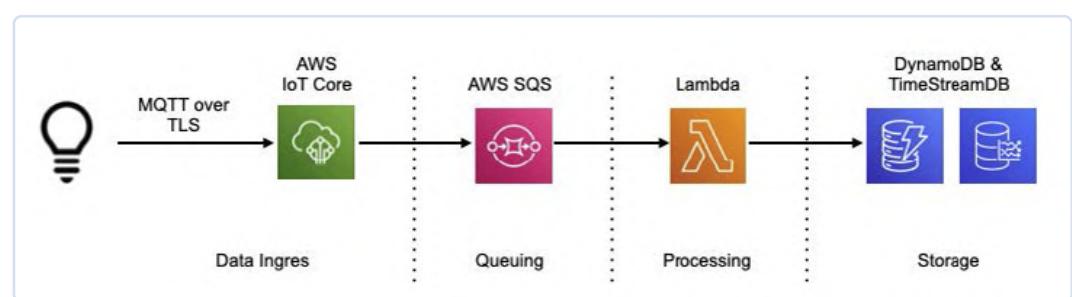


Bild 2: Beispiel des Processing von Device Messages in ESP Rainmaker.

Bild 3: Komponenten von ESP RainMaker.

Datensicherheit und Datenschutz

Die Nutzung von AWS Managed Services half auch bei der Gewährleistung der Datensicherheit. Es gibt Standardmethoden für Authentifizierung, Autorisierung, verschlüsselte Datenspeicherung, Anwendungs-Firewalls und so weiter. Der MQTT-Broker von AWS IoT Core bietet zum Beispiel eine auf X.509-Zertifikaten basierende Geräteauthentifizierung, die sowohl die Authentifizierung eines Geräts gegenüber der Cloud als auch durch die Cloud ermöglicht. IAM-Richtlinien ermöglichen eine fein abgestimmte Steuerung der Autorisierung. Die Web Application Firewall (WAF) bietet Schutz vor DoS- und DDoS-Angriffen auf die Endpunkte der RESTful-Dienste. ESP RainMaker verwendet all diese Sicherheitsmethoden in der vorgeschriebenen Weise und stellt eine angemessene Authentifizierung und Autorisierung mit Standardmethoden sicher. Es gibt keine unnötige Privilegien-Eskalation. Wir haben mit einem externen Test- und Zertifizierungsunternehmen zusammengetragen, um sicherzustellen, dass ESP RainMaker alle erforderlichen Funktionen zur Erfüllung der Datenschutzanforderungen implementiert, damit Kunden die GDPR-Compliance leicht einhalten können, wenn sie ESP RainMaker in ihrem eigenen Account einsetzen.

Eine IoT-Cloud allein ist nicht genug

Das Cloud-Backend ist zwar ein wichtiger Teil der kompletten IoT-Lösung, ist aber nicht erschöpfend. Aus Sicht angeschlossener IoT-Geräte gibt es neben der Cloud, mit der eine Verbindung hergestellt werden kann, weitere Aspekte: Es ist wichtig, über Geräte-Firmware, Handy-Apps zur Konfiguration und Nutzung der Steuerung sowie Sprachasistente-Integrationen zu verfügen, die für Nutzer die natürlichste Art der Kommunikation mit vernetzten Geräten im smarten Heim geworden sind (**Bild 3**).

Wir haben uns um alle diese Komponenten gekümmert. Das ESP RainMaker-SDK ist vollständig Open Source und lässt sich anhand der vielen Beispiele, die es bietet, leicht integrieren. Die Referenzanwendungen für iOS und Android sind ebenfalls vollständig Open Source. ESP RainMaker unterstützt Alexa und Google Assistant.

Eine Cloud wäre von geringem Nutzen, wenn sie nicht zur Verbesserung des Produkts durch technische und geschäftliche Erkenntnisse genutzt werden könnte. Zu diesem Zweck haben wir mit ESP-Insights ein Modul zur Beobachtung von Remote-Geräten entwickelt, das Remote-Geräteprotokolle, Crash-Analysen und Metrik-Überwachung bietet und vor allem umfangreiche Datenabfragen ermöglicht. Dieses Modul ist über das Web-Dashboard von ESP RainMaker verfügbar, das auch Remote-OTA, Gerätegruppierung, rollenbasierte Zugriffskontrolle und Geschäftseinblicke erlaubt.

Für die Zukunft gerüstet

Das Matter-Protokoll bringt die dringend benötigte Standardisierung für Smart-Home-Geräte. Da es sich bei Matter um ein lokales Netzwerkprotokoll handelt, kann die Cloud-Konnektivität als unabhängig von Matter betrachtet werden. Dennoch spielt die Cloud eine Rolle bei der Bereitstellung der Geräteverwaltung und des Fernzugriffs auf die Geräte. ESP RainMaker bietet Unterstützung für Matter Fabric, mit der man ein eigenes Matter-Ökosystem erstellen kann. Eine Kombination aus Cloud-Backend und Smartphone-Apps von ESP RainMa-



ker implementiert Matter Fabric mit der kompletten, dafür erforderlichen PKI-Infrastruktur. Die Geräte-, Benutzer- und ACL-Datenbanken werden sicher über mehrere Apps hinweg synchronisiert. Wenn man ein Smart-Home-Steuergerät mit RainMaker verbindet, kann man über mobile Anwendungen nicht nur die eigenen Geräte, sondern auch andere Matter-Geräte fernsteuern.

Wir haben auch die Unterstützung für die Integration der Mesh-Lite-Netzwerktopologie mit ESP RainMaker angekündigt. Mit Mesh-Lite kann man ein dediziertes WLAN-Mesh für IoT-Geräte einrichten, um eine größere Reichweite zu erzielen und tote Winkel im Haus abzudecken. ESP32-basierte IoT-Geräte fungieren als Zugangspunkte, die eine Verbindung zu anderen Geräten ermöglichen und eine Mesh-Topologie (eher eine Baumstruktur) bilden. Mit ESP RainMaker kann man einfach ein Mesh-Netzwerk erstellen und die Knoten darin einrichten.

Erste Schritte

Wir haben die ESP RainMaker-Cloud mit den gleichen Überlegungen entwickelt, die jeder Gerätehersteller bei der Implementierung seiner eigenen IoT-Cloud-Plattform anstellen muss. Das Wichtigste ist, dass Espressif eine vollständig anpassbare und Rebranding ermögliche Plattform bereitstellt. Für Entwickler werden eine von Espressif verwaltete ESP-RainMaker-Instanz zusammen mit über die jeweiligen App-Stores erhältlichen Smartphone-Apps geboten. Man kann das Web-Dashboard für die Geräteverwaltung nutzen und OTA-Updates durchführen. Damit lässt sich nicht nur die Funktionalität testen, sondern auch eigene Projekte für den persönlichen Gebrauch erstellt werden.

Für die ersten Schritte können Sie eines der Espressif-Entwicklungsboards verwenden und die in der Anleitung „Get Started“ [1] genannten Schritte befolgen.

Übersetzung von Dr. Thomas Scherer -- 230621-02

Über den Autor

Amey Indamar ist Direktor für technisches Marketing bei Espressif. Er verfügt über 20 Jahre Erfahrung im Bereich von Embedded-Systemen und vernetzten Geräten in den Bereichen Technik, Produktmanagement und technischem Marketing. Er hat mit vielen Kunden zusammengearbeitet, um erfolgreich vernetzte Geräte auf der Grundlage von WLAN- und Bluetooth zu entwickeln.

WEBLINK

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

Zusammenbau des *ELEKTOR-KITS CLOC 2.0*

Ein Elektor-Produkt, ausprobiert von Espressif

Von Jeroen Domburg, Espressif

Der Elektor Cloc 2.0 ist ein flexibler Wecker auf der Basis eines ESP32-Pico-Kits von Espressif. Er wird als Bausatz aus Teilen geliefert, die Sie selbst montieren müssen. In diesem Artikel haben die Mitarbeiter von Espressif den Wecker zusammengebaut und berichten über ihre Erfahrungen.

Wie wachen Sie morgens auf? Haben Sie einen Wecker auf Ihrem Handy, haben Sie noch einen alten Radio-wecker neben Ihrem Bett, werden Sie morgens vom Hahn geweckt oder lassen Sie einfach die Vorhänge offen, damit die ersten Sonnenstrahlen auf Ihr Gesicht treffen?

Sie haben vielleicht das Gefühl, dass all diese Methoden auch ihre Nachteile haben. Ihr Telefon erlaubt es Ihnen nicht wirklich, ein Auge halb zu öffnen, um zu sehen, wie viel Schlaf Sie noch haben können, der Radiowecker hat wahrscheinlich nur einen Alarm, der Sie auch am Wochenende oder an Feiertagen wecken wird, wenn Sie ihn nicht ausschalten, und obwohl der

Bild 1. Der Bausatz besteht aus einer Reihe von intelligent beschrifteten Beuteln und dem Hammond-Koffer.

Hahn und die Sonnenstrahlen billig und im Allgemeinen zuverlässig sind, ist es ziemlich schwierig, den Alarm bei ihnen neu zu konfigurieren. Im Allgemeinen ist keine der Lösungen wirklich flexibel.

Wenn Sie trotzdem flexibel sein wollen, ist die offensichtlich am besten konfigurierbare Option der Bau eines eigenen Weckers. Wenn Sie einen Wecker selbst entwerfen, können Sie alle gewünschten Funktionen einbauen. Wöchentliche Weckzeiten, WLAN-Konnektivität, ausgefallene Weckgeräusche, Anheben der Raumtemperatur, Einschalten der Kaffeemaschine, alles, was das Herz begehrts.

Eintritt in Cloc 2.0

Was aber, wenn Sie nicht die Fähigkeit, die Zeit oder die Energie haben, den gesamten Weg der Architektur, des Hardware- und Software-Designs und der Herstellung zu durchlaufen? Nun, Elektor kann hier mit der Cloc 2.0 helfen. Dieser Entwurf verfügt über eine doppelte 7-Segment-LED-Anzeige zur Anzeige der aktuellen Zeit und des Alarms, eine WLAN-Verbindung zum Internet, so dass er immer die korrekte Zeit anzeigt und viele konfigurierbare Optionen bietet. Darüber hinaus verfügt er über einen IR-Fernbedienungssender und -empfänger, mit dem Sie zum Beispiel ein Soundsystem einschalten können, wenn die Weckzeit gekommen ist. Das Beste daran ist, dass es auf einem ESP32-Pico-Kit basiert und die Software Open-Source ist, was bedeutet, dass Sie einfach eine neue Funktion hinzuzufügen können, ganz nach Belieben. Elektor war so nett, uns ein Kit zu schicken, damit wir es testen konnten. Das komplette Projekt von Cloc 2.0 ist auf der Website von Elektor Labs [1] veröffentlicht.

Der Bausatz

Die Cloc 2.0 ist ein Bausatz, und als solcher kam er in Teilen an (**Bild 1**). Am auffälligsten war das rote Hammond-Gehäuse, auf dem das Logo von Elektor Labs aufgedruckt war. Die Kombination aus rotem Gehäuse und roter LED-Anzeige sieht nicht nur gut aus, sondern verbessert auch den Kontrast der LEDs, so dass die Uhr besser ablesbar ist. Der Rest der Bauteile



befand sich in mehreren Tütchen, die mit den Werten der enthaltenen Komponenten beschriftet waren. Interessant ist, dass jede Tüte einen Satz erkennbar unterschiedlicher Bauteile enthält. Zum Beispiel waren in keiner Tüte mehr als zwei Widerstandswerte enthalten, so dass man nicht auf die Farbbänder der Widerstände starren muss, um herauszufinden, welches Teil welches ist. Das ist eine clevere Idee, die Plastik spart und trotzdem das Auffinden des richtigen Bauteils trivial macht; Hut ab vor dem Elektor-Team, das sich dieses System ausgedacht hat! Bemerkenswert ist, dass der Bausatz keine der von außen zugänglichen Knöpfe enthält - vielleicht finden Sie ja selbst noch irgendwo ein paar schicke Knöpfe in der Bastelkiste.

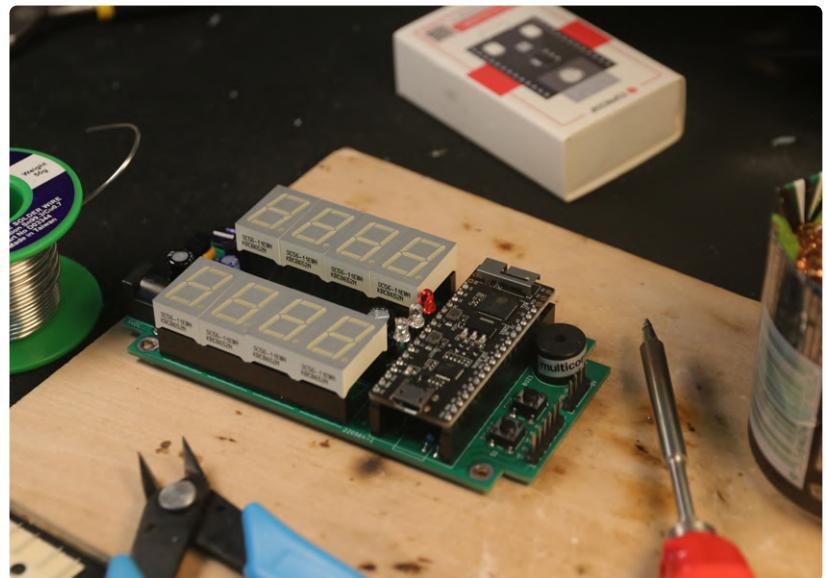
Die Dokumentation finden

Die Suche nach der Dokumentation zum Zusammenbau hätte etwas besser laufen können. Die Elektor-Website hat Seiten, die über etliche Domains (elektorlabs.com, elektor.com und elektormagazine.com) verteilt sind, mit unterschiedlichen Downloads und Texten auf jeder Seite. Das ist etwas verwirrend, und es wäre hilfreich gewesen, wenn es zumindest ein almodisches Blatt Papier mit Hinweisen zu den benötigten Downloads und Dokumentationen im Kit gegeben hätte. Nachdem wir jedoch alles gefunden hatten, erwies sich das Paket als ziemlich vollständig: Der Elektor-Artikel, der den Entwurf beschreibt, ist ebenso enthalten wie alle Downloads und sogar ein Video [2], das zeigt, wie man alles am besten zusammenbaut.

Mit der Dokumentation in der Hand ist der Zusammenbau der doppelseitigen Platine ein Kinderspiel (**Bild 2**). Dank der Durchkontaktierungen ist das Löten der Bauteile einfach, insbesondere, weil das Video eine Reihenfolge für das Löten der Komponenten vorschlägt, die bei uns sehr gut funktioniert hat. Eine beachtenswerte Sache ist, dass die Header für die 7-Segment-LED-Anzeigen der gleiche Typ sind, in den das ESP32-Pico-Kit passt. Dies ist nämlich ein kleines Problem, da die Pins für die LED-Anzeigen dünner sind und nicht immer einen guten Kontakt herstellen. Das ist aber verzeihlich, da es anscheinend keine Stifteleisten für dünne Pins mit der gleichen Höhe wie diese gibt; außerdem lässt sich das Problem leicht beheben, indem man die Pins der Displays ein wenig biegt.

Programmierung des ESP32-Pico-Kits

Nachdem das Board zusammengebaut war, musste das ESP32-Pico-Kit programmiert werden. Dies erforderte ein paar Schritte, um die richtigen Bibliotheken und das Board-Support-Paket zu installieren, war aber im Großen und Ganzen auch nicht allzu kompliziert. Ein solcher Prozess hätte zum Beispiel durch die Verwendung von ESP-Launchpad [3] noch etwas vereinfacht werden können, aber das aktuelle



Setup hat den Vorteil, dass es den Sketch aus dem Quellcode erstellt: Das bedeutet, dass man, wenn man etwas im Code ändern möchte, bereits alles hat, was man braucht, um die Firmware neu zu erstellen und zu flashen.

Mechanischer Aufbau

Beim Einbau der Platine in das Gehäuse gibt es ein wenig mehr Flexibilität, wie man es machen möchte: Elektor liefert ein Bohrmuster und in den neueren Bausätzen auch einige Kleinteile mit, so dass man eine Art Standard-Ausführung bauen kann. Es kann aber durchaus sein, dass Sie etwas ändern wollen oder müssen, um die Dinge so einzubauen, wie Sie es wollen. Je nach Form und Größe der Knöpfe, die Sie verwenden möchten, sollten Sie hier Ihren eigenen Bohrplan erstellen.

Die Stromversorgung

Obwohl wir sicher sind, dass wir (und Sie) Dutzende davon irgendwo haben, wurden wir in der großen Steckernetzteil-Dose nicht fündig. Wir konnten kein 5-V-Netzteil mit dem richtigen Klinkenstecker auftreiben. Was wir aber fanden, war eine kleine Platine mit einem USB-C-Anschluss, die wir modifizieren konnten, um einfach 5 V bei den erforderlichen 500 mA oder mehr auszugeben. So erfüllt jedes Steckernetzteil mit USB-C-Kabel den Zweck. Natürlich haben wir die Pläne für die Gehäusebohrung entsprechend geändert: Anstatt ein Loch für die Buchse zu bohren, haben wir ein Stück Plastik auf der Rückseite des Gehäuses für den USB-C-Anschluss weggeschnitten. Die USB-Platine konnte dann mit Epoxidharz an der hinteren Abdeckung sicher befestigt werden.

Bild 2. Da die Platine komplett durchkontakteert ist, war das Löten ein Kinderspiel.

Bild 3. Auf der Rückseite des Gehäuses ist genügend Platz. Hier passen die Tasten und Kabel hinein, aber man könnte dort auch eine Erweiterungsplatine verstauen.

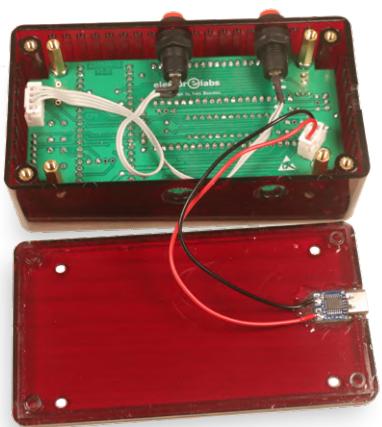




Bild 4. Unser fertiger Wecker, der über einen USB-C-Adapter mit Strom versorgt wird.

Sorgfältig bohren

An dieser Stelle noch ein Hinweis. Wenn Sie Löcher bohren und Bauteile einbauen, achten Sie auf die Reihenfolge der Dinge und die Ausrichtung des Gehäuses. Da wir hier bei Espressif Profis sind, sind wir uns dessen natürlich bewusst, und so können wir Ihnen versichern, dass die äh ... Abflusslöcher... im Boden unseres Gehäuses ja ganz offensichtlich von Anfang an sorgsam ausgedachter Teil unserer ursprünglichen Pläne waren (**Bild 3**).

Schließlich haben wir alles ordentlich zusammengebaut. Wir hatten noch ein paar hübsche rote Einbau-Knöpfe, die gut zum Gehäuse passen würden. Anstelle der mitgelieferten Stiftleisten verwendeten wir JST-Verbinder, um sie mit der Platine zu verbinden. Wir haben auch einen JST-Stecker verwendet, um die USB-Stromversorgungsplatine anzuschließen. Auf diese Weise können wir die Hauptplatine vollständig aus dem Gehäuse nehmen, falls sie überarbeitet oder repariert werden muss.

Konfigurieren der Cloc

Nachdem alles zusammengebaut war, war die Konfiguration der Cloc kinderleicht. Schalten Sie den Wecker ein und gehen Sie zur angegebenen IP-Adresse, und Sie können ihn mit dem lokalen WLAN verbinden. Von diesem Zeitpunkt an zeigt die Cloc bei jedem Start ihre zugewiesene IP-Adresse an. Das ist großartig, da man sich nicht mit Netzwerkscannern herumschlagen oder die Routerkonfiguration ansehen muss, um auf die Weboberfläche zuzugreifen. Die Weboberfläche selbst ist so einfach, dass man kaum ein Handbuch braucht, aber sehr vollständig: Wir hatten die Grundlagen wie Zeitzone und Sommerzeit in wenigen Augenblicken eingerichtet.

Alles in allem gefällt uns die Uhr sehr gut (**Bild 4**). Die LED-Anzeige ist gut lesbar, und die Tatsache, dass sie rot ist, bedeutet, dass sie nicht die Nachtsicht beeinträchtigt, wenn man im Dunkeln darauf schaut.

WEBLINKS

- [1] Projekt Cloc 2.0 auf der Elektor Labs Website: <https://elektormagazine.de/labs/cloc-le-reveil-20>
- [2] DIY-ESP32-Wecker - Bauanleitung Elektor Cloc 2.0 : <https://youtu.be/9VSLdFz6jyI>
- [3] ESP-Launchpad: <https://espressif.github.io/esp-launchpad>

Die Uhr ist über das Webinterface sehr gut konfigurierbar, und wenn man sie so modifizieren möchte, dass sie einen so weckt, wie man es bevorzugt, gibt es genug Raum dafür, sowohl im metaphorischen Sinne angesichts der niedrigen Einstiegshürde der Arduino-Umgebung als auch im physischen Sinne, da die Hammond-Box neben der Cloc selbst Platz für beispielsweise eine zusätzliche Platine bietet.

Vorschläge für Cloc 3.0

Es gibt einige Verbesserungen, die wir für einen Cloc V3 gerne sehen würden, zum Beispiel eine USB-Buchse zur Stromversorgung (Tipp: Wenn der Entwurf beispielweise einen ESP32-S3 verwenden würde, könnten die USB-GPIO-Pins, die er zur Verfügung stellt, auch an den USB-Stecker angeschlossen werden, was die Programmierung und das Debugging der Cloc ohne Öffnen des Gehäuses ermöglichen würde). Außerdem könnte der ESP32 vielleicht einen Lautsprecher ansteuern, um einen etwas weniger schrillen Weckton zu erzeugen als mit dem derzeitigen Piepser. Wir kennen den IR-Sender, der beispielsweise ein Radio einschalten kann, aber wir können uns vorstellen, dass heutzutage immer weniger Leute so etwas in ihrem Schlafzimmer haben. Andererseits, wenn das Dinge sind, die Sie wirklich brauchen, gibt es nichts, was Sie daran hindert, diese in die aktuelle Iteration der Cloc zu integrieren, angesichts der Offenheit des Entwurfs und der Software. 

RG - 230561-02

Haben Sie Fragen oder Kommentare?

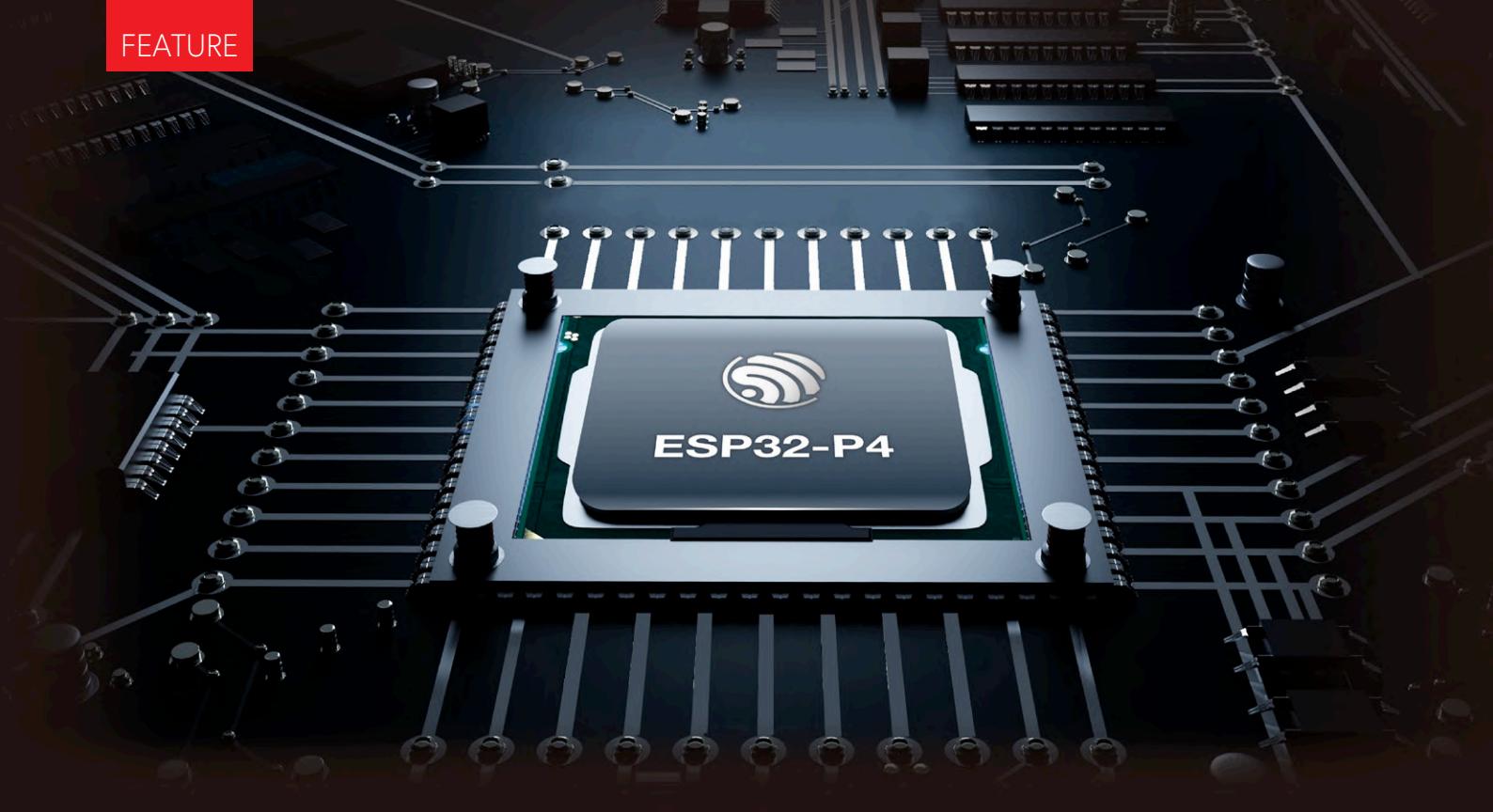
Bei technischen Fragen können Sie sich an die Elektor-Redaktion wenden: redaktion@elektor.de.



Passende Produkte

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





Ausgepackt: der ESP32-P4

Die nächste Ära der Mikrocontroller

Von Anant Raj Gupta, Espressif

Willkommen in der nächsten Ära der Mikrocontroller! Hier kommen erschwingliche Sicherheit, modernste Leistung und unvergleichliche Konnektivität zusammen. Der Hochleistungs-Mikrocontroller verspricht, die Welt der eingebetteten Systeme neu zu gestalten. Zudem öffnet er Entwicklern, Ingenieuren und der gesamten IoT-Community die Türen zu vielen neuen Möglichkeiten.

Das Herzstück dieses Technologiesprungs ist der ESP32-P4. Es handelt sich um ein System-on-Chip (SoC), das sorgfältig für hohe Leistung entwickelt wurde. Zudem hat es erstklassige Sicherheitsfunktionen. Der leistungsstarke Mikrocontroller unterstützt einen umfassenden internen Speicher, verfügt über beeindruckende Bild- und Sprachverarbeitungsfunktionen und besitzt Hochgeschwindigkeits-Peripherieeinheiten (**Bild 1**). Damit lassen sich die anspruchsvollen Anforderungen der nächsten Ära von Embedded-Anwendungen erfüllen.

Rechenleistung des ESP32-P4

Leistungsstarkes CPU- und Speicher-Subsystem

Die Leistungsfähigkeit des ESP32-P4 liegt in seiner Dual-Core-RISC-V-CPU begründet, die mit bis zu 400 MHz getaktet wird. Das Kraftpaket ist zusätzlich mit einfach präzisen Gleitkommaeinheiten (FPUs) und KI-Erweiterungen ausgestattet und bietet daher ein ganzes Arsenal an Rechenressourcen. Ergänzt wird dies durch einen LP-Core, der mit bis zu 40 MHz läuft. Diese asymmetrische „Big-Little“-Architektur ist entscheidend für die Unterstützung von Anwendungen, die einerseits eine extrem niedrige Stromaufnahme besitzen müssen, andererseits aber gelegentlich eine hohe Rechenleistung erfordern.

Wegen seiner außergewöhnlicher Leistung steht der ESP32-P4 hoch im Kurs, wie **Bild 2** anhand eines Leistungsvergleichs mit anderen Espressif-Produkten wie dem ESP32 und dem ESP32-S3 verdeutlicht.

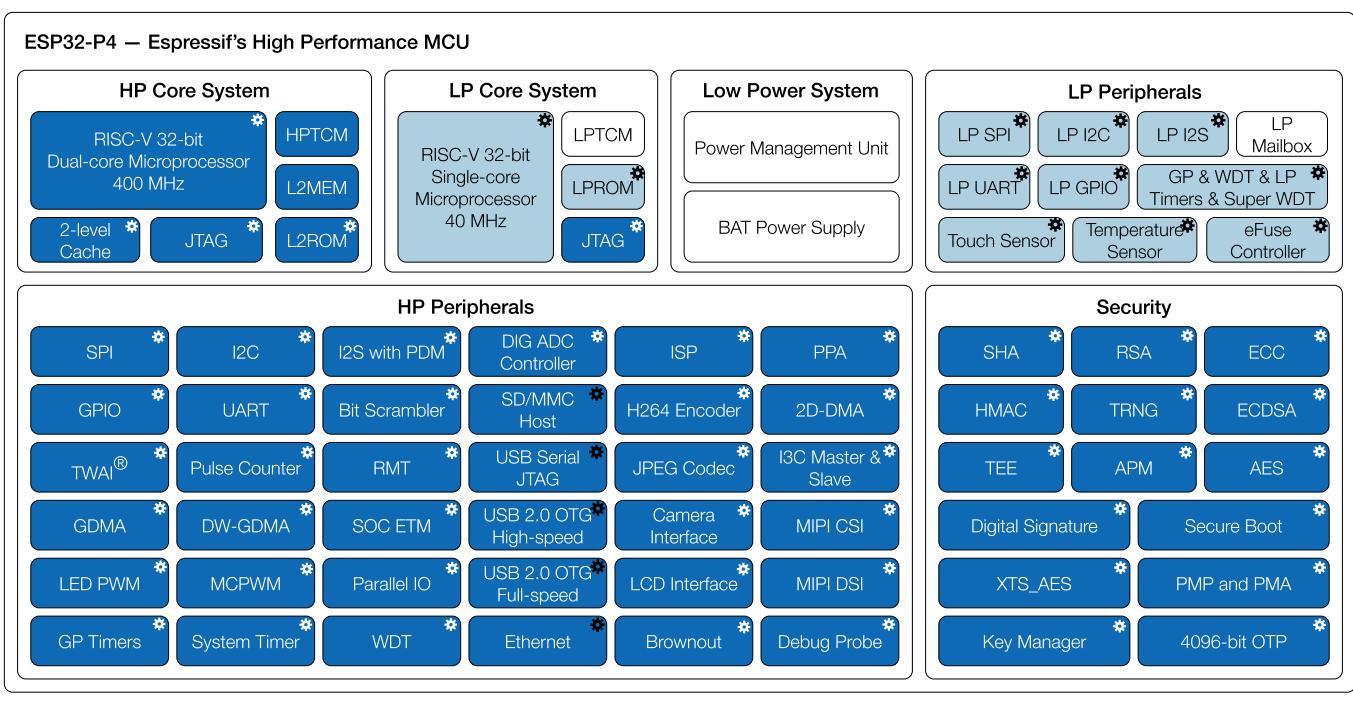


Bild 1. Blockschaltbild des ESP32-P4.

Speicherarchitektur für unübertroffene Effizienz

Der Speicherzugriff ist ein entscheidender Faktor für eine nahtlose Leistung. Das Speichersystem des ESP32-P4 ist höchst effizient. Mit 768 KB On-Chip-SRAM, das sich als Cache konfigurieren lässt, wenn externes PSRAM verfügbar ist, und 8 KB Zero-Wait-TCM-RAM für schnelle Datenpufferung sorgt dieses SoC dafür, dass Speicherzugriffslatenz und -kapazität niemals die Flaschenhälse für eine Anwendungen darstellen können.

Außerdem ist der externe Speicher des ESP32-P4 direkt zugänglich und bietet satte 64 MB zusammenhängenden Platz für den Zugriff auf Flash und PSRAM über den Cache. Der interne, 768 KB große Speicher lässt sich als L2MEM konfigurieren – geteilt für Befehle und Daten. Ergänzt wird die Sache durch einen dedizierten 16 KB großen L1-Befehls-Cache und einen 64 KB großen L1-Daten-Cache. Ein zusätzlicher 8 KB großer, eng mit dem HP-Kern gekopelter Speicher gewährleistet den Zugriff auf gespeicherte Daten in einem einzigen Zyklus. Das PSRAM-SPI ermöglicht, sechzehnzeilige DDR-Lese- und Schreibvorgänge durchzuführen und unterstützt gleichzeitig den Betrieb mit einer Taktrate von 250 MHz, was einen maximalen Datendurchsatz von 1 GB/s ergibt. Diese mehrstufige Speicherarchitektur wurde entwickelt, um den Speicherzugriff für Anwendungen nahezu transparent zu machen. Für Hochleistungsanwendungen ist das von großem Vorteil.

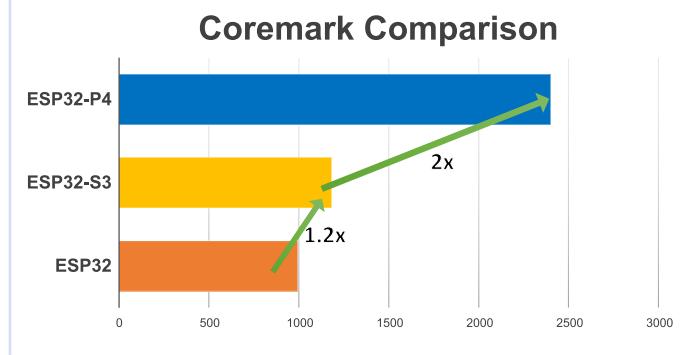


Bild 2. Coremark-Vergleich.

Maßgeschneiderte Verarbeitung für KI- und DSP-Workloads

Der ESP32-P4 verfügt über einen 32-bit RISC-V Dual-Core Prozessor, der die Standard-RV32IMAFZC-Z-Erweiterungen unterstützt. Was den Controller jedoch wirklich auszeichnet, ist sein spezieller, erweiterter Befehlssatz. Dieser Befehlssatz wurde entwickelt, um die Anzahl der Befehle in Schleifen zu reduzieren und so die Leistung erheblich zu steigern. Außerdem enthält das SoC kundenspezifische KI- und DSP-Erweiterungen, die die Betriebseffizienz bestimmter KI- und DSP-Algorithmen optimieren. Dank der benutzerdefinierten Vektorbefehle lässt sich der ESP32-P4 für Aufgaben wie neuronale Netzwerke und Deep Learning einsetzen. Es sind beschleunigte und effiziente Berechnungen möglich.

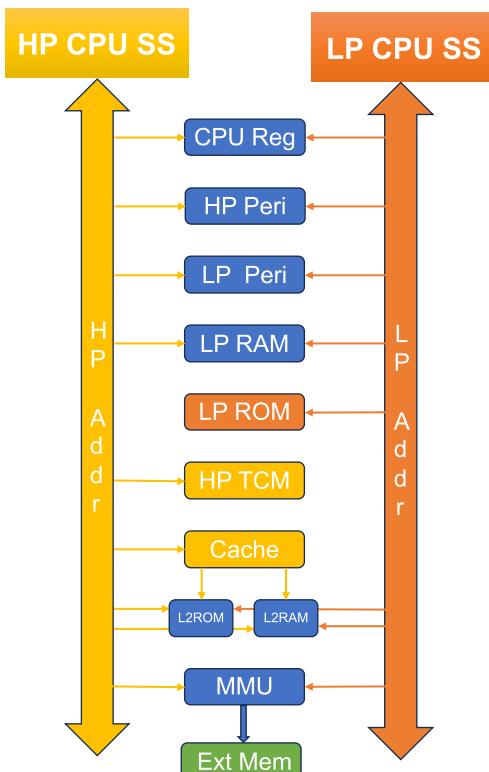


Bild 3. Zugriff auf den Adressraum des ESP32-P4.

Effizienter Speicherzugriff für jeden Kern

Wie in Bild 3 dargestellt, sorgt die Architektur des Speicherzugriffs beim ESP32-P4 dafür, dass alle Peripherieeinheiten und der Speicher sowohl von den HP- als auch von den LP-Cores zugänglich sind. Auf diese Weise kann der LP-Kern die meisten Funktionen ausführen, während der HP-Kern nur dann in Erscheinung tritt, wenn Hochleistungsberechnungen erforderlich sind. Ferner ermöglicht die Peripherie des LP-Kerns kritische Funktionen auch im niedrigsten Energiemodus. Mit diesen Eigenschaften ist der ESP32-P4 die ideale Wahl für Anwendungen, die hohe Rechenleistung bei gleichzeitig niedrigem Stromverbrauch über längere Zeiträume erfordern.

Die Zukunft sichern: erstklassige Sicherheit

Secure Boot

Sicherheit ist kein hübsches Beiwerk, sie ist der Eckpfeiler der Entwicklung des ESP32-P4. Das SoC wurde sorgfältig entwickelt, um erschwingliche Sicherheitslösungen für alle Anwendungen zu bieten. Secure Boot, eine zentrale Sicherheitsfunktion, schützt das Gerät vor der Ausführung von nicht autorisiertem und unsigniertem Code. Jede Software wird gründlich überprüft, einschließlich des Bootloaders der zweiten Stufe, und jede Anwendungsbinarydatei. Wie in Bild 4 gezeigt, bleibt der Bootloader der ersten Stufe als ROM-Code unveränderlich und muss nicht signiert werden. Der ESP32-P4 bietet Flexibilität, indem er sowohl RSA-PSS als auch ECDSA-basierte sichere Boot-Verifizierungsverfahren unterstützt. ECDSA bietet dabei eine vergleichbare Sicherheit wie RSA, allerdings mit kürzeren Schlüsseln.

Flash-Verschlüsselung

Die Flash-Verschlüsselung ist ein entscheidender Faktor für den Schutz des Inhalts des Off-Chip-Flash-Speichers des ESP32-P4. Ist diese Funktion aktiviert, wird die Firmware zunächst als Klartext geflasht und anschließend beim ersten Booten verschlüsselt. Das bedeutet, dass physische Versuche, den Flash-Speicher zu lesen und damit verwertbare Daten zu erlangen, zum Scheitern verurteilt sind. Ist die Flash-Verschlüsselung aktiviert, werden alle speicherbezogenen Lesezugriffe auf den Flash zur Laufzeit transparent und sicher entschlüsselt. Der ESP32-P4 verwendet den XTS-AES-Blockchiffiermodus mit einem robusten 256-Bit-Schlüssel für die Flash-Verschlüsselung und gewährleistet somit erstklassigen Datenschutz.

Kryptografische Hardware-Beschleunigung

Der ESP32-P4 ist mit einer umfassenden Suite von Hardware-Kryptobeschleunigern ausgestattet. Sie unterstützen diverse Standardalgorithmen für Netzwerk- und Sicherheitsanwendungen. Die Beschleuniger unterstützen unter anderem AES-128/256, SHA, RSA, ECC und HMAC. Die kryptografischen Möglichkeiten verbessern die Fähigkeit des ESP32-P4, Datenübertragung, -speicherung und -authentifizierung zu schützen. Daher ist das SoC eine ausgezeichnete Wahl für sicherheitssensible Anwendungen.

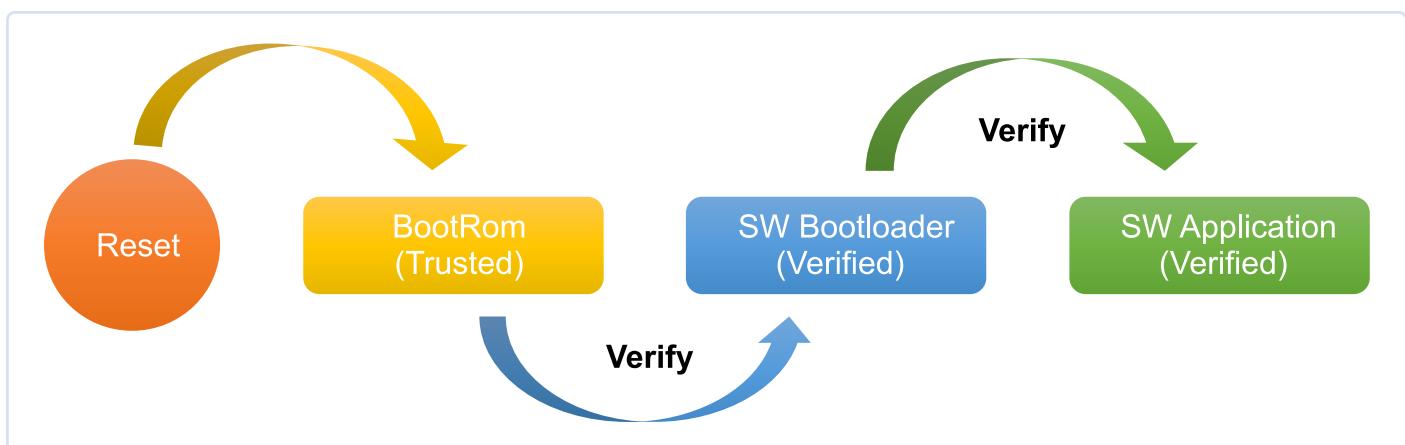


Bild 4. Flussdiagramm Secure Boot.

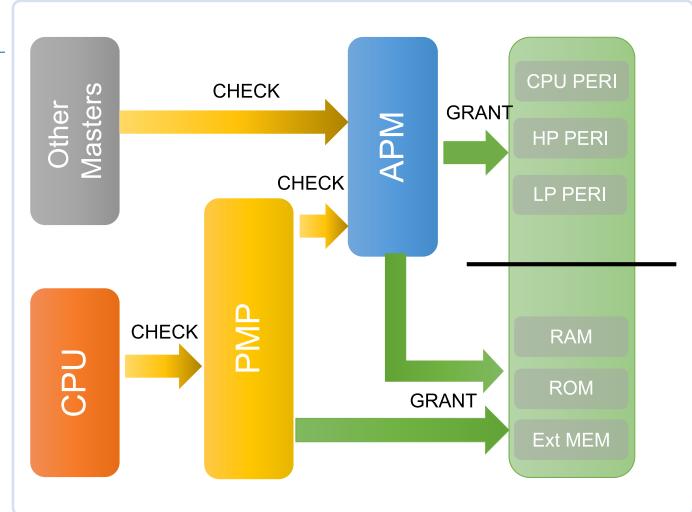


Bild 5. Zugriffskontrolle beim ESP32-P4.

Schutz privater Schlüssel

Der Schutz privater Schlüssel ist essenziell, und der ESP32-P4 setzt auf robuste Mechanismen, um ihre Sicherheit zu gewährleisten. Das SoC generiert private Schlüssel auf dem Chip, die für Software- oder physische Klartext-Angriffe unzugänglich sind. Hardware-beschleunigte digitale Signaturen werden erzeugt, sodass Anwendungen Signiervorgänge mit dem privaten Schlüssel des verschlüsselten Geräts durchführen können. Damit wird kein Klartext verarbeitet. Der ESP32-P4 nutzt einen Schlüsselmanager, der physikalisch nicht erklärbare Funktionen (PUF) verwendet, die für jeden Chip einzigartig sind, um den eindeutigen Hardware-Schlüssel (HUK) zu erzeugen. Dieser HUK dient als Root of Trust (RoT) für den Chip und wird bei jedem Einschaltvorgang automatisch erzeugt und verschwindet, wenn der Chip ausgeschaltet wird. Der hardwaregesteuerte Schutz des privaten Schlüssels des ESP32-P4 gewährleistet unvergleichliche Sicherheit für sensible Vorgänge.

Zugriffskontrolle

Der ESP32-P4 hebt mit einem Hardware-Zugriffsschutz die Zugriffskontrolle auf ein neues Niveau. Er ermöglicht die Verwaltung von Zugriffsrechten und die Trennung von Privilegien (**Bild 5**). Das System besteht aus zwei Komponenten: physischer Speicherschutz (PMP) und Zugriffsrechteverwaltung (APM). PMP verwaltet den CPU-Zugriff auf alle Adressräume, während APM den Zugriff auf ROM und SRAM regelt. PMP muss den CPU-Zugriff auf ROM und HPSRAM genehmigen. Erst dann kommt APM für andere Adressräume ins Spiel. Wird die PMP-Erlaubnis verweigert, werden keine APM-Prüfungen ausgelöst. Dieser mehrschichtige Ansatz gewährleistet eine robuste Zugriffskontrolle und macht den ESP32-P4 zu einer sicheren Wahl für viele Anwendungen.

Viel Peripherie und eine Mensch-Maschine-Schnittstelle

Erweitertes visuelles und haptisches Erlebnis

Der ESP32-P4 verbessert die Mensch-Maschine-Schnittstelle (HMI) durch die Unterstützung von MIPI-CSI (Camera Serial Interface) und MIPI-DSI (Display Serial Interface). Durch die integrierte Bildsignalverarbeitung (ISP) an der MIPI-CSI-Schnittstelle kann der SoC die wichtigsten Eingangsformate wie RAW8, RAW10 und RAW12 verarbeiten. Zudem unterstützt das System Auflösungen bis zu Full HD (1980×1080) bei bis zu 30 Bildern pro Sekunde (fps). Daher ist er ideal für Anwendungen wie IP-Kameras (**Bild 6**) und Video-Türklingeln, die hochauflösende Kameraeingänge erfordern. Bezüglich des Displays unterstützt die MIPI-DSI-Schnittstelle des ESP32-P4 zwei Kanäle mit 1,5 Gbps, was eine Display-Unterstützung für 720p-HD bei 60fps oder 1080p Full-HD bei 30fps bedeutet. Durch die Integration von kapazitiven Toucheingängen und Spracherkennungsfunktionen eignet sich der ESP32-P4 perfekt

für alle HMI-basierten Anwendungen, von interaktiven Bedienfeldern bis hin zu Laderegbern.

Medienverarbeitung

Der ESP32-P4 eignet sich nicht nur für Kameras und Bildschirme, sondern auch für die Codierung und Komprimierung von Medien. Mit eingebauter Unterstützung für H.264 und andere Codierungs-Formate ermöglicht der ESP32-P4 Videostreaming und -verarbeitung. Diese Funktionen lassen sich nutzen, um preisgünstige IP-Kameralösungen zu erstellen und die bereits erwähnte umfangreiche HMI-Peripherie zu nutzen. Das SoC verfügt außerdem über einen integrierten Hardware-Pixelverarbeitungsbeschleuniger (PPA), der für die GUI-Entwicklung geeignet ist.

Umfassende GPIO- und Peripherie-Unterstützung

Mit bemerkenswerten 55 programmierbaren GPIOs setzt der ESP32-P4 einen neuen Standard für Espressif-Produkte. Er unterstützt verschiedene häufig verwendete Peripherieeinheiten, darunter SPI, I²S, I²C, LED-PWM, MCPWM, RMT, ADC, DAC, UART und TWAITM. Der ESP32-P4 unterstützt USBOTG 2.0HS, Ethernet und SDIO Host3.0 für Höchstgeschwindigkeit-Verbindungen.

Nahtlose drahtlose Verbindung

Für Anwendungen, die eine drahtlose Verbindung erfordern, lässt sich der ESP32-P4 mühelos mit Produkten der ESP32-C/S/H-Serie als drahtloser Zusatzchip kombinieren. Das lässt sich über SPI/SDIO/UART mit ESP-Hosted oder ESP-AT Lösungen erreichen. Beim Einsatz einer ESP-Hosted-Lösung bleibt die Anwendungsentwicklung gleich beim Einsatz eines SoC, dank integrierter drahtloser Konnektivität. Zudem kann der ESP32-P4 als Host-MCU für andere Verbindungs-Lösungen wie ACK und AWS IoT Express-Link dienen. Die Entwickler können sich auf das vertraute und ausgereifte IoT-Entwicklungs-Framework von Espressif (ESP-IDF) verlassen, das bereits Millionen von vernetzten Geräten unterstützt.



Bild 6. Beispielhafter Datenfluss für eine IP-Kamera.

Mit ESP32-P4 in die Zukunft starten

Der ESP32-P4 steht für eine neue Ära von Mikrocontrollern. Er bietet Leistung, Sicherheit und Konnektivität, womit Entwickler ihre kühnsten Ideen verwirklichen können. Der ESP32-P4 ist eine vielseitige, sichere und leistungsstarke Basis.

Die Zukunft des Internets der Dinge (IoT) ist vielversprechend und der ESP32-P4 ist bereit, führend bei dieser Entwicklung zu sein. Das SoC ermöglicht eine schnellere und effizientere Verarbeitung. Edge Computing wird die Informationen näher an die Datenquellen bringen. Dank künstlicher Intelligenz und maschinelles Lernen werden IoT-Geräte intelligente und autonome Entscheidungen treffen können. Der ESP32-P4 ist bereit für diese Veränderungen und bietet Rechenleistung, robuste Sicherheit und eine dynamische Entwicklergemeinschaft.

Der ESP32-P4 ist im Universum der Mikrocontroller ein gewaltiger Sprung. Er ist nicht nur ein Chip, er ist eine Einladung zur Innovation. Egal, ob Sie ein erfahrener Entwickler oder ein ambitionierter Neuling sind, mit dem ESP32-P4 können Sie Ihre Ideen in die Realität umsetzen. Wagen Sie den Sprung, entdecken Sie das Potenzial des ESP32-P4 und werden Sie ein Teil der revolutionären Reise, die dieser Mikrocontroller ermöglicht!

Gemeinsam können wir die Zukunft des IoT gestalten und eine besser vernetzte, effizientere und sicherere Welt schaffen. ↗

Übersetzung von Jürgen Donauer – (230615-02)

Haben Sie Fragen oder Kommentare?

Haben Sie technische Fragen oder Kommentare zu diesem Artikel? Kontaktieren Sie Espressif über den QR-Code oder Elektor unter redaktion@elektor.de.



Über den Autor

Anant Gupta ist Technical Marketing Manager mit mehr als 15 Jahren Erfahrung in den Bereichen Systemarchitektur, IP-Architektur und SoC-Design. Seine Leidenschaft ist es, Innovationen voranzutreiben und Kundenprobleme bei Espressif Systems zu lösen.



Passende Produkte

- **Espressif ESP32-Reihe**
www.elektor.de/espressif



Finden Sie die spannendsten ESP-IDF-Komponenten

Node.js hat *npm* und Python hat *pip*, aber auch ESP-IDF verfügt auch über einen Komponentenmanager, mit dem sowohl von Espressif bereitgestellte als auch Komponenten von Drittanbietern in Ihr Projekt importiert werden können.

Komponenten bestehen aus einer Softwarebibliothek mit Beispielen und Dokumentationen, die Ihnen helfen, sie in Ihrem Projekt zu verwenden. Hier finden Sie eine Vielzahl von Softwarebibliotheken für Ihr Projekt, von verschiedenen Peripherietreibern über Middleware (zum Beispiel LVGL-Portierung) bis hin zu verschiedenen Board-Support-Paketen.

Diese Liste wird laufend erweitert, da sowohl Espressif- als auch Drittanbieter- ständig Komponenten

hinzufügen. Als Open-Source-Entwickler können Sie auch Ihre eigenen Komponenten erstellen und sie hier registrieren, damit andere Entwickler sie leicht finden können.

<https://components.espressif.com>



Rust + Embedded

Ein Power-Duo für die Entwicklung

Von Juraj Sadel (Espressif)

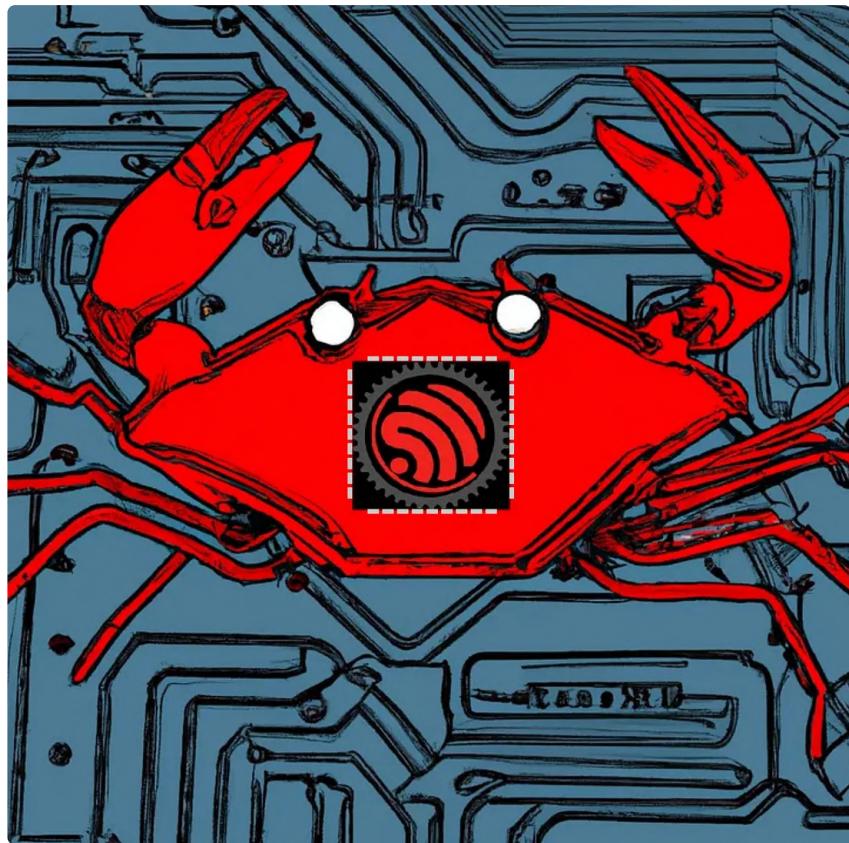
Mit seinem Fokus auf Speicher- und Thread-Sicherheit ist Rust eine beliebte Sprache für die Erstellung zuverlässiger und sicherer Software geworden. Aber ist Rust auch eine intelligente Lösung für eingebettete Anwendungen? Wie Sie erfahren werden, bietet Rust viele Vorteile gegenüber traditionellen Embedded-Entwicklungssprachen wie C und C++, einschließlich Speichersicherheit, Unterstützung von Gleichzeitigkeit und Leistungsfähigkeit.

Rust hat sich zur heißesten neuen Sprache der Welt entwickelt, für die sich jedes Jahr mehr Menschen interessieren. Der Schwerpunkt liegt auf der Speicher- und Threadsicherheit, und es das Hauptanliegen, zuverlässige und sichere Software zu entwickeln. Die Unterstützung von Rust für Nebenläufigkeit und Parallelität ist besonders für die Entwicklung eingebetteter Systeme von Bedeutung, bei denen die effiziente Nutzung von Ressourcen entscheidend ist.

Die Anfänge von Rust

Die ursprüngliche Idee einer Rust-Programmiersprache wurde durch einen Zufall geboren. Im Jahr 2006 kehrte Graydon Hoare in Vancouver in seine Wohnung zurück, aber der Aufzug war aufgrund eines Softwarefehlers wieder einmal außer Betrieb. Herr Hoare wohnte im 21. Stock, und als er die Treppe hinaufstieg, dachte er: „Wir Computerleute können nicht einmal einen Aufzug bauen, der ohne Softwareabsturz funktioniert!“ Dieser Missstand veranlasste Mr. Hoare, eine neue Programmiersprache zu entwickeln. Er hoffte, dass es möglich sein würde, kleinen, schnellen Code ohne Speicherfehler zu schreiben [1]. Wenn Sie an der detaillierten und technischen Geschichte von Rust interessiert sind, besuchen Sie bitte [2] und [3].

Fast 18 Jahre später ist Rust zur heißesten neuen Sprache der Welt geworden, und jedes Jahr interessieren sich mehr Menschen dafür. Im ersten Quartal 2020 gab es etwa 600.000 Rust-Entwickler und im ersten Quartal 2022 war ihre Zahl auf 2,2 Millionen gestiegen. [4] Riesige Tech-Unternehmen wie Mozilla, Dropbox, Cloudflare, Discord, Facebook (Meta), Microsoft und andere verwenden Rust in ihrer Codebasis. In den vergangenen sechs Jahren war Rust die beliebteste Programmiersprache [5] der Welt.



Quelle: Generiert von DALL-E.

Eingebettete Entwicklung

Die Embedded-Entwicklung ist nicht so populär wie die Web- oder Desktop-Entwicklung, und dies sind einige Gründe dafür, warum dies der Fall sein könnte:

- Hardware-Beschränkungen: Eingebettete Systeme verfügen in der Regel über begrenzte Hardweareressourcen, wie Leistungseigenschaften und Speicher. Dies kann die Entwicklung von Software für diese Systeme erschweren.
- Begrenzter Nischen-Markt: Der Markt für eingebettete Systeme ist begrenzter als der für Web- und Desktop-Anwendungen, was es für Entwickler, die sich auf die Programmierung von eingebetteten Systemen spezialisiert haben, finanziell weniger lohnend macht.
- Spezialisiertes Low-Level-Wissen: Spezialwissen über konkrete Hardware und Low-Level-Programmiersprachen ist bei der Embedded-Entwicklung ein Muss.
- Längere Entwicklungszyklen: Die Entwicklung von Software für eingebettete Systeme kann länger dauern als die Entwicklung von Software für Web- oder Desktop-Anwendungen, da der Code für die spezifischen Hardwareanforderungen getestet und optimiert werden muss.
- Low-Level-Programmiersprachen: Diese Sprachen, zum Beispiel Assembler oder C, bieten dem Entwickler kaum Abstraktionsmöglichkeiten und ermöglichen einen direkten Zugriff auf Hardwareressourcen und Speicher, was zu Speicherfehlern führen kann.

Dies sind nur einige Beispiele dafür, warum die Entwicklung eingebetteter Systeme etwas Besonderes ist und für junge Programmierer nicht so interessant und lukrativ wie die Webentwicklung. Wenn man an die gängigsten und modernsten Programmiersprachen wie Python, JavaScript oder C# gewöhnt ist, bei denen man nicht jeden Prozessorzyklus und jedes Kilobyte im Speicher zählen muss, ist die Umstellung auf eine Embedded-Entwicklung recht brutal. Es kann sehr entmutigend sein, in die Embedded-Welt einzusteigen, nicht nur für Anfänger, sondern auch für erfahrene Web-/Desktop-/Mobile-Entwickler. Aus diesem Grund wäre eine

Die Unterstützung von Rust für Nebenläufigkeit und Parallelität ist besonders für die Entwicklung eingebetteter Systeme von Bedeutung, in denen eine effiziente Nutzung von Ressourcen entscheidend ist.

moderne Programmiersprache für die Embedded-Entwicklung sehr interessant.

Warum Rust?

Rust ist eine moderne und relativ junge Sprache mit dem Schwerpunkt auf Speicher- und Thread-Sicherheit, um zuverlässige und sichere Software zu entwickeln. Außerdem ist die Unterstützung von Rust für Nebenläufigkeit (concurrency) und Parallelität (parallelism) besonders wichtig für die Entwicklung eingebetteter Systeme, bei denen die effiziente Nutzung von Ressourcen entscheidend ist. Die wachsende Popularität und das Ökosystem von Rust machen es zu einer attraktiven Option für Entwickler, die eine moderne, effiziente und sichere Sprache suchen. Aus diesen Gründen wird Rust immer beliebter, nicht nur in der Embedded-Entwicklung, sondern insbesondere für Projekte, bei denen Sicherheit und Zuverlässigkeit im Vordergrund stehen.

Vorteile

Schauen wir uns zunächst die bemerkenswerten Vorteile von Rust gegenüber C/C++ an.

- Speichersicherheit: Rust garantiert durch sein Ownership- und Borrowing-System eine starke Speichersicherheit. Es ist sehr hilfreich, um häufige speicherbezogene Fehler wie Null-Pointer-Dereferenzen oder Pufferüberläufe zu verhindern. Mit anderen Worten: Rust garantiert durch sein Ownership- und Borrowing-System Speichersicherheit zur Komplierzeit. Dies ist besonders wichtig bei der Entwicklung von eingebetteten Systemen, wo Speicher-/Ressourcen-

beschränkungen die Erkennung und Behebung solcher Fehler erschweren.

- Nebenläufige Verarbeitung: Rust bietet eine hervorragende Unterstützung für Zero-Cost-Abstraktionen (Hinzufügen höherer Programmierkonzepte ohne Laufzeitkosten) und sichere Gleichzeitigkeit und Multi-Threading, mit einer eingebauten `async/await`-Syntax und einem leistungsfähigen Typensystem, das häufige Gleichzeitigkeitsfehler wie Data-Races verhindert. Dies kann das Schreiben von sicherem und effizientem nebenläufigem Code erleichtern, nicht nur in eingebetteten Systemen.
- Leistung: Rust ist auf hohe Leistung ausgelegt und kann diesbezüglich mit C und C++ mithalten, während es gleichzeitig starke Speichersicherheitsgarantien und Unterstützung für Nebenläufigkeit bietet.
- Lesbarkeit: Die Syntax von Rust ist besser lesbar und weniger fehleranfällig als die von C und C++, mit Funktionen wie Mustervergleich, Typinferenz und funktionalen Programmierstrukturen. Dies kann das Schreiben und die Wartung von Code erleichtern, insbesondere bei größeren und komplexeren Projekten.
- Wachsendes Ökosystem: Rust verfügt über ein wachsendes Ökosystem von Bibliotheken (Crates), Tools und Ressourcen (nicht nur) für die Embedded-Entwicklung, was den Einstieg in Rust und die Suche nach der notwendigen Unterstützung und den Ressourcen für ein bestimmtes Projekt erleichtert.
- Paketmanager und Build-System: Die Rust-Distribution enthält ein offizielles Tool namens Cargo, mit dem der Build-, Test- und Veröffentlichungsprozess zusammen mit der Erstellung eines neuen Projekts und der Verwaltung seiner Abhängigkeiten automatisiert werden kann.

Nachteile

Andererseits ist Rust keine perfekte Sprache und hat auch einige Nachteile gegenüber anderen Programmiersprachen (nicht nur, aber auch gegenüber C und C++).

- Lernkurve: Rust hat eine steilere Lernkurve als viele andere Programmiersprachen, einschließlich C.

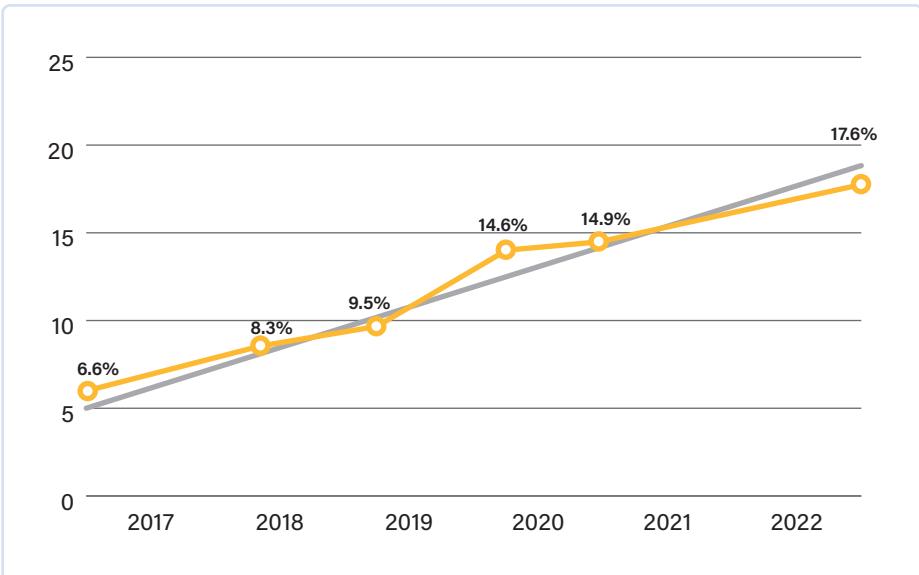


Bild 1. Der wachsende Prozentsatz der Rust-Entwickler in spe. (Quelle: Yalantis [4])

Seine einzigartigen Eigenschaften wie das bereits erwähnte Ownership und Borrowing können einige Zeit in Anspruch nehmen, um sie zu verstehen und sich daran zu gewöhnen, und sind daher eine größere Herausforderung für den Einstieg in Rust.

- Kompilierzeit: Rusts fortschrittliches Typsystem und der Borrow-Checker können zu längeren Kompilierzeiten im Vergleich zu anderen Sprachen führen, insbesondere bei großen Projekten.
- Tooling: Obwohl das Ökosystem von Rust schnell wächst, hat es möglicherweise noch nicht das gleiche Maß an Tooling-Unterstützung wie etabliertere Programmiersprachen. C und C++ zum Beispiel gibt es schon seit Jahrzehnten und sie haben eine riesige Code-Basis. Dies kann es für Rust schwieriger machen, die richtigen Tools für ein bestimmtes Projekt zu finden und einzusetzen.
- Fehlende Low-Level-Kontrolle: Die Sicherheitsfunktionen von Rust beschränken die Low-Level-Kontrolle auf C und C++. Es ist zwar prinzipiell möglich, aber deutlich schwieriger, bestimmte Low-Level-Optimierungen durchzuführen oder direkt mit der Hardware zu interagieren.
- Größe der Community: Rust ist im Vergleich zu etablierteren Sprachen wie C und C++ noch eine relativ neue Programmiersprache, was bedeutet, dass es eine kleinere Community von Entwicklern und Mitwirkenden sowie weniger Ressourcen, Bibliotheken und Tools gibt.

Insgesamt betrachtet bietet Rust viele Vorteile gegenüber traditionellen eingebetteten Entwicklungssprachen wie C und C++, einschließlich Speichersicherheit, Unterstützung von Gleichzeitigkeit, Leistung, Lesbarkeit des Codes und einem wachsenden Ökosystem. Infolgedessen wird Rust immer beliebter für die Embedded-Entwicklung, insbesondere für Projekte, bei denen Sicherheit und Zuverlässigkeit im Vordergrund stehen. Die Nachteile von Rust im Vergleich zu C und C++ hängen in der Regel mit der relativen Neuheit von Rust als Sprache und seinen einzigartigen Eigenschaften zusammen. Viele Entwickler sind jedoch der Meinung, dass die Vorteile von Rust für bestimmte Projekte eine überzeugende Wahl darstellen.

Wie kann Rust ausgeführt werden?

Es gibt mehrere Möglichkeiten, die Rust-basierte Firmware auszuführen, abhängig von der Umgebung und den Anforderungen der jeweiligen Anwendung. Rust-basierte Firmware kann in der Regel in einem von zwei Modi verwendet werden: *hosted-environment* oder *bare-metal*. Schauen wir uns an, was das ist.

Was ist eine gehostete Umgebung?

In Rust ähnelt *hosted-environment*, die gehostete Umgebung einer normalen PC-Umgebung [6], was bedeutet, dass Ihnen ein Betriebssystem zur Verfügung gestellt wird. Mit dem Betriebssystem ist es möglich, die Rust-Standardbibliothek (*std*) zu erstellen [7]. Die *std* bezieht sich auf die Sammlung von Modulen und Typen, die in jeder Rust-Installation enthalten sind. Die *std* stellt eine Reihe

von Funktionalitäten für die Erstellung von Rust-Programmen zur Verfügung, darunter Datenstrukturen, Netzwerke, Mutexe und andere Synchronisationsprimitive, Eingabe/Ausgabe und vieles mehr.

Mit dem Ansatz des *hosted-environment* können Sie die Funktionalität des C-basierten Entwicklungsframeworks ESP-IDF [8] nutzen, da es eine *newlib*-Umgebung [9] bietet, die mächtig genug ist, um die Rust-Standardbibliothek darauf aufzubauen. Mit anderen Worten, mit dem Ansatz *hosted-environment* (manchmal auch nur *std* genannt) verwenden wir ESP-IDF als Betriebssystem und bauen die Rust-Anwendung darauf auf. Auf diese Weise können wir alle oben genannten Funktionen der Standardbibliothek nutzen und auch bereits C-Funktionen der ESP-IDF-API implementieren.

In **Listing 1** können Sie ein Blinky-Beispiel [10] sehen, das auf ESP-IDF (FreeRTOS) läuft. Weitere Beispiele finden Sie in *esp-idf-hal* [11].

Wann Sie eine gehostete Umgebung verwenden sollten

- Reiche Funktionalität: Wenn Ihr eingebettetes System viele Funktionen benötigt, zum Beispiel Unterstützung für Netzwerkprotokolle, Datei-I/O oder komplexe Datenstrukturen, werden Sie wahrscheinlich den *hosted-environment*-Ansatz verwenden, da *std*-Bibliotheken eine breite Palette von Funktionen bereitstellen, mit denen sich komplexe Anwendungen relativ schnell und effizient erstellen lassen.
- Portabilität: Das *std*-Crate bietet einen standardisierten Satz von APIs, die auf verschiedenen Plattformen und Architekturen verwendet werden können, was das Schreiben von portierbarem und wiederverwendbarem Code erleichtert.
- Schnelle Entwicklung: Das *std*-Crate bietet eine Vielzahl von Funktionen zur schnellen und effizienten Entwicklung von Anwendungen, ohne dass man sich um Low-Level-Details kümmern muss.

Was ist Bare-Metal?

Bare-Metal bedeutet, dass es kein Betriebssystem gibt, mit dem wir arbeiten. Wenn ein Rust-Programm mit dem *no_std*-Attribut

but kompiliert wird, bedeutet dies, dass das Programm keinen Zugriff auf bestimmte Funktionen hat. Das heißt nicht unbedingt, dass man mit `no_std` keine Netzwerke oder komplexe Datenstrukturen verwenden kann. Man kann mit `no_std` alles tun, was man auch mit `std` tun kann, aber es ist komplexer und anspruchsvoller. `no_std`-Programme verlassen sich auf eine Reihe von Eigenschaften der Kernsprache, die in allen Rust-Umgebungen verfügbar sind, zum Beispiel Datentypen, Kontrollstrukturen oder Low-Level-Speicherverwaltung. Dieser Ansatz ist nützlich für die eingebettete Programmierung, bei der die Speichernutzung eingeschränkt und eine Low-Level-Kontrolle über die Hardware erforderlich ist.

In **Listing 2** sehen Sie ein Blinky-Beispiel [12], das auf Bare-Metal (ohne Betriebssystem) läuft. Weitere Beispiele finden Sie in `esp-hal` [13].

Wann sollten Sie Bare-Metal verwenden?

- Geringer Speicherbedarf: Wenn Ihr eingebettetes System über begrenzte Ressourcen verfügt und einen kleinen Speicher hat, werden Sie wahrscheinlich Bare-Metal verwenden, da `std` die letztendliche Größe der Binary und die Kompilierungszeit erheblich vergrößert.
- Direkte Hardware-Steuerung: Wenn Ihr eingebettetes System eine direktere Kontrolle über die Hardware erfordert, zum Beispiel Low-Level-Gerätetreiber oder Zugriff auf spezielle Hardwarefunktionen, werden Sie wahrscheinlich Bare-Metal verwenden, da `std` Abstraktionen hinzufügt, die die direkte Interaktion mit der Hardware erschweren.
- Echtzeit-Beschränkungen oder zeitkritische Anwendungen: Wenn Ihr eingebettetes System Echtzeit oder Reaktionszeiten mit geringer Latenz erfordert, ist Bare-Metal vorzuziehen, da `std` unvorhersehbare Verzögerungen und Overhead verursachen kann, was die Echtzeitleistung beeinträchtigt.
- Benutzerdefinierte Anforderungen: Bare-Metal ermöglicht mehr Anpassungen und eine „feinkörnige“ Kontrolle über das Verhalten einer Anwendung, was in speziellen oder nicht standardisierten Umgebungen nützlich sein kann.

Sollten Sie von C zu Rust wechseln?

Wenn Sie ein neues Projekt oder eine Aufgabe beginnen, bei der Speichersicherheit oder Gleichzeitigkeit erforderlich sind, kann sich der Wechsel von C zu Rust lohnen. Wenn Ihr Projekt jedoch bereits gut etabliert ist und in C funktioniert, überwiegen die Vorteile eines Wechsels zu Rust möglicherweise nicht die Kosten für das Portieren und erneute Testen Ihrer gesamten Codebasis. In diesem Fall können Sie in Erwägung ziehen, die aktuelle C-Codebasis beizubehalten und mit dem Schreiben und Hinzufügen neuer Features, Module und Funktionen in Rust zu beginnen - es ist relativ einfach, C-Funktionen aus dem Rust-Code aufzurufen. Es ist auch möglich, ESP-IDF-Komponenten in Rust zu schreiben [14]. Letztendlich sollte die Entscheidung, von C nach Rust zu wechseln, auf einer sorgfältigen Bewertung Ihrer spezifischen Bedürfnisse und der damit verbundenen Kompromisse beruhen. 

RG – 230569-02

Haben Sie Fragen oder Kommentare?

Wenn Sie technische Fragen oder Kommentare zu diesem Artikel haben, wenden Sie sich bitte an den Autor unter juraj.sadel@espresif.com oder an die Elektor-Redaktion unter redaktion@elektor.de.

Über den Autor

Juraj Sadel ist ein Entwickler von Embedded Software, der sich für Rust begeistert und sich der Verbesserung von Embedded Systemen widmet. Er ist auch ein aktives Mitglied des Rust-Teams, das seine Expertise in die Community einbringt, und begeistert von der Spitzentechnologie von Espresif.



Listing 1: Blinky-Beispiel, mit gehosteter Umgebung und 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-Beispiel. 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();

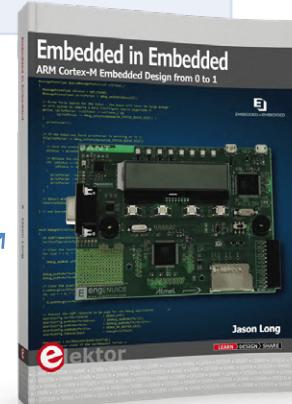
    // Initialize the Delay peripheral, and use it to toggle the LED state in a
    // loop.
    let mut delay = Delay::new(&clocks);
```

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



Passende Produkte

- **J. Long, Embedded in Embedded (Elektor 2018)**
Buch, Paperback, englisch: www.elektor.de/18876
E-Book, PDF, englisch: www.elektor.de/18877
- **A. He und L. He, Embedded Operating System (Elektor 2020)**
Buch, Paperback, englisch: www.elektor.de/19228
E-Book, PDF, englisch: www.elektor.de/19214



WEBLINKS

- [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 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.Bibliothek: <https://sourceware.org/newlib>
- [10] Blinky-Beispiel läuft auf 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-Beispiel läuft auf 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-Befehle in Rust: <https://github.com/espressif/rust-esp32-example>



ESP-Matter

Espressifs SDK für Matter



Espressif bietet eine einfach zu bedienende API auf dem Open-Source-SDK namens „connectedhomeip“, mit der Sie problemlos Matter-kompatible Projekte erstellen können. Dieses SDK unterstützt alle Espressif-SoCs wie ESP32, ESP32-C3, ESP32-C2, ESP32-S3, ESP32-H2 und ESP32-C6. Es unterstützt sowohl das Matter-Protokoll über WLAN als auch Thread. Zusätzlich zu den Standard-Matter-Gerätebeispielen, die verschiedene Gerätetypen unterstützen, bietet es auch Implementierungen von Matter ZigBee Bridge, Matter BLE Mesh Bridge und Matter ESP-Now Bridge.

<https://github.com/espressif/esp-matter>





Wer sind die Rust-begeisterten Embedded-Entwickler?

Wie Espressif Embedded Rust für den ESP32 kultiviert

Intro und Fragen von Stuart Cording (Elektor)

Sie ist erst etwas mehr als zehn Jahre alt, aber die Programmiersprache Rust hat bereits Platz 17 im TIOBE Programming Community Index [1] erreicht. Sie hat sich in dieser Zeit um 194 Plätze verbessert und steht nun in einer Reihe mit R, Ruby, Delphi, Scratch und MATLAB. Und nicht nur das: Linus Torvalds, der Vater von Linux, hat Vorschläge angenommen, Rust-Code im Kernel zu verwenden – etwas, das C++ seit Jahren vergeblich versucht hat. Rust hat auch eine wachsende Fangemeinde unter den Entwicklern von eingebetteten Systemen, und Espressif hat beschlossen, auf diesen Zug zu springen.

C ist seit Jahrzehnten die Standardsprache für eingebettete Systeme und überlässt Assembler denjenigen, die von Hand Optimierungen vornehmen oder Echtzeit-Kernel entwickeln. Sie wurde in den 1970er Jahren von Dennis Ritchie in den Bell Labs entwickelt und ist eng mit der Erschaffung des Betriebssystems Unix verbunden. Unix wurde in Assembler für den PDP-7 [2] geschrieben, musste aber für die Portierung auf den PDP-11 [3] neu verfasst werden (PDPs waren kleinere Allzweckcomputer, die in den 1960er Jahren als Alternative zu Großrechnern verkauft wurden). C ermöglichte die Entwicklung von Code, der wie Unix prozessorunabhängig und über viele verschiedene Architekturen hinweg portierbar war.

Eingebettete Systeme wurden anfangs in Assembler programmiert, aber als der Code komplexer wurde und die Entwickler nach besseren Möglichkeiten zur Wiederverwendung suchten, wurden auch sie von C angezogen. Zu den zusätzlichen Merkmalen, die ihnen das Leben erleichterten, gehörten die Unterstützung von Low-Level-Bitmanipulationen, die mühelose Definition von Variablen, die Leichtigkeit, mit der Algorithmen geschrieben werden konnten, und die

Einfachheit der SpeicherManipulation.

Dieser letzte Punkt führt jedoch dazu, dass die meisten Anwendungen nicht mehr funktionieren. Pointer erlaubten es dem Programmierer zwar, jederzeit auf (fast) jeden Speicherplatz zuzugreifen, was in einem eingebetteten System sehr leistungsfördernd und effizient ist. Sie können aber auch ziemlich gefährlich sein. Pointer können auf Speicher verweisen, der längst wieder freigegeben wurde, oder sie können manipuliert werden, um eine Funktion aufzurufen, die eine Sicherheitslücke verursacht. Tatsächlich führt Microsoft etwa 70 % der Probleme mit C/C++-Software auf Speicherkorruption zurück [4].

Rust geht dieses Problem an, indem es Speichersicherheit erzwingt. Darüber hinaus werden im Gegensatz zu C/C++-Code viele Programmierfehler bereits beim Kompilieren und nicht erst zur Laufzeit erkannt. Und dank der Ähnlichkeiten in Syntax und Leistung werden sich C- und C++-Entwickler sowohl auf ihren PCs als auch auf eingebetteten Systemen schnell zu Hause fühlen.

Aber wie ernst nehmen die Hersteller von Mikrocontrollern Rust als Sprache für eingebettete Systeme? Um mehr darüber zu erfahren, sprach Elektor mit Scott Mabin, einem jungen Entwickler, der sich als einer der ersten auf dem Gebiet dazu entschlossen hat, immer tiefer in die Welt von Rust einzutauchen. Als begeisterter Nutzer von ESP32s hat er einen großen Beitrag zu Embedded Rust geleistet, was ihm eine Anstellung bei Espressif einbrachte.

Stuart Cording: Eingebettete Systeme werden traditionell in C programmiert. Sie haben sich jedoch entschieden, das zu ignorieren und direkt in Rust einzusteigen. Warum?

Scott Mabin: Ich hatte Rust in meinem Abschlussprojekt verwendet. Ich habe eine Smartwatch gebaut – nach heutigen Maßstäben nicht wirklich „smart“, aber sie konnte mehr als nur die Tageszeit anzeigen. Ein Teil dieses Projekts untersuchte, ob Rust die Firmware-Entwicklung mit C ersetzen könnte und inwieweit Kompromisse gemacht werden müssen. Und das

war der Moment, in dem ich mich in Rust verliebte. Ich saß da und dachte: „Warum benutzt nicht jeder Rust?“ Es bot so viele Dinge an, die einfach funktionierten. Natürlich verstand ich, dass einige Projekte Altlasten haben und dass man nicht ohne Grund eine neue Sprache lernen und einsetzen möchte. Trotzdem erschien es mir verrückt, dass sich nicht mehr Leute Rust zuwenden.

Stuart Cording: Rust wird auf STM32 und einigen Nordic-Geräten bereits gut unterstützt. Warum haben Sie sich damals für Espressif entschieden?

Scott Mabin: Zu Hause hatte ich all diese ESP32-Brettschweine herumliegen und fragte mich: „Sicherlich kann ich diese Dinger auch in Rust programmieren?“ Dann führte eines zum anderen und auf einmal war ich mittendrin. Das größte Problem war, dass der Xtensa-Kern, der Geräte wie den ESP8266 antreibt, nur vom GCC-Compiler unterstützt wurde, nicht aber von LLVM. Ich schaute mir *mrustc* [5] an, ein Tool, das Rust in C konvertiert und dann kompiliert werden kann, aber dieser Querkompilierprozess konnte nicht überzeugen. Glücklicherweise veröffentlichte Espressif eine Fork für die LLVM-Toolchain, um Xtensa zu unterstützen, und obwohl es anfangs schwer zu benutzen war, bedeutete es, dass man natives Rust für ESP32-Geräte kompilieren konnte. Damit habe ich meinen ersten Code für eine blinkende LED in Rust geschrieben. Naja, ich sage „in Rust“, aber er schrieb im Grunde nur in Register und war in seiner Struktur nicht sehr Rust-artig. Dennoch war es ein früher Erfolg, den ich in meinem ersten Blogbeitrag [6] dokumentierte, in dem ich meine Reise mit Rust auf ESP32 festhielt.

Stuart Cording: Sie haben also mit Rust auf ESP32 herumgebastelt. Wie hat sich die Beziehung zu Espressif entwickelt?

Scott Mabin: Ich habe in meiner Freizeit die *esp-rs*-Gruppe auf GitHub gegründet, um Rust-Projekte für ESP32 zu sammeln. Zusammen mit anderen Community-Mitgliedern haben wir Unterstützung für Peripheriegeräte wie SPI implementiert. Leider hatte WiFi sich uns noch immer entzogen. Dann, nach einem Jahr Community-Arbeit und einem Blog über unsere Fortschritte, fragte Espressif an, ob sie mich einstellen könnten, um die Rust-Unterstützung für ihr Ökosystem sicherzustellen. Das Team von Espressif war uns gegenüber immer sehr hilfsbereit gewesen und hatte auf Support-Anfragen in ihren Foren oder auf Reddit geantwortet. Aber als sie sich an mich wandten, wurde schnell klar, dass sie sich schon eine Weile für Rust interessierten.

Stuart Cording: Gibt es bestimmte Anwendungsbezüge, die das Interesse an Rust wecken, oder handelt es sich um einen Blick in die Kristallkugel seitens

Espressif? Und wie sieht es mit dem Engagement aus?

Scott Mabin: Das Interesse der Kunden ist definitiv vorhanden. Die Kunden verwenden Rust vor allem in IoT-Projekten oder für Anwendungen mit Internetanbindung. Die meisten Projekte sind ziemlich einfach, zum Beispiel Datenlogger. Dann gibt es aber auch die verrückten Projekte, bei denen jemand GPS und Wi-Fi zu einer tragbaren Anwendung zusammengefügt hat, die einem die Wegbeschreibung direkt ins Ohr spricht! Espressif betreibt auch ein gewisses Maß an Zukunftsicherung und bereitet sich darauf vor, dass Rust im Leben von Entwicklern eingebetteter Systeme eine wichtigere Rolle spielen wird. Etwa zehn oder elf von uns arbeiten im Rust-Team von Espressif mit, davon ungefähr die Hälfte Vollzeitkräfte. Darüber hinaus haben eingebettete Rust-Anwendungen eine großartige Community, die ebenfalls ihren Beitrag leistet.

Stuart Cording: Sie haben einiges in Embedded C entwickelt verwendet jetzt täglich Rust auf Mikrocontrollern. Was müssen traditionelle C-Programmierer beachten, wenn sie zu Rust wechseln?

Scott Mabin: Wenn man aus der reinen C-Entwicklung kommt und keine Erfahrung mit C++ hat, ist es ziemlich schwierig. Die radikalste Veränderung ist, dass Rust ziemlich viel mit Typen arbeitet. Dann ist da noch die Ressourcenseite. Kleine Mikrocontroller mit ein paar Kilobytes SRAM und Flash werden aufgrund der Speicherbeschränkungen Schwierigkeiten haben. In solchen Fällen wird man am Ende ein C-ähnliches Rust schreiben, um die Speicherbelegung gering zu halten, und damit den Zweck dieser neuen Sprache zunichten machen. Die Vorteile von Rust in Bezug auf die Speichersicherheit bleiben erhalten, so dass Rust immer noch besser ist als C. Bei einem direkten Vergleich schneiden C- und Rust-Anwendungen in etwa gleich gut ab. In einigen Fällen ist Rust besser, aber die Binärdateien sind in der Regel größer.

Stuart Cording: Während die Hersteller von Mikrocontrollern großartige Unterstützung mit Peripherie-Treibern anbieten, ist die Versionskontrolle des gesamten Codes für den Entwickler oft chaotisch. Sind Rust-Crates ein Grund, die Programmiersprache zu wechseln?

Scott Mabin: Ja, ich denke, Crates sind ein großer Vorteil – ein großer Bonus, der über die Sprache selbst hinausgeht. Crates bieten ein Paketierungs-Ökosystem oder zumindest eine Methode, um Abhängigkeiten einzubinden, ohne sie manuell in Ihrem Build-System zu definieren. Die Crate *embedded_hal* [7] bietet eine trait-basierte Schnittstelle für gängige Peripheriegeräte wie I²C. Wenn Sie dann einen I²C-Temperatursensorortreiber nehmen, funktioniert er einfach darüber. Wenn man später den Mikrocontroller wechselt oder einen neuen Temperatursensor verwendet, funktioniert er immer noch, was sehr schön ist. Aber das

```

File Edit Selection View Go Run Terminal Help ↵ esp-wifi
examples-esp32c3 > examples > embassy_dhcp.rs > connection
88
89 #![embassy_executor::task]
90 v async fn connection() mut controller: WifiController<'static> {
91     println!("Start connection task");
92     println!("Device capabilities: {:?}", controller.get_capabilities());
93     loop {
94         match esp_wifi::wifi::get_wifi_state() {
95             WifiState::StaConnected => {
96                 // wait until we're no longer connected
97                 controller.wait_for_event(WiFiEvent::StaDisconnected).await;
98                 Timer::after(Duration::from_millis(5000)).await;
99             }
100        } => {}
101    }
102    if !matches!(controller.is_started(), OK(true)) {
103        let client_config: Configuration = Configuration::Client(ClientConfiguration {
104            ssid: SSID.into(),
105            password: PASSWORD.into(),
106            ..Default::default()
107        });
108        controller.set_configuration(client_config.unwrap());
109        println!("Starting wifi");
110        controller.start().await.unwrap();
111        println!("Wifi started!");
112    }
113    println!("About to connect...");
114
115    match controller.connect().await {
116        Ok(_) => println!("Wifi connected!"),
117        Err(e) => {
118            println!("Failed to connect to wifi: {e:?}");
119            Timer::after(Duration::from_millis(5000)).await;
120        }
121    }
122 }
123 } fn connection
124
125 #![embassy_executor::task]
126 v async fn net_task(stack: &mut Stack<WifiDevice<'static>>) {
127     stack.run().await
128 }

```

Bild 1. Ein Beispiel für eine WLAN-Anwendung, die in Rust für den ESP32 geschrieben wurde.

kratzt nur an der Oberfläche der Sache. Crates wie heapless [8] ermöglichen die Erstellung von statisch zugewiesenen Datenstrukturen wie Queues, Vektoren, Hash-Tabellen, Hash-Sets und Strings. In jedem zweiten C-Projekt, an dem ich gearbeitet habe, sitzt jemand da und schreibt eine Queue und verbringt viele Stunden damit, eine Standardprogrammierdatenstruktur zu perfektionieren. Diese Zeit kann jetzt für die Anwendungsentwicklung genutzt werden.

Stuart Cording: Ähnlich wie C bietet auch Rust eine Standardbibliothek. Embedded-Entwickler verabscheuen Standardbibliotheken, können sie also auch auf die von Rust verzichten?

Bild 2. Scott Mabin empfiehlt VS Code für die Rust-Entwicklung. Hier ist der klassische LED-Blinky-Code zu sehen.

Scott Mabin: In der Vergangenheit haben wir uns schwer getan, die Anwendungsfälle zu beschreiben, in denen ein professioneller Embedded-Programmierer eine Standardbibliothek verwenden sollte [9]. Sie werfen einen Blick darauf und sagen: „Das ist viel zu viel Aufwand“, und ich würde dem vollkommen zustimmen. Die Rust-Standardbibliothek ist

jedoch in vielerlei Hinsicht sehr nützlich. Erstens, wenn man zwar Rust, aber nicht die Embedded-Entwicklung kennt, fühlt sie sich vertraut an. Sie erhält alle Threads, Networking und andere Dinge, die Sie gewohnt sind. Zweitens, wenn Sie ESP32s mit dem ESP-IDF [10] (Espressif-Entwicklungsframework für das Internet der Dinge) programmieren, können Sie Rust-Standardbibliothek-Projekte erstellen und loslegen. Wenn man sie mit der Python-Standardbibliothek vergleicht, ist sie eigentlich recht schlank, und obwohl es einen Overhead gibt, ist er nicht so schlimm, wie es klingt. Sie setzt ein paar mehr Annahmen voraus als die Kernbibliothek (die Sie mindestens benötigen), zum Beispiel die Verfügbarkeit von Threads und Networking.

Viele Leute entwickeln eingebettetes Rust ohne die Standardbibliothek (`no_std` [11]), und wenn Sie wissen, was Sie tun, sollten Sie auf jeden Fall den `no_std`-Ansatz wählen, solange er für Ihr Projekt sinnvoll ist. Allerdings müssen Sie die Komponenten, die Sie verwenden wollen, auswählen und zusammensetzen. Die WLAN-Crate enthält zum Beispiel ein Muster für die Kommunikation mit einem HTTP-Server oder die Einrichtung eines Zugangspunkts und die Ausführung eines Servers (Bild 1). Sie können in `no_std` alles tun, was Sie mit `std` tun können - es ist nur mehr Arbeit. Bei `std` handelt es sich um eine Umgebung, bei der alles nötige Zubehör mitgeliefert wird.

Stuart Cording: Ist ein Teil des Problems, dass wir einfach akzeptieren müssen, dass wir Mikrocontroller mit großen Speichern brauchen?

Scott Mabin: Ich denke, egal wie groß Mikrocontroller werden, es wird immer jemanden geben, der ein Gerät in Millionenauflage bauen will und eine möglichst kleine und billige Lösung braucht. Es wird immer einen Anwendungsfall für Assembler und C – und vielleicht sogar Rust – in einem bestimmten Kontext für diese kleine, kompakte Anwendung geben. Ich glaube aber, dass es einen allgemeinen Trend zu größeren Mikrocontrollern gibt und dass die Erfahrung des Entwicklers immer wichtiger wird als die Hardwarekosten. Das ist zwar nur meine Beobachtung in meiner kurzen Zeit in der Branche, aber so sieht es für mich aus.

Stuart Cording: Wir befinden uns noch immer in der Jungendzeit von Rust. Wird Espressif Schulungen anbieten, um Entwickler auf den neuesten Stand zu bringen?

Scott Mabin: Wir haben uns mit Ferrous Systems [12], einem Rust-Beratungsunternehmen, zusammengetan, um einen Schulungskurs anzubieten, den wir quelloffen gemacht haben. Es gibt Schulungsmaterialien für den Standardbibliotheksansatz, und wir sind auch fast fertig mit den Materialien für den Ansatz ohne Standardbibliothek.

```

File Edit Selection View Go Run Terminal Help ↵ esp-hal
examples-esp32c3-hal > examples > blinky.rs > main
1 //!!! Blinks an LED
2 //!
3 //!!! This assumes that a LED is connected to the pin assigned to 'led'. (GPIO5)
4
5 #[no_std]
6 #[no_main]
7
8 v use esp32c3_hal::{clock::ClockControl, gpio::IO, peripherals::Peripherals, prelude::*, Delay};
9 use esp_backtrace as _;
10
11 #[entry]
12 v fn main() -> ! {
13     let peripherals = Peripherals::take();
14     let system = peripherals.SYSTEM.split();
15     let clocks = ClockControl::boot_defaults(system.clock_control).freeze();
16
17     // Set GPIO5 as an output, and set its state high initially.
18     let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
19     let mut led = io.pins.gpio5.into_push_pull_output();
20
21     led.set_high().unwrap();
22
23     // Initialize the Delay peripheral, and use it to toggle the LED state in a
24     // loop.
25     let mut delay = Delay::new(clocks);
26
27     loop {
28         led.toggle().unwrap();
29         delay.delay_ms(500u32);
30     }
31 }

```

Stuart Cording: Und was ist mit Entwicklungsumgebungen und Debugging-Tools?

Scott Mabin: Ich bewundere die Ästhetik von Umgebungen wie Vim, Neovim oder Emacs-Setups, aber da ich keine Erfahrung mit diesen Tools habe, sind sie mir einfach im Weg. Daher verwende ich VS Code (**Bild 2**) – für mich ist es eine gute Lösung. Bei den neueren ESPs (C3 und höher) haben wir ein Modul namens USB Serial JTAG. Es ist ein extrem kleines eingebautes Peripheriemodul, das eine serielle Schnittstelle und eine JTAG-Funktion bietet (**Bild 3**). Man schließt es einfach an den USB-Port eines PCs an, und es wird automatisch von OpenOCD erkannt oder von probe-rs, einem Rust-Debugging-Toolkit mit einem ähnlichen Zweck wie OpenOCD.

Stuart Cording: Eine neue Sprache auf einem Prozessor zu unterstützen ist eine Mammutaufgabe. Wo stehen Sie derzeit, und was muss noch erledigt werden?

Scott Mabin: Wir haben so ziemlich alles für die Standardbibliothek vorbereitet, aber wir haben noch nicht alles für ESP-IDF abgedeckt. ESP-IDF ist sehr umfangreich, existiert schon seit Jahren und ist in C geschrieben, also benutzen wir bindgen, um Bindings mit Rust zu erstellen. Im Grunde genommen müssen wir uns die Schnittstellen ansehen, die noch nicht implementiert sind, und eine Rust-API für sie erstellen. Wir müssen von Fall zu Fall entscheiden, was wir priorisieren. Bei no_std geht es um die Programmierung in reinem Rust, also müssen wir Code für die gesamte vorhandene Hardware schreiben. Wir haben WLAN-, Bluetooth- und Thread-Unterstützung auf allen Chips. Also, auf einer Skala von 0 bis 100 würde ich sagen, dass wir etwa 65...70 Prozent des Weges geschafft haben. Leider ist es gewissermaßen ein bewegliches Ziel, da wir ständig neue Geräte unterstützen müssen, sobald sie auf den Markt kommen. Wir haben darüber gesprochen, einige Elemente von ESP-IDF in Rust zu erstellen, wo es Sinn ergibt. Zum Beispiel ist Rust perfekt für die Übergabe von Byte-Streams geeignet. Wenn also eine neue Komponente benötigt wird, werden wir sie vielleicht in Rust erstellen und eine C-API anbieten. Wir werden sehen müssen, ob die Vorteile den Aufwand überwiegen.

Stuart Cording: Abschließend: Wo finden Sie als Embedded-Entwickler weitere Informationen über Rust?

Scott Mabin: Ich treibe mich hauptsächlich im ESP-Rust-Matrix-Kanal [13] und verschiedenen anderen Kanälen zum Thema Embedded Rust herum. Ansonsten Google, manchmal Reddit; das Lesen von gutem Rust-Code kann mir helfen, Dinge zu verstehen und zu lernen, die ich noch nicht kannte. 



Bild 3. Der ESP32-C3 enthält ein USB-basiertes Debug-Modul mit einer seriellen Schnittstelle, so dass keine externe Testhardware benötigt wird.

Haben Sie Fragen oder Kommentare?

Wenn Sie technische Fragen oder Kommentare zu diesem Artikel haben, wenden Sie sich bitte an die Elektor-Redaktion unter redaktion@elektor.de.



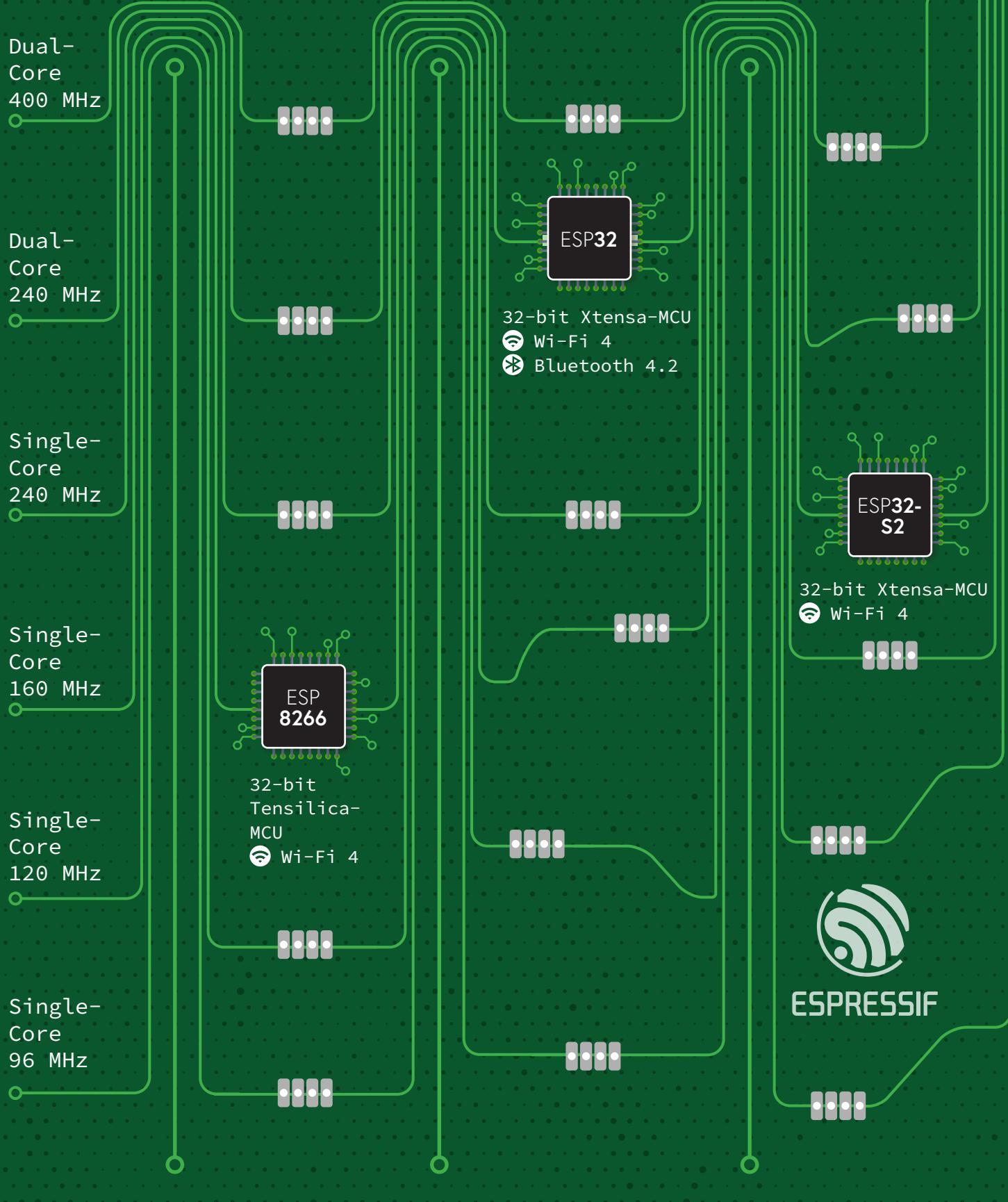
Über den Autor

Scott Mabin ist ein Ingenieur für eingebettete Software. Nachdem er sich bereits an der Universität mit den Möglichkeiten von Embedded Rust beschäftigt hatte, beschloss er, in seiner Freizeit mit dem ESP32 zu experimentieren. Diese Bemühungen führten dazu, dass Espressif ihn bat, ihrem Team beizutreten, um sich auf die Verbesserung der Unterstützung für Rust auf ihren Wireless MCUs und IoT-Lösungen zu konzentrieren.

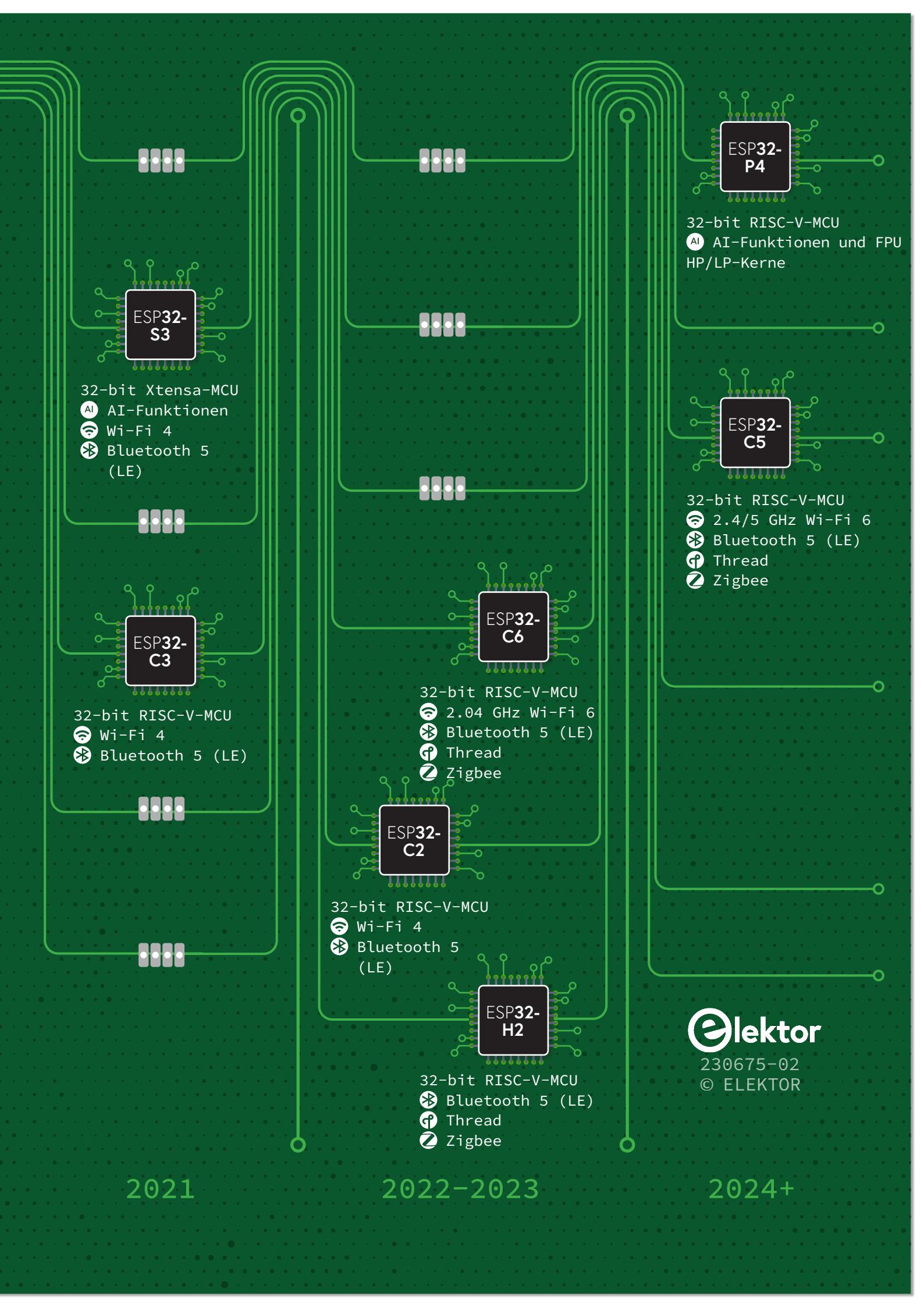
WEBLINKS

- [1] TIOBE-Index der Programmier-Community: <https://tinyurl.com/tioberrust>
- [2] PDP-7 (Wikipedia): <https://de.wikipedia.org/wiki/PDP-7>
- [3] PDP-11 (Wikipedia): <https://de.wikipedia.org/wiki/PDP-11>
- [4] MSRC-Team, „A proactive approach to more secure code“, Microsoft, 2019: <https://tinyurl.com/msrcsecurecode>
- [5] mrustc-Projekt: <https://tinyurl.com/mrustcproject>
- [6] S. Mabin, „Rust on the ESP32“, September 2019: <https://tinyurl.com/rustesp32>
- [7] Dokumentation der Crate embedded_hal: <https://tinyurl.com/embeddedhalcrate>
- [8] Dokumentation der Crate heapless: <https://tinyurl.com/heaplesscrate>
- [9] The Rust on ESP Book, „Using the Standard Library (std)“: <https://tinyurl.com/rustbookstd>
- [10] ESP-IDF-Programmierleitfaden, „Get started“: <https://tinyurl.com/espidogetstarted>
- [11] The Rust on ESP Book, „Using the Standard Library (std)“: <https://tinyurl.com/rustbooknostd>
- [12] Training, „Embedded Rust on Espressif“: <https://tinyurl.com/rustonespressif>
- [13] Rust-Usergruppe auf Matrix: <https://tinyurl.com/rustmatrixug>

Zeitleiste der SoC-Reihe von Espressif



↓ elektormagazine.de/posters





Aufbau einer SPS mit Espressif-Lösungen

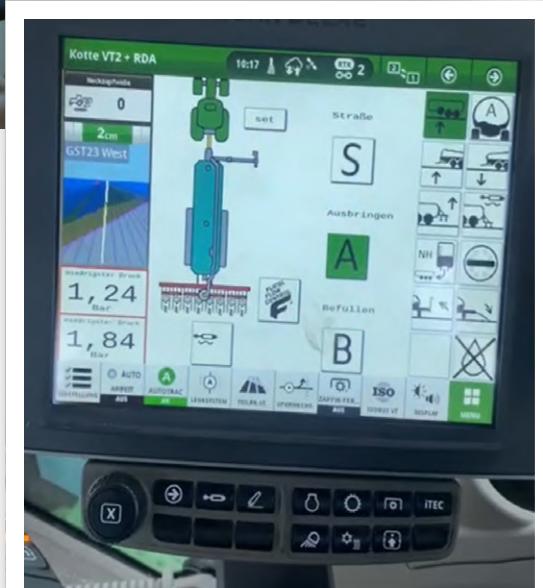
Mit den Fähigkeiten und Funktionen des ISOBUS-Protokolls

Von Franz Höpfinger, HR Agrartechnik

Die HR Agrartechnik GmbH benötigte eine kostengünstige und dennoch leistungsfähige SPS-Steuerung mit ISOBUS-Fähigkeit (IOS 11783). Aus der Kombination eines ESP32 von Espressif mit dem ESP-IDF und dem Framework Eclipse 4diac entstand das logiBUS-Projekt.

Die Herausforderung

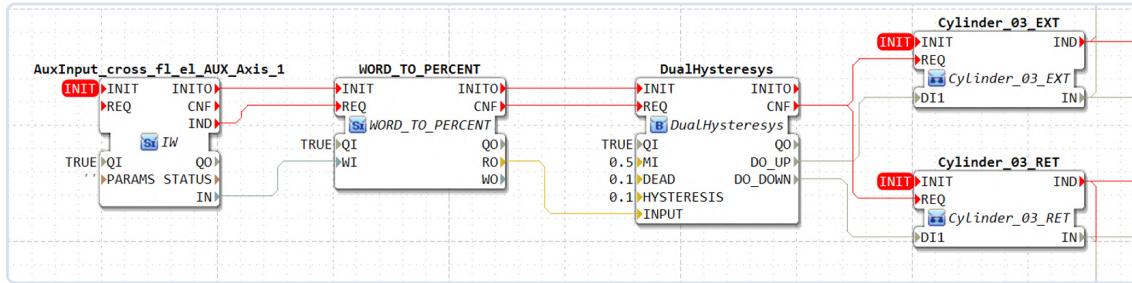
Landmaschinen benötigen heutzutage flexible Steuerungssysteme. Dabei ist die Stückzahl in der Regel klein und die Lebensdauer der Maschinen lang, so dass die Nachrüstung von Steuerungssystemen an bestehenden Maschinen im Feld durchaus üblich ist. In **Bild 1** sehen Sie die Benutzeroberfläche des Systemsteuerungsmonitors nach dem Retrofit.



▲ Bild 1. ISOBUS-Benutzeroberfläche (ISO 11783) auf einem John-Deere-Monitor. Ein Güllefass wird gesteuert.

Der Aufwand für das Erlernen von C oder C++ und die Verwaltung von Bibliotheken und Systemen ist hoch. Das Debuggen mit JTAG ist unflexibel und erfordert zusätzliche Tools und den Zugriff auf den Mikrocontroller. Auf der anderen Seite stehen viele Programmierer für die SPS-Programmierung zur Verfügung und die grafisch basierte Programmierung ist leichter zu erlernen.

ISOBUS, auch bekannt als ISO 11783, ist ein Kommunikationsprotokoll, das für landwirtschaftliche Geräte entwickelt wurde, um den Datenaustausch zwischen verschiedenen Arten von Landmaschinen und -geräten zu erleichtern. Der Begriff „ISOBUS“ setzt sich zusammen aus „ISO“ für die Internationale Organisation für Normung und „Bus“ für ein Kommunikationssystem zwischen mehreren Geräten. In diesem Zusammenhang wollten wir ein SPS-System entwerfen, das aus einem Laufzeitsystem auf



dem Gerät und einer IDE auf dem PC besteht und ein flexibles, einfach zu bedienendes und schnelles Automatisierungssystem mit modellbasierter Low-Code- beziehungsweise grafischer Programmierung und leistungsstarken Bibliotheken bietet. Eine strikte Trennung zwischen einer wiederverwendbaren, flexiblen Laufzeitumgebung und der Anwendung hatte in unserem Lastenheft höchste Priorität. Als zweites war es uns wichtig, wirklich alle Variablen und Funktionsbausteine online zu beobachten. Weitere wichtige Punkte waren die Support-Qualität des Software-Stack-Anbieters sowie die Ressourcenutzung. Wir wollten ein teures und ausgewachsenes Embedded-Linux-System vermeiden, vor allem wegen der Boot-Zeiten, denn landwirtschaftliche Systeme werden wahrscheinlich ein Dutzend Mal am Tag eingeschaltet.

Wir analysierten mehrere Lösungen, sowohl Closed- als auch Open-Source-Lösungen und entschieden uns schließlich für Eclipse 4diac [1]. Und warum nicht das OpenPLC-Projekt, werden sich einige Elektor-Leser fragen? Die Antwort liegt auf der Hand: Die beiden wichtigsten Anforderungen wurden nicht erfüllt: OpenPLC hat weder eine strikte Trennung zwischen Anwendung und Laufzeit noch eine Funktionalität zur Online-Überwachung.

Die Lösung

Für die logiBUS-Lösung [2] haben wir die Softwareseite (siehe unten) mit unserer Basisanwendung kombiniert:

- ESP-IDF-Entwicklungsframework (derzeit Version 5.1.1)
- Open-Source-IEC 61499-Laufzeit: Eclipse 4diac FORTE PLC
- CCI-ISOBUS-Treiber, ein ISO11783-Protokoll-Stack

Dadurch ergibt sich ein SPS-Laufzeitsystem, das unabhängig von der gewünschten Anwendung ist. Die Programmierung erfolgt über die IDE Eclipse 4diac. Das Flashen der Anwendung kann über eine TCP/IP-Verbindung, also über WLAN oder Ethernet, erfolgen. Die Lösungen bringen echte SPS-Funktionen wie Online-Überwachung und Online-Änderungen mit sich.

Auf der Hardware-Seite ist es wichtig, ein ESP32 mit PSRAM zu verwenden, da das SW-Modell viel RAM verbraucht, so dass es bei größeren SW-Modellen ohne PSRAM zu Problemen kommen kann. Für die ISOBUS-Anbindung verwenden wir das ESP32-TWAI-Peripheriegerät als CAN-Bus-Controller, zusammen mit einem externen CAN-Transceiver TLE9251VSJ von Infineon. Einige Modelle der Hardware, insbesondere solche, die für den Bildungsbereich bestimmt sind, sind Open Source [6].

Da das logiBUS-Projekt immer noch wächst und ausgebaut wird, haben wir bereits ein Dutzend realer Kundenprojekte realisiert. In **Bild 2** sehen Sie ein solches Kundenbeispiel für ein Quellcodelisting. Auch für andere Branchen wie die Gebäudeautomation scheint das System gut geeignet zu sein [5]. Daher haben wir eine Open-Source-Version von logiBUS ohne ISOBUS-Stack, aber mit der gleichen Funktionalität und der gleichen Leistungsklasse erstellt.

Einige Demo-Anwendungen, zum Beispiel Bale Counter [4] und Slurry Tanker [5], wurden als Open Source zur Verfügung gestellt.

Wir hoffen, eine Community rund um das Thema Open-Source-SPS-Systeme aufzubauen zu können [6].

Übersetzung von Jürgen Keller – (230610-02)

Haben Sie Fragen oder Anmerkungen?

Haben Sie technische Fragen oder Anmerkungen zu diesem Artikel? Kontaktieren Sie uns unter redaktion@elektor.de.

WEBLINKS

- [1] Eclipse 4diac: <https://eclipse.dev/4diac>
- [2] logiBUS-Homepage: <https://www.logibus.tech>
- [3] logiBUS-Open-Source-Derivate beispielsweise für Gebäudeautomation (ohne ISOBUS): <https://gitlab.com/meisterschulen-am-ostbahnhof-munchen>
- [4] Ballenzähler: https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac_EasyExampleCounter
- [5] Göllefass: <https://github.com/Meisterschulen-am-Ostbahnhof-Munchen/4diac-SlurryTanker-sample>
- [6] Ressourcen und Wiki: <https://github.com/Meisterschulen-am-Ostbahnhof-Munchen>

Bild 2. Dieses Quellcodelisting zeigt, wie die Programmierung in IEC 61499/Eclipse 4diac funktioniert. Der Anwender schreibt keine einzige Zeile Quellcode, sondern verwendet Funktionsblöcke.

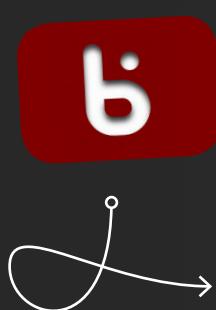


Import

Edit

Export

I_Made_a_VGA_Card...



Das ESP32-S3-VGA-Board

→ Bitlunis aufregende Reise in die Produktentwicklung

Zusammengestellt von Elektor und Espressif

Der YouTuber Bitluni entwickelte ein ESP32-S3-VGA-Board, um eine bemerkenswerte Auflösung und Farbtreue zu erreichen. Er nutzte die LCD-Peripherie des Espressif ESP32-S3, die den I²S der Vorgängerversion ersetzte. Lassen Sie uns seine Schritte nachverfolgen. Dabei werden Sie viel über den Prozess der Entwicklung eines neuen Produkts lernen.

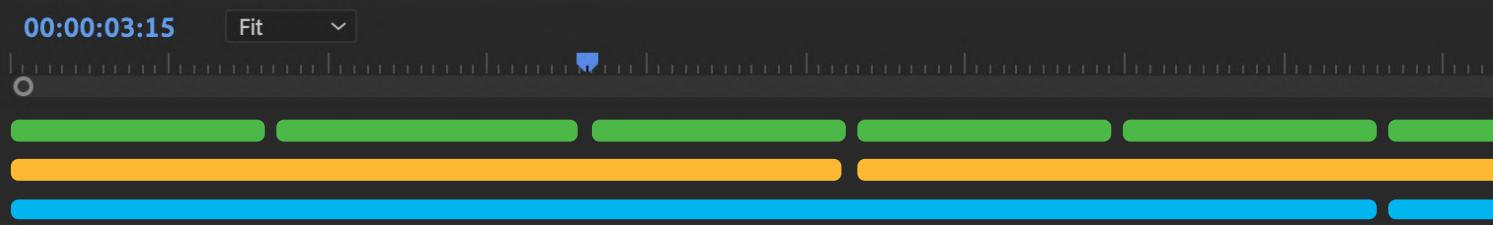
Vor Jahren hat der bekannte deutsche YouTuber Bitluni eine VGA-Bibliothek und ein paar VGA-Boards für den ESP32 entwickelt (**Bild 1**). Obwohl viele seiner treuen Follower den Entwurf liebten, betonte Bitluni stets, dass er kein Produzent wäre, so dass er nur wenige Exemplare verkauft hat. Seitdem haben sich die Dinge geändert! Die ESP32-API wurde einige Male aktualisiert, und der ESP32-S3 kam auf den Markt (**Bild 2**). Da so viele seiner Fans ihn baten, eine neue Version der Bibliothek und des Boards zu erstellen (da diese nicht mit dem S3 kompatibel waren), beschloss Bitluni, es zu versuchen. Auf diese Weise entstand die ESP32-S3-VGA (**Bild 3**). Als Bitluni beschloss, das neue Board (**Bild 4**) zu entwerfen, wollte er es leichter und auch für Nicht-Elektroniker zugänglich machen. Die früheren Boards mussten zusammengebaut werden und erforderten ein zusätzliches Entwicklungsboard. Glücklicherweise hatte er genug Erfahrung gesammelt, um eine Platine zu entwerfen, die alles enthält (**Bild 5**). Plug-and-play ohne viel Löten! Bitluni wollte zudem, dass man es problemlos auf ein Breadboard stecken konnte, wenn man entsprechende Header anlöste, und zwei zusätzliche Reihen für den Zugang zu den Pins zur Verfügung stellte (**Bild 6**). Der VGA-Anschluss war recht groß, und die Antenne brauchte etwas Spielraum, so dass diese Form sinnvoll erschien.

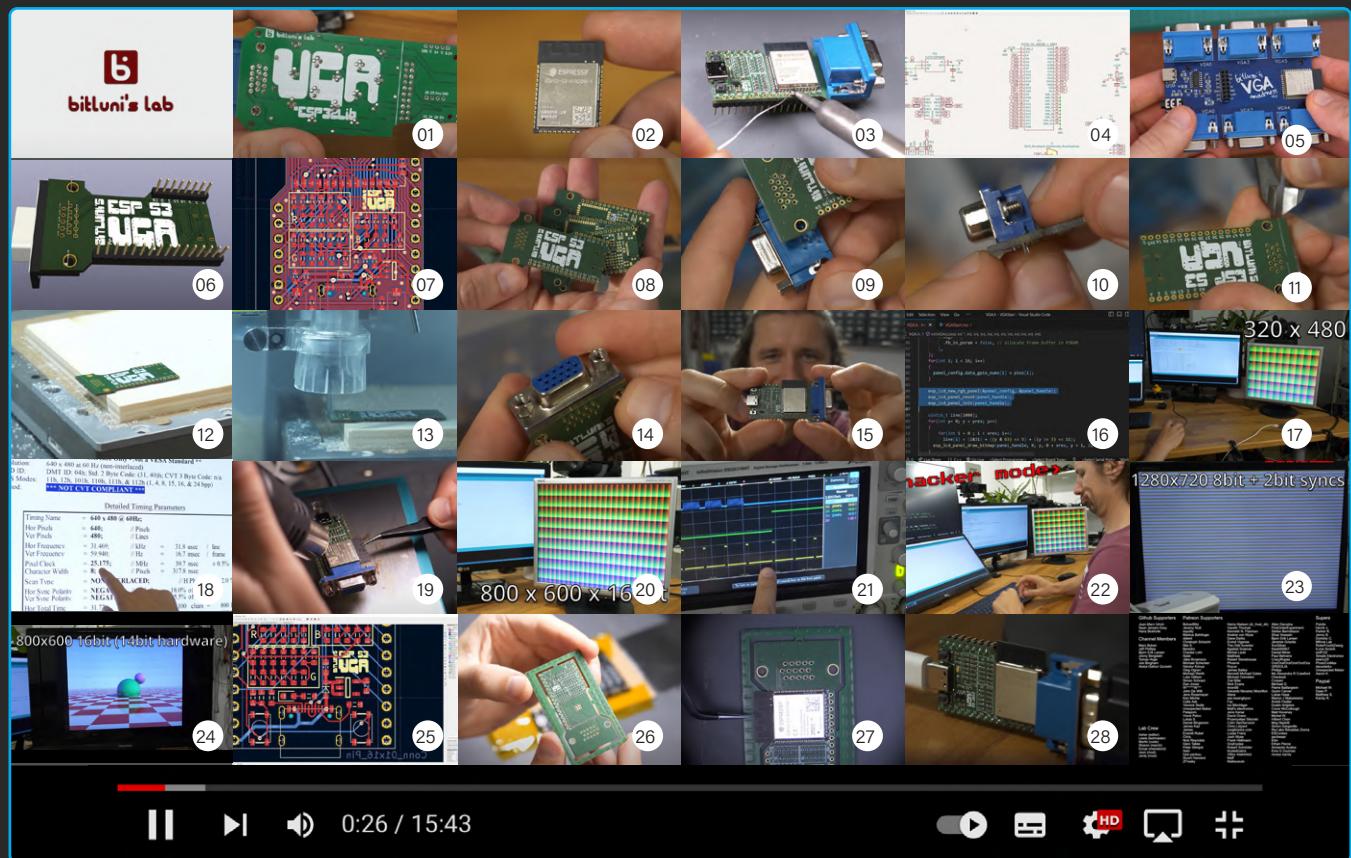
Das Layout wurde entworfen (**Bild 7**), Platten wurden bestellt, und das Ergebnis war umwerfend (**Bild 8**)! Bitluni ging davon aus, dass seine VGA-Buchsen mit dem Footprint der Platten kompatibel sein würden, aber er stellte bald fest, dass die Pins nicht übereinstimmten (**Bild 9**)! Zum Glück hatte er noch ein paar schmalere Buchsen zur Hand, die gerade so passten, obwohl die Platine ein paar Millimeter überstand (**Bild 10**). Da es in diesem Bereich keine Leiterbahnen gab, beschloss er, einfach ein paar Millimeter abzufräsen, und das Problem war gelöst (**Bilder 11...14**)! Nachdem die Montage abgeschlossen war, konnte er sich an den Code machen (**Bild 15**). Bis zu diesem Zeitpunkt hatte Bitluni nicht einen Blick in das technische Referenzhandbuch geworfen, aber als er dies tat, wurde ihm klar, warum seine alte Bibliothek für den S3 nicht funktionierte. Espressif hatte den parallelen I²S-Modus entfernt und eine neue Peripherie für Kameras und LCDs entwickelt. Es folgten also einige Studien (**Bild 16**), Überlegungen und Tests (**Bild 17**). Dann studierte er weiter, dachte weiter nach (**Bild 18**), testete weiter, entlötete und lötete neu, woraufhin er weiterhin neu entwarf, weiterlötete und weiter testete (**Bild 19**). Er war wirklich sehr beschäftigt! Glücklicherweise funktionierte das Gerät nach all der Mühe, die er auf sich genommen hatte, nicht nur mit 640×480, sondern auch mit einer Auflösung von 800×600 (**Bild 20**).

Nach einigen Tests auf verschiedenen Geräten entdeckte Bitluni einige Synchronisationsprobleme. Drei seiner anderen Bildschirme schienen die Synchronisation am Anfang jedes Bildes neu anzupassen. Das war ein Hindernis, erklärt er: Als er seinen Bildschirm näher betrachtete, stellte er eine leichte Verzögerung am Anfang jedes Bildes fest (**Bild 21**). Also studierte und überlegte er weiter und aktivierte den „Hacker-Modus“ (**Bild 22**)!

Es folgten positive Entwicklungen. „Ich war endlich in der Lage, ein sauberes, kontinuierliches Signal zu erhalten“, erklärt Bitluni. „Ich konnte endlich einige Erfolge im Livestream teilen.“ Er war sogar in der Lage, 1280×720p und 8-Bit herauszuholen (**Bild 23**). In diesem Bild ist nicht etwa ein Rauschen zu sehen, sondern superkleine Quadrate! Einen Tag später lächelte er wieder (**Bild 24**). Als Nächstes fügte er zusätzliche Bits hinzu, ordnete die Dinge neu (**Bild 25**) und bestellte neue Platten. Eine Woche später war es soweit: **Bild 26**, **Bild 27** und **Bild 28**. Erfolg! ↗

RG - 230529-02





bitluni

231K subscribers

**Passende Produkte**

- **Espressif ESP32-S3-EYE**
www.elektor.de/20626
- **D. Ibrahim, The Complete ESP32 Projects Guide** (Elektor, 2019)
Buch, Paperback, englisch:
www.elektor.de/18860
E-Buch, PDF, englisch:
www.elektor.de/18869



Sehen Sie: „I made a VGA card that blew my mind“

Ein ausführliches Video über dieses Projekt
finden Sie auf dem YouTube-Kanal von Bitluni:



Scannen
sie den
QR-VCode...

...und schauen
Sie das Video in
AR auf diesem
Artikel.

oder



Schauen Sie dieses Video
direkt auf YouTube
<https://youtu.be/muuhgrige5Q>



1/2 00:00:18:14

Akustischer Fingerabdruck mit ESP32

Songerkennung mit dem
Open-Source-Projekt Olaf

Von Joren Six, Universität Gent (Belgien)

Vor einigen Jahren wurde der Informatiker Joren Six damit beauftragt, eine tragbare Computerplattform – also einen Mikrocontroller – in die Lage zu versetzen, einen Song zu erkennen. Seine Experimente mit einem ESP32 führten zu einem vollwertigen, plattformübergreifenden Open-Source-Projekt, das er Olaf nannte, was für „Overly Lightweight Acoustic Fingerprinting“ steht. Olaf ist eine Musikerkennungsbibliothek, die auf Embedded-Plattformen mit stark begrenztem Speicher und begrenzter Rechenleistung einfach zu verwenden ist. Folgen wir ihm auf seiner Reise!

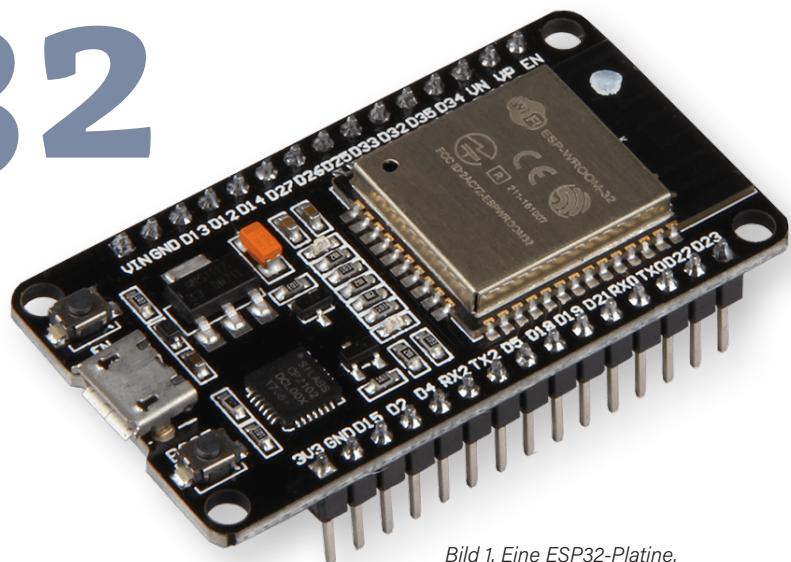


Bild 1. Eine ESP32-Platine.

Damals habe ich erfolgreich einen ersten Prototyp erstellt, aber die Idee blieb weiterhin in meinem Kopf.

Als der vierte Geburtstag meiner Tochter näher rückte, beschloss ich, den Prototyp in ein extravagantes Geburtstagsgeschenk zu verwandeln: ein „Elsa-Kleid“, das auf den Song „Let It Go“ aus dem Soundtrack von *Die Eiskönigin* reagiert [1]. Aufbauend auf dem ersten Prototyp bestellte ich einen RGB-LED-Streifen, einen robusten Li-Ionen-Akku, ein digitales I²S-Mikrofon und natürlich ein Elsa-Kleid. Ich hatte einen ESP32-Mikrocontroller (Bild 1) zur Hand und nutzte ihn als zentrale Komponente des Systems. Er unterstützt I²S, enthält eine Gleitkommaeinheit (FPU), lässt sich leicht mit LED-Streifen verbinden und verfügt über genügend RAM. Die FPU vereinfacht die Verwendung desselben Codes, sowohl auf herkömmlichen Computern als auch auf Embedded-Geräten, wodurch die Notwendigkeit von Festkommaarithmetik entfällt.

Aufbau der ESP32-Version

In der ersten Version des Projekts wurde ein ESP32 Thing von Sparkfun verwendet, das aufgrund seines integrierten Batteriemanagements ausgewählt wurde, und ein MEMS-Mikrofon, genauer gesagt ein INMP441. Ergänzt wurde dieser Aufbau durch einen LED-Streifen mit einzeln adressierbaren LEDs - ein generisches Bauteil, welches leicht bei eBay oder AliExpress zu finden ist.

Aus Hardware-Sicht ist es eine einfache Aufgabe. Es geht darum, die I/O-Pins des Mikrofonmoduls (SDA, SCK, WS) mit geeigneten GPIOs am ESP32 zu verbinden, zum Beispiel GPIO 32, 33 und 25. Natürlich braucht das Mikrofonmodul auch Strom.

Die Verkabelung der LEDs (Bild 2) hängt von der Art des verwendeten LED-Streifens ab. Für WS2812B-basierte Streifen werden drei Leitungen (Strom, Masse, Daten) benötigt. Ich habe die *FastLED*-Bibliothek im ESP32-Code verwendet. Wenn Sie also das Projekt nachbauen

Irgendwann um das Jahr 2019 herum war ich Informatiker an der Universität Gent in Belgien. Ich hatte die Aufgabe, eine Audioerkennungstechnologie für ein elektrifiziertes Kostüm zu entwickeln. Das Konzept sah vor, dass die Lichter im Kostüm mit einem bestimmten Lied synchronisiert werden. Dabei sollte aber nur ein bestimmtes Lied die Lichter aktivieren. Alle andere Musik sollte außer Acht gelassen werden. In der Regel werden Musikerkennung und -synchronisierung mithilfe von Audio-Fingerabdrucktechniken durchgeführt. Die Herausforderung bestand darin, dass diese Erkennung auf einem erschwinglichen, batteriebetriebenen Mikrocontroller mit begrenzter Rechenleistung und begrenztem Speicher betrieben werden musste.



Bild 2. RGB-LED-Streifen.

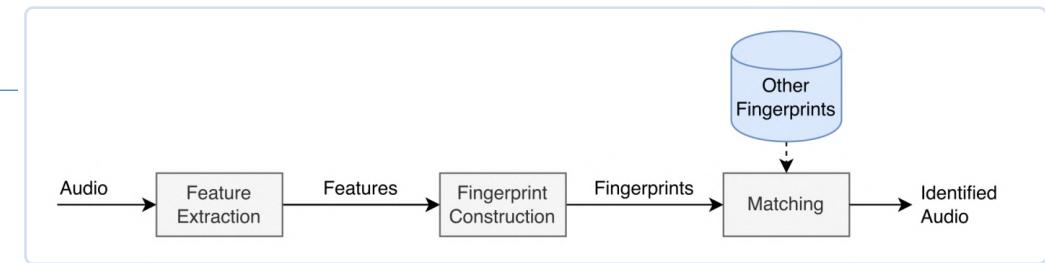


Bild 3. Blockdiagramm der Geräuscherkennung.

möchten, müssen Sie darauf achten, dass Ihr LED-Streifen mit *FastLED* kompatibel ist.

Nach dem Zusammenlöten der Bauteile und mit Hilfe meines Partners, der das Einnähen des LED-Streifens in das Kleid übernahm, wurde das Projekt nun realisiert. In dem unter [1] eingebetteten Video können Sie das Ergebnis unserer Arbeit begutachten. Das Video enthält zunächst einen Song, der nicht erkannt wird und die Beleuchtung nicht auslöst. Dann wird „Let It Go“ abgespielt und richtig erkannt. Sobald der Song pausiert, bleiben die Lichter für eine kurze Zeit eingeschaltet, bevor sie ausgeschaltet werden, um Lücken in der Erkennung auszugleichen. Schließlich wird das Lied fortgesetzt und wieder erkannt. Die neuesten Updates des Projekts können Sie unter [2] nachlesen. Wie so oft bei Mikrocontroller-Projekten steckt die ganze Magie im Code.

Was steckt dahinter?

Wie können wir Computer nutzen, um Lieder oder Musik zu erkennen? Es gibt verschiedene Möglichkeiten. Ein gängiger Ansatz basiert auf der spektralen Peak-Erkennung. Dies ist die gleiche Technologie, wie sie von Olaf und bekannten Musikerkennungsdiensten wie Shazam verwendet wird.

Das Konzept dahinter ist ganz einfach: Die App nimmt das auf Ihrem Handy aufgenommene Audio und wandelt es in ein Format um, welches ein Computer problemlos mit anderen Songs vergleichen kann (Bild 3). Im Bereich der Akustik wird dieser Prozess als Audio-Fingerprinting bezeichnet, bei dem ein Song zu einer einzigartigen Signatur verdichtet wird, die auch bei starken Hintergrundgeräuschen identifizierbar bleibt. Diese Signaturen basieren auf Spektrogrammen, die den Frequenzgehalt des Songs im Laufe der Zeit visuell darstellen. Spektrogramme zeigen auch den Signalleistungspegel an und spiegeln die wahrgenommene Lautstärke durch Farbvariationen wider.

Würde Shazam solche Spektrogramme jedoch direkt abgleichen, würde es außergewöhnlich lange dauern, bis ein Ergebnis vorliegt. Ein großer Durchbruch ist die Konzentration auf die Peaks im Spektrogramm, da diese Peaks wichtige Informationen enthalten, die auch vom menschlichen Gehirn verarbeitet werden.

Also nimmt Olaf ein Klangsample auf und berechnet dann ein Spektrogramm. Als nächstes wird das Spektrogramm zu einem Streudiagramm seiner Frequenzspitzen vereinfacht. Ein Beispiel für ein solches Spektrogramm mit hervorgehobenen Peaks ist in Bild 4 dargestellt. Nun müssen diese Streudiagramme, die im Wesentlichen die auffälligsten Signale bei verschiedenen Frequenzen im Zeitverlauf darstellen, mit einer Datenbank aus vielen bekannten Liedern abgeglichen werden. Oder, wie im Fall von Elsas Kostüm zu einem einzigen bestimmten Lied. Anstatt in einer Datenbank nach einer Übereinstimmung für all diese Punkte in der exakten Reihenfolge zu suchen, besteht eine clevere Technik darin, nahe gelegene Peaks zu Paaren zu verbinden. Anschließend sucht der Algorithmus nach den passenden Paaren in einer strukturierten Datenbank, die beliebig viele Songs enthält. Wenn genügend Übereinstimmungen gefunden werden und sie im Zeitverlauf korrekt übereinstimmen, kann Olaf (oder Shazam) den Song identifizieren.

Evolution des Projekts

Der ursprüngliche Code für den Prototyp von 2019 war nicht sehr gut strukturiert. Bei meinem zweiten Versuch habe ich jedoch erhebliche Verbesserungen vorgenommen und eine ausreichende Qualität erreicht, sodass ich den Code auf GitHub teilen konnte. Seit diesem Zeitpunkt hieß das Projekt *Olaf - Overly Lightweight Acoustic Fingerprinting* [3]. Der Code durchlief mehrere Iterationen und erweiterte sich über seinen ursprünglichen Zweck hinaus zu einem vielseitigen, universellen akustischen Fingerabdrucksystem mit zahlreichen potenziell

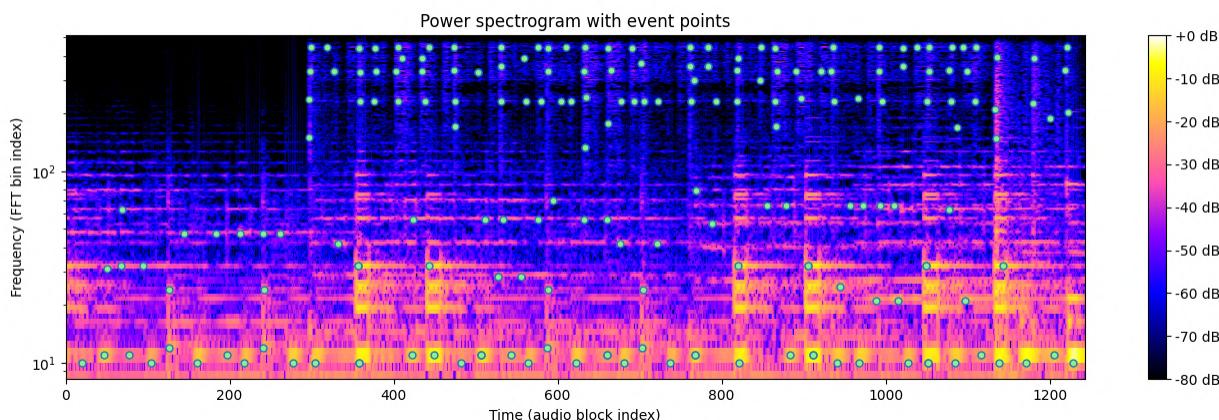
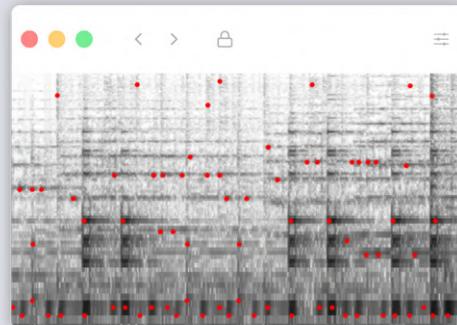


Bild 4. Ein Spektrogramm zeigt, wie präsent die Frequenzen in der Musik zu einem bestimmten Zeitpunkt sind. Die grünen Punkte werden von Olaf extrahiert und zeigen Peaks im Spektrogramm.

Ausführung im Webbrowser

Haben Sie schon von WebAssembly [5] (oder WASM) gehört? WASM ist ein portables Binärcodeformat, zusammen mit einem entsprechenden Textformat, das für ausführbare Programme entwickelt wurde. Sein Hauptziel ist es, die Entwicklung von Hochleistungsanwendungen innerhalb von Webseiten zu erleichtern. Wie sich herausstellt, gibt es Möglichkeiten, C-Code in WASM zu kompilieren, zum Beispiel mit dem Emscripten-Compiler. Laut seiner Website können Sie mit Emscripten C- und C++-Code im Web mit nahezu nativer Geschwindigkeit ausführen, ohne dass Plugins erforderlich wären. Die Kombination der Web-Audio-API mit der WASM-Version von Olaf eröffnet Möglichkeiten für webbasierte akustische Fingerabdruck-Anwendungen.

Auf meinem Blog können Sie direkt in Ihrem Browser mit Olaf experimentieren und meine neuesten Beiträge darüber lesen. Genau derselbe Code, der auf dem ESP32 läuft, wird jetzt in Ihrem Browser ausgeführt. Das bedeutet, dass Olaf aktiv zuhört, um den Song „Let It Go“ aus dem Soundtrack von „Die Eiskönigin“ zu erkennen. Der Einfachheit halber können Sie das Lied über das YouTube-Video auf der linken Seite Ihres Bildschirms abspielen, und auf der rechten Seite können Sie Olaf starten, damit es eingehende Audiodaten analysieren kann. Olaf berechnet die Fast Fourier-Transformation (FFT) und visualisiert sie mit Pixi.js. Nach einigen Sekunden sollten die roten Fingerabdrücke (siehe Screenshot) in grüne übergehen, was auf eine erfolgreiche Übereinstimmung hinweist. Wenn Sie den Song stoppen, werden die Fingerabdrücke schließlich wieder rot. Ähnlich wie bei der bereits erwähnten Videodemonstration dauert der Übergang von einer Übereinstimmung zu keiner Übereinstimmung einige Sekunden, um Lücken in der Erkennung zu berücksichtigen.



len Anwendungen. Die beeindruckende Leistung von Olaf ist unter anderem auf sein ressourceneffizientes Design und die Nutzung der *PFFT*-Bibliothek zurückzuführen.

Dieses Akronym mag vielen Lesern nicht viel sagen, vor allem, wenn Ihre Lieblingsprogrammiersprache Lötzinn ist, wie die von Bob Pease! *PFFT* [4] steht für *Pretty Fast FFT Library* und wurde als viel leichtere Alternative als das bekannte *FFTW* entwickelt. Die Bibliothek ist in der Tat kompakt und arbeitet sehr schnell, was sie ideal für den ESP32 macht.

Auf Embedded-Geräten werden Referenzfingerabdrücke im Speicher abgelegt, sodass keine Datenbank erforderlich ist. Auf herkömmlichen Computern werden Fingerabdrücke dagegen in einem leistungsstarken Datenbanksystem gespeichert – LMDB. Obwohl es den Rahmen dieses Artikels sprengen würde, sich mit den Vorteilen zu befassen, ermutige ich die Leser, dies bei entsprechendem Interesse selbst zu tun. Olaf ist nicht mehr nur ein cooles Gadget auf ESP32-Basis. Es wurde in eine vollständige Anwendung/Bibliothek umgewandelt, sodass es nun die Grundlage für den akustischen Fingerabdruck darstellt. Olaf ist in der Lage, Fingerabdrücke effizient aus einem Audiostream zu extrahieren. Dies ermöglicht zum einen die Speicherung dieser Fingerabdrücke in einer Datenbank oder zum anderen die Erkennung von Übereinstimmungen zwischen extrahierten und gespeicherten Fingerabdrücken.

Kleine akustische Fingerabdruckbibliotheken, die sich leicht in Embedded-Plattformen integrieren ließen, waren bisher Mangelware, da sie oft große Ansprüche in Bezug auf Speicher und Rechenressourcen hatten. Olaf ist in C geschrieben und ich habe mich bemüht den Code so portabel wie möglich zu halten. Er zielt in erster Linie auf 32-Bit-ARM-Geräte wie bestimmte Teensy-Modelle, bestimmte Arduino-Boards und den ESP32 ab. Andere moderne Embedded-Plattformen mit ähnlichen Spezifikationen könnten ebenfalls kompatibel sein. Es besteht kein Zweifel, dass Olaf, das Open Source ist, die Entwicklung innovativer Projekte, welche Musik-/Audioerkennung mit IoT-Geräten realisieren wollen, anregen wird!

Der geschickte Ansatz für Embedded Systeme hat zur Folge, dass Olaf auch auf PCs läuft, und zwar schnell. Der Code kann kompiliert werden, um lokal auf Ihrem PC ausgeführt zu werden, aber eine Alternative ist auch möglich: Olaf kann zudem auch in einem Webbrowser laufen! Weitere Informationen finden Sie im Kasten **Ausführung im Webbrowser**.

Den Quellcode mit einem funktionierenden Beispiel für ESP32 und kleinen Debug-Tools, die ich geschrieben habe, finden Sie auf meinem GitHub-Repository. Es gibt auch eine PlatformIO-Projektdatei als Referenz [6].

Selbstbau mit ESP32

Für die Audioverarbeitung benötigte das Olaf-Projekt ein wenig RAM, was bei vielen Mikrocontrollern oft ein Problem darstellte. Mit einem Key-Value-Speicher, der auf einem PC lief, benötigte Olaf etwa 512 KB RAM, während die ESP32-Version mit etwa 200 KB deutlich weniger benötigte. Ich arbeite nur sporadisch mit Mikrocontrollern, daher ist eine einfach zu bedienende Umgebung für mich entscheidend: Die begrenzte Zeit, die ich mit Embedded-Geräten verbringen kann, sollte nicht durch die Einrichtung und Wartung einer Entwicklungsumgebung verschwendet werden. Die IDE des ESP32 mit PlatformIO oder die Arduino-IDE erwiesen sich als geeignet und viele Mikrocontroller

funktionierten damit (Teensy 3+, RP2040, die Cortex-M-Familie). Eine FPU kann dazu beitragen, den Energiebedarf niedrig zu halten, was besonders bei Projekten mit Batterieversorgung relevant ist.

Ich habe auch ein paar M5StickCs der Firma M5Stack verwendet, da sie ein einfacher zu bedienendes Gesamtpaket anbieten und mit einem integrierten Mikrofon geliefert werden. Ich interessiere mich mehr für Software als für Hardcore-Hardware-Hacking, daher waren sie meine bevorzugte Wahl. Ich habe auch ein paar Hausautomationsgeräte gebaut (zu finden auf meiner Website) sowie einige andere Projekte, zum Beispiel die Steuerung der Haslüftung oder die Überwachung des Füllstands des Regenwassertanks.

Bei der Vorbereitung dieses Artikels habe ich den Code für eine ESP32-Demo sowie zusätzliche Dokumentation zu meinem GitHub-Repository hinzugefügt [7]. Jetzt arbeitet die neueste Version von Olaf zuverlässig auf dem ESP32 mit einem leicht erhältlichen MEMS-Mikrofon. Auf dieser Hilfeseite finden Sie auch Anweisungen zur Verwendung von I²S-Mikrofonen wie dem INMP441.

Es gibt ein Demoprogramm, das Audio vom Mikrofon über ein WLAN an einen Computer überträgt, sodass man dort das Signal hören kann. Dadurch wird sichergestellt, dass das Mikrofon wie vorgesehen funktioniert und die Audiobeispiele genau interpretiert werden. Es überprüft die I²S-Einstellungen, einschließlich Puffergrößen, Abtastraten, Audioformate und Stereo-/Monoeinstellung.

Es gibt auch ein weiteres Codebeispiel, welches veranschaulicht, wie eingehende Audiodaten von einem INMP441-Mikrofon mit Fingerabdrücken abgeglichen werden, die in einer Headerdatei gespeichert sind. Im Gegensatz zur PC-Version verwendet die eingebettete Version von Olaf keinen Key-Value-Speicher, sondern eine Liste von Hashes, die in der Header-Datei *src/olaf_fp_ref_mem.h* gespeichert sind und als Fingerabdruckindex dient.

Standardmäßig ist *src/olaf_fp_ref_mem.h* im ESP32-Code enthalten. Um diese Header-Datei zu testen und zu debuggen, fragen Sie die MEM-Version von Olaf auf Ihrem Computer ab: bin/olaf query olaf_audio_your_audio_file.raw "arandomidentifier". Die ESP32-Version ist im Wesentlichen identisch mit der MEM-Version, mit dem einzigen Unterschied, dass der Ton in der ESP32-Version von einem MEMS-Mikrofoneingang und nicht von einer Datei stammt.

Sobald Sie sicher sind, dass die mem-Version von Olaf wie erwartet funktioniert und das INMP441-Mikrofon getestet wurde, kann die ESP32-Version mit der Arduino-IDE auf der ESP32-Hardware realisiert werden.



Bild 5. Das Kostüm mit ESP32-Antrieb.

Was kommt als nächstes?

Ich hoffe, dass dieser Artikel einige Ideen für Projekte mit akustischer Erkennung ausgelöst hat. Es ist immer wieder spannend zu sehen, wie man mit einem kleinen, spielerischen Projekt beginnt und es dann Stück für Stück verfeinert, bis man beeindruckende Ergebnisse erzielt. In diesem Fall entwickelte sich das Projekt von einem ersten Prototyp, der meine Erwartungen nicht ganz erfüllte, zu einem zweiten

Prototyp auf Basis eines ESP32, der mehrere Verbesserungen und zweifellos viel Freude für meine Tochter brachte (**Bild 5**).

Im Laufe der Zeit und durch iterative Verfeinerungen entwickelte sich das Projekt zu einer umfassenden, portablen, quelloffenen, plattformübergreifenden, leistungsstarken Anwendung und Bibliothek. Ich habe zwei wissenschaftliche Arbeiten zu diesem Thema verfasst, damit diese Arbeit in der Forschungsgemeinschaft anerkannt und geschätzt werden kann. Nun stellt sich für die Leser die Frage: Welches spannende Projekt mit ihrem ESP32 schwebt Ihnen vor? ↗

Übersetzung von Jürgen Keller – 230578-02

Haben Sie Fragen oder Anmerkungen?

Wenn Sie technische Fragen oder Anmerkungen zu diesem Artikel haben, wenden Sie sich bitte an den Autor bei joren.six@ugent.be oder an die Elektor-Redaktion unter redaktion@elektor.de.



Über den Autor

Joren Six ist Informatiker, der auf dem Gebiet der Musikinformatik, der Musik-Informationsgewinnung (Music Information Retrieval) und der rechnergestützten Ethnomusikologie (Computational Ethnomusicology) forscht. Er promovierte an der Universität Gent, Belgien und ist derzeit an mehreren Projekten beteiligt, die IT und Akustik kombinieren.



Passende Produkte

- **JOY-iT NodeMCU ESP32 Entwicklungsboard**
www.elektor.de/19973
- **ESP32-DevKitC-32E**
www.elektor.de/20518
- **ESP32-C3-DevKitM-1**
www.elektor.de/20324

WEBLINKS

- [1] Erster Prototype des Projekts: https://0110.be/posts/Olaf_-_Acoustic_fingerprinting_on_the_ESP32_and_in_the_Browser
- [2] Letzte Updates des Projekts: <https://0110.be/search?q=olaf>
- [3] Repository des Projekts: <https://github.com/JorenSix/Olaf>
- [4] Pretty Fast FFT Library: <https://bitbucket.org/jpommier/pfft/src/master/>
- [5] WebAssembly auf Wikipedia: <https://de.wikipedia.org/wiki/WebAssembly>
- [6] Dateidownload: <https://0110.be/files/attachments/475/ESP32-Olaf.zip>
- [7] ESP32-Olaf-Projekt auf GitHub: <https://github.com/JorenSix/Olaf/tree/master/ESP32>



Zirkularer Weih- nachts- baum

Weihnachtszeit, eine
High-Tech-Annäherung

Von Ton Giesberts (Elektor)

Die WS2812D-F8 ist eine digital programmierbare 8-mm-RGB-LED, die individuell in Ketten von bis zu 255 LEDs adressiert werden kann. In diesem kreisförmigen 3D-Weihnachtsbaum-Projekt werden 36 dieser LEDs extern oder über einen integrierten Arduino Nano ESP32 gesteuert. Die Lichteffekte sind verblüffend - dieses leuchtende Elektor-Kit sollte man sich für die Feiertage nicht entgehen lassen!

Bei diesem Weihnachtsbaum dreht sich alles um die Form der Konstruktion und die Verwendung von digitalen 8-mm-RGB-LEDs. Er sieht eher wie ein echter Baum aus als die 2D-Designs mit flachen Platten im Umriss eines Tannenbaums. Es handelt sich um ein echtes 3D-Design, das auf den beiden früheren Versionen [1] und [2] basiert. Der neue Baum besteht aus fünf ringförmigen, konzentrischen Platten-Abschnitten im Bausatz, die mit einigen starren Drähten getrennt montiert werden. Dies verleiht diesem Projekt das typische Aussehen eines typischen Nadelbaums.

Das Projekt in [1] verwendete einen einfachen Zufallszahlengenerator mit zwei Standard-Logik-ICs und kleinen weißen SMD-LEDs.

Das Projekt in [2] war mikrocontrollerbasiert, wobei die LEDs in einer 6x6-Matrix angeschlossen waren. Diese neueste Version besitzt 36 große bedrahtete, digitale 8-mm-RGB-LEDs vom Typ WS2812D-F8. Das macht die Schaltung einfach, denn anstelle in einer Matrix sind die Ein- und Ausgänge der digitalen LEDs in Reihe geschaltet, wobei jede LED an die 5-V-Stromversorgung angeschlossen ist. Die LED-Kette kann wie eine NeoPixel-LED-Leiste angesprochen werden. Es gibt eine große Anzahl von Programmen im Internet, in vielen Sprachen, die detailliert erklären, wie man diese Bauteile ansteuert. Der Anschluss K1 ist mit den internen +5 V (4,3 V), der Masse und über Jumper JP1 mit dem Eingang der ersten LED verbunden. Auf diese Weise kann eine Vielzahl von externen Mikroprozessoren, Mikrocontrollern oder Modulen verwendet werden, um die LEDs unseres kreisförmigen Weihnachtsbaums zu steuern. Um den Baum unabhängig von einer externen Schaltung zu machen, kann ein Arduino Nano ESP32-Modul [3] auf der Basisplatine angebracht werden.

Der Schaltplan

Die Schaltung besteht im Wesentlichen aus 36 digitalen RGB-LEDs des Typs WS2812D-F8, deren Eingangs- (Din, Pin 4) und Ausgangspins (Dout, Pin 1) in Reihe geschaltet sind und die jeweils von einem 5-V-Netzteil gespeist werden (**Bild 1**). Die tatsächliche Spannung an den LEDs ist aufgrund des durch die Schutzdioden D1 und D2 verursachten Spannungsabfalls niedriger, etwa 4,3 V. Die LEDs können von einem externen Mikroprozessor, Mikrocontroller, Modul oder dem optionalen Arduino Nano ESP32-Modul (MOD1) gesteuert werden. Der Jumper JP1, der in **Bild 2** in der Nähe der Arduino-Nano-Platine zu sehen ist, wählt das Steuersignal von LED1 aus (die erste

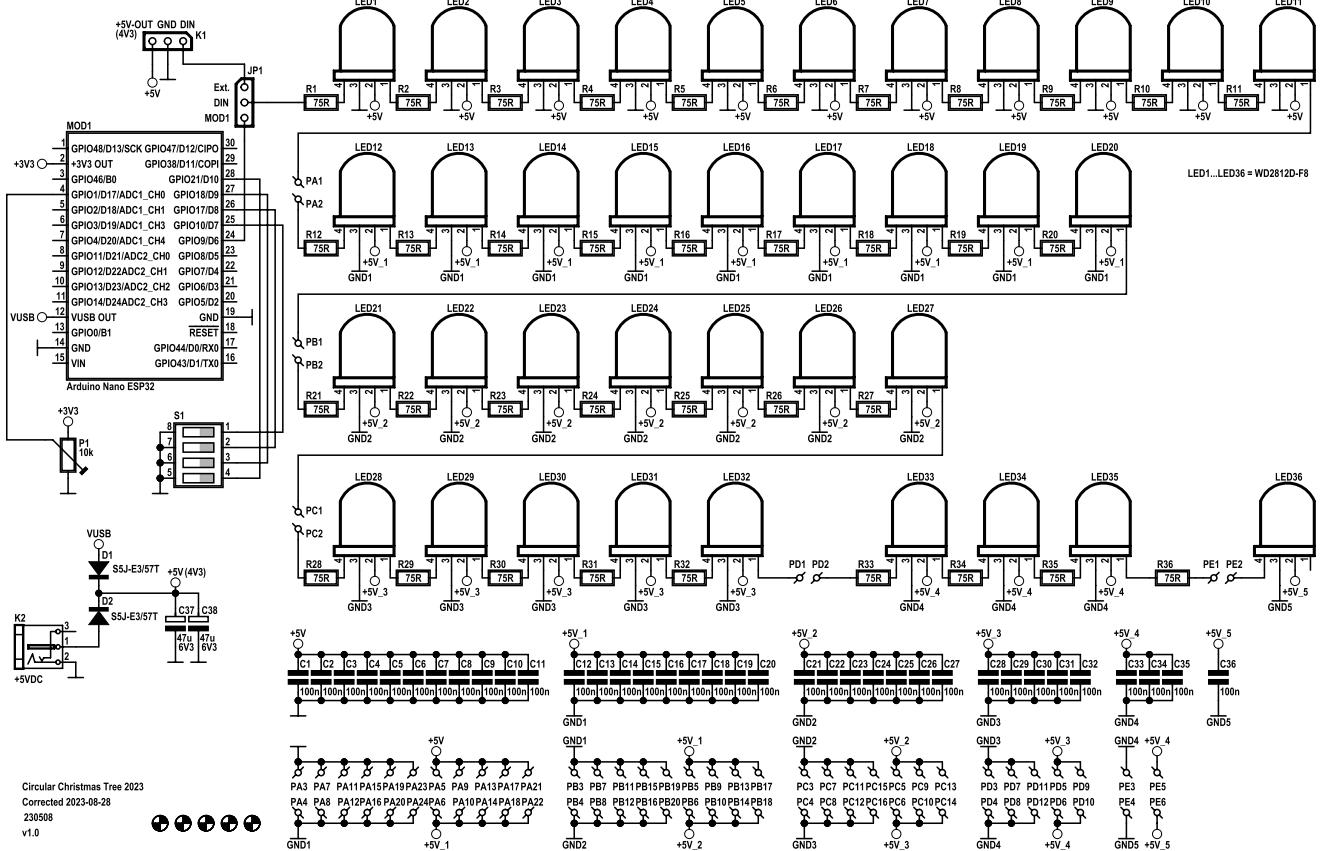


Bild 1. Schaltung des zirkulären Weihnachtsbaums.

der 36 verketteten LEDs). Durch den Arduino Nano ESP32 wird der Baum noch vielseitiger, denn über die Software ist es möglich, die Beleuchtungsmuster des Baums aus der Ferne über das integrierte WLAN und/oder Bluetooth LE zu ändern. Alle LED-Eingänge sind mit $75\text{-}\Omega$ -Widerständen in Reihe geschaltet (R1..R36, gemäß den im Datenblatt des WS2812D-F8 [4] angegebenen Anforderungen). Jede LED ist außerdem durch einen 100-nF-Kondensator (C1..C36) entkoppelt. Die Stromversorgung für die LEDs auf der Basisplatine wird zusätzlich durch die beiden 47- μF -Tantalkondensatoren C37 und C38 entkoppelt. All diese Bauteile sind als SMDs ausgeführt und unsichtbar auf der Unterseite der Platine platziert. Die LEDs können über ein 5-V_{DC} -Netzteil, vorzugsweise mit mindestens 1,5 A, über den Gleichstromanschluss K2 oder über den VBUS-Pin auf der Arduino-Nano-Platine (falls vorhanden) im Bereich der MOD1-Platine mit Strom versorgt werden. Die Dioden D1 und D2 (5 A, 50 V_{DC}, S5J-E3/57T, SMD-Gehäuse, Größe SMC), die ebenfalls auf der Unterseite der Platine montiert sind, verhindern Schäden, wenn die beiden Stromversorgungen gleichzeitig angeschlossen sind. Es ist zwar möglich, sowohl MOD1 als auch K2 mit einem Netzteil zu versorgen, aber nicht notwendig. Das Modul selbst wird über seinen USB-C-Anschluss mit Strom versorgt, und der maximale Strom des VBUS-Pins ist ausreichend (vorausgesetzt natürlich, dass das angeschlossene Netzteil stark genug ist). Die Dioden sind absichtlich keine Schottky-Typen. Der Spannungsabfall über die verwendeten Dioden ist höher, weil wir dafür sorgen müssen, dass die 3,3-V-Logiksignale innerhalb der Spezifikation des WS2812D-F8 liegen. Wenn die LED-Kette extern gesteuert wird, ist es daher immer die beste Option, diese Schaltung über die interne Stromquelle von K1 zu versorgen. Die tatsächliche Spannung an K1 ist



Bild 2. Detail von MOD1, S1 und P1 auf der Basisplatine.

wegen des Spannungsabfalls an D2 niedriger als die 5 V von K2 - etwa 4,3 V (oder sogar niedriger, bei hoher Strom-/Helligkeitseinstellung). Um die Platinen zu verbinden, sorgen die 39 Drahtsegmente nicht nur für Abstand, sondern leiten die +5 V und die Masse von einer Platine zu der darüber liegenden (Bild 3). Im Schaltplan sind dies die Verbindungen PA1..PA2 bis PE5..PE6. Ein zusätzliches Kabel verbindet jeden LED-Dout-Pin auf einer unteren mit der ersten LED auf dem darüber liegenden Platinenring.

Wenn ein Arduino Nano ESP32 zur Steuerung der LEDs verwendet wird, können der DIP-Schalter S1 und der kleine Trimmer P1, die beide in Bild 2 zu sehen sind, ebenfalls auf der Platine angebracht werden. Die Software kann diese Komponenten auslesen, um verschiedene Muster auszuwählen oder die Helligkeit einzustellen. Ohne das Onboard-Modul haben diese Komponenten keinen Daseinszweck.



Bild 3. Ein Draht markiert und ein anderer abisoliert.
Die verbleibende Isolierung sollte 3 cm lang sein.

Stromversorgung

Die Stromversorgung für die LEDs erfolgt über einen +5V-Steckernetzteil mit Hohlstecker, der an K2 angeschlossen wird. Der in unserem Shop erhältliche Bausatz enthält nur die Teile für die Platine, einschließlich des kleinen Trimmotors und der DIP-Schalter. Das Steckernetzteil und das optionale Arduino Nano ESP32-Modul sind nicht enthalten und müssen separat erworben werden.

Das Modul besitzt einen USB-C-Anschluss, der auch die LEDs über seinen VBUS-Pin mit Strom versorgt. Wie oben erwähnt, verhindern zwei Dioden (D1, D2), dass sich die beiden Stromversorgungen ins Gehege kommen. VBUS ist intern über eine Schottky-Diode vom Typ PMEG6020AELR (Nexperia) mit den 5 V des USB-C-Anschlusses verbunden. Diese Diode ist für 2 A ausgelegt. Leider ist der maximale Strom des VBUS-Pins im Datenblatt nicht angegeben [5]. Der Strom der WS2812D-F8 beträgt 12 mA, wie aus einer Übersicht über diese Familie auf der Website des Herstellers hervorgeht.

Im Datenblatt sind jedoch 15 mA angegeben, und viele Parameter werden unter der Bedingung $I_F = 20 \text{ mA}$ bewertet. Bei unserem Prototyp lag der maximale Strom bei maximaler Einstellung ($3 \times 255_D$ pro LED) bei knapp 11 mA pro Farbe - weit entfernt von diesen 20 mA. Die Abweichungen von den Spezifikationen sind ein wenig verwirrend, denn offenbar liegt ein realistischerer Wert für den Strom bei etwa 12 mA. Angenommen, dies ist der richtige Wert, so ergibt sich der maximale Strom bei maximaler LED-Helligkeit:

$$36 \text{ (LEDs)} \times 3 \text{ (Farben)} \times 12 \text{ mA} = 1,3 \text{ A.}$$

In unserem Weihnachtsbaum-Prototyp beträgt die maximale Gesamtstromaufnahme gemessene 1.156 mA. Der VBUS-Pin am MOD1 sollte mit diesem Strom kein Problem haben. Generell raten wir aber dringend davon ab, den maximalen Nennstrom einer LED zu verwenden, da sich dadurch die Lebensdauer der LED erheblich verkürzt! Der Helligkeitsunterschied zwischen zum Beispiel einem

Drittel des Nennstroms und dem Maximalwert ist gering. Berücksichtigen Sie dies beim Schreiben der Software! Die Einstellung aller LEDs auf weißes Licht für eine sehr helle Beleuchtung in einem gut beleuchteten Raum erfordert einen Gesamtstrom von etwa 200 mA. Dies ist in den meisten Fällen mehr als ausreichend und ermöglicht es Ihnen, jeden Standard-USB-Typ-A-Anschluss zur Stromversorgung des Moduls zu verwenden.

Runde Platinen

Wie erwartet, besteht die 136x136 mm große Platine aus der Basisplatine und fünf ringförmigen Platinen, die von 24 Trennbrücken zusammengehalten werden.

Die Abbildungen im Rahmen der **Stückliste** zeigen die Gesamtplatine, während **Bild 4** alle kleineren Komponenten der Platine nach dem Abbrechen (besser: Durchknipsen) der Brücken veranschaulicht. Mit einer Halbrundfeile kann man stehengebliebene Reste der Brücken sauber entfernen. Eine Bauanleitung auf Elektor Labs [6] enthält alle Details zum Bau des Baums. Das Kupfer der 39 Drahtsegmente ist 0,8 mm dick. Da diese Drähte auch den Baum formen, haben wir massiven Draht mit grüner Isolierung gewählt. Wenn Sie jedoch der Meinung sind, dass ein blanker, verzinnerter Draht besser aussieht, können Sie natürlich die gesamte Isolierung entfernen.

Achten Sie genau darauf, dass alle Platinen parallel montiert sind, mit einem Abstand von 3 cm zur nächsten Platine. Die große Anzahl von Drähten macht die Konstruktion recht steif, wie Sie in **Bild 5** sehen können. Diese zeigt den fertigen Prototyp ohne das Zusatzmodul. Um den Spannungsabfall von unten nach oben so gering wie möglich zu halten, sind alle Drähte (außer der Signaldraht) abwechselnd an +5 V und Masse angeschlossen. Achten Sie beim Abisolieren darauf, dass keine Kurzschlüsse zwischen den blanken Drähten entstehen! Die Leiterbahnen für die Spannungsversorgung sind 1 mm breit. Alle SMD-Bauteile sind auf der Unterseite der Leiterplatten angebracht, wie in **Bild 6** zu sehen ist. Trennen Sie die Platinenabschnitte, bevor

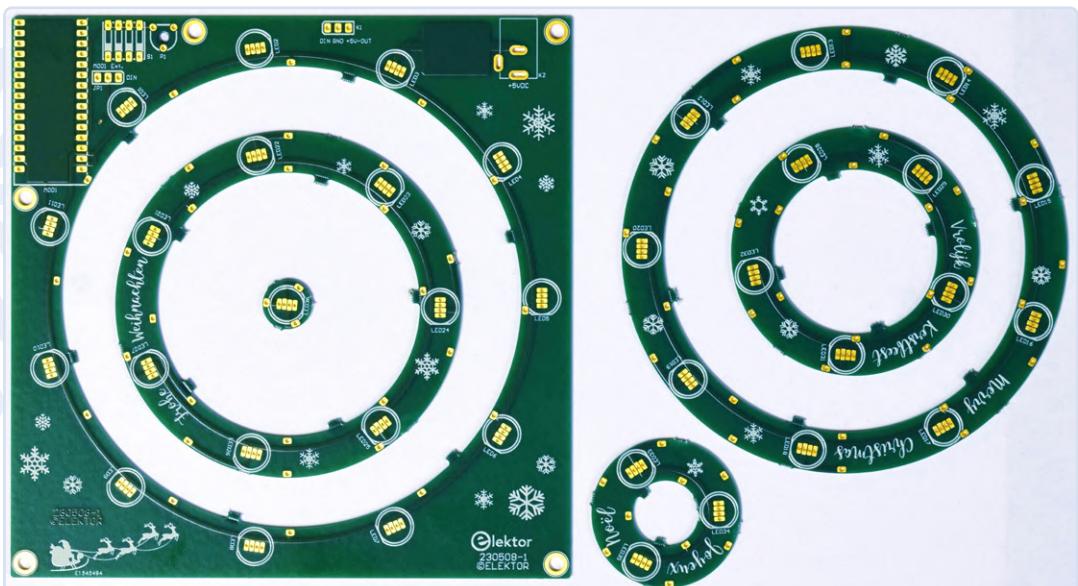
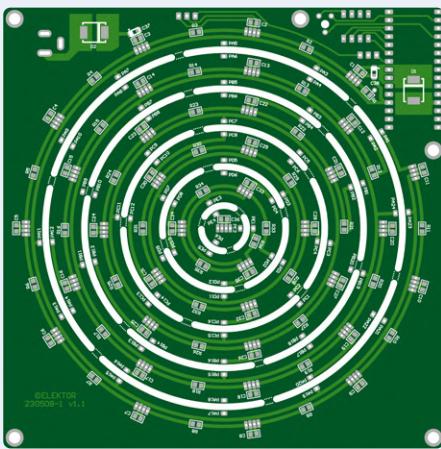


Bild 4. Alle Platinen sind getrennt. Die Überbrückungssegmente, die die sechs Platinen verbunden haben, können nun mit einem Seitenschneider und einer Halbrundfeile entfernt werden.



Stückliste



Widerstände:

R1...R36 = 75 Ω , 0W125, 5 %, SMD 0805

P1 = 10 k, 0W1, 20 %, Trimmpot, von oben einstellbar, 6 mm, rund
(Piher PT6KV-103A2020)

Kondensatoren:

C1...C36 = 100 n, 50 V, 5 %, X7R, SMD 0805

C37, C38 = 47 μ , 6V3, 10 %, Tantal, Gehäuse A (1206)

Halbleiter:

D1, D2 = S5J-E3/57T, SMD, Gehäuse SMC

LED1...LED36 = WS2812D-F8, 8 mm, THT

MOD1 (optional) = Arduino Nano ESP32 mit Stifteleisten

Außerdem:

K1, JP1 = 1x3-polige Stifteleiste, vertikal, Raster 2,54 mm

Jumper für JP1, Raster 2,54 mm

K2 = MJ-179PH (Multicomp Pro), DC-Buchse, 4 A,

Stiftdurchmesser 1,95 mm

S1 = DIP-Schalter, 4-fach

PA1...PE6 = 2 m massives Kabel, 0,81 mm, 0,52 mm² / 20AWG,
grün isoliert (Alpha Wire 3053/1 GR005)

H1...H5 = Nylon-Abstandhalter, Buchse-Buchse, M3, 5 mm

H1...H5 = Nylonschraube, M3, 5 mm

Platine 230508-1 (136 x 136 mm)

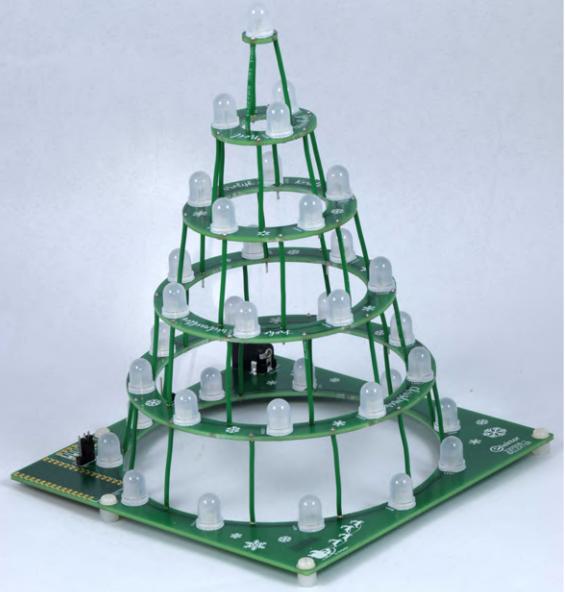


Bild 5. Der fertige Prototyp, ohne das optionale Arduino-Modul.

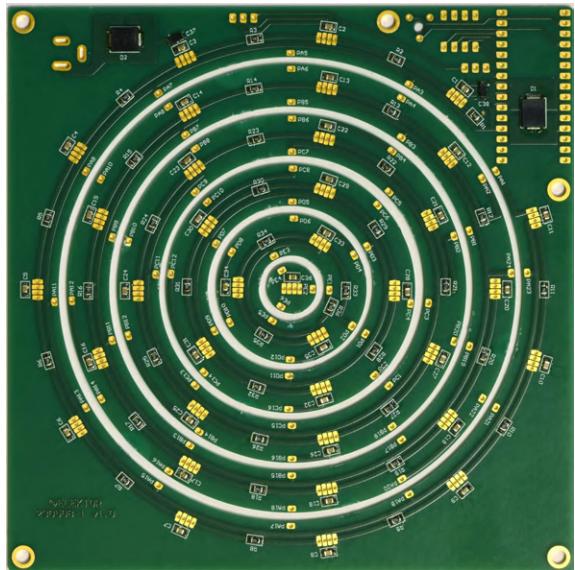


Bild 6. Alle SMDs sind auf den Rückseiten der Platinen verlötet:
R1...R36, C1...C38, und D1 und D2.

Sie mit dem Löten beginnen. Auf der Oberseite der ringförmigen Platten befinden sich keine weiteren Bauteile, mit Ausnahme der LEDs, deren Leitungen wie in Bild 7 gezeigt abgeschnitten werden müssen. Die bedrahteten Bauteile JP1, K1, K2, P1, S1 und MOD1 befinden sich auf der quadratischen Basisplatine. Da in diesem Projekt SMD-Bauteile verwendet werden, ist es ratsam, einige Erfahrung mit deren Löten zu haben.

Mit einem Lötkolben mit feiner Spitze und dünnem Lötzinn sind die 0805-Widerstände und -Kondensatoren nicht so schwierig zu löten. Es wird empfohlen, sehr dünnes Lötzinn (beispielsweise 0,35 mm Durchmesser) zu verwenden, damit nicht zu viel Zinn auf die Löstellen gelangt. Der Baum wird auf fünf 5 mm dicke weiße Nylon-Unterlegscheiben gesetzt und mit fünf weißen unauffälligen Nylonschrauben befestigt. Das optionale Modul MOD1 wird in einer Ecke platziert, um die Basisplatine so klein wie möglich zu halten. P1 und S1 werden neben dem Modul angebracht.



Bild 7. Die Leitungen der RGB-LED WS2812D-F8 werden knapp oberhalb der Verbreiterung abgeschnitten.

Optionales Modul Arduino Nano ESP32

Damit der Weihnachtsbaum unabhängig von einer externen Schaltung funktioniert, kann das Arduino-Modul (MOD1) zusammen mit dem kleinen Trimpot P1 und dem DIP-Schalter S1 auf die gesetzt werden. Dann gehört der Jumper JP1 auf die Position MOD1. Die beiden zusätzlichen Bauteile können zur Einstellung und/oder Auswahl von Helligkeit/Geschwindigkeit und verschiedenen Mustern/Modi verwendet werden.

Die Programmierung des Moduls erfolgt über die (neuste Version der) Arduino-IDE in der Programmiersprache C. Wenn Sie es einmal mit MicroPython probieren wollen, auch dann ist dieser Weihnachtsbaum eine gute Wahl. Es gibt eine spezielle IDE namens *Arduino Lab for MicroPython* [7], die von GitHub heruntergeladen werden kann. Für diese IDE ist ein MicroPython-Installer [8] erforderlich, den man ebenfalls bei GitHub findet. Das Ganze ist auf der Arduino-Website gut dokumentiert; es gibt dort auch für Einsteiger ausreichend Hinweise, um diese IDE zum Laufen zu bringen [9].

Elektor hat eine Basissoftware entwickelt, die von der Elektor-Labs-Webseite dieses Projekts heruntergeladen werden kann [6]. Dort finden Sie auch einige Details zur Funktionalität.

Während der Software-Entwicklung kann man das Projekt über den USB-Anschluss des PCs mit Strom versorgen, aber nach der Fertigstellung, wenn es ernst wird, benötigen Sie besagtes AC-DC-Steckernetzteil mit UCB-C-Anschluss, um den Baum mit Strom zu versorgen. Die maximal verfügbare Stromstärke hängt natürlich von dem jeweiligen Anschluss ab. Über einen USB-C-auf-USB-A-Adapter oder ein Adapterkabel kann der Arduino Nano auch an einen „alten“ USB-Anschluss angeschlossen werden, solange die Helligkeit niedrig gehalten wird. Der Strom ist dann auf 500 mA begrenzt, was beim Testen der Software zu beachten ist! Man kann auch ein Netzteil an K2 anschließen und die Spannung etwas höher als 5 V einstellen, damit die LEDs mit Strom versorgt werden können. Bild 8 zeigt nicht wirklich die volle, helle Farbe der LEDs, vermittelt aber dennoch einen guten Eindruck davon, was durch verschiedene Einstellungen erreicht werden kann. ↗

RG – 230665-02



Über den Autor

Ton Giesberts begann nach seinem Studium bei Elektuur (jetzt Elektor) zu arbeiten, als wir jemanden mit einer Affinität zu Audio-Elektronik suchten. Im Laufe der Jahre hat er hauptsächlich an Audio-Projekten gearbeitet. Analoges Design war schon immer seine Vorliebe. Natürlich gehören auch Projekte in anderen Bereichen der Elektronik zu seinem Job. Eines der Mottos von Ton ist: „Wenn du es besser haben willst, mach es selbst.“ Zum Beispiel ist bei einem Leiterplattenentwurf für ein Audioprojekt mit Verzerrungswerten in der Größenordnung von 0,001 % ein gutes Layout entscheidend!

Haben Sie Fragen oder Kommentare?

Wenn Sie technische Fragen oder Kommentare zu diesem Artikel haben, senden Sie bitte eine E-Mail an die Elektor-Redaktion unter redaktion@elektor.de.



Passende Produkte

- **Bausatz Zirkularer Weihnachtsbaum**
www.elektor.de/20672
- **Arduino Nano ESP32 mit Stifteleisten**
www.elektor.de/20529



Bild 8. Testaufbau des Weihnachtsbaums.

WEBLINKS

- [1] Weihnachtsbaum Version 1: <https://elektormagazine.de/labs/130478-xmas-tree-2014>
- [2] Weihnachtsbaum Version 2: <https://elektormagazine.de/labs/circular-christmas-tree-150453>
- [3] Dokumentation über den Arduino Nano ESP32: <https://docs.arduino.cc/hardware/nano-esp32>
- [4] Datenblatt (PDF) WS2812D-F8: https://soldered.com/productdata/2021/03/Soldered_WS2812D-F8_datasheet.pdf
- [5] Datenblatt (PDF) Arduino Nano ESP32: <https://docs.arduino.cc/resources/datasheets/ABX00083-datasheet.pdf>
- [6] Dieses Projekt auf Elektor Labs: <https://elektormagazine.de/labs/circular-christmas-tree-2023-230508>
- [7] Arduino Lab for Windows (Zip): <https://tinyurl.com/arduinolab4win>
- [8] Installer für Arduino MicroPython: <https://github.com/arduino/lab-micropython-installer/releases/tag/v1.2.1>
- [9] MicroPython-Kurs MicroPython 101 von Arduino: <https://docs.arduino.cc/micropython-course/>

Für ein einfacheres und komfortableres Leben

Ein Amateurprojekt basierend auf dem ESP8266-Modul von Espressif

Ein Betrag von Transfer Multisort Elektronik Sp. Z.o.o.

Mit jedem Jahr wird das Konzept des SmartHome immer populärer und die Verfügbarkeit von hilfreichen Lösungen, unseren Lebensraum effizienter zu verwalten, wächst. Darüber hinaus sind einige Produkte auf dem Markt, die mit älteren Geräten kompatibel sind, so dass wir diese zusammen mit den neuesten technologischen Errungenschaften nutzen können. Die Fernsteuerung von Haushaltsgeräten und die Automatisierung verschiedener Prozesse helfen, die Energieeffizienz zu verbessern, die Umwelt zu schützen, unseren Komfort zu erhöhen und Geld zu sparen. Das Projekt Smart ESP8266 remote, das für einen Wettbewerb von TechMasterEvent entwickelt wurde, vereint all diese Vorteile.

Espressif ist ein Hersteller integrierter SoC-Schaltungen und drahtloser Übertragungsmodule, von denen viele bei TME erhältlich sind. Dank ihrer kompakten Größe und ihres geringen Energiebedarfs können Espressif-Produkte sowohl in der Konsum- als auch in der Industrielektronik erfolgreich eingesetzt werden.

Im Folgenden sollen Sie ein Gerät auf der Basis des ESP8266-Moduls kennenlernen. Es handelt sich um ein Amateurprojekt, das von einem Teilnehmer eines TechMasterEvent-Wettbewerbs entwickelt wurde. Die Teilnehmer sollten ein Elektronikprojekt entwerfen, das „das Leben einfacher macht“. Sie finden das ESP8266-Modul sowie verschiedene andere Komponenten, die beim Bau Ihrer IoT-Projekte nützlich sein können (Einplatinencomputer, Kommunikations- und Speichermodule, Displays und vieles mehr) unter [1].



TME TECH
MASTER
EVENT

Da die smarte ESP8266-Fernbedienung Signale von herkömmlichen Fernbedienungen lesen und speichern kann, ermöglicht sie die Steuerung älterer Geräte, die möglicherweise nicht mit dem Internet oder anderen Smart-Home-Systemen verbunden werden können. Dies macht sie zu einer kostengünstigen Alternative zur Aufrüstung Ihrer Geräte oder zum Neukauf teurer Smart-Home-Geräte. Die IR-LED und der IR-Empfänger werden zum Senden beziehungsweise Empfangen von IR-Signalen verwendet, die die Haushaltsgeräte steuern. Das Projekt kann Signale von herkömmlichen Fernbedienungen lesen und speichern, so dass der Benutzer ältere Geräte steuern kann, die möglicherweise nicht die Fähigkeit haben, sich mit dem Internet oder anderen Smart-Home-Systemen zu verbinden. Zusätzlich zu den Hardwarekomponenten benötigt das Smart-ESP8266-Fernbedienungsprojekt auch Software, die Sie unter [2] finden.

Die smarte ESP8266-Fernbedienung bietet mehrere Vorteile wie Bequemlichkeit und Benutzerfreundlichkeit, Kosteneffizienz und Flexibilität. Sie macht den Kauf teurer Smart-Home-Geräte oder die Aufrüstung älterer Geräte überflüssig und ist damit eine kostengünstige Alternative. Das Projekt ist außerdem so flexibel, dass es leicht modifiziert oder angepasst werden kann, um mit verschiedenen Geräten und unterschiedlichen IR-Protokollen zu arbeiten, was es zu einer vielseitigen Lösung für die Steuerung verschiedener Geräte mit einer einzigen App macht. ↗

RG - 230656-02

WEBLINKS

- [1] TME-Shop: <http://www.tme.eu/de/>
- [2] Quellcode für dieses Projekt:
<https://t1p.de/9hlzn>

Wie man IoT-Apps ohne Software-Expertise erstellt

Mit der Blynk IoT-Plattform und Espressif-Hardware

Ein Beitrag von Blynk Inc.

Stellen Sie sich vor, Sie könnten eine mobile App entwickeln, ohne eine Zeile Code zu schreiben, sie individuell gestalten und innerhalb eines Monats in den App-Stores veröffentlichen. Eine professionelle IoT-Software ohne Software-Ingenieure starten? Mit Blynk IoT ist dies innerhalb eines Monats und nicht in Jahren möglich!

Die Blynk-Firmware-Bibliothek unterstützt:
 • ESP32
 • ESP32-S2
 • ESP32-S3
 • ESP32-C3
 • ESP8266
 • und Andere

Was ist in der Blynk IoT-Plattform enthalten?

Blynk ist eine Low-Code-IoT-Softwareplattform mit Cloud, Firmware-Bibliotheken, einem nativen mobilen No-Code-App-Builder und einer Webkonsole zur Verwaltung. Sie erhalten WLAN-Gerätebereitstellung, Datenvisualisierung, Automatisierung,

Benachrichtigungen, OTA-Updates und ein robustes Benutzer- und Gerätemanagementsystem als integrierte Standardfunktionen [1].

App-Builder von Blynk für iOS und Android

Der App-Builder ermöglicht die Erstellung von schnellen Prototypen und voll funktionsfähigen Apps ohne Programmierkenntnisse. Im Konstruktionsmodus können Sie aus über 50 anpassbaren UI-Elementen wählen und diese per Drag-and-Drop auf die Arbeitsfläche ziehen, um eine benutzerdefinierte Benutzeroberfläche für Ihr vernetztes Produkt zu erstellen.

Web-Dashboard-Builder

Es bietet eine ähnliche Architektur zur Erstellung von historischen und Echtzeit-Datenvisualisierungen sowie zur Steuerung und Überwachung von Geräten mit vorgefertigten UI-Elementen.



Bild 1. No-Code-Schnittstellen erstellt mit Blynk.

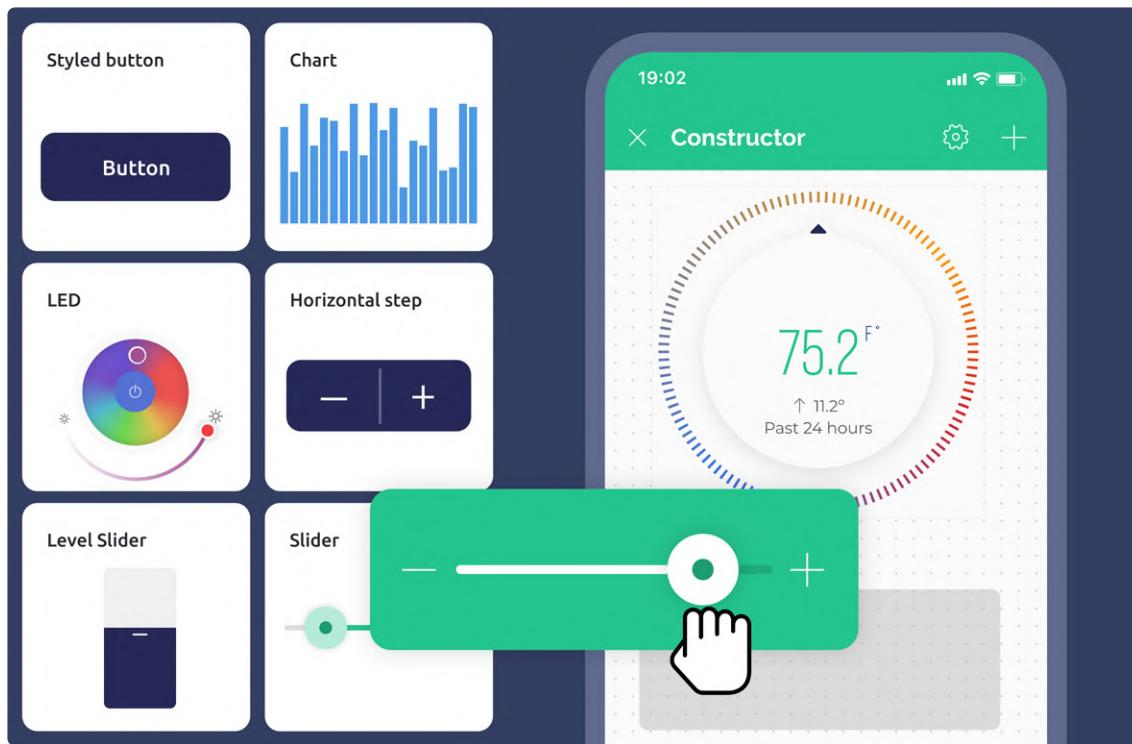


Bild 2. Blynk Drag-n-Drop App Builder.

Erweitertes Benutzermanagementsystem

Es hält alles strukturiert, auch im Unternehmensmaßstab. Sie können eine mehrstufige Organisationsstruktur erstellen und Geräte sowie Benutzerrollen und -berechtigungen verwalten.

Integriertes Geräte-Lebenszyklusmanagement

Diese Funktion deckt alle Anforderungen im Zusammenhang mit Token-Management und WLAN-Bereitstellung ab. Sie bietet zuverlässige und sichere OTA-Firmware-Updates über eine einfache Schnittstelle.

No-Code-Automatisierungsszenarien

Diese können basierend auf Datum, Tageszeit, Benutzeraktionen oder Gerätezustand eingestellt werden. Sie können Benutzer über wichtige Ereignisse auf der Hardware über Pushes, In-Apps, E-Mails oder SMS informieren.

Wie verbinden Sie Ihr ESP mit Blynk? Wie groß ist der Integrationsaufwand?

Je nach Ihrer Hardware-Konfiguration können Sie sich für Blynk.Edgent [3] bei einer Single-MCU-Konfigura-

tion oder für Blynk.NCP [4][5] bei einer Dual-MCU-Konfiguration entscheiden. Beide Wege verfügen über alle integrierten Blynk-IoT-Funktionen und eine gesicherte Verbindung zur Blynk Cloud.

Mit den von Blynk bereitgestellten Code-Beispielen ist der Aufwand minimal. Bei der Dual-MCU-Konfiguration verwenden Sie eine fertiges Binärdatei für den NCP und eine leichte Bibliothek für den primären MCU, der über die UART-Schnittstelle mit dem NCP kommuniziert.

Ihre Reise von der Gerätekonfiguration bis zur vollständigen IoT-Infrastruktur und App-Einführung kann nur wenige Wochen dauern [6]. ◀

230659-02

**Erhalten Sie 30% Rabatt
auf den Blynk PRO Tarif
für das erste Jahr!**

Aktionscode: **ELEKTOR**

Gültig bis zum 31. Januar 2024 [2]

WEBLINKS

- [1] Offizielle Webseite: <https://bit.ly/blynk-io>
- [2] Blynk.Console - Erstellen Sie Ihr kostenloses Konto: <https://bit.ly/web-cloud>
- [3] Blynk.Edgent Dokumentation: <https://bit.ly/docs-edgent>
- [4] Blynk.NCP Dokumentation: <https://bit.ly/docs-ncp>
- [5] Was ist Blynk.NCP: <https://bit.ly/info-ncp>
- [6] Fertiges Wetterstationsprojekt zum Ausprobieren: <https://bit.ly/blueprint-weather>

Erstellung einer intelligenten Benutzeroberfläche auf ESP32

Ein Beitrag von Slint

Smartphones haben das Nutzererlebnis von touch-basierten Steuerungen neu definiert. Die Erstellung einer intelligenten Benutzeroberfläche erfordert den Einsatz moderner Werkzeuge. In diesem Artikel teilen wir Tipps und präsentieren Slint, ein Toolkit für interaktive Benutzeroberflächen, die Nutzererwartungen erfüllen und übertreffen.

Über Slint - Ein Toolkit der nächsten Generation zur Erstellung nativer, grafischer Oberflächen in C++, Rust und JavaScript mit einer breiten plattformübergreifender Unterstützung wie Bare-Metal, RTOSs und Linux und 10 K+ GitHub-Sternen.

Wählen Sie Ihre Programmiersprache – C/C++ oder Rust

In der Embedded-Programmierung sind C/C++ schon lange die

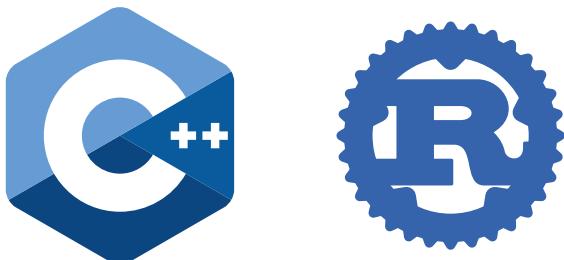


Bild 1. Logos von C++ und Rust.

beliebtesten Programmiersprachen. Aber Rust, bekannt für seine Speichersicherheit und Leistung, wird bei Embedded-Entwicklern immer beliebter.

Slint, das einzige Toolkit mit nativen APIs für C++ und Rust (Bild 1), bietet Entwicklern die Wahl: Schreiben der Geschäftslogik in einer der beiden Sprachen. Darüber hinaus bietet es einen Übergangspfad für diejenigen, die ihren Code von C/C++ auf Rust umstellen möchten.

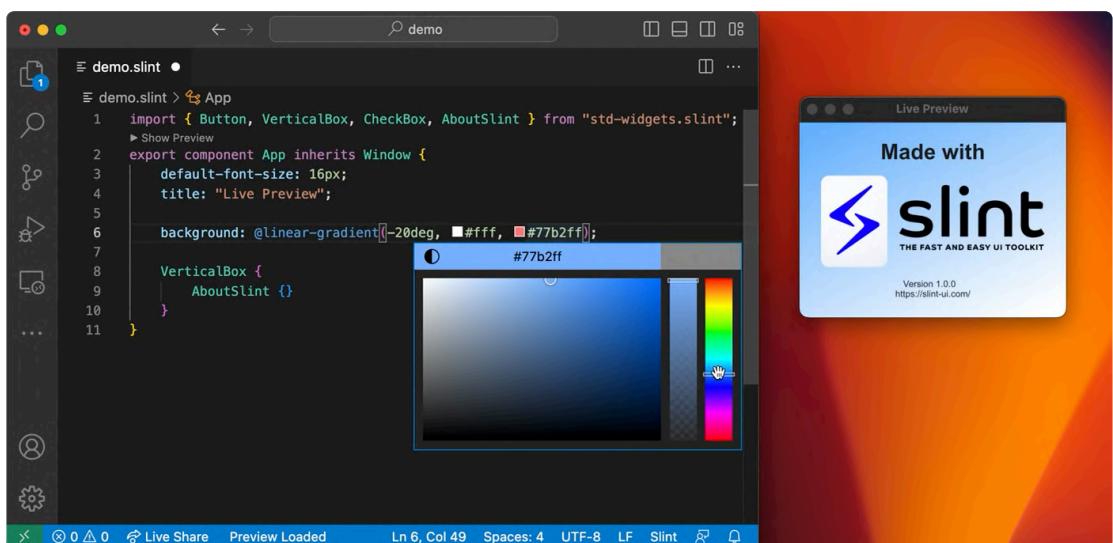


Bild 2. Schnelle Iteration mit der Live-Preview von Slint.

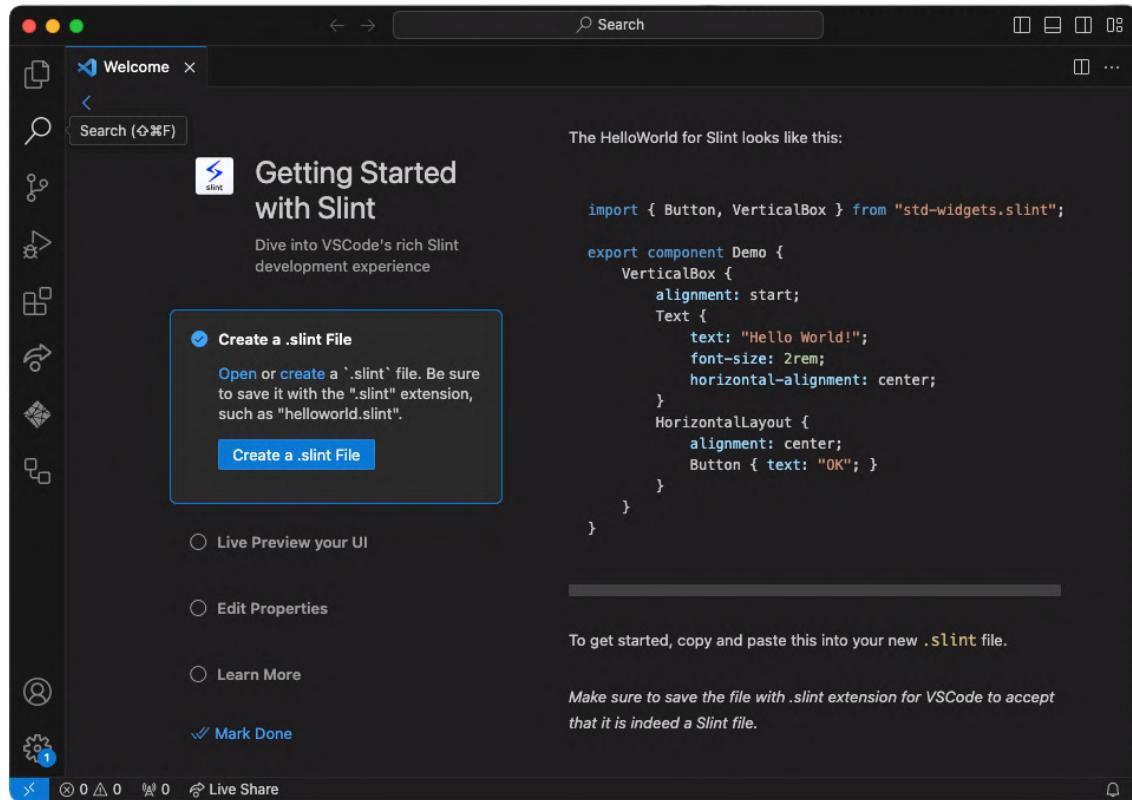


Bild 3.
Starten Sie mit Slint.

Trennen Sie das Benutzerinterface von der Geschäftslogik

MVC- und MVVM-Entwurfsmustern fordern die Trennung von Benutzerinterface und Geschäftslogik, um die Effizienz und Codequalität zu steigern.

In Slint ist die Benutzeroberfläche in einer HTML/CSS-ähnlichen Sprache definiert, die eine klare Trennung zwischen Präsentation und Geschäftslogik gewährleistet. Die Live-Preview von Slint ermöglicht die Verfeinerung des UI-Designs durch schnelle Iterationen (**Bild 2**).

Genießen Sie eine gute Entwicklererfahrung

Die heutige Komplexität in der Softwareentwicklung erfordert eine gute Entwicklererfahrung: Entwickler können mit Zuversicht arbeiten, eine größere Wirkung erzielen und sich zufrieden fühlen. Verwenden Sie weiterhin Ihre Lieblings-IDE. Wählen Sie zwischen Slint's VSCode-Extension (**Bild 3**) und dem allgemeinen Sprachserver (LSP): Nutzen Sie Codevervollständigung, Syntaxhervorhebung, Fehlerdiagnose, Live-Preview und mehr. Slint bietet eine ESP-IDF-Komponente, die die Integration mit Espressif's IoT-Development-Framework (IDF) vereinfacht.

Bieten Sie ein außergewöhnliches Nutzererlebnis

Die Leistung der Benutzeroberfläche ist entscheidend für ein außergewöhnliches Nutzererlebnis. Genießen Sie die Flexibilität im Hardware-Design mit Slints Fähigkeiten zur zeilenweisen und Framebuffer-Darstellung auf der ESP32-Plattform, um einen vielseitigen Ansatz für die Geräteentwicklung sicherzustellen (**Bild 4**).

Beginnen Sie mit Slint auf ESP32, besuchen Sie [1].



RG – 230670-02



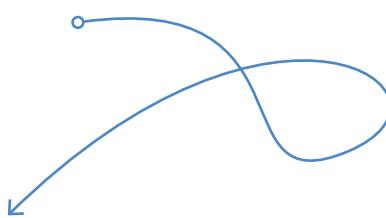
Bild 4. Eine Slint-Demo auf ESP32.

WEBLINK

[1] Slint auf dem ESP32: <https://slint.dev/esp32>

Schnelle und einfache IoT-Entwicklung mit M5Stack

Ein Beitrag von M5Stack



M5STACK, der weltbekannte Anbieter der modularen, auf ESP32 basierenden IoT-Entwicklungsplattform hat Hunderte von Controllern, Sensoren, Aktoren und Kommunikationsmodulen in modularer Bauweise im Angebot, die sich über Standardschnittstellen verbinden lassen. Durch das Kombinieren von Modulen mit unterschiedlichen Funktionen können Anwender die Produktprüfung und -entwicklung beschleunigen.

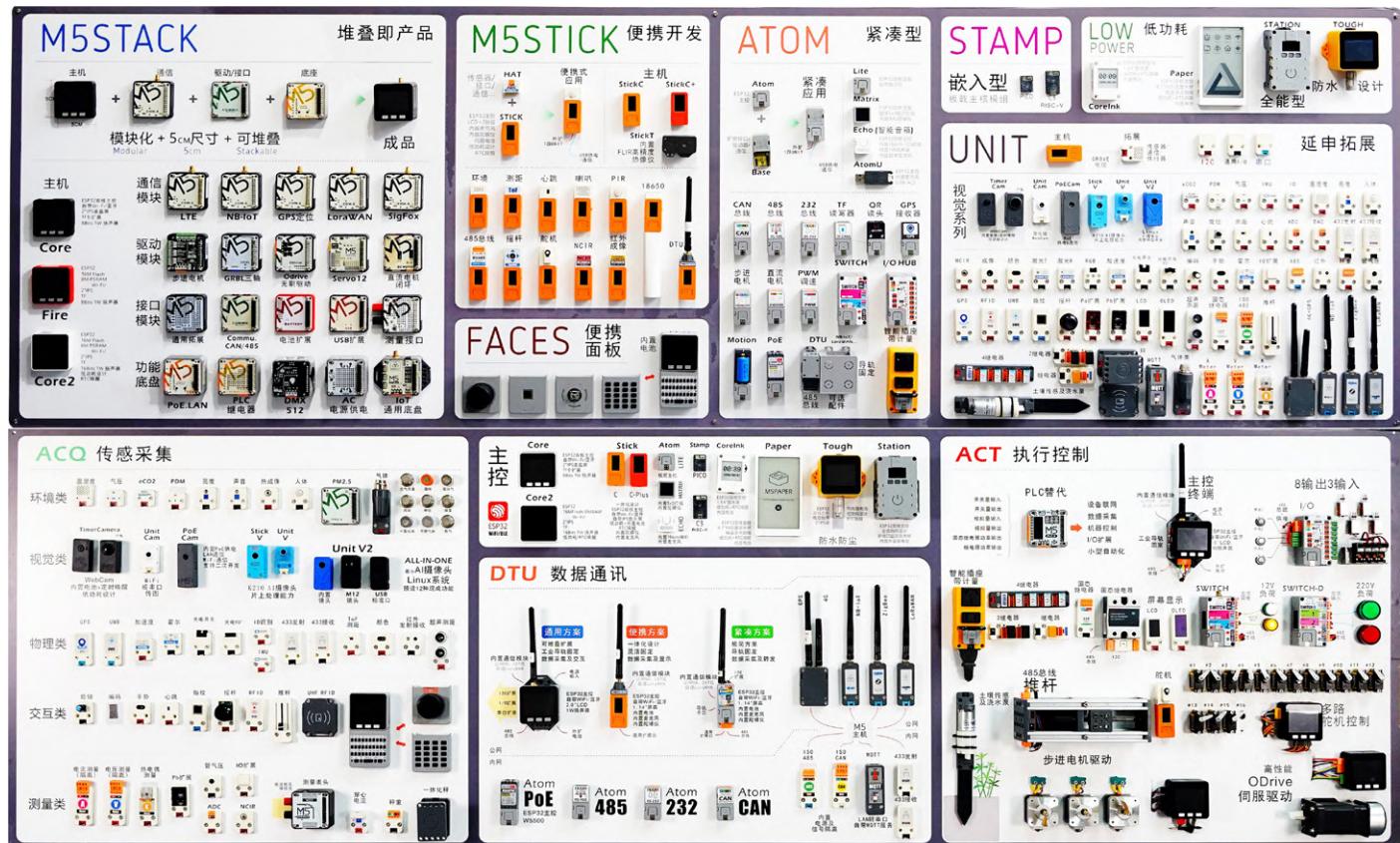


Bild 1. M5Stack-Ökosystem-Familie.



Bild 2. M5Dial ist für das Smart Home geeignet.



Die M5Stack-Module (**Bild 1**) [1] lassen sich mit der grafischen Low-Code-Programmier-IDE UIFlow „zusammenstecken“, so dass Sie damit experimentieren können, um das beste Erlebnis für das Prototyping von IoT-Projekten zu erhalten. Das gilt sowohl für Einsteiger als auch für professionelle Entwickler. Mit stapelbaren Hardware-Modulen und einer benutzerfreundlichen grafischen Programmierplattform bietet M5Stack Kunden aus den Bereichen Industrielles IoT, Gebäudeautomatisierung, Smarter Einzelhandel, Smarte Landwirtschaft und MINT eine effiziente sowie zuverlässige IoT-Entwicklungserfahrung, die schnell und einfach ist.

Neu: Der M5Dial

Das kürzlich vorgestellte M5Dial [2] eignet sich hervorragend für das Smart Home. Als vielseitiges Embedded-Entwicklungsboard vereint das M5Dial verschiedene Funktionen und Sensoren, die für die Smart-Home-Steuering notwendig sind (**Bild 2**).

Sind Sie ein Fan von ESP32, dann ist M5Stack ein Muss!

Der Hauptcontroller des M5Dial ist der M5StampS3, ein Mikrocontroller, der auf dem ESP32-S3-Chip basiert. Er ist für seine hohe Leistung und seinen geringen Stromverbrauch bekannt. Das Gerät unterstützt WLAN- und Bluetooth-Verbindungen sowie mehrere Peripherieschnittstellen wie SPI, I²C, UART, ADC und mehr. Der M5StampS3 hat zudem 8 MByte integrierten Flash-Speicher, der ausreichend Platz für Nutzer bietet. Der M5Dial verfügt über einen runden 1,28-Zoll-TFT-Touchscreen, einen Drehknopf, ein RFID-Erkennungsmodul, eine RTC-Schaltung, einen Summer, physische Tasten sowie andere Funktionen, womit Benutzer verschiedene Projekte leicht umsetzen können.

Das herausragende Merkmal von M5Dial ist der Drehknopf, der die Position und Richtung des Knopfes genau aufzeichnet und den Nutzern ein verbessertes interaktives Erlebnis bietet. Mit dem Drehknopf können Nutzer Einstellungen wie Lautstärke, Helligkeit und Menüs anpassen oder Haushaltsgeräte wie Licht, Klimaanlage, Vorhänge und so weiter steuern. Auf dem eingebauten Bildschirm des Geräts lassen sich zudem verschiedene interaktive Farben und Effekte darstellen. Dank seiner kompakten Größe von 45 mm × 45 mm × 32,2 mm und dem geringen Gewicht von 46,6 g lässt sich M5Dial einfach benutzen.

Ob zur Steuerung von Haushaltsgeräten im Smart Home oder zur Überwachung und Steuerung von Systemen in der Industrieautomation, M5Dial lässt sich einfach integrieren und bietet intelligente Steuerungs- sowie interaktive Funktionen. 

230662-02

WEBLINKS

[1] Der Innovator der modularen IoT-Entwicklungsplattform | M5Stack: <https://m5stack.com>

[2] Intelligenter Drehknopf ESP32-S3 mit rundem 1,28"-Touchscreen:

<https://shop.m5stack.com/products/m5stack-dial-esp32-s3-smart-rotary-knob-w-1-28-round-touch-screen>

Prototyping eines Energiezählers mit ESP32



Bild 1. Test-Rendering, wie unser Energiezähler aussehen könnte.

In der Elektronik kann die richtige Kombination der richtigen Technologien zu bedeutenden Fortschritten führen. Dieses Projekt zielt darauf ab, einen Energiezähler zu entwickeln, der den ESP32-Mikrocontroller von Espressif und das Energiemess-IC ATM90E32AS von Microchip verwendet. Dieser Artikel beschreibt die Auswahl der Komponenten bis hin zum ersten Prototyping. Das Ziel ist einfach: ein zuverlässiges System für eine genaue Energieerfassung im heimischen Zählertkasten oder in der Werkstatt zu schaffen. Mit diesem Messgerät können die Benutzer ihren Stromverbrauch in Echtzeit verfolgen und so Erkenntnisse gewinnen, die zu einer effizienteren Energienutzung führen können.

Design und Anforderungen

Das Projekt hat klare Ziele und Entwicklungsanforderungen: Echtzeit-Überwachung des einphasigen Stroms mit drei Stromwandlern (CTs), Bezahlbarkeit und Benutzerfreundlichkeit. Die Wahl der Hauptbestandteile ESP32 und ATM90E32AS orientierte sich an diesen Zielen und bot sowohl Kosteneffizienz als auch zuverlässige Leistung. Ein weiteres Ziel war es, die Abmessungen unter 100×80×30 mm (L×B×H) zu halten, damit das Gerät in einem Schaltschrank untergebracht werden kann. Um die Benutzerfreundlichkeit zu erhöhen, ist auch eine mobile Schnittstelle für die Fernüberwachung sowie ein OLED-Display mit Tasten für die direkte Interaktion vorgesehen. Das Design ermöglicht auch künftige Software-Updates, so dass ein langfristiger Nutzen für den Verbraucher gewährleistet ist. In Bild 1 ist ein (gerendertes) Bild des aktuellen Gehäuses des Prototyps dargestellt.

Auswahl des Mikrocontrollers

Die Wahl des ESP32-Mikrocontrollers beruhte auf einer detaillierten Analyse seiner Fähigkeiten. Der Chip zeichnet sich in mehreren Punkten aus, die für den Erfolg dieses Projekts von Belang waren. Erstens lässt

Saad Imtiaz (Elektor)

Dieser Artikel beschreibt die Entwicklung eines Energiezählers mit einem ESP32 von Espressif, wobei die Überwachung der Stromaufnahme in Echtzeit und die Sicherheit im Vordergrund stehen. Es werden die ersten Schritte, die Anforderungen und Überlegungen beleuchtet, die beim Start eines Embedded-Projekts anfallen. Mit dem Fortschreiten des Projekts werden wir in den nächsten Elektor-Ausgaben über die Ergebnisse berichten.

er sich leicht in verschiedene Schaltungsentwürfe integrieren und bietet dadurch Flexibilität in der Entwicklungsphase. Zweitens macht ihn seine Kosteneffizienz zu einer attraktiven Wahl für einen Prototyp, der ein Gleichgewicht zwischen Leistung und Kosten anstrebt. Drittens bietet die Kompatibilität mit einer breiten Palette von Sensoren und ICs erhebliche Vorteile. Und schließlich trägt die umfangreiche Unterstützung der Community für den ESP32-Chip zu seiner Eignung für dieses Projekt bei. Bild 2 hebt die wichtigsten Merkmale und Vorteile des ESP32-D0WD-V3 hervor, die dazu führten, dass er für dieses Projekt ausgewählt wurde.

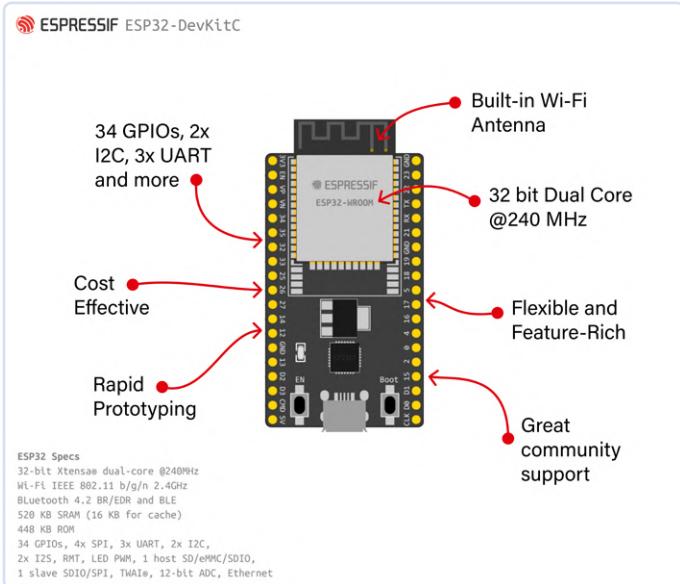


Bild 2. Hauptmerkmale und Vorteile des ESP32.

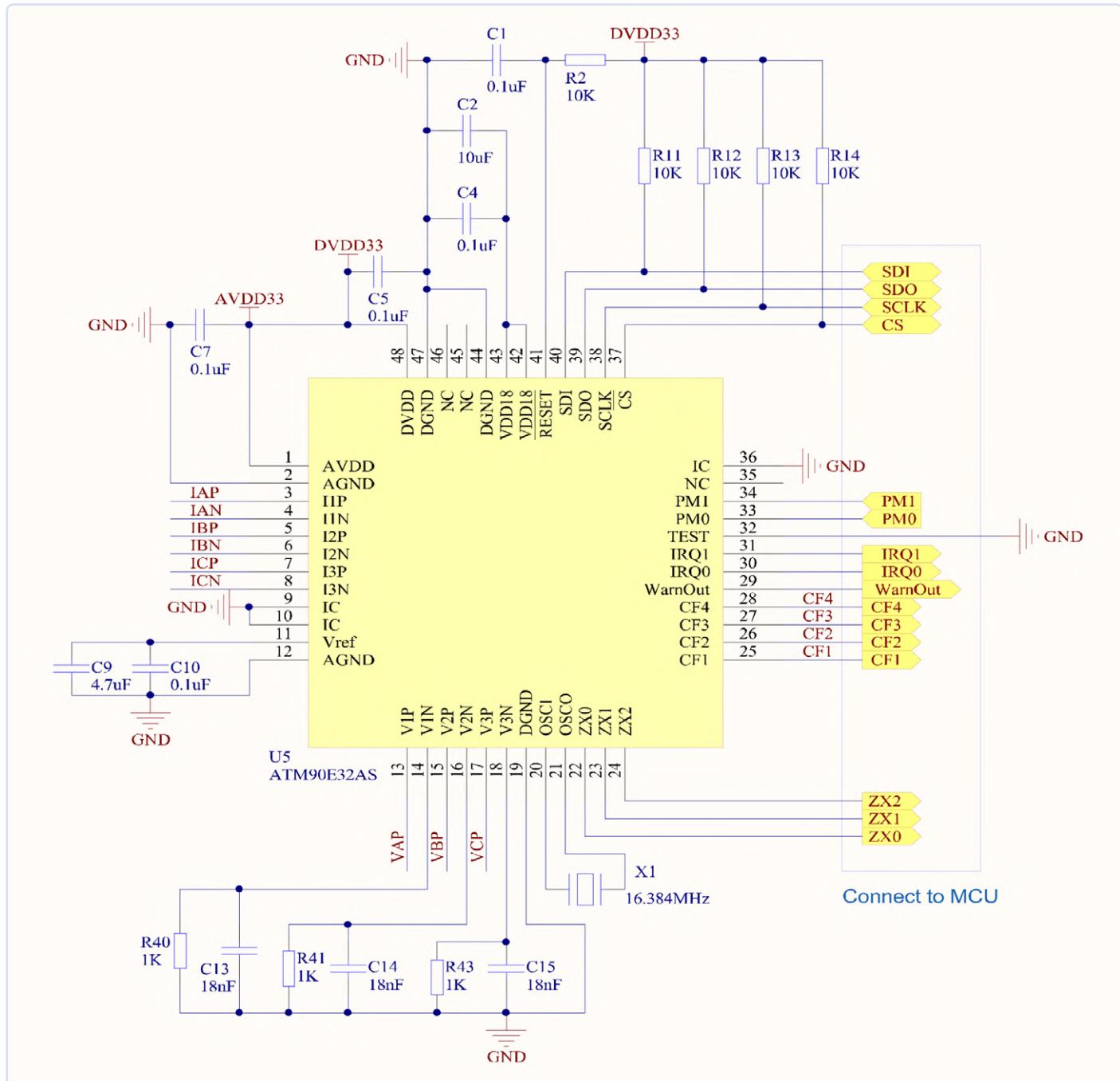


Bild 3. Der Energiezähler basiert auf einer Application Note von Atmel [2]. Hier sieht man die Schaltung rund um das Zähler-IC.

Integration des Zähler-ICs

Der ATM90E32AS-IC von Microchip wurde gemäß den Anwendungshinweisen [2] des Herstellers verwendet. Das Dokument diente als Eckpfeiler, damit das Energiemess-IC nahtlos mit dem ESP32-Mikrocontroller kommunizierte konnte. Dennoch war diese Phase nicht frei von Problemen und Herausforderungen. Die Beschaffung der richtigen Bauteile (unter Einhaltung des Budgetrahmens) erforderte angesichts der eingeschränkten Verfügbarkeit eine akribische Planung. In Bild 3 ist die von Atmel (jetzt Microchip) zur Verfügung gestellte Application Note dargestellt.

Entwurfsphase und elektrische Sicherheitsstandards

Die Entwurfsphase ist in der Tat ein entscheidender Teil des Entwicklungsprozesses, vor allem wenn die Sicherheit ein unverzichtbarer Aspekt ist. Bei einem Gerät, das für die Interaktion mit Netzwechsel-

spannungen ausgelegt ist, muss die Einhaltung etablierter Sicherheitsnormen genauestens beachtet werden. In Bild 4 ist das Blockdiagramm des Projekts dargestellt.

Um die elektrische Sicherheit zu gewährleisten, wurden mehrere spezielle Bauteile in das Design aufgenommen: Metalloxid-Varistoren (MOVs) wurden zur Unterdrückung transienter Spannungen eingesetzt, um die Schaltung vor Spannungsspitzen zu schützen, wesentliche Sicherungsbauteile sollen Überstrombedingungen verhindern.

Neben der Bauteile-Auswahl wurden bei der Schaltungsentwicklung auch Überlegungen zum Layout angestellt, um die Sicherheitsnormen einzuhalten. Zwischen den leitenden Elementen auf der Platine wurden angemessene Kriech- und Luftstrecken eingehalten, um Lichtbögen und ähnliche unliebsame Erscheinungen zu verhindern. Die Leiterbahnbreiten für die Wechselspannungsverbindungen wurden sorgfältig berechnet, damit sie die Nennströme bewältigen, wobei die IPC-2221-Normen [1] beachtet wurden. Dies war entscheidend,

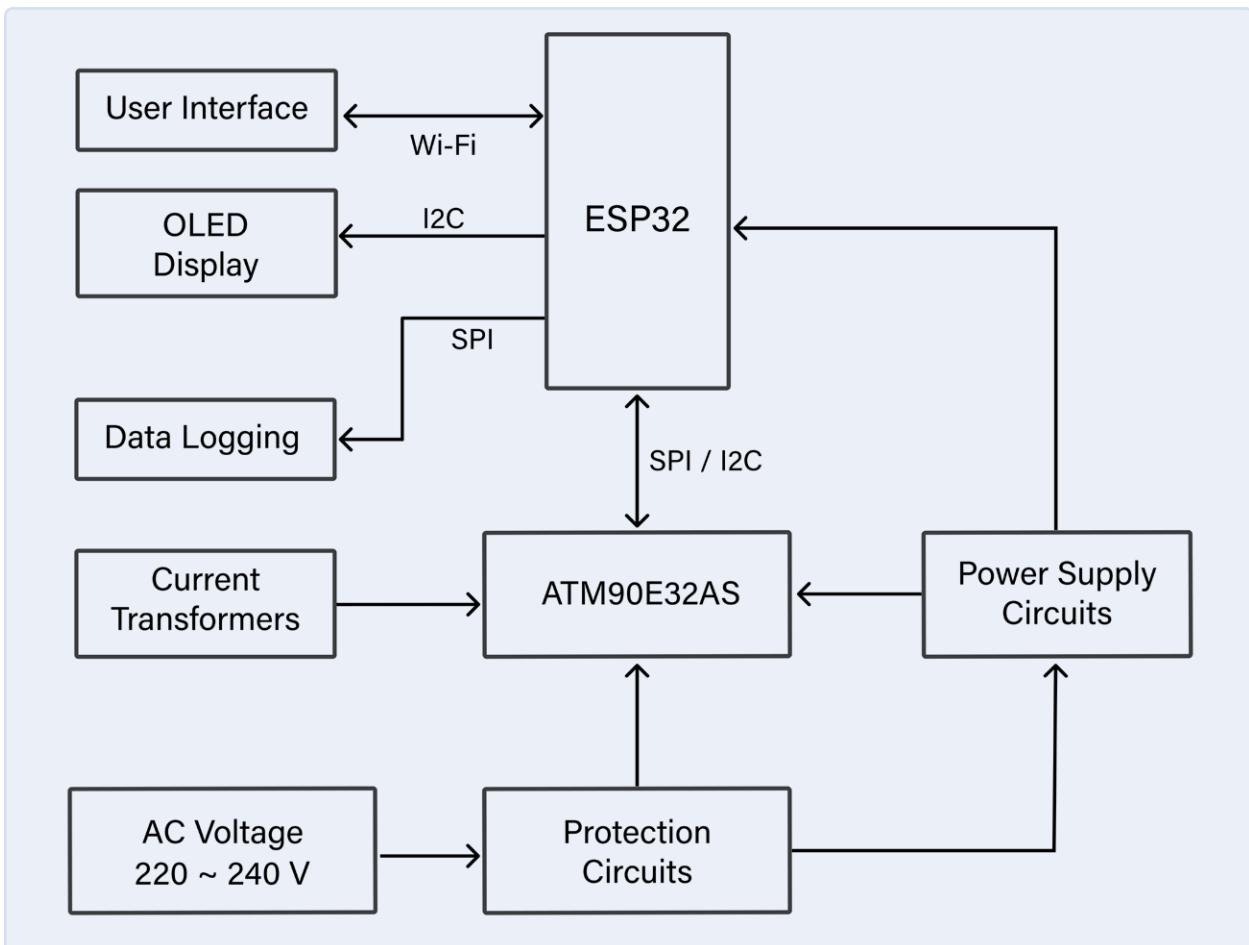


Bild 4. Blockschaltbild unseres Energiezählerprojekts.

um die thermische Leistung der Platine unter Volllastbedingungen zu gewährleisten. Für eine saubere Masseverbindung wurde eine solide Massefläche geplant. Besonderes Augenmerk wurde auf die Ausgestaltung von Differenzialpaaren gelegt, um die Signalintegrität zu gewährleisten, wobei darauf geachtet wurde, dass das Routing einer präzisen Geometrie folgt, um elektromagnetische Störungen zu minimieren.

Die Entscheidung für JLC PCB

Nach Prüfung verschiedener Platinen-Services und -Bestückungsdienste fiel die Wahl auf JLC PCB. Der Hauptgrund für diese Wahl war die Ausgewogenheit von Kosteneffizienz und Zuverlässigkeit, die das Unternehmen bietet. Diese Entscheidung war wichtig, um das Projekt im Rahmen unseres Budgets zu halten, ohne Kompromisse bei der Qualität der bestückten Platine eingehen zu müssen. Derzeit werden der Schaltplan des Prototyps und die Leiterplattenentwürfe fertiggestellt, so dass die Prototypen bald in die Produktion geschickt werden können.

Ein Blick zurück und einer nach vorne

Rückblickend zeigt dieses Projekt, was erreicht werden kann, wenn sorgfältige Planung auf gute Technik trifft. Die Hürden, mit denen wir konfrontiert wurden, haben uns geholfen, unseren Entwurf zu verbessern. Wir gehen davon aus, dass wir von der Herstellung eines Prototyps zu einer möglichen Massenproduktion übergehen können, und vor allem, dass das Gerät einen echten Unterschied im Umgang der Menschen mit ihrem Energiebedarf bewirken wird. Über den Fortgang dieses Projekts werden wir in den nächsten Elektor-Ausgaben ausführlich berichten - wir sind ja noch dabei, den Prototyp herzustellen, zu

testen und an der Software zu basteln, mit der er betrieben werden soll. Es wird also noch mehr kommen, bleiben Sie dran! Wir werden in der kommenden Januar/Februar-Ausgabe 2024 von Elektor, die dem Thema Strom und Energie gewidmet ist, über die Fertigstellung berichten. ↗

RG – 230646-02

Haben Sie Fragen oder Kommentare?

Wenn Sie technische Fragen zu diesem Artikel haben, wenden Sie sich bitte an die Elektor-Redaktion unter redaktion@elektor.de.



Passende Produkte

➤ **ESP32-DevKitC-32E**
www.elektor.de/20518

➤ **ESP32-C3-DevKitM-1**
www.elektor.de/20324



WEBLINKS

- [1] IPC-2221A-Standards: <https://t1p.de/wjwop>
- [2] Application Note ATM90E32AS : <https://t1p.de/i5axy>

Ein Distributor für IoT und mehr - mit Mehrwert

Ein Beitrag von Steliau Technology

Steliau Technology ist ein innovatives Unternehmen, das sich auf elektronische Lösungen spezialisiert hat. Das Unternehmen zeichnet sich durch sein Fachwissen im Ingenieurwesen und seine Leidenschaft für Innovation aus. Steliau Technology bietet über seinen Partner Espressif wichtige elektronische Komponenten für drahtlose Konnektivität und IoT an, wie z.B. ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6 und viele weitere Produkte.

Steliau Technology [1] wurde 2018 gegründet und versteht sich als Mehrwertvermarkter von elektronischen Lösungen. Mensch-Maschine-Schnittstellen, Touchscreens und -Lösungen, Konnektivität und IoT - all diese Fachgebiete werden von Steliau weitgehend beherrscht und verleihen dem Unternehmen einen guten Ruf auf dem Elektronikmarkt.

Steliau Technology ist für seine strategischen Partnerschaften mit führenden Elektronikunternehmen bekannt, durch die es seine Position in den Bereichen IoT und Konnektivität stärken kann.

Espressif und Steliau Technology pflegen eine historische Partnerschaft und als erster Vertriebspartner eine über Jahre hinweg gefestigte Beziehung. Als offizieller Vertriebspartner für Frankreich und Italien ist Steliau Technology die zentrale Anlaufstelle für Espressif-Lösungen.

Steliau Technology ist bestens gerüstet, um einen umfassenden Support für die gesamte Espressif-Produktpalette anzubieten, sowohl was die Hardware als auch eingebettete Software betrifft. Das Team

verfügt über ein umfassendes technisches Fachwissen, das alle Aspekte der Konnektivität abdeckt, vom Hardwaredesign bis zur Softwareprogrammierung. Das bedeutet, dass Steliau Technology in der Lage ist, den Kunden umfassende Unterstützung zu bieten und so robuste und leistungsfähige Konnektivitätslösungen zu gewährleisten. Diese starke Partnerschaft garantiert unseren Kunden einen privilegierten Zugang zu den besten Konnektivitätslösungen auf dem Markt, zum Beispiel die neuesten Espressif-Generationen: ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6. Steliau Technology liefert über seinen Partner Espressif elektronische Komponenten, die für die drahtlose Vernetzung und das IoT in einer Vielzahl von Märkten und Branchen unerlässlich sind, und trägt so zur ständigen Weiterentwicklung der Technologie bei.

IoT-Lösungen und mehr

Im Bereich des Internet of Things (IoT) werden oft WLAN- und Bluetooth-Mikrocontroller von Espressif eingesetzt.

WEBLINK

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



Diese Komponenten sind entscheidend für Smart-Home-Anwendungen wie vernetzte Thermostate, Sicherheitskameras und Lichtsteuerungsgeräte. Sie spielen eine wichtige Rolle bei der Interoperabilität von vernetzten Produkten im Haus mit kompatiblen MATTER-Lösungen (insbesondere über Wi-Fi und Thread). Die Lösungen von Espressif werden auch in der Industrie zur Fernüberwachung, Datenerfassung und Steuerung von Maschinen eingesetzt. Darüber hinaus sind die Produkte von Espressif im Gesundheitssektor vertreten, wo sie vernetzte medizinische Geräte und Geräte zur Überwachung der körperlichen Fitness mit Strom versorgen.

Als globaler Elektronikpartner kann Steliau Lösungen anbieten, die die Produkte von Espressif für die Steuerung von Touchscreens integrieren. Steliau Technology ist in der Lage, integrierte Lösungen zu entwickeln, bei denen die Espressif-Produkte den Touchscreen steuern, und hat bereits mehrere Erfolge erzielt, insbesondere bei Bildschirmgrößen von 7 Zoll. Unsere Fähigkeit, eine umfassende Lösung anzubieten, wird durch einen speziellen technischen Support verstärkt, der all diese Bereiche abdeckt. ▶

230661-02

Haben Sie Fragen?

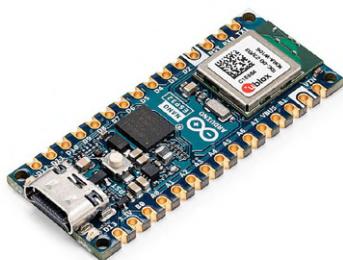
Steliau steht für alle Anfragen zur Verfügung. Bitte wenden Sie sich für Anfragen zu Espressif-Lösungen an remi.krief@steliau-technology.com.

Tiefe Einblicke: Ein Interview mit Arduino über den Nano ESP32

Alessandro Ranellucci und Martino Facchin
diskutieren über die Zusammenarbeit mit Espressif

Fragen von Saad Imtiaz und
Clemens Valens (Elektor)

Im Sommer 2023 brachte Arduino den Nano ESP32 auf den Markt. Das neue Board basiert auf dem ESP32-S3 von Espressif und bietet 2,4 GHz, WLAN nach 802.11 b/g/n und Bluetooth 5 (LE) mit großer Reichweite in einem Nano-Formfaktor. Der Nano ESP32 ist nicht das erste Arduino-Board mit einem Prozessor von Espressif, aber hier bildet er das Herzstück und nicht nur ein Drahtlos-Kommunikationsmodul, das eine andere MCU unterstützt. Elektor befragte Alessandro Ranellucci (Leiter der Arduino Makers Division) und Martino Facchin (Hardware und Firmware Manager, Arduino) zu dieser Zusammenarbeit zwischen Espressif und Arduino.



Elektor: Können Sie erklären, warum Sie anstelle eines anderen Mikrocontrollers den ESP32 für die neue Arduino Nano-Reihe gewählt haben?

Alessandro Ranellucci: Die Arduino-Nano-Reihe ist eine Gruppe von Boards, die in Bezug auf Form, Pin-Layout und technische Kernfunktionen einheitlich gestaltet sind. Diese Konsistenz ist etwas, das unsere Maker-Community zu schätzen weiß, denn sie ermöglicht eine nahtlose Austauschbarkeit innerhalb der Nano-Reihe. Allerdings hatten wir bisher noch kein Board auf Basis der beliebten ESP32-Architektur vorgestellt. Wir hielten es für notwendig, eine leistungsstarke Option innerhalb der Nano-Reihe anzubieten, die den ESP32 nutzt – daher die Entscheidung.

Martino Facchin: Ja, in den letzten vier Jahren haben wir die Nano-Reihe durch die Integration von Mikrocontrollern verschiedener Hersteller erweitert, mit denen wir zuvor nicht zusammen-gearbeitet hatten. Wir begannen mit der Integration von Mikrocontrollern von Nordic, gefolgt von Raspberry Pi. Im Wesentlichen handelt es sich um unsere Testplattform, auf der wir sowohl neue als auch bewährte Architekturen ausprobieren und gleichzeitig einen einzigartigen Mehrwert bieten, der nicht anderswo zu bekommen ist. Auf dem ESP32-Bare-Chip gab es zum Beispiel keinen USB-Anschluss. Beim C3 gibt es zwar USB, aber mit einer festen Hersteller- und Produkt-ID, so dass man nicht wirklich zwischen den einzelnen Boards unterscheiden konnte. Der S3 ist der erste, der diese Möglichkeit bietet, also haben wir statt irgendwelcher Chips aus vorherigen Produktionsreihen ihn verwendet.

Elektor: Welche einzigartigen technischen Vorteile hat der ESP32 gegenüber anderen Optionen?

Martino Facchin: Ja, wir hatten die Möglichkeit, direkt mit u-blox zusammenzuarbeiten, um einen speziellen Chip zu erwerben, der nicht allgemein erhältlich ist und PSRAM im Chip selbst integriert hat. Wir bieten einen Chip an, der PSRAM und die größtmögliche Menge an externem Flash-Speicher enthält. Diese Details sind sozusagen der technische Tieftauchgang, zugeschnitten auf die Enthusiasten und Geeks. Die Funktionen des Boards treiben den ESP32 zu seinem höchsten derzeit verfügbaren Potenzial.