ESP32 to go

Originalartikel

Backup

<html> <p class=„printversionback-to-article printversion–hide“>zurück zum Artikel</p><figure class=„printversionlogo“><svg preserveaspectratio=„xMinYMin“ xmlns=„http://www.w3.org/2000/svg“ viewbox=„0 0 600 85“ width=„360“ height=„51“><path d=„M230.68,63.64V59.82h6V20.65h-6V16.9h24c8.4,0,14.86,2,19.28,6s6.68,9.75,6.68,17.33S278.4,53.59,274,57.64s-10.88,6-19.28,6Zm18.08-3.75h4.35c4.88,0,8.48-1.58,10.73-4.73s3.38-8.1,3.38-14.93-1.13-11.78-3.38-14.86-5.85-4.65-10.73-4.65h-4.35Zm73.76-11.33H299.63v.23c0,4.28.68,7.43,2,9.3s3.38,2.85,6.3,2.85a8.71,8.71,0,0,0,5.85-1.88,9.2,9.2,0,0,0,2.85-5.55h5.18a13.44,13.44,0,0,1-5.33,8.33c-2.7,1.8-6.38,2.7-11.26,2.7-5.78,0-10.2-1.5-13.28-4.58s-4.65-7.35-4.65-13.06c0-5.55,1.58-9.83,4.73-12.91s7.58-4.65,13.13-4.65,9.75,1.65,12.68,4.88S322.37,42.33,322.52,48.56Zm-12.16-3.68c0-4.35-.38-7.43-1.2-9.23A4.19,4.19,0,0,0,305,33a4.33,4.33,0,0,0-4.13,2.63c-.83,1.73-1.2,4.65-1.2,8.7V45h10.73Zm30.69,18.76L329.27,34.15H325.6V30.33h19.13v3.83h-4.05l8.4,21,8.4-21h-4.35V30.33h12.53v3.83h-3.83L350.06,63.64Zm62.2-15.08H380.37v.23c0,4.28.68,7.43,2,9.3s3.38,2.85,6.3,2.85a8.71,8.71,0,0,0,5.85-1.88,9.2,9.2,0,0,0,2.85-5.55h5.18a13.44,13.44,0,0,1-5.33,8.33c-2.7,1.8-6.38,2.7-11.26,2.7-5.78,0-10.2-1.5-13.28-4.58S368,52.61,368,46.91c0-5.55,1.58-9.83,4.73-12.91s7.58-4.65,13.13-4.65,9.75,1.65,12.68,4.88S403.11,42.33,403.26,48.56ZM391.1,44.88c0-4.35-.38-7.43-1.2-9.23A4.19,4.19,0,0,0,385.7,33a4.33,4.33,0,0,0-4.13,2.63c-.83,1.73-1.2,4.65-1.2,8.7V45H391.1Zm34.37,15h4.73v3.83H409.64V59.89h4.73V18.62h-4.73V14.87h15.83v45Zm29.11,4.65c-5.85,0-10.5-1.58-13.81-4.65s-5-7.43-5-12.91,1.65-9.83,5-12.91,8-4.65,13.81-4.65,10.5,1.58,13.81,4.65,5,7.43,5,12.91-1.65,9.75-5,12.91S460.44,64.54,454.58,64.54Zm0-3.53a5.05,5.05,0,0,0,5-3c1-2,1.43-5.7,1.43-11s-.45-9-1.43-11a5.05,5.05,0,0,0-5-3,5,5,0,0,0-5,3c-1,2-1.43,5.7-1.43,11s.45,9,1.43,11A4.88,4.88,0,0,0,454.58,61Zm28.89-26.86h-4.73V30.33h15.83v4.2a9.59,9.59,0,0,1,3.83-3.9,12.61,12.61,0,0,1,5.93-1.28c4.73,0,8.48,1.58,11.18,4.65s4.13,7.43,4.13,12.83-1.35,9.75-4.13,12.91-6.45,4.73-11.18,4.73a13,13,0,0,1-5.93-1.28,8.76,8.76,0,0,1-3.83-3.9V73.1h5.18v3.83h-21V73.1h4.73Zm11.11,11.18v3.3c0,3.9.53,6.68,1.5,8.4a5.17,5.17,0,0,0,4.8,2.63,4.89,4.89,0,0,0,4.8-2.78c.9-1.8,1.43-5.18,1.43-9.9s-.45-8.1-1.43-9.9a5,5,0,0,0-4.8-2.78,5.09,5.09,0,0,0-4.8,2.63C495,38.66,494.58,41.51,494.58,45.33Zm66.78,3.23H538.47v.23c0,4.28.68,7.43,2,9.3s3.38,2.85,6.3,2.85a8.71,8.71,0,0,0,5.85-1.88,9.2,9.2,0,0,0,2.85-5.55h5.18a13.44,13.44,0,0,1-5.33,8.33c-2.7,1.8-6.38,2.7-11.26,2.7-5.78,0-10.2-1.5-13.28-4.58s-4.65-7.35-4.65-13.06c0-5.55,1.58-9.83,4.73-12.91s7.58-4.65,13.13-4.65,9.75,1.65,12.68,4.88S561.13,42.33,561.36,48.56Zm-12.23-3.68c0-4.35-.38-7.43-1.2-9.23a4.19,4.19,0,0,0-4.2-2.63,4.33,4.33,0,0,0-4.13,2.63c-.83,1.73-1.2,4.65-1.2,8.7V45h10.73ZM600,30v9.9h-3.53a6.44,6.44,0,0,0-1.43-4,4.61,4.61,0,0,0-3.6-1.28,6.39,6.39,0,0,0-5.7,3.23c-1.43,2.1-2.1,5.1-2.1,8.85V59.82h6.08v3.83h-22V59.82h4.73V34.15h-5.1V30.33h16.21v5.93a11.9,11.9,0,0,1,4.28-5.18,12.15,12.15,0,0,1,6.6-1.65c.68,0,1.43.08,2.4.15S598.8,29.8,600,30Z“ transform=„translate(0 0)“ fill=„#661136“/><path d=„M99.72,27.6h4.39V42.76h.07a8.46,8.46,0,0,1,3.15-3.07,9.34,9.34,0,0,1,4.54-1.24c3.22,0,8.49,2,8.49,10.4V63.26H116V49.35c0-3.88-1.46-7.25-5.64-7.25a6.33,6.33,0,0,0-5.93,4.39,5.94,5.94,0,0,0-.29,2.12V63.26H99.72V27.6ZM130,51.91c.07,5.93,3.88,8.42,8.35,8.42a16.93,16.93,0,0,0,6.74-1.24l.73,3.15a18.82,18.82,0,0,1-8.05,1.54c-7.47,0-11.93-5-11.93-12.3s4.32-13.11,11.35-13.11c7.91,0,10,7,10,11.42a13.18,13.18,0,0,1-.15,2H130Zm12.89-3.22c.07-2.78-1.17-7.17-6.15-7.17-4.47,0-6.44,4.1-6.74,7.17Zm14.64-16.55a2.56,2.56,0,0,1-2.78,2.71,2.6,2.6,0,0,1-2.64-2.71,2.74,2.74,0,0,1,2.78-2.78A2.62,2.62,0,0,1,157.55,32.14Zm-4.91,31V38.95H157V63.18Zm11.13-4.47a11.26,11.26,0,0,0,5.78,1.76c3.22,0,4.69-1.61,4.69-3.59s-

1.24-3.22-4.54-4.47c-4.39-1.54-6.44-4-6.44-6.88,0-3.88,3.15-7.1,8.35-7.1A11.71,11.71,0,0,1,177.54,40l-1.1,3.22a9.62,9.62,0,0,0-5-1.39c-2.64,0-4,1.54-4,3.29,0,2,1.46,2.93,4.61,4.1,4.17,1.61,6.37,3.73,6.37,7.32,0,4.25-3.29,7.32-9.08,7.32a14.37,14.37,0,0,1-6.81-1.68Zm22.92-6.81c.07,5.93,3.88,8.42,8.35,8.42a16.93,16.93,0,0,0,6.74-1.24l.73,3.15a18.82,18.82,0,0,1-8.05,1.54c-7.47,0-11.93-5-11.93-12.3s4.32-13.11,11.35-13.11c7.91,0,10,7,10,11.42a13.18,13.18,0,0,1-.15,2H186.69Zm12.89-3.22c.07-2.78-1.17-7.17-6.15-7.17-4.47,0-6.44,4.1-6.74,7.17ZM70.58,57,67.5,54.32a28,28,0,0,0-6.15-17.13c0-17.79-13.91-29.36-31.12-29.36-23.06,0-31.12,18.23-31.12,38.58,0,18,13.25,32.87,31.55,32.87A45.36,45.36,0,0,0,69,71a27.71,27.71,0,0,0-4.17-3.59L75,70.06A42.67,42.67,0,0,1,43.42,85C19.62,85.07,0,67.79,0,43.42,0,18.67,18.08,0,43,0c20.57,0,37.7,13,37.7,34.63C80.68,42.61,76.43,51.54,70.58,57ZM48.69,27.38,34.92,58.72c-1.17,2.64-2.93,5.71-6.22,5.71a4.12,4.12,0,0,1-4.32-4.17c0-1.9,1-3.66,1.68-5.34L41,20.87c1.17-2.56,2.56-4.47,5.49-4.47a4,4,0,0,1,4.17,4.17C50.66,23,49.64,25.26,48.69,27.38ZM59.6,46.49,54.1,58.86c-1.24,2.64-3,5.56-6.3,5.56a4,4,0,0,1-4.17-4.17A12.72,12.72,0,0,1,45.32,55L52,40c1.17-2.49,2.56-4.47,5.49-4.47a4,4,0,0,1,4.17,4.17A16,16,0,0,1,59.6,46.49Z“ transform=„translate(0 0)“ fill=„#888“/></svg></figure> 2016 stellte Espressif eine leistungsfähige Familie von Microcontrollern auf Basis des ESP32 vor. Dieses Blog hat den ESP32 zwar bereits früher thematisiert, aber zum Auftakt einer Reihe von Beiträgen zu diesem Thema werden ESP32 und entsprechende Boards noch mal genauer beleuchtet. Das chinesische Chip-Unternehmen Espressif hat vor wenigen Jahren durch seinen ESP8266-Chip große Euphorie bei Makern ausgelöst. Der ESP8266 als System-on-a-Chip (SoC) integriert sowohl einen leistungsstarken Microcontroller als auch eine WiFi-Komponente. Entsprechende Boards sind inzwischen schon für eine Handvoll Euros zu haben. Ich habe ESP-01-Boards bereits in diesem Blog genutzt, um Arduinos preiswert mit dem WLAN beziehungsweise dem Internet zu verbinden. Es ist dabei aber leicht zu übersehen, dass der ESP8266 in vielen Aspekten mit Arduinos ATMEL-Chips mithalten kann. <h3 class=„subheading“>Warum ist der ESP32 interessant?</h3> Wie auch bei Vierrad-Enthusiasten üblich soll zuerst ein Blick unter die Motorhaube erfolgen. ESP32-Chips enthalten eine ganze Reihe von interessanten Merkmalen: <ul class=„rtelist rtelist–unordered“>
Sie beherbergen meistens zwei 32-Bit-Prozessorkerne des Typs Tensilica Xtensa LX6, die mit 160 MHz bis 240 MHz Taktfrequenz arbeiten. “Meistens” deshalb, weil es auch eine ESP32-Variante mit nur einem Kern gibt. Der “Laderaum” sorgt für komfortables Wohngefühl: Mit 520 KB RAM und 448 KB ROM dürften für die meisten Embedded-Anwendungen genügend Reserven vorhanden sein. Im Gegensatz zu vielen Arduino-Boards verwendet der ESP32 3,3V als Betriebsspannung. Der entscheidende Vorteil: Weil viele Sensoren und Aktuatoren ebenfalls mit 3,3V arbeiten, entfällt der sonst notwendige Pegelwandler zwischen 3,3V- und 5V-Komponenten. Da heutzutage das Thema Energiesparen vorherrscht, sei auf den Low-Power-Modus des ESP32 verwiesen. Im Tiefschlaf (Deep Sleep) verbraucht ein ESP32-Board nur einen Bruchteil der sonst benötigten Leistung. Das ermöglicht autonome Systeme, die im Batterie-/Akku-Betrieb mit langer Laufzeit auskommen müssen. Diverse Schnittstellen verbinden den Microcontroller mit der Außenwelt. Darunter befinden sich UARTs, SPI, CAN, I2C, I2S, PWM – ich spare mir aufgrund des Trägheitsprinzips die Beschreibung der diversen Hardwareschnittstellen, da sie bis auf CAN schon in vergangenen Folgen zur Sprache kamen. Mit CAN ist ein standardisiertes Bussystem gemeint, das häufig in Anwendungen der Automatisierungstechnik zum Einsatz kommt. Autohersteller und ihre Zulieferer nutzen CAN zur Kommunikation verschiedener Steuereinheiten miteinander. Damit sich analoge Werte in digitale umwandeln lassen, verfügt der Chip über diverse ADCs (Analog Digital Converter). Umgekehrt erlauben DACs (Digital Analog Converter) die Wandlung digitaler in analoge Signale. Zur drahtlosen Kommunikation mit anderen Geräten integriert der ESP32 sowohl WLAN als auch Bluetooth. Letzteres war beim ESP8266 nicht möglich. Ohne Firmware lässt sich ein Microcontroller nicht nutzen. Die Firmware des ESP32 ist in einem seriell anschließbaren externen Flash-Speicher untergebracht. Es macht daher meistens nur wenig

Sinn, einen Standalone-ESP32-Chip zu erwerben, sondern stattdessen ein Board, das den Firmwarespeicher neben anderer Features – zum Beispiel einen USB-Anschluss – umfasst. Zu guter Letzt: Ein integrierter Hall-Sensor erlaubt die Messung elektromagnetischer Felder. Die im ESP32 integrierte Crypto-Einheit finden in Anwendungen Einsatz, um die Kryptographie-Operationen zu beschleunigen. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12 c2“></figure> Allerdings gibt es nicht nur eine Variante des ESP32, sondern gefühlt ein gutes Dutzend. Die heißen dann auch mal ESP32S, firmieren je nach Größe als WROOM oder WROVER, haben verschiedene Erweiterungen. Uns sollen deren Unterschiede kalt lassen. Für die Hardware-Interessierten verweise ich auf die Webseite esp32.net [1]. <h3 class=„subheading“>Come on Board</h3> Selbstredend existiert nicht das eine ESP32-Board, sondern Dutzende. Solche mit Display und solche ohne, solche mit LoRA-Kommunikation und solche ohne. Dazu verschiedenste Formfaktoren, nach außen gelegte Pins und dergleichen mehr. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12 c3“><figcaption class=„rteinlinebild_source akwa-caption“>Ein Board des Typs ESP32 (Bild: amazon.de)</figcaption></figure> Als NodeMCU firmieren die Boards, die im Auslieferungszustand die Skriptsprache LUA und die entsprechende Firmware beherbergen. Grundsätzlich gibt es unterschiedliche Firmware-Optionen, die auf ESP32-Boards laufen, darunter zum Beispiel MicroPython (oder CircuitPython auf Adafruit-Boards), FreeRTOS (RTOS = Real-Time Operating System), und der Arduino Core für ESP32. Letztere Option nutze ich für den Rest dieses Blog-Postings. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“><figcaption class=„rteinlinebild_source akwa-caption“>Pin-Belegung des ESP32 Smart Thing von SparkFun (Bild: SparkFun.com)</figcaption></figure> Zu den häufig anzutretenden Vertreter ihrer Gattung gehören beispielsweise das Original-ESP32-Dev-Modul von Espressif und seine zahlreichen Klone sowie das DOIT ESP32 DevKit V1. Diese gibt es in der Regel für Preise von 5 bis 10 Euro bei den üblichen Verdächtigen (eBay, Amazon, Watterott, EXP-TECH, Alibaba ....). Das finanzielle Risiko hält sich also in Grenzen. Eine Bestellung in China kommt noch etwas billiger, lohnt sich wegen der längeren Lieferzeiten aber nur, wenn es einem nicht schon in den Fingern juckt. <h3 class=„subheading“>Eine Frage der Programmierung</h3> Es gibt diverse Optionen, die eine Host-/Target-Entwicklung mit ESP32-Boards unterstützen: <ul class=„rtelist rtelist–unordered“>
Von Espressif selbst steht die ESP-IDF-Plattform im Angebot. Das Open-Source-Projekt ESP32 Arduino Core liefert eine Sammlung von Bibliotheken und Werkzeugen, mit der sich die Arduino IDE nutzen lässt, um Software für ESP32-Boards zu programmieren. Weiterer Vorteil: Leistungsstarke Entwicklungsumgebungen wie Visual Studio Code verfügen über Plug-ins für

Arduino und Arduino ESP32 Core. In Rahmen dieses Blogs dient die Arduino IDE in Kombination mit dem Arduino Core für ESP32 als Werkzeug der Wahl. Sie ist kostenlos und bietet ausreichende Funktionalität. Man könnte sie gewissermaßen als MVP (Minimal Viable Produkt) bezeichnen. Aber keine Sorge! In späteren Folgen kommt auch noch die Alternative Visual Studio Code zur Sprache. <h3 class=„subheading“>Arduino IDE</h3> Um ESP32-Boards unter der Arduino IDE zu nutzen, müssen Entwickler zunächst eine möglichst aktuelle Version der IDE für Windows, Linux, oder macOS herunterladen. Das Installationspaket steht auf der Downloadseite von arduino.cc [2] zur Verfügung. Auf die Installation gehe ich an dieser Stelle nicht weiter ein und verweise auf frühere Postings. Dazu gibt es YouTube-Ressourcen wie die hier [3]. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“><figcaption class=„rteinlinebild_source akwa-caption“>Arbeiten mit der Arduino IDE</figcaption></figure> Nach Installation der Arduino IDE ist folgende URL des gewünschten ESP32-Boardmanagers in der Einstellungsseite (Windows: File | Preferences, macOS: Arduino | Preferences) einzutragen: https://dl.espressif.com/dl/package_esp32_index.json. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“></figure> Anschließend sollte man im Boardsmanager (Tools | Boards | Boardsmanager) nach „ESP32“ suchen. Dort müsste das Paket „esp32 by Espressif Systems“ auftauchen, das sich mit Install installieren lässt. <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“></figure> Das war es auch schon. Besser gesagt fast. Die (chinesischen) ESP32-Boards enthalten häufig einen UART-to-USB-Baustein von SiLabs. UART steht für Universal Asynchronous Receiver Transmitter und dient der seriellen Kommunikation zwischen Host-PC und Embedded Board über USB. Um am Windows-, Linux-, macOS-Computer mit dem Board kommunizieren zu können, ist ein entsprechender Treiber notwendig. Den gibt es über das Internet unter der URL <SiLabs> [4]). <h3 class=„subheading“>Von Mappings und anderen Schikanen</h3> Um die verschiedenen Pins eines ESP32-Boards mit symbolischem Namen innerhalb der Arduino IDE anzusprechen, existieren sogenannte Mappings, bereitgestellt durch den Arduino Core. Dazu finden sich im Installationsverzeichnis der IDE (Arduino | hardware | espressif | esp32 | variants | esp32) entsprechende Deklarationen in pins_arduino.h. Wie aus dem folgenden Bild ersichtlich, erweist sich ein solches Mapping als wichtig, um etwas Ordnung ins Namenschaos zu bringen.

<figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“><figcaption class=„rteinlinebild_source akwa-caption“>Das Mapping der ESP32-Pins auf die Arduino IDE (Bild: LastMinuteEngineers.com)</figcaption></figure><h3 class=„subheading“>Funktion der Tasten</h3> Im Regelfall befinden sich zwei Tasten auf einem ESP32-Board, eine EN-Taste und eine BOOT-Taste. Der ESP32 kennt zwei Betriebsmodi, Normalmodus und Firmware-Update-Modus. Für das Einspielen neuer Software (eigenes Programm plus Laufzeitumgebung/Firmware) müssen Entwickler das Board daher in den Uploadmodus versetzen. Die EN-Taste sorgt lediglich für einen Restart. Stattdessen ist für den Software-Upload folgendes Vorgehen notwendig: <ol class=„rtelist rtelist–ordered“>
BOOT-Taste drücken und gedrückt halten EN-Taste betätigen und wieder loslassen BOOT-Taste loslassen Achtung: <ul class=„rtelist rtelist–unordered“>
Dieses Vorgehen können einige Boards auch automatisieren. Die Tasten haben auf unterschiedlichen Boards unterschiedliche Namen (z. B. „RS“ und „0“, „BOOT“ und „RST“). <figure class=„rteinlinebild akwa-inline-img akwa-inline-img col-lg-12 col-md-12 col-sm-12 col-xs-12“><figcaption class=„rteinlinebild_source akwa-caption“>Ein Board des Typs SparkFun ESP32 Thing mit Tasten „RST“ und „0“</figcaption></figure> Sollte beim Kompilieren eines Programms innerhalb der IDE zu einer Fehlermeldung mit roter Schrift kommen, genügt es meist, die BOOT-Taste ein bisschen gedrückt zu halte, um sie anschließend wieder los zu lassen. <h3 class=„subheading“>Das erste Mal</h3> Nun erfolgt die erste Prüfung des jungfräulichen ESP32-Boards. Die Arduino IDE soll einen ersten Einblick vermitteln, dass die Programmierung exakt auf dieselbe Weise erfolgen kann wie bei Arduino-Boards. Sobald der ESP32 am Hostcomputer angeschlossen ist, sucht man im Tools-Menü nach dem benutzten Board und stellt es ein. Zudem gibt im Ports-Bereich den vom Board verwendeten seriellen (SLab-)Port ein. Ich selbst nutze zum Experimentieren das ESP32 Smart Thing von SparkFun sowie ein ESP32-Board von Watterott, besitze aber aus Preisgründen zahlreiche Boards des Typs NodeMCU32 sowie Lolin32 aus chinesischer Produktion. <figure class=„rteinlinebild akwa-inline-img rtepos_left col-lg-6 col-md-6 col-sm-6 col-xs-12 akwa-inline–left“><figcaption class=„rteinlinebild_source akwa-caption“>ESP32-Boards gibt es fast so viele wie Sand am Meer</figcaption></figure> Um das ESP32-Board zum ersten Mal zu programmieren, öffnet man in der Arduino IDE ein neues Projekt beziehungsweise einen neuem Sketch und gibt nachfolgenden Code ein. Der Sketch geht davon aus, dass sich die eingebaute LED an Pin 2 befindet. Sollte Entwickler ein anderes Board besitzen, lesen sie dessen Beschreibung und definieren sie gegebenenfalls für die Konstante LED einen anderen Wert. Danach lässt man das Programm übersetzen und aufs Board übertragen. Nach dem Öffnen des seriellen Monitors gibt man 115.200 Baud als Geschwindigkeit ein. Um den seriellen Monitor zu

öffnen, navigiert man über den Menüpfad Tools | Serial Monitor. Jetzt müsste sowohl die Onboard-LED im Einsekundentakt blinken als auch der Text “

```
Hallo, ESP32
```

” wiederholt auf dem seriellen Monitor erscheinen. <pre class=„rtetx–listing“><code>const int LED = 5; *Eingebaute blaue LED an Pin 5
* Setup - läuft nach jedem Reset genau einmal
void setup() {
  *Digitaler Ausgang steuert die LED
 pinMode(LED, OUTPUT);
 Serial.begin(115200); * Seriellen Port mit 115200 Baud initialisieren
}

 *Unendliche Ereignisschleife; HIGH & LOW sind Spannungslevels!
void loop() {
 Serial.println(“Hallo, ESP32”); * Ausgabe an seriellen Monitor senden
 digitalWrite(LED, HIGH); *Es werde Licht
 delay(1000); * Wartezeit von einer Sekunde
 digitalWrite(LED, LOW); *Licht aus
 delay(1000); * Wartezeit von einer Sekunde
}</code></pre> Damit ist die Jungfernfahrt bereits erledigt. Wer experimentieren will, könnte zum Beispiel an Pin 5 auch eine externe LED anschließen oder andere Schikanen einbauen. <h3 class=„subheading“>Arbeitsteilung</h3> Da der ESP32 eine Mehrkernarchitektur aufweist, lassen sich echt parallele Threads beziehungsweise Tasks einsetzen. Das folgende Programm nutzt diese Möglichkeit exemplarisch, um Tasks mittels xTaskCreatePinnedToCore() zu erzeugen und sie an einen der beiden Kerne zu binden. Der erste Task taskOne läuft auf dem ersten Kern 0 der zweite taskTwo auf dem zweiten Kern 1. Beide Tasks haben Priorität 1 und erhalten 10.000 Bytes Stackgröße. taskOne lässt die eingebaute LED jede halbe Sekunde blinken, während taskTwo jede halbe Sekunde Text am seriellen Monitor ausgibt. <pre class=„rtetx–listing“>TaskHandle_t task1; *Jeder Task benötigt einen Handle
TaskHandle_t task2;
* LED Pin ist die eingebaute Pin
const int ledPIN = 5;
void setup() {
 Serial.begin(115200);
 pinMode(ledPIN, OUTPUT);

  *task ausgeführt in taskOne()
* auf dem ersten Core (Core 0), Prio: 1
 xTaskCreatePinnedToCore(
 taskOne, /* Funktion mit Code des Tasks */
 „TaskOne“, /* Name des Tasks */
 10000, /* Stackgröße des Tasks */
 NULL, /* Parameter des Tasks */
 1, /* Priorität des Tasks */
 &task1, /* Handle auf Task */
 0); /* Task soll auf Kern 1 laufen */
 delay(500);
  *task ausgeführt in taskTwo()
* auf dem zweiten Core (Core 1), Prio: 1
 xTaskCreatePinnedToCore(
 taskTwo, /* Funktion mit Code des Tasks */
 „TaskTwo“, /* Name des Tasks */
 10000, /* Stackgröße des Tasks */
 NULL, /* Parameter des Tasks */
 1, /* Priorität des Tasks */
 &task2, /* Handle auf Task */
 1); /* Task soll auf Kern 2 laufen */
 delay(500);
}
 *taskOne: LED soll alle 2 Sekunden blinken
void taskOne(void * optionalArgs){
 for(;;){
 digitalWrite(ledPIN, HIGH);
 delay(500);
 digitalWrite(ledPIN, LOW);
 delay(500);
 }
}
* taskTwo: Ausgabe am seriellen Monitor alle 100 msec
void taskTwo(void * optionalArgs){
 Serial.print(„Task 2 läuft auf Kern “);

Serial.println(xPortGetCoreID())
 for(;;){
 Serial.println(„LED an“);
 delay(500);
 Serial.println(„LED aus“);
 delay(500);
 }
}
void loop() { /* braucht es nicht */ }
</pre> Die Methode loop() bleibt in diesem Fall ungenutzt, weil sich die Ereignisschleife ohnehin in den endlos laufenden Tasks abspielt. Das Beispiel ist zwar lehrreich, aber etwas theoretisch. In der Praxis könnte einer der Tasks die Kommunikation mit der Außenwelt übernehmen, während der zweite Messwerte von Sensoren liest. <h3 class=„subheading“>Let's talk</h3> In jedem Fall wäre es hilfreich, würden Tasks auch Information austauschen. Genau das demonstriert das nachfolgende Beispiel. Mittels xQueueCreate legt das Hauptprogramm eine Queue mit einem Slot des Datentyps unsigned long an. Das Kreieren der Tasks bewerkstelligt diesmal die Methode xTaskCreate. Das Programm implementiert eine simple Produzenten-Konsumenten-Konstellation. Der Aufruf von vTaskDelay ist das ESP32-Pendant zu Arduinos delay, arbeitet aber mit höherer Auflösung, weshalb wir die gewünschte Zeitdauer noch durch die Konstante portTICK_PERIOD_MS teilen müssen. Die Konstante gibt an, wie viele Taktzyklen pro Millisekunde durchgeführt werden. Um sicherzustellen, dass ein Wert in der Queue vorhanden und ungelesen ist oder bereits ein neuer geschrieben werden kann, fragt der Produzent queue auf Null ab, bevor er schreibt. Entsprechend prüft der Konsument, ob tatsächlich ein Wert in der Queue vorliegt. <pre class=„rtetx–listing“>QueueHandle_t queue = NULL; *Queue anlegen
void setup()
{
 printf(„Starte zwei Tasks und eine Queue /n /n“);
 queue = xQueueCreate(20,sizeof(unsigned long));
 if(queue != NULL){
 printf(„Queue kreiert \n“);
 vTaskDelay(1000/portTICK_PERIOD_MS); * Eine Sekunde warten
 xTaskCreate(&produzent,„produzent“,2048,NULL,5,NULL);
 printf(„Produzent gestartet \n“);
 xTaskCreate(&konsument,„konsument“,2048,NULL,5,NULL);
 printf(„Konsument gestarted \n“); *Hier nutzen wir mal C/C++
 } else {
 printf(„Queue konnte nicht angelegt werden \n“);
 }
}
void konsument(void *pvParameter)
{
 unsigned long counter;
 if (queue == NULL){
 printf(„Queue nicht bereit \n“);
 return;
 }
 while(1){
 xQueueReceive(queue,&counter,(TickType_t)(1000/portTICK_PERIOD_MS));
 printf(„Empfangener Wert über Queue: %lu \n“,counter);

 vTaskDelay(500/portTICK_PERIOD_MS);* halbe Sekunde warten
 }
}
void produzent(void *pvParameter){
 unsigned long counter=1;
 if(queue == NULL){
 printf(„Queue nicht bereit \n“);
 return;
 }
 while(1){
 printf(„An Queue gesendeter Wert: %lu \n“,counter);
 xQueueSend(queue,(void *)&counter,(TickType_t)0);
 * … schreibt den wert von Counter in die Queue
 counter++;
 vTaskDelay(500/portTICK_PERIOD_MS); * halbe Sekunde warten
 }
}
void loop() {}</pre> <h3 class=„subheading“>Deep Sleep</h3> Ein ESP32-Board braucht im aktiven Zustand schon einige mA, im Schnitt etwa 150 bis 260 mA. Im aktiven Modus inklusive aller

Komponenten wie WiFi und Bluetooth können das sogar bis zu 800 mA bei Spitzenlasten sein. Innerhalb von IoT-Anwendungen kommt es aber durchaus häufig vor, dass Geräte mittels Sensoren nur alle Minuten oder sogar Stunden kurzzeitige Messungen vornehmen, um sie anschließend per Kommunikationsprotokoll nach außen zu übertragen. Es macht also keinen Sinn, einen Embedded-Controller ständig aktiv arbeiten zu lassen. Bei einem autonomen, batteriebetriebenen Gerät müsste man ansonsten alle paar Stunden die Batterie ersetzen. Stellt man sich vor, dass das Gerät auf einem Baum in großer Höhe angebracht ist, dürfte die praktische Dimension dieses Problems klar sein. Deshalb unterstützt der ESP32 diverse Sparmodi. Beim Deep-Sleep-Modus sind nur noch der RTC (Echtzeituhr) und der ULP-Koprozessor (ULP = Ultra Low Power) aktiv. Der benötigte Reststrom beträgt in diesem Fall gerade einmal 2,5 micro Ampere. Damit lässt sich gut auskommen. Um vom Tiefschlaf zu erwachen, braucht der ESP32 keinen schönen Prinzen, obwohl auch das möglich wäre. Es gibt drei Optionen: Betätigen eines Touch-Pins, Ereignis an einem externen Pin (= Signalflanke) oder zeitgesteuertes Aufwecken mittels des ULP-Koprozessors. Diesen letzteren Fall beleuchtet nachfolgendes Beispiel. Allerdings führt der Tiefschlaf auch eine Art digitale Demenz nach sich, weil die Daten (= Variablen) bis auf eine Ausnahme verloren gehen. Alle Daten, die mit dem Schlüsselattribut RTC_ATTR_DATA deklariert sind, bleiben erhalten. Immerhin beherbergt dieser persistente Speicher rund viermal so viele Daten wie ein gewöhnlicher Arduino Uno/Mega/Nano Speicher insgesamt mitbringt. Im folgenden Programm liegt die Variable restarted im besagten Speicher. Sie zählt einfach mit, wie oft bereits ein Neustart erfolgte. Beim ersten Start (restarted == 0) blinkt die eingebaute LED kurz. Sind schon mehrere Restarts nach Tiefschafphasen erfolgt (restarted != 0), blinkt die LED häufiger. Die Einleitung eines zeitgesteuerten Tiefschlafs beginnt mit dem Aufruf von esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR. Die Zeit in Sekunden gibt TIME_TO_SLEEP wieder. Da intern die Berechnung in Microsekunden erfolgt, muss der Umrechnungsfaktor uS_TO_S_FACTOR noch dazu multipliziert werden. Den eigentlichen Tiefschlaf leitet der folgende Aufruf ein: esp_deep_sleep_start()<code>:</code> <pre class=„rtetx–listing“>#define uS_TO_S_FACTOR 1000000 *Microsekunden =>*

*Sekunden
#define TIME_TO_SLEEP 3 * Schlaflänge in Sekunden
RTC_DATA_ATTR int restartet = 0;
#define LEDPIN 2 *eingebaute LED
#define BLINK_DELAY 200 * Blinkfrequenz = 200 / BLINK_DELAY
void setup() {
 Serial.begin(115200);
 Serial.println(“Deep Sleep mittels Timer - Demo”);
 pinMode(LEDPIN,OUTPUT); *LED

 delay(500); * a bisserl Geduld

 if(restartet == 0) *das erste Mal
 {
 Serial.println(“Initialer Start”);
 digitalWrite(LEDPIN, HIGH); * Licht an
 delay(10 * BLINK_DELAY);
 digitalWrite(LEDPIN, LOW); *Spot aus
 restartet++;
 } else * nicht mehr jungfräulich
 {
 Serial.println(“Schon ein paar Mal aus Schlaf erwacht: “ + String(restartet)));
 blinken(restartet);
 }

  *In Tiefschlaf versetzen
 Serial.print(“Tiefschlaf wird initiiert für “ + String(TIME_TO_SLEEP));
 Serial.println(“ Sekunden”);
 esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
 esp_deep_sleep_start(); * Schlaf beginnt
}
void blinken(byte n) { *n mal Blinken im*

*BLINK_DELAY msec Abstand
 Serial.println(“Blinken startet”);
 for (i = 0; i < n; i++) {
 Serial.println(“LED an”);
 digitalWrite(LEDPIN, HIGH);
 delay(BLINK_DELAY);
 Serial.println(“LED aus”);
 digitalWrite(LEDPIN, LOW);
 }
}
void loop() {
 Serial.println(“Unerreichter Code”);
}</pre> Da das Programm den Tiefschlaf in setup() durchführt, kommt es in diesem Beispiel natürlich nie zur Ausführung der Methode loop()<code>.</code> <h3 class=„subheading“>Fazit</h3> Nun sind wir am Ende dieses Beitrags angekommen, womit die Basis für eigene Experimente gelegt wäre. Dabei lag der Fokus auf die inneren Werte des ESP32. Wir haben die funktionale Architektur beleuchtet, einige besondere Aspekte wie Parallelisierung und Tiefschlaf betrachtet und dazu Beispiele kennen gelernt. Im Mittelpunkt der Programmierung stand dabei die Arduino IDE. Beim nächsten Beitrag kommt die Interaktion des ESP32 mit der Außenwelt zur Sprache. Themen sind dann insbesondere die WiFi-Funktionalität des Microcontrollers. Bis dahin viel Spaß beim Experimentieren! <hr/> URL dieses Artikels:
<small><code>http://www.heise.de/-4452689</code></small> Links in diesem Artikel:
<small><code>[1] http://esp32.net</code></small>
<small><code>[2] https://www.arduino.cc/en/Main/Software</code></small>
<small><code>[3] https://www.youtube.com/watch?v=39yjdj1b9bs</code></small>
<small><code>[4] https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers</code></small>
 Copyright © 2019 Heise Medien </html>*