c't Make:

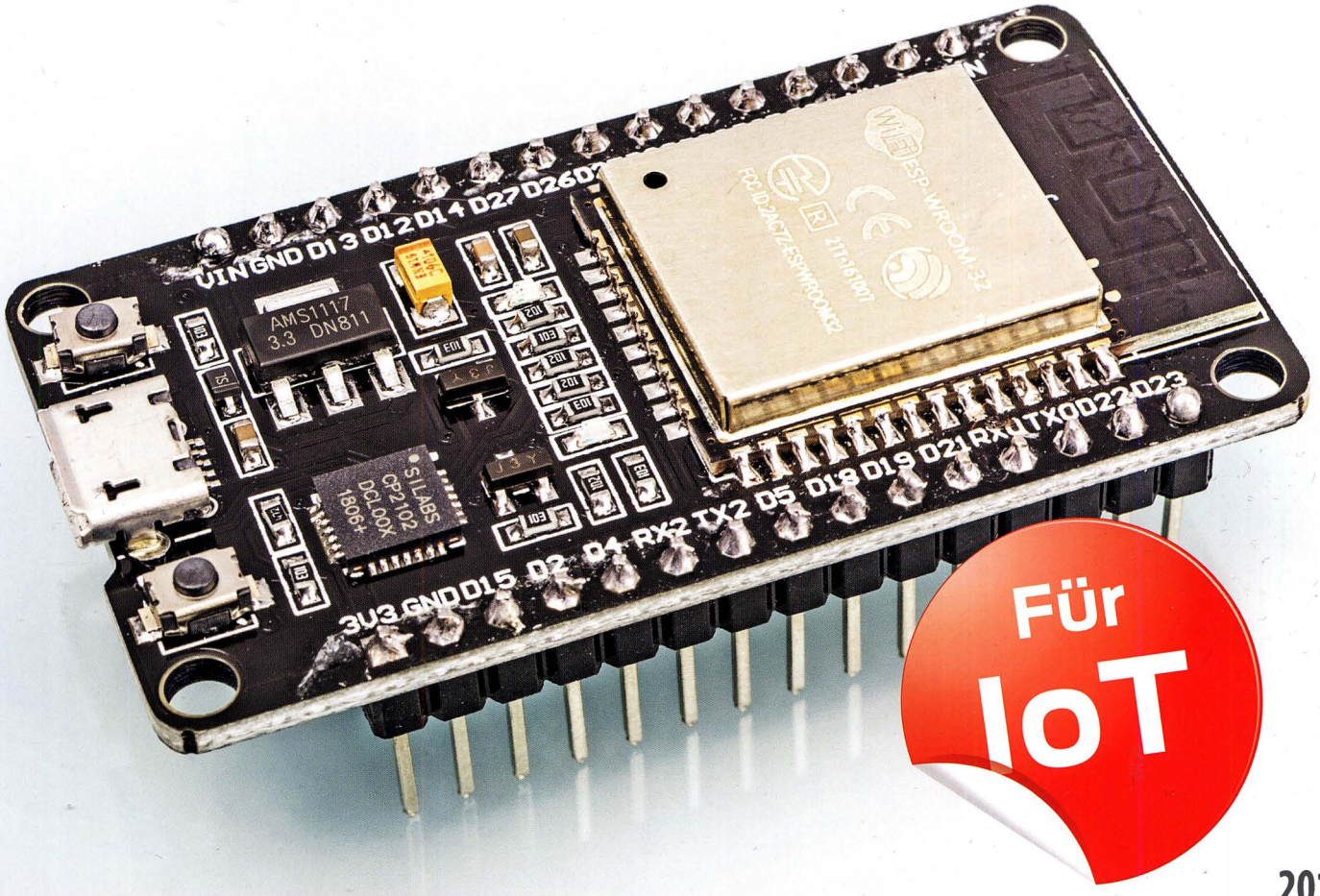
Make:

inklusive ESP32 mit WLAN und Bluetooth

ESP32 SPECIAL

Grundlagen

- Software schnell einrichten
- Hardware-Funktionen verstehen
- ESP32 ins WLAN bringen
- Per Bluetooth Daten austauschen
- Digital-Analog-Wandler steuern
- Touch-Eingänge auslesen
- Hallsensoren nutzen
- BASIC-Interpreter freischalten



Für IoT

PROJEKTE

Mini-Webserver mit Temperaturmessung •
Bluetooth-Sensoren • Funktionsgenerator •
Piano mit Toucheingabe • Türüberwachung

2019
€ 24,95

4 190645 424951 01

www.make-magazin.de

Komplett digital!

Arduino-Projekte für Fortgeschrittene:

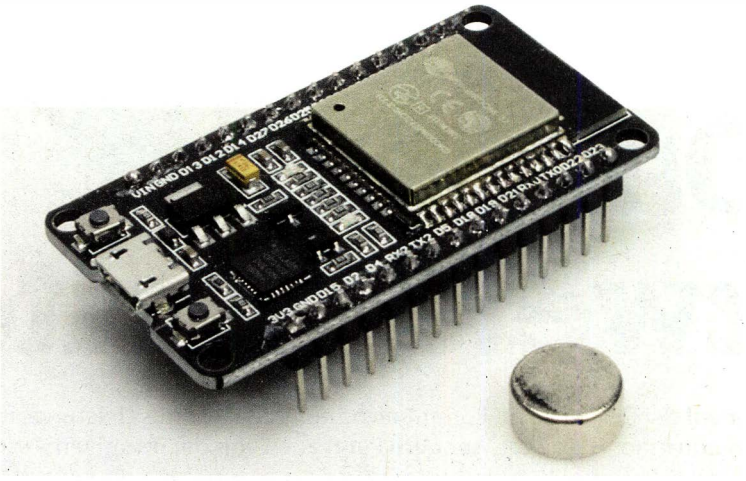
Digital als PDF

Auch erhältlich:
Teil 1 für Einsteiger

Mit dem zweiten Teil der Special-Reihe aus der Make-Redaktion zum Arduino UNO bekommen Sie mehr von allem: mehr Tiefe, mehr Grundlagen, mehr Projekte! Freuen Sie sich auf über 60 Seiten Profiwissen: die Arduino-IDE durchleuchtet, Hardware-Interna, Audio und VGA-Monitor am Arduino und Cheat Sheet - die wichtigsten Befehle. **Bestellen Sie unter: shop.heise.de/make-arduino**

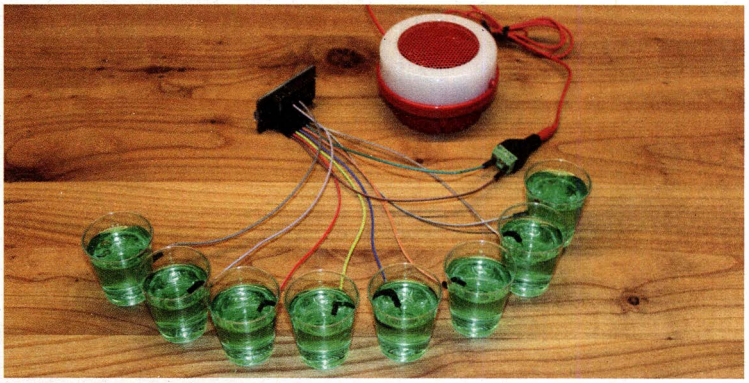
heise shop
shop.heise.de/make-arduino

Inhalt



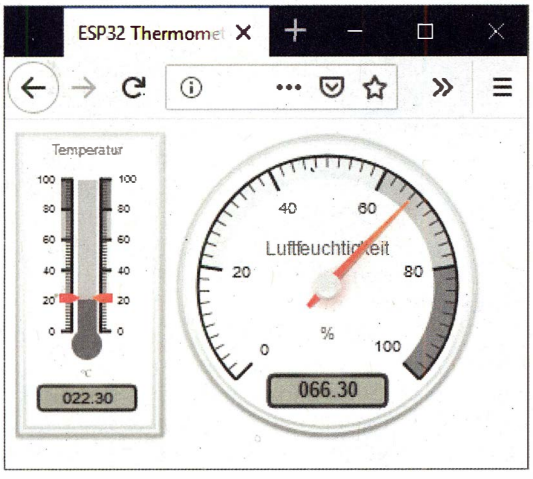
50 Türsensor mit Cloud-Anbindung

Durch die Kombination des integrierten Hallsensors mit externen Magneten wird der ESP32 zur Cloud-fähigen Fernüberwachung von Türen und Fenstern.



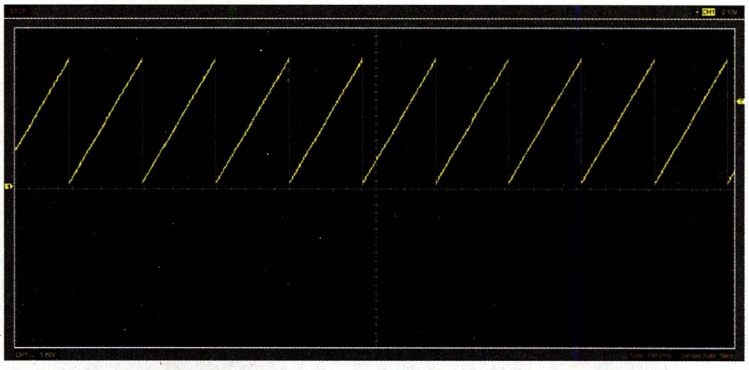
56 Wackelpudding-Piano

Die Touch-Eingänge des ESP32 machen den Anschluss teurer Tasten überflüssig. Acht Puddingbecher dienen als Tastatur für das Jelly-Piano, dessen Töne der Mikrocontroller intern erzeugt.



18 ESP32 als WLAN-Server

Mit wenigen Zeilen Code wird der ESP32 zum Webserver, der per WLAN die aktuelle Temperatur und Luftfeuchtigkeit im Browser anzeigt.



44 Funktionsgenerator

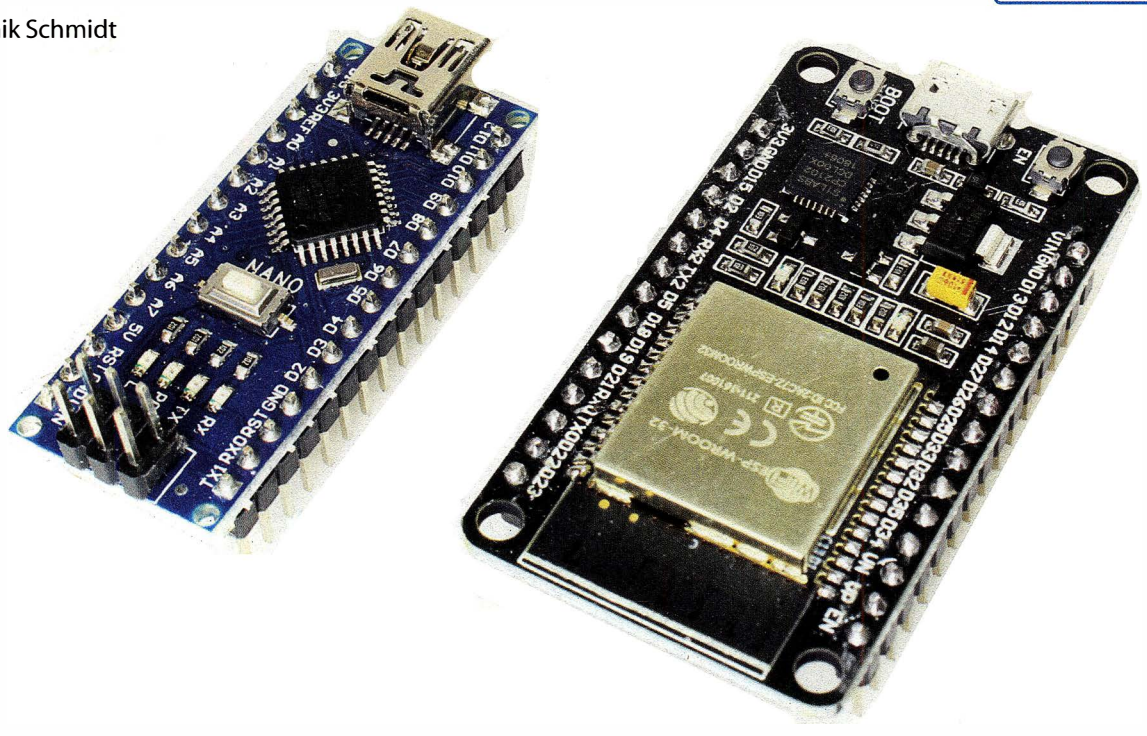
Dank seiner zwei Digital-/Analog-Konverter kann der ESP32 Sinus-, Dreieck-, Sägezahn- und Rechtecksignale bis in den Kilohertz-Bereich erzeugen, die sich für Experimente nutzen lassen.

Das Beste zweier Welten

Wer auf den ESP32 setzt, kommt nicht nur in den Genuss leistungsstarker und günstiger Hardware, sondern kann auch von der intuitiven Arduino-Entwicklungsumgebung profitieren.

Alles zum Artikel im Web unter *make-magazin.de/xrtn*

von Maik Schmidt



Zur Programmierung des ESP32 und des ESP8266 gibt es viele Alternativen. Neben der Entwicklung mittels C/C++ auf Basis des IoT Development Framework (ESP-IDF) gibt es auch die Möglichkeit, den Chip in Lua, JavaScript, MicroPython oder gar BASIC zu programmieren.

Sonderlich einsteigerfreundlich sind diese Kandidaten aus verschiedenen Gründen alle nicht und zum Erfolg des ESP32 und des ESP8266 hat maßgeblich beigetragen, dass sich beide mit der Arduino-IDE programmieren lassen. Die hat sich über die letzten Jahre zu einem fast konkurrenzlosen Werkzeug für Elektronik-Einsteiger entwickelt und erfreut sich auch unter ESP32-Fans wachsender Beliebtheit.

Prinzipiell kann die Arduino-IDE schon lange mit einer ganzen Reihe unterschiedlicher Boards umgehen, aber von Haus aus erkennt sie die ESP32-Boards nicht. Das lässt sich aber mit wenigen Handgriffen ändern.

Andocken

Bevor ein ESP32-Board mit der Arduino-IDE verbunden wird, muss es an den PC oder den Mac angeschlossen werden. Das erfolgt über einen USB-Anschluss, der das Board mit Strom und neuer Software versorgen kann. Wie alle USB-Geräte benötigen die Boards einen Betriebssystem-Treiber. In diesem Fall für den CP2102-Chip des Herstellers SiLab, der die serielle Kommunikation erledigt.

Windows 10, Linux und aktuelle Versionen von macOS bringen den Treiber in der Regel bereits mit. Auf der Seite des Herstellers (https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers) gibt es ihn bei Bedarf aber auch als kostenlosen Download.

Wenn man das ESP32-Board an einen Windows-Rechner mit dem richtigen Treiber anschließt, sollte im Geräte-Manager ein neuer Eintrag unter dem Punkt Anschlüsse erscheinen.

Unter Linux kann man im Terminal prüfen, ob das Board erkannt wurde. Dazu gibt man, bevor das Board angeschlossen wurde, das Kommando `ls /dev/tty.*` ein. Dann schließt man das Board an und führt das Kommando erneut aus. Der Eintrag, der neu hinzugekommen ist, verweist auf das ESP32-Board. Unter Ubuntu kann das ein Eintrag wie zum Beispiel `/dev/tty/tty-USB0` sein.

Ähnlich läuft es unter macOS. Nur gibt man hier das Kommando `ls /dev/cu.*` ein, bevor und nachdem man das Board angestöpselt hat. Typischerweise heißt der neue Eintrag `/dev/cu.SLAB_USBtoUART`.

Her mit der IDE!

Zur Installation der Arduino-IDE bietet die Webseite des Arduino-Projekts (https://www.arduino.cc/en/Main/Software) je nach Betriebssystem unterschiedliche Möglichkeiten. Für Windows steht zum Bei-

spiel neben einem klassischen Setup auch eine ZIP-Datei zur Verfügung. Die muss man lediglich auspacken, ohne die Anwendung zu installieren. Darüber hinaus ist die IDE kostenlos im Microsoft App Store (https://www.microsoft.com/de-de/p/arduino-ide/9nblggh4rsd8) erhältlich. Für macOS und Linux stehen auf der Arduino-Seite ebenfalls Installationspakete bereit.

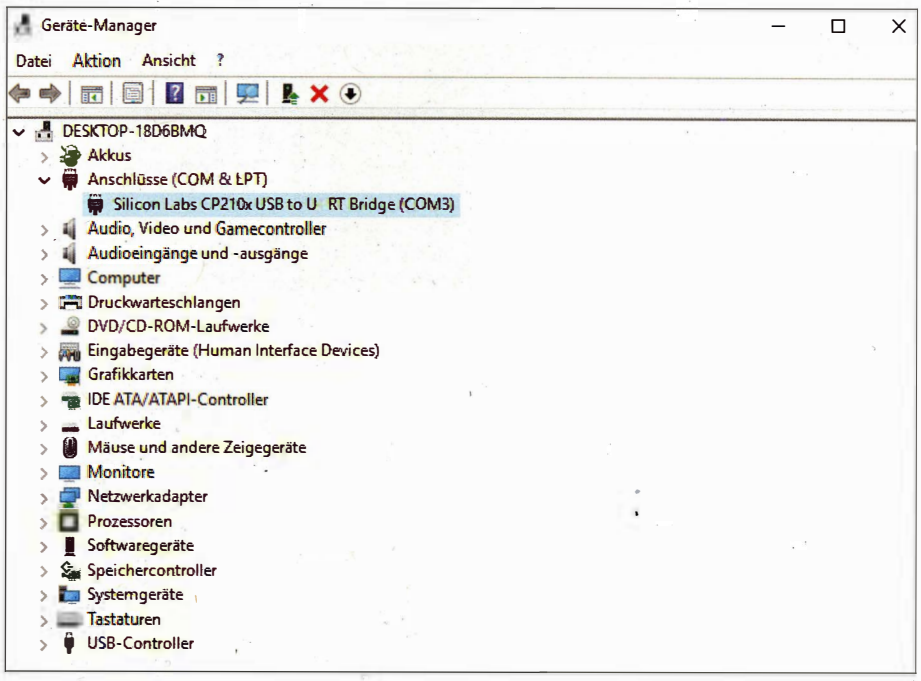
Der erste Start der Arduino-IDE ist nicht sonderlich spektakulär, denn die Software wirkt etwas schlicht. Den oberen Bereich dominieren eine Menüleiste mit sechs großen Knöpfen und ein Text-Editor. Im unteren Bereich gibt es einen Bereich für Ausgaben der IDE.

Die sechs Knöpfe bieten schnellen Zugriff auf die wichtigsten Kommandos. Ganz links steht der Knopf zur Überprüfung des aktuellen Programms. Wird er betätigt, werden die Quelltexte, die im aktuellen Fenster geöffnet sind, übersetzt und auf Syntaxfehler überprüft. Etwaige Fehler werden im Text-Editor gekennzeichnet und im unteren Ausgabefenster vermerkt.

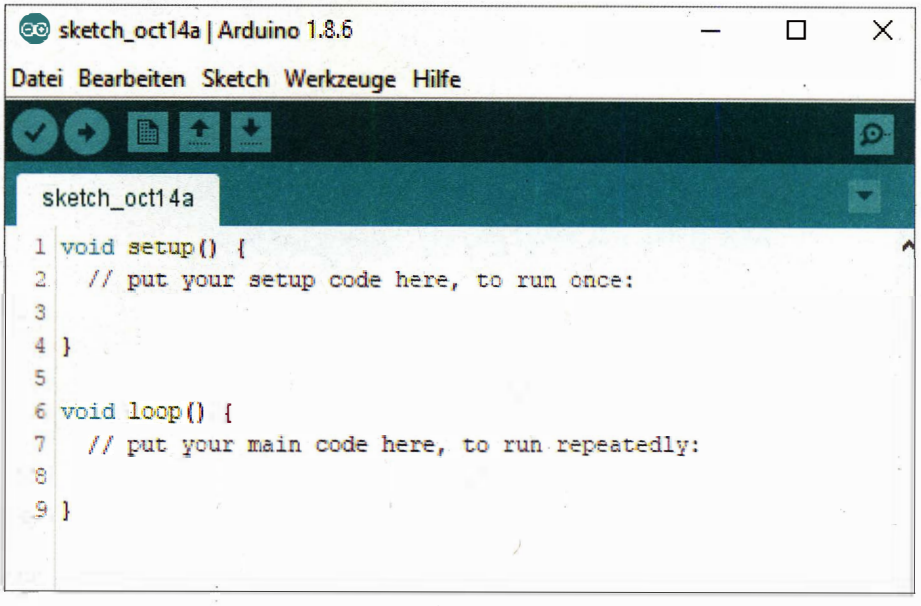
Eine ähnliche Funktion bietet der zweite Knopf. Allerdings übersetzt er das aktuelle Programm nicht nur, sondern überträgt es auch auf das angeschlossene Mikrocontroller-Board. Selbstverständlich nur, wenn das Programm keine Syntaxfehler enthält und das Board korrekt angeschlossen und konfiguriert ist.

Weniger spektakulär sind die nächsten drei Knöpfe. Mit ihnen kann man neue Programme anlegen, bestehende Programme öffnen und das aktuelle Programm speichern.

Schließlich gibt es ganz rechts noch einen vermeintlichen Außenseiter, der aber zu den wichtigsten Funktionen der IDE gehört. Hinter ihm verbirgt sich nämlich ein serieller Monitor, mit dem man bequem mit dem angeschlossenen Board kommunizieren kann. Zwar gibt es einige gute und frei verfügbare serielle Terminals, aber die Integration in die IDE vereinfacht die tägliche Arbeit enorm.



Wurde der Treiber erfolgreich installiert, erscheint der ESP32 so im Geräte-Manager.

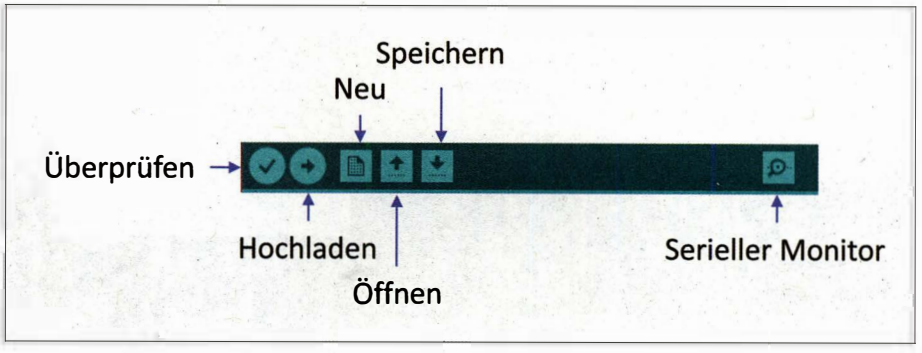


Die Arduino-IDE wirkt spartanisch, kann aber eine ganze Menge.

Kein ESP32 weit und breit

Bevor man ein erstes Programm schreiben und aufs ESP32-Board laden kann, muss das Board in der IDE konfiguriert werden. Die Arduino-IDE unterstützt bereits eine große Anzahl unterschiedlicher Boards, die im Menü Werkzeuge > Board ausgewählt werden können. Ein erster Blick sorgt erst einmal für Ernüchterung, denn ein ESP32-Board ist hier nirgendwo zu sehen. Offenbar unterstützt die Standard-Installation längst nicht alle möglichen Produkte.

Viele Boards lassen sich mittlerweile über den Boardverwalter im Menü Werkzeuge > Boardverwalter installieren, aber der ESP32



Für die wichtigsten Kommandos gibt es große Knöpfe.

gehört per Voreinstellung noch nicht dazu. Um ihn zu integrieren, bedarf es ein wenig Handarbeit. Dabei hilft es zu verstehen, wie die Unterstützung verschiedener Boards in der Arduino-IDE funktioniert.

In den Anfängen des Arduino-Projekts konnte niemand dessen Erfolg vorausahnen und so verlief die Entwicklung zunächst recht hemdsärmlig. Wichtig war in erster Linie, dass alles irgendwie funktionierte und über etwaige Erweiterungen dachte niemand nach.

Dieser Ansatz lief lange Zeit gut, stieß aber an Grenzen, als nicht mehr alle Arduino-Boards auf derselben Mikrocontroller-Architektur basierten. Neben den AVR-Chips mussten plötzlich auch ARM-Geräte mit der IDE funktionieren.

Zu diesem Zeitpunkt wurde die IDE um die Möglichkeit erweitert, neue Hardware ohne Code-Änderungen hinzuzufügen. Diesmal achteten die Entwickler auf Erweiterbarkeit, sodass sogar Boards unterstützt werden können, die nicht vom Arduino-Team stammen.

Davon machen mittlerweile einige Hersteller Gebrauch und die Firma Espressif, die den ESP32 entwickelt hat, bildet keine Ausnahme. Auf Github (https://github.com/espressif/arduino-esp32) hat sie zu diesem Zweck das Projekt „Arduino Core for ESP32" veröffentlicht.

Fremdgänger

Der Code auf Github nützt nicht viel, solange er der Arduino-IDE nicht hinzugefügt wurde. Die einfachste Möglichkeit, Unterstützung für neue Boards zu installieren, ist der Boardverwalter der IDE. Der sucht per Voreinstellung die Liste der unterstützten Boards in den offiziellen Arduino-Quellen, aber die enthalten zurzeit den ESP32 noch nicht. Wie den meisten Paket-Managern kann man dem Boardverwalter aber zusätzliche Quellen für Board-Konfigurationen hinzufügen.

Die zusätzlichen Quellen werden über das Menü Datei > Voreinstellungen festgelegt. Für den ESP32 fügt man hier unter dem Punkt „Zusätzliche Boardverwalter-URLS" die URL https://dl.espressif.com/dl/package_esp32_index.json hinzu. Bei Bedarf lassen sich auch weitere Quellen hinzufügen. Die werden jeweils durch ein Komma getrennt.

Wenn man den Dialog für Voreinstellungen ohnehin schon geöffnet hat, ist es eine gute Idee, die Kontrollkästchen für die ausführliche Ausgabe bei der Kompilierung und beim Hochladen zu aktivieren. Das wird sich im weiteren Verlauf noch als nützlich erweisen.

Nachdem die neue URL hinzugefügt wurde, erscheint bei der Suche nach dem ESP32 im Boardverwalter ein neuer Eintrag. Ein Klick auf den Installieren-Knopf lädt die passende Konfiguration herunter und installiert sie sogleich.

Es lebt!

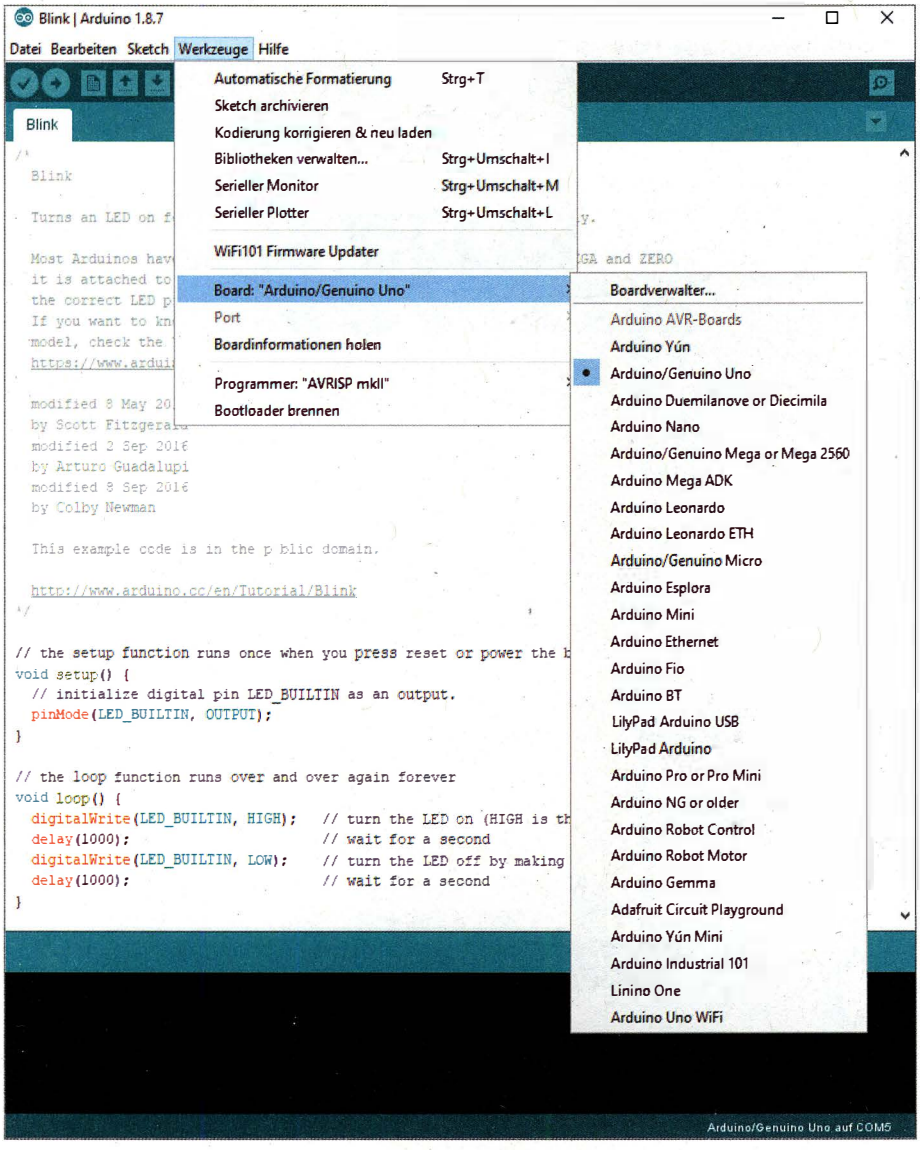
Der obligatorische Funktionstest für jedes Mikrocontroller-Board ist es, eine LED zum Blinken zu bringen. Eine solche Anwendung stiftet zwar nur wenig Sinn, stellt aber sicher, dass die gesamte Entwicklungsumgebung so funktioniert, wie sie soll.

Die meisten Boards verfügen zumindest über eine Power-LED. Die lässt sich in der Regel aber nicht per Software kontrollieren und leuchtet nur dann, wenn das Board mit Strom versorgt wird. Darüber hinaus spendieren viele Hersteller ihren Produkten mindestens eine weitere LED, die sich in eigenen Programmen beliebig verwenden lässt.

Beim mitgelieferten Board ist das der Fall und Listing Blink.ino lässt die eingebaute Status-LED im Sekundentakt blinken.

Für einen ersten Test muss das Board der Arduino-IDE bekannt gemacht werden. Dazu wählt man im Menü Werkzeuge > Board den Eintrag „ESP32 Dev Module". Anschließend wählt man unter Werkzeuge > Port den seriellen Port, über den das ESP32-Board mit dem Rechner verbunden ist. Schließlich muss man noch die „Flash Frequency" auf 80MHz und „Upload Speed" auf 115200 setzen. Diese und alle weiteren Parameter werden im folgenden noch ausführlich erklärt.

Ein Klick auf den Upload-Knopf übersetzt das Programm und überträgt es anschlie-



Ohne Hilfsmittel unterstützt die Arduino-IDE den ESP32 nicht.

ßend aufs ESP32-Board. Die Übertragung dauert ein paar Sekunden und den aktuellen Status zeigt das Ausgabefenster der Arduino-IDE an. Auf manchen ESP32-Boards zeigen separate LEDs Aktivität auf der seriellen Schnittstelle an. Das heißt, diese LEDs blinken beim Senden und Empfangen von Daten. Das mitgelieferte Board hat solche LEDs nicht. Nachdem das Programm übertragen wurde, sollte aber die blaue Status-LED im Sekundentakt blinken.

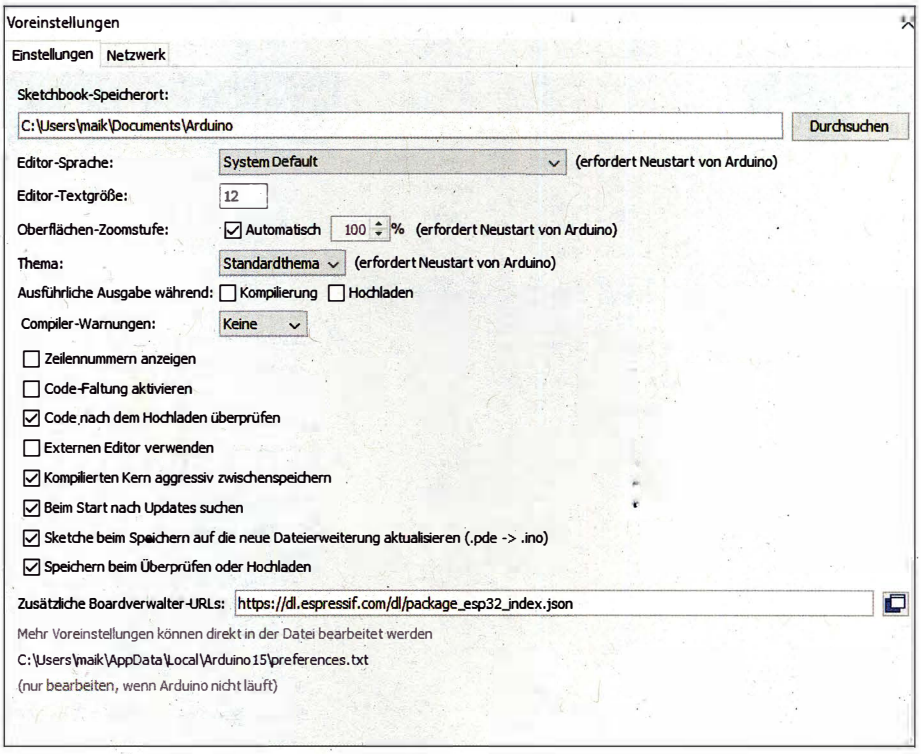
Das Blink-Programm ist einfach aufgebaut. Im Wesentlichen besteht es aus den Funktionen `setup()` und `loop()`. Das ist eine Konvention, der alle Arduino-Programme gehorchen. Die `setup()`-Funktion wird einmalig aufgerufen, wenn der Mikrocontroller startet und ist der perfekte Ort, um Initialisierungen vorzunehmen. Die `loop()`-Funktion enthält die tatsächliche Programm-Logik und wird vom Mikrocontroller in einer Endlosschleife immer wieder aufgerufen.

Vor den beiden Funktionen definiert das Blink-Programm eine Konstante namens `LED_PIN`, welche die Nummer des Pins enthält, der mit der Status-LED des Boards verbunden ist. Beim mitgelieferten Board ist das der Pin mit der Nummer 2. Bei anderen Boards ist es mit hoher Wahrscheinlichkeit ein anderer Pin und manche Geräte haben überhaupt keine Status-LED. Im Zweifel hilft ein Blick ins Datenblatt des Herstellers.

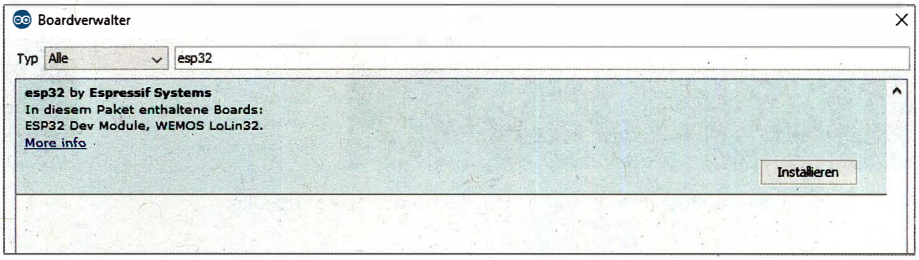
Die `setup()`-Funktion nutzt die `pinMode()`-Funktion, um den LED-Pin in den Ausgabemodus zu schalten. Die `loop()`-Funktion versetzt anschließend mittels `digitalWrite()` den Pin in den Zustand `HIGH`. Dadurch liegen am Pin 3,3V an und die LED leuchtet. Die Funktion `delay()` wartet 1000 Millisekunden und ein weiterer Aufruf von `digitalWrite()` mit dem Argument `LOW` schaltet die LED wieder aus. Nach einer weiteren Pause von einer Sekunde beginnt das Spiel von vorn, weil die `loop()`-Funktion automatisch wieder aufgerufen wird. Damit blinkt die LED im Sekundentakt.

Beim Code ist zu beachten, dass Funktionen, wie zum Beispiel `setup()`, `pinMode()` oder `loop()` durch die Arduino-Umgebung definiert werden. Das gilt auch für das Verhalten, also zum Beispiel dafür, dass die `setup()`-Funktion automatisch beim Start des Mikrocontrollers aufgerufen wird. In anderen Entwicklungsumgebungen würde ein Programm, das eine LED zum Blinken bringt, gänzlich anders aussehen.

Leider ist die Arduino-Umgebung nicht vollständig kompatibel mit allen ESP32-Boards und wird es vermutlich auch nie sein. Beispielsweise definiert die Umgebung eine Konstante namens `LED_BUILTIN`, die die Pin-Nummer der eingebauten LED auf den Arduino-Boards enthält. Für viele ESP32-Boards ist diese Konstante nicht definiert und das wäre auch schwierig, weil der Wert für unterschiedliche ESP32-Boards unterschiedlich sein kann. Dieses Problem lässt sich leicht umschiffen und für spezielle Boards sogar lösen. Es gibt aber auch schwerwiegendere Mängel. Beispielsweise ist die Funktion `analogWrite()` zurzeit noch nicht implementiert.



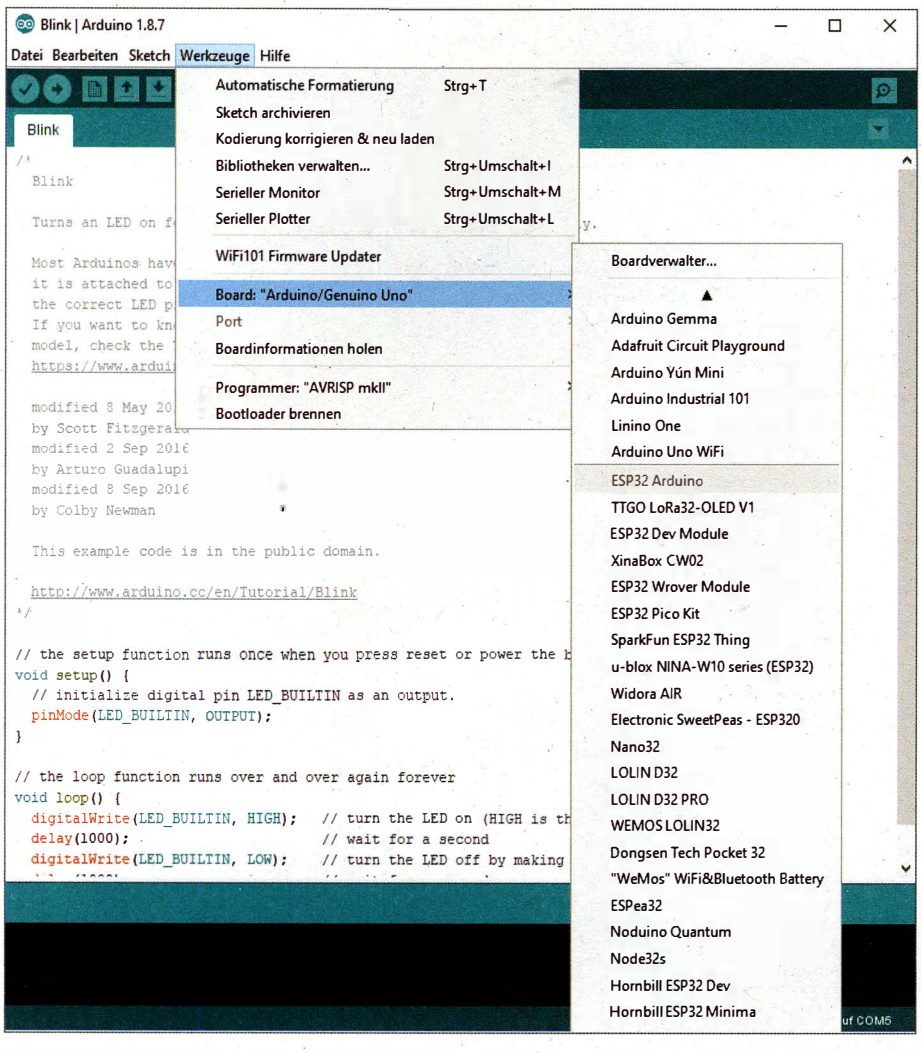
Das Hinzufügen neuer Quellen für den Boardverwalter ist einfach.



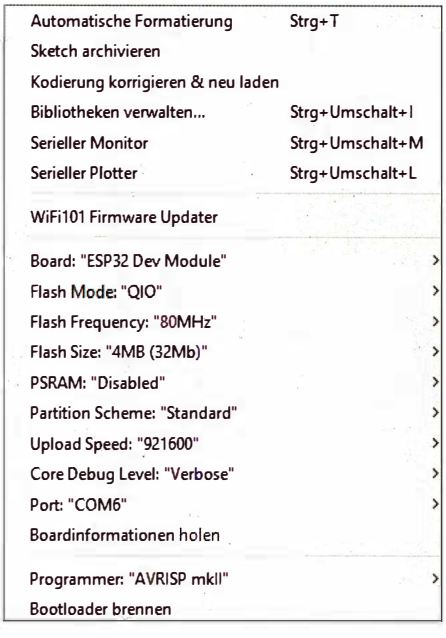
Den ESP32 kann man leicht über den Boardverwalter installieren.

Blink.ino

```
1 const uint8_t LED_PIN = 2;
2
3 void setup() {
4   pinMode(LED_PIN, OUTPUT);
5 }
6
7 void loop() {
8   digitalWrite(LED_PIN, HIGH);
9   delay(1000);
10  digitalWrite(LED_PIN, LOW);
11  delay(1000);
12 }
```



Nach der Installation stehen viele verschiedene ESP32-Boards zur Verfügung.



Für viele ESP32-Boards kennt das Werkzeuge-Menü viele Einstellungen.

Viele Stellschrauben

Ganz egal, auf welche Weise die ESP32-Unterstützung installiert wurde, am Ende kann man über das Menü Werkzeuge > Board aus einer Fülle von ESP32-Boards wählen. Für das mitgelieferte Board ist „ESP32 Dev Module" die richtige Wahl, weil dieser Board-Typ einige Freiheitsgrade lässt und somit an eine Vielzahl von ESP32-Boards angepasst werden kann.

Bei den ESP32-Boards herrscht nämlich ein ziemlicher Wildwuchs und so funktioniert die Konfiguration mit der Bezeichnung „ESP32 Dev Module" prinzipiell für einen ganzen Zoo von Geräten. Wie schon zuvor bemerkt, haben zum Beispiel nicht alle Boards eine Status-LED und falls sie doch eine haben, ist sie nicht immer mit demselben Pin verbunden. Gravierendere Unterschiede gibt es in anderen Bereichen und so finden sich in unterschiedlichen Produkten verschiedene Ausprägungen von Flash-Speichern. Die kommen nicht nur mit unterschiedlichen Speichergrößen daher, sondern variieren unter anderem auch in der Lese- und Schreibgeschwindigkeit. Daher bietet die Arduino-IDE Einstellmöglichkeiten für die wichtigsten Flash-Parameter.

„Flash mode" legt die Geschwindigkeit des Flash fest. Genauer gesagt legt diese Option fest, über wie viele Leitungen und auf welche Art der ESP32 mit dem Flash per SPI kommuniziert. Anders als etwa beim Arduino ist der Flash-Speicher zur Aufnahme der Programme nicht im Mikrocontroller-Chip enthalten, sodass die Hersteller einen Extra-Chip extern anschließen und ihn ansteuern müssen.

Der schnellste Modus zur Kommunikation mit dem externen Flash ist QIO (Quad I/O Fast Read) und knapp dahinter kommt QOUT (Quad Output Fast Read). An dritter Stelle steht DIO (Dual I/O Fast Read) und das Schlusslicht bildet DOUT (Dual Output Fast Read). Während die ersten beiden Verfahren vier SPI-Pins einsetzen, benötigen die letzten beiden nur zwei. Nicht alle Bausteine kommen mit allen Verfahren zurecht, aber das mitgelieferte Board versteht QIO.

Die Option „Flash frequency" spezifiziert die Taktfrequenz für Zugriffe auf das Flash. Mit 40MHz sollten die meisten Bausteine arbeiten können, aber 80MHz können zu einer deutlichen Beschleunigung führen. Auf Umgebungen, die mit 80MHz nicht arbeiten können, führt diese Einstellung schnell zu Abstürzen.

Schließlich definiert „Flash size" die Größe des externen Flash-Bausteins. Mögliche Größen für den ausgewählten Board-Typen sind 2MB oder 4MB (die Größen in Klammern geben die Kapazität in Megabit an). Ein zu groß gewählter Wert würde dafür sorgen, dass der Bootloader des ESP32 einen falschen Speicherbereich liest und das gespeicherte Programm würde nicht starten.

Für einige Anwendungen reicht selbst das vergleichsweise üppige RAM auf dem ESP32 nicht aus. Manche Boards haben daher zusätzliches PSRAM (pseudostatisches RAM), das man mit der entsprechenden Menüoption namens PSRAM aktivieren beziehungsweise deaktivieren kann.

Im Vergleich zum Hauptspeicher ist ein Flash oft recht groß und wie bei Festplatten ist es sinnvoll, den Speicher zu partitionieren, also aufzuteilen. Auf diese Weise kann das Flash parallel mehrere Anwendungen und die dazugehörigen Daten beherbergen. Prinzipiell kann die Partitionierung auf fast beliebige Weise erfolgen, aber die Arduino-IDE bietet unter der Option „Partition Scheme" die gängigsten Modelle an. Für erste Experimente ist der Wert „Standard" sinnvoll.

Wenig überraschend definiert die Option „Upload Speed", mit welcher Geschwindigkeit Daten von der Arduino-IDE zum Board übertragen werden. Nicht alle Boards kommen mit allen Übertragungsgeschwindigkeiten klar und es empfiehlt sich, mit 115.200 Baud zu beginnen.

Die letzte Option ist „Core Debug Level" und die steuert die Geschwätzigkeit von Log-Ausgaben auf dem Board. Sowohl während der Entwicklung als auch während des Betriebs einer Anwendung helfen Log-Ausgaben auf der seriellen Schnittstelle. Mit ihrer Hilfe lässt sich der aktuelle Zustand des Systems ermitteln oder ein Fehler einkreisen. Allerdings haben Log-Ausgaben ihren Preis und so ist es sinnvoll, während des regulären Betriebs deutlich weniger Log-Ausgaben über die serielle Schnittstelle zu senden, als zum Beispiel während einer Debugging-Sitzung.

Zu diesem Zweck können Log-Ausgaben mit einer Dringlichkeitsstufe (Level) versehen werden. Auf dem ESP32 gibt es – nach Dringlichkeit geordnet – die Stufen Error (Fehler), Warn (Warnung), Info (Information), Debug (Fehlersuche) und Verbose (wortreich). Listing LogBeispiel.ino zeigt die Verwendung der dazugehörigen Log-Funktionen.

Die `setup()`-Funktion startet die Kommunikation auf der seriellen Schnittstelle mit einer Geschwindigkeit von 115.200 Baud. In der `loop()`-Funktion erzeugen dann `log_e()`, `log_w()` und so weiter im Sekundentakt Nachrichten der verschiedenen Stufen. Per Voreinstellung werden allerdings keine Log-Ausgaben auf die serielle Schnittstelle gesendet. Davon kann man sich überzeugen, indem man das Programm aufs Board lädt, den seriellen Monitor startet und die Übertragungsgeschwindigkeit auf 115.200 Baud setzt. Die Ausgabe im seriellen Monitor bleibt leer.

Setzt man die Option „Core Debug Level" hingegen auf Warn und schiebt das Programm erneut aufs Board, werden die Ausgaben der Stufen Fehler und Warnung kontinuierlich ausgegeben. Neben dem Nachrichtentext enthält jede Zeile eine Abkürzung der Stufe (in diesem Fall „E" und „W"), die Nummer der Programmzeile, in der die Ausgabe erzeugt wurde und den Namen der Funktion, aus der die Ausgabe stammt. Das sind wertvolle Informationen bei einer Fehlersuche.

Setzt man schließlich „Core Debug Level" auf Verbose, werden nach einem erneuten Upload alle Meldungen ausgegeben.

Festzuhalten ist noch, dass all diese Parameter nicht für jedes ESP32-Board konfiguriert werden können. Die Einträge im Werkzeuge-Menü variieren daher von Board zu Board.

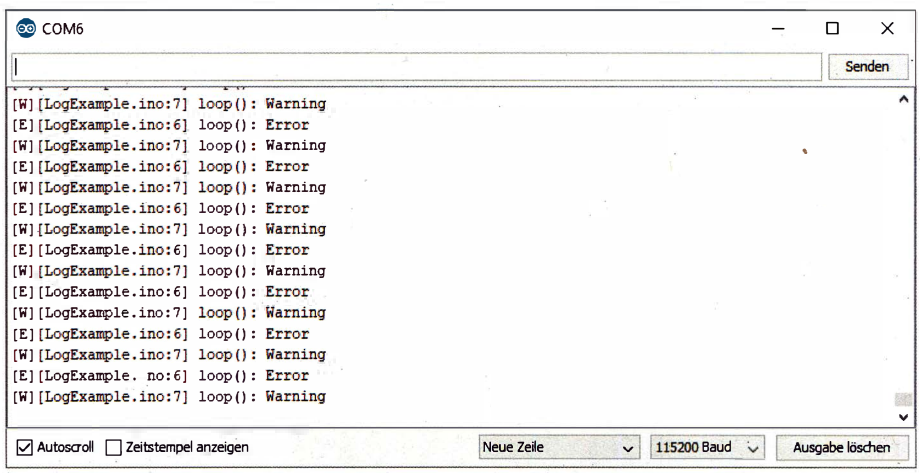
LogBeispiel.ino

```
void setup() {
  Serial.begin(115200L);
}

void loop() {
  log_e("Error");
  log_w("Warning");
  log_i("Info");
  log_d("Debug");
  log_v("Verbose");t
  delay(1000);
}
```

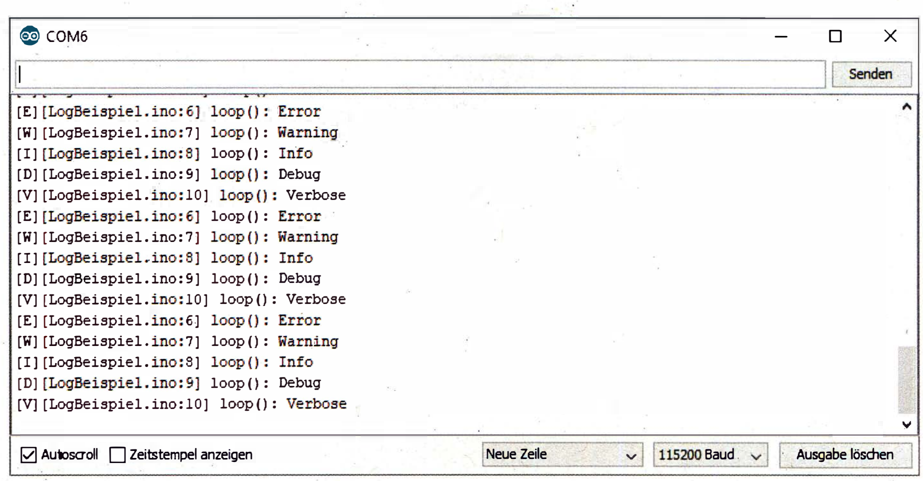
Fazit

Die Arduino-IDE ist längst nicht das einzige Werkzeug, mit dem Entwickler Software für den ESP32 erstellen können und auf den ersten Blick hat sie einige Unzulänglichkeiten. Sie bietet weder Zugang zu allen originären ESP32-Funktionen noch zu allen des Arduino-Universums. Darüber hinaus hat sie auch keine Features wie eine automatische Code-Komplettierung oder einen Debugger.

Trotzdem – oder gerade deswegen – ist sie für erste Experimente und kleinere Projekte optimal. Sie überfrachtet und überfordert den Einsteiger nicht mit Funktionen, die sich erst mit mehr Erfahrung als nützlich erweisen. Ferner nimmt sie ihren Anwendern genau die Handgriffe ab, die bei Anfängern erfahrungsgemäß zu viel Frustration führen. *—dab*



So werden nur Warnungen und Fehler ausgegeben.



Hier sind alle Log-Ausgaben zu sehen.

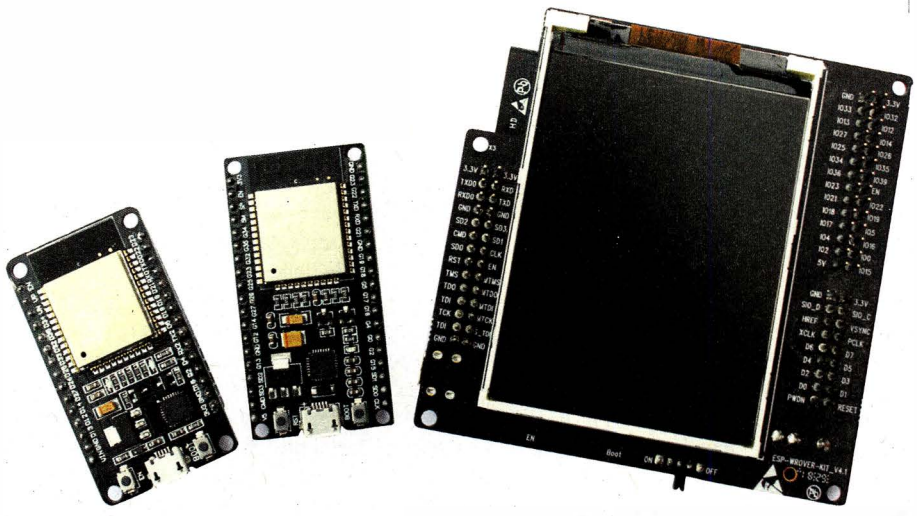
Hardware-Interna

Für ein paar Euros bietet der ESP32 Konnektivität und Rechenleistung satt. Auf dem Chip gibt es aber noch viel mehr Nützliches zu entdecken und wer das Maximum aus dem ESP32 herausholen möchte, sollte alle seine Eigenschaften kennen.

von Maik Schmidt

Alles zum Artikel im Web unter
make-magazin.de/xnxr

Die Hardware des ESP32 bietet bisher ungeahnte Leistung zu einem sehr niedrigen Preis. Auf der Fläche eines Daumennagels tummeln sich neben WLAN und Bluetooth diverse Sensoren und das Gerät hat jede Menge Schnittstellen, oft sogar in mehrfacher Ausführung. Vor dem Start eines neuen Projekts lohnt es sich daher, einen genauen Blick auf den Chip und sein Ökosystem zu werfen.



Mittlerweile gibt es eine Vielzahl unterschiedlicher Development-Boards für den ESP32.

Artenvielfalt

Entwickelt wurde der ESP32 von Espressif Systems in China (Shanghai), derselben Firma, die den ESP8266 entwickelt hat. In großen Stückzahlen ist er seit Ende 2016 verfügbar und seitdem erfreut er sich wachsender Beliebtheit unter Elektronik-Bastlern. Weil der ESP32 noch verhältnismäßig neu ist, wurden noch nicht alle seine Fähigkeiten in aller Tiefe dokumentiert und zu manchem Thema findet sich selbst nach längerer Recherche nicht allzu viel. Das gilt ebenso für die ESP32-Software, denn auch die befindet sich noch im Fluss.

Wichtig zu verstehen ist zuerst, dass es nicht „den" ESP32 gibt, sondern dass unterschiedliche Ausprägungen existieren. Im Wesentlichen unterscheiden sie sich in der Anzahl der Prozessor-Kerne, der Größe des eingebauten Flash-Speichers und in der Gehäuse-Form. Sie haben so wenig klangvolle Namen wie ESP32-DOWDQ6, ESP32-D0WD, ESP32-D2WD, ESP32-S0WD.

Die Bezeichnungen wirken zunächst kryptisch, folgen aber einem einfachen Schema. Der erste Buchstabe spezifiziert die Anzahl der Prozessor-Kerne. D steht für Dual-Core und S für Single-Core. Die darauf folgende Zahl gibt an, wie groß der interne Flash-Speicher des Bausteins ist. 0 bedeutet, dass er keinen hat und die 2 steht für 2MB.

Weiter geht es mit einer Kennung für die Netzwerk-Fähigkeiten des Chips. Die wird mit zwei Buchstaben kodiert und kann die folgenden Werte annehmen:

WD (Wi-Fi b/g/n, BT/BLE Dual Mode), AD (Wi-Fi a/b/g/n, BT/BLE Dual Mode) und CD (Wi-Fi ac/c/b/n/g, BT/BLE Dual Mode).

In Bezug auf die Bluetooth-Fähigkeiten sind die Produkte gleich, aber beim WLAN gibt es durchaus Unterschiede.

Schließlich kommen die Chips noch in verschiedenen Gehäusen daher und die kann man an den letzten beiden Stellen der Produktkennung ablesen. Steht hier nichts, dann handelt es sich um einen Chip mit den Abmessungen 5mm × 5mm in der QFN-Bauform (Quad-Flat No-Leads Package). Der Wert Q6 steht hingegen für 6mm × 6mm, ebenfalls in der Bauform QFN.

Die Basis-Chips des ESP32 eignen sich nicht für den Heimgebrauch. Sie sind eher etwas für die industrielle Fertigung. Darüber hinaus fehlen ihnen trotz ihrer vielen Funktionen noch ein paar wichtige Eigenschaften. Beispielsweise haben sie fast alle nur wenig beziehungsweise gar keinen Flash-Speicher.

Deshalb gibt es Module, die auf den Basis-Chips aufbauen und sie um nützliche Hardware erweitern. Ein gutes Beispiel dafür ist der ESP32-WROOM-32. Er basiert auf dem ESP32-D0WDQ6 und spendiert ihm unter anderem 4MB Flash-Speicher.

Doch selbst mit den „nackten" ESP32-Modulen dürften die meisten Hobbyisten nicht viel anfangen können. Sie sind sehr klein und lassen sich nicht ohne Weiteres mit Drähten oder einem Breadboard verbinden. Daher dürfte die Mehrheit der Bastler zu einem Development-Board wie dem mitgelieferten greifen.

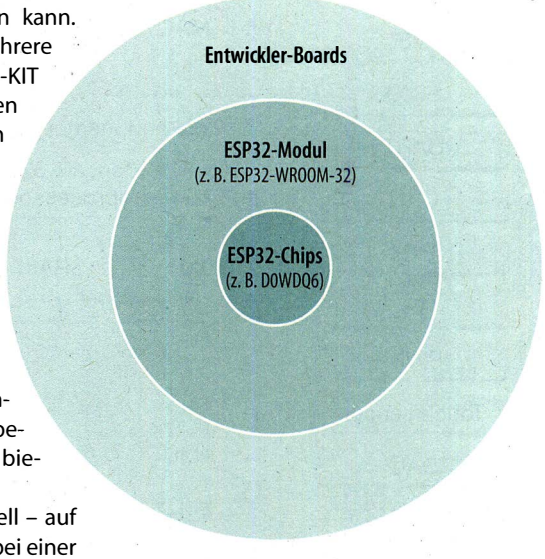
Development-Boards gibt es in vielen Ausprägungen und sie unterscheiden sich teilweise deutlich voneinander. Die einfachsten führen lediglich alle Pins des ESP32 nach außen, so dass das Gerät problemlos mit anderen Bauteilen verbunden werden kann. Meist haben sie noch eine oder mehrere LEDs. Boards wie das ESP-WROVER-KIT verfügen sogar über einen kleinen Bildschirm, einen Slot für SD-Karten und einen Kamera-Anschluss.

Doppelher(t)z

Trotz dieser Vielfalt ticken die Boards intern alle gleich und haben einen oder zwei Prozessor-Kerne. Jeder Kern ist eine 32-Bit-CPU Xtensa LX6, die neben einer Fließkomma-Einheit auch Unterstützung für DSP-Operationen (Digital Signal Processing) bietet.

Die CPU bringt es – je nach Modell – auf 200 bis 600DMIPS (Dhrystone MIPS) bei einer Taktfrequenz zwischen 80MHz und 240MHz. Die Module unterscheiden sich in ihrer Geschwindigkeit also durchaus, weil zum Beispiel der ESP32-S0WD und der ESP32-D2WD nur mit höchstens 160MHz gegetaktet werden können. Prinzipiell ist die Taktrate bei allen Modulen per Software bis zur maximalen Taktfrequenz einstellbar. In der Praxis ist aber über den ESP32-Core für Arduino die Frequenz in den Bibliotheken auf 240MHz (bzw. 160MHz) fest eingestellt. Eine Anleitung, wie man die Frequenz anpasst, finden Sie unter den Links.

In der Brust der meisten Modelle schlagen also gleich zwei Herzen, das heißt, zwei CPU-Kerne. In der Dokumentation tragen die beiden Kerne die Namen Protocol CPU (PRO_CPU) und Application CPU (APP_CPU). Der Kern namens PRO_CPU ist verantwortlich für die gesamte Kommunikation, also für WLAN und Bluetooth, aber auch für andere Kommunikationssysteme, wie zum Beispiel I2C und SPI. Um den Code der Anwendungen kümmert sich der APP_CPU-Kern. Obwohl prinzipiell möglich, ist es nicht ganz



Module basieren in der Regel auf Chips und Entwickler-Boards.

einfach, die beiden Kerne parallel in eigenen Programmen zu nutzen.

In der CPU gibt es vier Timer mit jeweils 64 Bit, die sich auf vielfältige Art und Weise nutzen lassen. Sie eignen sich zum Rauf- und Runterzählen und haben einen Prescaler, um die Frequenz des Timers vom CPU-Takt zu entkoppeln. Über diverse Interrupts lassen sich die Timer einfach in eigenen Programmen nutzen.

Dazu gibt es eine Echtzeituhr (RTC, Real-Time Clock), die nicht nur zur Ermittlung der Zeit dienen kann, sondern auch eine wichtige Rolle bei den Stromspar-Funktionen spielt. Allerdings wird diese Uhr in der Regel nicht durch eine separate Batterie versorgt.

Platz satt

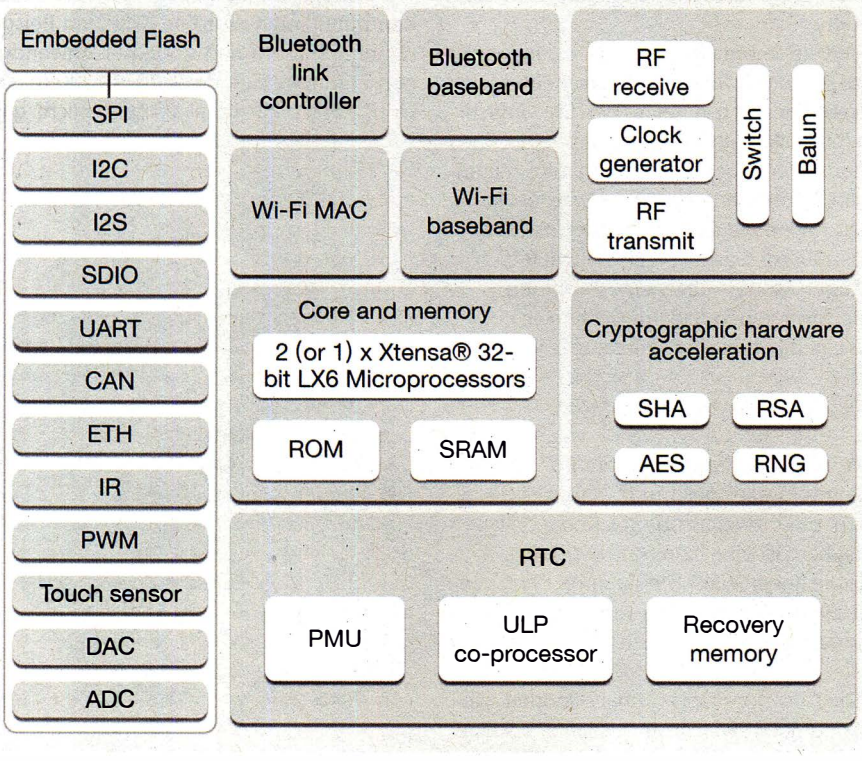
Über den Chip verteilt hat der ESP32 jede Menge Speicher. 448KB ROM (Read Only Memory) enthalten grundlegende Funktionen, die sich unter anderem um die drahtlose Kommunikation und um die Verschlüsselung kümmern. Hier finden sich auch exotischere Dinge, wie zum Beispiel ein kompletter BASIC-Interpreter.

Programme und Daten teilen sich während der Ausführung 520KB SRAM (Static Random Access Memory). Werte wie 448KB und 520KB klingen zunächst ungewöhnlich. Sie resultieren aus der Art und Weise, in der der Speicher aufgeteilt wird. Das ROM zerfällt zum Beispiel in zwei Adressbereiche mit jeweils 384KB und 64KB. Beim SRAM sind es sogar drei mit 192KB, 128KB und 200KB. In der Programmierung spielen diese Interna allerdings keine große Rolle.

Neben dem üblichen RAM und ROM existieren auf dem Chip noch weitere Speicherbausteine. Der Echtzeituhr stehen zum Beispiel gleich zwei mit jeweils 8KB zur Verfügung.

Darüber hinaus gibt es noch 1KB, also 1024 Bits, eFuse-Speicher, von dem 256 Bits für das System reserviert sind. Mit dieser besonderen Art des Speichers ist es möglich, Informationen persistent zu speichern, also über Ausschaltvorgänge hinweg. Der Name Fuse steht im Englischen für Sicherung. Jedes Bit im eFuse steht für eine solche Sicherung und die kann man wortwörtlich durchbrennen lassen. Sobald eine solche Sicherung durchgebrannt ist, verwandelt sich das dazugehörige Bit in nicht mehr löschbaren ROM.

Dieser Mechanismus eignet sich unter anderem dazu, einem Chip eine eindeutige Kennung einzubrennen. Allerdings muss man bei der Verwendung von eFuse Vorsicht walten lassen, denn eine Schreiboperation lässt sich nicht rückgängig machen. Hobby-Programmierer werden eher selten auf eFuse-Speicher zugreifen, aber für kommerzielle Produkte ist dieses Feature sehr wichtig. Es hat unter anderem eine wichtige Rolle bei der Verschlüsselung von Speicherinhalten.



Auf dem ESP-SOC finden sich erstaunlich viele nützliche Funktionen.

Der ESP32 spielt, was den Speicher angeht, in einer anderen Liga als der ESP8266 oder die meisten Arduino-Modelle. Der ATMega328P, der den Arduino Uno antreibt, bringt es zum Beispiel nur auf 2KB SRAM und selbst der Arduino Due kommt mit gerade einmal 96KB daher. Bei solchen Vergleichen darf man natürlich nicht vergessen, dass die verschiedenen Produkte nicht auf derselben Prozessor-Architektur basieren. Beispielsweise operiert der ATMega328P auf 8Bit während es beim Due und beim ESP32 32Bit sind.

Auch gegenüber dem ESP8266 hat der ESP32 die Nase vorn, denn der Vorgänger musste sich mit 32 KB RAM für den Programm-Code und 80 KB für die Programm-Daten begnügen.

Es reicht trotzdem nicht

Für einen Mikrocontroller wirkt ein Hauptspeicher mit 520KB schon recht üppig, aber für manche Anwendungen ist selbst das nicht genug. Erfreulicherweise lässt sich der ESP32 bei Bedarf mit PSRAM (Pseudostatic Random Access Memory) erweitern. PSRAM ist im Grunde DRAM (Dynamic Random Access Memory), das so tut, als wäre es SRAM (Static Random Access Memory). Das größte Problem mit DRAM ist, dass diese Form des Speichers unentwegt aufgefrischt werden muss, damit keine Informationen verloren gehen. Bei SRAM ist dieses Auffrischen nicht notwendig, erfordert aber mehr Logik und damit Platz auf einem Chip. PSRAM liegt genau dazwischen und ist im Wesentlichen nichts anderes als DRAM mit ein wenig Zusatzhardware, die das automatische Auffrischen übernimmt.

Der ESP32 integriert PSRAM nahtlos in seinen Adressraum, aber es unterliegt einigen Einschränkungen. Zum Beispiel können Anwendungen nicht aufs PRAM zugreifen, während der Flash-Speicher beschrieben wird beziehungsweise während der Cache des Flash-Speichers ausgeschaltet ist. Das PRAM kann auch nicht für DMA-Zugriffe (Direct Memory Access) verwendet werden und es gibt einige Situationen, in denen das PRAM deutlich langsamer ist als das SRAM. Trotzdem kann PRAM für spezielle Anwendungen genau die richtige Lösung sein und mit ein wenig Planung, lassen sich die Restriktionen oft umschiffen.

Immer noch nicht genug

Abgesehen vom eFuse haben die bisher vorgestellten Speicher allesamt gemeinsam, dass sie nicht persistent sind. Das heißt, sie überdauern selbst eine kurzzeitige Unterbrechung der Stromzufuhr nicht. Darüber hinaus wären sie für so manche Anwendung auch immer noch zu klein. Abhilfe schafft hier nur ein Flash-Speicher.

Während Flash zum Beispiel auf allen Arduinos obligatorisch ist, ist es auf dem ESP32 optional. Für industrielle Anwendungen kann das durchaus Sinn stiften, aber typische Heim-Anwendungen kommen ohne dauerhaften Speicher kaum aus. Daher haben die meisten Entwickler-Boards mindestens 2MB oder 4MB Flash-Speicher.

Der zusätzliche Speicher muss an den ESP32 angeschlossen werden. Das geschieht via SPI (Serial Peripheral Interface) und abhängig vom verwendeten Flash-Baustein werden dazu entweder vier oder sechs Pins benötigt. Selbstverständlich kann die verwendete SPI-Schnittstelle dann nicht mehr anderweitig benutzt werden, aber der Hersteller empfiehlt ohnehin, sie nicht für andere Zwecke einzusetzen.

Als Konsequenz ergibt sich, dass Flash-Speicher auf unterschiedlichen ESP32-Boards unterschiedliche Charakteristiken aufweisen kann. Der Speicher variiert nicht nur in der Größe, sondern auch bezüglich der Geschwindigkeit beim Lesen, Schreiben und beim Upload.

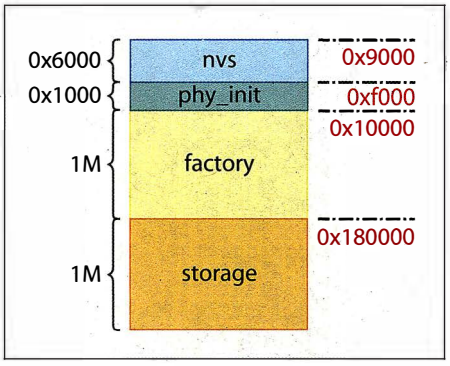
Diese Flexibilität ist im professionellen Umfeld Gold wert und auch Hobbyisten profitieren von einer großen Anzahl von Entwickler-Boards mit unterschiedlichen Leistungsmerkmalen zu unterschiedlichen Preisen. Auf der anderen Seite wird an solchen Stellen deutlich, dass der ESP32 nicht in erster Linie für Einsteiger konzipiert wurde. Wer mit dem ESP32 hantiert, muss sich jederzeit darüber im Klaren sein, wie genau das gerade eingesetzte Board funktioniert.

Die Komplexität liegt nicht nur in der heterogenen Hardware begründet, sondern oft auch in ihrer Leistungsfähigkeit. Beispielsweise ist der Flash-Speicher vieler Boards so groß, dass er nicht nur eine einzelne Anwendung und ihre Daten beherbergen kann. Daher kann er entsprechend partitioniert werden, um die verschiedenen Programme samt Daten sauber voneinander zu trennen.

Stromversorgung

Die Betriebsspannung des ESP32 liegt zwischen 2,3V und 3,6V und angeschlossene Elektronik muss mit 3,3V operieren. Das ist genauso wie beim Raspberry Pi, aber ganz anders als bei den meisten Arduino-Boards, denn die laufen fast ausschließlich mit 5V. Es ist also Vorsicht geboten, wenn man Schaltungen kopiert, die ursprünglich für einen Arduino gedacht waren!

Angesichts der Funktionen, die der ESP32 bietet, dürften sich viele Maker fragen, was der Winzling denn so verbraucht. Schließlich ist das Gerät für mobile Anwendungen geradezu prädestiniert und es wäre doch ein Jammer, wenn ein angeschlossener Akku schon nach kurzer Zeit schlapp machen würde.



Der SPI-Speicher ist partitioniert, Nutzer-Programme residieren in „factory“.

Wie so oft im Leben, lautet die Antwort nach dem Energiebedarf des ESP32: Es kommt darauf an. Im regulären Betrieb verbraucht der Chip zwischen 20mA und 260mA. Das ist eine breite Spanne und der tatsächliche Bedarf hängt maßgeblich davon ab, was der ESP32 so anstellt und welche seiner eingebauten Komponenten er benutzt. Beispielsweise ist der Einsatz von WLAN und Bluetooth vergleichsweise ressourcenintensiv und führt zu einem hohen Stromverbrauch.

Die meisten Anwendungen werden aber nicht permanent alle Funktionen nutzen müssen und zum Beispiel die Funk-Funktionen nur bei bestimmten Ereignissen benötigen. Daher verfügt die CPU über verschiedene Betriebsmodi, mit denen sich signifikant Strom sparen lässt:

Active Mode: In diesem Modus sind alle Einheiten des Chips, inklusive WLAN und Bluetooth, aktiviert. Je nach Funkverhalten, also ob das Gerät mehr sendet oder mehr empfängt, liegt der typische Verbrauch zwischen 95mA und 240mA.

Modem Sleep Mode: In diesem Zustand sind WLAN und Bluetooth deaktiviert, aber die CPU arbeitet potenziell mit voller Kraft. Abhängig von der Anzahl der Kerne und ihrer Taktfrequenz schwankt der Stromverbrauch zwischen 20mA und 68mA. Je nachdem, welche Peripheriegeräte aktiv sind und wie hoch die Prozessorlast ist, passt der ESP32 die Taktfrequenz automatisch an.

Light Sleep Mode: Jetzt ist die CPU ausgeschaltet. Zumindest gilt das für die Haupt-CPU. Der ESP32 hat nämlich einen ULP-Coprozessor (Ultra Low Power), der auch dann noch mit GPIO und I2C arbeiten kann, wenn der große Bruder schläft. Neben dem ULP arbeitet in diesem Zustand noch die Echtzeituhr (RTC) und externe Interrupts sind ebenfalls möglich. Der Stromverbrauch liegt bei geringen 0,8mA.

Deep Sleep Mode: Per Voreinstellung ist jetzt nur noch die RTC aktiviert. Der ULP kann eingeschaltet werden und mit GPIO und I2C arbeiten. Je nach Konfiguration und Last liegt der Stromverbrauch zwischen 10µA und 150µA.

Hibernation: Jetzt ist wirklich nur noch die RTC wach und begnügt sich mit 5µA.

Der Strombedarf bei voller Last wirkt zunächst erschreckend hoch, insbesondere für mobile Anwendungen, die durch eine Batterie oder eine USB-Powerbank angetrieben werden sollen. Die Entwickler des ESP32 hatten den Strombedarf aber immer im Hinterkopf und so ist es mit sorgfältiger Planung und Programmierung möglich, den Energiehunger drastisch zu reduzieren.

Rein und raus

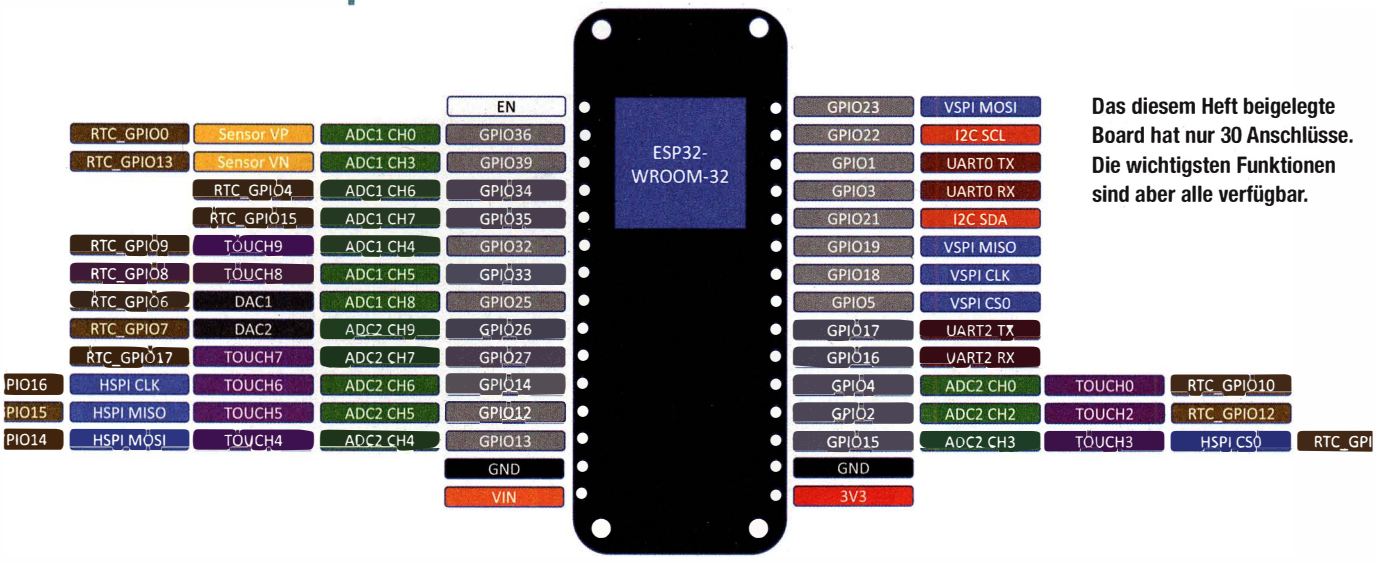
CPU und Speicher eines Mikrocontrollers sind wichtig, aber mindestens genauso wichtig sind die Schnittstellen zur Außenwelt, und hier gibt es beim ESP32 so einiges zu entdecken.

Insgesamt hat der ESP32 48 Pins und den Großteil können Entwickler in ihren eigenen Kreationen verwenden. 34 Pins sind GPIO-Pins und können fast völlig frei als Ein- beziehungsweise Ausgabe-Pins dienen. Als digitale Pins können sie alle fungieren, aber es gibt auch Pins mit speziellen Eigenschaften. Zum Beispiel gibt es analoge Pins und solche, die als kapazitive Sensoren (Touch) dienen. Fast alle GPIO-Pins verfügen darüber hinaus über interne Pullup- und Pulldown-Widerstände.

Bei allen Freiheitsgraden gibt es jedoch auch ein paar Einschränkungen und so sind beispielsweise die sechs GPIO-Pins 34 bis 39 eigentlich nur GPI-Pins, das heißt, sie sind reine Eingabe-Pins und können keine Ausgangssignale erzeugen. Das Sechserpack der GPIO-Pins 6 bis 11 kann in der Regel gar nicht verwendet werden, weil es auf den meisten Boards mit dem Flash-Speicher verbunden ist.

Ferner sind, wie bei den meisten Mikrocontrollern, fast alle Pins mehrfach belegt und können unterschiedliche Funktionen erfüllen. Genau genommen ist die Situation sogar noch etwas komplizierter, denn der ESP32 beherrscht das PIO-Multiplexing. Das bedeutet, dass viele der Funktionen, die der Chip anbietet, auf beinahe beliebige Pins gelegt werden können. Bevor der ESP32 eine Funktion ausführt, sieht er in einer Tabelle nach, welcher Pin gerade dafür zuständig ist. Es gibt für viele Funktionen voreingestellte Pins, aber darauf sollte man sich bei der Analyse von Projekten nicht verlassen.

Eine weitere Spezialität, die den ESP32 von der Konkurrenz abhebt, ist die eingebaute Unterstützung für Berührungssensoren (Touch-Sensoren). Er nennt gleich zehn kapazitive Eingänge sein Eigen und damit



Das diesem Heft beigelegte Board hat nur 30 Anschlüsse. Die wichtigsten Funktionen sind aber alle verfügbar.

lassen sich neben vielen nützlichen Anwendungen auch schnell ein paar Spaß-Projekte umsetzen, etwa ein Synthesizer mit Obst oder Knete als Eingabeelemente.

Digital ist besser?

Bei all diesen fast extravaganten Fähigkeiten darf man das Brot- und Buttergeschäft eines Mikrocontrollers nicht aus den Augen verlieren und auch bei der Verarbeitung analoger und digitaler Signale weiß der ESP32 zu punkten.

Der ESP32 verfügt über einen Digital-/Analog-Wandler (DAC), der auf zwei 8 Bit-Kanälen digitale Signale in analoge Ausgangsspannungen umwandeln kann. Für den umgekehrten Weg bilden 12-Bit SAR ADCs (Successive Approximation Register Analog-to-Digital Converters) analoge Signale auf bis zu 18 Kanälen auf digitale Werte ab. Hierbei gibt es einiges zu beachten.

Die 18 analogen Kanäle zerfallen in zwei Klassen namens ADC1 und ADC2. ADC1 enthält die acht Kanäle, die mit den GPIO-Pins 32 bis 39 verbunden sind. Diese Pins unterliegen keinerlei Einschränkungen. Die zehn Pins der zweiten Gruppe (GPIO 0, 2, 4, 12-15 und 25-27) können nicht unbedingt frei verwendet werden. Generell werden sie vom WLAN-Treiber benötigt und sind blockiert, wenn der Treiber läuft. Darüber hinaus wurden ihnen auf verschiedenen Entwickler-Boards andere Funktionen zugewiesen und so ist es ratsam, sich zuvor über die konkrete Pin-Belegung zu informieren.

Leider klingen die Leistungsmerkmale der Analog-/Digital-Wandler auf dem Papier besser, als sie in Wirklichkeit sind. In den Entwickler-Foren gibt es ellenlange Diskussionen über die Qualität der gemessenen Daten und es scheint auf dem ESP32 diesbezüglich tatsächlich ein Problem zu geben. Die vom ADC erzeugten Werte sind nicht-linear und unterliegen einem mehr oder weniger starken Rauschen. Bisher scheint niemand eine befriedigende Lösung für dieses Problem gefunden zu haben.

Übrigens war für den ESP32 ursprünglich ein Low-Noise Amplifier (LNA) angekündigt. Er scheint es aber nicht ins finale Produkt geschafft zu haben, denn seine Beschreibung wurde in Version 2.1 des Datenblatts entfernt.

Mustererzeugung

Wenig überraschend unterstützt der ESP32 die Pulsweiten-Modulation (PWM), mit der sich unter anderem Motoren steuern lassen. Dank des Multiplexings funktioniert die PWM auf jedem GPIO-Pin, der ein Ausgangssignal erzeugen kann.

Eine weitere Besonderheit ist der LED-PWM-Controller. Mit ihm lässt sich die Intensität von LEDs auf sechzehn Kanälen auf vielfältige Weise steuern. Bei Bedarf erzeugen die sechzehn Kanäle unterschiedliche Schwingungsformen, um RGB-LEDs zu kontrollieren.

Neben diversen Timern, die für hohe Flexibilität bei der Erzeugung schicker Muster sorgen, helfen Funktionen zum Ein-, Aus- und Überblenden von LEDs bei der kreativen Beleuchtung. Solche Spielereien belasten die CPU übrigens nicht und natürlich muss der LED-PWM-Controller nicht unbedingt LEDs ansteuern, sondern kann auch für andere Zwecke eingesetzt werden.

Schnittstellen satt

Große Auswahl bietet der ESP32 bei den Schnittstellen zur Außenwelt, insbesondere bei den seriellen. Gleich drei UART-Einheiten (Universal Asynchronous Receiver/Transmitter) namens UART0, UART1 und UART2 bringt er mit. Sie unterstützen neben RS232 und RS485 auch IrDA und kommunizieren mit einer Geschwindigkeit von bis zu 5 Mbit/s. Alle drei UART-Schnittstellen können von der CPU und per DMA (Direct Memory Access) mit Daten versorgt und ausgelesen werden. Das ist eine feine Sache für Anwendungen, die umfangreiche Daten über die serielle Schnittstelle transportieren wollen, denn DMA-Zugriffe erfolgen parallel zur Arbeit der CPU.

Dank Multiplexing kann beinahe jeder Pin zu jedem Zeitpunkt die serielle Kommunikation übernehmen. Nur UART0 ist etwas besonders, denn diese Schnittstelle wird vom Bootloader im ROM zur Programmierung des ESP32 verwendet. Sie lässt sich allerdings im

DIRECT MEMORY ACCESS

Bei den beispielsweise im Arduino verwendeten ATmega-Mikrocontrollern ist die CPU allein für das Lesen und Schreiben von Speicherbereichen zuständig. Will man etwa Daten seriell per UART oder SPI senden, muss immer wieder die CPU bemüht werden, um neue Daten in die Sende-Register zu schreiben. Beim Direct Memory Acccess (DMA) hilft ein spezieller Controller der Peripherie, indem er die zu sendenden Daten ohne Umweg über die CPU an die Schnittstelle übergibt. Damit kann die CPU ungestört andere Arbeiten übernehmen. Einzige Hürde bei DMA ist die Zugriffsteuerung auf den Speicher, auch Arbitrierung genannt, damit sich CPU und Controller nicht ins Gehege kommen.

laufenden Betrieb – nach dem Bootvorgang – neu konfigurieren.

Der ESP32 unterstützt noch mehr serielle Protokolle und hat sogar gleich zwei I2C-Schnittstellen (Inter-Integrated Circuit). Beide können an einem I2C-Bus sowohl als Slave als auch als Master agieren. Sie kommen mit der Standard-Geschwindigkeit (100 KBit/s) und mit dem Fast-Mode (400 KBit/s) zurecht.

Noch besser sieht es auf den ersten Blick bei den SPI-Schnittstellen (Serial Peripheral Interface) aus, denn davon hat der ESP32 sogar drei. Eine davon ist aber so gut wie immer belegt, denn sie wird benötigt, um den ESP32 mit einem Flash-Baustein zu verbinden. Trotzdem bleiben dann noch zwei SPI-Schnittstellen übrig, über die sich eine Menge Sensoren und andere Bauteile anschließen lassen. Wie die I2C-Schnittstellen können auch die SPI-Schnittstellen per DMA versorgt werden.

I2C und SPI sind weit verbreitet und gehören zum Standard-Repertoire vieler Mikrocontroller. Etwas exotischer ist die I2S-Schnittstelle (Inter-IC Sound), von denen der ESP32 gleich zwei hat. I2S wurde speziell für die Bedürfnisse von Audio-Anwendungen entwickelt und hilft bei der Verbindung digitaler Audio-Geräte, die zum Beispiel PCM-Daten (Pulse-Code Modulation) austauschen wollen. Mit diesem Standard ist es ein Leichtes, Audio-Daten auf einem Lautsprecher auszugeben oder sie über ein günstiges I2S-Microphone-Breakout einzusammeln. Wie bei SPI und I2C kann die Übertragung der Daten per DMA erfolgen.

Schon viel dabei

Im Vergleich zu anderen populären Mikrocontrollern hat der ESP32 eine beeindruckende Zahl von Schnittstellen. SPI und I2C sind zwar auf allen Arduino-Boards zu finden, aber die meisten haben jeweils nur eine Schnittstelle und Verfahren wie I2S sind dort meist gar nicht zu finden. Darüber hinaus spendiert der ESP32 seinen Nutzern noch diverse Sensoren und andere Feinheiten.

Zum Beispiel profitieren ESP32-Nutzer vom SDIO/SPI-Controller (SD Input/Output), der es ziemlich einfach macht, SD-Karten mit dem ESP32 zu lesen und zu schreiben. Zwar lassen sich SD-Karten seit jeher per SPI einbinden, aber eine separate SDIO-Schnittstelle beschleunigt die ganze Angelegenheit erheblich und blockiert keine SPI-Schnittstelle.

Ungewöhnlich ist die Unterstützung des ESP32 für Infrarot-Signale typischer Fernbedienungen. Der ESP32 hat nämlich einen Baustein zur Kodierung und Dekodierung von modulierten Signalen, wie sie in gängigen Infrarot-Fernbedienungen verwendet werden. Das bedeutet aber nicht, dass man sofort mit seiner eigenen Multi-Fernbedienung loslegen kann. Zum einen fehlen den meisten Entwickler-Boards eine Infrarot-LED beziehungsweise ein Infrarot-Empfänger, und zum anderen ist dieses Feature des ESP32 zurzeit nur spärlich dokumentiert.

Trotzdem kann das Feature eine große Hilfe in Projekten sein, die mittels Infrarot kontrolliert werden. Potenziell lässt sich diese Eigenschaft für gänzlich andere Zwecke, zum Beispiel zur Erzeugung interessanter Signale abseits von Infrarot-Licht, missbrauchen. Es ist zu erwarten, dass die Dokumentation in Zukunft deutlich besser wird und mehr Projekte von den IR-Eigenschaften Gebrauch machen.

Tief im Innern des ESP32 existiert ein Temperatur-Sensor. Der ist leider nicht sonderlich genau und liegt in einem Bereich des Chips, in dem er bei hoher Last die Abwärme seiner Kollegen mitbekommt und auch misst. Für absolute Messungen taugt er daher ohne eine ordentliche Kalibrierung nicht, aber er kann unter Umständen verwendet werden, um Schwankungen in der Umgebungstemperatur zu erkennen.

Für den Praxiseinsatz besser geeignet ist der eingebaute Hall-Sensor. Dieser misst die Stärke und Richtung von Magnetfeldern. Hall-Sensoren haben in der Industrie, insbesondere in der Auto-Industrie, wichtige Anwendungen. Zum Beispiel werden sie in Gurtschlössern eingesetzt, um zu erkennen, ob das Gurtschloss verriegelt ist. Auch bei der Messung des Tank-Füllstands werden sie verwendet. In Verbindung mit den Kommunikationssystemen des ESP32 ist der Hall-Sensor eine feine Sache, denn damit lassen sich zum Beispiel leicht kontaktlose Schalter bauen, die ein drahtloses Signal senden.

Listing HallSensor.ino zeigt, wie einfach sich der Hall-Sensor auslesen lässt. Lässt man das Programm laufen und bringt einen Magneten nah an das Board heran, kann man im seriellen Monitor schnell eine deutliche Veränderung der Messwerte beobachten. Allerdings sind die so ermittelten Werte alles andere als genau, können aber ohne großen Aufwand verbessert werden.

Übrigens gibt es im Menü „Datei/Beispiele" der Arduino-IDE einen eigenen Abschnitt nur für den ESP32. Es lohnt sich, zumindest mal einen Blick auf alle Beispiele zu werfen, um ein Gefühl dafür zu bekommen, was alles möglich ist.

Der Netzwerker

Die meisten Bastler dürften auf den ESP32 wegen seiner vielfältigen Kommunikationsmöglichkeiten aufmerksam werden. Problemlos verbindet sich das Gerät mit WLAN-Netzen der Versionen 802.11b/g/n (manche Modelle mit noch mehr) und kommt dabei auf Übertragungsgeschwindigkeiten von bis zu 150MBit/s. Das reicht zum Streamen von Audiodaten und selbst für Video-Streams ist das für manche Projekte genug. Verschlüsselte Verbindungen per WPA/WPA2 und WAPI (WLAN Authentication and Privacy Infrastructure) bereiten dem Mini-Funker keine Probleme.

Genauso gut sieht die Lage beim Thema Bluetooth aus, denn hier unterstützt der ESP32 moderne Standards wie Bluetooth

Viele Beispiele zeigen in der Arduino-IDE, wie sich die Bausteine des ESP32 nutzen lassen.

HallSensor.ino

```
1 void setup() {
2   Serial.begin(9600);
3 }
4
5 void loop() {
6   int wert = hallRead();
7   Serial.println(wert);
8   delay(500);
9 }
```

RandomNumbers.ino

```
1 void setup() {
2   Serial.begin(115200);
3 }
4
5 void loop() {
6   Serial.println("***");
7   Serial.println(esp_random());
8   Serial.println(random(10));
9   Serial.println(random(10, 20));
10  delay(1000);
11 }
```

4.2 und Bluetooth Low Energy (BLE). Bandbreite und Geschwindigkeit reichen für Audio-Streams und nach der reinen Faktenlage sollte die Ansteuerung von Audio-Headsets oder die Realisierung eines Bluetooth-Lautsprechers funktionieren. Es gibt ein paar vielversprechende Projekte im Netz, aber die Firmware scheint noch nicht hundertprozentig ausgereift zu sein. Allerdings gab es hier in letzter Zeit große Fortschritte.

Bei all den drahtlosen Funk-Technologien kann man den Ethernet Media Access Controller (MAC) auf dem ESP32 schon mal schnell übersehen. Tatsächlich kann sich der Chip darüber mit einem kabelgebundenen LAN verbinden. Schließlich hat der ESP32 noch einen Controller für den CAN-Bus (Controller Area Network), der in der Auto-Industrie weit verbreitet ist.

Nicht wenige Nutzer dürften sich für einen ESP32 entscheiden, weil er mit WLAN und Bluetooth zwei wichtige Technologien zur drahtlosen Kommunikation zu einem günstigen Preis anbietet. Für viele Anwendungsfelder, wie zum Beispiel bei der Heimautomatisierung, ist das Gold wert.

Geheimniskrämerei

Bei der ganzen Funkerei darf man heutzutage das Thema Sicherheit nicht aus den Augen verlieren. Das wohlige Gefühl des Erfinderstolzes ist schnell verflogen, wenn nerdige Nachbarskinder oder gar kriminelle Gesellen die eigens entwickelte Garagentor-Steuerung mit ihrem Smartphone übernehmen und aushebeln.

Verschlüsselung ist heutzutage bei jedweder öffentlichen Kommunikation ein Muss und das sollte man auch bei Hobby-Projekten im Hinterkopf haben. Starke kryptografische Algorithmen stellen aber selbst für eine vergleichsweise leistungsstarke CPU, wie sie auf dem ESP32 ihren Dienst verrichtet, eine große Herausforderung dar. Da ist es umso erfreulicher, dass der ESP32 ein paar nützliche Helfer an Bord hat, die ihm die stupide Rechnerei abnehmen.

Bei der symmetrischen Verschlüsselung greift ihm beispielsweise ein AES-Beschleuniger unter die Arme. Der Advanced Encryption Standard (AES) gehört zu den wichtigsten symmetrischen Verschlüsselungsalgorithmen und der ESP32 kommt mit Schlüsseln der Länge 128, 192 und 256 Bit klar.

Symmetrische Verschlüsselungsalgorithmen haben eine ganze Menge Vorteile. Sie sind schnell, einfach zu implementieren, einfach zu verstehen und ziemlich sicher. Ihr größtes Problem ist die Verteilung der Schlüssel, denn zwei Partner, die im Geheimen miteinander kommunizieren wollen, müssen denselben Schlüssel besitzen. Dieser dient sowohl zum Verschlüsseln als auch zum Entschlüsseln einer Nachricht. Wie können sich jedoch beide Partner auf einen Schlüssel einigen, ohne dass er von Dritten abgehört wird?

Um dieses Problem zu lösen, werden bei der sicheren Kommunikation heute daher asymmetrische Verfahren, wie zum Beispiel RSA (die Abkürzung steht für die Namen der Erfinder Rivest, Shamir und Adleman), immer wichtiger. RSA wird nicht nur bei der Verschlüsselung, sondern oft auch bei der Berechnung digitaler Signaturen verwendet.

Der ESP32 bietet zwar keine direkte Unterstützung für den RSA-Algorithmus, aber für die zugrunde liegende Mathematik. Die basiert auf modularer Arithmetik mit sehr großen ganzen Zahlen und dabei müssen diese Zahlen unter anderem multipliziert werden. Der ESP32 unterstützt die Multiplikation von Zahlen mit einer Länge von bis zu 2048 Bits und erlaubt die modulare Potenzierung von Zahlen mit einer Länge von bis zu 4096 Bits. Auf dieser Basis lässt sich der RSA-Algorithmus mit ausreichend großen Schlüssellängen implementieren und eine solche Implementierung gehört auch zur Standard-Bibliothek.

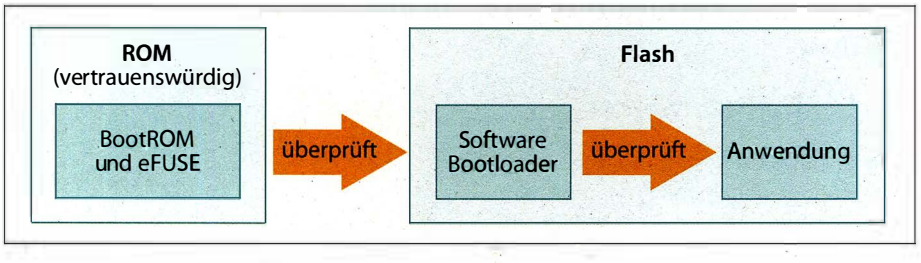
In der Absicherung von Kommunikationssystemen spielen nicht nur Verschlüsselungsalgorithmen eine wichtige Rolle, denn selbst über einen verschlüsselten Kanal können kompromittierte Dokumente und Informationen übertragen werden. Um die Integrität und Authentizität von Informationen zu gewährleisten, kommen kryptografische Hash-Funktionen zum Einsatz. Sie helfen bei der Berechnung von sicheren Prüfsummen und bei der Erstellung digitaler Signaturen.

Es gibt einige solcher Hash-Funktionen, aber die derzeit wichtigsten stammen aus der SHA-Familie (Secure Hash Algorithm). Insbesondere gelten heute SHA-256, SHA-384 und SHA-512 noch als sicher und kommen in vielen Kontexten zum Einsatz. Für alle drei bietet der ESP32 eine Hardware-Beschleunigung. Das gilt auch für den Algorithmus SHA-1, der nicht mehr als sicher gilt und vermutlich nur aus Gründen der Kompatibilität angeboten wird.

Würfeleien

Viele Laien überrascht zunächst, dass der Zufall in der Kryptografie eine große Rolle spielt. Gute Zufallszahlen bilden aber die Basis für die Sicherheit vieler Verfahren. Auf deterministischen Maschinen ist es sehr schwer bis unmöglich, wirklich zufällige Zahlen zu erzeugen. Das ist auf dem ESP32 prinzipiell nicht anders, aber im Gegensatz zu vielen anderen Rechnern verfügt er über echte Entropie-Quellen, also über Eingangssignale, die tatsächlich zufällig erzeugt werden.

Diese Quellen echten Zufalls sind das Rauschen in den WLAN- und Bluetooth-Daten. Wie bei allen Funkmedien fallen hier jede Menge zufällige Störgeräusche an und die sammelt der ESP32 kontinuierlich in einem Register, so dass das System unentwegt mit neuen Zufallszahlen versorgt wird. Sowohl die Geschwindigkeit, mit der neue Zufalls-



Secure Boot sorgt für sichere Projekte.

zahlen erzeugt werden, als auch die Qualität der Zufallszahlen ist erstaunlich hoch.

Listing RandomNumbers.ino zeigt, wie die Zufallszahlen ausgelesen werden können. Die Funktion `esp_random()` gibt eine Zufallszahl zwischen 0 und 4.294.967.295 zurück. Solch große Werte sind eher unhandlich und die Funktion `random()` stutzt sie auf einen vorgegebenen Wertebereich zurecht. Übergibt man der Funktion nur einen Wert, so liefert sie Zufallszahlen zwischen 0 und dem übergebenen Wert minus 1. Das heißt, `random(10)` gibt Werte zwischen 0 und 9 zurück. Analog gibt `random(10, 20)` Zufallszahlen im Bereich von 10 bis 19 zurück.

Wenn WLAN und Bluetooth abgeschaltet sind, ist es mit der zufälligen Pracht natürlich sofort vorbei. In diesem Fall liefert der ESP32 nur noch Pseudozufallszahlen und die sollten niemals Grundlage für ernsthafte kryptografische Anwendungen sein. Sie sind aber nicht völlig nutzlos und können zum Beispiel in Spielen verwendet werden.

Noch sicherer

Die bisher vorgestellten Verfahren zur Verschlüsselung kommen alle auf Anwendungsebene zum Tragen. Das ist für einige Szenarien aber bereits zu spät, denn die Anwendung liegt ja in der Regel unverschlüsselt im Flash-Speicher und kann ohne großen Aufwand inspiziert werden.

Deshalb bietet der ESP32 Sicherheitsfunktionen an, die auf viel niedrigerer Ebene greifen. Beispielsweise kann er die Inhalte des angeschlossenen Flash-Speichers verschlüsseln. Die Verschlüsselung erfolgt via AES-256 und der Mechanismus ist vergleichsweise einfach. Mittels eines speziellen Kommandos wird der Flash-Speicher mit einem neuen Inhalt überschrieben und gleichzeitig verschlüsselt. Den Schlüssel kann der Anwender vorgeben oder vor dem Schreibvorgang erzeugen lassen.

In beiden Fällen wird der Schlüssel im internen eFuse-Speicher des ESP32 für alle Ewigkeit gespeichert. Startet man den ESP32 und die gespeicherte Anwendung, so entschlüsselt das Gerät transparent alle Daten mithilfe des eingebrannten Schlüssels. Angreifer, die versuchen, den Flash-Baustein auszulesen, sehen nur Datenmüll und an den Schlüssel im eFuse-Speicher kommen sie nicht heran.

Wer im Besitz des Schlüssels ist, kann nicht nur den Inhalt des Flash-Speichers lesen, sondern ihn auch mit neuen Inhalten überschreiben. Dieses Verfahren ist nicht narrensicher, bietet aber ausreichend Sicherheit, um den Großteil potenzieller Angreifer abzuwehren.

All das war den Entwicklern bei Espressif noch immer nicht sicher genug und so haben sie dem ESP32 noch einen Secure Boot-Mechanismus spendiert. Der stellt sicher, dass weder der Bootloader noch der Code der Anwendung im Flash manipuliert wurden.

Die Sicherheit des Systems basiert darauf, dass der eFuse-Speicher des ESP32 so konfiguriert werden kann, dass ein Auslesen und Beschreiben per Software unmöglich wird. Weil auch das ROM nicht verändert werden kann, gelten alle Schlüssel, die im eFuse gespeichert werden und der Boot-Code im ROM als vertrauenswürdig.

Der Mechanismus zur Überprüfung der Anwendung und des Bootloaders arbeitet in zwei Schritten und das Boot-ROM prüft zuerst die Integrität des Software-Bootloaders. Wenn diese Prüfung erfolgreich war, steht fest, dass der Software-Bootloader nicht manipuliert wurde und das Boot-ROM führt ihn aus. Jetzt prüft der Software-Bootloader die Integrität der Anwendung und führt sie nach erfolgreicher Prüfung ebenfalls aus.

Der ganze Prozess ist komplex, aber nicht sonderlich kompliziert. Insbesondere muss man darauf achten, wie die verwendeten Schlüssel erzeugt und gespeichert werden. Wer auf Basis des ESP32 möglichst sichere Hardware produzieren will, kommt um ein intensives Studium der Dokumentation nicht herum.

Secure Boot funktioniert übrigens ohne verschlüsselten Flash-Speicher und umgekehrt, aber es ist empfehlenswert, beides zu aktivieren.

Fazit

Der ESP32 ist ein komplexes, aber attraktives Biest. Gemessen an dem, was dieser kleine Chip so bietet, wirken der Arduino Uno und auch der ESP8266 geradezu winzig. Die Firma Espressif hat die Limits des ESP8266 ganz klar identifiziert und in fast allen Punkten ausgemerzt. Lediglich bei der Qualität der Software und bei der Dokumentation gibt es noch ein paar Defizite.

Nicht nur die leistungsstarke CPU, die vielen Schnittstellen und Sensoren, sondern auch die ausgefeilten Sicherheitsfunktionen machen den ESP32 zu einer perfekten Spielwiese für Bastler und gleichzeitig zu einer ernstzunehmenden Plattform für kommerzielle Produkte. —*dab*

Mikrocontroller

	ESP32	ESP8266	Arduino Uno
CPU	Xtensa Dual-Core 32-bit LX6	Xtensa Single-core 32-bit L106	ATmega 328P
max. Taktfrequenz CPU	240MHz	160MHz	16MHz
SRAM	520KB	160KB	2KB
Flash	meistens 4MB	meistens 4MB	32KB
GPIOs	36	17	14
SPI-Schnittstellen	4	2	1
I2C-Schnittstellen	2	1	1
I2S-Schnittstellen	2	1	-
UART-Schnittstellen	3	2	1
CAN 2.0-Schnittstellen	1	-	-
SD/SDIO/MMC Host-Controller	1	-	-
SDIO/SPI Slave-Controller	1	1	-
ADC-Kanäle	18 (12 Bit)	1 (10 Bit)	6 (10 Bit)
IR-Funktionalität	ja (Hardware)	ja (größtenteils Software)	-
Ethernet MAC-Schnittstelle	1	-	-
GPIOs für Touch-Sensoren	10	-	-
Hall-Sensor	1	-	-
Temperatur-Sensor	1	-	-
DAC-Kanäle	2 (8 Bit)	-	-
WLAN	802.11 b/g/n	802.11 b/g/n	-
Bluetooth	Bluetooth 4.2 und BLE	-	-
Betriebsspannung	2,3V bis 3,6V	2,3V bis 3,6V	7V bis 12V

Im direkten Vergleich hat der ESP32 gegenüber dem ESP8266 und dem Arduino Uno die Nase vorn.

Flinker Funkwellenreiter

Ein Killer-Feature des ESP32 ist gewiss WLAN, denn sowohl beim Preis als auch bei der Leistung hängt er diesbezüglich die Konkurrenz locker ab. Mit Leichtigkeit wird er zum Client oder Server im heimischen Netz und kommt problemlos mit allen gängigen Sicherheitsverfahren zurecht.

von Maik Schmidt

Alles zum Artikel im Web unter ***make-magazin.de/xm8v***

WLAN (Wireless Local Area Network) ist längst eine Standard-Technologie und die meisten Nutzer verschwenden kaum einen Gedanken an die Komplexität der Mechanismen, die eine stabile und sichere drahtlose Verbindung überhaupt möglich machen. Dankenswerterweise versteckt der ESP32 die Details weitestgehend hinter einer einfachen Fassade. Trotzdem hilft es bei der Entwicklung von Projekten, zumindest die grundlegenden Prinzipien zu verstehen.

Irgendwer zu sehen?

Bevor man sich mit einem WLAN verbindet, ist es ratsam, zunächst nach allen Netzwerken in der Umgebung zu suchen. Eine erfolgreiche Suche stellt sicher, dass die WLAN-Hardware des ESP32 ordentlich funktioniert und für einen solchen Test benötigt man keinerlei Zugangsdaten. Selbstverständlich muss dazu mindestens ein WLAN in Reichweite sein. Dank der WiFi-Bibliothek ist ein Netzwerk-Scan recht einfach und Listing ScanWLAN.ino zeigt, wie es geht.

Das Programm sucht nach allen drahtlosen Netzwerken, die es finden kann und gibt sie dann auf der seriellen Schnittstelle aus. Dazu bindet es in der ersten Zeile die WiFi-Bibliothek ein und initialisiert in der `setup`-Funktion die serielle Schnittstelle.

In der `loop`-Funktion liefert der Aufruf von `WiFi.scanNetworks()` die Anzahl der gefundenen Netzwerke zurück. Wurde keins gefunden, gibt das Programm eine entsprechende Meldung aus und versucht es nach fünf Sekunden noch einmal. Andernfalls gibt es die Anzahl und anschließend die wichtigsten Eigenschaften der gefundenen Netzwerke in tabellarischer Form über die serielle Schnittstelle aus.

Den Kopf der Tabelle erzeugen drei Aufrufe der Funktion `Serial.println`. Die Ausgabe der Netzwerkeigenschaften übernimmt eine `for`-Schleife, die drei Aufrufe der Funktion `Serial.printf` enthält. Hier muss man etwas genauer hinsehen, denn `println` und `printf` sehen sich recht ähnlich, verhalten sich aber sehr unterschiedlich.

Während `println` einen übergebenen Wert unverändert ausgibt, kann `printf` ihn vor der Ausgabe formatieren. Beispielsweise kann die Funktion dafür sorgen, dass ein Wert immer mit einer minimalen oder einer maximalen Anzahl an Zeichen ausgegeben wird. Dazu muss man der Funktion als erstes Argument einen Format-String übergeben. Der beschreibt, wie die nachfolgenden Werte dargestellt werden sollen. Das wirkt auf den ersten Blick etwas kryptisch, wird aber schnell zu einem Helfer, den man nicht mehr missen möchte.

ScanWLAN.ino

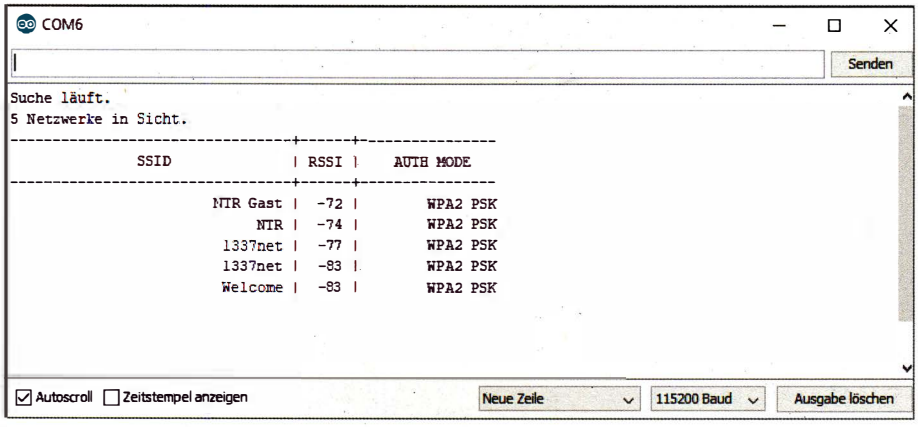
```
 1 #include <WiFi.h>
 2
 3 void setup() {
 4   Serial.begin(115200);
 5 }
 6
 7 void loop() {
 8   Serial.println("Suche läuft.");
 9   const uint8_t n = WiFi.scanNetworks();
10
11   if (n == 0) {
12     Serial.println("Kein Netzwerk gefunden.");
13   } else {
14     Serial.print(n);
15     Serial.println(" Netzwerke in Sicht.");
16     Serial.println("--------------------------------+------+-----
       -----------");
17     Serial.println("                SSID              | RSSI |
       AUTH MODE   ");
18     Serial.println("--------------------------------+------+-----
       -----------");
19     for (uint8_t i = 0; i < n; i++) {
20       Serial.printf("%32.32s | ", WiFi.SSID(i).c_str());
21       Serial.printf("%4d | ", WiFi.RSSI(i));
22       Serial.printf("%15s\n",
         authModeToText(WiFi.encryptionType(i)).c_str());
23     }
24   }
25   Serial.println("");
26   delay(5000);
27 }
28
29 String authModeToText(wifi_auth_mode_t authMode) {
30   switch (authMode) {
31     case WIFI_AUTH_OPEN:
32       return "Offen";
33     case WIFI_AUTH_WEP:
34       return "WEP";
35     case WIFI_AUTH_WPA_PSK:
36       return "WPA PSK";
37     case WIFI_AUTH_WPA2_PSK:
38       return "WPA2 PSK";
39     case WIFI_AUTH_WPA_WPA2_PSK:
40       return "WPA/WPA2 PSK";
41     case WIFI_AUTH_WPA2_ENTERPRISE:
42       return "WPA2 ENTERPRISE";
43     case WIFI_AUTH_MAX:
44       return "MAX";
45     default:
46       return "Unbekannt";
47   }
48 }
```

Die erste Anweisung innerhalb der `for`-Schleife gibt die SSID (Service Set Identifier), also den Namen des Netzwerks aus. Den liefert die Funktion `WiFi.SSID` zurück und erwartet als Argument den Index des Netzes, dessen Name gewünscht wird. Die `printf`-Anweisung formatiert den Namen rechtsbündig mit exakt 32 Zeichen. So lang darf eine SSID nämlich maximal sein.

Weil `WiFi.SSID` ein Objekt der Klasse `String` zurückliefert, wird die Funktion `c_str` aufgerufen. Die ermöglicht den Zugriff auf die interne Repräsentation der Zeichenkette als `const char*`, wie er von `printf` erwartet wird.

Die nächste Anweisung gibt mithilfe von `WiFi.RSSI` die RSSI (Received Signal Strength Indication), also die Signalstärke des Netzes, aus. Der RSSI-Wert ist eigentlich relativ und hat keine Einheit. Wenn man aber die Eigenschaften des zugrundeliegenden Gerätes kennt, kann man die RSSI in der Einheit dBm (Dezibel Milliwatt) ausdrücken. Genau das tut die Funktion `WiFi.RSSI` und es gilt: Je größer der Rückgabewert, umso stärker ist das Signal. Bei negativen Zahlen ist das Signal daher umso stärker, je näher es an der 0 liegt.

Zuletzt ermittelt das Programm mittels `WiFi.encryptionType` eine weitere wichtige Eigenschaft eines jeden WLAN-Netzes, nämlich die verwendete Authentifizierungsmethode. Vor der Ausgabe wandelt die Hilfsfunktion `authModeToText` den nume-



Das erste Programm stellt die drahtlosen Netzwerke in der Umgebung übersichtlich im seriellen Monitor dar.

rischen Enumerationstypen in einen lesbaren Text um.

Es geht los

Jetzt ist es an der Zeit, das Programm endlich auf das ESP32-Board zu laden und auszuprobieren. Weil der Code nur nach Netzwerken sucht und nicht versucht, sich mit einem zu verbinden, sollte er in jeder Umgebung, in der mindestens ein drahtloses Netzwerk in Reichweite ist, funktionieren.

Schon kurz nachdem man den seriellen Monitor in der Arduino-IDE geöffnet hat, erscheint eine Tabelle mit den Eigenschaften aller gefundenen Netzwerke. Diese Tabelle wird alle fünf Sekunden aktualisiert und enthält alle Netzwerke, die auch der heimische PC oder das Smartphone anzeigen.

Zu jeder SSID enthält die Tabelle die aktuelle Signalstärke und die verwendete Authentifizierungsmethode. Dabei kann es passieren, dass manche SSIDs mehrfach erscheinen, weil sie zum Beispiel von mehreren Routern beziehungsweise Repeatern verwendet werden. Um die Netzwerke eindeutig voneinander unterscheiden zu können, gibt es noch die BSSID (Base Service Set ID), eine 48-Bit-Zahl, die denselben Konventionen gehorcht wie MAC-Adressen (Media Access Control). Diese Information kann bei Bedarf ebenfalls über die WiFi-Bibliothek ermittelt werden.

Ich bin drin!

Nach diesem ersten Test, der sicherstellt, dass die WLAN-Funktionen prinzipiell richtig arbeiten, wird es Zeit für ein umfangreicheres Beispiel. Listing MiniWebServer.ino macht den ESP32 zu einem kleinen Web-Server, der auf Anfragen eines Web-Browsers wartet und sie dann beantwortet.

DRUCKEN MIT FORMAT

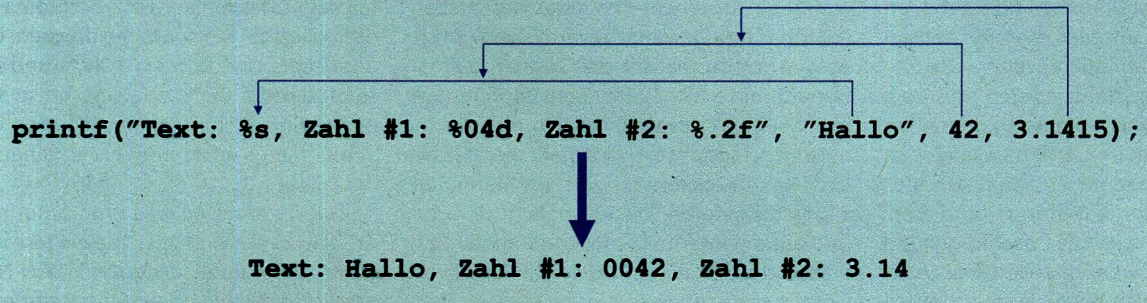
Zwar hat der ESP32 keinen Bildschirm, aber die meisten Projekte geben Informationen über die serielle Schnittstelle aus. Manchmal sind es Meldungen über den aktuellen Stand des Programms und oft sind es Messdaten, die in einem festen Format erscheinen sollen.

Die Formatierung von Daten ist mit der herkömmlichen Zeichenketten-Funktion oft etwas friemelig. Daher bietet die `Serial`-Klasse die `printf`-Funktion (print formatted) an, die es in dieser oder ähnlicher Form schon seit mehr als 50 Jahren gibt.

`printf` gibt eine Liste von Werten aus und erwartet neben den Werten einen Formatstring, der deren Formatierung beschreibt. Der Format-String enthält einen Platzhalter für jeden Wert, der beschreibt, wie der Wert vor der Ausgabe zu formatieren ist. Alle Zeichen, die nicht zu einem Platzhalter gehören, werden unverändert ausgegeben.

Jeder Platzhalter beginnt mit einem Prozentzeichen und muss den Typen des Parameters enthalten. Ein s steht beispielsweise für einen String und ein d für eine vorzeichenbehaftete Ganzzahl. Zusätzlich zum Typen können noch eine Mindestlänge, eine Maximallänge, die Anzahl der Nachkommastellen und Optionen zum Auffüllen leerer Ausgabefelder angegeben werden. Damit lassen sich Werte zum Beispiel linksbündig oder rechtsbündig ausgeben und Zahlen können durch führende Nullen auf eine gleichbleibende Breite gebracht werden.

Eine vollständige Beschreibung aller unterstützten Formate würde den Rahmen sprengen, aber mithilfe der Dokumentation ist es ganz einfach.



Formatierte Ausgaben werden mit printf zum Kinderspiel.

Das Programm ist zwar nicht allzu umfangreich, verwendet aber eine ganze Menge der WLAN-Funktionen, die der ESP32 bietet. Nachdem alle benötigten Bibliotheken eingebunden wurden, werden zwei Konstanten namens `SSID` und `PASSWORD` definiert. Sie enthalten die SSID und das Passwort des Netzwerks, mit dem sich der ESP32 verbinden soll. Ihr Inhalt muss an die lokalen Gegebenheiten des eigenen Netzes angepasst werden.

Als Nächstes erzeugt das Programm ein globales Objekt der Klasse `WebServer` mit dem Namen `server`. Der Konstruktor der Klasse erhält das Argument 80, denn das ist der Standard-Port für HTTP (HyperText Transfer Protocol) und auf dem soll der Web-Server lauschen. HTTP ist das Protokoll des World Wide Web, über dass alle Browser und Web-Server miteinander kommunizieren. Die Klasse `WebServer` implementiert das Protokoll für den ESP32.

Nachdem die `setup`-Funktion die serielle Schnittstelle initialisiert hat, versetzt `WiFi.mode` den ESP32 in den Station-Modus. In einem WLAN können die Teilnehmer entweder eine Station, ein Access Point oder beides sein. Eine Station ist jedes Gerät, das sich mit einem drahtlosen Netzwerk verbinden kann. Typischerweise verbindet sich eine Station mit einem Router, um Zugang zum Internet oder anderen Geräten innerhalb desselben Netzwerks zu bekommen. Ein Router ist wiederum ein Access Point.

Der ESP32 kann in allen drei Modi operieren, das heißt, er kann sich mit dem heimischen Router verbinden, so dass ihn alle anderen Geräte erreichen können. Er kann auch als Access Point fungieren und so ein eigenes WLAN aufspannen. Auf diese Weise können sich andere Geräte direkt, also ohne den Umweg über einen Router, mit dem ESP32 verbinden. Schließlich ist der ESP32 noch in der Lage, gleichzeitig als Station und als Access Point zu arbeiten.

Weiter geht es im Code mit der Funktion `WiFi.begin`. Sie verbindet den ESP32 mit einem WLAN und benötigt dazu die SSID und das Passwort. Der Vorgang kann durchaus ein paar Sekunden dauern. Daher prüft die folgende `while`-Schleife alle 500 Millisekunden, ob die Verbindung bereits zustande gekommen ist. Während der Wartezeit gibt das Programm immer wieder einen Punkt auf der seriellen Schnittstelle aus, so dass man sehen kann, ob sich überhaupt etwas tut. Wurde die Verbindung erfolgreich hergestellt, gibt das Programm die SSID und die zugewiesene IP-Adresse aus.

Ab diesem Zeitpunkt kann jeder andere Teilnehmer im WLAN mit dem ESP32 kommunizieren, wenn er dessen IP-Adresse kennt. Schöner wäre es, wenn man dem ESP32 einen eigenen Namen im Netz geben könnte, unter dem andere Geräte ihn erreichen können. Für diesen Zweck gibt es das mDNS-Protokoll (Multicast Domain Name System), mit dem sich in kleinen Netzwerken Namen ohne einen lokalen Name-Server auflösen lassen.

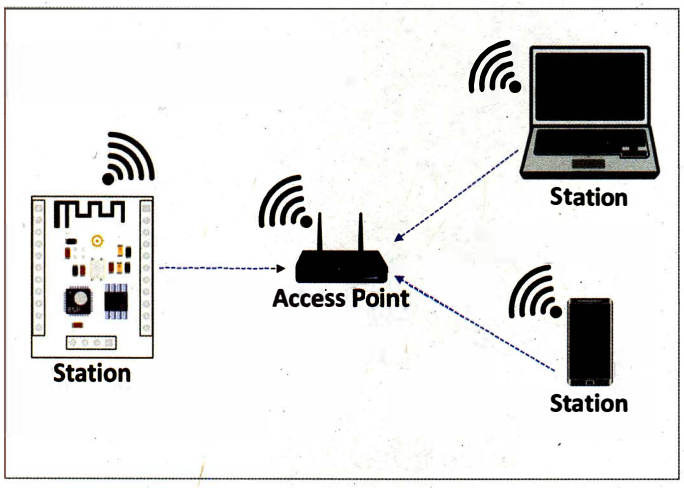
Genau das erledigt die `MDNS`-Klasse und deren `begin`-Funktion macht den ESP32 im Netz in diesem Beispiel unter dem Namen „esp32" bekannt. Selbstverständlich kann man dem Gerät einen anderen Namen geben und wer mehrere Geräte betreibt, sollte sich ein vernünftiges Namensschema überlegen, um nicht irgendwann die Übersicht zu verlieren.

MiniWebServer.ino

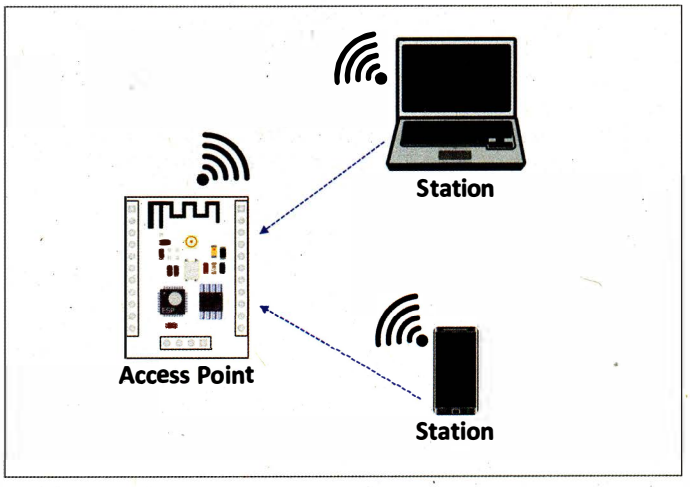
```
 1 #include <WiFi.h>
 2 #include <WiFiClient.h>
 3 #include <WebServer.h>
 4 #include <ESPmDNS.h>
 5
 6 const char* SSID = "SSID eintragen";
 7 const char* PASSWORD = "Passwort eintragen";
 8
 9 WebServer server(80);
10
11 void setup() {
12   Serial.begin(115200);
13   WiFi.mode(WIFI_STA);
14   WiFi.begin(SSID, PASSWORD);
15
16   while (WiFi.status() != WL_CONNECTED) {
17     delay(500);
18     Serial.print(".");
19   }
20   Serial.println("");
21   Serial.printf("Verbunden mit %s.\n", SSID);
22   Serial.printf("IP-Adresse: %s.\n",
     WiFi.localIP().toString().c_str());
23
24   if (MDNS.begin("esp32")) {
25     Serial.println("MDNS-Responder gestartet.");
26   }
27
28   server.on("/", handleRoot);
29
30   server.on("/make", []() {
31     server.send(200, "text/plain", "Was soll ich machen?");
32   });
33
34   server.onNotFound(handleNotFound);
35
36   server.begin();
37   Serial.println("HTTP-Server gestartet.");
38 }
39
40 void loop() {
41   server.handleClient();
42 }
43
44 void handleRoot() {
45   server.send(200, "text/plain", "Hier ist der ESP32!");
46 }
47
48 void handleNotFound() {
49   String message = "Pfad ";
50   message += server.uri();
51   message += " wurde nicht gefunden.\n";
52   server.send(404, "text/plain", message);
53 }
```

Wo ist das Web?

Ab diesem Zeitpunkt ist der ESP32 mit dem WLAN verbunden und unter dem Namen „esp32" erreichbar. Allerdings tut er noch nicht das, was man von einem Web-Server erwartet, das heißt, er beantwortet keine



Der ESP32 funktioniert prima als Station in einem WLAN.



Der ESP32 arbeitet als WLAN-Access-Point.

HTTP-Anfragen. Dieses Manko lässt sich aber mit wenigen Handgriffen beheben.

Gleich zu Anfang des Programms wurde ein Objekt der Klasse `WebServer` mit dem Namen `server` definiert. Dieses Objekt übernimmt die gesamte Kommunikation mit dem Web-Browser und muss entsprechend konfiguriert werden.

Die WebServer-Klasse folgt einem recht modernen Entwurf und unterstützt die Registrierung von Event-Handlern. Das sind Funktionen, die im Falle bestimmter Ereignisse aufgerufen werden und sie können unter anderem mit der Funktion `on` an das WebServer-Objekt angehängt werden. Die Anweisung `server.on( „/“ , handleRoot)` lässt sich daher wie folgt lesen: Wenn ein Web-Browser nach dem Pfad „/“ fragt, dann rufe die Funktion `handleRoot` auf. Das heißt, wenn das Programm läuft und in einem Browser die URL http://esp32/ aufgerufen wird, wird die Funktion `handleRoot` aufgerufen.

Diese Funktion besteht nur aus einer einzigen Anweisung, nämlich dem Aufruf der Funktion `send`, die drei Parameter erwartet. Der erste Parameter ist der HTTP-Status-Code, der an den Web-Browser gesendet wird. Die 200 steht für OK, das heißt, die Verarbeitung der Anfrage verlief erfolgreich. Der zweite Parameter enthält den Content-Type und beschreibt, welches Format die Antwort hat. In diesem Fall ist es `text/plain`, also enthält die Antwort ganz ordinären Text. Der dritte Parameter enthält den Inhalt der Antwort.

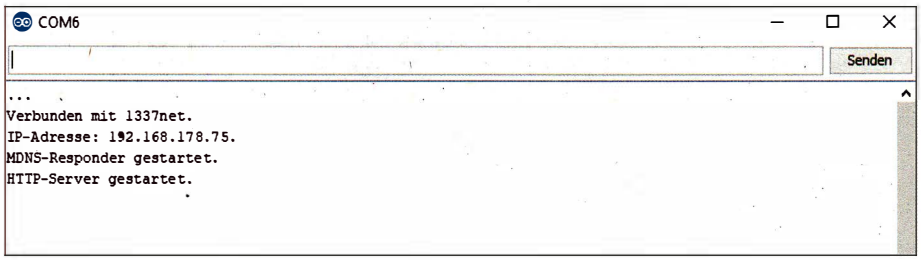
Für Event-Handler, die so kurz sind wie `handleRoot`, lohnt es sich kaum, eine eigene Funktion zu definieren. Das gilt ebenso für den zweiten Event-Handler, der sich um den Pfad /make kümmert. Der wird daher mit einer anonymen Funktion definiert.

Anonyme Funktionen sind eine relativ neue Erweiterung der Programmiersprache C++ und sie sind gerade in solchen Situationen sehr praktisch. Die Konstruktion `[]()` definiert eine namenlose Funktion ohne Parameter und die Definition der Funktion erfolgt gleich anschließend und wie gewohnt in geschweiften Klammern. Das sieht auf den ersten Blick wüst aus, dient aber letzten Endes der Lesbarkeit und befreit Entwickler von der lästigen Aufgabe, sich ständig neue Namen für die Event-Handler auszudenken.

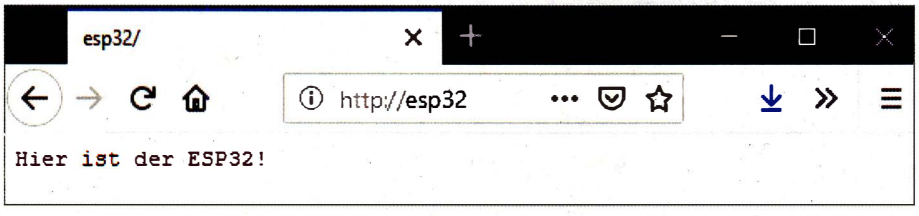
Die bisherigen Event-Handler haben sich um die Behandlung fest definierter Pfade gekümmert. Was aber passiert, wenn ein Client einen Pfad anfragt, für den kein Event-Handler festgelegt wurde? In diesem Fall ruft die WebServer-Klasse die Funktion auf, die zuvor mittels `onNotFound` registriert wurde. In diesem Programm ist das `handleNotFound`. Sie baut eine entsprechende Fehlernachricht zusammen und sendet sie mit dem Status-Code 404 (File Not Found) an den Client zurück.

Zum Schluss ruft die `setup`-Funktion `server.begin` auf und der Web-Server wartet ab diesem Moment auf neue Anfragen auf Port 80. Die tatsächliche Bearbeitung dieser Anfragen übernimmt `handleClient` in der Funktion `loop`.

Nachdem das Programm übersetzt und aufs Board geladen wurde, wird es Zeit für einen ersten Test. Im seriellen Monitor kann man zuerst einmal prüfen, ob die WLAN-Verbindung etabliert werden konnte. Wenn das der Fall ist, ruft man mit einem beliebigen Web-Browser – es darf auch einer auf einem Smartphone im selben WLAN sein – die Adresse http://esp32/ auf. Die führt zu einer spartanischen Begrüßungsseite. Auch die Adresse http://esp32/make (Achtung:



Der ESP32 hat Kontakt mit dem heimischen WLAN-Router.



Der kleine Web-Server antwortet auf eine Anfrage.

Am Ende der Adresse steht kein Schrägstrich!) sollte ein sinnvolles Ergebnis liefern und für alle anderen Pfade zeigt der ESP32 zumindest eine ordentliche Fehlermeldung an. Je nach Konfiguration des lokalen Netzwerkes und des benutzten Clients kann es notwendig sein, die Adresse http://esp32.local/ zu verwenden. Selbstverständlich ist der ESP32 auch über seine IP-Adresse erreichbar.

Mit knapp 50 Zeilen Code wird der ESP32 nicht nur zu einem vollwertigen Mitglied eines abgesicherten WLANs, sondern verrichtet gleichzeitig seinen Dienst als Web-Server. Das Zusammenspiel der guten Hardware und der guten Software machen das nicht nur möglich, sondern sogar recht einfach.

Web-Sensorik

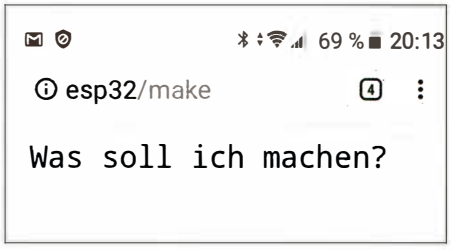
Prinzipiell eignet sich der ESP32 als klassischer Web-Server, der zum Beispiel HTML-Seiten und Bilder verteilt. Das können andere Geräte aber besser und die Besonderheit des ESP32 liegt darin, dass er nicht nur als Web-Server taugt, sondern auch ein Mikrocontroller mit vielen interessanten Schnittstellen ist. Er ist also gerade prädestiniert dazu, Sensor-Daten per Web-Schnittstelle zu verteilen oder Geräte über einen Web-Browser zu steuern.

Ein kleines Beispielprojekt macht den ESP32 zu einem Web-Sensor und stellt allen Geräten im selben WLAN die aktuelle Temperatur und die relative Luftfeuchtigkeit zur Verfügung. Das Projekt wird schrittweise aufgebaut, um die Komplexität überschaubar zu halten. Zuerst wird der Sensor ausgelesen und dann werden dessen Daten über einen Web-Server verteilt.

Als kombinierter Sensor zum Erfassen der Temperatur und der Luftfeuchtigkeit kommt der HDC1008 (http://www.ti.com/product/hdc1008) zum Einsatz. Der HDC1008 kommuniziert per I2C und wird daher mit vier ESP32-Pins verbunden. VCC kommt an den 3,3V-Anschluss des ESP32 und GND natürlich an GND. Der SCL-Anschluss gehört an GPIO-Pin 22 und SDA an GPIO-Pin 21.

Zum Auslesen des Sensors dient die Adafruit HDC1000-Bibliothek, die sich über den Bibliotheksverwalter der Arduino-IDE installieren lässt. Weil der HDC1008 kompatibel zum HDC1000 ist, funktioniert die Bibliothek mit beiden Sensoren.

Listing SensorTest.ino nutzt den HDC1008, um die Temperatur und die relative Luftfeuchtigkeit zu messen und auf der seriellen Schnittstelle auszugeben. Dazu legt das Programm auf der globalen Ebene ein Objekt der Klasse `Adafruit_HDC1000` mit dem Namen `sensor` an. Dieses Objekt wird



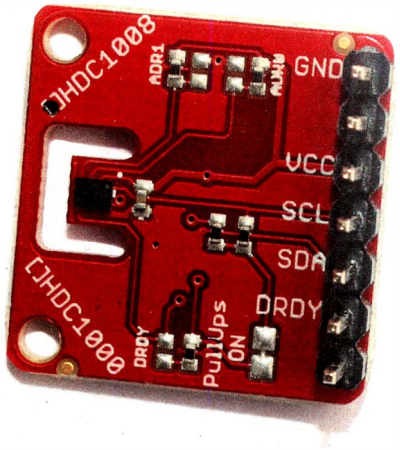
Der ESP32-WebServer liefert Seiten auch an Smartphones.

in der `setup`-Funktion mit der `begin`-Funktion initialisiert. Die Zahl 0x43 ist die I2C-Adresse des Sensors und die kann je nach Produkt durchaus variieren.

Es kommt bei Sensoren für Luftfeuchtigkeit schon mal vor, dass sich Feuchtigkeit in der Nähe des Sensors ablagert. Um die zu verdampfen, lassen sich manche Sensoren aufheizen und das tut die Funktion `drySensor`. Der Vorgang dauert circa 15 Sekunden, das heißt, das Programm gibt die Sensor-Daten erst nach dieser Wartezeit aus. Während der Entwicklung empfiehlt es sich daher, diese Zeile auszukommentieren, so dass man nicht nach jedem Neustart 15 Sekunden lang warten muss, bis etwas passiert.

Die Funktion `loop` gibt die Sensor-Informationen mittels der Funktionen `readTemperature` und `readHumidity` im Sekundentakt aus.

Der HDC1008 wird übrigens vom Hersteller nicht mehr empfohlen und Vergleichstests mit anderen Sensoren zeigen, dass die



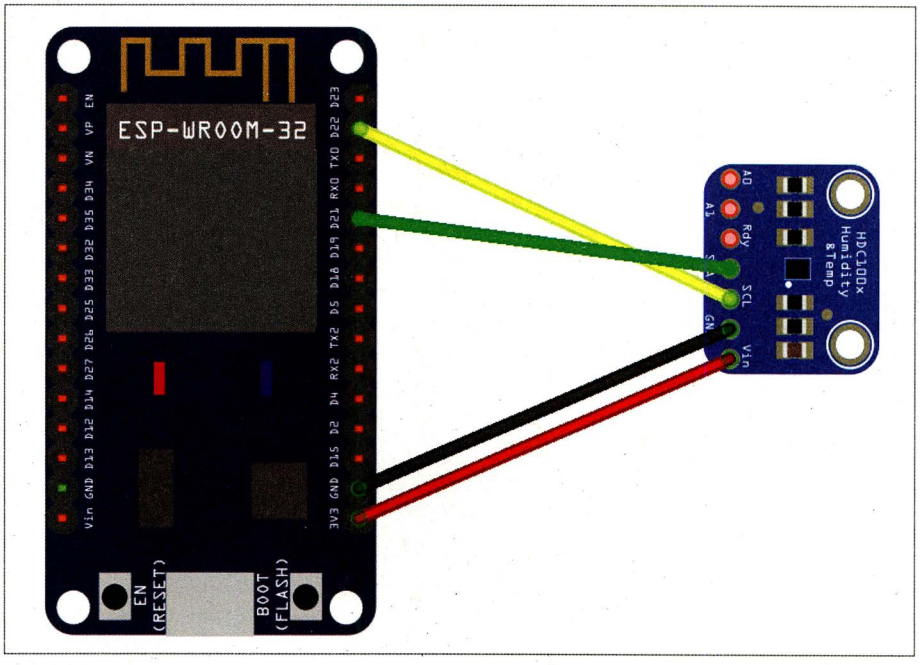
Mit einem Breakout-Board lässt sich der HDC1008 leicht mit dem ESP32 verbinden.

Werte für die Luftfeuchtigkeit teilweise deutlich von der Realität abweichen.

Kleine Schritte

Dank der Arduino-Bibliothek ist die Arbeit mit dem Sensor ein Leichtes und es ist ebenfalls kein großes Problem, die Daten im WLAN zu verteilen. Prinzipiell ginge das mit den schon beschriebenen Mechanismen, aber für dieses Projekt wird der ESP32 in einen Access Point verwandelt und es kommt eine neue WebServer-Bibliothek zum Einsatz.

Die Bibliothek heißt ESPAsyncWebServer (https://github.com/me-no-dev/ESPAsync-



So wird der ESP32 mit dem HDC1008 verbunden.

SensorTest.ino

```
#include "Adafruit_HDC1000.h"

Adafruit_HDC1000 sensor = Adafruit_HDC1000();

void setup() {
  Serial.begin(115200);

  if (!sensor.begin(0x43)) {
    Serial.println("Konnte Temperatursensor nicht finden.");
    return;
  }
  Serial.println("Sensor wird initialisiert.");
  sensor.drySensor(); // Braucht 15 Sekunden!
}

void loop() {
  Serial.printf("Temperatur: %.2f C\t", sensor.readTemperature());
  Serial.printf("Luftfeuchtigkeit: %.2f %%\n", sensor.readHumidity());
  delay(1000);
}
```

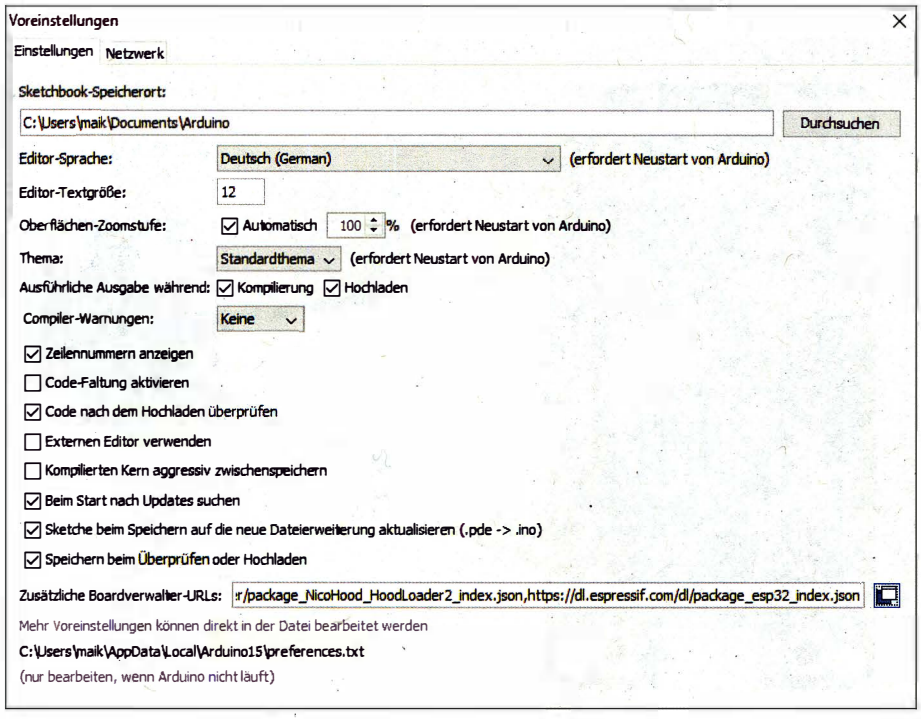
WebServer) und tut im Grunde dasselbe wie die bisherige WebServer-Bibliothek, das heißt, sie implementiert das HTTP-Protokoll und ermöglicht so die Kommunikation mit Web-Clients. ESPAsyncWebServer ist aber deutlich leistungsstärker, weil sie mehrere Anfragen parallel beantworten kann und ausgeklügelte Mechanismen zur Verarbeitung von Dateien hat.

Zurzeit gehört sie noch nicht zum Lieferumfang der Arduino-IDE und ist nicht über den Bibliotheksverwalter verfügbar. Hinzu kommt, dass sie die Bibliothek AsyncTCP (https://github.com/me-no-dev/AsyncTCP) benötigt, die ebenfalls nicht im Bibliotheksverwalter auftaucht. Beide Bibliotheken müssen daher manuell installiert werden. Am einfachsten erledigt man dies, indem man die Bibliotheken als ZIP-Dateien herunterlädt und über den Menüpunkt „ZIP-Bibliothek hinzufügen" in der Arduino-IDE importiert.

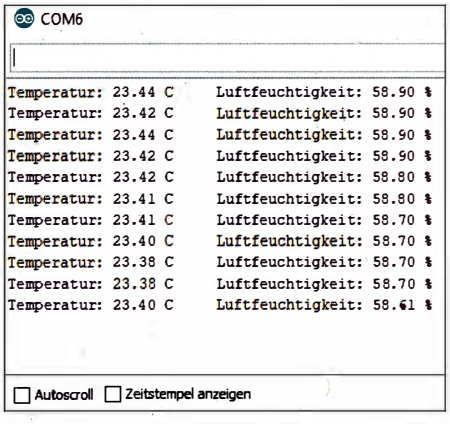
Dazu lädt man die Zip-Archive von ESPAsyncWebServer (https://github.com/me-no-dev/ESPAsyncWebServer/archive/master.zip) und AsyncTCP (https://github.com/me-no-dev/AsyncTCP/archive/master.zip) herunter und entpackt sie. Der Zusatz „-master" sollte aus beiden Dateinamen entfernt werden. Anschließend öffnet man für jede der genannten Bibliotheken einmal den Punkt „Sketch > Bibliothek einbinden > ZIP-Bibliothek einbinden" und wählt die jeweils umbenannte Datei aus.

Listing WebHDC1008.ino implementiert einen vollständigen Web-Server, der auf seinem eigenen WLAN-Access-Point arbeitet und allen anderen Teilnehmern im WLAN die Daten eines HDC1008-Sensors zur Verfügung stellt. Zwar gewinnt die Aufbereitung der Sensor-Daten keine Design-Preise, aber an dieser Stelle geht es erst einmal um die reine Machbarkeit.

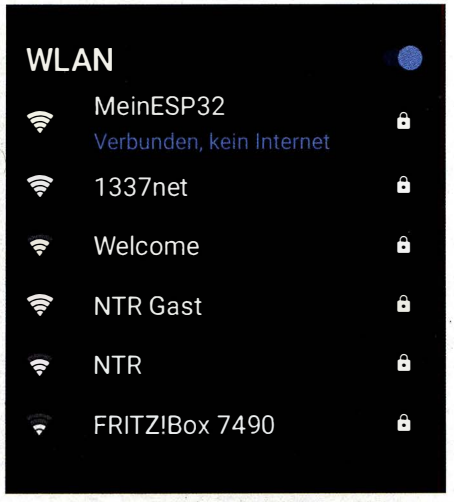
Der Code importiert alle benötigten Bibliotheken und definiert dann Konstanten für die SSID und das WLAN-Passwort. Das sieht zunächst alles wie gewohnt aus, aber in diesem Fall sind es nicht die SSID und das Passwort eines bestehenden WLANs, son-



Den Sketchbook-Speicherort findet man in den Voreinstellungen.



Der ESP32 kommt mit I2C-Sensoren prima zurecht.



Der ESP32 kann sein eigenes WLAN aufspannen.

dern es sind die Zugangsdaten für ein neues WLAN, das der ESP32 aufspannen wird. Diese Zugangsdaten benötigen also später alle Clients, die sich mit dem ESP32 verbinden wollen.

Bevor es so weit ist, werden auf der globalen Ebene zwei Objekte der Klassen `AsyncWebServer` und `Adafruit_HDC1000` angelegt. Letztere wird wie schon zuvor in der `setup`-Funktion initialisiert. Dort erfolgt auch die Initialisierung des WLANs und diesmal wird nicht die `begin`-Funktion der `WiFi`-Klasse verwendet, sondern die Funktion `softAP`. Die sorgt dafür, dass aus dem ESP32 ein Access Point mit den übergebenen Zugangsdaten wird.

Anschließend wird die IP-Adresse des Access Points mit `WiFi.softAPIP` ausgegeben. Die ist übrigens per Voreinstellung `192.168.4.1`, so dass sich ein ESP32, der einen Access Point betreibt, schnell finden lässt. Die Adresse lässt sich bei Bedarf leicht ändern.

Wie die Klasse `WebServer` arbeitet die Klasse `AsyncWebServer` mit Event-Handlern und verwendet sogar dieselben Funktionsnamen, um die Event-Handler zu registrieren. Die Parameter sind ganz ähnlich, unterscheiden sich aber im Detail. Wie zuvor steht der erste Parameter für den Pfad, für den der Event-Handler zuständig ist. In diesem Fall ist das der Pfad „/".

Der zweite Parameter ist neu und steht für den Typen der HTTP-Anfrage, die von diesem Event-Handler bearbeitet werden soll. HTTP definiert eine Reihe von Anfrage-Typen, die alle auf denselben Pfad angewendet werden können, aber eine andere Bedeutung haben. Für reguläre Anfragen, die keinerlei Daten verändern, verwendet man den Typen GET. Zum Löschen von Daten dient DELETE und Änderungen erfolgen über PUT oder POST. In diesem Fall sollen

WebHDC1008.ino

```
 1 #include <WiFi.h>
 2 #include <ESPAsyncWebServer.h>
 3 #include <Adafruit_HDC1000.h>
 4
 5 const char* SSID = "MeinESP32";
 6 const char* PASSWORD = "testpasswort";
 7
 8 AsyncWebServer server(80);
 9 Adafruit_HDC1000 sensor = Adafruit_HDC1000();
10
11 void setup(){
12   Serial.begin(115200);
13
14   if (!sensor.begin(0x43)) {
15     Serial.println("Konnte Temperatursensor nicht finden.");
16     return;
17   }
18   Serial.println("Sensor wird initialisiert.");
19   sensor.drySensor(); // Braucht 15 Sekunden!
20
21   WiFi.softAP(SSID, PASSWORD);
22   Serial.print("IP-Adresse: ");
23   Serial.println(WiFi.softAPIP());
24
25   server.on("/", HTTP_GET, [](AsyncWebServerRequest* request) {
26     String message = "<!doctype html>\n";
27     message += "<html>\n";
28     message += "  <head>\n";
29     message += "    <title>ESP32 Thermometer /
       Hygrometer</title>\n";
30     message += "  </head>\n";
31     message += "  <body>\n";
32     message += "    <h1>Temperatur: ";
33     message += String(sensor.readTemperature(), 2);
34     message += " &#8451;</h1>\n";
35     message += "    <h1>Luftfeuchtigkeit: ";
36     message += String(sensor.readHumidity(), 0);
37     message += " %</h1>\n";
38     message += "  </body>\n";
39     message += "</html>\n";
40     request->send(200, "text/html", message);
41   });
42
43   server.begin();
44 }
45
46 void loop() {}
```

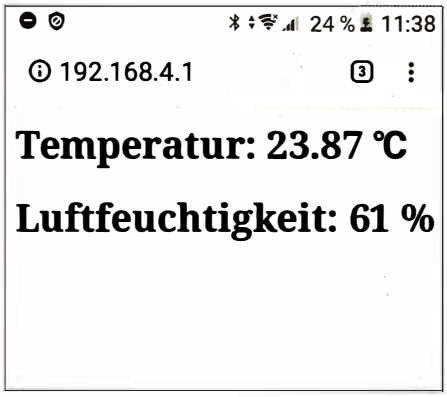
nur Daten gelesen werden und daher kommt `HTTP_GET` zum Einsatz.

Ferner erwartet die `on`-Funktion ebenfalls eine anonyme Funktion, die die eigentliche Arbeit übernimmt. Allerdings ist die anonyme Funktion in diesem Fall nicht parameterlos, sondern erwartet einen Zeiger auf ein Objekt der Klasse `AsyncWebServerRequest`. Das ist notwendig, weil die Klasse `AsyncWebServer` mehrere Anfragen gleichzeitig verarbeiten kann und somit potenziell mehr als ein Objekt der Klasse `AsyncWebServerRequest` im Speicher existiert. Der Pointer zeigt dann auf das jeweils richtige.

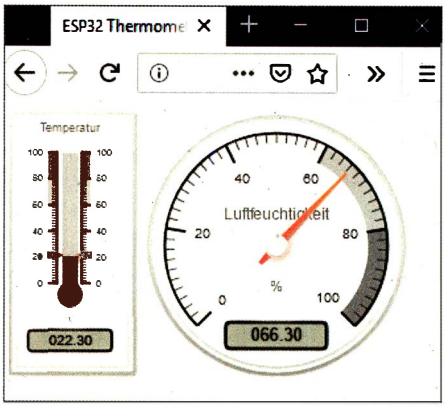
Wie bei der `WebServer`-Klasse wird das Ergebnis der Anfrage mit der `send`-Funktion an den Client gesendet. Diesmal gehört die Funktion zur Klasse `AsyncWebServerRequest`, erwartet aber dieselben Parameter, also den HTTP-Status-Code, den Content-Type und den Inhalt der Nachricht.

Der Inhalt der Nachricht ist diesmal zwar auch ein Text, aber er repräsentiert eine HTML-Seite. Die ist etwas komplexer und wird daher Zeile für Zeile mithilfe der Zeichenkette `message` aufgebaut. Sonderlich kompliziert ist der Inhalt der Seite nicht und der wesentliche Kunstgriff besteht darin, die Sensor-Daten dynamisch in den Quelltext der Seite einzubauen. So sieht der anfragende Client immer den aktuellen Stand der Dinge. Weil die Antwort an den Client eine HTML-Seite ist, muss der Content-Type auf text/html gesetzt werden. Andernfalls würde der Client lediglich den Quelltext der Seite anzeigen.

`server.begin` startet den WebServer und übernimmt die Behandlung der Anfra-



Die erste Version des Web-Sensors ist noch recht spartanisch.



Das macht schon mehr her als spröde Textausgaben.

gen, so dass die `loop`-Funktion diesmal komplett leer bleibt.

Andocken

Nachdem das Programm übersetzt und auf den ESP32 geladen wurde, führt der erste Weg ausnahmsweise nicht in den seriellen Monitor, denn dort gibt es nichts Nennenswertes zu sehen. Vielmehr lohnt es sich, auf dem PC oder auf dem Smartphone nach neuen WLAN-Netzen in der Umgebung zu suchen. Dort sollte nämlich ein Neuzugang mit dem Namen MeinESP32 auftauchen.

Das ist der Name des Access Points, den der ESP32 errichtet hat und mit dem kann man sich nun wie mit einem WLAN-Router verbinden. Das Passwort lautet `testpasswort` und sollte vor einem etwaigen Produktionsbetrieb dringend geändert werden!

Sobald man mit dem ESP32 verbunden ist, befindet man sich in dessen Netz. Das heißt zuerst einmal, dass es keinen Internetzugang gibt, denn der ESP32 ist ja nicht mit dem Internet verbunden. Das einzige Ziel, das man erreichen kann, ist der ESP32 selbst beziehungsweise der Web-Server, der auf dem Gerät auf Anfragen wartet. Wie oben bereits beschrieben lautet die IP-Adresse per Voreinstellung `192.168.4.1` und so führt der Aufruf von http://192.168.4.1/ zur HTML-Seite mit den aktuellen Sensor-Daten.

Ein paar Zeilen Code definieren einen Web-Server, der auf seinem eigenen WLAN-Access-Point läuft. Es ist schon erstaunlich, wie wenig notwendig ist, um eine solche Infrastruktur aufzubauen.

Jetzt wird's bunt

Obwohl das schon alles sehr vielversprechend aussieht, gibt es doch noch ein paar Mängel. Hauptsächlich ist die Optik der Web-Seite nicht sonderlich ansprechend, aber es ist bestimmt kein Vergnügen, einen ansprechenderen Auftritt in Form von Zeichenketten-Operationen direkt im Quelltext zu kodieren. Dankenswerterweise gibt es dafür weitaus bequemere und effizientere Alternativen.

Bevor die zum Tragen kommen, ist es ratsam, das gewünschte Ergebnis ganz herkömmlich auf dem PC oder dem Mac zu erstellen, denn dort ist die Erstellung von Web-Seiten mittlerweile keine große Herausforderung mehr. Insbesondere, weil es heute so viele nützliche CSS-Frameworks und Java Script-Bibliotheken gibt.

Für die Visualisierung von Daten ist das Projekt Canvas Gauges (https://canvas-gauges.com/) sehr interessant. Es ist frei verfügbar, läuft in so gut wie allen Browsern und bietet eine Fülle unterschiedlicher Anzeige-Instrumente. Listing index.html zeigt, wie sich Temperatur und Luftfeuchtigkeit im Handumdrehen schick anzeigen lassen.

Die HTML-Seite besteht im Wesentlichen aus zwei canvas-Elementen für das Thermometer und das Hygrometer. Beide Elemente werden über `data`-Attribute konfiguriert und die tatsächlichen Sensor-Werte gehören ins Attribut mit dem Namen `data-value`. Möglich macht dies die JavaScript-Bibliothek des Canvas-Gauges-Projekts, die in der Datei gauge.min.js gespeichert ist und aus Gründen der Bequemlichkeit über das Internet geladen wird. Im finalen Projekt müssen also die Inhalte der data-value-Attribute dynamisiert werden und die JavaScript-Datei muss aus einer anderen Quelle kommen, weil der ESP32 – und damit in der Regel auch der anfragende Client – keine Internet-Verbindung hat.

Es ist natürlich nur die halbe Miete, wenn das alles auf dem PC gut läuft, denn die erstellten HTML-, CSS- und JavaScript-Dateien lassen sich nicht ohne Weiteres auf den ESP32 kopieren, obwohl das Development-Board über ausreichend Flash-Speicher verfügt. Selbst wenn es so wäre, gäbe es keine offensichtliche Möglichkeit, sie von einem Programm aus zu lesen. Glücklicherweise gibt es für beide Probleme einfache Lösungen.

Der „Arduino ESP32 Filesystem Uploader" (https://github.com/me-no-dev/arduino-esp32fs-plugin) erweitert die Arduino-IDE um die Möglichkeit, beliebige Dateien in den Flash-Speicher des ESP32 zu kopieren. Dazu

index.html

```
<!doctype html>
<html>
  <head>
    <title>ESP32 Thermometer / Hygrometer</title>
  </head>
  <body>
    <canvas id='thermometer'
            data-width='100'
              data-height='200'
              data-type='linear-gauge'
              data-title='Temperatur'
              data-units='&#8451;'
              data-value='22.3'>
    </canvas>
    <canvas id='hygrometer'
            data-width='200'
            data-height='200'
              data-type='radial-gauge'
              data-title='Luftfeuchtigkeit'
              data-units='&#37;'
              data-value='66.3'>
    </canvas>
    <script src='http://cdn.rawgit.com/Mikhus/canvas-gauges/
    gh-pages/download/2.1.5/all/gauge.min.js'>
    </script>
  </body>
</html>
```

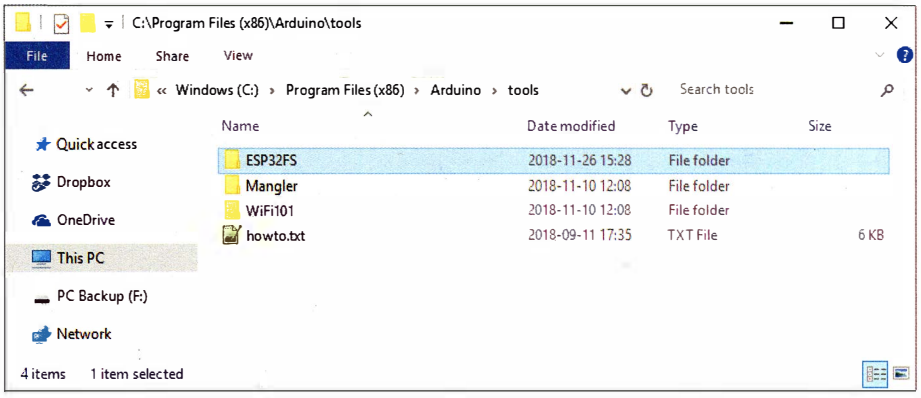
muss das Plug-in zuerst heruntergeladen (https://github.com/me-no-dev/arduino-esp32fs-plugin/releases/download/v0.1/ESP32FS-v0.1.zip) und installiert werden.

Installation ist dabei ein hochtrabender Begriff, denn die Zip-Datei des Plug-ins muss nur ausgepackt und ins tools-Verzeichnis der Arduino-Installation kopiert werden. Nach einem Neustart der IDE taucht im Menü Werkzeuge ein neuer Eintrag namens „ESP32 Sketch Data Upload" auf.

Dieser Menüpunkt ist nicht ganz einfach zu verstehen und setzt ein wenig Wissen voraus. Alle Dateien, die zu einem Programm gehören und zusammen mit dem Code des Programms auf den ESP32 geladen werden sollen, müssen in einem Verzeichnis namens data im Programm-Verzeichnis abgelegt werden. In diesem Fall sind das die Dateien index.html und gauge.min.js.

Sobald aber alle benötigten Dateien im data-Verzeichnis liegen, genügt ein Klick auf den neuen Menüpunkt und schon werden sie in den Flash-Speicher übertragen und stehen dort unter ihren Originalnamen zur Verfügung.

Die Datei index.html (siehe Seite 26) wurde geringfügig modifiziert und enthält jetzt die Platzhalter `%TEMPERATURE%` und `%HUMIDITY%`, die im laufenden Betrieb jeweils durch die Sensor-Daten ersetzt werden müssen. Ferner wird die Datei gauge.min.js nicht mehr von der Download-Seite des Canvas-Gauges-Projekts geladen, sondern muss im selben Verzeichnis liegen wie index.html.



So sieht es aus, nachdem das Filesystem-Plug-in installiert wurde.

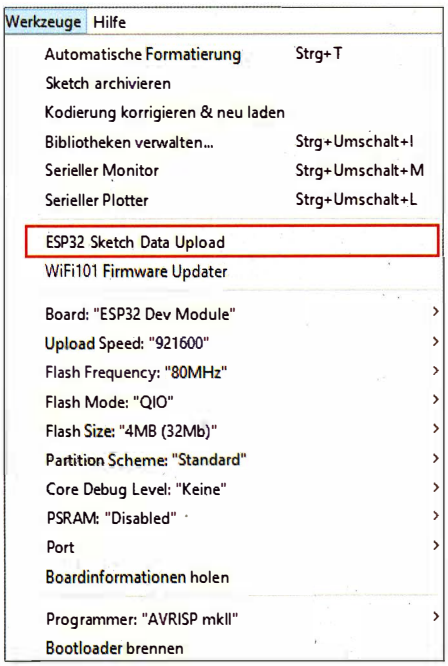
Endspurt

Jetzt gilt es nur noch, die einzelnen Komponenten zu einem großen Ganzen zu verbinden und das erledigt Listing WebHDC1008Pro.ino. Es ist eine leicht modifizierte Version des Vorgänger-Programms, ist aber um Längen flexibler und deutlich einfacher zu erweitern.

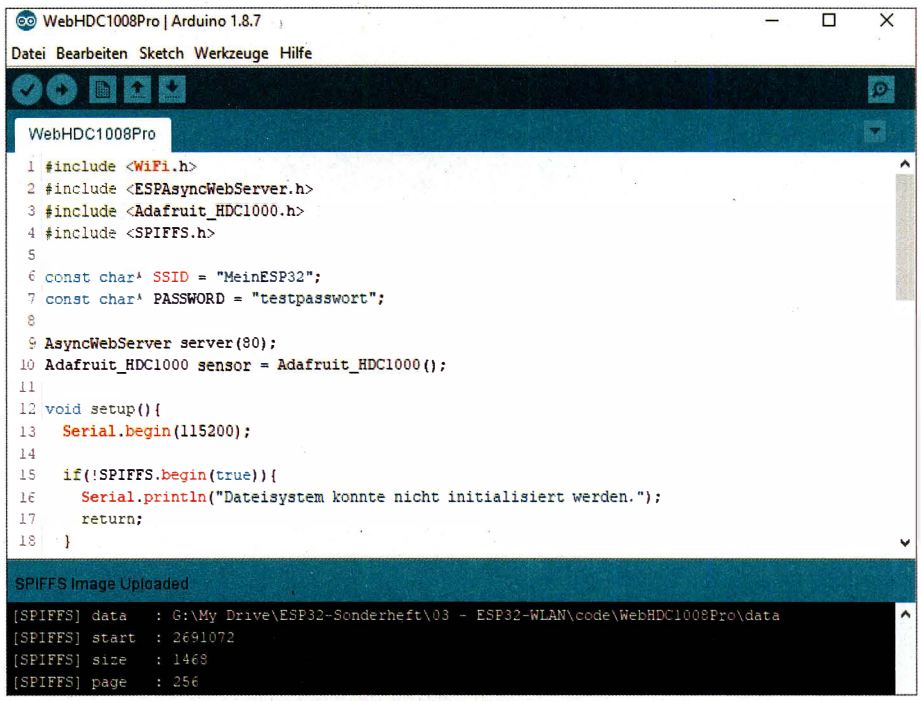
Zu den bisher verwendeten Bibliotheken wird am Anfang des Programms noch die SPIFFS-Bibliothek (Serial Peripheral Interface Flash File System) eingebunden. Sie macht es möglich, Teile des Flash-Speichers als Dateisystem zu betrachten und dort Dateien zu erzeugen, zu lesen und zu modifizieren. Die Bibliothek wird in der `setup`-Funktion durch den Aufruf von `SPIFFS.begin` initialisiert. Das Argument `true` bewirkt, dass im Falle eines Fehlers das Dateisystem formatiert wird. In der Regel passieren Fehler nur, wenn zuvor noch nie ein Dateisystem angelegt wurde und so ist dies das gewünschte Verhalten.

Weitere Änderungen gibt es bei der Definition der Event-Handler. Der erste kümmert sich um die Datei gauge.min.js. Statt die Datei in den Quelltext einzubetten, wird sie jetzt mit der Klasse SPIFFS aus dem Flash-Speicher gelesen. Den Content-Type ermittelt die AsyncWebServer-Bibliothek in diesem Fall automatisch. Er kann aber auch explizit gesetzt werden.

Ähnlich erfolgt die Auslieferung der Datei index.html. Die kann allerdings nicht unverändert an den Client weitergegeben werden,



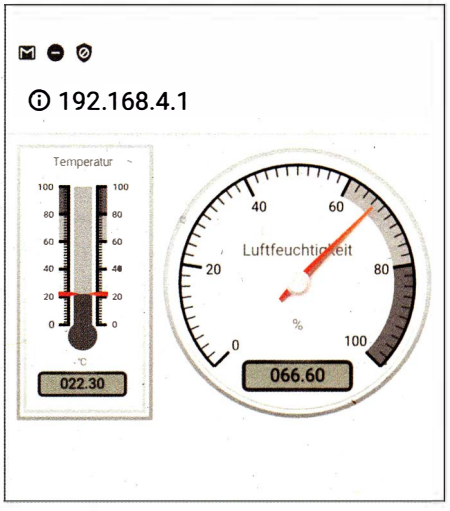
Das Filesystem-Plug-in erweitert die Arduino-IDE um einen Menüpunkt.



Dateien lassen sich einfach per Plug-in auf den ESP32 laden.

WebHDC1008/Proindex.html

```
<!doctype html>
<html>
  <head>
    <title>ESP32 Thermometer / Hygrometer</title>
  </head>
  <body>
     <canvas id='thermometer'
             data-width='100'
               data-height='200'
               data-type='linear-gauge'
               data-title='Temperatur'
               data-units='&#8451;'
               data-value='%TEMPERATURE%'>
     </canvas>
    <canvas id='hygrometer'
             data-width='200'
             data-height='200'
               data-type='radial-gauge'
               data-title='Luftfeuchtigkeit'
               data-units='&#37;'
               data-value='%HUMIDITY%'>
     </canvas>
    <script src='gauge.min.js'>
    </script>
  </body>
</html>
```



So sieht die fertige Webseite auf dem Smartphone aus.

weil sie noch Platzhalter für die Sensor-Werte enthält. Praktischerweise unterstützt die `AsyncWebServerRequest`-Klasse die Ersetzung von Platzhaltern, die durch zwei Prozentzeichen eingerahmt werden. Das passt perfekt zu %TEMPERATURE% und %HUMIDITY% in der Datei index.html.

Für die Ersetzung ruft `AsyncWebServerRequest` für jeden Platzhalter eine Funktion auf, die man zuvor registrieren muss. In diesem Fall ist es die Funktion `replaceVariable`. Für jeden Platzhalter erfolgt ein Aufruf von `replaceVariable` mit dem Namen des Platzhalters. Die Funktion liefert den zu ersetzenden Wert zurück und der wird in die Zeichenkette, die den Datei-Inhalt repräsentiert, geschrieben. Dabei wird die Datei im Flash-Speicher nicht verändert.

Für einen finalen Test wird das Programm übersetzt und auf den ESP32 geladen. Ferner müssen die dazugehörigen Dateien über den Menüpunkt Werkzeuge > ESP32 Sketch Data Upload in den Flash-Speicher kopiert werden. Diesen Schritt darf man bei späteren Änderungen nicht vergessen, sonst drohen lange Debug-Sitzungen.

Die hier verwendeten Techniken zeigen übrigens nur einen kleinen Ausschnitt der Möglichkeiten, die die AsyncWebServer-Bibliothek bietet. Sie kann noch sehr viel mehr, und ein genaues Studium der Dokumentation liefert gewiss Ideen für neue Projekte.

Fazit

Es ist oft die Rede davon, wie leistungsstark die Hardware des ESP32 ist, aber erst wenn man die WLAN-Fähigkeiten des Geräts ausprobiert, bekommt man ein gutes Gefühl dafür, was wirklich in dem Winzling steckt. Mühelos verbindet er sich mit allem, was sich so an WLANs findet und hat darüber hinaus genug Reserven, um noch sinnvolle Dinge zu tun.

Hinzu kommt, dass die entsprechenden Bibliotheken es leicht machen, anspruchsvolle Anwendungen innerhalb kürzester Zeit zu erstellen. Drahtlose IoT-Projekte waren noch nie so einfach und günstig zu haben. *—dab*

WLAN-SICHERHEIT

Die meisten Netze dürften vor unbefugtem Zugang geschützt sein und für diesen Schutz gibt es unterschiedliche Protokolle. Die sichern nicht nur den Zugang, sondern sorgen auch dafür, dass alle Daten, die über die Luftschnittstelle wandern, verschlüsselt werden.

Seit der ersten WLAN-Spezifikation gab es im Wesentlichen drei Verfahren zur Absicherung: WEP (Wired Equivalent Privacy), WPA (Wi-Fi Protected Access) und WPA2 (Wi-Fi Protected Access Version 2). Schon lange gilt WEP als unsicher und sollte nicht mehr verwendet werden. Viel besser sieht die Lage bei WPA auch nicht aus und selbst WPA2 ist nicht unangreifbar. Noch scheint zumindest WPA2 ausreichend sicher zu sein und es ist das beste Verfahren, das der Standard derzeit bietet. Wie bei fast allen Verfahren zur Zugangskontrolle entscheidet nicht zuletzt die Qualität der verwendeten Passwörter über die Sicherheit des Gesamtsystems.

WPA und WPA2 gibt es jeweils in zwei Ausprägungen. Die erste heißt Personal oder PSK (Pre-Shared Key) und ist für den Einsatz daheim oder in kleinen Netzwerken gedacht. In diesem Szenario teilen sich alle Netzteilnehmer einen gemeinsamen Schlüssel mit einer Länge von 256 Bit. Jeder Teilnehmer leitet zur Absicherung der eigenen Kommunikation von diesem Schlüssel einen Schlüssel mit einer Länge von 128 Bit ab. Weil jeder Teilnehmer im WLAN somit einen eigenen Schlüssel hat, ist das Abhören untereinander nicht möglich, auch wenn alle den gleichen PSK verwenden.

Für das Enterprise-Verfahren benötigt man zusätzlich einen separaten Authentifizierungsserver, den in der Regel nur größere Unternehmen betreiben.

WebHDC1008Pro.ino

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <Adafruit_HDC1000.h>
#include <SPIFFS.h>

const char* SSID = "MeinESP32";
const char* PASSWORD = "testpasswort";

AsyncWebServer server(80);
Adafruit_HDC1000 sensor = Adafruit_HDC1000();

void setup(){
  Serial.begin(115200);

  if(!SPIFFS.begin(true)){
    Serial.println("Dateisystem konnte nicht initialisiert werden.");
    return;
  }

  if (!sensor.begin(0x43)) {
    Serial.println("Konnte Temperatursensor nicht finden.");
    return;
  }
  Serial.println("Sensor wird initialisiert.");
  sensor.drySensor(); // Braucht 15 Sekunden!

  WiFi.softAP(SSID, PASSWORD);
  Serial.print("IP-Adresse: ");
  Serial.println(WiFi.softAPIP());

  server.on("/gauge.min.js", HTTP_GET, [](AsyncWebServerRequest* request) {
    request->send(SPIFFS, "/gauge.min.js");
  });

  server.on("/", HTTP_GET, [](AsyncWebServerRequest* request) {
    request->send(SPIFFS, "/index.html", String(), false, replaceVariable);
  });

  server.begin();
}

void loop() {}

String replaceVariable(const String& var) {
  if (var == "HUMIDITY")
    return String(sensor.readHumidity(), 0);
  if (var == "TEMPERATURE")
    return String(sensor.readTemperature(), 2);
  return String();
}
```

Strahlendblauer Funkenflug

Was die Kommunikation mit der Außenwelt angeht, ist der ESP32 ein wahres Multitalent. Neben WLAN spricht er nämlich auch fließend Bluetooth, was ihn zu einer exzellenten Basis für viele IoT-Projekte macht.

von Maik Schmidt

Alles zum Artikel im Web unter *make-magazin.de/xa4r*

Genau wie WLAN hilft Bluetooth bei der drahtlosen Kommunikation zwischen Geräten. Während WLAN in erster Linie als Übertragungstechnik für beliebige Dienste und Protokolle dient, wurde Bluetooth für eine Vielzahl konkreter Anwendungen konzipiert. Vornehmlich geht es bei Bluetooth darum, Kabelverbindungen zwischen Geräten durch Funkverbindungen zu ersetzen. Das betrifft unter anderem Tastaturen, Mäuse, Kopfhörer, Lautsprecher oder Smartphones. Der ESP32 unterstützt den Bluetooth-Standard in der Version 4.2, was nicht zuletzt bedeutet, dass er prinzipiell mit allen zuvor genannten Geräte-Typen kommunizieren kann.

Der aktuelle Bluetooth-Standard zerfällt grob in zwei Teile, die sich stark voneinander unterscheiden. Auf der einen Seite gibt es das klassische Bluetooth, das zur Kopplung zweier Geräte dient, die langfristig und über kurze Distanzen miteinander kommunizieren. Gebräuchliche Anwendungen sind die Freisprecheinrichtung im Auto oder die drahtlose Übertragung von Musik vom Smartphone zum Kopfhörer. Bei diesen Anwendungen sind Sender und Empfänger permanent aktiv und benötigen eine Menge Energie, um die Kommunikation aufrechtzuhalten.

Darüber hinaus gibt es aber auch Bluetooth Low Energy (BLE), das speziell die Bedürfnisse von IoT-Geräten (Internet of Things) adressiert. BLE eignet sich zum Beispiel hervorragend für mobile Sensoren, die ihre Daten an alle Empfänger in der näheren Umgebung senden. Ebenso ermöglicht BLE Ortung und Navigation innerhalb von Gebäuden.

Bluetooth-Geräte müssen nicht beide Teile des Standards implementieren. Das heißt, es gibt Geräte, die nur die klassische Variante oder nur BLE anbieten. Viele Geräte implementieren aber beides und der ESP32 gehört dazu. Für beide Formen der Kommunikation gibt es eine Vielzahl unterschiedlicher Einsatzmöglichkeiten.

Kabellos auf Draht

Eine der einfachsten und sinnvollsten Anwendungen des klassischen Bluetooth ist der Aufbau einer drahtlosen seriellen Verbindung. Ursprünglich wurde Bluetooth sogar entwickelt, um eine RS232-Schnittstelle ohne Kabel zu realisieren. Programmtechnisch funktioniert diese Schnittstelle sowohl aus Sicht des Clients als auch aus der Sicht des Servers wie eine drahtgebundene, aber sie kommt ohne eine physikalische Verbindung aus.

Erfreulicherweise ist die serielle Kommunikation Bestandteil der Bluetooth-Spezifikation und so gibt es Bibliotheken, wie zum Beispiel BluetoothSerial, die es kinderleicht machen, zwei Geräte miteinander zu verbinden. Dabei ist es völlig egal, um welche Geräte es sich handelt, solange sie eine Bluetooth-Einheit haben und auf die serielle Schnittstelle zugreifen können.

SerialBluetooth.ino

```
1  #include "BluetoothSerial.h"
2
3  BluetoothSerial bluetooth;
4
5  void setup() {
6    bluetooth.begin("ESP32");
7  }
8
9  void loop() {
10   bluetooth.println("Hier bin ich!");
11   delay(1000);
12 }
```

Listing SerialBluetooth.ino verwandelt den ESP32 in ein Bluetooth-Gerät, das Nachrichten über eine drahtlose serielle Schnittstelle verteilt. Dazu bindet das Programm zunächst die BluetoothSerial-Bibliothek ein und definiert ein globales Objekt der Klasse `BluetoothSerial` mit dem Namen `bluetooth`. Dieses Objekt initialisiert die `setup`-Funktion mittels der Funktion `begin` und übergibt ihr dabei die Zeichenkette ESP32. Das ist der Name, unter dem der ESP32 erscheint, wenn man nach Bluetooth-Geräten in der Umgebung sucht.

Die `loop`-Funktion gibt anschließend im Sekundentakt einen Text auf der seriellen Bluetooth-Schnittstelle aus. Das ist genauso einfach, wie die Ausgabe über die herkömmliche serielle Schnittstelle, der Text wird aber per Funk übertragen.

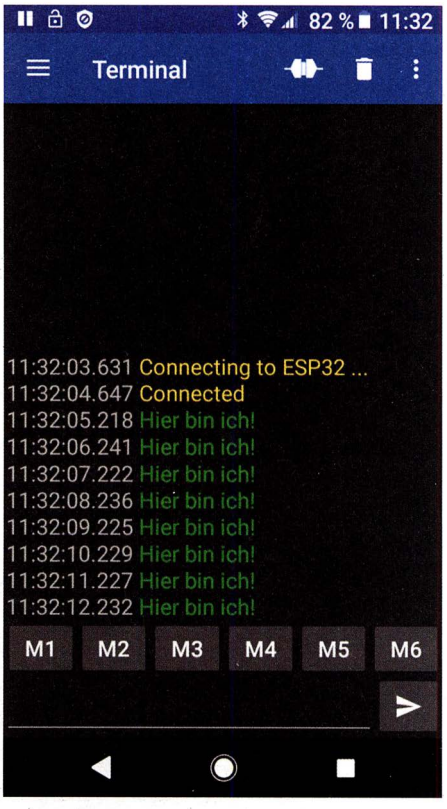
Gegenüber

Jetzt fehlt noch ein Empfänger und dazu eignet sich jedes Gerät, das eine serielle Schnittstelle via Bluetooth etablieren kann und über ein serielles Terminal verfügt. Prinzipiell geht das also zum Beispiel mit fast allen PCs und Smartphones.

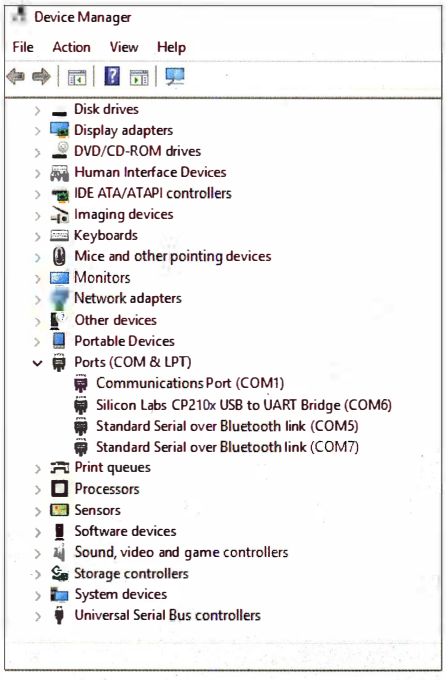
Es gibt tatsächlich recht gute serielle Terminals für Smartphones und sie lassen sich in der Regel einfach bedienen. Die Android-App Serial Bluetooth Terminal (https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal) kommt beispielsweise ohne großen Schnickschnack daher und zeigt die Meldungen des ESP32 an, sobald man das Gerät gekoppelt hat. Wichtig ist natürlich, zuvor die Bluetooth-Funktion des Smartphones auch zu aktivieren.

Selbstverständlich funktioniert die serielle Kommunikation auch auf dem PC oder dem Mac. Moderne Betriebssysteme erstellen nämlich beim Koppeln mit einem Bluetooth-

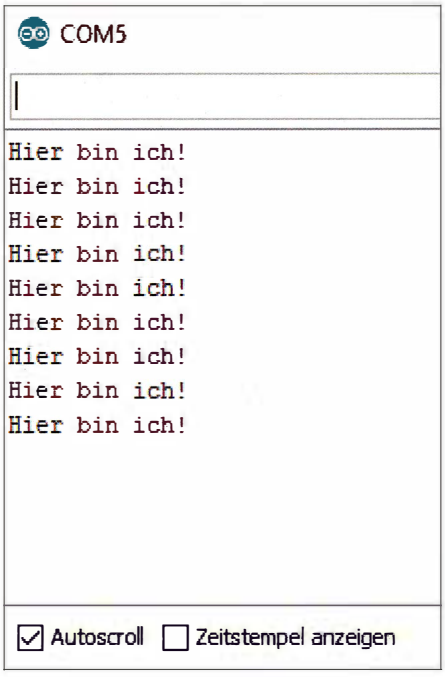
Der ESP32 erscheint als ein ganz normales Bluetooth-Gerät.



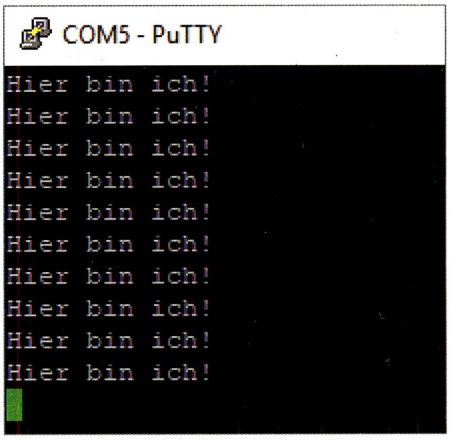
Smartphone-Terminals sind unverzichtbare Helfer bei der Bluetooth-Entwicklung.



Windows erzeugt gleich zwei COM-Ports für den ESP32.



Bluetooth-Nachrichten kann man auch im seriellen Monitor der Arduino-IDE lesen.



Ausgaben funktionieren in PuTTY problemlos.

Gerät automatisch eine serielle Schnittstelle, über die man mit dem Gerät kommunizieren kann. Unter Windows findet man im Geräte-Manager nach der Kopplung mit dem ESP32 gleich zwei neue COM-Ports. Die Gründe dafür sind technischer Natur und für erste Experimente ist der erste COM-Port die richtige Wahl.

Jetzt benötigt man nur noch ein serielles Terminal, um die Nachrichten des ESP32 zu lesen beziehungsweise um ihm neue Nachrichten zu senden. Am einfachsten geht das mit dem seriellen Monitor der Arduino-IDE. Dort wählt man im Menü Werkzeuge > Port die serielle Schnittstelle der Bluetooth-Verbindung aus und schon trudeln die Nachrichten ein.

Allzu komfortabel ist der serielle Monitor der Arduino-IDE nicht und es wäre in vielen Fällen auch ein wenig übertrieben, die ganze Arduino-IDE nur für den seriellen Monitor zu installieren. Sinnvoller ist es, ein Programm wie PuTTY (https://www.putty.org/) zu installieren, das genau für diesen Zweck erstellt wurde. Neben diversen anderen Protokollen unterstützt PuTTY nämlich auch die serielle Kommunikation und man kann sie direkt nach dem Start zusammen mit der seriellen Schnittstelle auswählen. Sobald man die Verbindung durch einen Klick auf den Open-Button geöffnet hat, gibt PuTTY munter die Nachrichten aus, die per Bluetooth empfangen werden.

Vice Versa

Das Ganze funktioniert auch andersherum, das heißt, man kann über die Schnittstelle auch Nachrichten an den ESP32 senden. Listing SerialBluetoothBidirectional.ino liest zum Beispiel alle Daten, die über die Bluetooth-Schnittstelle hereinkommen und gibt sie direkt wieder auf der regulären seriellen Schnittstelle aus.

Um zu sehen, wie das funktioniert, muss man zwei serielle Terminals öffnen. Den seriellen Monitor öffnet man in der Arduino-IDE wie gewohnt für die drahtgebundene Schnittstelle des ESP32. Parallel öffnet man PuTTY und stellt den COM-Port der Bluetooth-Schnittstelle ein. Alle Nachrichten, die man dann im Eingabefenster von PuTTY eingibt, erscheinen sofort im seriellen Monitor der Arduino-IDE.

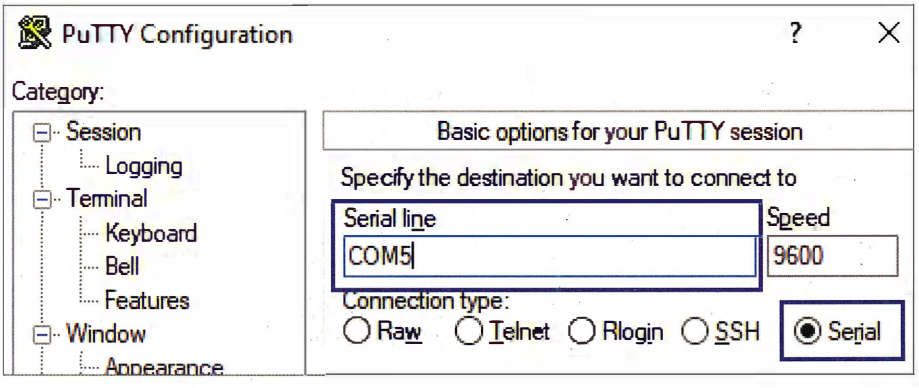
Mit der Kommunikation in beide Richtungen lässt sich natürlich eine Menge anfangen. Zum Beispiel kann der ESP32 Sensordaten per Funk übertragen. Der ESP32 kann aber auch Kommandos empfangen, und Listing BluetoothSwitch.ino implementiert einen primitiven Lichtschalter. Empfängt der ESP32 eine 1, so schaltet er die Status-LED auf dem Developer-Board ein. Sendet man eine 0, schaltet er sie wieder aus.

Dank einer weitsichtigen Spezifikation und robuster Implementierungen in modernen Betriebssystemen unterscheidet sich der Zugriff auf Bluetooth kaum vom Zugriff auf drahtgebundene serielle Schnittstellen. Der ESP32 wird so im Handumdrehen zu einem vollwertigen Bluetooth-Gerät, das sich auf vielfältige Weise einsetzen lässt.

Natürlich muss die Kommunikation nicht über ein serielles Terminal erfolgen. Das geht genauso gut mit selbst entwickelten Anwendungen in beinahe jeder Programmiersprache, denn so gut wie jede Programmiersprache kann auf serielle Schnittstellen zugreifen.

Das geht auch smarter

Das klassische Bluetooth funktioniert auf dem ESP32 prima, aber für viele IoT-Anwen-



PuTTY kommt gut mit seriellen Verbindungen zurecht.

dungen ist Bluetooth Low Energy (BLE) deutlich besser geeignet. Das liegt zum einen am geringen Stromverbrauch, aber auch an der Art und Weise, wie die Kommunikation bei BLE abgewickelt wird. Während in der klassischen Variante Geräte gekoppelt werden müssen und immer nur über eine Point-to-Point-Verbindung kommunizieren können, ermöglicht BLE Broadcast-Nachrichten und Mesh-Netzwerke. Optimiert wurde BLE für das schnelle Versenden und Empfangen kleiner Datenpakete über Distanzen bis zu 40 Metern.

BLE-Geräte benötigen keine permanente und energiehungrige Verbindung. Sie schlafen die meiste Zeit über und kommunizieren nur bei Bedarf. Für Fitness-Tracker, medizinische Anwendungen oder viele Geräte zur Heimautomatisierung ist das ideal, denn sie müssen nicht unentwegt auf Sendung sein und können so sehr lange von einer Knopfzelle zehren.

Die vielfältigen Anwendungsmöglichkeiten machen es notwendig, Datenformate und Protokolle weitestgehend zu standardisieren, aber trotzdem noch genug Raum für Neuentwicklungen zu lassen. Das geschieht bei BLE mit relativ einfachen Mitteln, die im Detail aber einigermaßen komplex sind.

Erfreulich ist, dass die Beteiligten einer BLE-Kommunikation nur Server oder Clients sein können. Während BLE-Server ihre Umgebung in regelmäßigen Abständen beschallen und über ihre Existenz informieren, suchen BLE-Clients umgekehrt nach BLE-Servern, mit denen sie spielen können. Haben sie was Interessantes gefunden, bauen sie eine Verbindung auf, um mehr Informationen zu bekommen.

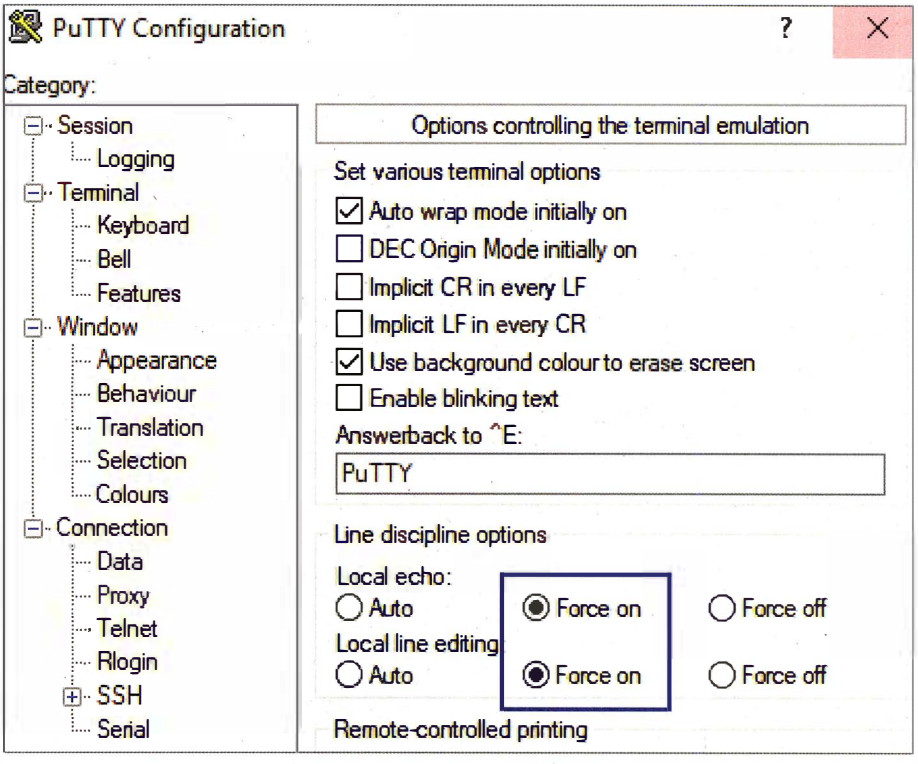
Profile suchen

Doch wie kann ein Client feststellen, ob ein bestimmter Server überhaupt interessante Informationen anbietet und ob der Client überhaupt in der Lage ist, diese zu verstehen? Beispielsweise könnte eine Smartphone-App ja nach Fitness-Trackern in der Umgebung suchen. Die App wäre in diesem Fall der Client und der Fitness-Tracker der Server. Wie können die beiden sicherstellen, dass sie dieselbe Sprache sprechen und Daten austauschen können? Sicherlich wäre es nicht hilfreich, wenn jeder Fitness-Tracker die aktuelle Pulsfrequenz in seiner eigenen Kodierung senden würde.

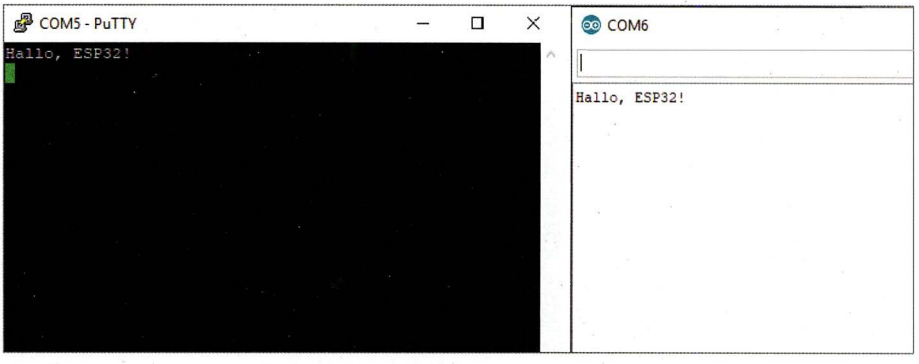
Die Lösung des Problems sind die Bluetooth-Profile. Davon gibt es gleich mehrere Dutzend und sie legen fest, über welche Fähigkeiten ein Gerät verfügt. Mäuse und Tastaturen implementieren zum Beispiel das Human Interface Device Profile (HID), und wer eine serielle Schnittstelle anbietet, im-

SerialBluetoothBidirectional.ino

```
#include "BluetoothSerial.h"

BluetoothSerial bluetooth;

void setup() {
  Serial.begin(115200);
  if (!bluetooth.begin("ESP32")) {
    Serial.println("Bluetooth konnte nicht aktiviert werden.");
  } else {
    Serial.println("Der ESP32 kann gekoppelt werden.");
  }
}

void loop() {
  while (bluetooth.available()) {
    Serial.write(bluetooth.read());
  }
  delay(20);
}
```



Damit man in PuTTY auch etwas sehen kann, muss man ein paar Einstellungen ändern.



Hier kommunizieren zwei serielle Terminals über Bluetooth miteinander.

BlueToothSwitch.ino

```
#include "BluetoothSerial.h"

const uint8_t LED_PIN = 2;

BluetoothSerial bluetooth;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(115200);
  if (!bluetooth.begin("ESP32")) {
    Serial.println("Bluetooth konnte nicht aktiviert werden.");
  } else {
    Serial.println("Der ESP32 kann gekoppelt werden.");
  }
}

void loop() {
  while (bluetooth.available()) {
    const int c = bluetooth.read();
    Serial.write(c);
    if (c == '1') {
      digitalWrite(LED_PIN, HIGH);
    } else if (c == '0') {
      digitalWrite(LED_PIN, LOW);
    }
  }
  delay(20);
}
```

BLUETOOTH UNTER LINUX

Auch unter Linux kann man per Bluetooth mit dem ESP32 seriell Daten austauschen, allerdings ist dazu mehr Handarbeit nötig. Unter Ubuntu 16.04 gelang es, den ESP32 zu koppeln. Allerdings fragt Linux unter Umständen nach einer Pin, die man auf dem ESP32 eingeben soll. Ignorieren Sie die Aufforderung und klicken einfach auf „Fortfahren". Anschließend können Sie über das Tool rfcomm per `sudo rfcomm connect 1 uu:vv:ww:xx:yy:zz 2` eine serielle Verbindung mit 9600 Baud auf der Schnittstelle /dev/rfcomm1 aufbauen.

Die Beispiele mit BLE in Chrome funktionierten bei uns leider nicht.

plementiert das Serial Port Profile (SPP). Smartphones implementieren oft eine ganze Reihe von Profilen, wie zum Beispiel das Advanced Audio Distribution Profile (A2DP), das Headset Profile (HSP), das Hands Free Profile (HFP) und das Phonebook Access Profile (PBAP). In Kombination ermöglichen diese Profile das bequeme Telefonieren im Auto.

Profil zeigen

So gut wie alle Profile sind optional, aber alle Geräte müssen das Generic Access Profile (GAP) implementieren, denn das realisiert die grundlegenden Zugriffsmechanismen und die Verwaltung von Verbindungen. Übrigens lassen sich Profile per Software nachrüsten, was insbesondere für den ESP32 interessant ist. Dass es für bestimmte Profile heute noch keine Implementierung gibt, bedeutet also nicht, dass es in Zukunft auch keine geben wird.

Das Konzept der Profile wurde schon in den Anfängen der Standardisierung definiert. Weil aber für BLE eine ganze Menge neuer Dienste zu erwarten ist, sieht der Bluetooth-Standard dafür ein übergreifendes Profil mit dem Namen Generic Attribute Profile (GATT; https://www.bluetooth.com/specifications/gatt) vor. Unterhalb dieses Profils können Hersteller flexibel weitere Dienste (Services) definieren.

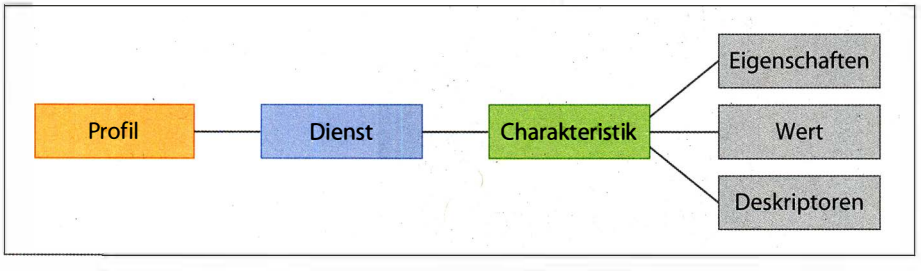
Die Liste der BLE-Dienste ist schon jetzt recht lang und vielfältig. Angefangen bei einem Dienst, der über den aktuellen Ladezustand der Batterie informiert, bis hin zu einem Dienst, der kontinuierlich den Blutzuckerspiegel überwacht, ist schon eine ganze Menge dabei.

Um mit BLE-Diensten zu arbeiten, ist es wichtig, Profile und deren Aufbau zu verstehen. Profile bilden eine Hierarchie und können aus einem oder mehreren Diensten bestehen. Dienste wiederum bestehen aus einer oder mehreren Charakteristiken. Eine Charakteristik setzt sich aus verschiedenen Eigenschaften, einem Wert und einer Liste von Deskriptoren zusammen.

Die Eigenschaften einer Charakteristik definieren, auf welche Weise Clients auf die Charakteristik zugreifen können. Beispielsweise können manche Charakteristiken gelesen und/oder geschrieben werden. Beim Schreiben gibt es zwei unterschiedliche Spielarten, nämlich eine, bei der dem Client eine Quittung übermittelt wird („write request"), und eine, bei der das nicht der Fall ist („write command" oder „write without response").

Ferner können Clients sich automatisch über Veränderungen des Wertes einer Charakteristik informieren lassen. Die dazugehörigen Eigenschaften heißen Notifikation („notify") und Indikation („indicate"). Eine Notifikation muss der Client nicht quittieren, eine Indikation hingegen schon. Darüber hinaus gibt es noch eine ganze Reihe weiterer Eigenschaften, wie zum Beispiel die Broadcast-Eigenschaft, die festlegt, dass der Wert einer Charakteristik auch via Broadcast bekannt gegeben werden kann.

Zusätzlich zu den Eigenschaften speichert die Charakteristik den tatsächlichen Wert und eine Liste optionaler Deskriptoren, die die Charakteristik noch weiter beschreiben.



BLE-Profile haben eine hierarchische Struktur.

Beispielsweise können Deskriptoren Wertebereiche einschränken oder Ausgabeformate definieren.

Das klingt alles halbwegs kompliziert, ist in der Praxis aber ganz einfach. Konkret definiert der Bluetooth-Standard im GATT-Profil zum Beispiel einen Dienst namens „Battery Service" für den Ladezustand der Batterie. Die Charakteristik des Dienstes hat die Lese-Eigenschaft und optional die Notify-Eigenschaft. Der Dienst liefert einen vorzeichenlosen 8-Bit-Wert zwischen 0% und 100%, der den Ladezustand der Batterie repräsentiert.

Im Dschungel zurechtfinden

Schon heute gibt es eine große Anzahl an Diensten, Charakteristiken und Deskriptoren, und in Zukunft werden gewiss noch ein paar hinzukommen. Damit Clients schnell und sicher feststellen können, ob ein Server einen bestimmten Dienst anbietet, haben alle Bluetooth-Entitäten nicht nur einen Namen, sondern werden darüber hinaus auch mit einer UUID (Universally Unique Identifier) versehen.

Eine UUID ist eine Zahl mit 128 Bit, also 16 Bytes. Diese Zahlen können mithilfe diverser Algorithmen erzeugt werden und können für viele Anwendungsfälle als eindeutig betrachtet werden. Zwar ist es durchaus möglich, dass ein und dieselbe UUID zweimal erzeugt wird, aber dieser Fall ist eher unwahrscheinlich und für viele Anwendungen auch nicht relevant.

Dargestellt werden UUIDs in der Regel als Hexadezimalzahlen mit 32 Stellen. Zur Verbesserung der Lesbarkeit hat sich das Schema 8-4-4-4-12 durchgesetzt und so sieht eine typische UUID wie folgt aus: 123e4567-e89b-12d3-a456-426655440000. In dieser Darstellung benötigt die UUID wegen der Bindestriche 36 Zeichen.

Zur Erzeugung von UUIDs gibt es jede Menge Tools auf der Kommandozeile oder in Form von Internet-Diensten. Auch bieten die meisten Programmiersprachen Bibliotheken zur Generierung von UUIDs an. Wer also einen neuen Bluetooth-Dienst entwickelt, kann sich schnell und kostenlos die dazu benötigten UUIDs besorgen.

Der Bluetooth-Standard verwendet zur Identifizierung von Diensten und Charakteristiken übrigens UUIDs, die nur 16 Bits lang sind. Das macht die Spezifikation etwas übersichtlicher und es vereinfacht auch die Identifizierung von Standard-Diensten für Clients und Server. Beispielsweise hat der Battery Service die Kurz-UUID 0x180F.

Hat's gefunkt?

Nach der ganzen Theorie implementiert Listing BleServer.ino mit wenig Aufwand einen BLE-Server. Der definiert einen neuen Dienst inklusive einer Charakteristik und bietet ihn Clients in der Umgebung an.

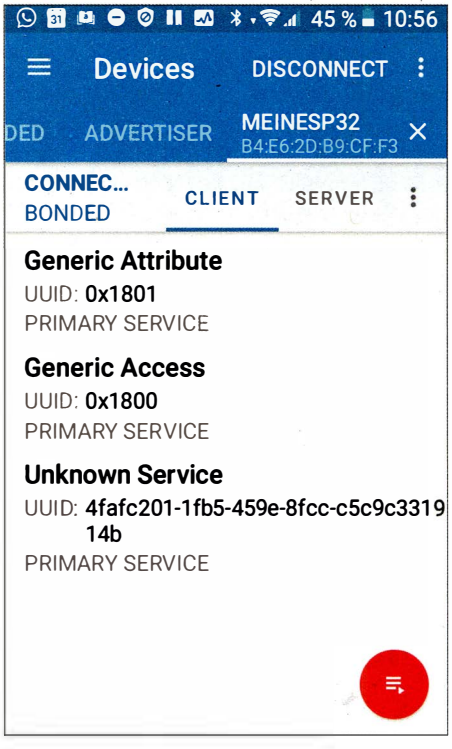
BleServer.ino

```
1  #include <BLEDevice.h>
2
3  const char* SERVICE_UUID = "4fafc201-1fb5-459e-8fcc-c5c9c331914b";
4  const char* CHARACTERISTIC_UUID = "beb5483e-36e1-4688-b7f5-
   ea07361b26a8";
5
6  void setup() {
7    BLEDevice::init("MeinESP32");
8    BLEServer* server = BLEDevice::createServer();
9    BLEService* service = server->createService(SERVICE_UUID);
10   BLECharacteristic* characteristic = service->
     createCharacteristic(
11     CHARACTERISTIC_UUID,
12     BLECharacteristic::PROPERTY_READ | BLECharacteristic::
       PROPERTY_WRITE);
13   characteristic->setValue("Hallo, Make-Magazin!");
14   service->start();
15   server->startAdvertising();
16 }
17
18 void loop() {
19   delay(2000);
20 }
```

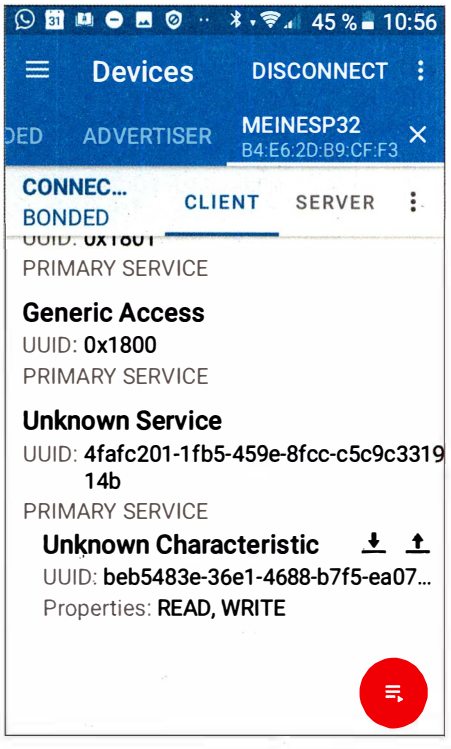
Das Programm bindet die BLEDevice-Bibliothek ein, die alles bietet, um BLE-Server und Clients in vielen verschiedenen Ausprägungen zu schreiben. Erfreulicherweise finden sich dazu auch einige Beispiele in der Arduino-IDE.

Auf der globalen Ebene definiert das Programm zwei Konstanten für die UUID des Dienstes und der Charakteristik. Diese UUIDs können beliebig gewählt werden, weil der Dienst kein Standard-Dienst ist und auch keine Standard-Charakteristik verwendet.

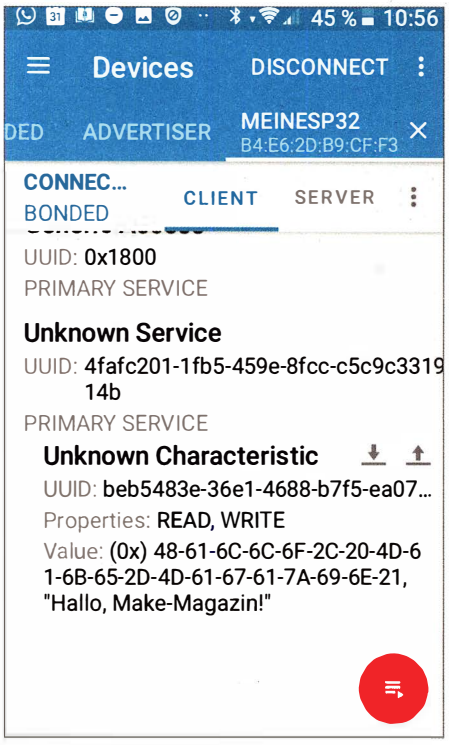
Die eigentliche Arbeit findet in der `setup`-Funktion statt und sie macht mit der Funktion `BLEDevice::init` den ESP32 im Bluetooth-



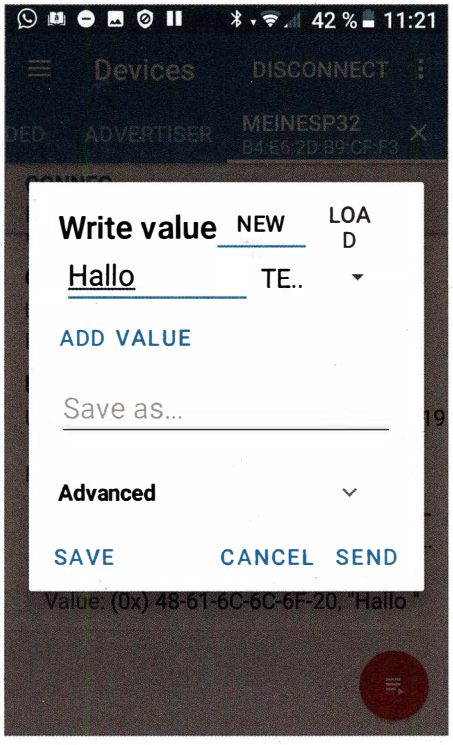
nRF Connect findet BLE-Dienste in der Umgebung zuverlässig.



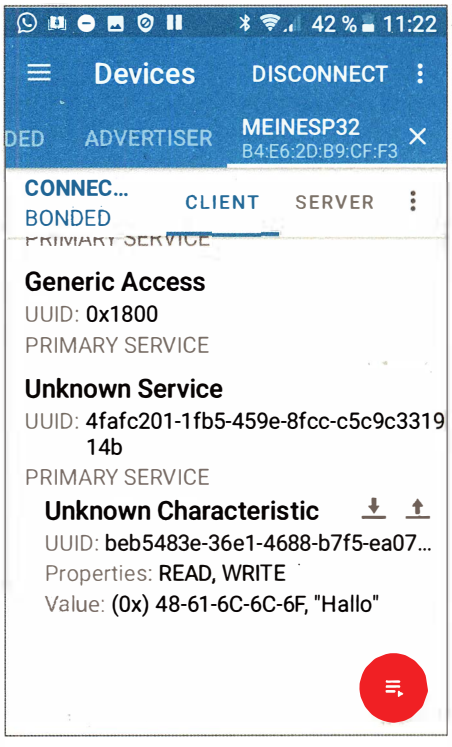
Die eigens definierte Charakteristik sieht exakt so aus, wie sie definiert wurde.



Auch der Wert der Charakteristik stimmt.



Neue Werte zu setzen ist auch kein Problem.



Der neue Wert steht sofort zur Verfügung.

Spektrum unter dem Namen MeinESP32 bekannt. Anschließend erzeugt sie einen neuen Server und für den Server einen neuen Dienst. Dem Dienst wird dabei die zuvor festgelegte UUID zugewiesen. Zu beachten ist hierbei, dass die meisten Funktionen der BLE-Bibliothek Zeiger auf Objekte zurückgeben. Beispielsweise liefert `BLEDevice::createServer` einen Zeiger auf ein `BLEServer`-Objekt.

Im weiteren Verlauf erzeugt das Programm eine Charakteristik mit der zu Anfang festgelegten UUID und den Schreib- und Lese-Eigenschaften. Die Charakteristik erhält den Wert „Hallo, Make-Magazin!“.

Schließlich wird sowohl der Dienst als auch der Server gestartet. Ab diesem Moment können Clients in der Umgebung den neuen Dienst finden. Weil all diese Aktionen im Hintergrund ausgeführt werden, kann sich die `loop`-Funktion schlafen legen.

Um das Programm zu testen, benötigt man einen BLE-Client und dazu eignet sich ein Smartphone am besten. Allerdings fehlt noch eine passende App und während der Entwicklung ist es sinnvoll, einen generischen Bluetooth-Client zu verwenden, der minutiös darüber Auskunft gibt, was sich auf der Luftschnittstelle gerade so tut. Sowohl unter iOS als auch unter Android ist nRF Connect eine gute Wahl. Wichtig ist es, sowohl Bluetooth als auch BLE in den Einstellungen des Smartphones zu aktivieren, wenn das noch nicht geschehen ist.

Sucht man in der App nach Geräten in der Umgebung, sollte der ESP32 unter dem Namen MeinESP32 auftauchen. Die Funktion Connect verbindet die App dann mit dem ESP32. Der neu definierte Dienst trägt den Namen „Unknown Service“, weil seine UUID nicht zu den bekannten Service-UUIDs gehört. Über die UUID, die im Programm vergeben wurde, lässt er sich aber trotzdem einwandfrei erkennen.

Ein Klick auf den Dienst zeigt die Charakteristik des Dienstes an. Auch sie lässt sich über die UUID identifizieren und hat die Eigenschaften Lesen (READ) und Schreiben (WRITE). Ein Klick auf den kleinen Pfeil, der nach unten zeigt, fördert den aktuellen Wert zutage. Mit dem Icon, dessen Pfeil nach oben weist, lässt sich ein neuer Wert festlegen. Dazu muss man zuerst den Typ des Werts auswählen, was in diesem Fall Text ist. Zwar kann man den Wert auch in Form hexadezimal kodierter Bytes eintippen, aber das ist schon etwas umständlich.

Hat man einen neuen Wert eingegeben, wird er direkt übernommen. Schritt für Schritt kann man sich so in einem unbekannten BLE-Dienst orientieren oder aber prüfen, ob der eigene Dienst sich Clients gegenüber genauso präsentiert, wie es gedacht war.

Geht das auch in schön?

Zu Debuggingzwecken sind Apps wie nRF Connect unverzichtbare Helfer, denn sie geben bis aufs Bit genau darüber Auskunft, was in der näheren Umgebung gerade die Funkwellen reitet. Ferner bieten sie eine generische Benutzungsschnittstelle, mit der sich zum Beispiel Werte beliebiger Datentypen lesen und schreiben lassen.

Für reguläre Benutzer und reale Anwendungen taugen Apps wie nRF Connect hingegen nicht. Hier muss schon eine eigene Anwendung her und das ist dank aktueller Technologien nicht mehr ganz so kompliziert wie noch vor ein paar Jahren.

Zu einem guten und nützlichen Client gehört aber auch ein sinnvoller Server. Listing BleServer.ino veranschaulicht zwar gut, wie man eigene Charakteristiken definieren kann, aber die gesendeten Daten sind proprietär und können von Werkzeugen wie nRF Connect nur angezeigt und nicht interpretiert werden. Daher wäre ein BLE-Server hilfreich, der auf einen der Standard-Dienste der Bluetooth-Spezifikation setzt.

Als Beispiel-Anwendung drängt sich geradezu auf, die Werte eines HDC1008-Sensors diesmal nicht per WLAN und HTTP, sondern per Bluetooth Low Energy zu verteilen. Wenig überraschend definiert der Bluetooth-Standard bereits einen Dienst für solche Umweltsensoren. Der Dienst hat den Namen Environmental Sensing und die ID 0x181A. Innerhalb des Dienstes gibt es eine lange Liste möglicher Charakteristiken und die für Temperatur und Luftfeuchtigkeit haben die IDs 0x2A6E beziehungsweise 0x2A6F. Weitere Sensoren lassen sich bei Bedarf leicht nachrüsten.

Listing BleSensor.ino implementiert einen BLE-Sensor, der die Daten eines HDC1008-Sensors kontinuierlich ausliest und an seine Umgebung sendet. Die Übertragung erfolgt mithilfe des Standard-Dienstes, so dass alle Clients, die dessen Protokolle implementieren, die Daten verstehen können.

Das Programm ist nicht übermäßig lang, nutzt aber einige neue Funktionen und Funktionalitäten. Der Anfang wirkt jedoch vertraut, denn dort werden nur alle benötigten Bibliotheken eingebunden und ein globales Objekt der Klasse `Adafruit_HDC1000` definiert, das zum Auslesen des HDC1008-Sensors dient.

Dann folgen weitere globale Variablen, die größtenteils zur Definition der BLE-Charakteristiken dienen. Die ersten beiden sind Instanzen der Klasse `BLEDescriptor`. Sie werden im weiteren Verlauf verwendet, um die Charakteristiken mit textuellen Beschreibungen zu versehen, die der Client den Benutzern anzeigen kann. Dazu sieht der Standard die ID 0x2901 vor, die zur Konstruktion der beiden Objekte verwendet wird. Allerdings kann die Kurzform der UUID nicht direkt verwendet werden, sondern muss mit dem Makro `BLEUUID` in die richtige Form gebracht werden.

Die zwei Objekte der Klasse `BLECharacteristic` werden auf ähnliche Weise konstruiert, dienen aber einem gänzlich anderen Zweck. Sie werden zur Definition der Charakteristiken des Dienstes benötigt und enthalten später neben den Deskriptoren auch die Informationen über die Temperatur und die Luftfeuchtigkeit. Der Standard sieht für die Temperatur die ID 0x2A6E und für die Luftfeuchtigkeit die ID 0x2A6F vor. Zusätzlich zur UUID muss die Charakteristik aber auch festlegen, auf welche Weise Clients auf die Daten zugreifen können. In diesem Fall ist das lesend oder per Notifikation möglich.

Callbacks

Die Boolesche Variable `bleClientConnected` zeigt den aktuellen Verbindungsstatus an. Wenn der ESP32 gerade mit einem Client verbunden ist, hat sie den Wert `true`, sonst den Wert `false`. Das funktioniert nicht automatisch, sondern muss mithilfe von Callback-Funktionen erledigt werden. Die definiert die Klasse `MyServerCallbacks`, die von der Klasse `BLEServerCallBacks` abgeleitet wird.

Die Klasse enthält die Funktionen `onConnect` und `onDisconnect`, die immer dann aufgerufen werden, wenn eine Verbindung mit einem Client hergestellt beziehungsweise abgebaut wurde. Sie setzen den Wert der Variablen `bleClientConnected` entsprechend. Beide Funktionen erhalten einen Zeiger auf das `BLEServer`-Objekt, das die Verbindung auf- beziehungsweise abgebaut

BleSensor.ino

```
 1 #include <BLEDevice.h>
 2 #include <BLEServer.h>
 3 #include <BLEUtils.h>
 4 #include <BLE2902.h>
 5 #include <Adafruit_HDC1000.h>
 6
 7 Adafruit_HDC1000 sensor = Adafruit_HDC1000();
 8
 9 BLEDescriptor humidityDescriptor(BLEUUID((uint16_t)0x2901));
10 BLEDescriptor temperatureDescriptor (BLEUUID((uint16_t)0x2901));
11
12 BLECharacteristic humidityCharacteristic(
13   BLEUUID((uint16_t)0x2A6F),
14   BLECharacteristic::PROPERTY_READ | BLECharacteristic::
     PROPERTY_NOTIFY);
15 BLECharacteristic temperatureCharacteristic(
16   BLEUUID((uint16_t)0x2A6E),
17   BLECharacteristic::PROPERTY_READ | BLECharacteristic::
     PROPERTY_NOTIFY);
18
19 bool bleClientConnected = false;
20
21 class MyServerCallbacks : public BLEServerCallbacks {
22   void onConnect(BLEServer* server) {
23     bleClientConnected = true;
24   };
25
26   void onDisconnect(BLEServer* server) {
27     bleClientConnected = false;
28   }
29 };
30
31 void setup() {
32   Serial.begin(115200);
33   if (!sensor.begin(0x43)) {
34     Serial.println("Konnte Temperatursensor nicht finden.");
35     //return;
36   }
37   Serial.println("Sensor wird initialisiert.");
38   // sensor.drySensor(); // Braucht 15 Sekunden!
39
40   BLEDevice::init("ESP32-SENSOR");
41   BLEServer* server = BLEDevice::createServer();
42   server->setCallbacks(new MyServerCallbacks());
43   BLEService* service = server->
     createService(BLEUUID((uint16_t)0x181A));
44
45   humidityDescriptor.setValue("Luftfeuchte 0-100%");
46   humidityCharacteristic.addDescriptor(&humidityDescriptor);
47   humidityCharacteristic.addDescriptor(new BLE2902());
48   service->addCharacteristic(&humidityCharacteristic);
49
50   temperatureDescriptor.setValue("Temperatur -40-125°C");
51   temperatureCharacteristic.addDescriptor(&temperatureDescriptor);
52   temperatureCharacteristic.addDescriptor(new BLE2902());
53   service->addCharacteristic(&temperatureCharacteristic);
54
55   service->start();
56   server->startAdvertising();
57 }
58
59 void loop() {
60   if (bleClientConnected) {
61     const uint16_t humidity = 50;
62     const int16_t temperature = 50;
63     humidityCharacteristic.setValue((uint8_t*)&humidity, 2);
64     humidityCharacteristic.notify();
65     temperatureCharacteristic.setValue((uint8_t*)&temperature, 2);
66     temperatureCharacteristic.notify();
67   }
68   delay(500);
69 }
```

hat. Für den vorliegenden Anwendungsfall wird dieser Parameter aber nicht benötigt.

Damit der ganze Mechanismus funktioniert, muss noch ein Objekt der Klasse `MyServerCallbacks` beim entsprechenden `BLEServer`-Objekt registriert werden. Darum kümmert sich die `setup`-Funktion, die aber erst einmal die serielle Schnittstelle und den HDC1008-Sensor initialisiert. Hier gibt es keinerlei Unterschiede zur WLAN-Variante und wie auch dort empfiehlt es sich, den Aufruf der Funktion `drySensor` während der Entwicklung auszukommentieren.

Nur geringe Unterschiede gibt es bei der Initialisierung der Bluetooth-Kommunikation. Diesmal operiert der ESP32 unter dem Namen ESP32-SENSOR und wie angekündigt wird das Objekt der Klasse `MyServerCallbacks` beim BLE-Server registriert. Anschließend wird ein Dienst mit der ID 0x181A (Environmental Sensing) erzeugt.

Dem Dienst selbst werden dann zwei Charakteristiken – eine für die Temperatur und eine für die Luftfeuchtigkeit – zugewiesen. Die Charakteristiken enthalten wiederum jeweils zwei Deskriptoren. Der erste Deskriptor ist vom Typ 0x2901 (Characteristic User Description) und beschreibt den Inhalt des Sensorwerts etwas genauer. Zum Beispiel können Clients für den Temperatur-Wert den Text „Temperatur -40-125°C“ anzeigen. Damit können Nutzer etwas mehr über den Sensor und seine Eigenschaften erfahren. Der Text kann frei gewählt werden und muss daher nicht unbedingt sinnvolle Informationen enthalten.

Der zweite Deskriptor hat die ID 0x2902 und die steht für Client Characteristic Configuration (CCCD). Mit diesem Deskriptor hat der Client die Möglichkeit, das Sendeverhalten des Servers zu steuern. Im vorliegenden Fall kann der Server Sensor-Daten über den Notifikationsmechanismus an einen verbundenen Client verteilen. Weil dieses Verhalten, bei dem potenziell viele Datenpakete gesendet werden, für manche Clients unerwünscht ist, lässt es sich über den CCCD ein- und ausschalten.

Nachdem die beiden Charakteristiken konfiguriert wurden, startet die `setup`-Funktion sowohl den Dienst als auch den Server.

Die `loop`-Funktion trägt jetzt nur noch in regelmäßigen Abständen die aktuellen Sensor-Daten in die Charakteristiken ein. Dazu prüft sie zunächst, ob überhaupt ein Client mit dem Server verbunden ist. Ist dies der Fall, liest sie die Sensor-Daten aus, multipliziert sie mit 100 und weist sie Variablen vom Typ `uint16_t` beziehungsweise `int16_t` zu. Dabei werden etwaige Nachkommastellen abgeschnitten.

Diese Form der Aufbereitung ist notwendig, weil die Messwerte gemäß der Spezifikation in exakt diesem Format übertragen werden müssen. Das heißt, sowohl die Temperatur als auch die Luftfeuchtigkeit jeweils als zwei Bytes, die eine vorzeichenbehaftete und eine vorzeichenlose Zahl repräsentieren, versendet. Die Klasse `BLECharacteristic` hat daher eine `setValue`-Funktion, die neben einem `uint8_t`-Zeiger auch noch die Anzahl der zu übertragenden Bytes erwartet.

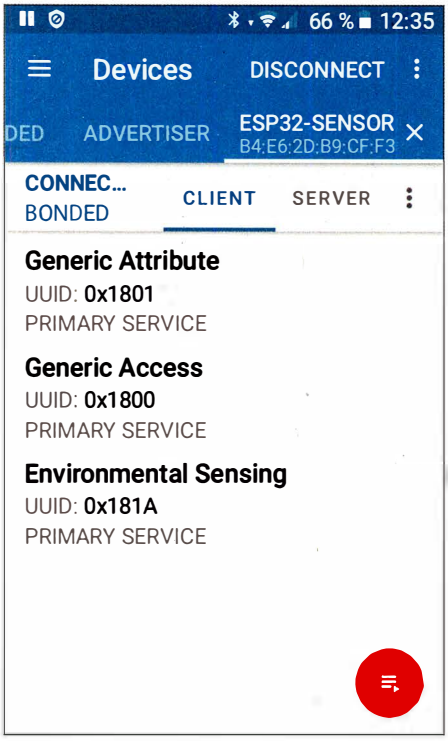
Schließlich sorgt die notify-Funktion dafür, dass die neuen Werte auch über die Luftschnittstelle bekannt gemacht werden.

Ein erster Blick

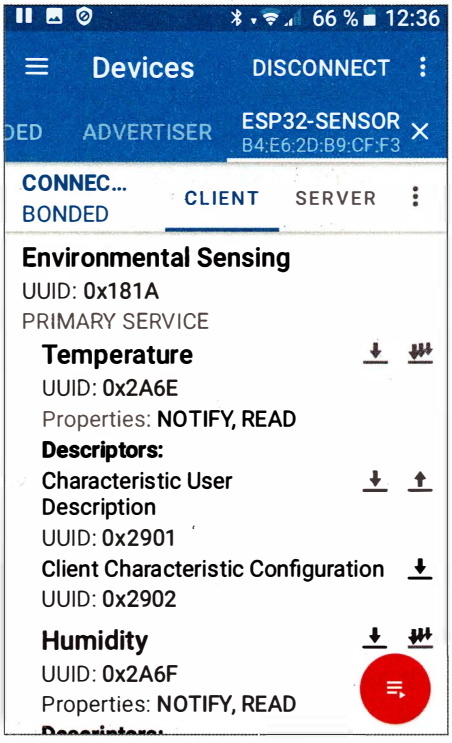
Das Programm ist nicht allzu lang und tut im Grunde auch nichts Spektakuläres und doch ist die implizite Komplexität hoch. Bevor man also einen passenden Client baut, ist es ratsam, mit generischen Werkzeugen, wie zum Beispiel nRF Connect, sicherzustellen, dass auch die richtigen Bits übertragen werden.

Das ist schnell erledigt und nachdem man sich mit dem Gerät namens ESP32-SENSOR verbunden hat, zeigt nRF Connect an, dass es den Environmental Sensing-Dienst gefunden hat. Wenn man den durch einen Klick expandiert, erscheinen die Charakteristiken für die Temperatur und die Luftfeuchtigkeit.

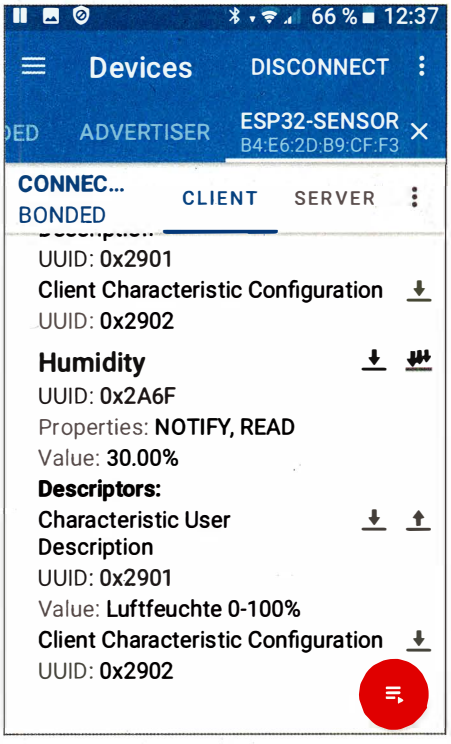
Die Charakteristiken selbst lassen sich weiter aufklappen und so kann man die zuvor festgelegten Beschreibungen aus den Deskriptoren und die aktuellen Sensorwerte lesen. Es gibt sogar ein Icon, mit dem sich der Notifikationsmechanismus aktivieren lässt,



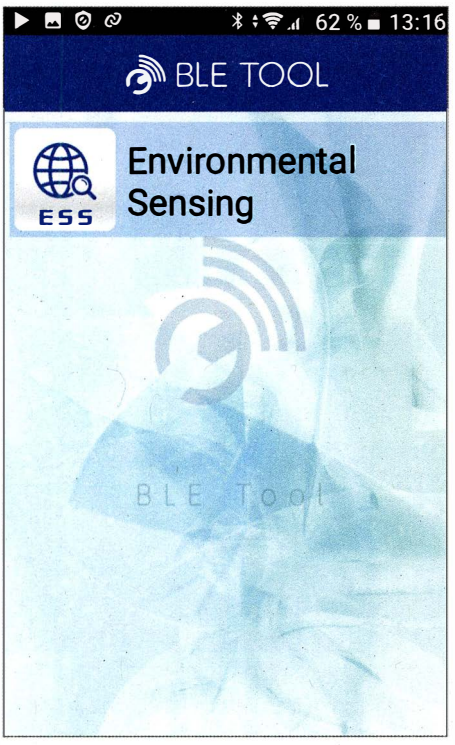
nRF Connect erkennt den Environmental-Sensing-Dienst auf dem ESP32 sofort.



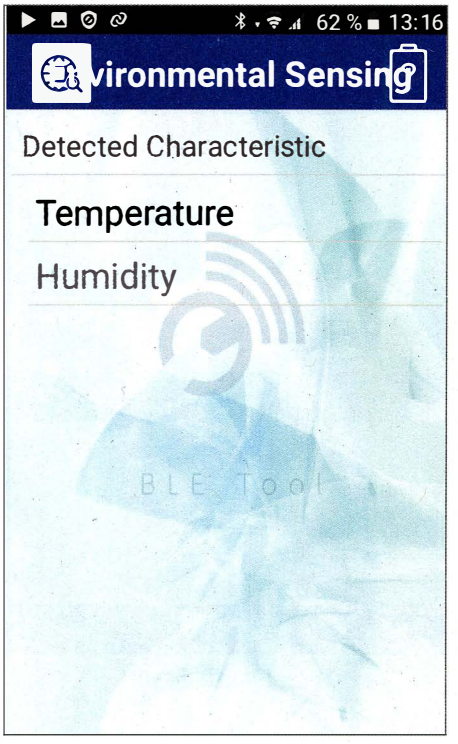
nRF Connect kann auch die gesendeten Charakteristiken interpretieren.



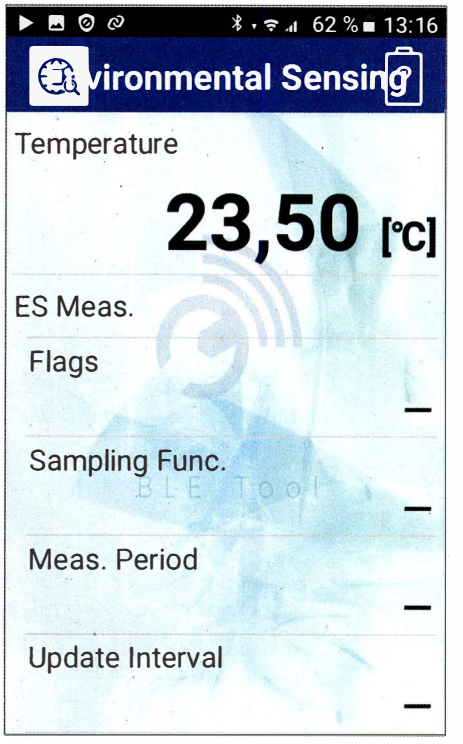
Die Sensor-Daten zeigt nRF Connect ebenfalls an.



BLE Tool erkennt sofort, dass auf dem ESP32 der Environmental-Sensing-Dienst läuft.



Beide Charakteristiken werden von BLE Tool erkannt.



BLE Tool kann nur die Temperatur lesen, bereitet sie aber nett auf.

so dass die Werte automatisch aktualisiert werden.

Es gibt auch Apps, wie zum Beispiel BLE Tool, die Standard-Dienste etwas ansprechender visualisieren können. BLE Tool spielt klaglos mit dem ESP32 zusammen und erkennt den Environmental-Sensing-Dienst und seine beiden Charakteristiken. Zurzeit kann die App aber nur die Temperatur anzeigen.

Marke Eigenbau

Letzten Endes wird man für die meisten Projekte aber eine eigene Anwendung benötigen, die im günstigsten Fall nicht nur auf verschiedenen Smartphones, sondern auch auf dem PC oder Mac läuft. Damit fallen native Apps für Android oder iOS schon mal durchs Raster, denn die laufen nur auf Smartphones und sind obendrein auch schwierig zu entwickeln und zu verteilen. Entwicklungsumgebungen wie Thunkable und App Inventor lindern anfangs den größten Schmerz, können aber längst nicht alle Probleme nativer Apps lösen.

Eine sinnvolle Alternative sind Web-Anwendungen, die auf HTML5, CSS3 und JavaScript basieren. Sie laufen überall dort, wo ein moderner Web-Browser läuft, sind einfach zu entwickeln und können leicht an die unterschiedlichsten Clients verteilt werden.

Ihr größtes Manko ist, dass sie keinen uneingeschränkten Zugriff auf die Hardware haben, auf der sie laufen. Aus Sicht der Sicherheit ist diese Einschränkung nachvollziehbar, aber es macht die Plattform für manche hardwarenahen Anwendungen höchst unattraktiv.

Dankenswerterweise gilt diese Einschränkung nicht für Bluetooth, denn es gibt den Web Bluetooth-Standard, den mehrere Browser bereits unterstützen. Mittels Web-Bluetooth ist es möglich, per JavaScript mit Bluetooth-Diensten zu kommunizieren. Besonders gut und einfach funktioniert dies momentan mit Googles Chrome-Browser. Der zeigt unter der URL chrome://bluetooth-internals/ zum Beispiel mehr Informationen über Bluetooth-Adapter und -Geräte an, als manch dediziertes Bluetooth-Tool. Das gilt sowohl für den Desktop als auch für die mobile Version.

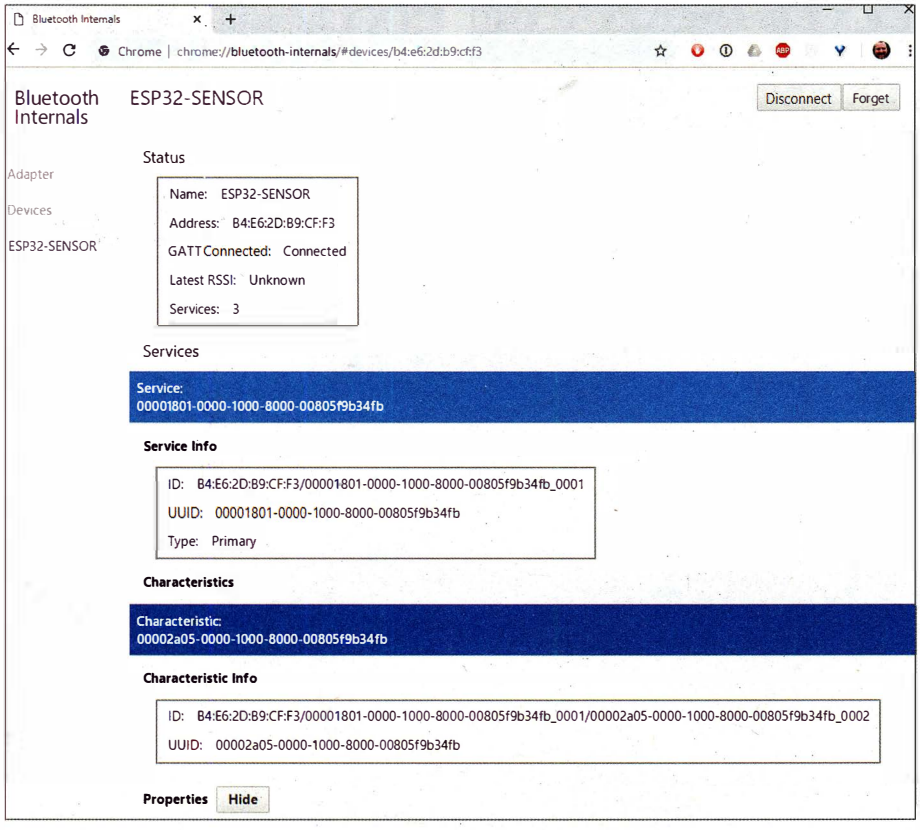
Prinzipiell sollte Web Bluetooth mit allen Geräten funktionieren, die Bluetooth 4.0 oder höher implementieren. Die Web-Blue-

ANONYME FUNKTIONEN UND LAMBDA-AUSDRÜCKE

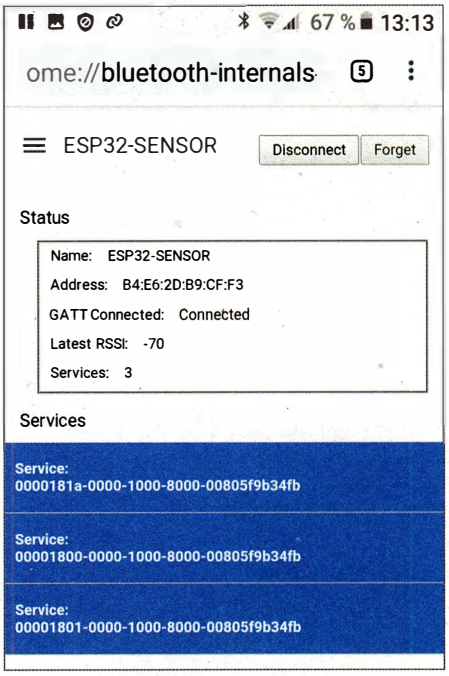
Das Konzept der anonymen Funktionen kommt aus der funktionalen Programmierung. Vereinfacht gesagt definiert man eine anonyme Funktion an der Stelle, wo man sie bei der imperativen Programmierung aufrufen würde. Anstatt also eine Funktion separat zu deklarieren und zu definieren, schreibt man sie einfach dorthin, wo man sie benötigt.

Prinzipiell kann man diese Funktionen über Referenzen oder Zeiger ansprechen beziehungsweise an andere Funktionen übergeben. Der Begriff Lambda-Ausdruck steht synonym für anonyme Funktionen und hat seinen Ursprung im Lamba-Kalkül, einer formalen Sprache zur Untersuchung von Funktionen.

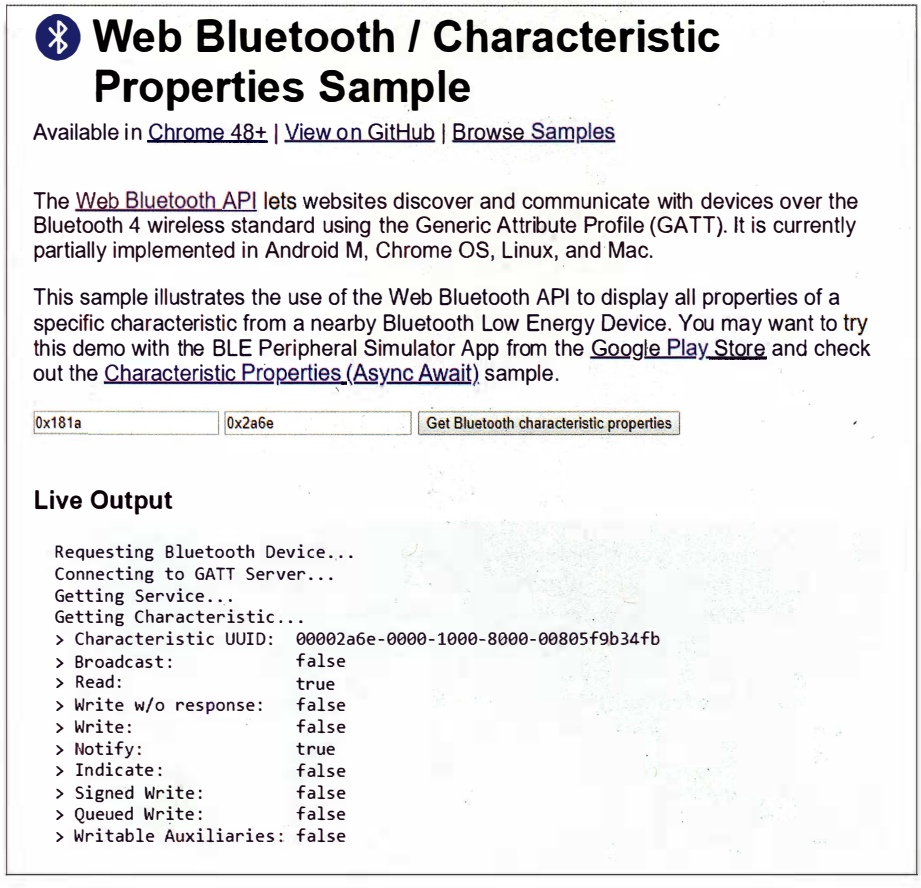
Wer bislang in C oder C++ programmiert hat, benötigt eine Weile, um sich an anonyme Funktionen zu gewöhnen. Sie machen Code aber weniger fehleranfällig und lesbarer.



Chrome gibt bereitwillig Auskunft über Bluetooth-Geräte in der Umgebung.



Auch in der mobilen Version ist Chrome in Bezug auf Bluetooth auf dem neuesten Stand.



Mit den Google-Beispielen kann man auf den ESP32 zugreifen.

tooth-Schnittstelle ist einigermaßen kompakt und Google bietet im Netz jede Menge Beispiele, insbesondere für Bluetooth Low Energy (https://googlechrome.github.io/samples/web-bluetooth/). Mit diesen Beispielen kann man unter anderem schon die Meta-Daten des Environmental-Sensing-Dienstes auf dem ESP32 auslesen (https://googlechrome.github.io/samples/web-bluetooth/read-descriptors.html?service= 0x181a&characteristic= 0x2a6e).

Viele Versprechungen

Spannender ist es natürlich, eine eigene Anwendung zu schreiben und dazu benötigt man nur ein wenig HTML und JavaScript. Eine Beispiel-Anwendung soll kontinuierlich die Temperatur-Werte als Notifikationen vom ESP32 empfangen und darstellen. Als Benutzungsschnittstelle dienen ein Start- und ein Stop-Button, um den Empfang der Werte zu starten beziehungsweise zu stoppen. Um das Beispiel möglichst übersichtlich zu gestalten, zeigt es die Luftfeuchtigkeit nicht an.

Das kurze Listing index.html definiert die Oberfläche der Anwendung mit einfachsten Mitteln. Die aktuelle Temperatur steht im Element mit der ID temperature und die beiden Button-Elemente dienen zum Starten und Stoppen der Kommunikation. Am Ende des Dokuments wird die Datei notifications.js geladen, die die eigentliche Arbeit erledigt.

Der JavaScript-Code besteht im Wesentlichen aus drei Funktionen und einer globalen

Variablen namens `myCharacteristic`, in der die Temperatur-Charakteristik gespeichert wird. Die Funktion `onStartButtonClick` wird aufgerufen, wenn der Start-Button aktiviert wird. Sie definiert zwei Variablen für die UUIDs des Environmental Sensing-Dienstes und der Temperatur-Charakteristik. Dann sucht sie nach Bluetooth-Geräten in der Nähe.

Im Browser kapselt das Objekt `navigator.bluetooth` den Bluetooth-Adapter des Rechners. Die Funktion `requestDevice` sucht nach Geräten, die bestimmte Kriterien erfüllen. In diesem Fall sucht sie nach Geräten, die den Service mit der ID 0x181A anbieten. Die Funktion kann aber auch nach vielen anderen Kriterien, wie zum Beispiel dem Namen des Geräts, filtern.

Das wirklich besondere an der Funktion `requestDevice` ist der Typ ihres Rückgabewertes. Sie liefert nämlich ein Promise-Objekt zurück und das ist ein Muster, das sich in modernem JavaScript-Code bewährt hat. Weil viele Funktionsaufrufe in JavaScript asynchron, also parallel zum Hauptfluss des Programms, erfolgen, hat man früher oft Callback-Funktionen verwendet und diese an andere Funktionen weitergereicht. Das funktionierte bei einfachen Kontrollflüssen noch einigermaßen gut, artete bei halbwegs komplexen Programmen aber schnell aus und wurde sehr unübersichtlich.

Versprechen

Promises lösen dieses – und noch ein paar andere Probleme – auf elegante Art und Weise. Ein Promise-Objekt kapselt nämlich eine zu erledigende Aufgabe und zwei Funktionen. Wenn die Aufgabe erfolgreich erledigt werden konnte, wird die erste Funktion aufgerufen, andernfalls die zweite. Dabei kann die zu erledigende Aufgabe asynchron bearbeitet werden. Das muss aber nicht zwangsläufig der Fall sein.

Die Web Bluetooth-API macht regen Gebrauch von Promises, was man schnell an der langen Kaskade von Aufrufen der Funktion `then` erkennen kann. Diese Funktion ist eine Methode der `Promise`-Klasse und liefert selbst wieder ein `Promise`-Objekt zurück. Daher bekommt auch `then` potenziell zwei Funktionen übergeben, aber in vielen Fällen ist es nur die für den Erfolgsfall, weil sich am Ende der Kaskade die Funktion `catch` um die Behandlung von Fehlern kümmert.

Die Übergabe der Funktionen erfolgt oft in Form eines Lambda-Ausdrucks. Das ist eine besonders kompakte Schreibweise zur Definition einer anonymen Funktion, also einer Funktion ohne Namen. Ein solcher Ausdruck wird mit dem =>-Operator erzeugt.

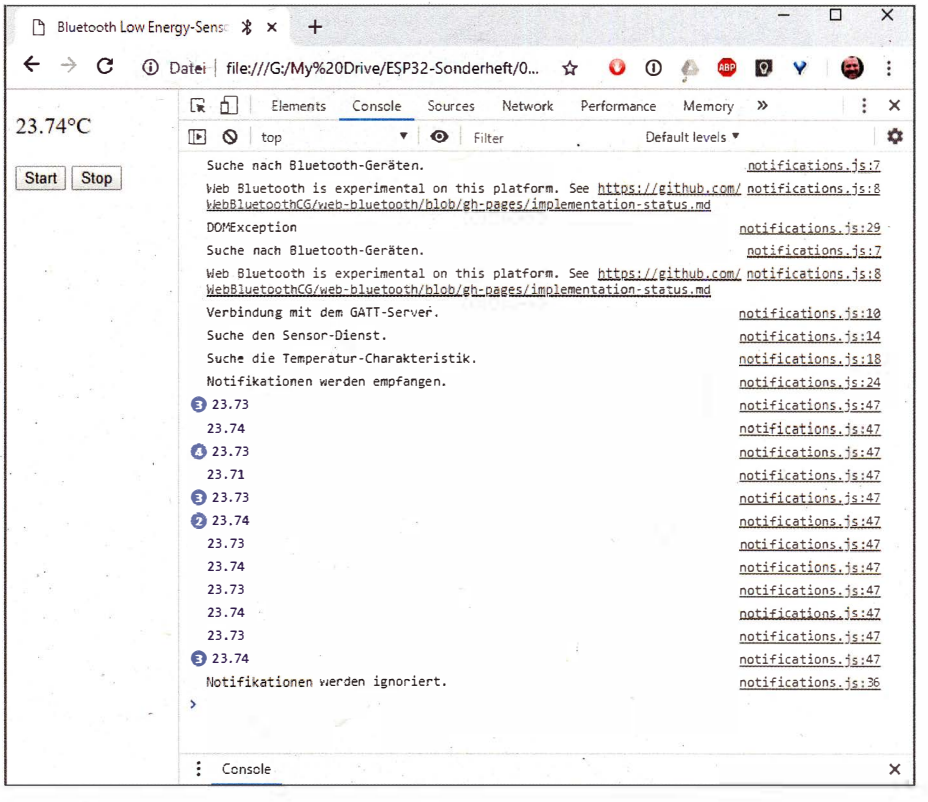
Schließlich ist es noch wichtig zu wissen, dass der Rückgabewert der anonymen Funktion automatisch als Argument an die Funktion des folgenden `then`-Aufrufs übergeben wird. Das bedeutet im konkreten Fall, dass das `Device`-Objekt, das von `requestDevice` gefunden wurde, automatisch an das nachfolgende `then` übergeben wird. Der Lambda-Ausdruck des nachfolgenden `then` liefert ein Objekt der Server-Klasse zurück und das wird dann wiederum an den folgenden Aufruf von `then` übergeben und so weiter. Es dauert einen Moment, bis man sich an diesen Stil gewöhnt hat, aber er findet immer mehr Verbreitung und hat viele Vorteile gegenüber dem alten Spaghetti-Code.

Wenn man das Prinzip der Promises durchdrungen hat, liest sich der Code ganz einfach. Die Funktion `onStartButtonClick` sucht nach einem Bluetooth-Gerät, das den Environmental Sensing-Dienst anbietet. Hat es ein solches gefunden, verbindet es sich mit dem GATT-Server auf dem Gerät und besorgt sich dort eine Referenz auf den Sensor-Dienst. Bei diesem erfragt das Programm dann die Charakteristik des Temperatur-Sensors und weist sie der globalen Variablen `myCharacteristic` zu.

Dann startet das Programm den Notifikationsmechanismus und hier passiert noch etwas Besonderes. Innerhalb eines `then`-Aufrufs erfolgt ein weiterer `then`-Aufruf auf einem neuen Promise-Objekt. Das ist völlig legitim, sollte aber nur maßvoll eingesetzt werden. Bemerkenswert ist noch der Name des Parameters, der an den inneren Lambda-

index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Bluetooth Low Energy-Sensor</title>
  </head>
  <body>
    <p id="temperature">0.0 &#x00b0;C</p>
    <button id="btn_start">Start</button>
    <button id="btn_stop">Stop</button>
    <script src="notifications.js"></script>
  </body>
</html>
```



Auf dem PC kann man BLE-Clients bequem im Browser bauen.

Ausdruck übergeben wird. Der ist nämlich ein Unterstrich („_"), weil der Parameter innerhalb des Lambda-Ausdrucks nicht verwendet wird.

Zum Schluss wird mit der Funktion `addEventListener` noch eine Callback-Funktion bei der Temperatur-Charakteristik registriert. Dieser Aufruf sorgt dafür, dass immer dann, wenn sich der Wert der Charakteristik geändert hat, die Funktion `handleNotifications` aufgerufen wird. Die wird weiter unten noch definiert.

Völlig analog arbeitet die Funktion `onStopButtonClick`. Sie stoppt den Notifikationsmechanismus und entfernt die Callback-Funktion wieder.

`handleNotifications` bekommt als Parameter ein Ereignis mit dem aktuellen Temperatur-Wert übergeben. Der ist genauso kodiert, wie es der Standard vorsieht und muss in eine vorzeichenbehaftete 16 Bit-Zahl umgewandelt werden. Die Variable `value` ist vom Typ `DataView` und solche Objekte kapseln den Zugriff auf ein `ArrayBuffer`-Objekt, das in JavaScript zur Verwaltung binärer Daten dient. Dieses `ArrayBuffer`-Objekt enthält zwei Bytes, die wieder zu einer 16 Bit-Zahl zusammengebaut werden sollen. Dabei hilft die Funktion `getInt16`. Sie erhält den Index der 16 Bit-Zahl im `ArrayBuffer` und das Argument `true` legt fest, in welcher Reihenfolge die Bytes abgelegt sind („little endian").

Am Ende des Programms werden noch die beiden Callback-Funktionen für die Buttons mit den entsprechenden Buttons verknüpft.

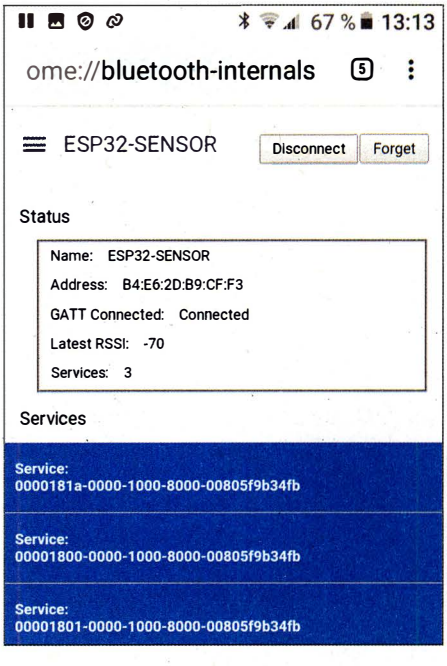
notifications.js

```
var myCharacteristic;

function onStartButtonClick() {
  let serviceUuid = 0x181a;
  let characteristicUuid = 0x2a6e;

  console.log('Suche nach Bluetooth-Geräten.');
  navigator.bluetooth.requestDevice({filters:
  [{ services: [serviceUuid] }]})
  .then(device => {
    console.log('Verbindung mit dem GATT-Server.');
    return device.gatt.connect();
  })
  .then(server => {
    console.log('Suche den Sensor-Dienst.');
    return server.getPrimaryService(serviceUuid);
  })
  .then(service => {
    console.log('Suche die Temperatur-Charakteristik.');
    return service.getCharacteristic(characteristicUuid);
  })
  .then(characteristic => {
    myCharacteristic = characteristic;
    return myCharacteristic.startNotifications().then(_ => {
      console.log('Notifikationen werden empfangen.');
      myCharacteristic.addEventListener
      ('characteristicvaluechanged',
          handleNotifications);
    });
  })
  .catch(error => { console.log(error); });
}

function onStopButtonClick() {
  if (myCharacteristic) {
    myCharacteristic.stopNotifications()
    .then(_ => {
      console.log('Notifikationen werden ignoriert.');
      myCharacteristic.removeEventListener
      ('characteristicvaluechanged',
          handleNotifications);
    })
    .catch(error => { console.log(error); });
  }
}

function handleNotifications(event) {
  let value = event.target.value;
  const t = value.getInt16(0, true) / 100.0;
  console.log(t);
  document.querySelector('#temperature')
  .innerHTML = t + '&#x00b0;C';
}

document.querySelector('#btn_start').addEventListener
('click', onStartButtonClick);
document.querySelector('#btn_stop').addEventListener
('click', onStopButtonClick);
```

Klick mich!

Der Vorteil von Web-Anwendungen ist, dass sie auf so vielen Plattformen funktionieren. Die Datei index.html kann man zum Beispiel mit dem Chrome-Browser auf dem PC laden und auf den Start-Button klicken. Dann öffnet sich ein Dialog-Fenster, in dem der Browser nach Bluetooth-Geräten in der Nähe sucht. Hat er den ESP32 gefunden, kann man ihn koppeln und schon sollten die Temperatur-Daten eintrudeln. Ein Klick auf den Stop-Button beendet den Datenreichtum. Wenn man noch die JavaScript-Konsole über das Menü Weitere Tools > Entwicklertools öffnet, kann man sich parallel auch noch die Log-Ausgaben des Programms ansehen.

Genauso funktioniert die Anwendung im Chrome-Browser auf dem Smartphone. Aller-



Auch in der mobilen Version ist Chrome in Bezug auf Bluetooth auf dem neuesten Stand.

dings gibt es noch ein paar Dinge zu beachten. Während der Entwicklung auf dem Desktop kann man über Dateien, die man lokal oder über einen Web-Server, der auf localhost lauscht, geladen hat, beinahe beliebigen Code ausführen. Das ist aus Sicherheitsgründen nicht erlaubt, wenn man die Dateien über andere Quellen bezieht. Wer eine Web-Anwendung, die auf Bluetooth zugreift, anbietet, muss das daher per HTTPS tun.

Darüber hinaus muss der Bluetooth-Einsatz durch eine Nutzer-Aktion initiiert werden. In der vorliegenden Anwendung ist das der Klick auf den Start-Button. Es ist also nicht erlaubt, sich direkt nach dem Laden einer Seite mit einem Gerät zu verbinden und Daten auszutauschen.

Fazit

Wie so oft, weiß der ESP32 auch beim Thema Bluetooth zu überzeugen. Die Hardware und die Software sind stabil und bieten alles, was das Entwicklerherz begehrt. Selten war es so leicht und preiswert, Bluetooth- oder Bluetooth-Low-Energy-Anwendungen zu entwickeln.

Das bedeutet aber nicht automatisch, dass die Umsetzung eigener Projekte ein Kinderspiel ist. Wer verschiedene Bluetooth-Geräte im Einsatz hat, weiß sicherlich, wie zickig manche davon sein können und wie zeitaufwendig und nervenaufreibend sich die Kopplung und Synchronisation zweier Geräte gestalten kann. Daher ist es wichtig, sich vor und während der Entwicklung mit den richtigen Werkzeugen auszustatten und vertraut zu machen.

Man kann gar nicht genug Bluetooth-Apps auf dem Smartphone haben, wenn man seine eigenen Ideen umsetzen will und man sollte jeden Winkel der Web Bluetooth-API zumindest einmal gesehen haben. Ferner schadet es nicht, einen tiefen Blick in die Spezifikation zu werfen und sich mit den bestehenden Diensten vertraut zu machen. Nur so kann man verhindern, dass man das Rad unnötiger Weise neu erfindet.

Halbwegs erfahrene Entwickler finden im ESP32 und seiner Software aber alles, um mal auf die Schnelle einen neuen GATT-Client oder -Server zu bauen. *—dab*

Auch mobil funktioniert die Temperatur-Anwendung.

Signalerzeugung mit dem ESP32

Der ESP32 ist ein Meister der drahtlosen Kommunikation, aber er beherrscht auch das klassische Mikrocontroller-Repertoire. Unter anderem kann er analoge Signale nicht nur lesen, sondern auch erzeugen.

von Maik Schmidt

Alles zum Artikel im Web unter *make-magazin.de/x43f*

Selbst in einer Welt, die immer mehr von digitalen Technologien abhängt, spielen analoge Signale weiterhin eine große Rolle. Das wird sich vermutlich auch nie ändern, weil die meisten physikalischen Phänomene, wie zum Beispiel der Schall, analog sind.

Für viele Anwendungen müssen Mikrocontroller daher analoge Daten verarbeiten oder erzeugen können. Dazu brauchen sie Bausteine, die analoge Informationen in digitale umwandeln und umgekehrt. Solche Helfer heißen Analog-/Digital-Wandler (ADC; Analog-/Digital Converter) beziehungsweise Digital-/Analog-Wandler (DAC; Digital-/Analog Converter).

Der ESP32 beherrscht beide Disziplinen gut und er hebt sich insbesondere bei der Erzeugung analoger Signale von der Konkurrenz ab. Er hat nämlich gleich zwei DACs mit einer Auflösung von jeweils 8 Bit. Somit kann der ESP32 auf zwei Kanälen Spannungen von 0 bis 3,3V in 256 Abstufungen erzeugen. Verbunden sind die DACs mit den GPIO-Pins 25 und 26.

Konstant gespannt

Listing Constant.ino dürfte die Minimalversion eines Programms sein, das den DAC auf dem ESP32 verwendet. In fünf Zeilen Code setzt es den Ausgangswert des DAC auf Kanal 1 (GPIO-Pin 25) auf den Maximalwert 255. Dazu nutzt es die Funktion `dacWrite` und die Konstante `DAC1`.

Die Spannung, die am GPIO-Pin 25 anliegt, wenn man das Programm auf den ESP32 lädt, beträgt circa 3,2V. Das ist etwas weniger, als die erwarteten 3,3V. Umgekehrt liegt die Ausgangsspannung knapp über 0V, wenn man eine 0 an DAC1 sendet. Die analoge Welt ist eben nicht so deterministisch wie die digitale und hier muss man Fünfe schon mal gerade sein lassen. Sowohl bei der Erzeugung als auch bei der Messung eines Signals kann einiges schiefgehen und etwaige Fehler summieren sich schnell auf.

Das gilt auch für die Abstufung des Signals. Rein theoretisch sollte sich die Spannung in Schritten von 3,2V/256 = 12,5mV verändern lassen. In der Praxis ist das Verhalten der DACs auf dem ESP32 aber nicht ganz so linear. Die Qualität ist dennoch für viele Anwendungen, wie zum Beispiel die Ausgabe von Audiosignalen, völlig ausreichend.

Insbesondere reicht die Qualität aus, um grundlegende Signale synthetischer Musik zu erzeugen. Dazu gehören unter anderem Rechteckwellen, Dreieckwellen und Sinuswellen. In der Zeit der Home-Computer haben Musiker zum Beispiel auf dem Commodore 64 mit nur vier Wellenformen und drei Stimmen epochale Kompositionen erstellt und auch heute noch können sich Musiker für diese sehr spezielle Klangart begeistern.

Zähne zeigen

Um dem ESP32 solche Wellenformen zu entlocken, muss man den DAC mit den Werten der entsprechenden Kurve füttern. Je nachdem, wie schnell man das tut, ändert sich die Frequenz des erzeugten Signals.

Viele Wellenformen lassen sich programmatisch sehr einfach erzeugen. Listing Sawtooth.ino generiert zum Beispiel eine Sägezahnkurve. Die heißt so, weil sie einem Sägeblatt ähnelt und sie zeichnet sich dadurch aus, dass ihr Signal kontinuierlich linear ansteigt und dann schlagartig wieder auf den Anfangswert abfällt.

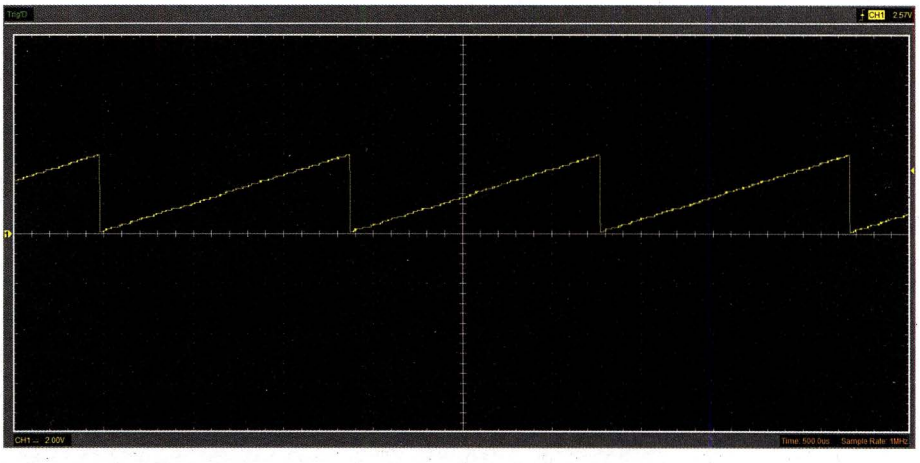
Das Programm erledigt den Anstieg der Kurve mithilfe einer `for`-Schleife und gibt Schritt für Schritt die Werte von 0 bis 255 über den DAC auf Kanal 1 aus. Danach sorgt die `loop`-Funktion dafür, dass das Spiel von vorne beginnt. Mit einem Oszilloskop lässt sich der typische Verlauf der Kurve visualisieren.

Constant.ino

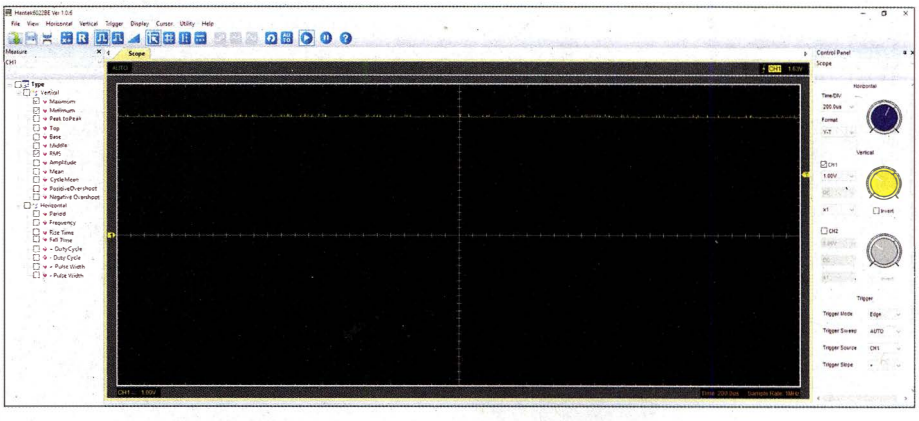
```
1 void setup() {
2   dacWrite(DAC1, 255);
3 }
4
5 void loop() {}
```

Sawtooth.ino

```
1 void setup() {}
2
3 void loop() {
4   for (uint16_t i = 0; i <
    256; i++)
5     dacWrite(DAC1, i);
6 }
7
```



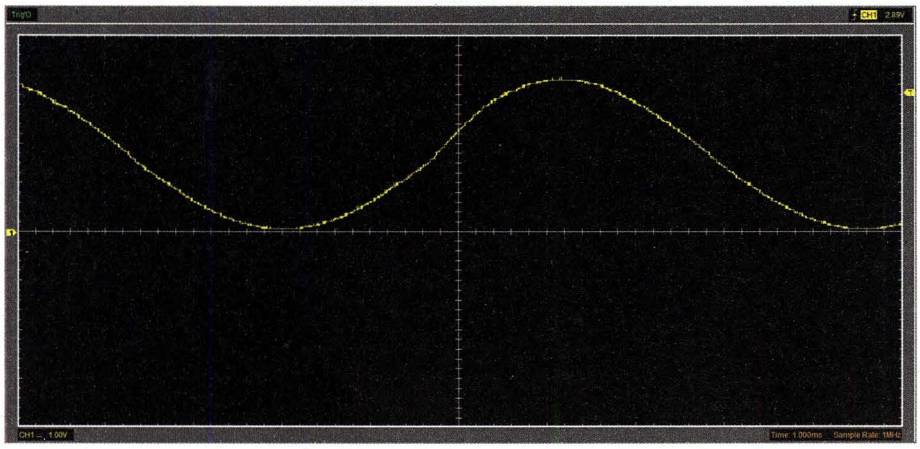
Eine Sägezahnkurve ist schnell erzeugt.



Mit dem DAC-Maximalwert von 255 gibt der ESP32 knapp 3,2V aus.

Sine.ino

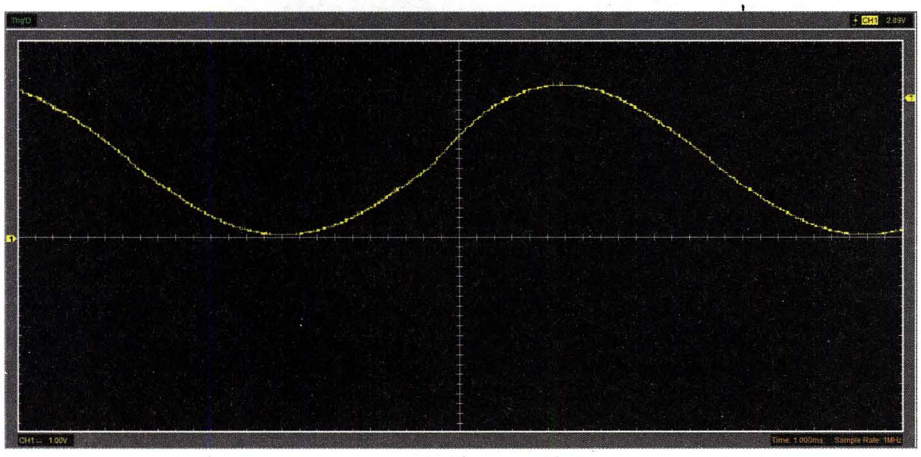
```
1 void setup() {}
2
3 void loop() {
4   for (uint16_t i = 0; i < 360; i++) {
5     dacWrite(DAC1, int(128 + 127 * sin(i * PI / 180)));
6   }
7 }
```



Auch eine Sinuskurve ist kein Problem.

SineWithTable.ino

```
 1 const uint16_t SAMPLES = 360;
 2 uint8_t WaveForm[SAMPLES] = { 0 };
 3 uint16_t position = 0;
 4
 5 void setup() {
 6   for (uint16_t i = 0; i < SAMPLES; i++) {
 7     WaveForm[i] = int(128 + 127 * sin(i * PI / 180));
 8   }
 9 }
10
11 void loop() {
12   dacWrite(DAC1, WaveForm[position++]);
13   position %= SAMPLES;
14 }
```



Eine Sinuskurve kann auch mit vorberechneten Werten erzeugt werden.

Das Verfahren lässt sich auf beinahe beliebige Signalformen übertragen. Listing Sine.ino sendet beispielsweise eine Sinuskurve über den GPIO-Pin 25. Auch dieses Programm ist kurz und nicht viel anspruchsvoller als der Sägezahn-Generator, wenn man mit den wichtigsten Eigenschaften der Sinusfunktion vertraut ist.

Zuerst einmal ist es wichtig zu wissen, dass die `sin`-Funktion in der Arduino-Bibliothek ein Argument im Bogenmaß (Radian) und nicht in Grad erwartet. Somit kann der Schleifenindex nicht direkt übergeben werden, sondern wird durch die Multiplikation mit PI/180 in Radian umgewandelt. Der Rückgabewert der Funktion liegt zwischen -1 und 1 und muss sinnvoll auf den Zielbereich von 0 bis 255 abgebildet werden. Dazu wird er zunächst mit 127 multipliziert, womit sich Werte zwischen -127 und +127 ergeben. Durch die anschließende Addition von 128 ergeben sich insgesamt Werte von 1 bis 255.

Selbstverständlich lässt sich die Form der Sinuskurve beliebig modifizieren. So kann man beispielsweise die Amplitude variieren, wenn man nicht den ganzen Wertebereich des DACs verwenden will.

Konservenwellen

Angespornt durch diese ersten Erfolge kommt man schnell auf die Idee, aus dem ESP32 einen günstigen Funktionsgenerator zu bauen. Das ist ein Gerät, das periodische Signale, wie zum Beispiel Rechteck, Dreieck oder Sinus, erzeugt. Ferner lassen sich oft die Frequenz und die Amplitude des erzeugten Signals einstellen.

Auch wenn die bisherigen Beispielprogramme überschaubar sind und sich leicht auf andere Wellenformen übertragen lassen, haben sie ein paar gravierende Nachteile, wenn sie in einem möglichst generischen Funktionsgenerator zum Einsatz kommen sollen.

Zuerst einmal unterscheiden sie sich in der Geschwindigkeit, mit der das Signal erzeugt wird. Während der Sägezahn seine Ausgangswerte mit annähernd ungebremster Geschwindigkeit an den DAC sendet, muss sich der Sinuswellen-Generator durch die träge `sin`-Funktion und langwierige Fließkomma-Operationen quälen, bis er endlich zu einem Ergebnis kommt. Insbesondere auf einem Mikrocontroller wie dem ESP32 schlagen sich solche Unterschiede deutlich in der Anzahl der benötigten CPU-Zyklen nieder.

Ein weiteres Problem ist die Anzahl der Werte (Samples), die zur Erzeugung der unterschiedlichen Signale verwendet werden. Während es beim Sägezahn nur 256 sind, benötigt die Sinuskurve 360. Um den Ressour-

cenbedarf für alle Wellenformen möglichst gleich zu halten, wäre es von Vorteil, jede Wellenform mit derselben Anzahl an Samples zu definieren.

Für beide Probleme gibt es eine einfache Lösung: Statt alle Funktionswerte immer wieder neu zu berechnen, wenn sie benötigt werden, berechnet man sie einmal zu Anfang des Programms und legt sie in einer separaten Tabelle für jede Funktion ab. Die Zugriffszeit auf die Tabelleneinträge ist konstant und so benötigt die Berechnung eines Funktionswerts für alle Wellenformen dieselbe Zeit.

Listing SineWithTable.ino demonstriert das Prinzip anhand der Sinus-Funktion. Das Programm definiert eine Konstante für die Anzahl der Funktionswerte, die berechnet und ausgegeben werden sollen. In diesem Fall sind das 360 Werte und daher wird ein Feld namens `WaveForm` mit ausreichend Platz angelegt. Die Variable `position` dient zur Speicherung der aktuellen Position innerhalb des Feldes.

Die Funktion `setup` berechnet die 360 Werte der Sinuskurve und speichert sie im Feld `WaveForm`. Anschließend muss die `loop`-Funktion die zuvor berechneten Werte nur noch kontinuierlich ausgeben. Dazu wird die Variable `position` bei jedem Aufruf erhöht. Mittels des Modulo-Operators (%) wird geprüft, ob der aktuelle Wert von `SAMPLES` durch 256 teilbar ist und gegebenenfalls auf 0 zurückgesetzt. So werden die Werte im Feld `WaveForm` immer wieder ausgegeben.

Obwohl die Programme SineWithTable.ino und Sine.ino im Prinzip dieselben Werte ausgeben, unterscheiden sich die erzeugten Signale auf dem Oszilloskop erheblich. Während bei der direkten Berechnung ungefähr eine Schwingung zu sehen ist, sind es bei der tabellengestützten Variante fast fünf. Daraus folgt, dass die Berechnung der Sinus-Werte den Prozess um den Faktor Fünf verlangsamt. Es lohnt sich also, die Funktionswerte vorzuberechnen.

Auf diese Weise können periodische Funktionen mit maximaler Effizienz erzeugt werden. Sie können sich aber weiterhin in der Anzahl der Samples unterscheiden. Damit würden die dazu gehörenden Signale noch immer in unterschiedlicher Zeit erzeugt werden. Dieses Problem wird gelöst, indem die Tabellen mit den Funktionswerten alle dieselbe Größe – zum Beispiel 256 – verwenden.

Für Signalformen wie Rechteck, Dreieck oder Sägezahn ist die Anzahl der vorberechneten Werte einigermaßen unerheblich, aber eine Sinuskurve scheint zunächst an die 360 Grad gebunden zu sein. Wie kann man diese also auf 256 Werte runterrechnen?

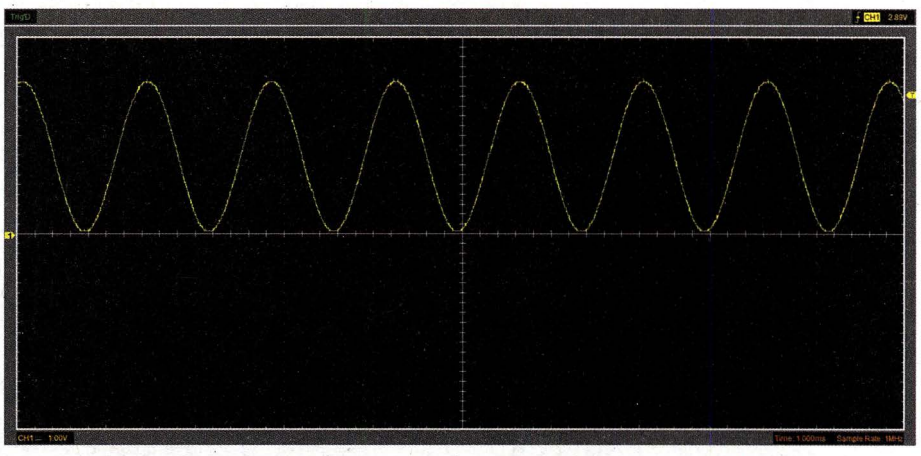
Listing SineWithFixedTable.ino zeigt, wie es geht. Das Programm unterscheidet sich vom Vorgänger nur im Wert der Konstanten `SAMPLES` und in der `setup`-Funktion. Die berechnet statt 360 Werten nämlich nur noch 256. Trotzdem ist die Sinus-Formel annähernd die gleiche. Sie verwendet den Schleifenindex allerdings nicht mehr direkt, sondern die lokale Variable `degree`.

Die wiederum bedient sich der `map`-Funktion der Arduino-Bibliothek, mit der man einen Wertebereich auf einen anderen abbilden kann. In diesem Fall bildet sie den Wert des Schleifenindex proportional vom Intervall 0 bis 255 auf das Intervall von 0 bis 359 ab.

Auf dem Oszilloskop sind jetzt noch mehr Perioden der Sinuskurve zu sehen und das ist logisch, denn zur Erzeugung der Kurve werden 104 Werte weniger (360 – 256 = 104) an den DAC gesendet. Obwohl die Genauigkeit auf dem Papier deutlich geringer ausfällt, reicht sie für die meisten Anwendungen aus. Die Zahl 256 wurde dabei mehr oder weniger willkürlich gewählt und in vielen Fällen reichen weniger Werte.

SineWithFixedTable.ino

```
const uint16_t SAMPLES = 256;
uint8_t WaveForm[SAMPLES] = { 0 };
uint16_t position = 0;

void setup() {
  for (uint16_t i = 0; i < SAMPLES; i++) {
    uint16_t degree = map(i, 0, 255, 0, 359);
    WaveForm[i] = int(128 + 127 * sin(degree * PI / 180));
  }
}

void loop() {
  dacWrite(DAC1, WaveForm[position++]);
  position %= SAMPLES;
}
```



256 Werte reichen völlig, um eine Sinuskurve darzustellen.

Formenvielfalt

Bisher haben alle Programme ihre Daten ungebremst mit der Funktion `dacWrite` an den DAC gesendet. Für viele Anwendungen reicht das aus, aber es wäre sinnvoll, wenn man die Frequenz des Signals ändern könnte. Darüber hinaus wäre es auch hilfreich, wenn man die Signalform wählen könnte. Listing WaveForms.ino implementiert all das und macht aus dem ESP32 einen echten Funktionsgenerator. Das Programm kann vier verschiedene Wellenformen (Rechteck, Sinus, Dreieck und Sägezahn) mit konfigurierbarer Frequenz erzeugen.

Dazu benutzt es die bisher vorgestellten Techniken und definiert zuerst einmal ein paar Konstanten. `MAX_WAVE_TYPES` enthält die Anzahl der Wellenformen, die das Programm erzeugen kann und `SAMPLES` die Anzahl der Funktionswerte, mit denen die einzelnen Signale repräsentiert werden. In `MICROS_PER_SAMPLE` findet man die Anzahl der Mikrosekunden, die man zwischen zwei DAC-Schreibvorgängen warten muss, damit

WaveForms.ino

```
const uint8_t MAX_WAVE_TYPES = 4;
const uint16_t SAMPLES = 256;
const uint16_t MICROS_PER_SAMPLE = 1000000 / SAMPLES;

enum WaveTypes {
  SINE,
  TRIANGLE,
  SAWTOOTH,
  SQUARE
};


uint8_t WaveForms[MAX_WAVE_TYPES][SAMPLES] = {
   { 0 }, // Sine
   { 0 }, // Triangle
   { 0 }, // Sawtooth
   { 0 }, // Square
};


uint16_t position = 0;

void calculate_square() {
  for (uint16_t i = 0; i < SAMPLES / 2; i++) {
    WaveForms[SQUARE][i] = 255;
    WaveForms[SQUARE][SAMPLES - i - 1] = 0;
  }
}


void calculate_triangle() {
  uint8_t value = 0;
  for (uint16_t i = 0; i < SAMPLES / 2; i++) {
    WaveForms[TRIANGLE][i] = value;
    WaveForms[TRIANGLE][SAMPLES - i - 1] = value;
    value += 2;
  }
}


void calculate_sawtooth() {
  for (uint16_t i = 0; i < SAMPLES; i++) {
    WaveForms[SAWTOOTH][i] = i;
  }
}


void calculate_sine() {
  for (uint16_t i = 0; i < SAMPLES; i++) {
    uint16_t degree = map(i, 0, 255, 0, 359);
    WaveForms[SINE][i] = int(128 + 127 * sin(degree * PI / 180));
  }
}


void setup() {
  calculate_square();
  calculate_triangle();
  calculate_sawtooth();
  calculate_sine();
}


const WaveTypes wave_type = SAWTOOTH;
const uint16_t frequency = 10;


void loop() {
  dacWrite(DAC1, WaveForms[wave_type][position]);
  position++;
  position %= SAMPLES;
  delayMicroseconds(MICROS_PER_SAMPLE / frequency);
}
```

das generierte Signal mit einer Frequenz von einem Hertz, also einer Schwingung pro Sekunde, ausgegeben wird.

Zur Verbesserung der Lesbarkeit listet der Aufzählungstyp `WaveTypes` alle Wellenformen auf. Anschließend wird das zweidimensionale Feld `WaveForms` angelegt. Es dient zur Speicherung der vorberechneten Funktionswerte für alle unterstützten Signalformen. Wie gewohnt enthält die Variable `position` die aktuelle Position innerhalb des zu erzeugenden Signals.

Es folgen vier Funktionen zur Vorberechnung der Signalwerte. Neu sind nur die Funktionen zur Berechnung der Rechteck- und Dreieckkurven, aber sie dürften leicht zu verstehen sein. Die `setup`-Funktion ruft lediglich alle vier Funktionen auf.

Vor der `loop`-Funktion werden noch zwei weitere Konstanten definiert. `wave_type` legt fest, welches Signal auf dem DAC landet und `frequency` bestimmt die Frequenz des Signals. Die `loop`-Funktion selbst unterscheidet sich von den bisherigen nur durch den Aufruf der Funktion `delayMicroseconds`. Der sorgt dafür, dass das Signal nicht mehr ungebremst, sondern mit der gewünschten Frequenz ausgegeben wird. Die Konfiguration im Listing erzeugt also ein Sägezahn-Signal mit einer Frequenz von 10Hz.

Durch Änderung zweier Konstanten lassen sich mit diesem Programm die Eigenschaften des erzeugten Signals recht einfach variieren. Allerdings muss dazu das Programm jedes Mal neu übersetzt und auf den ESP32 gespielt werden. Sonderlich komfortabel ist das nicht.

Eine sinnvolle Erweiterung wäre also eine Möglichkeit zur Änderung der Werte von `wave_type` und `frequency` von außen. Das könnte zum Beispiel über die serielle Schnittstelle, per WLAN oder per Bluetooth erfolgen. Denkbar wäre auch der Einsatz weiterer Hardware, wie zum Beispiel eines Potentiometers.

Prinzipiell sind solche Erweiterungen kein Problem, aber man muss dabei beachten, dass zusätzlicher Code in der `loop`-Funktion sich auf das Laufzeitverhalten auswirkt. Zum Beispiel benötigen Zugriffe auf die serielle Schnittstelle vergleichsweise viel Zeit, und die müsste bei der Berechnung der Verzögerung mittels `delayMicroseconds` berücksichtigt werden. Andernfalls hätte das Signal nicht mehr die gewünschte Frequenz.

Das würde aber nur dann funktionieren, wenn die Laufzeit aller Anweisungen, die in der loop-Funktion zusätzlich zu `dacWrite` ausgeführt werden, konstant wäre. Das ist höchst unwahrscheinlich und lässt sich zum Beispiel für Code, der auf die serielle Schnittstelle zugreift, kaum garantieren. Der Königsweg wäre es daher, das Senden der Daten interruptgesteuert oder per DMA zu erledigen.

Das würde aber den Rahmen des Artikels sprengen.

Am schnellsten

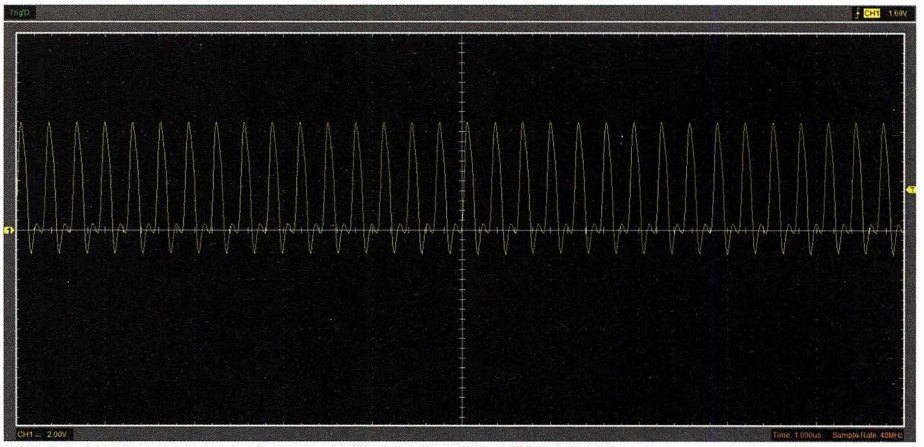
Alternativ zur Erzeugung einer Rechteckspannung per DAC kann man natürlich auch einfach einen GPIO-Pins des ESP32 nutzen und einfach seine Ausgangslevel zwischen High und Low hin und her schalten. Dabei erreicht man Frequenzen von knapp 4MHz. Listing TooglePin.ino zeigt, wie es funktioniert.

Fazit

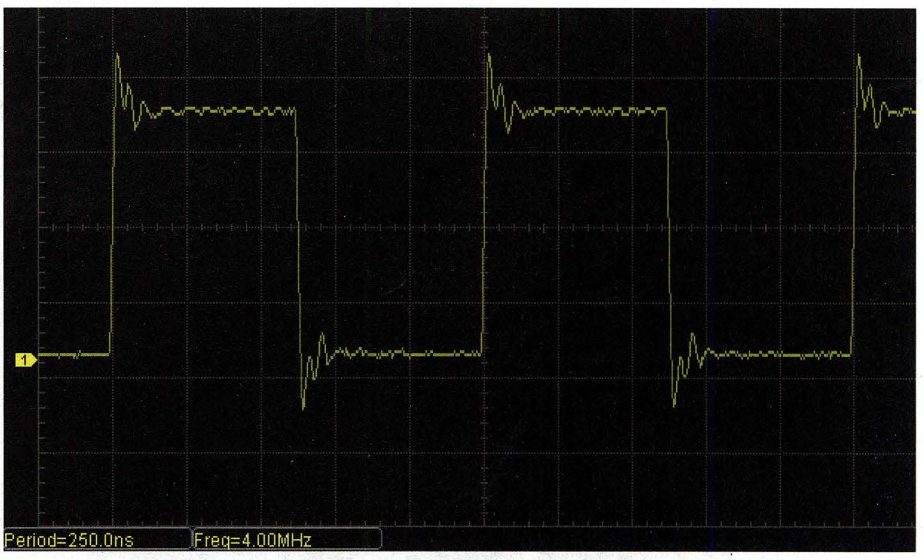
Die Digital-/Analog-Wandler werten den ESP32 als Plattform nochmal deutlich auf, insbesondere weil sie so einfach zu bedienen sind. Zur Ausgabe von Audiosignalen reichen sie allemal und für diesen Zweck gibt es bereits ausgefeilte Bibliotheken (http://www.xtronical.com/the-dacaudio-library-download-and-installation/). Aber auch für viele andere Anwendungen, die auf analogen Signalen basieren, sind sie eine große Hilfe. —*dab*

TogglePin.ino

```
1  const uint8_t TOGGLE_PIN = 25;
2
3  void setup() {
4    pinMode(TOGGLE_PIN, OUTPUT);
5  }
6
7  void loop() {
8    digitalWrite(TOGGLE_PIN, HIGH);
9    digitalWrite(TOGGLE_PIN, LOW);
10 }
```



Das USB-Oszilloskop ist mit 4MHz bei der Darstellung schon überfordert.



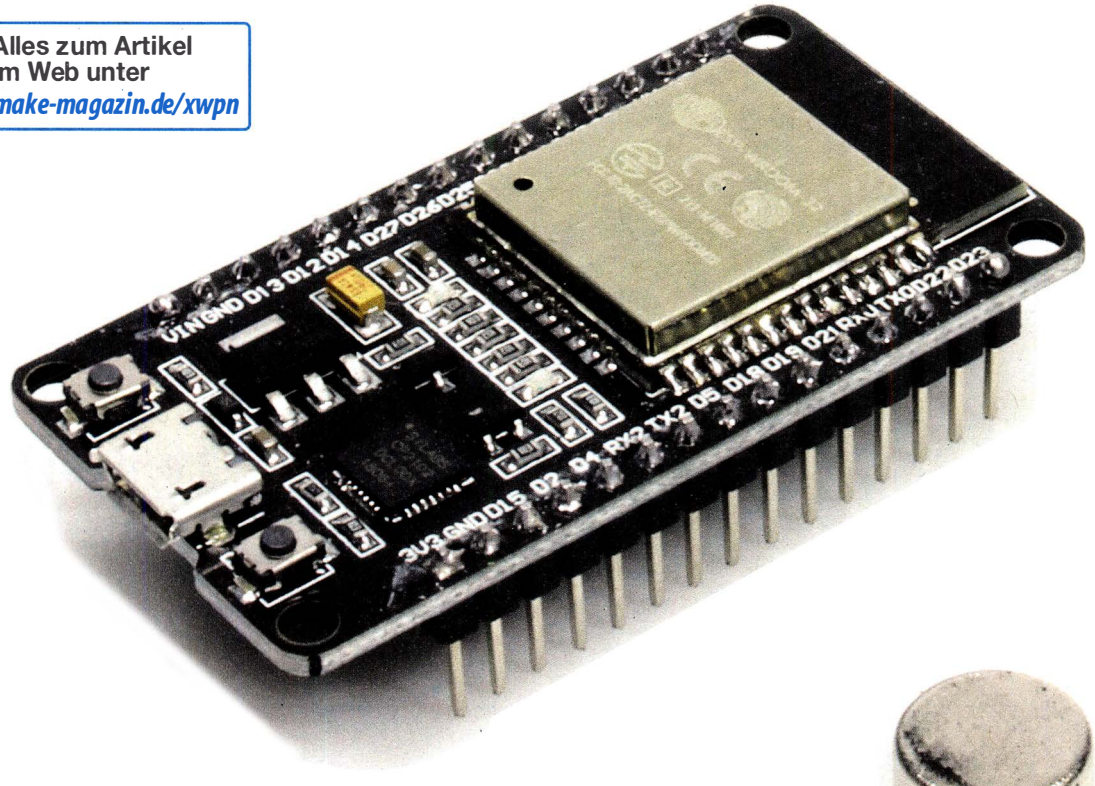
Mit einem leistungsfähigeren Oszilloskop kann man die Signale genauer anzeigen.

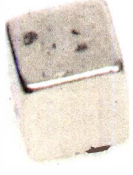
Magnetfelder messen

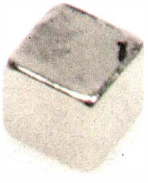
Nicht nur Funkwellen dressiert der ESP32 mit Leichtigkeit. Dank eines eingebauten Hall-Sensors fühlt er sich auch in Magnetfeldern schnell heimisch.

von Maik Schmidt

Alles zum Artikel im Web unter
make-magazin.de/xwpn







Hall-Sensoren nutzen den nach Edwin Hall benannten Hall-Effekt, um Magnetfelder zu messen. Der Hall-Effekt beschreibt das Entstehen einer Spannung an einem stromdurchflossenen Leiter, der sich in einem Magnetfeld befindet.

Das klingt nach dem Physik-Unterricht in der Schule, aber Hall-Sensoren sind eine praktische Angelegenheit, wenn man zum Beispiel kontaktlose und weitestgehend verschleißfreie Schalter braucht. Viele Menschen kommen beinahe täglich mit Hall-Sensoren in Berührung und merken es gar nicht. Die Sensoren stecken unter anderem oft in den Gurtschlössern moderner Autos, um zu erkennen, ob jemand angeschnallt ist. Auch zur Erkennung von offenen Türen werden sie gerne verwendet.

Eingeordnet

Wie die Berührungssensoren lässt sich der Hall-Sensor des ESP32 mit einer einzigen Funktion auslesen. In diesem Fall heißt sie `hallRead`. Listing HallSensor.ino ruft die Funktion alle 100 Millisekunden auf und sendet ihren Rückgabewert an die serielle Schnittstelle.

Nachdem das Programm auf das Development-Board geladen wurde, kann man im seriellen Monitor sehen, welchen Wert der Hall-Sensor gerade misst. Nähert man sich dem Board mit einem Magneten, kann man die Veränderung in den Werten sofort sehen. Die Abstände zwischen zwei benachbarten Werten entsprechen dabei ungefähr einem Millitesla (mT).

Der verwendete Magnet muss allerdings schon einigermaßen stark sein und die besten Effekte erzielen Neodym-Magnete. Neodym zählt zu den Seltenen Erden und wird zur Herstellung der derzeit stärksten Dauermagneten verwendet. Solche Magnete stecken oft in Lautsprechern, Kopfhörern und in manchen Kinderspielzeugen. Sie sind aber auch bei vielen Online-Händlern in unterschiedlichen Formen und Größen erhältlich.

Beim Umgang mit Neodym-Magneten ist eine gewisse Vorsicht ratsam, denn solche Magnete können magnetisierte Datenträger, wie zum Beispiel Magnetstreifen auf EC- oder Kreditkarten durchaus zerstören. Ferner sind die Magnete oft sehr klein und können von Kleinkindern verschluckt werden.

Für erste Experimente mit dem Hall-Sensor ist es hilfreich, die Messwerte im seriellen Plotter der Arduino-IDE zu betrachten. Der verbirgt sich hinter dem Menü Werkzeuge > Serieller Plotter. Bei einem ersten Test wurde der Hall-Sensor für die ersten knapp 100 ausgegebenen Werte in Ruhe gelassen. Dann wurde ein Neodym-Magnet bis ungefähr zum zweihundertsten Wert in die Nähe des Sensors gebracht. Nach einer weiteren Ruhephase kam der Magnet noch einmal in umgekehrter Richtung zum Einsatz.

Die Messdaten ergeben durchaus eine Rechteckkurve, aber es gibt einige Ausreißer. Insbesondere ist die Kurve durchweg recht zittrig, obwohl sich insbesondere dann, wenn kein Magnet in der Nähe ist, nichts tun sollte. Das lässt sich aber leicht beheben.

Magnetisch und stromlinienförmig

Wie bei vielen Sensoren, ist es auch beim Hall-Sensor des ESP32 sinnvoll, die Messwerte nicht direkt zu interpretieren, sondern Durchschnittswerte über kurze Zeiträume zu ermitteln. Listing HallSensorAvg.ino tut genau das.

Statt die Messwerte direkt auf der seriellen Schnittstelle auszugeben, werden jeweils 1000 Werte akkumuliert und am Ende der Durchschnittswert gebildet. Zwischen den einzelnen Messungen wird eine Pause von

HallSensor.ino

```
1 void setup() {
2   Serial.begin(115200);
3 }
4
5 void loop() {
6   Serial.println(
    hallRead());
7   delay(100);
8 }
```

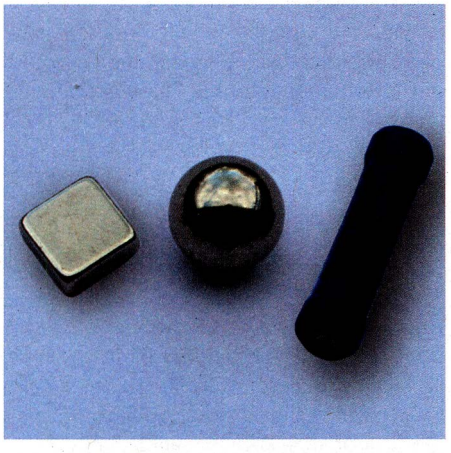
100 Mikrosekunden eingelegt. Bei 1000 Messungen summiert sich die Gesamtwartezeit so auf eine Millisekunde, was für die meisten Anwendungen vernachlässigbar sein dürfte.

Besonders zu beachten ist in der `loop`-Funktion der Datentyp der Variablen `value`. Der muss so gewählt werden, dass es nicht zu einem Überlauf kommt.

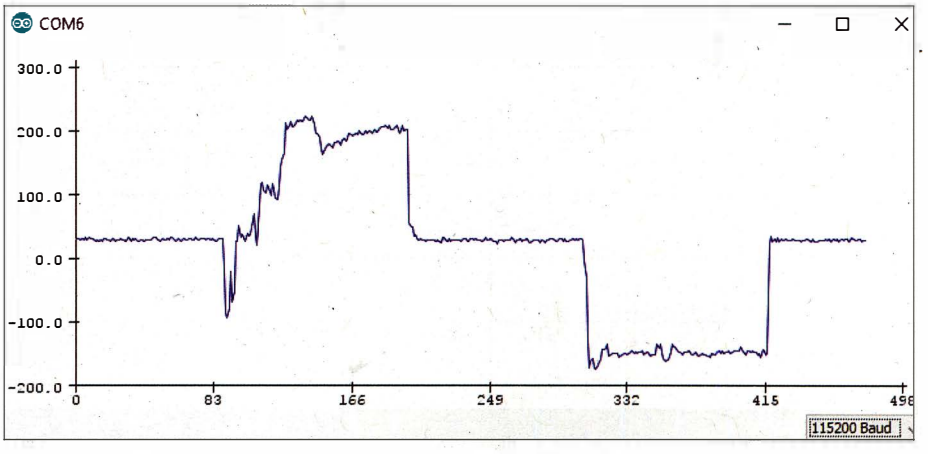
Lädt man das Programm auf den ESP32 und wiederholt das Experiment mit einem Neodym-Magneten, dann ergibt sich ein deutlich ruhigeres Bild im seriellen Plotter der Arduino-IDE. Ungefähr bei Messwert 280 wurde der Magnet noch einmal in derselben Ausrichtung an den Sensor gebracht wie zu Anfang. Diesmal war der Abstand zum Sensor allerdings geringer und so fielen die Messwerte entsprechend höher aus.

Magnetlichtschalter

Mit verlässlichen Sensor-Daten ist es ein Leichtes, den Hall-Sensor als kontaktlosen Schalter einzusetzen. Das kurze Programm in Listing HallSwitch.ino schaltet die Status-LED des Entwicklungsboards ein, sobald ein Magnet nah genug am Board ist. Entfernt man den Magneten, schaltet das Programm die LED wieder aus.



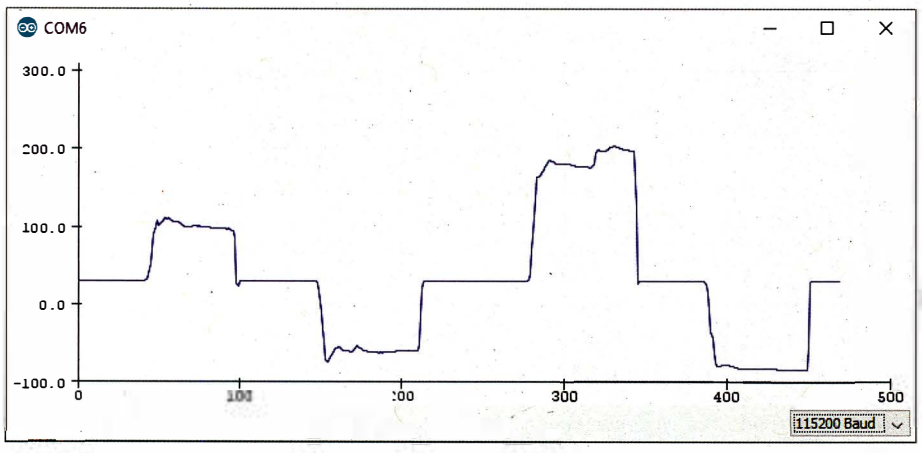
Neodym-Magnete in vielen Formen und Größen



Die Hall-Sensor-Werte erscheinen manchmal etwas zittrig.

HallSensorAvg.ino

```
 1 const uint16_t SAMPLES = 1000;
 2
 3 void setup() {
 4   Serial.begin(115200);
 5 }
 6
 7 void loop() {
 8   int32_t value = 0;
 9   for (uint16_t i = 0; i < SAMPLES; i++) {
10     value += hallRead();
11     delayMicroseconds(100);
12   }
13   Serial.println(value / SAMPLES);
14 }
```



Die Sensor-Werte werden deutlich sauberer, wenn man sie aggregiert.

HallSwitch.ino

```
 1 const uint8_t LED_PIN = 2;
 2 const uint8_t THRESHOLD = 40;
 3 const uint16_t SAMPLES = 1000;
 4
 5 int16_t cleanHallRead() {
 6   int32_t value = 0;
 7   for (uint16_t i = 0; i < SAMPLES; i++) {
 8     value += hallRead();
 9     delayMicroseconds(100);
10   }
11   return value / SAMPLES;
12 }
13
14 void setup() {
15   Serial.begin(115200);
16   pinMode(LED_PIN, OUTPUT);
17 }
18
19 void loop() {
20   if (abs(cleanHallRead()) > THRESHOLD)
21     digitalWrite(LED_PIN, HIGH);
22   else
23     digitalWrite(LED_PIN, LOW);
24 }
```

Der Code ist eine leichte Abwandlung von Listing HallSensorAvg.ino und definiert zunächst Konstanten für den LED-Pin, für den Schwellwert unterhalb dessen der Hall-Sensor als aktiv gilt und für die Anzahl der zu aggregierenden Sensor-Werte. Anschließend folgt eine Funktion namens `cleanHallRead`, die das „saubere" Auslesen des Hall-Sensors übernimmt und leicht in anderen Kontexten wiederverwendet werden kann.

Die `setup`-Funktion initialisiert die serielle Schnittstelle und versetzt den LED-Pin in den Ausgabemodus. Ähnlich ruhig geht es in der Funktion `loop` zu. Die liest den aktuellen Sensor-Wert und berechnet dessen Absolutbetrag mithilfe der `abs`-Funktion. Für den Schalter soll es nämlich unerheblich sein, welche Seite des Magneten sich ihm nähert. Daher geht es nur um den absoluten Wert und nicht um dessen Vorzeichen.

Schließlich vergleicht das Programm noch den gelesenen Wert mit dem Schwellwert und schaltet die LED entweder ein oder aus. Der Schwellwert muss auf jeden Anwendungsfall abgestimmt werden, weil er maßgeblich vom eingesetzten Magneten und der minimalen Distanz zum Hall-Sensor abhängt. Das ist besonders relevant, wenn der ESP32 in einem Gehäuse steckt.

Nähert man sich dem ESP32-Board mit einem Magneten, nachdem man das Programm aufs Board geladen hat, wird die Status-LED ab einem bestimmten Abstand eingeschaltet. Sobald der Magnet diesen Abstand wieder überschreitet, erlischt die LED.

Irgendwas mit Internet

Ein solcher Magnetschalter könnte prinzipiell dazu dienen, offene Türen oder Fenster zu erkennen. Auch könnte man feststellen, ob ein Gegenstand bewegt wurde. Dazu müsste man den Gegenstand lediglich in der Nähe des Sensors platzieren und mit einem Magneten versehen. Weil es Neodym-Magnete auch als Klebestreifen gibt, ist das gar nicht so schwierig.

Jedoch würde der Schalter in seiner jetzigen Form nicht viel nutzen, denn er müsste die Information darüber, dass er aktiviert wurde, ja noch irgendwie weitergeben. Dazu drängen sich die WLAN-Fähigkeiten des ESP32 geradezu auf. Das heißt, wann immer der Schalter aktiviert wird, soll er eine Nachricht senden.

Zu klären ist dann immer noch, wohin der ESP32 eine Nachricht senden soll. Beispielsweise könnte er eine E-Mail oder eine SMS über einen entsprechenden Internet-Dienst schicken. Für diesen Fall wäre das eine halbwegs akzeptable Lösung, aber wenn man gleichzeitig eine Historie aller Schalterbewegungen führen möchte, kommt man damit

nicht weit. Noch deutlicher wird das, wenn der ESP32 andere Sensor-Werte, wie zum Beispiel Temperatur-Daten senden soll. Die stiften per SMS oder E-Mail nur wenig Sinn.

Eine weitere mögliche Lösung des Problems wäre die Programmierung einer Web-Anwendung, die über den Zustand des Schalters informiert. Der ESP32 könnte dann im Falle einer Änderung eine Anfrage an diese Web-Anwendung stellen. Allzu erquicklich ist diese Vorstellung aber nicht, denn eine solche Web-Anwendung zu programmieren und zu betreiben ist anstrengend und zeitraubend.

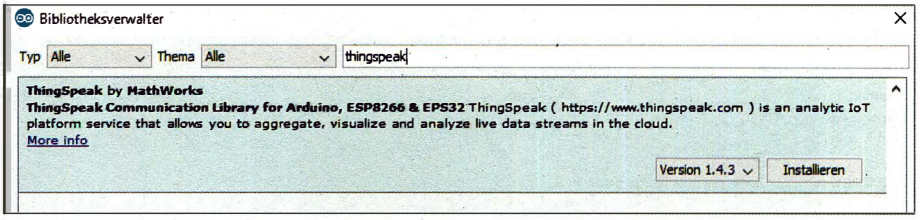
Dankenswerter Weise gibt es mittlerweile viele IoT-Plattformen, die all diese Probleme auf einen Schlag lösen. Ihr einziger Zweck ist es, Daten von IoT-Geräten zu sammeln, sie an zentraler Stelle zur Verfügung zu stellen und ansprechend aufzubereiten. Ferner bieten sie die Möglichkeit, auf bestimmte Muster in den Daten automatisch zu reagieren.

Für den jetzigen Anwendungsfall fiel die Wahl ohne besonderen Grund auf ThingSpeak (http://thingspeak.com/). Die Plattform ist populär, für viele Anwendungsfälle kostenlos und sie wird von vielen Programmierumgebungen unterstützt. Auch für die Arduino-IDE gibt es eine Bibliothek, die sich über den Bibliotheksverwalter installieren lässt. Dort muss man im Suchfeld nur ThingSpeak eingeben und dann den Installieren-Knopf drücken.

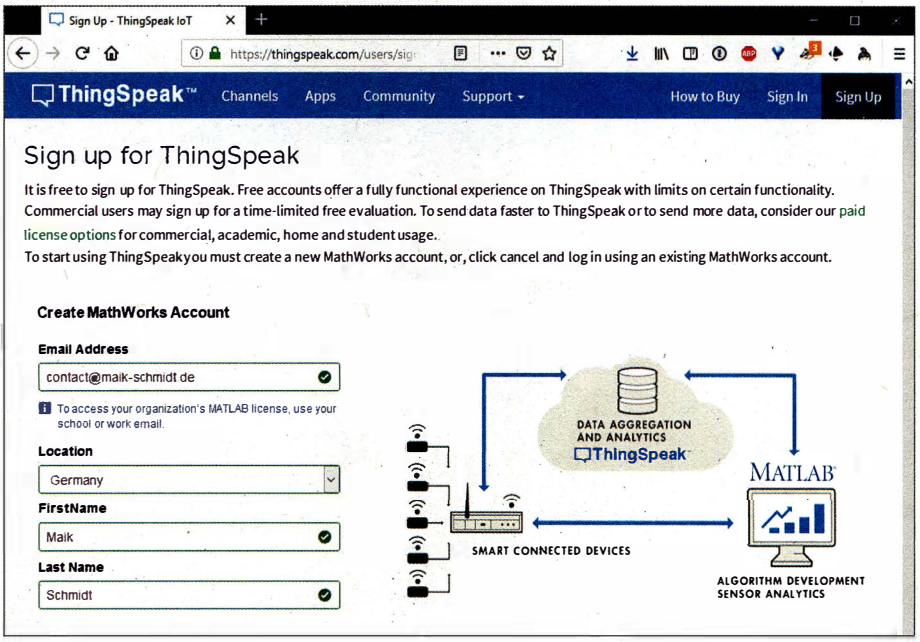
Damit wäre die Unterstützung auf der Seite des Clients schon mal vorbereitet. Um ThingSpeak nutzen zu können, muss man sich dort zunächst registrieren. Wie üblich gibt man eine E-Mail-Adresse und ein Passwort an. Zusätzlich wird noch ein Benutzername verlangt. Hat man sich erfolgreich registriert und seine E-Mail-Adresse verifiziert, kann es direkt losgehen.

Wer Daten über ThingSpeak sammeln möchte, muss dies in Form von Kanälen (Channels) tun. Die werden über die Funktion New Channel angelegt. Jeder Kanal bekommt einen Namen und eine Beschreibung. Ferner kann er gleichzeitig bis zu acht verschiedene Sensor-Daten in Form so genannter Felder (Field) sammeln. Für den Magnetschalter reicht ein einziges Feld.

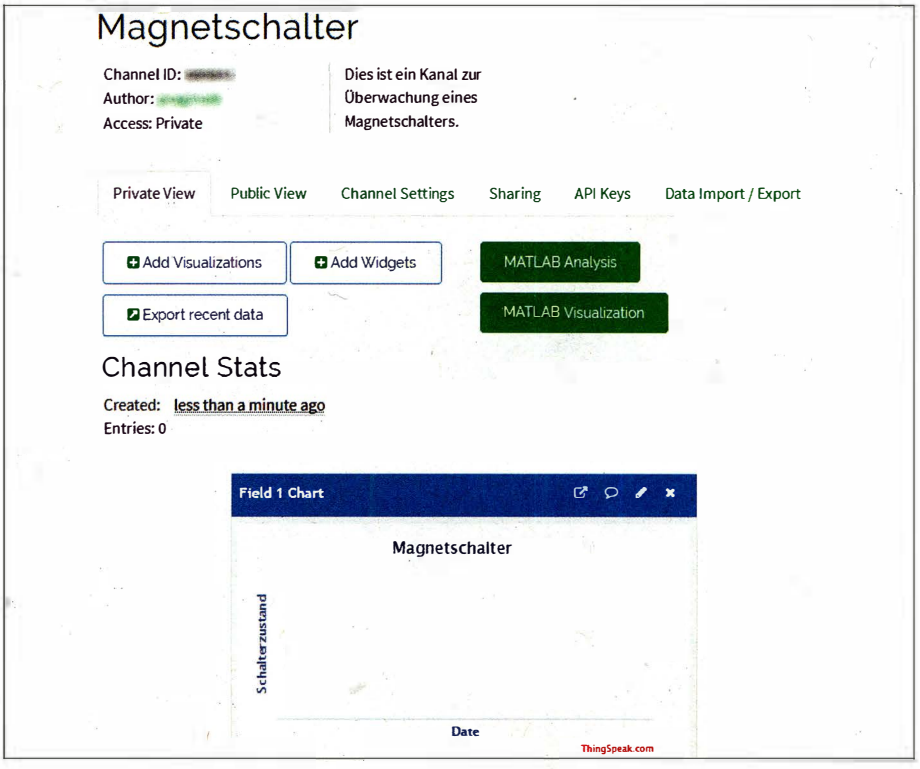
Darüber hinaus kann man jeden Kanal mit einer ganzen Reihe von Meta-Informationen versehen. Dazu gehören unter anderem Links auf eine zum Kanal gehörende Webseite oder ein Github-Projekt. Nützlich ist die Möglichkeit, das IoT-Gerät mit einer Position zu versehen, denn insbesondere für Geräte, die Umwelt-Daten aufzeichnen, ist das eine wichtige Information. Für einen ersten Test-Kanal kann man die meisten Felder leer lassen.



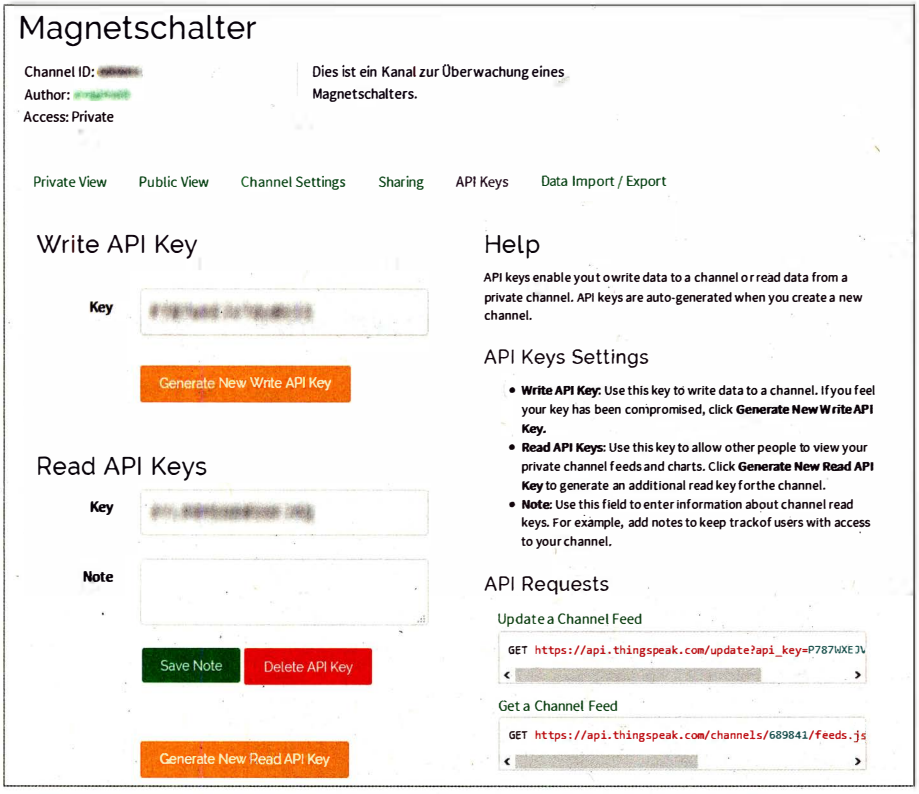
Die ThingSpeak-Bibliothek kann über den Bibliotheksverwalter installiert werden.



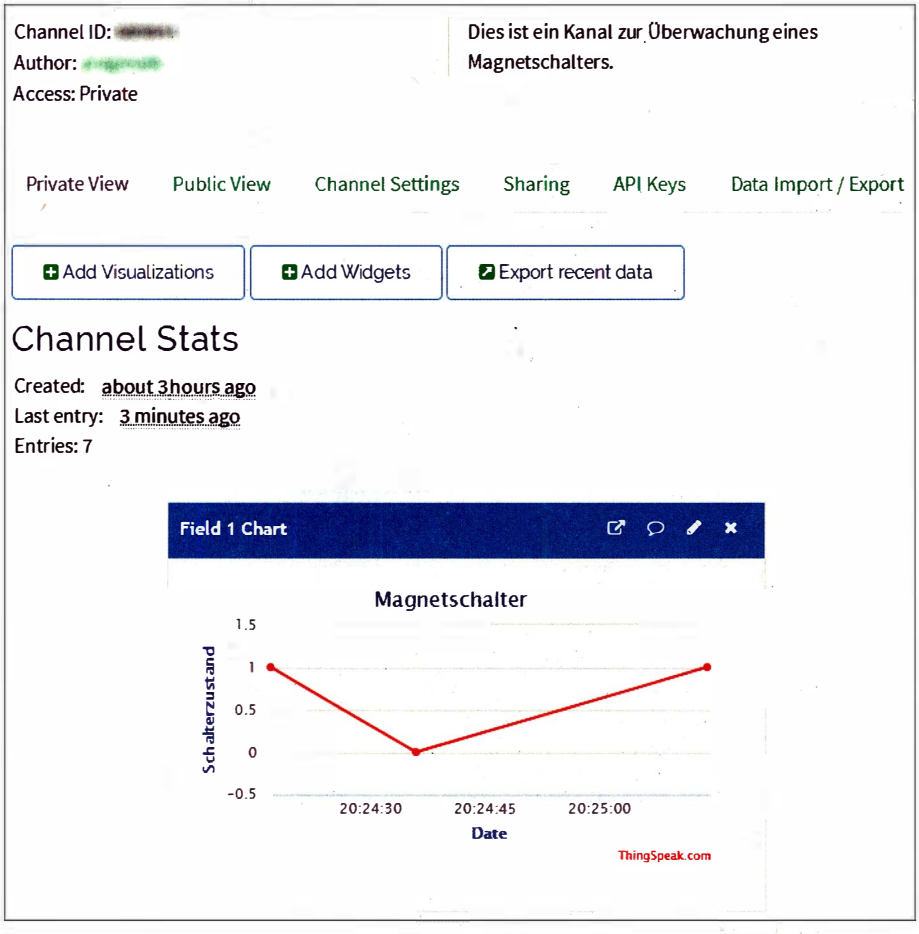
Eine Anmeldung bei ThingSpeak ist schnell erledigt.



Schon ist der Kanal für den Magnetschalter angelegt.



API-Keys sichern den Zugang zum Kanal ab.



ThingSpeak bereitet die Schalterdaten ansprechend auf.

Sobald man den Save Channel-Knopf drückt, wird der Kanal angelegt und bekommt eine Channel-ID, die ihn eindeutig identifiziert. Die benötigt man, um vom ESP32 auf den Kanal zugreifen zu können. Darüber hinaus braucht man noch einen API-Key, der sich auf der ThingSpeak-Seite hinter dem Reiter API Keys verbirgt. API-Keys gibt es für lesende und schreibende Zugriffe. In diesem Fall reicht der für schreibende, weil der ESP32 die Daten nicht liest. Mit der Channel-ID und dem API-Key fürs Schreiben steht der Implementierung des Clients nichts mehr im Wege.

Digitaler Türwächter

In einem finalen Projekt werden nun alle Fäden zusammengeführt. Listing HallSensorIoT.ino implementiert einen digitalen Türwächter, der eine 1 an ThingSpeak sendet, wenn der Magnetschalter aktiviert („Tür zu“) wird. Wird der Magnetschalter deaktiviert, sendet das Programm eine 0 („Tür auf“).

Das Programm besteht im Wesentlichen aus bekannten Teilen und beginnt mit der Einbindung der notwendigen Bibliotheken. Anschließend werden diverse Konstanten definiert, von denen lediglich `CHANNEL_ID` und `API_KEY` neu sind. Sie müssen mit den eigenen Werten für die ThingSpeak-Channel-ID und den API-Key belegt werden. Analoges gilt für die Konstanten `SSID` und `PASSWORD`, die an die Gegebenheiten des lokalen WLAN-Netzes angepasst werden müssen.

Des Weiteren werden auf der globalen Ebene zwei Variablen namens `client` und `isClosed` angelegt. Erstere wird für den Zugriff aufs WLAN benötigt und die zweite repräsentiert den Zustand der imaginären Tür.

An der Funktion `cleanHallRead` hat sich nichts geändert. Neu ist hingegen die Funktion `updateChannel`. Sie sendet den aktuellen Zustand der Tür an ThingSpeak und dokumentiert den Vorgang ausführlich auf der seriellen Schnittstelle. Der eigentliche Schreibvorgang wird durch den Aufruf von `ThingSpeak.writeField` bewerkstelligt. Diese Funktion erwartet die Channel-ID, die Nummer des zu beschreibenden Feldes, den zu schreibenden Wert und den API-Key.

Die Funktion gibt den Wert 200 zurück, wenn alles in Ordnung ist. Wer mit dem HTTP-Standard vertraut ist, wird sich schon denken können, was unter der Haube der Funktion passiert. Sie setzt lediglich HTTP-Anfragen ab und reicht den Status-Code der Anfrage ungefiltert durch. Im Prinzip ist die Schnittstelle von ThingSpeak eine sehr einfache Web-Schnittstelle und es wäre problemlos möglich, sie auch direkt mithilfe der WiFi-Bibliothek zu implementieren. Die ThingSpeak-Bibliothek macht es jedoch ein wenig

bequemer. Unter anderem hat sie schicke Funktionen, um mehr als einen Wert gleichzeitig zu schreiben.

Die `setup`-Funktion bietet auch kaum Neues und initialisiert hauptsächlich die serielle Schnittstelle, das WLAN und den Pin der Status-LED. Die wird nämlich zu Debugging-Zwecken immer noch ein- und ausgeschaltet. Neu ist lediglich der Aufruf von `ThingSpeak.begin`, der die ThingSpeak-Bibliothek mit dem WLAN-Client verbindet.

Schließlich folgt die `loop`-Funktion und die musste geringfügig geändert werden, damit die Funktion den Status des ThingSpeak-Kanals auch wirklich nur dann ändert, wenn sich der Status des Magnetschalters ändert.

In der Vorgängerversion hat die Funktion nämlich die Status-LED so lange immer wieder eingeschaltet, wie der Magnetschalter aktiviert war. Selbst wenn die LED schon eingeschaltet war. Bei LEDs ist das kein Problem, aber bei Online-Diensten schon. Zwar kostet die Nutzung von ThingSpeak für private Zwecke nichts, aber der Dienst hat diverse Beschränkungen. Unter anderem darf man einem Kanal nur alle 15 Sekunden neue Werte hinzufügen.

Vor einem ersten Test sollte man die Web-Seite des ThingSpeak-Kanals aufrufen, auf dem die Anfragen des ESP32 landen. Dann lädt man das Programm auf den ESP32, öffnet den seriellen Monitor und hält einen Magneten bereit. Nähert man sich dem Hall-Sensor mit dem Magneten, sollte dies im seriellen Monitor vermerkt werden. Kurz nachdem dort die Meldung erscheint, dass der Kanal aktualisiert wurde, sollte auch die Webseite entsprechend angepasst werden.

Entfernt man den Magneten nach einer Pause von mehr als 15 Sekunden, wird eine weitere Nachricht an den ThingSpeak-Dienst gesendet und die Webseite erneut aktualisiert. Ist die Pause zu kurz, gibt der Dienst den Fehlercode 401 zurück.

Viel einfacher kann der Umgang mit einer IoT-Plattform nicht sein und ThingSpeak bietet noch jede Menge weiterer Möglichkeiten. Dasselbe gilt allerdings auch für die Konkurrenz, wie zum Beispiel Cayenne (https://mydevices.com/cayenne/features/) oder Adafruit IO (https://io.adafruit.com/). Die Auswahl ist riesig und sicherlich ist für jeden etwas dabei.

HallSensorIoT.ino

```
#include <WiFi.h>
#include "ThingSpeak.h"

const uint8_t LED_PIN = 2;
const uint8_t THRESHOLD = 40;
const uint16_t SAMPLES = 1000;

const char* SSID = "SSID eintragen";
const char* PASSWORD = "Passwort eintragen";
const uint32_t CHANNEL_ID = 0; // Channel-ID eintragen
const char* API_KEY = "API-Key eintragen";

WiFiClient client;
bool isClosed = false;

int16_t cleanHallRead() {
  int32_t value = 0;
  for (uint16_t i = 0; i < SAMPLES; i++) {
    value += hallRead();
    delayMicroseconds(100);
  }
  return value / SAMPLES;
}

void updateChannel(const uint8_t value) {
  Serial.println("Neuer Wert für den Kanal: " + String(value));
  int16_t result = ThingSpeak.writeField(CHANNEL_ID, 1, value,
  API_KEY);
  if (result == 200) {
    Serial.println("Kanal wurde aktualisiert.");
  } else {
    Serial.println("Kanal konnte nicht aktualisiert werden: " +
    String(result));
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
  WiFi.mode(WIFI_STA);
  ThingSpeak.begin(client);
  WiFi.begin(SSID, PASSWORD);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.printf("Verbunden mit %s.\n", SSID);
  Serial.printf("IP-Adresse: %s.\n",
  WiFi.localIP().toString().c_str());
}

void loop() {
  bool switchActivated = abs(cleanHallRead()) > THRESHOLD;
  if (switchActivated && !isClosed) {
    digitalWrite(LED_PIN, HIGH);
    isClosed = true;
    updateChannel(1);
  } else if (!switchActivated && isClosed) {
    digitalWrite(LED_PIN, LOW);
    isClosed = false;
    updateChannel(0);
  }
}
```

Fazit

Hall-Sensoren gehören zu den vielen unsichtbaren Helfern, die das Leben einfacher und sicherer machen. Manch moderne Technologie wird durch sie überhaupt erst möglich. Trotzdem befinden sie sich längst noch nicht in der Werkzeugkiste aller Maker und daher ist es begrüßenswert, dass der ESP32 automatisch einen Hall-Sensor mitbringt.

Besonders charmant ist es, dass sich der Sensor so leicht auslesen lässt. Es besteht also Anlass zur Hoffnung, dass neugierige Bastler ihn für sich entdecken und in immer mehr innovativen Projekten einsetzen.

In die Werkzeugkiste gehören heute aber auch IoT-Plattformen wie ThingSpeak, denn sie machen es einfacher als je zuvor, Geräte mit dem Internet und somit mit dem Rest der Welt zu verbinden. Hier eröffnen sich gänzlich neue Möglichkeiten für spannende Anwendungen. *—dab*

Touch me!

Nicht nur die drahtlosen Kommunikationsmittel des ESP32 sind topmodern. Er punktet auch bei den Eingabemedien und bringt einen ganzen Satz an Berührungssensoren mit.

von Maik Schmidt

Alles zum Artikel im Web unter *make-magazin.de/xucu*

Kapazitive Berührungssensoren gehören heute zum Alltag, denn sie sind es, die die bequemen Touch-Screens in Smartphones und Tablets erst möglich machen.

Wie der Name schon sagt, reagieren solche Sensoren auf Berührung, zum Beispiel mit einem Finger. Dabei muss die Berührung nicht sonderlich stark sein und je nach Sensor reicht er sogar schon aus, wenn sich die Hand nur in der Nähe aufhält.

Das Prinzip hinter den Sensoren ist vergleichsweise einfach. Der Sensor ist im Grunde ein Kondensator, der elektrische Ladung speichern kann. Genau genommen bildet ein Teil des Sensors zusammen mit der zu berührenden Oberfläche einen Kondensator. Dessen tatsächliche Kapazität ist dabei unerheblich, weil der Sensor lediglich versucht, Änderungen in der Kapazität zu erkennen. Diese Änderungen können zum Beispiel durch einen Finger hervorgerufen werden.

Das funktioniert deshalb, weil die Kapazität eines Kondensators nicht nur durch die Größe und den Abstand der Kondensatorplatten bestimmt wird. Sie hängt auch vom Material zwischen den Platten ab, dem sogenannten Dielektrikum. Wenn man dieses ersetzt beziehungsweise verändert, dann ändert sich auch die Kapazität des Kondensators. In diesem Fall wird die Luft zwischen den Elektroden durch den Finger ersetzt und der ist ein ziemlich gutes Dielektrikum, weil er viel Wasser enthält.

Eigentlich ist der Effekt sogar noch etwas subtiler, denn der Finger beeinflusst den Kondensator bereits dadurch, dass er sich durch das elektrische Feld, das den Kondensator umgibt, bewegt.

Eine Berührung kann deshalb sogar durch Materialschichten hindurch erkannt werden. So lassen sich berührungsempfindliche Oberflächen, wie zum Beispiel Touch-Screens, entwickeln, bei denen der eigentliche Sensor nicht mehr zu erkennen ist. Auch für einfachere Bedienelemente wie Taster oder Schieberegler eignen sich Touch-Sensoren hervorragend.

Machen, machen!

Touch-Sensoren sind eine nützliche Sache und erfreulicher Weise hat der ESP32 gleich zehn davon. Leider sind sie nicht alle immer nutzbar. So ist zum Beispiel der GPIO-Pin 0, der für den Touch-Sensor T1 zuständig ist, auf den meisten Entwicklungsboards mit einem Drucktaster verbunden. In den Pinout-Diagrammen dieser Boards taucht er daher zumeist gar nicht auf. Das mitgelieferte Board bildet da keine Ausnahme. Ähnliches gilt für den Sensor T2, den der ESP32 als Strapping-Pin zum Erkennen von Signalpegeln während des Bootvorgangs verwendet.

TouchTest.ino

```
1 void setup() {
2   Serial.begin(115200);
3 }
4
5 void loop() {
6   Serial.println
    (touchRead(T0));
7   delay(100);
8 }
```

Die verbleibenden acht Sensoren (T0 und T3-T9) lassen sich aber einfach in eigenen Projekten einsetzen. Wie bei allen Sensoren ist es für erste Experimente das Beste, einfach loszulegen und Daten abzufragen. Listing TouchTest.ino tut genau das mithilfe der Funktion `touchRead`. Das Programm gibt die Werte des Touch-Sensors T0, der mit GPIO-Pin 4 verbunden ist, kontinuierlich auf der seriellen Schnittstelle aus. Diesen Pin verbindet man zu Testzwecken daher erst einmal mit einem Stück Draht beziehungsweise einem Jumper-Kabel (male-female).

Wie die meisten Sensoren muss man auch Touch-Sensoren kalibrieren. In diesem Fall heißt das, dass man die Sensorwerte zunächst einmal im Ruhezustand beobachtet.

GPIO-Pins mit Touch-Sensoren

Touch- Pin	GPIO-Pin
T0	4
T1	0
T2	2
T3	15
T4	13
T5	12
T6	14
T7	27
T8	33
T9	32

Anschließend sieht man sich an, wie sich die Werte entwickeln, wenn der Sensor auslöst.

Statt langweilige Zahlenkolonnen zu lesen, ist es hilfreich, den seriellen Plotter der Arduino-IDE über den Menüpunkt Werkzeuge > Serieller Plotter zu aktivieren. Der liest Zahlen, die auf der seriellen Schnittstelle ankommen, und stellt sie grafisch dar.

Obwohl der Draht nicht berührt wurde, gab es gleich zu Anfang einen deutlichen kurzen Ausschlag nach unten. Ansonsten liefert der Sensor gleichmäßig einen Wert von knapp unter 80. Als der Draht zwischen

TouchSwitch.ino

```
1  const uint8_t THRESHOLD = 20;
2  const uint8_t LED_PIN = 2;
3
4  bool touchedT0 = false;
5  bool touchedT3 = false;
6
7  void T0wasActivated() {
8    touchedT0 = true;
9  }
10
11 void T3wasActivated() {
12   touchedT3 = true;
13 }
14
15 void setup() {
16   Serial.begin(115200);
17   pinMode(LED_PIN, OUTPUT);
18   touchAttachInterrupt(T0, T0wasActivated, THRESHOLD);
19   touchAttachInterrupt(T3, T3wasActivated, THRESHOLD);
20 }
21
22 void loop(){
23   if (touchedT0){
24     touchedT0 = false;
25     Serial.println("Sensor T0");
26     digitalWrite(LED_PIN, HIGH);
27   }
28
29   if (touchedT3){
30     touchedT3 = false;
31     Serial.println("Sensor T3");
32     digitalWrite(LED_PIN, LOW);
33   }
34 }
```

Mit Instant-Pulver sind die Klavier-Tasten in knapp zwei Stunden fertig.

Aktive Lautsprecher müssen nicht unbedingt aus dem Elektronik-Handel kommen.

Daumen und Zeigefinger genommen wurde, sank der Messwert schlagartig unter 20 und wurde kontinuierlich noch etwas kleiner. Sobald der Draht losgelassen wurde, stieg der Messwert schlagartig wieder auf knapp unter 80.

Um eine Berührung zuverlässig zu erkennen, muss man also lediglich prüfen, ob der Rückgabewert der Funktion `touchRead` kleiner als ein Schwellwert, wie zum Beispiel 30, ist. Gegebenenfalls muss man diesen Wert je nach Sensor und Umgebungsbedingungen anpassen.

Kurze Unterbrechung

Einen Sensor im Code immer wieder direkt abzufragen ist kein sonderlich guter Stil. Schon bei wenigen Sensoren wird das Programm unübersichtlich und schwer wartbar. Besser wäre ein ereignisgesteuerter Mechanismus, bei dem ein Event-Handler aufgerufen wird, wenn ein Touch-Sensor auslöst. Zu diesem Zweck gibt es die Funktion `touchAttachInterrupt` und Listing TouchSwitch.ino zeigt, wie sie verwendet wird.

Das Programm implementiert einen primitiven Lichtschalter. Es schaltet die Status-LED des Entwicklungsboards ein, wenn der Touch-Sensor T0 aktiviert wurde und es schaltet die LED aus, wenn der Touch-Sensor T3 auslöst.

Zu Anfang werden Konstanten für den Sensor-Schwellwert und für den Pin der Status-LED definiert. Es folgen zwei Boolesche Variablen, die anzeigen, ob die Sensoren T0 und T3 jeweils gerade aktiv (`true`) oder inaktiv (`false`) sind. Diese beiden Variablen werden in den Funktionen `T0WasActivated` beziehungsweise `T3WasActivated` auf den Wert `true` gesetzt. Diese beiden Funktionen sollen also als Event-Handler agieren.

Nachdem die `setup`-Funktion die serielle Schnittstelle initialisiert und den LED-Pin in den Ausgabemodus versetzt hat, registriert sie die beiden Event-Handler mit der Funktion `touchAttachInterrupt`. Die erwartet als erstes Argument den Pin des zu überwachenden Touch-Sensors. An zweiter Stelle steht die Funktion, die ausgeführt werden soll, wenn der Sensor aktiviert wurde. Am Schluss steht der Schwellwert, der unterschritten werden muss, um den Sensor auszulösen.

Die `loop`-Funktion kann sich nun darauf beschränken, die Booleschen Variablen abzufragen. Wenn `touchedT0` den Wert `true` hat, schaltet das Programm die Status-LED ein. Hat `touchedT3` den Wert `true`, wird die LED ausgeschaltet.

Wenn man nun das Programm auf den ESP32 lädt und die GPIO-Pins 4 und 15 mit Drähten verbindet, kann man die Status-LED durch Berührungen mit den Fingern ein- und ausschalten. Berührt man beide gleichzeitig, dann blinkt die LED.

Übrigens mag es verlockend erscheinen, die LED direkt innerhalb der Event-Handler ein- und auszuschalten. Das ist jedoch keine gute Idee, denn diese Funktionen werden im Falle eines Interrupts aufgerufen und Interrupt-Handler sollten sich immer auf ein Minimum an Aktivität beschränken.

Schwabbeltasten

Zur Bedienung der Touch-Sensoren dienten bisher Jumper-Kabel oder Drähte. Das ist zu Test- und Entwicklungszwecken ausreichend, macht aber nicht viel her. Eine der interessantesten Eigenschaften der Sensoren ist es ja, dass sie durch viele Materialien hindurch funktionieren und nicht direkt mit einem Finger in Kontakt kommen müssen.

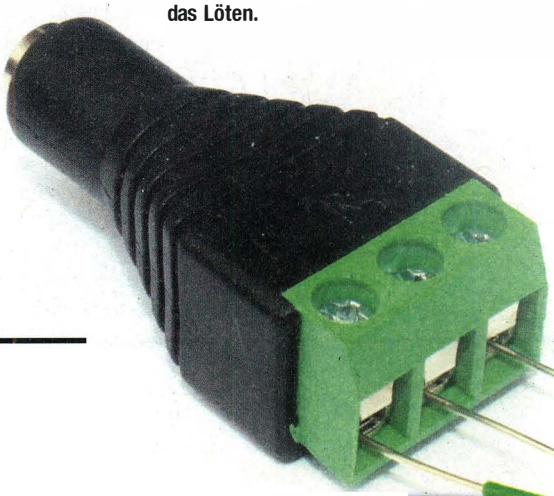
Das Internet ist daher voll von Projekten, die man zum Beispiel mit Tasten aus Alu-Folie oder mit Bananen bedienen kann. Oft kommen dabei Krokodilklemmen zum Einsatz, um die Berührungssensoren mit den GPIO-Pins zu verbinden.

Ein finales Projekt nutzt aber ein ganz anderes Material, das über hervorragende dielektrische Eigenschaften verfügt, lecker schmeckt und auch sonst einiges hermacht. Die Rede ist von Wackelpudding.

Wackelpudding lässt sich leicht herstellen und verarbeiten und dient in diesem Fall als Ausgangsprodukt für ein Wackelpudding-Klavier. Das Klavier gibt unterschiedliche Töne von sich, wenn der Pudding in verschiedenen Behältern mit einem Finger berührt wird.

Wackelpudding gibt es in verschiedenen Formen und für schnelle Experimente eignen sich Instant-Pulver recht gut. Die brauchen knapp zwei Stunden, bis die Tasten fest genug sind, und sie lassen sich einfach dosieren und in die verschiedensten Formen pressen.

Für das vorliegende Projekt dienen Schnapsgläser aus Kunststoff als Tasten. Ein Lötkolben mit einer feinen Spitze ist das per-



Ein Terminaladapter für eine 3,5mm-Klinkenbuchse erspart das Löten.

fekte Werkzeug, um Löcher für die Jumper-Kabel in die Gläschen zu pieksen, nachdem der Inhalt fest geworden ist. Nadeln oder Ähnliches eignen sich nicht so gut, weil das Material schnell splittert.

Das Ganze funktioniert auch prima mit Fertig-Produkten aus dem Kühlregal des örtlichen Supermarkts und selbstverständlich auch in unterschiedlichen Farben und Geschmacksrichtungen. Der Fantasie sind hier kaum Grenzen gesetzt.

Um das Wackelpudding-Klavier zu realisieren, sind zwei Probleme zu lösen. Zum einen muss der ESP32 verschiedene Töne erzeugen. Darüber hinaus muss es eine Möglichkeit geben, unterschiedliche Eingaben zu erkennen. Um das Projekt einfach zu halten, unterstützt das Klavier nur die acht Töne einer Oktave. Außerdem stehen mehr Sensoren ja auch nicht zur Verfügung.

Die Abfrage der Berührungssensoren ist vergleichsweise einfach. Aufwendiger ist die Erzeugung ansprechender Klänge. Ursprünglich sollte das Projekt die Töne mittels eines Piezo-Summers ausgeben und dazu die `tone`-Funktion der Arduino-Umgebung verwenden. Die gibt es aber für den ESP32 noch nicht und so wurde eine alternative Lösung gewählt, die geringfügig komplizierter ist, aber weitaus mehr Potenzial hat.

Statt eines Summers nutzt das Wackelpudding-Klavier nämlich eine 3,5mm-Klinkenbuchse, die sich direkt mit einem Lautsprecher oder einem Ohrhörer verbinden lässt. Solche Klinkenbuchsen gibt es in unterschiedlichen Ausprägungen, aber in der Regel kommen sie ohne Anschlussdrähte daher. Wer sich etwaige Lötarbeiten ersparen möchte, greift am besten zu einem kostengünstigen und leicht wiederverwendbaren Terminaladapter.

Auf einen Verstärker wurde bewusst verzichtet, weil es einfacher ist, einen Ohrhörer oder einen Lautsprecher mit einem eingebauten Verstärker zu verwenden. Die gibt es oft schon für kleines Geld und viele bieten einen ordentlichen Sound.

Hardware, Software, Wetware

Sobald die Hardware steht, muss der ESP32 für jeden Touch-Sensor noch unterschiedliche Töne erzeugen und dazu eignen sich die Digital-Analog-Wandler (DAC) hervorragend.

Die eigentliche Klangerzeugung übernimmt die Bibliothek XT_DAC_Audio (http://www.xtronical.com/the-dacaudio-library-download-and-installation/). Deren Zip-Archiv lässt sich nach dem Herunterladen über den Bibliotheksverwalter der Arduino-IDE installieren.

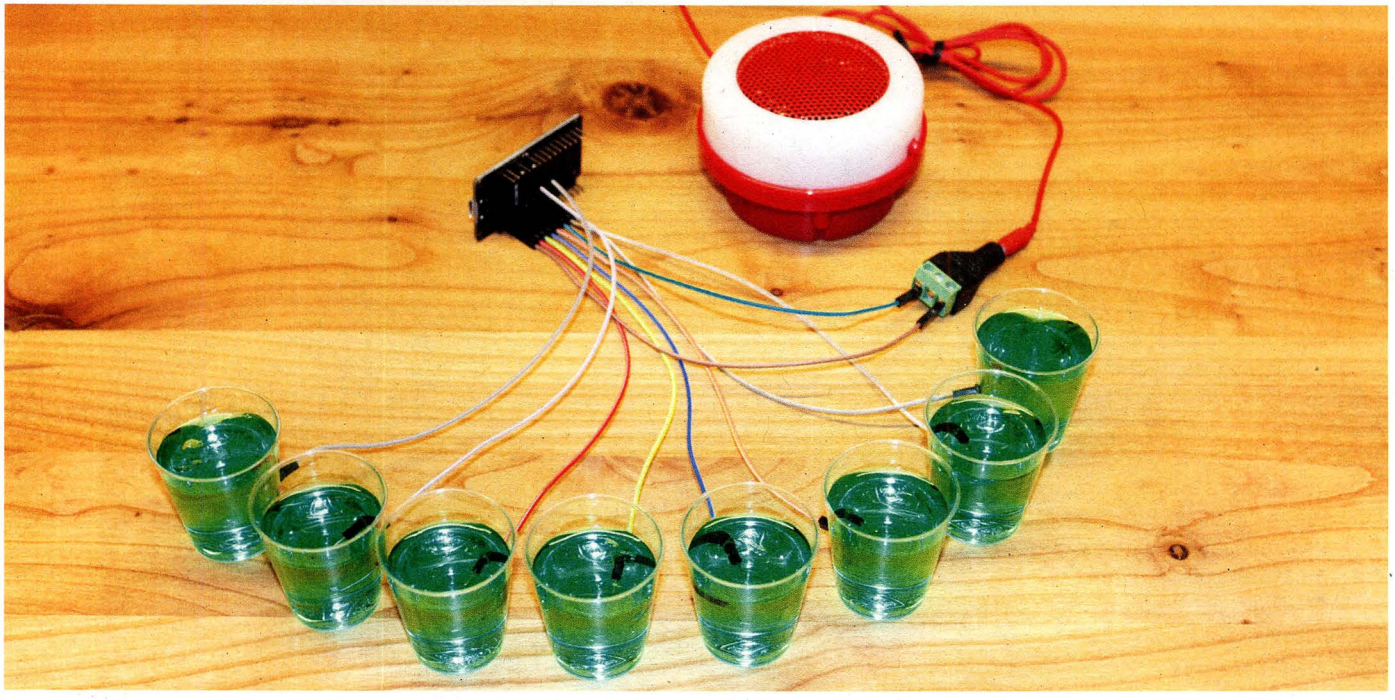
Der Funktionsumfang der Bibliothek ist recht groß und neben diversen Instrumenten kann sie auch WAV-Dateien abspielen. Allerdings ist ihre Schnittstelle stellenweise etwas eigenwillig.

JellyPiano.ino

```
#include "MusicDefinitions.h";
#include "XT_DAC_Audio.h";

const uint8_t THRESHOLD = 15;

XT_DAC_Audio_Class DacAudio(DAC1, 0);

int8_t SCORE_C5[] = { NOTE_C5, SCORE_END };
int8_t SCORE_D5[] = { NOTE_D5, SCORE_END };
int8_t SCORE_E5[] = { NOTE_E5, SCORE_END };
int8_t SCORE_F5[] = { NOTE_F5, SCORE_END };
int8_t SCORE_G5[] = { NOTE_G5, SCORE_END };
int8_t SCORE_A5[] = { NOTE_A5, SCORE_END };
int8_t SCORE_B5[] = { NOTE_B5, SCORE_END };
int8_t SCORE_C6[] = { NOTE_C6, SCORE_END };

XT_MusicScore_Class C5_NOTE(SCORE_C5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class D5_NOTE(SCORE_D5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class E5_NOTE(SCORE_E5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class F5_NOTE(SCORE_F5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class G5_NOTE(SCORE_G5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class A5_NOTE(SCORE_A5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class B5_NOTE(SCORE_B5, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);
XT_MusicScore_Class C6_NOTE(SCORE_C6, TEMPO_ALLEGRO,
INSTRUMENT_PIANO, 1);

void setup() {
  Serial.begin(115200);
}

void loop() {
  DacAudio.FillBuffer();
  play_note(T0, C5_NOTE);
  play_note(T3, D5_NOTE);
  play_note(T4, E5_NOTE);
  play_note(T5, F5_NOTE);
  play_note(T6, G5_NOTE);
  play_note(T7, A5_NOTE);
  play_note(T8, B5_NOTE);
  play_note(T9, C6_NOTE);
}

void play_note(uint16_t pin, XT_MusicScore_Class& note) {
  const uint8_t value = touchRead(pin);
  Serial.println(value);
  if (value < THRESHOLD) {
    if (note.Playing == false) {
      DacAudio.Play(&note);
      delay(100);
    }
  }
}
```

Das Listing JellyPiano.ino bindet zunächst die zur Bibliothek gehörenden Dateien ein und definiert eine Konstante für den Schwellwert der Touch-Sensoren. Dann wird ein Objekt der Klasse `XT_DAC_Audio_Class` angelegt, das sich um die Kommunikation mit dem DAC kümmert. Als Argumente bekommt es die Pin-Nummer des DACs, der verwendet werden soll und die Nummer des zu verwendenden Timers. Die Bibliothek erzeugt die Klänge nämlich interruptgesteuert im Hintergrund.

Es folgen acht Variablen, welche die zu spielenden Töne definieren. Prinzipiell sollte es möglich sein, die Töne direkt abzuspielen, aber die Dokumentation der Bibliothek ist diesbezüglich sehr spärlich. Deshalb werden die Variablen in Form von Musikstücken definiert, die nur eine Note lang sind. Diese



Der Aufbau des Wackelpudding-Klaviers wirkt wüst, ist aber ganz einfach.

Musikstücke müssen als `int8_t`-Felder angelegt werden, deren letztes Element immer die Konstante `SCORE_END` ist.

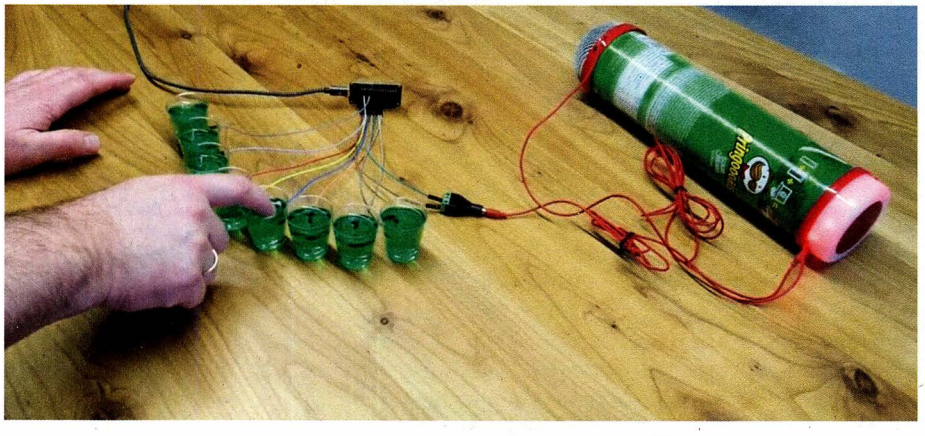
Die eigentlichen Musikstücke werden als Objekte der Klasse `XT_MusicScore_Class` angelegt. Neben den Noten muss man dazu noch die Geschwindigkeit, das zu spielende Instrument und die Anzahl der Wiederholungen angeben. In diesem Fall ist die Anzahl der Wiederholungen gleich 1, weil jede Note bei einem Tastendruck nur einmal gespielt werden soll. Gibt man hier eine 0 an, wird das Musikstück übrigens endlos wiederholt.

Die `setup`-Funktion initialisiert nur die serielle Schnittstelle und die eigentliche Musik spielt wortwörtlich in der `loop`-Funktion. Die ruft gleich zu Anfang die Funktion `FillBuffer` des Objekts `DacAudio` auf. Dadurch werden noch ausstehende Klangdaten an den DAC übertragen. Anschließend wird die Funktion play_note für jeden Sensor und der zum Sensor gehörenden Note aufgerufen.

Die Funktion `play_note` wiederum liest den Sensor aus und wenn er aktiv ist, prüft sie mit dem Attribut `Playing`, ob die zu spielende Note gerade schon gespielt wird. Nur wenn das nicht der Fall ist, wird die Note mit der `Play`-Funktion gespielt und eine Pause von 100 Millisekunden eingelegt. Auf diese Weise werden die Tasten entprellt, sodass die ihnen zugeordneten Töne nicht bei der kleinsten Berührung mehrfach erklingen.

Das kurze Programm ist alles, was benötigt wird, um das Wackelpudding-Klavier zum Leben zu erwecken. Sobald der Code auf dem Board und der Pudding ordnungsgemäß verkabelt ist, kann das Konzert beginnen. Zuvor verbindet man die Klinkenbuchse mit **Pin 25** und einem Masse-Pin des ESP32. Die Drähte an den Touch-Sensor-Pins gehören in den Wackelpudding. Dabei sollte man schon darauf achten, die Schnapsgläser in der musikalisch korrekten Reihenfolge anzuordnen.

Durch einfache Änderungen kann man dem Instrument gänzlich andere Töne entlocken. Weil die verwendete Bibliothek WAV-Dateien abspielen kann, lassen sich die Tasten mit beliebigen Klangeffekten versehen. So wird aus dem Klavier zum Beispiel ganz schnell ein Drum-Computer. Die einzige Beschränkung liegt in der Anzahl der Tasten.



Eine Videovorschau unter den Links gibt einen Vorgeschmack auf das Projekt.

Fazit

Die Touch-Sensoren sind eine feine Sache, und zwar sowohl während der Entwicklung als auch für fertige Projekte. Während der Entwicklung können sie zum Beispiel als schneller Ersatz für Drucktaster herhalten. Im Gegensatz zu denen benötigen die Touch-Sensoren nämlich kein Breadboard und keine Widerstände. Ein einfacher Draht genügt und schon hat man eine simple Benutzungsschnittstelle.

So richtig schick und spannend werden die Berührungssensoren aber erst, wenn sie ihre Stärken ausspielen dürfen. Die liegen ganz klar in innovativen Formen der Benutzerinteraktion. Die helfen nicht nur bei der Umsetzung spaßiger Projekte, sondern ermöglichen auch schicke und subtile Bedienelemente für ernsthafte Anwendungen. —*dab*

IMPRESSUM

Redaktion

Make: Magazin
Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-300
Telefax: 05 11/53 52-417
Internet: www.make-magazin.de

Leserbriefe und Fragen zum Heft: info@make-magazin.de

Die E-Mail-Adressen der Redakteure haben die Form xx@make-magazin.de oder xxx@make-magazin.de. Setzen Sie statt „xx" oder „xxx" bitte das Redakteurs-Kürzel ein. Die Kürzel finden Sie am Ende der Artikel und hier im Impressum.

Chefredakteur: Daniel Bachfeld (dab)
(verantwortlich für den Textteil)

Stellv. Chefredakteur: Peter König (pek)

Redaktion: Heinz Behling (hgb), Helga Hansen (hch), Carsten Meyer (cm), Florian Schäffer (fls), Elke Schick (esk)

Assistenz: Susanne Cölle (suc), Christopher Tränkmann (cht), Martin Triadan (mat)

DTP-Produktion: Nicole Judith Hoehne (Ltg.), Martina Bruns, Martina Fredrich, Jürgen Gonnermann, Birgit Graff, Angela Hilberg, Astrid Seifert, Dieter Wahner

Art Direction: Martina Bruns (Junior Art Director)

Layout-Konzept: Martina Bruns

Layout: Nicole Wesche, www.tieste23.de

Fotografie und Titelbild: Andreas Wodrich, Melissa Ramson

Verlag

Maker Media GmbH
Postfach 61 0407, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-0
Telefax: 05 11/53 52-129
Internet: www.make-magazin.de

Herausgeber: Christian Heise, Ansgar Heise

Geschäftsführer: Ansgar Heise, Dr. Alfons Schräder

Verlagsleiter: Dr. Alfons Schräder

Stellv. Verlagsleiter: Daniel Bachfeld

Anzeigenleitung: Michael Hanke (-167)
(verantwortlich für den Anzeigenteil),
www.heise.de/mediadaten/make

Leiter Vertrieb und Marketing: André Lux (-299)

Service Sonderdrucke: Julia Conrades (-156)

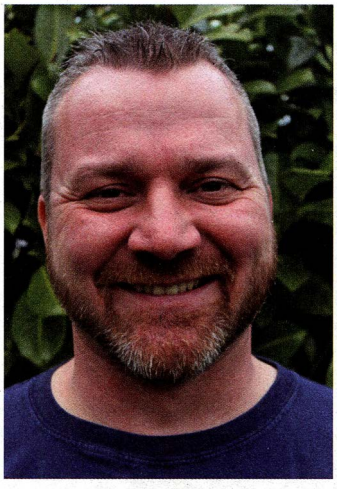
Druck: Goldschmidt GmbH, Josefstraße 35, 49809 Lingen

Vertrieb Einzelverkauf:
VU Verlagsunion KG
Meßberg 1
20086 Hamburg
Tel.: 040/3019 1800, Fax.: 040/3019 145 1800
E-Mail: info@verlagsunion.de
Internet: www.verlagsunion.de

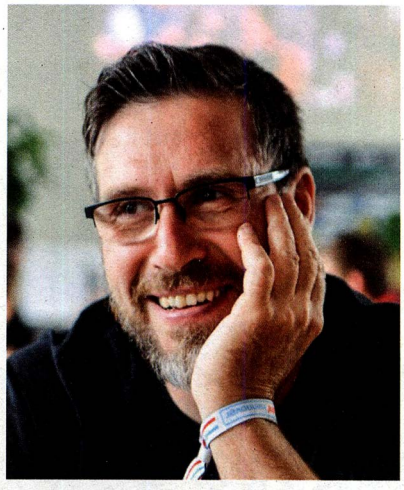
Abo-Service:

Bestellungen, Adressänderungen, Lieferprobleme usw.:
Maker Media GmbH
Leserservice
Postfach 24 69
49014 Osnabrück
E-Mail: leserservice@make-magazin.de
Telefon: 0541/80009-125
Telefax: 0541/80009-122

DIE AUTOREN DIESES HEFTS



Maik Schmidt arbeitet seit über 25 Jahren als Softwareentwickler für mittelständische und Großunternehmen. Außerdem schreibt er Buchkritiken und Artikel für internationale Zeitschriften und hat selbst einige Bücher verfasst. Unter anderem ist er Autor des Buchs „Arduino: A Quick-Start Guide, 2nd ed.", das bei The Pragmatic Bookshelf erschienen ist und unter dem Titel „Arduino – Ein schneller Einstieg in die Microcontroller-Entwicklung" vom dpunkt-Verlag ins Deutsche übersetzt wurde.



Daniel Bachfeld ist Chefredakteur beim Make-Magazin. Mit Mikrocontrollern bastelt er seit 1999. Er ist froh, dass ihm das Arduino-Konzept nun viel Arbeit beim Aufbau der Hardware und der Entwicklung der Software spart.
Er freut sich über Projekteinreichungen und Artikelideen von Lesern, die auf Grundlage dieses Hefts entstanden sind:
dab@make-magazin.de

Wie in alten Zeiten

Der ESP32 muss nicht notwendigerweise in C oder C++ programmiert werden. Es stehen eine ganze Menge Alternativen zur Verfügung und eine steckt sogar ganz tief drin im Chip.

Alles zum Artikel im Web unter *make-magazin.de/xe5f*

von Maik Schmidt

Mittlerweile ist es keine Besonderheit mehr, Mikrocontroller in Hochsprachen wie JavaScript oder Python zu programmieren. Der ESP32 bildet keine Ausnahme und wird von Projekten wie Espruino (http://www.espruino.com/) oder MicroPython (http://micropython.org/) voll unterstützt.

Etwas exotischer ist da schon die Programmiersprache BASIC, die hauptsächlich auf den PIC-Boards sehr beliebt ist. Auf anderen Plattformen ist sie zwar oft verfügbar, führt aber eher ein Schattendasein.

Wenn aber ein BASIC-Interpreter auf einem Mikrocontroller läuft, dann ist es meistens Tiny BASIC (https://en.wikipedia.org/wiki/Tiny_BASIC) oder eine Implementierung, die auf Tiny BASIC basiert. Das liegt daran, dass diese Version extrem sparsam mit dem Speicher eines Rechners umgeht.

Interessanterweise hat die Firma Espressif einen Tiny BASIC-Interpreter ins ROM des ESP32 integriert und ihn anfangs mehr oder weniger versteckt beziehungsweise nicht dokumentiert. Seit er aber zufällig entdeckt worden ist, gibt es auch eine knappe Beschreibung in der offiziellen Dokumentation (https://docs.espressif.com/projects/esp-idf/en/latest/apiguides/romconsole.html).

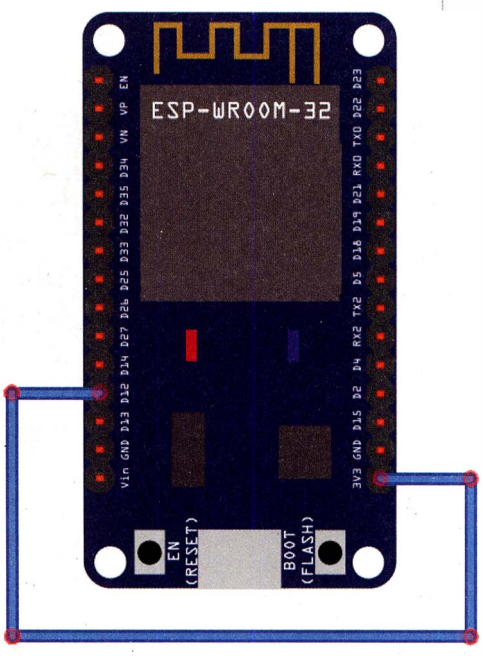
Wo steckt er denn?

Die Dokumentation bezeichnet den BASIC-Interpreter als ROM-Konsole. Die wird immer dann gestartet, wenn der ESP32 beim Booten den Code im Flash-Speicher nicht ausführen kann. In diesem Moment kann man sich mit einem seriellen Monitor mit dem ESP32 verbinden und die ROM-Konsole nutzen.

Um das BASIC zu nutzen, müsste man also nicht-ausführbaren Code ins Flash schreiben. Es gibt allerdings einen kleinen Trick, der den Zugang zum Interpreter vereinfacht. Der ESP32 landet nämlich auch in der ROM-Konsole, wenn man den GPIO-PIN 12 auf HIGH setzt und den ESP32 bootet. Dazu verbindet man den GPIO-Pin 12 und den 3,3V-Pin mit einem Jumper-Kabel (female-female) und drückt den Reset-Knopf.

Anschließend verbindet man sich mit einem seriellen Monitor, wie zum Beispiel PuTTY, mit dem ESP32 und drückt die Return-Taste. Die Fehlermeldungen, die zuvor angezeigt wurden, kann man getrost ignorieren. Sie besagen lediglich, dass der ESP32 nicht auf den Flash-Speicher zugreifen kann.

Wichtig ist, dass man den seriellen Monitor so einstellt, dass er am Ende einer übertragenen Zeile nur ein Linefeed und kein Carriage-Return sendet. Hat alles geklappt, dann



Ein einzelner Draht aktiviert die ROM-Konsole.

grüßt die ROM-Konsole mit einem lapidaren Prompt (>).

Erste Schritte

Wenn man so gar nicht weiter weiß, schadet es nie, einfach mal um Hilfe zu bitten. Gibt man in der Konsole das Kommando `help` ein, listet der Interpreter alle Befehle auf, die er versteht. Darunter ist auch der Befehl `about`

BEVOR EINE SICHERUNG DURCHBRENNT

Neuere Versionen des ESP-IDF haben die unangenehme Eigenheit, eine der Sicherungen (eFuse) im ESP32 durchbrennen zu lassen, wenn neue Software aufs Board gespielt wird. Sobald diese Sicherung durchgebrannt ist, kann man nicht mehr auf die ROM-Konsole und ihren BASIC-Interpreter zugreifen. Damit wäre es dann nicht mehr möglich, die Beispiele aus diesem Artikel nachzuvollziehen.

Es gibt aber eine einfache Möglichkeit, die Sicherung vor dem Durchbrennen zu schützen. Wer also später noch Zugriff auf die ROM-Konsole haben möchte, sollte dies unbedingt tun!

Zur Manipulation der eFuses bietet Espressif das Werkzeug `espefuse.py` an, das automatisch mit dem `esptool` (https://github.com/espressif/esptool) installiert wird. Das ist eine Sammlung von Programmen, die im täglichen Umgang mit dem ESP32 großen Nutzen stiftet. Alle Programme sind in der Programmiersprache Python (https://www.python.org/) geschrieben und funktionieren sowohl mit Python 2.7 als auch mit Python 3.4.

Wer die ESP-IDF-Toolchain installiert hat, hat mit großer Wahrscheinlichkeit auch einen Python-Interpreter. Das gilt insbesondere für die MinGW-Installation unter Windows. Mit dem Python-Paketmanager `pip` kann man `esptool` wie folgt installieren:

```
pip install esptool
```

Auf manchen Umgebungen, unter anderem unter MinGW, führt dieser Befehl zu einem Fehler. Dort kann man alternativ das folgende Kommando verwenden:

```
python -m pip install esptool
```

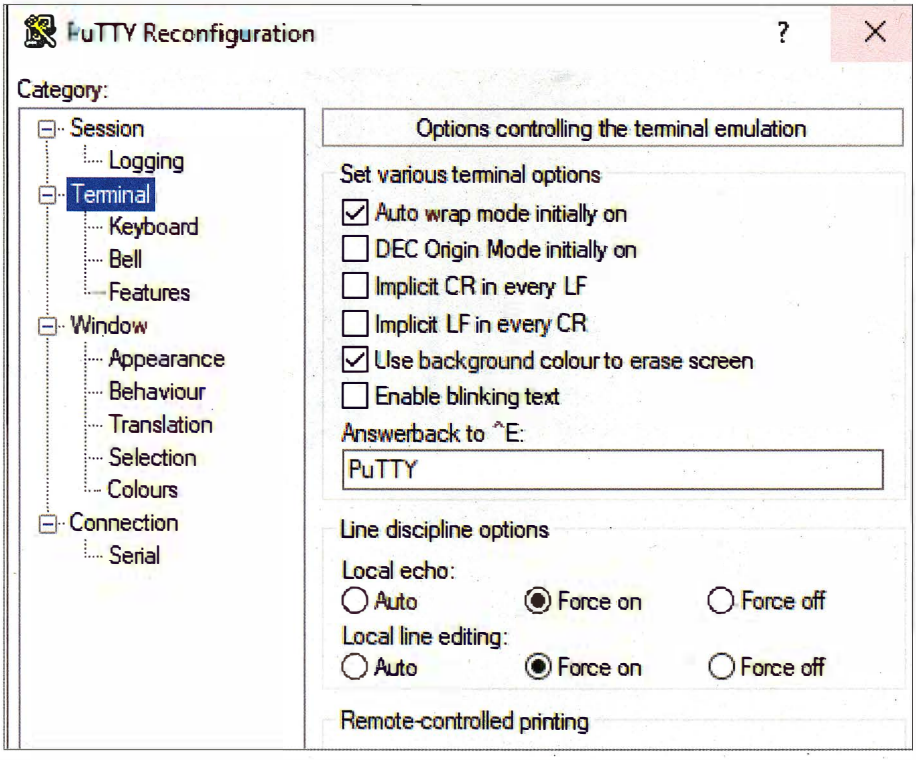
Anschließend muss man mit `espefuse.py` den Schreibschutz für die Sicherung mit dem Namen `CONSOLE_DEBUG_DISABLE` aktivieren:

```
espefuse.py -port <port-name>
write_protect_efuse
CONSOLE_DEBUG_DISABLE
```

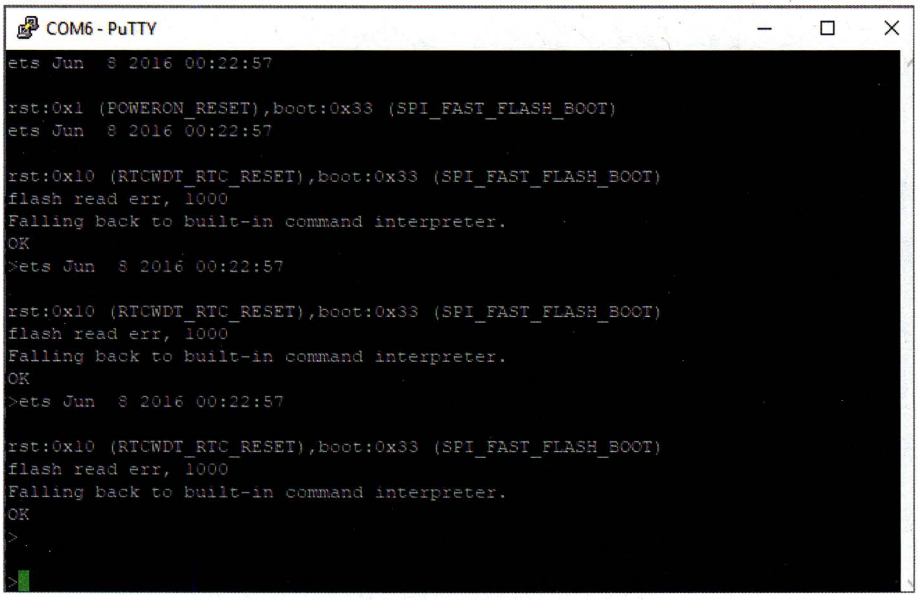
Der Parameter port-name muss durch den Namen der seriellen Schnittstelle ersetzt werden, mit der das ESP32-Board verbunden ist. Hängt das Board beispielsweise unter Windows an Port COM6, dann lautet die Anweisung:

```
espefuse.py -port COM6 write_protect_efuse CONSOLE_DEBUG_DISABLE
```

Weil die Auswirkungen des Kommandos unumkehrbar sind, muss man die Aktion bestätigen, indem man das Wort BURN (alles in Großbuchstaben) eingibt. Anschließend kann die ROM-Konsole nie mehr deaktiviert werden.



So sollte die PuTTY-Konfiguration für die Arbeit mit der ROM-Konsole aussehen.



Die erste Begegnung mit der ROM-Konsole ist wenig spektakulär.

und der gibt ein paar Informationen über den Interpreter aus. Die Konsole ist offenbar eine Erweiterung von TinyBasic Plus (https://github.com/BleuLlama/TinyBasicPlus) und Espressif hat dieser Variante den Namen „ESP32 ROM Basic" gegeben. Dass dem Basic 32.532 Bytes an Hauptspeicher zur Verfügung stehen, weiß der `mem`-Befehl zu berichten.

Bei der Eingabe unterscheidet der Interpreter nicht zwischen Groß- und Kleinschreibung und führt jeden Befehl sofort aus. Unbekannte Befehle quittiert er mit einem „What?" und unvollständige Anweisungen mit der Meldung „How?".

Wie bei den meisten BASIC-Dialekten kann man auch mit dem ESP32-BASIC nicht nur einzelne Befehle absetzen, sondern ganze Programme entwickeln, indem man den Anweisungen Zeilennummern voranstellt. Die Nummerierung kann dabei frei gewählt werden und man sollte zwischen zwei benachbarten Zeilen genügend Abstand lassen, damit man bei Bedarf noch ein paar Zeilen einschieben kann.

Der folgende Klassiker gibt den Text „Hallo, Make-Magazin!" so lange aus, bis man das Programm stoppt:

```
10 PRINT  Hallo, Make-Magazin!
20 GOTO 10
```

In Zeile 10 gibt das Programm den gewünschten Text mit dem `print`-Befehl aus. Die `goto`-Anweisung in Zeile 20 ist eine unbedingte Sprunganweisung, die die Kontrolle wieder an die Zeile 10 übergibt. Damit ergibt sich eine Endlosschleife, die nichts anderes tut, als immer dieselbe Nachricht auszugeben. Benannte Programmpunkte (Labels) kennt Tiny BASIC nicht und die Arbeit mit den Zeilennummern wird schnell unübersichtlich.

Damit man das Programm stoppen kann, muss man es erst einmal starten und das geschieht mit dem Befehl `run`. Wenn man sich an der Ausgabe satt gesehen hat, hält man das Programm an, indem man Strg-C und danach die Return-Taste drückt. Die Ausgabe des Programms läuft währenddessen weiter.

Zwischen die Zeilen 10 und 20 kann man jetzt noch weitere Anweisungen einschieben, indem man eine Zeilennummer zwischen 11 und 19 wählt. Zum Beispiel kann man mithilfe des Schlüsselwortes `rem` wie folgt eine Kommentarzeile einfügen:

```
11 REM Zurück zum Anfang
```

Den Quelltext des Programms fördert das `list`-Kommando zutage und für das aktuelle Programm sieht er so aus:

```
10 PRINT  Hallo, Make-Magazin!
11 REM ZURÜCK ZUM ANFANG
20 GOTO 10
```

Der Interpreter gibt Schlüsselwörter und Kommentare immer in Großbuchstaben aus. Ansonsten sieht der Code genauso aus, wie er eingegeben wurde. Wenn man genug von seinem Programm hat, kann man es mit `new` löschen und ein neues eingeben.

Hacken wie in den Achtzigern

Sobald man mit den grundlegenden Eigenschaften des Interpreters vertraut ist, kann man sich fortgeschrittenen Themen zuwenden. Ältere Semester erinnern sich gewiss noch an die Peek- und Poke-Befehle, mit denen man Speicheradressen auslesen und beschreiben konnte. Die funktionieren auch auf dem ESP32.

Die folgende Anweisung gibt zum Beispiel den Inhalt des UART_DATE-Registers der seriellen Schnittstelle UART0 aus:

```
PHEX PEEK(&h3FF40078)
```

Weil es für die Register-Adressen keine Konstanten gibt, muss man die Adresse des

poke_blink.bas

```
10 ' Blinker
20 POKE &H3FF44020, 4
30 POKE &H3FF44004, 4
40 DELAY 200
50 POKE &H3FF44004, 0
60 DELAY 200
70 GOTO 30
```

io_blink.bas

```
10 IODIR 2,1
20 FOR A=1 TO 10
30   IOSET 2,1
40   DELAY 200
50   IOSET 2,0
60   DELAY 200
70 NEXT A
```

UART_DATE-Registers direkt angeben. In diesem Fall ist das die Hexadezimal-Zahl 3FF40078. Solche Zahlen beginnen in Tiny BASIC mit dem Präfix „&h". Der `peek`-Befehl liest den Speicher-Inhalt an der Adresse aus und gibt ihn als 32-Bit-Zahl zurück. Diese Zahl wird dann wiederum mithilfe der `phex`-Funktion als hexadezimale Zahl ausgegeben.

Solche Adressen muss man sich über die Memory Map im Datenblatt (https://www.espressif.com/sites/default/ files/documentation/esp32_datasheet_en.pdf) und die Beschreibungen der Register im Technical Reference Manual (https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf) zusammensuchen. Leider ist die Dokumentation nicht allzu üppig.

Das Gegenstück zu `peek` heißt `poke` und mit diesem Befehl kann man Speicher-Inhalte verändern. Weil der ESP32 mit memory-mapped IO arbeitet, also Speicheradressen auf Hardware-Register abbildet, wirken sich solche Änderungen oft auf das Verhalten der Bausteine im ESP32 aus. Beispielsweise kann man durch Veränderung der Werte an den richtigen Speicheradressen GPIO-Pins ein- und ausschalten. Listing poke_blink.bas demonstriert das und lässt die Status-LED des Entwicklungsboards blinken.

Das Programm beginnt mit einer Kommentarzeile, die diesmal nicht mit dem Schlüsselwort `rem`, sondern mit einem Apostroph eingeleitet wird. Das ist eine alternative Schreibweise für Kommentare.

Zeile 20 benutzt den `poke`-Befehl, um das Register `GPIO_ENABLE_REG` zu beschreiben. Dieses Register kontrolliert, welcher GPIO-Pin ein Ausgabe-Pin ist. In diesem Fall soll es Pin 2 sein, weil der mit der Status-LED des Boards verbunden ist. Dazu muss der Wert 4 an die Register-Adresse geschrieben werden, weil das Register 32 Bits breit ist und jedes dieser Bits für einen der GPIO-Pins 0-31 steht. GPIO-Pin 2 wird also durch das dritte Bit repräsentiert und wenn man das auf 1 setzt, ergibt das den Wert 4. Dieser Befehl ist also das Äquivalent zur Anweisung `pinMode(2, OUTPUT)` in der Arduino-Umgebung.

Völlig analog schaltet Zeile 30 die LED ein, indem im Register `GPIO_OUT_REG` das Bit für den GPIO-Pin 2 gesetzt wird. Damit wird der Pin in den Zustand HIGH versetzt und die Status-LED leuchtet. Somit entspricht diese Anweisung dem Aufruf von `digitalWrite(2, HIGH)` in der Arduino-IDE.

Der Delay-Befehl wartet 200 Millisekunden lang und der anschließende `poke`-Befehl schaltet die Status-LED wieder aus. Nach einer weiteren Pause sorgt die `goto`-Anweisung dafür, dass das ganze Spiel von vorn beginnt.

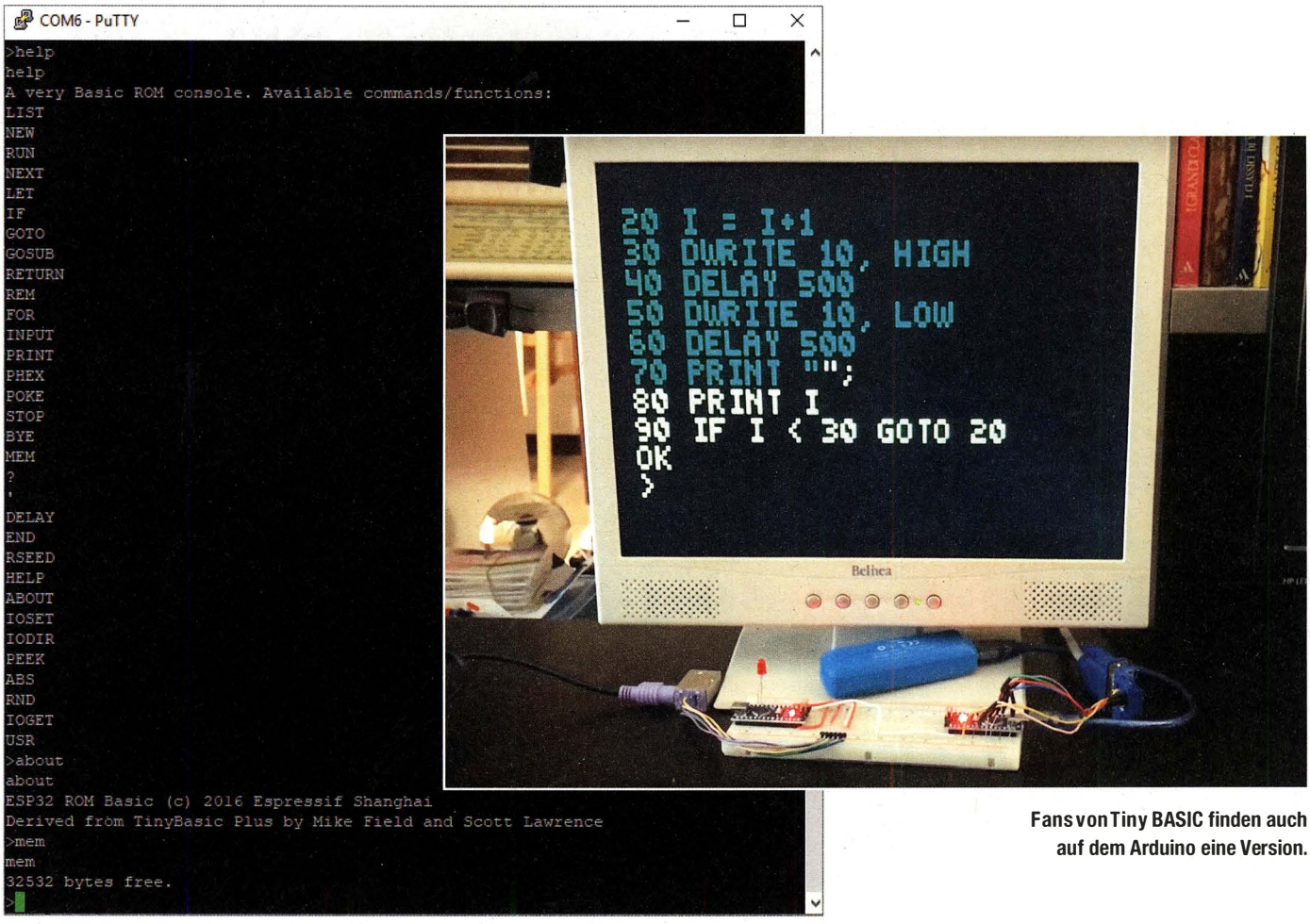
Statt das Programm mühsam in den Interpreter zu tippen, kann man es auch in einen beliebigen Text-Editor laden und mittels der normalen Kopieren- und Einfügen-Operationen in das Fenster des seriellen Monitors kopieren. Der `run`-Befehl startet dann das große Blinken.

Weil das Hantieren mit festen Adressen und einzelnen Bits mühsam und fehleranfällig ist, hat Espressif seiner BASIC-Variante ein paar Befehle spendiert, die zumindest den Zugriff auf die GPIO-Pins vereinfachen. Wie sie funktionieren zeigt Listing io_blink.bas, das die Status-LED zehn Mal blinken lässt.

Die erste Anweisung im Programm nutzt den `iodir`-Befehl, der kontrolliert, ob ein GPIO-Pin ein Eingang oder ein Ausgang ist. Dazu übergibt man die Nummer des Pins und eine 1, wenn er ein Ausgang sein soll. Andernfalls eine 0. In diesem Fall macht der Befehl den GPIO-Pin 2 zum Ausgang.

In Zeile 20 beginnt eine `for`-Schleife, die in BASIC immer eine Zählschleife ist, also von einem Anfangswert bis zu einem Endwert zählt. Hier zählt die Schleife von 1 bis 10 und weist der Variablen A den jeweiligen Schleifenindex zu. Variablen müssen in BASIC übrigens nicht deklariert werden und können jeden beliebigen Wert annehmen.

In der Schleife setzt das Programm den GPIO-Pin 2 auf den Wert HIGH und schaltet damit die Status-LED ein. Dazu dient die `ioset`-Anweisung. Die erwartet die Nummer des GPIO-Pins und entweder den Wert 1 (HIGH) oder 0 (LOW). Nach einer kurzen Pause sorgt `ioset` dann dafür, dass das Licht für 200 Millisekunden wieder ausgeht. Mit



Das BASIC gibt gern ein wenig über sich preis.

Fans von Tiny BASIC finden auch auf dem Arduino eine Version.

der `next`-Anweisung beginnt die nächste Iteration der `for`-Schleife.

Lässt man das Programm im Interpreter laufen, dann verhält es sich exakt so, wie die `poke`-Variante. Allerdings ist der Quelltext um Längen lesbarer.

Das Trio der IO-Funktionen wird komplettiert durch den Befehl `ioget`. Damit lässt sich der Wert, der gerade an einem GPIO-Pin anliegt, auslesen. Die folgenden Anweisungen geben zum Beispiel den aktuellen Wert von GPIO-Pin 0 aus, nachdem der Pin in den Eingabemodus versetzt wurde:

```
IODIR 0,0
PRINT IOGET(0)
```

Ist das denn sicher?

Der ungeschützte lesende und schreibende Zugriff auf den Speicher des ESP32 dürfte den meisten Sicherheitsexperten einen Schauer über den Rücken jagen. Mittels des `peek`-Befehls ließe sich schließlich der Inhalt des gesamten Speichers auf die serielle Schnittstelle schreiben. Für Hersteller, die den ESP32 in ihren Produkten einsetzen und die ihre Geheimnisse vor den Augen der Öffentlichkeit verbergen wollen, kann das ein echtes Problem sein.

Espressif ist sich dessen bewusst und hat daher eine einfache Möglichkeit geschaffen, den BASIC-Interpreter abzuschalten. Dazu muss man lediglich eine bestimmte eFuse des ESP32 durchbrennen lassen. Im Werkzeug `espefuse.py` (https://github.com/espressif/esptool/wiki/espefuse) nennt Espressif diese Sicherung `CONSOLE_DEBUG_DISABLE`.

Aktuelle Versionen des ESP-IDF lassen diese Sicherung grundsätzlich durchbrennen, so dass nach dem Übersetzen und Aufspielen eines Programms der BASIC-Interpreter auf dem ESP32 nicht mehr erreichbar ist. Wer das verhindern möchte, muss die eFuse zuvor mittels `espefuse.py` vor dem Schreibzugriff schützen. Allerdings ist auch dieser Eingriff unumkehrbar, das heißt, danach kann der BASIC-Interpreter nie wieder abgeschaltet werden. Die Arduino-IDE ist nicht so invasiv wie das ESP-IDF und lässt die eFuses in Ruhe.

Fazit

Die ROM-Konsole inklusive BASIC-Interpreter erscheint zunächst wie ein Easter-Egg der Espressif-Entwickler und vermutlich war sie auch so gedacht. Trotzdem ist sie eine nützliche Einrichtung und erlaubt unter anderem die Programmierung des ESP32 ganz ohne IDE.

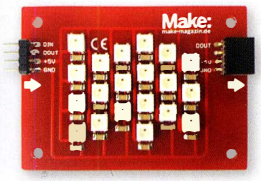
Zwar kann man die so erstellten Programme nicht auf dem Board speichern, aber man kann sie bequem im Copy-/Paste-Verfahren vom PC auf den ESP32 kopieren. Damit eignet sich die Konsole hervorragend, um neue Sensoren oder Aktoren prototypisch auszuprobieren.

Die Versuchung liegt nahe, den BASIC-Interpreter in eigene Programme einzubetten. Das ist aber zum einen nicht ohne Weiteres möglich und zum anderen ist es nicht sonderlich sinnvoll, weil der Interpreter ohnehin nur 4 KB beansprucht. Wer also tatsächlich BASIC-Programme ausführen möchte, sollte lieber einen eigenen Interpreter einbinden, selbst wenn es ebenfalls ein Tiny BASIC ist. *—dab*

Für Maker!

Bauprojekte und Zubehör

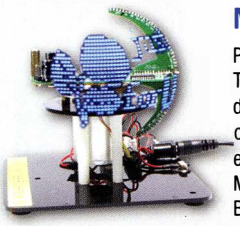
shop.heise.de/gadgets



LED-Nixie Platine

Komplett bestückte Platine für eine Stelle mit bis zu zehn Zeichen einer LED-Nixie-Anzeige, wie sie in der Make 4/2018 vorgestellt wurde. Die Platinen sind über die Stiftleisten anreihbar. Je zwei der RGB-LEDs beleuchten eine Ziffer – die Farbe wird in der Software auf dem Arduino eingestellt.

shop.heise.de/nixie-platine **29,90 € ›**

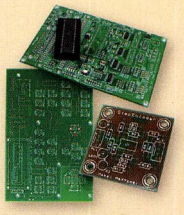


Mini POV-Globus

Persistence of Vision nutzt die Trägheit des Auges und gaukelt dem Betrachter ein dreidimensionales Bild vor, obwohl sich nur ein Streifen mit LEDs bewegt. Mit diesem umfangreichen Bausatz holen Sie sich die beeindruckende Illusion nach Hause.

shop.heise.de/pov-globus **64,90 € ›**

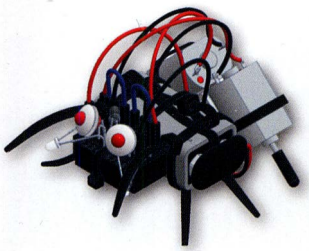
BEST-SELLER



MaXYposi-Bundle

Mit dem Komplett-Bundle zum MaXYposi-Projekt haben Sie drei entscheidende Teile zum Bau Ihres eigenen Multitools: die Schrittmotorsteuerung mit CPU, die Kontrollpult-Platine und die Step-Encoder-Platine. Im Komplettpaket sparen!

shop.heise.de/maxyposi-bundle **99,90 € ›**

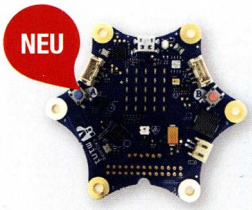


VARIOBOT Bausätze

Die krabbelnden Monster zum Löten oder Zusammenstecken ermöglichen einen spielerischen Einstieg in die Welt der Elektronik und Robotik. Ausgestattet mit Sensoren und Schaltern, im Handumdrehen zusammengebaut, erhältlich in mehreren Farben!

shop.heise.de/variobot **29,90 € ›**

NEU

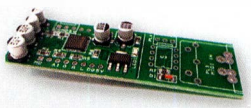


Calliope mini

Alle können coden - und das mit ganz viel Spaß! Mit wenigen Klicks können auf einem angeschlossenen Rechner eigene Programme für den Microprozessor entworfen werden, die den Calliope mini zum Leben erwecken.

Auch erhältlich: Case oder Starter-Set!

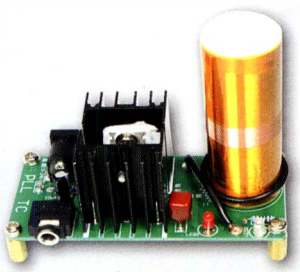
shop.heise.de/calliope **39,90 € ›**



Make NanoSynth

Der SAM2695 von DREAM ist ein mehrstimmiger MIDI-Wavetable-Synthesizer mit Effekteingang auf 5 x 5 Quadratmillimetern und bietet volle Polyphonie mit 128 GM-Standardinstrumenten, verschiedenen Drumkits und Effekten. Vorgestellt in Make 1/18!

shop.heise.de/make-synth **29,90 € ›**



Tesla-Spule

Mit diesem Bausatz entdecken Sie die Welt von Tesla und erzeugen selbst eine ungefährliche Hochspannung die am Drahtende der großen Spule einen Lichtbogen erzeugt. Mit allen benötigten Teilen und Anleitung online.

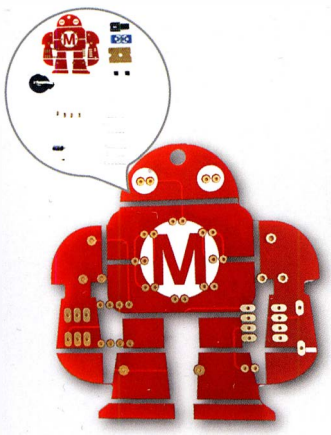
shop.heise.de/tesla-bausatz **29,90 € ›**

BEST-SELLER

ODROID-GO

Mit diesem Bausatz emulieren Sie auf Basis eines ESP32-Moduls nicht nur Spiele-Klassiker, sondern programmieren auch in der Arduino-Entwicklungsumgebung.

shop.heise.de/odroid **49,90 € ›**

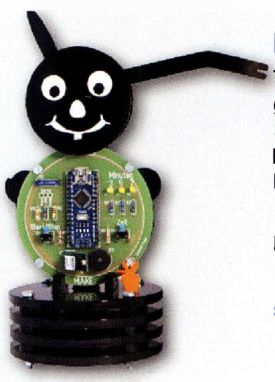


Makey Lötbausatz

Das blinkende Maker-Faire-Maskottchen Makey ist ein Hingucker und auch der ideale Einstieg für die ersten eigenen Löterfahrungen. Die konturgefräste Platine kommt zusammen mit Zubehör und Leuchtdioden, die den Eindruck eines pulsierenden Herzens erwecken.

Jetzt neu mit Schalter!

shop.heise.de/makey-bausatz **ab 4,90 € ›**



NEU

Make: Teatimer „Teeodohr“

Teeodohr überwacht die Ziehzeit Ihres Heissgetränks zwischen 3 und 12 Minuten auf die Sekunde genau, sobald Sie ihm die Karotte ins Pfötchen drücken. Kompletter Bausatz ohne Batterie und Lötzinn.

Nachproduktion läuft. Jetzt vorbestellen!

shop.heise.de/make-teehase **39,90 € ›**

PORTOFREI AB 15 € BESTELLWERT

Ab einem Einkaufswert von 15€ und für Heise Medien- und Maker Media-Abonnenten sind alle Produkte versandkostenfrei. Preisänderungen vorbehalten.

Bestellen Sie ganz einfach online unter **shop.heise.de** oder per E-Mail: **service@shop.heise.de**

heise shop

shop.heise.de ›