Kapitel 1

Der Mikrocontroller ESP32 und seine Funktionalitäten

Dieses Kapitel soll Sie mit der Hardware ESP32 vertraut machen. Dargestellt werden die implementierten Funktionalitäten des Chips, die offiziellen auf dem Chip basierenden Module und verschiedene Boards, die diese Module enthalten. Außerdem erfolgt eine Festlegung auf das in diesem Buch verwendete Entwicklungsboard.

In diesem Kapitel möchte ich Ihnen zunächst einige Begriffe und Bezeichnungen erläutern, die Ihnen bei der Arbeit mit dem ESP32 oftmals begegnen werden. Nach einer kurzen Vorstellung der unterschiedlichen Boards, Chips und Module können Sie in Abschnitt 1.5 mit der Installation loslegen.

1.1 Die ESP32-SoCs

Unter *System-on-a-Chip* (SoC, dt. »Ein-Chip-System«), auch System-on-Chip, versteht man die Integration aller oder eines großen Teils der Funktionen eines programmierbaren elektronischen Systems auf einem Chip.

Beim ESP32 handelt es sich genau genommen um eine Chip-Familie, die aus verschiedenen Chips besteht:

- ESP32-DOWD (zwei Kerne, kein Embedded Flash),
- ESP32-DOWD-Q6 (wie ESP32-DOWD, andere Abmessungen),
- ESP32-D2WD (zwei Kerne, 16-Mbit Embedded Flash) und
- ESP32-SOWD (einfacher Prozessorkern, kein Embedded Flash).

Die übrigen Eigenschaften sind identisch. Entwickler des 32-Bit-Mikrocontrollers ist die chinesische Firma Espressif. Er ist seit 2016 auf dem Markt und zielt auf Anwendungen aus dem Bereich mobile Geräte, Wearables und *Internet of Things* (IoT).

Der Prozessorkern ist vom Typ *Xtensa LX6* und arbeitet mit einem Systemtakt von 160 MHz bis 240 MHz. Das Blockdiagramm zeigt die wesentlichen Teile des Systems (siehe Abbildung 1.1).

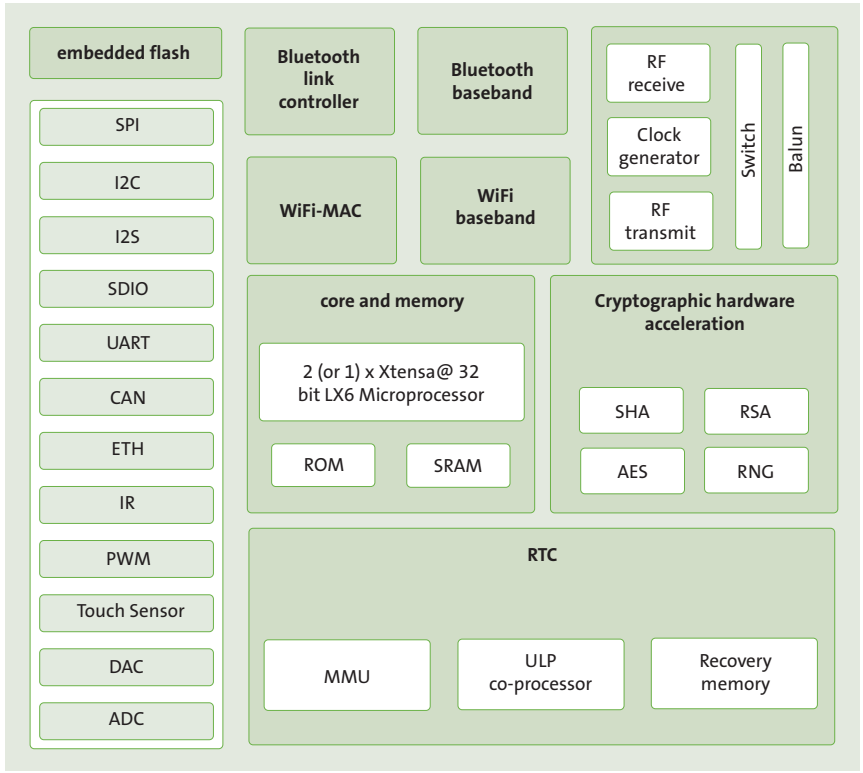

Abbildung 1.1 Blockschaltbild des ESP32

Die ESP32-Mikrocontroller weisen einige Eigenschaften auf, die sie für eine Vielzahl von Projekten interessant machen:

- Der Chip bietet **internen Speicher** in unterschiedlichen Konfigurationen:
  - 520 KB SRAM für Daten und Befehle
  - 448 KB ROM für das Booten und Kernfunktionen
  - 8 KB SRAM im RTC für Daten (RTC-FAST-Memory); sie werden von der CPU während des RTC-Boot aus dem Tiefschlaf gelesen
  - 8 KB SRAM im RTC für Daten (RTC-SLOW-Memory); sie werden vom Coprozessor während des Tiefschlafs genutzt
- Es werden bis zu **4 × 16 MB externer Speicher** unterstützt.
- Beim **Systemtakt** besteht die Möglichkeit, den internen Takt (8 MHz) oder einen externen Quarztakt mit üblicherweise 160 MHz zu nutzen. Dieser übernimmt auch bei einem Zurücksetzen des Prozessors das System-Timing.
- Ein **interner Hall-Sensor** kann für die Messung von Magnetfeldschwankungen genutzt werden.

- Ein **interner Temperatursensor** ist für einen Messbereich von –40 bis 125 Grad ausgelegt. Wie bei dem Hall-Sensor werden die analogen Messwerte von einem internen Analog-Digital-Wandler digitalisiert. Bei neueren Modellen ist der Temperatursensor jedoch nicht mehr vorhanden.
- Es sind **34 GPIOs** (universelle Ein- und Ausgänge) vorhanden. Sie dienen der Ein- und Ausgabe analoger und digitaler Signale. Die nach außen geführten Pins sind bezüglich ihrer Funktion überwiegend mehrfach belegt. Mit internen Pull-down- und Pull-up-Widerständen können definierte Zustände herbeigeführt werden.
- Der ESP32 kann Signale von bis zu zehn unterschiedlichen **kapazitiven Touch-Sensoren** verarbeiten. Die Eingänge sind diversen GPIO-Pins zugewiesen.
- **Transceiver**: Für die WLAN- und Bluetooth-Kommunikation hat der ESP32 interne 2,4-GHz-Sende- und -Empfangsbausteine.
- Das **WLAN** arbeitet nach dem Standard IEE 802.11 b/g/n.
- Es wird Bluetooth 4.2 (sowohl in der herkömmlichen als auch in der Betriebsart *Bluetooth Low Energy*) unterstützt.
- **UART** (*Universal Asynchronous Receiver Transmitter*) bezeichnet eine Form der seriellen Kommunikation. Hier stehen drei Bausteine zur Verfügung.
- Impulse bzw. Impulsflanken können mit acht **Impulszählern** erfasst werden. Bei Erreichen eines bestimmten Wertes kann ein Interrupt ausgelöst werden.
- Der ESP32 verfügt über **vier 64-Bit-Universaltimer**. Sie können softwareseitig gesteuert werden und Interrupts erzeugen.
- Es stehen **drei Watchdog-Timer** zur Verfügung. Dabei wird zwischen *Main-Watchdog-Timern* (davon zwei im Part der Universal-Timer) und *RTC-Watchdog-Timern* (einer im RTC-Modul) unterschieden. Es können ein Interrupt, ein Core-Reset oder ein CPU-Reset durch ein Reset des Watchdog-Timers ausgelöst werden.
- Es ist ein **12-Bit-A/D-Wandler** (*Analog-Digital-Wandler*) mit 18 Kanälen verbaut.
- Es können zwei unabhängige 8-Bit-DAC (Digital-Analog-Wandler) genutzt werden.
- Der ESP verfügt über **drei periphere SPI-Schnittstellen** (*SPI1*, *HSPI* and *VSPI*), die im Master- oder Slave-Modus betrieben werden können.
- Es werden **zwei I^2C-Bus-Schnittstellen** vorgehalten, die im Master- oder Slave-Modus betrieben werden können.
- **Pulsweitenmodulation** (PWM) dient dazu, Geräte wie Motoren, elektrische Heizungen oder Ähnliches zu steuern. Der Chip verfügt über ein programmierbares Hardware-PWM-Modul und 16 unabhängige Software-PWM-Module. Die Software-PWMs können bis zu 16 unabhängige digitale Wellenformen erzeugen. Tastverhältnis und Perioden sind frei wählbar. Software-PWMs eignen sich besonders zur LED-Ansteuerung.

- Ein **Infrarot-Controller**, der bis zu acht Kanäle einer programmierbaren Infrarot-Fernbedienung bedienen kann, rundet das Angebot an Schnittstellen ab.

Weitere Einzelheiten können Sie dem Datenblatt entnehmen:

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Darin finden Sie nicht nur den Funktionsumfang detailliert erläutert, sondern Sie sollten dort auch bei allen Unklarheiten nachsehen.

ESP32-S2

Mitte 2019 hat Espressif einen neuen SoC angekündigt, der im Frühjahr 2020 erschienen ist. Die Hardware ist mit nur einem Hauptprozessorkern Xtensa LX7, getaktet bei 240 MHz, etwas schwächer als der erste ESP32. Er verfügt über 320 KB SRAM und 128 KB ROM. Externe Flash-Chips werden bis zu einer Größe von einem GB, externes SPI RAM bis 128 MB unterstützt. Als IO-Optionen sind 42 GPIOs vorgesehen, die per SPI, I^2C, I2S und UART kommunizieren können, sowie ADC/DAC- und PWM-Pins. Weitere Schnittstellen können Kameras und Displays sowie ein 2,4-GHz-WLAN-Interface (IEEE 802.11 b/g/n) bedienen. Bluetooth hat der Chip im Gegensatz zu den anderen SoCs nicht. Neu ist allerdings die USB-OTG-Unterstützung.

1.2 Die Module

Espressif entwickelt und fertigt verschiedene Module und Entwicklungsplatinen, um den Anwendern die Einschätzung des Potenzials der ESP32-Chipfamilie zu erleichtern. Sie sind im Original oder auch als Klone bei den einschlägigen Bezugsquellen erhältlich.

Im Folgenden finden Sie eine kurze Beschreibung der Module, die derzeit (Stand Frühjahr 2020) von Espressif gefertigt werden. Weitere Informationen finden sich unter *https://docs.espressif.com/projects/esp-idf/en/latest/hw-reference/modules-and-boards.html*.

1.2.1 Die WROOM-, SOLO-, WROVER- und PICO-Module

Die ESP32-Modul-Familie basiert auf Modulen mit einigen integrierten Schlüsselkomponenten, einschließlich eines Quarzoszillators und einer Antennenanpassungsschaltung. Die Module sind vorgefertigte Lösungen für die Integration in Endprodukte. In Kombination mit einigen zusätzlichen Komponenten wie einer Programmierschnittstelle, Pull-up-Widerständen und Stiftleisten können diese Module auch zur Evaluierung der ESP32-Funktionalität verwendet werden.

Die wichtigsten Eigenschaften dieser Module sind in Tabelle 1.1 zusammengefasst. Einige zusätzliche Details werden in den folgenden Abschnitten behandelt. Vollständig kann diese Liste allerdings nicht sein – es werden ständig neue Modelle vorgestellt. Manche werden nur für den chinesischen Markt gefertigt, andere lassen sich nur in sehr kleinen Stückzahlen beziehen.

Module	Chip	Flash, MB	PSRAM, MB	Ant.	Abmessungen (mm)
ESP32-WROOM-32	ESP32-D0WDQ6	4	–	MIFA	18 × 25,5 × 3,1
ESP32-WROOM-32D	ESP32-D0WD	4, 8 oder 16	–	MIFA	18 × 25,5 × 3,1
ESP32-WROOM-32U	ESP32-D0WD	4, 8 oder 16	–	U.FL	18 × 25,5 × 3,1
ESP32-SOLO-1	ESP32-s0WD	4	–	MIFA	18 × 25,5 × 3,1
ESP32-WROVER (PCB)	ESP32-D0WDQ6	4	8	MIFA	18 × 31,4 × 3,3
ESP32-WROVER (IPEX)	ESP32-D0WDQ6	4	8	U.FL	18 × 31,4 × 3,3
ESP32-WROVER-B	ESP32-D0WD	4, 8 oder 16	8	MIFA	18 × 31,4 × 3,3
ESP32-WROVER-IB	ESP32-D0WD	4, 8 oder 16	8	U.FL	18 × 31,4 × 3,3

Tabelle 1.1 Die Module der ESP32-Familie

Zu den Punkten der Tabelle gehören allerdings noch einige Erläuterungen:

- ESP32-**D** kennzeichnet einen Dual-Core-Chip, ESP32-**S** kennzeichnet einen Single-Core-Chip.
- MIFA steht für *Meandered Inverted-F Antenna*.
- U.FL und U.FL/IPEX-Antennen-Konnektor
- ESP32-WROOM-32x-, ESP32-WROVER-B- und ESP32-WROVER-IB-Module haben üblicherweise 4 MB Flash-Memory, sie sind aber auch mit Flash-Memory-Größen von 8 MB und 16 MB erhältlich.
- Die Erstversion des ESP32-WROVER-Moduls hatte 4 MB PSRAM.
- ESP32-WROOM-32 wurde früher als ESP-WROOM-32 bezeichnet.

ESP32-WROOM-32

Der ESP32-WROOM-32 ist ein grundlegendes und allgemein das gebräuchlichste ESP32-Modul mit integriertem ESP32-D0WDQ6-Chip. Es war das erste Modul der WROOM/WROVER-Familie, das auf den Markt kam (siehe Abbildung 1.2).

Abbildung 1.2 Modul ESP-WROOM-32, Vorder- und Rückseite

Das Modul wird mit den Pins direkt auf die Platine gelötet. Das Pinout des Moduls sehen Sie in Abbildung 1.3.

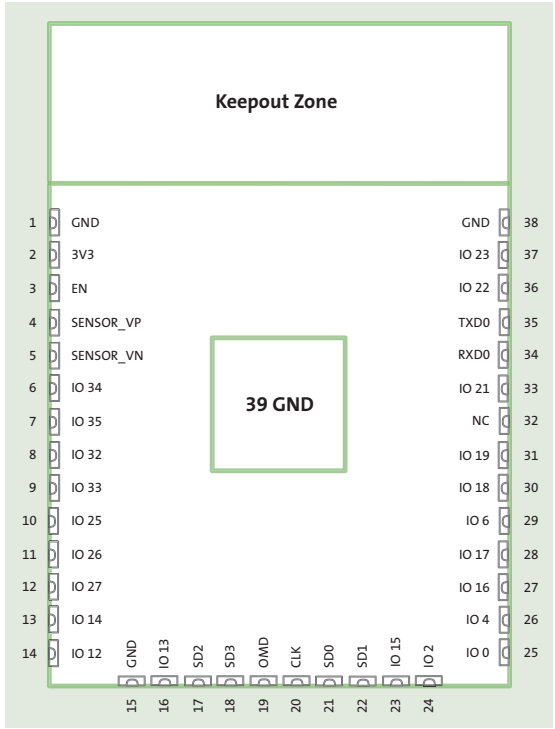

Abbildung 1.3 ESP32-WROOM-32, Pinout

Das Datenblatt steht unter *https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf* zur Verfügung.

ESP32-WROOM-32D und ESP32-WROOM-32U OOM-32

Beide Module integrieren den ESP32-D0WD-Chip, der weniger Platz benötigt als der im ESP32-WROOM-32 installierte Chip ESP32-D0WDQ6. Der ESP32-WROOM-32U ist der kleinste Vertreter der gesamten WROOM/WROVER-Modulfamilie.

ESP32-SOLO-1

Der ESP32-SOLO-1 ist eine vereinfachte Version des ESP32-WROOM-32D-Moduls. Es enthält einen Single-Core-ESP32-Chip, der eine Taktfrequenz von bis zu 160 MHz unterstützt.

ESP32-WROVER-Serie

Die ESP32-WROVER-Serie besteht aus einigen Modifikationen der ESP32-WROOM-32x-Module, die unter anderem ein zusätzliches 8-MB-SPI-PSRAM (*Pseudo Static RAM*) enthalten.

- ESP32-WROVER (PCB) und ESP32-WROVER (IPEX) verfügen über ein PSRAM, das mit 1,8 V betrieben wird und eine Taktrate von bis zu 144 MHz unterstützt.
- ESP32-WROVER-B und ESP32-WROVER-IB verfügen über ein PSRAM, das mit 3,3 V betrieben wird und eine Taktrate von bis zu 133 MHz unterstützt.

1.3 Die Boards

Der ESP32-Chip wird auf einer Vielzahl von Boards verwendet. Der Hersteller Espressif führt selbst einige Module in seinem Programm auf.

- ESP32-PICO-KIT ist das kleinste Entwicklungsboard von Espressif. Es passt in ein Mini-Breadboard, ist aber mit der minimalen Anzahl diskreter Komponenten voll funktionsfähig, weil alle ESP32-Pins zugänglich sind.
- Das ESP-WROVER-KIT ist das vielseitige Entwicklungsboard von Espressif. Es hat eine umfangreiche Liste von Funktionen, z. B. LCD-Anzeige an Bord, JTAG, Kamera-Header, RGB-LEDs usw. Es können verschiedene Module aufgenommen werden (ESP32-WROVER, ESP-WROOM-32). Mehrere Stiftleisten und Steckbrücken machen die Anbindung und Konfiguration des ESP-WROVER-KIT sehr flexibel.
- Das ESP32-Dev-KitC ist eine Entwicklungsplattform für Einsteiger. Nahezu alle ESP32-Pins wurden herausgeführt und können einfach angeschlossen und verwendet werden.

- ESP32-LyraTD-MSC ist eines der Audio Development Boards von Espressif, die auf die Spracherkennungsanwendungen ausgerichtet sind. Es unterstützt auch Nahfeld- und Fernfeld-Weckfunktionen.

Einen Überblick über die aktuellen Module und Boards von Espressif finden Sie auf der Seite *https://docs.espressif.com/projects/esp-idf/en/latest/hw-reference/modules-and-boards.html*.

Darüber hinaus wird der ESP32 auch von anderen Herstellern und Distributoren für eigene Boards eingesetzt.

- **LoLin32 ESP32 Development Board**: Dieses Board ist in der Regel mit 4 MB Flash-Speicher und einer Lithium-Batterie ausgestattet.
- **ESP32-OLED Development Board**: Dieses Board hat bereits ein OLED-Display.
- **SparkFun ESP32 Thing**: Die Firma SparkFun bietet eine Fülle von Modulen, Sensoren und Platinen für eigene Entwicklungen an. Das von ihr vertriebene ESP32-Board hat bereits einige zusätzliche LEDs und Tasten.
- Das **ESP32-HiLetGo Development Board** kommt dem ESP32-Dev-KitC sehr nahe. Der Fritzing Part wird zum Download angeboten.
- **Adafruit HUZZAH32 – ESP32 Feather Board** ist ein Development Board von Adafruit. Auch hier wird der Fritzing Part zum Download angeboten.

Ob es sich nun um ein Original-Board von Espressif handelt, um einen mehr oder weniger gelungenen Klon oder um die Eigenentwicklung eines Herstellers, die mehr oder weniger gut funktioniert, lässt sich im Vorhinein nur schwer einschätzen. Wesentlich ist hier das Vertrauen in die Bezugsquelle. Beim Erwerb preiswerter Ware aus Fernost lässt sich das Risiko etwas streuen, indem Sie das gleiche Bauteil bei unterschiedlichen Anbietern erwerben.

1.4 Das ESP32-Dev-KitC V4

Momentan zählt das ESP32-Dev-KitC V4 zu den gängigsten Entwicklungsboards auf Basis des ESP32. Aus diesem Grund soll es auch für die Beispiele in diesem Buch verwendet werden; die Kerninformationen zu diesem Modul lassen sich auch auf viele andere Varianten übertragen.

Abbildung 1.4 zeigt die wesentlichen Bauteile des ESP32-Development-Kit V4. Das Board selbst kann auf jedes Breadboard gesteckt werden. Mit den Maßen von ca. 53 mm × 25 mm lassen sich komfortabel sehr kompakte Schaltungen realisieren. Nur in den seltensten Fällen werden Sie auf das nackte Modul ausweichen müssen.

Womit Sie es genau zu tun haben, entnehmen Sie Abbildung 1.4.

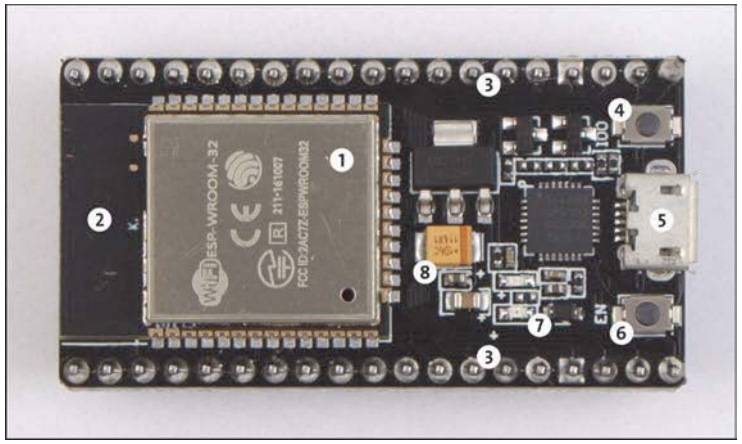

Abbildung 1.4 ESP32-Dev-KitC V4

❶ **ESP32-WROOM-32 Modul**: Bei manchen Boards kann hier auch das größere ESP32-WROVER-Modul sein.

❷ **Antenne**

❸ **Header-Pins**: Die meisten Pins des ESP-Moduls sind auf die Pin-Header auf der Platine herausgeführt. Es gibt zwei Reihen zu je 19 Pins.

❹ **Reset-Knopf**: Bei manchen Boards ist dieser Knopf auch als **EN** bezeichnet.

❺ **USB-Anschluss**: Stromversorgung für das Board sowie die Kommunikationsschnittstelle zwischen einem Computer und dem ESP32-Modul.

❻ **Boot-Knopf** (Download-Knopf): Je nach gewählter Flash-Methode bzw. Entwicklungsumgebung ist der Knopf für das Aufspielen der Firmware zu betätigen. Es gibt da unterschiedliche Szenarien:

- Arduino-IDE: Manchmal wird der Upload ohne zusätzliches Betätigen von Knöpfen initiiert. In der Regel müssen Sie aber den Boot-Knopf für das Initiieren des Uploads drücken.
- Offiziell: Halten Sie den Boot-Knopf gedrückt, um den Firmware-Download-Modus zum Herunterladen der Firmware über die serielle Schnittstelle zu starten und drücken Sie ggf. anschließend EN für einen Reboot.

❼ **LED**: Sie leuchtet kurz auf, wenn an das Board Spannung angelegt wird. (Bei manchen alternativen Boards ist die LED auch mit GPIO2 verbunden.)

❽ **USB-UART-Bridge**: Verbaut ist eine CP2102 USB to UART Bridge, die Transferraten bis zu 3 Mbps unterstützt.

Stromversorgung

Das Modul arbeitet mit einer Spannung von typischerweise +3,3 V. Die einzelnen Stromversorgungsmöglichkeiten sind:

- Micro-USB-Port: Dies wird als hauptsächliche Stromversorgungsoption gesehen. Da USB normalerweise mit +5 V arbeitet, wird die Spannung durch einen On-Board-Chip auf +3,3 V reduziert. Warnung: Periphere Geräte können nicht ausnahmslos und unbesehen an den 3V3-Ausgang angeschlossen werden, da der On-Board-Chip nur begrenzt belastbar ist.
- 5V/GND-Header-Pins
- 3V3/GND-Header-Pins

Je nach Einsatz (z. B. intensive WLAN-Nutzung) kann der ESP32 viel Strom ziehen. Laut dem Datenblatt können dies bis zu 500 mA sein.

1.4.1 Das Pinout

Das Pinout des ESP32-Dev-KitC V4, soweit es in dem Buch von Bedeutung ist, finden Sie in Abbildung 1.5.

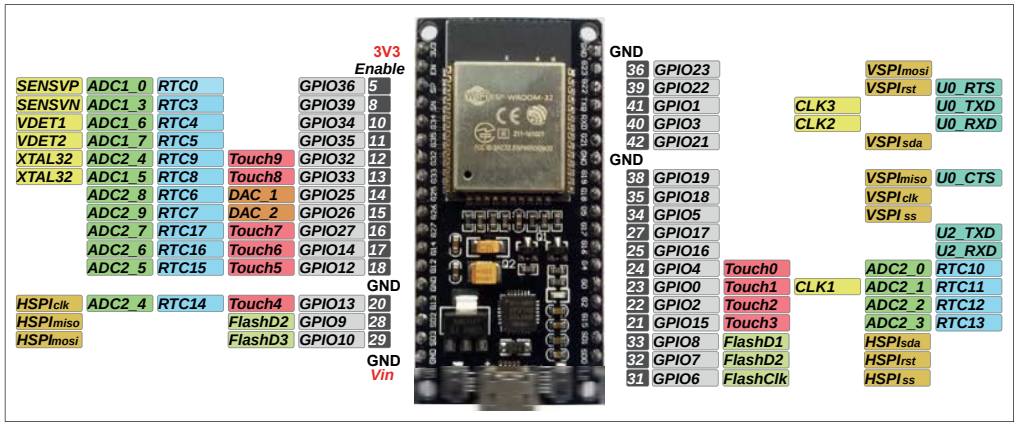

Abbildung 1.5 ESP32-Dev-KitC V4, Pinout

Die einzelnen Pins sind auf dem Board entsprechend bezeichnet. Dabei kann die aufgedruckte Beschriftung je nach Hersteller unterschiedlich sein. So bezeichnen manche Hersteller z. B. den Pin GPIO mit G14 (siehe Abbildung 1.5), manche mit D14 oder auch nur 14.

Auch ist es so, dass das Pinout nicht für alle Boards normiert ist; jeder Hersteller gönnt sich da eigene Freiheiten. Im Extremfall kann sogar bei besonders einfach gehaltenen Klonen der Aufdruck bei einzelnen Pins gänzlich unzutreffend sein, obwohl die eigentliche Zuordnung richtig ist. Deshalb ist es besonders wichtig, sich vor jeder Inbetriebnahme mit dem Pinout zu befassen, und grundsätzlich ist natürlich etwas Vorsicht beim Anschluss geboten.

In dem obigen Schaubild ist den meisten Pins eine weiße, grau gekästelte Zahl zugeordnet. Sie weist auf den entsprechenden Pin des Moduls.

Die GPIO-Pins sind spannungssensitiv. Es kann also zu Schädigungen oder Totalausfall des Mikrocontrollers führen, wenn die an einen Pin angelegte Spannung den Wert von 3,3 V überschreitet. Die Pins können nur mit maximal 40 mA belastet werden. Eine höhere Belastung kann ebenfalls zur Zerstörung des Boards führen.

1.4.2 Funktionen der GPIO-Pins

Das Schaubild zeigt auch, dass Pins durchaus mehrere Funktionen haben können. Darüber hinaus wird ein akribischer Leser feststellen, dass nicht alle GPIOs des Moduls auf dem Board nach außen geführt werden. Das ist dem Umstand geschuldet, dass manche Pins für interne Zwecke reserviert sind. Darüber hinaus gibt es Pins mit spezifischen Merkmalen, die die allgemeine Funktion einschränken und sie nur für bestimmte Zwecke einsetzbar machen.

Eine Funktionsübersicht der Pins finden Sie in Tabelle 1.2.

GPIO	Input	Output	Bemerkungen
0	pulled up	OK	gibt PWM-Signal beim Boot aus
1	TX PIN	OK	Debug Output beim Boot
2	OK	OK	verbunden mit Board-LED
3	OK	RX Pin	HIGH beim Boot
4	OK	OK	
5	OK	OK	gibt PWM-Signal beim Boot aus
6	X	X	verbunden mit internem SPI-Flash
7	X	X	verbunden mit internem SPI-Flash
8	X	X	verbunden mit internem SPI-Flash
9	X	X	verbunden mit internem SPI-Flash
10	X	X	verbunden mit internem SPI-Flash
11	X	X	verbunden mit internem SPI-Flash

Tabelle 1.2 Funktionen der GPIO-Pins

GPIO	Input	Output	Bemerkungen
12	OK	OK	bei Boot-Fehler auf HIGH gesetzt
13	OK	OK	
14	OK	OK	gibt PWM-Signal beim Boot aus
15	OK	OK	gibt PWM-Signal beim Boot aus
16	OK	OK	
17	OK	OK	
18	OK	OK	
19	OK	OK	
20	OK	OK	
21	OK	OK	
22	OK	OK	
23	OK	OK	
24	OK	OK	
25	OK	OK	
26	OK	OK	
27	OK	OK	
28	OK	OK	
29	OK	OK	
30	OK	OK	
31	OK	OK	
32	OK	OK	
33	OK	OK	
34	OK		nur Input

Tabelle 1.2 Funktionen der GPIO-Pins (Forts.)

GPIO	Input	Output	Bemerkungen
35	OK		nur Input
36	OK		nur Input
39	OK		nur Input

Tabelle 1.2 Funktionen der GPIO-Pins (Forts.)

Interner SPI-Flash des ESP-WROOM-32

Die GPIOs 6 bis 11 können nicht genutzt werden, obwohl manche Boards sie nach außen führen. Grund ist, dass sie für den internen SPI reserviert sind.

GPIO-Pins 34 bis 39

Die GPIOs 34 bis 39 sind keine richtigen GPIOs, da sie nur als Input funktionieren (GPI). Sie haben keine internen Pull-up- oder Pull-down-Widerstände. Folgende GPIOs sind also nur Input-GPIOs:

- GPIO34
- GPIO35
- GPIO36
- GPIO39

PWM

Der ESP32-LED-PWM-Controller hat 16 unabhängige Kanäle, die mit unterschiedlichen Eigenschaften konfiguriert werden können. Alle richtigen Output-Pins können PWM generieren (also nicht die GPIOs 34 bis 39).

Interrupts

Alle GPIOs können für Interrupts konfiguriert werden.

Touch-GPIOs

Der ESP32 verfügt über zehn interne kapazitive Berührungssensoren. Diese können Schwankungen von elektrischen Ladungen registrieren, die z. B. durch Kontakt mit der menschlichen Haut entstehen (Berühren der GPIOs mit einem Finger). Anwendungsfall ist u. a. die Integration in kapazitive Touchpads, wo sie mechanische Tasten ersetzen können. Über die kapazitiven Touch-Pins kann der ESP32 auch aus dem Tiefschlaf erweckt werden. Die Touch-Pins sind folgendermaßen zugeordnet:

- T0 (GPIO4)
- T1 (GPIO0)
- T2 (GPIO2)
- T3 (GPIO15)
- T4 (GPIO13)
- T5 (GPIO12)
- T6 (GPIO14)
- T7 (GPIO27)
- T8 (GPIO33)
- T9 (GPIO32)

Analog to Digital Converter (ADC)

Der ESP32 hat 18 ADC-Input-Kanäle, die mit 12-Bit-Auflösung operieren. Das bedeutet, dass die Kanäle analoge Messwerte im Bereich von 0 bis 4095 aufnehmen können, wobei »0« 0 V und »4095« 3,3 V entspricht. Auflösung und ADC-Bereich können softwareseitig eingestellt werden.

- ADC1_CH0 (GPIO36)
- ADC1_CH1 (GPIO37)
- ADC1_CH2 (GPIO38)
- ADC1_CH3 (GPIO39)
- ADC1_CH4 (GPIO32)
- ADC1_CH5 (GPIO33)
- ADC1_CH6 (GPIO34)
- ADC1_CH7 (GPIO35)
- ADC2_CH0 (GPIO4)
- ADC2_CH1 (GPIO0)
- ADC2_CH2 (GPIO2)
- ADC2_CH3 (GPIO15)
- ADC2_CH4 (GPIO13)
- ADC2_CH5 (GPIO12)
- ADC2_CH6 (GPIO14)
- ADC2_CH7 (GPIO27)
- ADC2_CH8 (GPIO25)
- ADC2_CH9 (GPIO26)

ADC2-Pins und WLAN

Die ADC2-Pins können nicht gleichzeitig mit WLAN genutzt werden. Bei der Nutzung von WLAN muss auf die ADC1-Pins ausgewichen werden. Darüber hinaus ist der Verlauf nicht vollständig linear. So ist es nicht möglich, zwischen 0 und 0,1 V sowie zwischen 3,2 und 3,3 V zu unterscheiden.

Digital to Analog Converter (DAC)

Es gibt zwei DAC-Kanäle mit einer 8-Bit-Auflösung. Sie konvertieren digitale Signale oder analoge Werte in analoge Signale. Die Kanäle sind:

- DAC1 (GPIO25)
- DAC2 (GPIO26)

RTC-GPIOs

RTC steht für Real Time Clock. Der ESP32 bietet RTC-GPIO-Unterstützung. Das RTC-System selbst benötigt extrem wenig Strom und ist verantwortlich für die verschiedenen Energiesparmodi, die bis zum Tiefschlaf reichen. Mit den RTC-GPIOs kann der ESP32 geweckt werden. Die folgenden GPIOs können als externe Weckquelle verwendet werden.

- RTC_GPIO0 (GPIO36)
- RTC_GPIO3 (GPIO39)
- RTC_GPIO4 (GPIO34)
- RTC_GPIO5 (GPIO35)
- RTC_GPIO6 (GPIO25)
- RTC_GPIO7 (GPIO26)
- RTC_GPIO8 (GPIO33)
- RTC_GPIO9 (GPIO32)
- RTC_GPIO10 (GPIO4)
- RTC_GPIO11 (GPIO0)
- RTC_GPIO12 (GPIO2)
- RTC_GPIO13 (GPIO15)
- RTC_GPIO14 (GPIO13)
- RTC_GPIO15 (GPIO12)
- RTC_GPIO16 (GPIO14)
- RTC_GPIO17 (GPIO27)

I²C

Der ESP32 hat zwei *I²C*-Kanäle. Dabei kann jeder Pin als SDA oder SCL fungieren.

Die Arduino-IDE hat die Pins für einen Kanal vorbelegt (SDA: GPIO21, SVL: GPIO22).

SPI

Es stehen zwei *SPI*-Kanäle zur Verfügung. Die Voreinstellung sehen Sie in Tabelle 1.3.

SPI	MOSI	MISO	CLK	CS
VSPI	GPIO23	GPIO19	GPIO18	GPIO5
HSPI	GPIO13	GPIO12	GPIO14	GPIO15

Tabelle 1.3 SPI-GPIOs

Strapping Pins

Ein IC kann eine oder mehrere Betriebsarten aufweisen. Festgelegt wird dies über einzelne Konfigurations-Pins. Sofern ein IC genügend Pins hat, können hierfür diese (Eingangs-)Pins auf LOW-Pegel (z. B. GND) bzw. HIGH-Pegel (z. B. 3,3 V) gelegt werden. Soll jedoch die Anzahl der Pins z. B. für ein kleines Gehäuse minimiert werden, werden zum Konfigurieren (*Strappen*) merkwürdigerweise Output-Pins genutzt. Über diese wird dann direkt nach dem Einschalten oder nach einem Reset als Erstes die gewünschte Konfiguration eingelesen.

Der ESP32 hat folgende Strapping Pins:

- GPIO0
- GPIO2
- GPIO4
- GPIO5 (muss während des Boots HIGH sein)
- GPIO12 (muss während des Boots LOW sein)
- GPIO15 (muss während des Boots HIGH sein)

Diese GPIOs werden benötigt, um den ESP32 in den Bootloader- oder Flash-Modus zu versetzen. Allerdings ist es bei dem ESP32-Dev-KitC V4, wie übrigens bei den meisten anderen Entwicklungsboards auch, so, dass das Board die richtigen Zustände für die verschiedenen Modi setzt. Weitere recht ausführliche Informationen finden sich in der Dokumentation *ESP32-Boot-Mode-Selection* (*https://github.com/espressif/esptool/wiki/ESP32-Boot-Mode-Selection*).

Probleme beim Hochladen

Es kann jedoch zu Problemen beim Hochladen von neuem Code, beim Flashen des ESP32 mit neuer Firmware oder beim Reset des Boards kommen, wenn diese Pins mit Peripheriegeräten belegt sind. Dies kann darauf zurückzuführen sein, dass diese Geräte den ESP32 hindern, in den richtigen Modus zu wechseln. Auch hier kann die oben genannte Dokumentation weiterhelfen. Hilfreich ist in solchen Fällen auch, die gesamte Peripherie abzutrennen und schrittweise wieder anzuschließen, um so das störende Element zu isolieren.

Pins HIGH bei Boot

Einige GPIOs wechseln beim Booten oder beim Reset ihren Status zu HIGH oder geben PWM-Signale aus. Für möglicherweise angeschlossene Peripherie bedeutet dies, dass es bei ihnen dann zu unerwartetem Verhalten kommen kann.

- GPIO1
- GPIO3
- GPIO5
- GPIO6 bis GPIO11
  Diese sind mit dem ESP32-integrierten SPI Flash Memory verbunden und sollten deshalb nicht benutzt werden.
- GPIO14
- GPIO15

Enable (EN)

Enable (EN) ist der Enable-Pin für den 3,3-V-Spannungsregler. Normalerweise liegt er per Widerstand auf HIGH (pulled up). Der Spannungsregler wird außer Betrieb gesetzt, wenn der Pin auf LOW (GND) gesetzt wird. Auf diese Weise lässt sich mit einem Button ein Restart des ESP32 erzwingen.

1.4.3 Höhere Eingangsspannung an GPIOs

Bedauerlicherweise ist es so, dass verschiedene im Mikrocontrollerbereich eingesetzte Sensoren nur mit Spannungen über 3,3 V fehlerfrei betrieben werden können. Dementsprechend liegt ihr Ausgangspegel in der Regel ebenfalls über dieser Marke. Das führt gewissermaßen zu einem Dilemma, denn eine zu hohe Eingangsspannung an einem GPIO kann das Modul zerstören; auf der anderen Seite ist vielleicht gerade dieses Bauteil unverzichtbar. Die Eingangsspannung muss also abgesenkt werden. Hierzu gibt es zwei praktikable Möglichkeiten.

Spannungsteiler

Bei einem *Spannungsteiler* wird die Spannung durch eine Reihenschaltung passiver Elemente aufgeteilt. Einfache Spannungsteiler können bereits mit ein bzw. zwei Widerständen realisiert werden. Für die Absenkung der Eingangsspannung an einem GPIO könnte die Schaltung etwa wie folgt aussehen (siehe Abbildung 1.6).

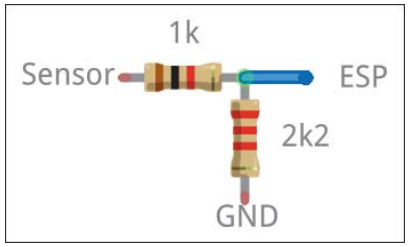

Abbildung 1.6 Sensor-Spannungsteiler mit Widerständen

Eine weitere Möglichkeit ist die Spannungsbegrenzung des Eingangssignals mit einer Zenerdiode (siehe Abbildung 1.7).

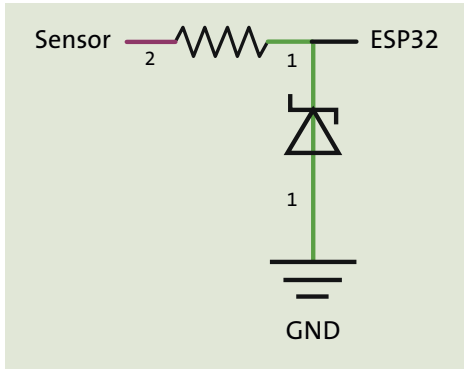

Abbildung 1.7 Sensor-Spannungsteiler mit Zenerdiode

Pegelwandler (Level Shifter)

Eleganter, aber auch etwas aufwändiger sind integrierte Pegelwandler (Pegelumsetzer, *Level Shifter*). Pegelwandler können unidirektional, also nur in eine Richtung (z. B. bei Sensoren), oder bidirektional, also in zwei Richtungen für BUS-Systeme, arbeiten. Entsprechende Module sind auf dem Markt verfügbar (Abbildung 1.8).

Die Anschlüsse sind:

- GND: für den gemeinsamen Ground-Anschluss
- HV: High-Voltage, das hohe VCC, z. B. 5 V
- LV: Low-Voltage, das niedrige VCC, z. B. 3,3 V

- HV1–HV4: die Ein-/Ausgänge Hochvolt
- LV1–LV4: die Ein-/Ausgänge Niedrigvolt

Abbildung 1.8 Pegelwandler

Der Einsatz von Pegelwandlern kann aber auch mit ein paar Nachteilen einhergehen:

- Überlastung einer oder beider Seiten bis hin zur Zerstörung
- inkompatible Logikpegel und daraus resultierendes Nichtfunktionieren der Schaltung oder, noch schlimmer, sporadische Fehlfunktionen
- Verzögerungen der Signale durch die Pegelwandlung und daraus resultierende maximale Signalfrequenzen

1.5 ESP32-Dev-KitC V4 – Erstinbetriebnahme am PC

Nun können Sie das ESP32-Dev-KitC V4 über ein Micro-USB-B-Kabel erstmalig mit Ihrem PC verbinden. Die Betriebszustands-LED sollte einige Male deutlich rot blinken, bei anderen Boards kann sie dauerhaft leuchten. Es liegt Spannung an, und der On-Board-Spannungsregler arbeitet einwandfrei.

1.5.1 Windows

Das Board sollte von Windows selbständig erkannt werden. Dies lässt sich einfach im Geräte-Manager (Windows • Systemsteuerung • Hardware und Sound • Geräte-Manager) überprüfen (siehe Abbildung 1.9).

Hier ist das Modul an Port COM4 angeschlossen. Windows hat die USB zu UART Bridge CP210x erkannt und den nötigen Treiber installiert.

Sollte das Modul nicht identifiziert worden sein, kann das an fehlenden Datenleitungen im USB-Kabel liegen. Können Sie dies ausschließen, hilft eventuell eine manuelle Installation des Treibers für die CP2102-Bridge (siehe Bauteil 8 des ESP32-Development-Kit V4).

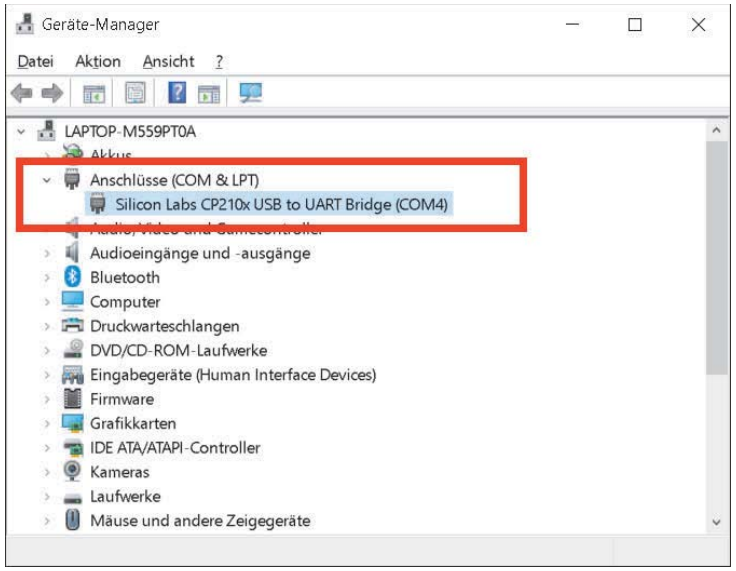

Abbildung 1.9 ESP32-Dev-KitC V4 im Geräte-Manager unter Windows

1.5.2 Linux

Auch Linux erkennt üblicherweise das Board eigenständig – erfolgreich getestet wurde dies mit Ubuntu 18.04. Die Überprüfung erfolgt in einem Terminal-Fenster mit dem Befehl `lsusb` (siehe Abbildung 1.10).

Abbildung 1.10 ESP32-Development-Kit V4 an Linux (Ubuntu)

Damit schließt die Beschreibung der Hardware rund um den Chip ESP32, die von Espressif gefertigten Module und erhältlichen Boards. Dabei hat sich das ESP32-Dev-KitC V4 als äußerst vielfältiger Baustein und dennoch recht einfach zu handhabende Komponente präsentiert, die geradewegs dazu einlädt, es für eigene Projekte zu nutzen. Deshalb ist dieser Baustein die Basis für die folgenden Projekte.

Kapitel 6

Peripherie und Funktionen des ESP32

Mit diesem Kapitel beginnt der Anwendungsteil des Buches. Es befasst sich mit internen und externen Sensoren. Darüber hinaus werden eingebaute Funktionen zur Steuerung des ESP32 vorgestellt.

Mit dem ESP32 können Sie noch deutlich spannendere Sachen machen, als LEDs leuchten zu lassen. Dafür benötigen Sie allerdings zusätzliche Hardware, die ich Ihnen hier vorstellen möchte.

6.1 Der interne Hall-Sensor

Das ESP32-Dev-KitC V4 bietet mit dem *internen Hall-Sensor* eine einfache Möglichkeit, um zunächst ohne den Ballast zusätzlicher Hardware einen Demonstrationssketch zu erstellen.

Ein Hall-Sensor greift das Auftreten einer Spannung in einem stromdurchflossenen Leiter auf, wenn dieser sich in einem Magnetfeld befindet (*Hall-Effekt*, nach dem Physiker Edwin Hall). Elektrisch fällt dabei die Spannung senkrecht sowohl zur Stromfluss- als auch zur Magnetfeldrichtung am Leiter ab. Dieser Spannungsabfall kann gemessen und zu Steuerungszwecken genutzt werden. Zum Einsatz kommen derartige Sensoren in der Fertigungstechnik, bei Zugangskontrollen oder bei der Regelungstechnik. Der ESP32 verfügt ebenfalls über einen internen Hall-Sensor. Der Beispielsketch aus Listing 6.1 demonstriert das Auslesen von Werten des internen Hall-Sensors.

```
void setup() {
  Serial.begin(115200);
}
void loop() {
  int messWert = hallRead();
  Serial.print("Hall sensor Messung: ");
  Serial.println(messWert);
  delay(3000);
}
```

Listing 6.1 Hall-Sensor

Gelangt ein Magnetfeld in den Bereich des Hall-Sensors, ändern sich die Messwerte und werden im seriellen Monitor dargestellt (siehe Abbildung 6.1).

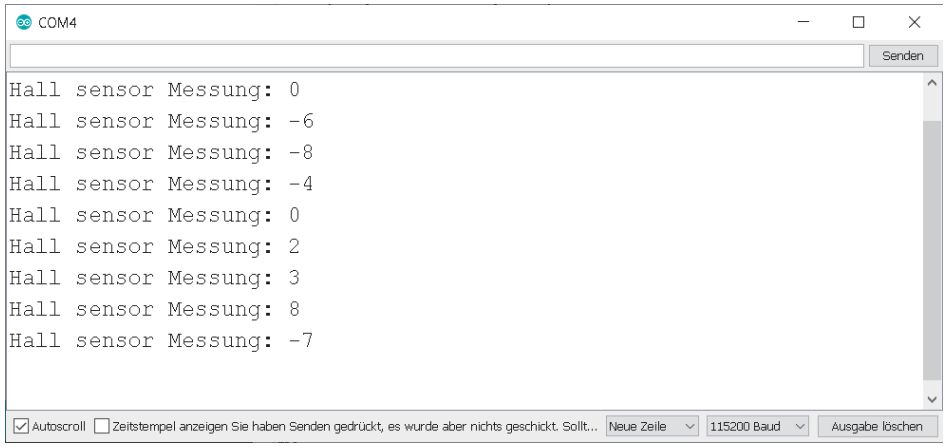

Abbildung 6.1 Messwerte des Hall-Sensors im seriellen Monitor

Bedauerlicherweise braucht der interne Hall-Sensor einiges an magnetischer Kraft, um Veränderungswerte anzuzeigen (Messwerte hier: freilaufend –4 bis –8; einfacher Magnet unmittelbar auf der Moduloberfläche 2 bis 8). Geeignete Magnete sind Neodym-Magnete, und auch diese müssen schon recht nahe an die Moduloberfläche gebracht werden.

Sollte in Ihrem Projekt ein Hall-Sensor zum Einsatz kommen, ist es sicherlich eine Überlegung wert, auf externe Sensormodule zurückzugreifen. Es gibt sie als fertiges Modul (z. B. digitaler Hall-Sensor Iduino SE054) oder auch als einzelnes Bauteil.

Der alte Temperatursensor

Im Netz sind noch Hinweise und Programmierbeispiele bezüglich eines weiteren internen Sensors, nämlich eines Temperatursensors, zu finden. In aktuellen Modulen bzw. Boards ist er nicht mehr verbaut. Da er ohnehin aufgrund der Nähe zu einem sich im Betrieb erwärmenden Modul nur bedingt aussagefähige Messwerte geliefert hat, wiegt der Verlust nicht besonders schwer.

6.2 Digitale Aus-/Eingänge

»Digital«, das klingt zunächst einfach. Ist es auch, aber es gibt ein paar Fallstricke.

Der programmtechnische Einstieg in die digitale Aus- und Eingabe ist, den vorgesehenen Pin entsprechend zu konfigurieren. Die Syntax ist:

```
pinMode(pin, mode);
```


`pin` kennzeichnet den Pin nach dem Pinout (siehe Kapitel 1). `mode` kennzeichnet den »Betriebsmodus« und ist entweder `OUTPUT` für die digitale Ausgabe oder `INPUT` bzw. `INPUT_PULLUP` für die digitale Eingabe.

6.2.1 Digitale Ausgänge

Die meisten – aber nicht alle! – GPIO-Pins können als digitale Ausgänge dienen (siehe Kapitel 1). Die Funktion `digitalWrite(pin, value)` bestimmt den Ausgabepegel des Pins `pin`. Er entspricht der Modulspannung (3,3 V beim ESP32), wenn der Parameter `value` gleich `HIGH` ist. Ist der Parameter `LOW`, ist dies 0 V bzw. Ground.

Ein Sketch, der digitale Ausgänge verwendet, beinhaltet immer die nachstehenden Codesegmente:

```
...
setup {
pinMode(5, OUTPUT);          // GPIO5 als OUTPUT schalten
...
}
loop {
...
digitalWrite(5, HIGH)
...
digitalWrite(5, LOW)
...
}
```

Listing 6.2 Grundbeispiel für Digital Output

Einige Nutzungsbeispiele sind im Folgenden dargestellt.

LED

Eine LED an einem GPIO-Pin ist der Klassiker einer digitalen Ausgabe. Sie wird gerne für die Anzeige von Betriebszuständen oder noch viel mehr für Testzwecke eingesetzt. Allerdings ist die Ausgangsspannung bei `HIGH` größer als die maximal zulässige Betriebsspannung der LED. Deshalb muss die LED durch einen Vorwiderstand geschützt werden. Hierfür reicht normalerweise ein 470-Ω-Widerstand (die exakte Berechnung erfolgt nach dem *ohmschen Gesetz*, das in Abschnitt 6.3.2 noch erwähnt wird.

Ein erstes Programm, das eine LED an einem GPIO an- und wieder ausschaltet, haben Sie bereits in Kapitel 4 in Form eines einfachen Blink-Programms kennen gelernt.

Relaismodul

Ein *Relaismodul* (z. B. das KY-01, siehe Abbildung 6.2) bietet eine komfortable Möglichkeit, höhere Lasten zu schalten. Das Modul kann direkt an den ESP32 angeschlossen werden. Obwohl als Betriebs- und Schaltspannung 5 V angegeben ist, funktioniert das Modul auch mit der niedrigeren Spannung der ESP32-GPIO erfahrungsgemäß einwandfrei. Die Ausgänge des Relaismoduls schalten entweder in den *Ruhe-* (Anschluss NC; *normally connected*) oder im *Arbeitszustand* (Anschluss NO, *normally open*).

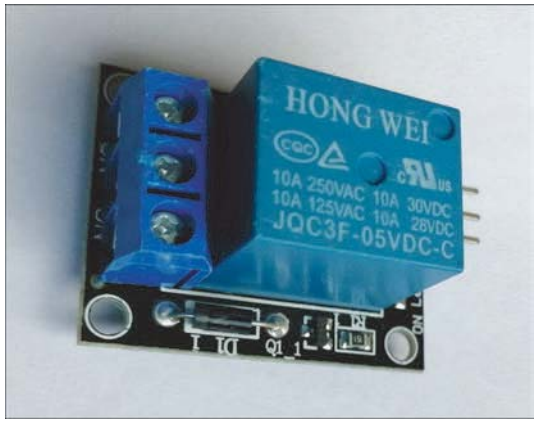

Abbildung 6.2 Relaismodul

Lebensgefahr

Bei Arbeiten mit 220 V an den Ausgängen des Relaismoduls besteht Lebensgefahr! Sie bleiben Personen mit entsprechenden Kenntnissen und Fähigkeiten vorbehalten.

Sonstige digitale Ausgänge

Unter sonstige digitale Ausgänge fallen alle anderen Komponenten, die entweder ein- oder ausgeschaltet sein können. Probleme entstehen jedoch, wenn diese Geräte für ihre Schaltvorgänge mehr Strom ziehen, als ein GPIO liefern kann (z. B. LED-Stripes). Eine Überlastung des GPIO wäre die Folge, mit der Konsequenz, dass das gesamte Modul zerstört würde. Abhilfe schafft ein in *Emitterschaltung* betriebener Transistor, z. B. ein BC547B (siehe Abbildung 6.3).

Eine weitere Besonderheit betrifft Komponenten mit einer Spule (z. B. Relais oder Motoren). Beim Ausschalten von induktiven Lasten sinkt der Strom nicht abrupt auf null. Vielmehr befindet sich in dem durch die Induktivität aufgebauten Magnetfeld noch Energie, die sich langsam abbaut. Das führt zu *Überspannungen*. Es gibt mehrere Möglichkeiten, derartige *Induktionsspitzen* zu begrenzen. Eine ist die Verwendung einer Schutzdiode (siehe Abbildung 6.4).

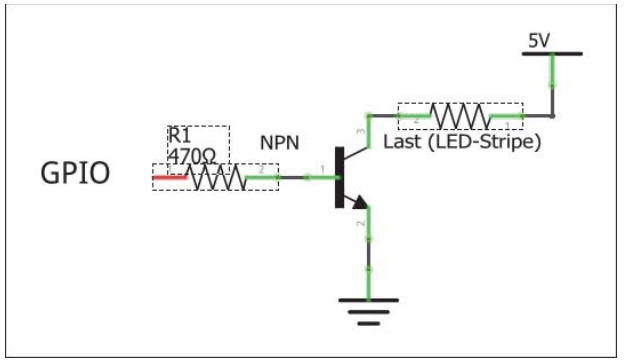

Abbildung 6.3 Digital Output mit Emitterschaltung

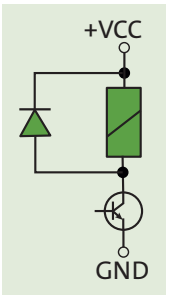

Abbildung 6.4 Digital Output mit Schutzdiode

6.2.2 Digitale Eingänge

Die als digitale Eingänge schaltbaren GPIO-Pins sind ebenfalls in Kapitel 1 aufgelistet. Die Funktion `digitalRead(pin, value)` liest den Eingangspegel des Pins `pin` und liefert `1` (`HIGH`), wenn Spannung anliegt, und `0` (`LOW`), wenn an dem Pin 0 V anliegt.

Pin nicht »offen« beschalten

Wenn ein Pin »offen«, d. h. nicht beschaltet ist, befindet sich der Eingang im Zustand »schwebend/hochohmig«. Dies ist irgendwo zwischen `HIGH` und `LOW`. Bedauerlicherweise sind Schaltungen Umgebungseinflüssen ausgesetzt. Dazu zählen auch elektrische Felder, die eine Schaltung als Wechsel zwischen `HIGH` und `LOW` interpretiert. In der Folge führen solche Störsignale ohne weitere Vorsichtsmaßnahme zu unerklärlichen und unregelmäßig auftretenden Fehlern im Programmablauf.

An dieser Stelle heißt das Zauberwort *Pull-up-Widerstand* bzw. *Pull-down-Widerstand*. Beide Schaltungsausführungen ziehen die Spannung auf ein definiertes Niveau. Während Pull-down-Widerstände die Spannung auf GND ziehen (engl.: »pull«), erfolgt das Hinaufziehen der Spannung meistens auf Betriebsspannung bzw. maximale Eingangsspannung an den GPIOs.

Das Ziehen auf ein definiertes Spannungsniveau ist zum einen über externe Hardware möglich. Dabei wird ein Widerstand (10 kΩ) gegen GND (Pull-down) oder VCC (Pull-up) geschaltet. Zum anderen bieten die meisten Mikrocontroller die Möglichkeit, interne Pull-up-Widerstände oder Pull-down-Widerstände programmgesteuert zu aktivieren. Der ESP32 verfügt über beide Wege.

Allerdings kann mit der Arduino-IDE nur der Pull-up-Widerstand angesprochen werden (`pinMode(pin, INPUT_PULLUP)`. In der Regel ist dies bei entsprechendem Programmdesign für die meisten Anwendungsfälle ausreichend. Das ESP-IDF geht da weiter und beinhaltet auch eine Funktion, die den Pull-down-Widerstand aktiviert:

`esp_err_tgpio_set_pull_mode(gpio_num_t gpio_num, gpio_pull_mode_tpull)`

Taster und Schalter

Taster bzw. Schalter sind im Prinzip nicht mehr State of the Art, seitdem es Touch-Sensoren gibt, und finden eher nur noch Ausnahmefällen Anwendung. Grund ist das *Prellen*. Es handelt sich dabei um Störeffekte, die beim Betätigen von elektromechanischen Tastern und Schaltern dadurch entstehen, dass der Kontakt z. B. beim Schließen erst hin und her federt. Aus elektronischer Sicht finden damit mehrere Öffnen- und Schließvorgänge statt, obwohl aus Sicht desjenigen, der die Taste betätigt, nur ein Schaltvorgang vorliegt. Programme registrieren dies dann aus Anwendersicht fälschlich als mehrere Schließvorgänge.

Das verdeutlicht das Beispiel aus Listing 6.3, in dem GPIO19 über einen Taster an VCC angeschlossen ist. Die Codezeilen, die für einen *Debounce*-Filter notwendig sind, habe ich fett dargestellt. Sie können also das Beispiel:

- ohne Debounce-Filter laufen lassen, indem Sie diese fett dargestellten Zeilen weglassen und die auskommentierte `if`-Abfrage am Schleifenbeginn (`if (statusIn == LOW) z++;`) aktivieren, oder es
- mit Debounce-Filter nutzen, indem Sie den Code in der dargestellten Form verwenden.

```
#define INPUT_PIN   19                  // GPIO19 für digitalRead
int i = 0, statusIn = LOW, z = 0;
int statusSchalter = LOW;               // für Debounce
int statusSchalterAlt = LOW;
unsigned long letzteDebounceZeit = 0, debounceDelay = 60;
void setup() {
  Serial.begin (115200);
  delay (2);
  pinMode(INPUT_PIN, INPUT_PULLUP);
}
```



```
void loop() {
  statusIn = digitalRead (INPUT_PIN);
  //if (statusIn == LOW)    z++;        // ohne Debounce-Filter
  if (statusIn != statusSchalterAlt) {  // Beginn Debounce-Filter
    letzteDebounceZeit = millis();
  }

  if ((millis() - letzteDebounceZeit) > debounceDelay) {
    if (statusIn != statusSchalter) {
      statusSchalter = statusIn;
      z++;
    }
  }                                     // Ende Debounce-Filter
  i++;
  if (i > 1000) {
    i = 0;
    Serial.println ("Zähler: " + String(z));
  }
  statusSchalterAlt = statusIn;
  delay (1);
}
```

Listing 6.3 Digital Input eines Tasters

Der serielle Monitor zeigt nun die Werte der Variablen `z` an, die bei Betätigen des Tasters nicht um 1, sondern ein Vielfaches von 1 erhöht wird (Abbildung 6.5).

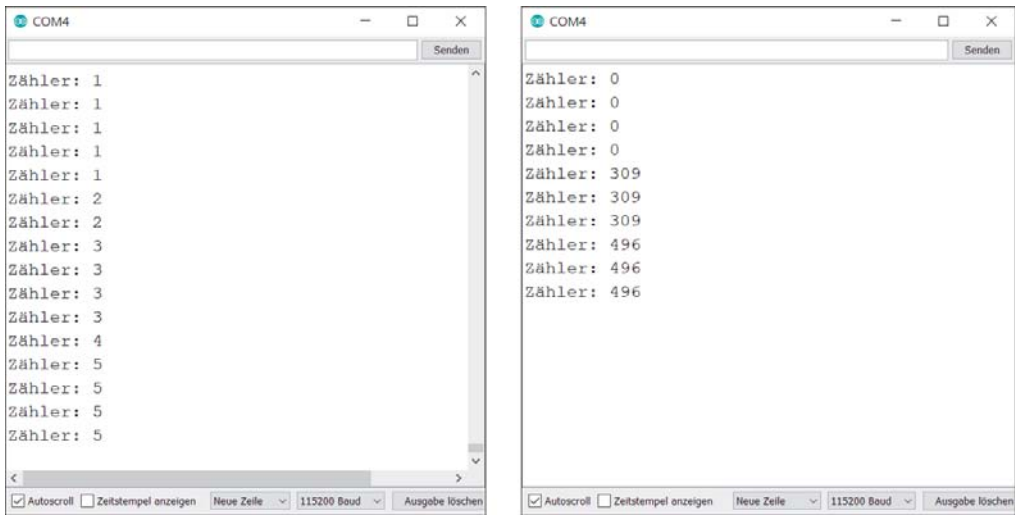

Abbildung 6.5 Taster mit und ohne Debounce-Filter

Der PIR-Bewegungsmelder HC-SR501

Bewegungsmelder erkennen Bewegungen in der Umgebung. Sie sind entweder als aktive Melder auf der Basis von elektromagnetischen Wellen (z. B. Mikrowellen, Dopplerradar, Ultraschall) konzipiert oder als passive, die anhand der Infrarotstrahlung (*PIR*-Bewegungsmelder; PIR = *Passiv-Infrarot*) arbeiten. Passive PIR-Bewegungsmelder beinhalten temperaturabhängige Pyrosensoren, die auf Infrarotlicht einer bestimmten Wellenlänge von wärmestrahlenden Objekten (z. B. der menschlichen Körpertemperatur) reagieren. Ein Vertreter dieser Kategorie ist der PIR-Bewegungsmelder HC-SR501, der hier bereits in einem fertigen Modul verbaut ist (siehe Abbildung 6.6). Der eigentliche Sensor ist unter der *Fresnel-Linse* eingebaut. Sie sorgt dafür, dass statische Wärmequellen wie Heizungen oder Lampen nicht zum Auslösen des Sensors führen.

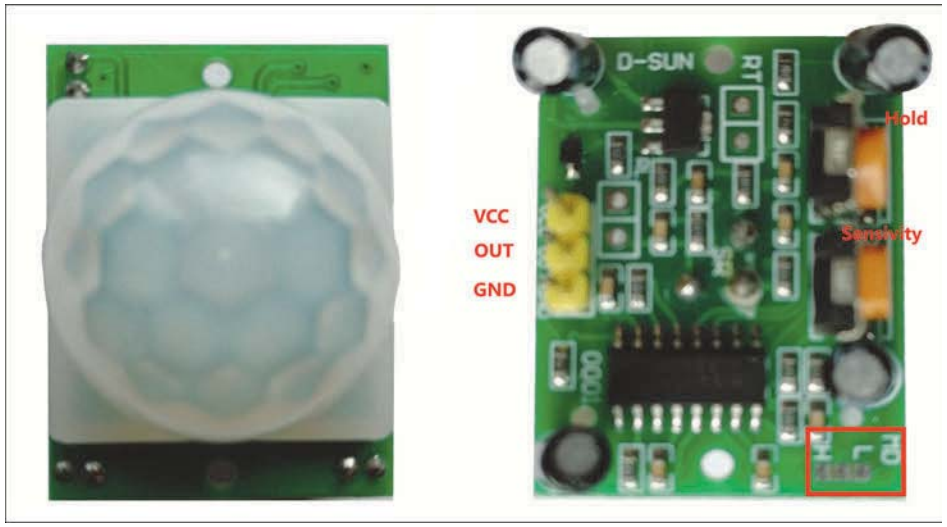

Abbildung 6.6 PIR-Bewegungsmelder HC-SR501

Das Modul verfügt über einen Betriebsspannungsbereich von 4,5 V bis 20 V Gleichspannung bei einem Ruhestrom < 60 µA. Einstellbar sind der Erkennungsabstand (3 bis 7 m) und die Verzögerungszeit (5 bis 200 Sekunden; Standard: 5 Sekunden +/–3 %). Über Lötbrücken, bei manchen Modulen auch über Jumper, kann das Auslöseverhalten (Trigger) angepasst werden:

- L: Nicht wiederholbarer Trigger, d. h., das Modul hält den Ausgang für die eingestellte Haltezeit auf `HIGH` und lässt ihn im Anschluss wieder auf `LOW` fallen. Währenddessen kann keine weitere Bewegung erkannt werden! Nach Ende der `HIGH`-Phase ist der Sensor für ca. 3 Sekunden »blind«, und erst dann können wieder neue Bewegungen erkannt werden.

- H: Repeat Trigger (Standard), d. h., neue Bewegungen, die während einer bereits erkannten Bewegung auftreten, führen dazu, dass sich der Impuls am Ausgang um die eingestellte Haltezeit verlängert.

Der Ausgang liefert ein 3,3-V-High/Low-Signal.

Die Schaltung

Die Schaltung (siehe Abbildung 6.7) ist mit nur zwei Bauteilen recht übersichtlich. Der Ausgang des PIR-Bewegungsmelders ist mit GPIO13 verbunden. Die Stromversorgung geht über den 5-V-Anschluss des ESP32-Dev-KitC V4.

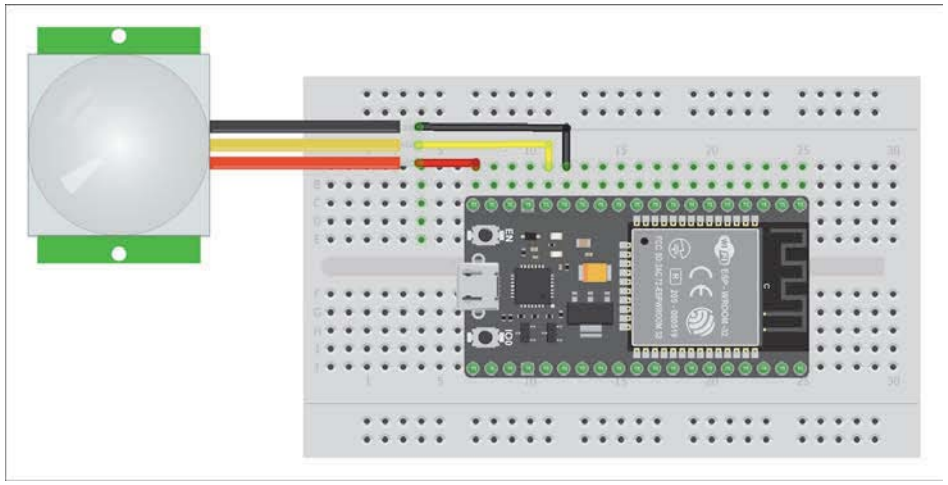

Abbildung 6.7 PIR-Bewegungsmelder

Der Sketch

Der Sketch beschränkt sich auf wenige Zeilen:

```
#define PIR_SENSOR_PIN     13          // Pin für den PIR-Sensor
int pirStatusAlt = LOW;                // PIR-Status sichern

// PIR-Ausgang lesen, auswerten und Aktion durchführen
void lesenPIR(){
  if (digitalRead(PIR_SENSOR_PIN)) {  // PIR lesen und Status prüfen
    if (pirStatusAlt == LOW) {
      Serial.println("Bewegung erkannt!");
      pirStatusAlt = HIGH;             // Status sichern
      delay (100);
     }
  } else {
    if (pirStatusAlt == HIGH){
      Serial.println("Bewegung vorbei!");
```

```
      pirStatusAlt = LOW;
      delay (100);
    }
  }
}

void setup() {
  pinMode(PIR_SENSOR_PIN, INPUT);     // öffnet Pin für den Sensor als Input
  Serial.begin(115200);
}

void loop(){
  lesenPIR();
  delay (10);
}
```

Listing 6.4 PIR-Bewegungsmelder

Der Sketch beinhaltet keine Besonderheiten. Die Funktion für das Lesen des PIR-Sensors und die Auswertung des Status ist in eine eigene Funktion ausgelagert, damit diese für spätere Projekte als »fertiges« Modul übertragen werden kann.

Die Hauptschleife ruft in diesem kleinen Sketch die Funktion `lesenPIR()` auf; dort erfolgt das Lesen und Prüfen des PIR-Sensors. Als einzige Aktion ist hier eine Ausgabe am seriellen Monitor vorgesehen (siehe Abbildung 6.8); in einem anderen Zusammenhang könnte hier das Senden von E-Mail-Benachrichtigungen, das Schalten von Alarmsirenen oder Ähnliches stehen.

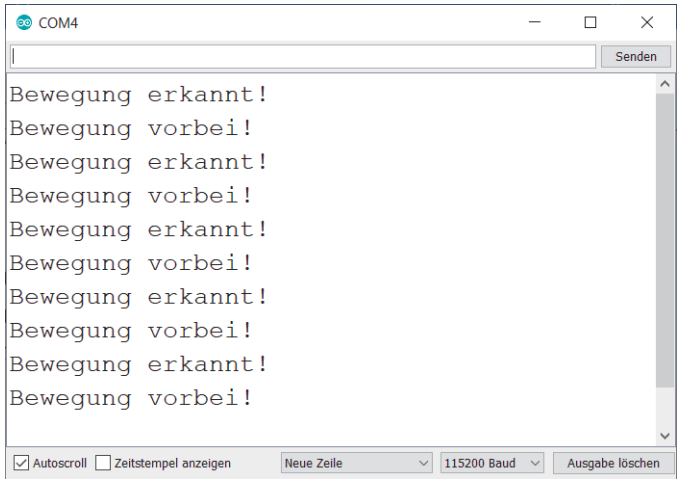

Abbildung 6.8 PIR-Sensor im seriellen Monitor

6.3 Analoge Aus-/Eingänge

Das System zur analogen IO des ESP32 präsentiert sich als mächtiges und vielfältig einsetzbares Werkzeug zur Steuerung externer Komponenten und zur Verarbeitung von Sensordaten. Da jedoch ein Mikrocontroller letztlich nur digital arbeitet, hat jedwede analoge Verarbeitung von Daten letztlich doch einen digitalen Hintergrund. Das Bindeglied oder besser die »Übersetzer« zwischen den beiden Welten sind entsprechende Konverter. Dabei handelt es sich um *Digital to Analog Converter* (DAC) und *Analog to Digital Converter* (ADC) (siehe Abschnitt 1.1).

Anders aber als bei der direkten Beschaltung von Pins als digitaler Aus-/Eingang ist eine explizite Festlegung des `pinMode` nicht erforderlich.

6.3.1 Analoge Ausgänge

Die beiden 8-Bit-DC ermöglichen es, die Ausgänge in Stufen von 0 bis 255 auf vorgegebene Spannungswerte zu setzen. Jede Stufe entspricht etwa 13 mV (also 3,3 V bei Stufe 255).

In der Arduino-IDE steht hierfür bezüglich des ESP32 die Funktion `dacWrite(pin, wert)` bereit. `pin` ist einer der DAC-Pins GPIO25 und GPIO26. `wert` ist die Angabe der Spannungsstufe zwischen 0 und 255.

Damit lässt sich der DAC zur Steuerung von LED, Motoren oder zur Erzeugung von Schwingungskurven einsetzen.

Anwendungsbeispiel

Ein simples Beispiel soll zeigen, wie ein Buzzer (siehe Abbildung 6.9), dessen I/O-Anschluss mit GPIO25 verbunden ist, rudimentäre Töne ausgibt, die mittels DAC erzeugt werden.

Abbildung 6.9 Buzzer

Der nachfolgende Sketch aus Listing 6.5 erzeugt Sinuskurven mit einer Frequenz von 1,25 kHz.

```
void setup() {
}
void loop() {
  for (int grad = 0; grad < 360; grad = grad + 8) {
     dacWrite (25, int (128 + 64 * sin (grad * PI / 180 )));
  }
}
```

Listing 6.5 Beispiel einer Sinuskurve mit DAC

Die Schrittweite bestimmt die Frequenz. Je Schritt erhöht sich die Frequenz dabei um 160 Hz (`grad + 8` = 160 Hz * 8 ~ 1,25 kHz).

Für eine Sägezahnkurve mit steil abfallender Flanke gilt:

```
for(int i=0;i<256;i++) {
  dacWrite(25,i);
}
```

Listing 6.6 Codeausschnitt für Sägezahnkurve mit steiler Flanke

Für eine Sägezahnkurve mit gleichen Flanken gilt:

```
for(int i=0;i<256;i++) {
  dacWrite(25,i);
}
for(int i=254;i>0;i--) {
  dacWrite(25,i);
}
```

Listing 6.7 Codeausschnitt für Sägezahnkurve mit gleichen Flanken

6.3.2 Analoge Eingänge

Analoge Eingangssignale wandelt der ADC in digitale Werte um. Die Auflösung der 18 Kanäle ist 12 Bit. Das bedeutet, dass die Eingangsspannung in dem Bereich zwischen 0 V und 3,3 V in 4.096 Einzelstufen (2^{12}) übertragen wird. Der niedrigste Wert ist 0, der höchste also 4.095.

Die entsprechende Arduino-Funktion heißt `analogRead(pin)`. Sie liest den Wert an dem analogen Pin `pin` ein, wandelt ihn um und gibt ihn aus. `analogReadResolution(resolution)` ermöglicht es, die Auflösung zwischen 9 Bits (0 bis 511) und 12 Bits (0 bis 4.095) einzustellen. Voreinstellung ist die 12-Bit-Auflösung. Darüber hinaus gibt es noch weitere Möglichkeiten der Parametrisierung (siehe *https://github.com/espressif/arduino-esp32/blob/master/cores/esp32/esp32-hal-adc.h*).

Eigentlich erwarten Sie an dieser Stelle einen linearen Verlauf von angelegter Spannung und ermitteltem Digitalwert. Dies ist allerdings trotz des recht guten DAC-ADC-Systems des ESP32 nicht der Fall. Sie können dies nachprüfen, indem Sie in einer Schleife die Ausgabe an einem DAC-Kanal stufenweise erhöhen und den ausgegebenen Wert an einem ADC-Eingang einlesen. In ein Diagramm übertragen, weicht die ADC-Kurve deutlich von der eigentlich zu erwartenden linearen Linie ab (siehe *https://ckblog2016.net/2018/03/03/esp32-adc-dac/*). Insofern eignet sich der ESP32 nicht für exakte Labormessungen; für die folgenden Beispiele ist dies aber zu vernachlässigen.

Poti

Ein *Poti* (Kurzform für *Potenziometer*) ist das ideale Bauteil, um die ADC-Funktionalität zu erproben und darüber hinaus einen Blick auf die Wirkungsweise eines Spannungsteilers zu werfen.

Der Begriff »Potenziometer« ist eine Kombination aus den Wörtern Potenzialdifferenz und Metering. Sie stammt aus den Anfängen der Elektronikentwicklung und beruht auf der Annahme, dass die jeweilige Einstellung bei den damaligen großen drahtgewickelten Widerstandsspulen eine bestimmte Menge an *Potenzialdifferenz* auslöst, deren Messung das Bauteil gewissermaßen zu einer Art Spannungsmessgerät macht. Heute sind Potis kleiner, das Wirkungsprinzip ist aber geblieben: Ein Kontakt (Schleifer) fährt über einen Widerstandskörper. Damit hat das Potenziometer drei Anschlüsse (je einen für die Endpunkte des Widerstandskörpers und einen für den Schleifkontakt). Der Gesamtwiderstand des Widerstandskörpers wird durch die Position des Kontakts geteilt. Liegt an diesen Endpunkten eine Spannung an, wirkt das Poti als *Spannungsteiler* (Abbildung 6.10).

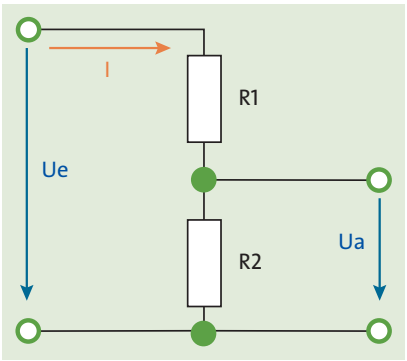

Abbildung 6.10 Spannungsteiler

Für Spannungsteiler wiederum gilt das ohmsche Gesetz. Danach ist der elektrische Widerstand (R) eines Werkstoffs der Quotient aus elektrischer Spannung (U) und elektrischem Strom (I), also $R = U / I$.

Sind Spannung und Widerstand bekannt, dann ergibt sich der elektrische Strom nach $I = U / R$, bei einem Spannungsteiler also: $I = Ue / (R1 + R2)$.

Nach den Regeln für Reihenschaltungen ist der Strom durch alle Bauteile identisch. Somit ergibt sich für Ua: $I = Ua / R2$.

Damit ist

$$\frac{Ua}{R2} = \frac{Ue}{(R1 + R2)}$$

oder

$$Ua = \frac{Ue}{(R1 + R2)} \times R2$$

Angenommen also, dass die Eingangsspannung 3,3 V beträgt, das Poti einen Gesamtwiderstand von 10 kΩ hat und der Schleifer (bei einem linearen Poti) in der Mitte steht, ergibt sich bezüglich der Ausgangsspannung:

$$Ua = \frac{3{,}3}{10000} \times 5000 = 1{,}65$$

Dies gilt allerdings nur für unbelastete Spannungsteiler. Da der Stromverbrauch durch einen an den Schleifer angeschlossenen Pin des ESP32 äußerst gering ist, kann dieser Effekt vernachlässigt werden.

Die Schaltung

Den Testaufbau zeigt die nachstehende Schaltung (siehe Abbildung 6.11):

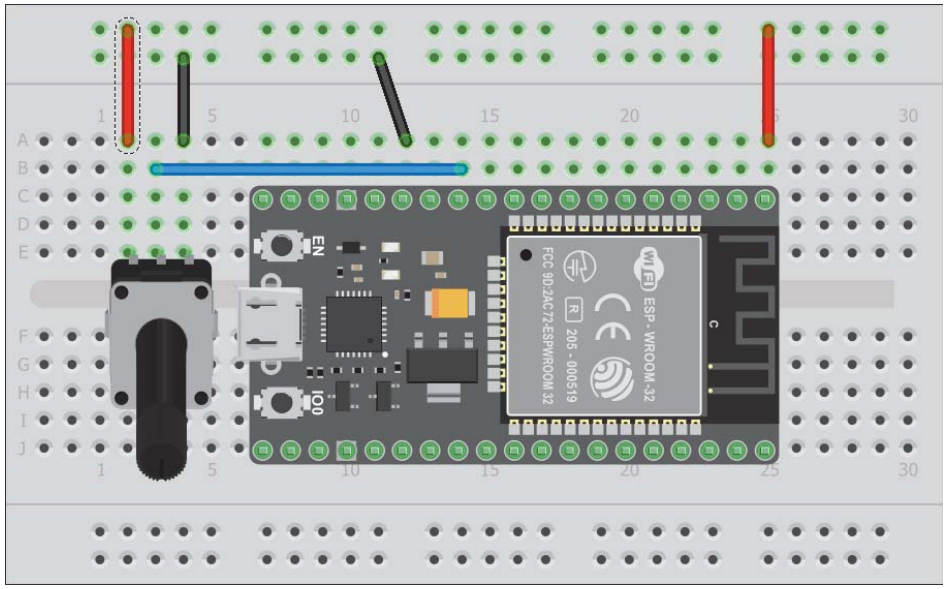

Abbildung 6.11 ADC-Umwandlung mit einem Poti

Der Sketch

Der Sketch aus Listing 6.8 ist so aufgebaut, dass er als Vorlage für weitere `analogRead()`-Auswertungen nutzbar ist. Deshalb sind das Lesen des Sensors, die Auswertung des Ergebnisses (z. B. Übersteigen eines Schwellenwertes) und sich daraus ergebende Aktionen in eine eigene Funktion ausgelagert.

```
#define ADC_PIN    14              // ADC2-7 an GPIO14
int schwellenWert = 3273;          // Schwellenwert 80 % von 4095
void lesenADC () {
  int anlgRead = analogRead(ADC_PIN);
  Serial.println("gelesener ADC-Wert: " + String (anlgRead) );
  if (anlgRead > schwellenWert) {
    Serial.println("Schwellenwert überschritten !");
  }
}
void setup() {
  Serial.begin(115200);
}
void loop() {
  lesenADC();
  delay(1000);
}
```

Listing 6.8 ADC mit Poti

Stellvorgänge des Potis beeinflussen die Eingangsspannung an Pin 14. Die Funktion `analogRead()` liest sie ein und gibt das digitale Äquivalent in die Variable `anlgRead` aus. Anschließend erfolgt eine Überprüfung des Schwellenwertes mit einer Hinweismeldung, wenn er überschritten wird (siehe Abbildung 6.12).

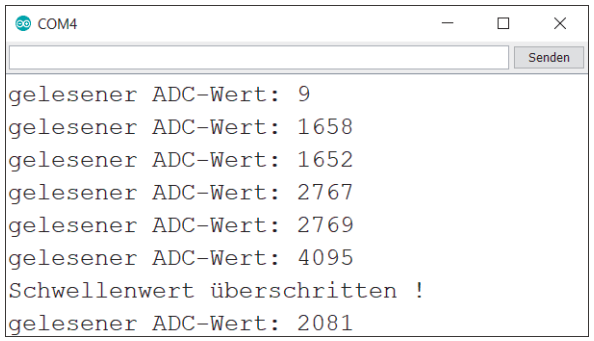

Abbildung 6.12 ADC – Poti-Ausgabe im seriellen Monitor

Fotowiderstand

Ein *Fotowiderstand* (LDR, *Light Dependent Resistor*) ist ein lichtempfindliches passives Bauelement (siehe Abbildung 6.13). Er ändert seinen Widerstand auf Basis des einfallenden Lichts. Bei Dunkelheit (Dunkelwiderstand) liegt der Widerstand über 1 MΩ; den Hellwiderstand von 100Ω bis 2 kΩ erreicht der LDR bei einer Beleuchtungsstärke von ca. 1.000 Lux.

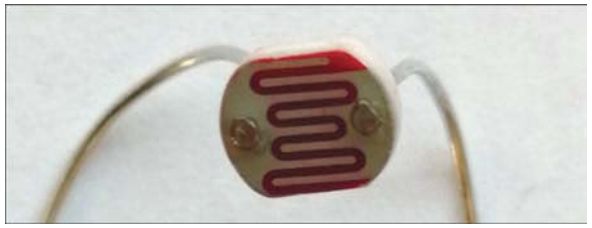

Abbildung 6.13 LDR (Fotowiderstand)

Damit eignet sich der LDR hervorragend als Sensor, um lichtgesteuerte Schaltvorgänge (z. B. als Dämmerungsschalter) auszulösen. Das Wirkprinzip ist das eines Spannungsteilers. Der LDR wird mit einem Widerstand (z. B. 10 kΩ) in Reihe geschaltet. Die Bauteilgruppe entspricht dann quasi einem Poti; der Schleifer ist dabei die Verbindung von LDR und Widerstand (siehe Abbildung 6.14).

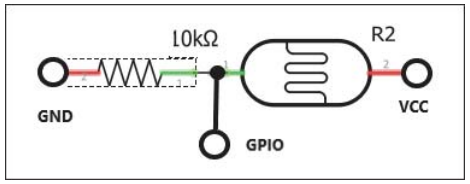

Abbildung 6.14 LDR als Spannungsteiler

Damit kann der Sketch aus dem vorherigen Beispiel übernommen werden. Je mehr Licht auf den LDR fällt, desto höher ist der Wert des `analogRead()`, und je dunkler es ist, desto geringer fällt der ermittelte ADC-Wert aus.

Heißleiter und Kaltleiter

Heißleiter (NTC, *Negative Temperature Coefficient Thermistor*) verändern mit der Umgebungstemperatur ihren elektrischen Widerstand. Sie weisen einen negativen Temperaturkoeffizienten auf, d. h., je höher die Temperatur ist, desto geringer ist der Widerstand. Bei 25 °C hat der NTC seinen Grundwiderstand (in diesem Fall 1 kΩ, siehe Abbildung 6.15).

Kaltleiter (PTC, *Positive Temperature Coefficient Thermistor*) weisen ein genau gegensätzliches Verhalten auf, sie leiten bei niedrigen Temperaturen den Strom besser.

Abbildung 6.15 Heißleiter

Heißleiter und Kaltleiter eignen sich daher für temperaturbezogene Steuerungsvorgänge. Zur Anwendung kommt auch hier das Wirkprinzip eines Spannungsteilers, sodass Schaltungsaufbau und Sketch aus dem vorherigen Beispiel in ihrer Grundstruktur übernommen werden können. Anpassungen ergeben sich bezüglich des Widerstands (1 kΩ anstelle von 10 kΩ führt bei einem Grundwiderstand des NTC von 1 kΩ und bei 25 °C zu einer Eingangsspannung von 1,65 V) und unter Umständen des Schwellenwertes.

Komfortabler und genauer in der Temperaturmessung sind jedoch komplette Module wie der DHT22 (siehe Abschnitt 6.5).

Ultraschallsensor

Ultraschall sind akustische oder mechanische Schwingungen oberhalb von 20 kHz. Die Funktionsweise eines Ultraschallsensors beruht darauf, dass ein Ultraschalllautsprecher Schallimpulse ausgibt. Treffen diese Impulse auf ein Objekt, sendet das Objekt ein Echo zurück, das von einem Mikrofon erfasst wird. Die gemessene Zeitspanne zwischen dem Senden des Signals und dem Empfang des Echos geht dann als ein Faktor in die Berechnung der Distanz ein.

Der andere Faktor ist die Schallgeschwindigkeit. Sie beträgt in der Luft 343,2 m/s bei 20 °C. Da die Temperatur die Ausbreitung des Schalls wesentlich beeinflusst, ist dies nur ein näherungsweiser Rechenwert; die exakte Formel ist: $c \approx 331{,}5 + (0{,}6\ \vartheta)$ in m/s (ϑ ist die Temperatur in Grad Celsius).

Das Einsatzfeld eines Ultraschallsensors ist vornehmlich die Messung von Entfernungen, z. B. als Füllstandswächter, zur Schließkontrolle von Türen und Fenstern, als Alarmgeber oder in der Robotik, wo es darum geht, Abstände eines Fahrzeugs zu einem Hindernis zu messen.

Das Modul HC-SR04 beinhaltet einen kompletten Ultraschallsensor (siehe Abbildung 6.16). Er erfasst bei einer Ruhestromaufnahme < 2 mA Distanzen von 2 cm bis 400 cm (Erfassungswinkel: ca. 45° im Nahbereich und < 15° bei maximaler Distanz). Neben den Anschlüssen für die Stromversorgung gibt es zwei weitere: je einen für das Senden (Trigger) und Empfangen (Echo) des Ultraschallsignals.

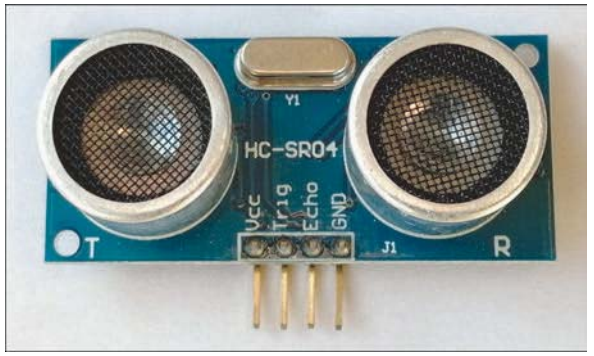

Abbildung 6.16 Ultraschallsensor HC-SR04

Die Schaltung

Die Schaltung besteht idealerweise aus zwei Bauteilen (siehe Abbildung 6.17).

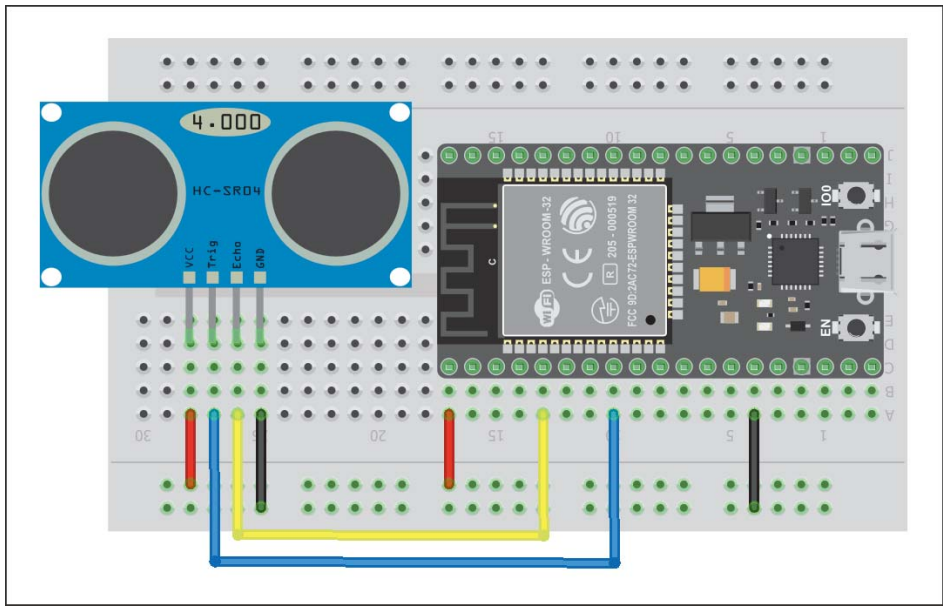

Abbildung 6.17 Schaltung des Ultraschallsensors

Versorgungsspannung HC-SR04

Weit verbreitet sind HC-SR04-Module, die eine Versorgungsspannung von 5 V benötigen. Dadurch hat auch das Ausgangssignal *Echo* einen Pegel von 5 V. Wenn Sie ein solches Modul einsetzen, ist zwingend ein Spannungsteiler (470 Ω / 1 kΩ) oder Level-Shifter vorzusehen. Das Modul kommt mit einem Trigger-Pegel von 3,3 V aus, sodass an dieser Stelle keine weiteren Maßnahmen notwendig sind.

Der Sketch

Der Sketch aus Listing 6.9 umfasst nur wenige Zeilen:

```
#define US_SENSOR_TRIG_PIN   33       // Trigger-Pin für den Ultraschallsensor
#define US_SENSOR_ECHO_PIN   34       // Echo-Pin für den Ultraschallsensor
float entfernung;                     // berechnete Entfernung
float dauer;                          // Zeitspanne des Echos

void lesenHCSR04() {
  digitalWrite(US_SENSOR_TRIG_PIN, LOW);   // definierten Zustand schaffen
  delayMicroseconds(5);                    // warten auf def. Pin-Zustand
  digitalWrite(US_SENSOR_TRIG_PIN, HIGH);  // Trigger 10 Mikrosekunden ein
  delayMicroseconds(10);
  digitalWrite(US_SENSOR_TRIG_PIN, LOW);
  dauer = pulseIn(US_SENSOR_ECHO_PIN, HIGH);    // Pulslänge messen,
                                                // Mikrosek.
  entfernung = dauer * 0.034 / 2;               // Entfernung berechnen
  Serial.println("Entfernung (cm): " + String (entfernung));
}

void setup() {
  pinMode(US_SENSOR_TRIG_PIN, OUTPUT);
  pinMode(US_SENSOR_ECHO_PIN, INPUT);
  Serial.begin(115200);
}

void loop() {
  lesenHCSR04();
  delay(2000);
}
```

Listing 6.9 Ultraschallsensor HC-SR04

Der Sketch ist recht simpel und inline recht gut kommentiert.

Eine kleine Anmerkung verdient, dass die gemessene Dauer noch durch 2 dividiert werden muss, da ja das Signal die tatsächliche Entfernung für Hin- und Rückweg zweimal durchläuft. Die Ausgabe im seriellen Monitor sieht dann wie in Abbildung 6.23 aus.

Abbildung 6.18 Ultraschallsensor HC-SR04 im seriellen Monitor

Feuchtigkeitssensor-Modul YL-69

Der *Bodenfeuchtesensor* oder das *Hygrometer* wird üblicherweise zur Erfassung der Bodenfeuchte verwendet. Die Funktionsweise des YL-69 besteht darin, mittels einer Sonde den Wassergehalt des Bodens durch Messen des Widerstands zu erkennen. Dies hat allerdings den Nachteil, dass der Stromfluss *elektrolytische Vorgänge* auslöst und diese mehr oder minder schnell zu einem Verschleiß der Messsonden (YL-38) führen. Aus diesem Grund werden diese teilweise mit Goldfinish veredelt oder aber nur dann mit Strom versorgt, wenn eine Messung durchzuführen ist.

Das Feuchtigkeitssensor-Modul YL-69 (siehe Abbildung 6.19) besitzt einen analogen (A0) und einen digitalen Ausgang (D0). Ein Potenziometer erlaubt eine Anpassung der Empfindlichkeit des digitalen Ausgangs (D0), zwei LEDs zeigen den Logikpegel des digitalen Ausgangs bzw. den Betriebszustand an.

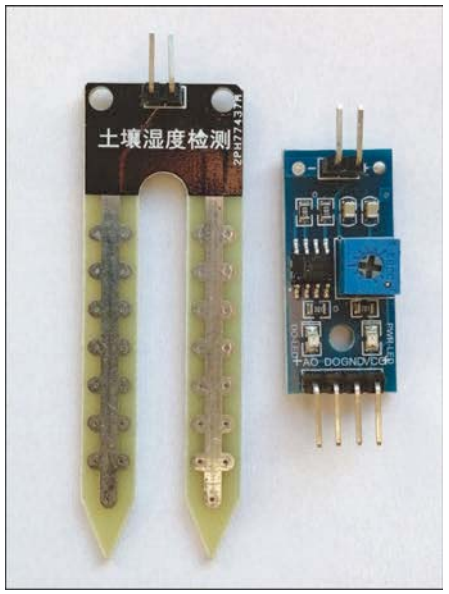

Abbildung 6.19 Bodenfeuchtesensor-Modul YL-69

Die Schaltung

In der Schaltung (siehe Abbildung 6.20) ist VCC des YL-69 mit GPIO34 verbunden, sodass eine Versorgung mit Strom softwaregesteuert auf Messvorgänge beschränkt werden kann.

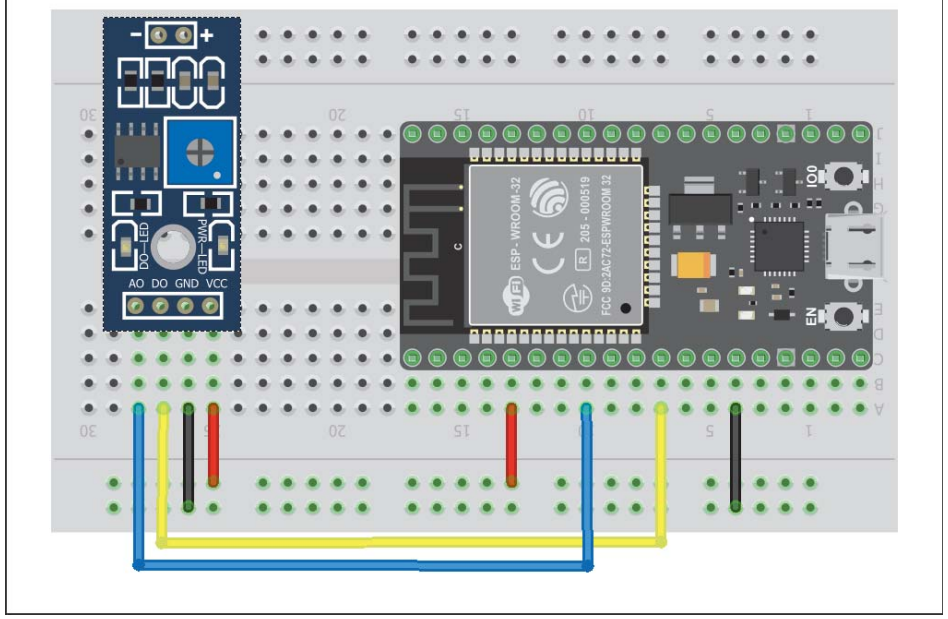

Abbildung 6.20 Schaltung des Bodenfeuchtemessers YL-69

Der Sketch

Der Sketch umfasst folgende Zeilen:

```
#define BODEN_SENSOR_A_PIN   33          // Pin für den Bodensensor analog
#define BODEN_SENSOR_D_PIN   27          // Pin für den Bodensensor digital
#define BODEN_SENSOR_VCC_PIN 34          // Pin für VCC Bodensensor

int schwellenWert = 50;                  // anzupassender Schwellenwert

void lesenYL69() {
  digitalWrite(BODEN_SENSOR_VCC_PIN, HIGH);
  delay (1000);
  int analogValue = analogRead(BODEN_SENSOR_A_PIN);
  int digitalValue = digitalRead(BODEN_SENSOR_D_PIN);
  //map Analogwert: 100% - sehr nass / 0% - sehr trocken
  int feuchtigkeit = map(analogValue, 0, 4095, 100, 0);
  Serial.print("Sensor (digital/analog): ");
```

```
  Serial.print(String(digitalValue) + "/");
  Serial.print(analogValue);
  Serial.println("  Feuchtigkeit in Prozent; " + String(feuchtigkeit) + "%");

  if (feuchtigkeit > schwellenWert) {
    Serial.print(" -> Keine Bewässerung nötig");
  } else {
    Serial.print(" -> Die Pflanzen brauchen Wasser");
  }
  if (digitalValue == LOW) {
    Serial.println(" - alles in Ordnung ");
  } else {
    Serial.println(" - Alarm! Bewässerung dringend");
  }
  digitalWrite(BODEN_SENSOR_VCC_PIN, LOW);
}

void setup() {
  pinMode(BODEN_SENSOR_VCC_PIN, OUTPUT);
  pinMode(BODEN_SENSOR_D_PIN, INPUT);
  digitalWrite(BODEN_SENSOR_D_PIN, LOW);
  Serial.begin(115200);
}

void loop() {
  lesenYL69();
  delay(5000);
}
```

Listing 6.10 Sketch Bodenfeuchtemessers YL-69

Die Hauptschleife ruft die Funktion `lesenYL69()` auf. Als Erstes erhält das Modul durch `digitalWrite(BODEN_SENSOR_VCC_PIN, HIGH)` Strom. Das anschließende `delay (1000)` sorgt für die nötige Zeit, einen Messvorgang durchzuführen. Die Messergebnisse gelangen durch `analogRead()` bzw. `digitalRead()` an den ESP32, wobei die `map()`-Anweisung den Analogwert (von 0 bis 4.095) in einen Prozentwert (100 % bis 0 % Feuchtigkeit) transformiert.

Nach erfolgreichem Hochladen erlauben Ausgaben im seriellen Monitor, das Geschehen zu verfolgen (siehe Abbildung 6.21).

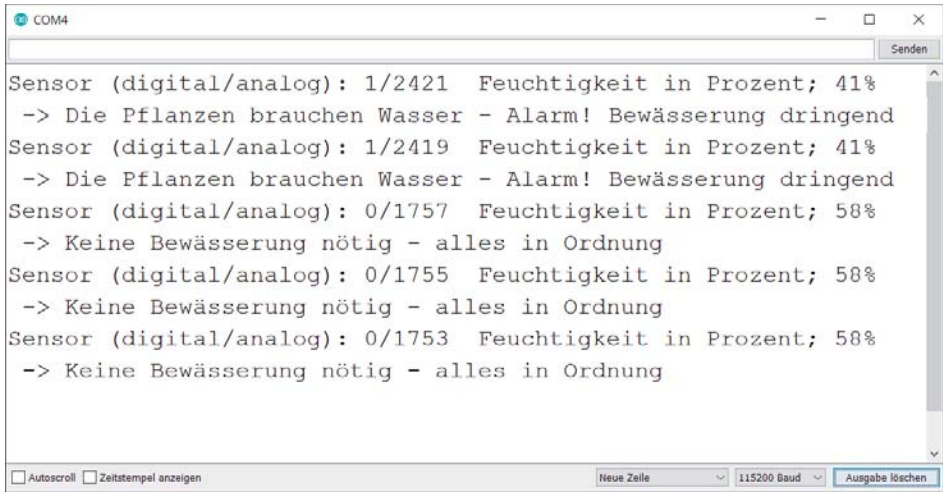

Abbildung 6.21 Bodenfeuchtemesser YL-69 im seriellen Monitor

Letztendlich bleibt noch, das System zu kalibrieren und an die Bodenverhältnisse anzupassen. Dies geht am besten mit dem zu überwachenden Boden, da verschiedene Bodenarten den Sensor beeinflussen können, mit Aufzeichnungen und ein wenig mit Trial und Error. Beim Betrieb im Freien empfiehlt es sich, die Sensorplatine vor Feuchtigkeit und Sonneneinstrahlung in einem Gehäuse zu schützen.

Kapitel 9
Projektideen

Die bisherigen Kapitel haben das Fundament für das Arbeiten mit dem ESP32 gelegt. Dieses Kapitel baut darauf auf, wandelt ab und ergänzt das Spektrum.

Die folgenden Abschnitte befassen sich mit ein paar Projektideen, die auch längeren Code enthalten – dieser kann tatsächlich ziemlich umfangreich werden und soll Ihnen als Vorlage für eigene Projekte dienen. Sie müssen aber natürlich nicht seitenweise Code abtippen! Alle Codebeispiele finden Sie unter *https://www.rheinwerk-verlag.de/5045/* zum Download.

Die Ideen, die Sie in diesem Kapitel finden, bauen auf dem auf, was Sie bisher kennen gelernt haben, beinhalten aber auch die eine oder andere Erweiterung bzw. Neuerung. Die Vorschläge verstehen sich als Anregung. Wandeln Sie sie ab, schneiden Sie sie auf Ihre persönlichen Erfordernisse zu, lassen Sie sich inspirieren und gehen Sie Ihren Weg.

Ich wünsche Ihnen viel Freude und Erfolg bei der Verwirklichung Ihrer Projekte!

9.1 The Evil Dice

Die kleine Sammlung von Projektideen soll mit einem Beispiel beginnen, das gut und gerne an einem verregneten Samstag umgesetzt werden kann: der »Evil Dice«, ein smarter Würfel.

Was der smarte Würfel leisten soll

Für eine reine Würfelschaltung ist der ESP32 eigentlich überdimensioniert, denn ein rollender elektronischer Würfel lässt sich bereits mit deutlich kleineren Mikroprozessoren wie etwa dem ATTINY45 bauen. Etwas Pfiff bekommt die Angelegenheit über verborgene Sonderfunktionen. So soll der Würfel alternativ über zwei nebeneinanderliegende Touch-Sensoren aktiviert werden können. Wird einer der beiden Sensoren berührt, ist das Würfelergebnis eine zufällige Zahl zwischen 1 und 6. Wird aber der linke Sensor nur kurz und dann mit einer Wischbewegung der rechte Sensor berührt, ist das Ergebnis eine 1 bzw. mit einer Wischbewegung von rechts nach links

eine 6. Eine solche Funktion kann ja bei spannenden Brettspielabenden im Familienkreis ganz nützlich sein ...

Die Schaltung

Für erste Gehversuche bietet es sich an, die Schaltung auf dem Steckbrett aufzubauen (siehe Abbildung 9.1).

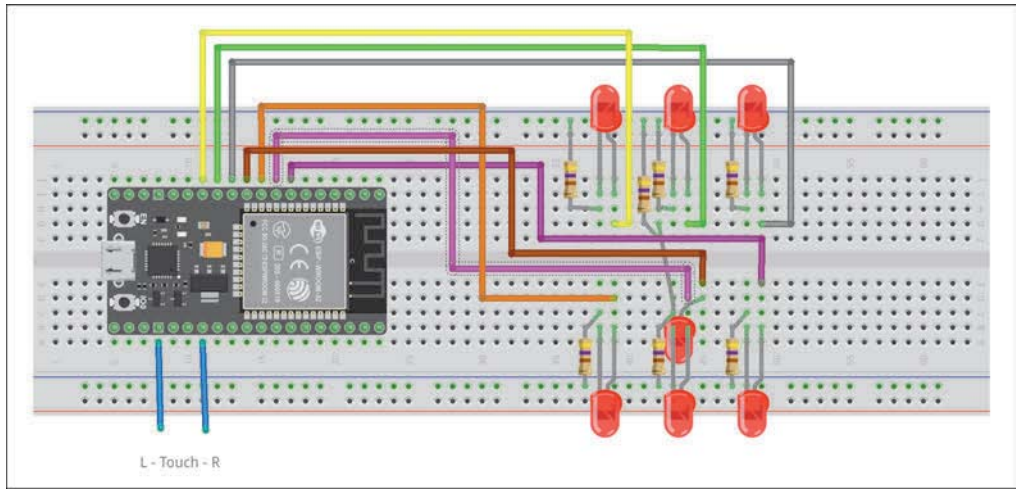

Abbildung 9.1 Schaltung für den smarten Würfel

Die Steckbrettschaltung können Sie dann in ein selbst gebasteltes Würfelgehäuse überführen.

Der Sketch

Der Sketch aus Listing 9.1 ist ziemlich umfangreich, die Komplexität ist aber noch überschaubar.

```
#define TOUTCH_PIN1 T3                  // ESP32 Pin D0
#define TOUTCH_PIN2 T0                  // ESP32 Pin D4
#define LED_PIN 2
#define LED_PIN1 12
#define LED_PIN2 14
#define LED_PIN3 27
#define LED_PIN4 25
#define LED_PIN5 33
#define LED_PIN6 32
#define LED_PIN7 26
```



```
unsigned long zeit1, zeit1Diff;         // Dauer Touch 1 berührt
unsigned long zeit2, zeit2Diff;         // Dauer Touch 2 berührt
unsigned long zeit3, zeit3Diff;         // Dauer Touch für Würfeln
int touchValue1, touchValue2;
int maxS = 30;

// LEDs ausschalten
void setLedLow() {
  digitalWrite (LED_PIN1, LOW);
  digitalWrite (LED_PIN2, LOW);
  digitalWrite (LED_PIN3, LOW);
  digitalWrite (LED_PIN4, LOW);
  digitalWrite (LED_PIN5, LOW);
  digitalWrite (LED_PIN6, LOW);
  digitalWrite (LED_PIN7, LOW);
  return;
}

// Ausgeben der gewürfelten Zahl
void displayZahl (int zahl) {
  switch (zahl) {
    case 1: digitalWrite (LED_PIN7, HIGH);
  switch (zahl) {
    case 1: digitalWrite (LED_PIN7, HIGH);
            break;
    case 2: digitalWrite (LED_PIN2, HIGH);
            digitalWrite (LED_PIN5, HIGH);
            break;
    case 3: digitalWrite (LED_PIN1, HIGH);
            digitalWrite (LED_PIN7, HIGH);
            digitalWrite (LED_PIN6, HIGH);
            break;
    case 4: digitalWrite (LED_PIN1, HIGH);
            digitalWrite (LED_PIN3, HIGH);
            digitalWrite (LED_PIN4, HIGH);
            digitalWrite (LED_PIN6, HIGH);
            break;
    case 5: digitalWrite (LED_PIN1, HIGH);
            digitalWrite (LED_PIN3, HIGH);
            digitalWrite (LED_PIN7, HIGH);
            digitalWrite (LED_PIN4, HIGH);
            digitalWrite (LED_PIN6, HIGH);
            break;
    case 6: digitalWrite (LED_PIN1, HIGH);
            digitalWrite (LED_PIN2, HIGH);
```

```
            digitalWrite (LED_PIN3, HIGH);
            digitalWrite (LED_PIN4, HIGH);
            digitalWrite (LED_PIN5, HIGH);
            digitalWrite (LED_PIN6, HIGH);
            break;
  }
  return;
}

void wuerfeln (int art) {
  int zahl;
  for (int i = 0; i < 30; i++) {        // Rollen des Würfels simulieren
    zahl = random (1, 6);
    displayZahl(zahl);
    delay(15 * i);                      // verlangsamen
    setLedLow();
  }
  if (art == 1)      zahl = 1;
  else if (art == 6) zahl = 6;
  displayZahl(zahl);
  delay (1000);
  setLedLow();
  return;
}

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(LED_PIN1, OUTPUT);
  pinMode(LED_PIN2, OUTPUT);
  pinMode(LED_PIN3, OUTPUT);
  pinMode(LED_PIN4, OUTPUT);
  pinMode(LED_PIN5, OUTPUT);
  pinMode(LED_PIN6, OUTPUT);
  pinMode(LED_PIN7, OUTPUT);
  digitalWrite (LED_PIN, LOW);
  setLedLow();
  zeit1 = micros();
  zeit2 = micros();
  zeit3 = micros();
}

void loop() {
  touchValue1 = touchRead(TOUTCH_PIN1);
```



```
touchValue2 = touchRead(TOUTCH_PIN2);

if (touchValue1 < maxS ||
    touchValue2 < maxS) {
  if (touchValue1 > 5 && touchValue1 < maxS) {
    zeit1Diff = micros() - zeit1;
  }
  if (touchValue2 > 5 && touchValue2 < maxS) {
    zeit2Diff = micros() - zeit2;
  }

  int antT1, antT2;
  zeit3Diff = micros() - zeit3;
  if (zeit3Diff > 500000) {
    antT1 = zeit1Diff * 100 / (zeit1Diff+zeit2Diff);  // %-Anteil
    antT2 = zeit2Diff * 100 / (zeit1Diff+zeit2Diff);  // %-Anteil
    if (antT1 > 75 || antT2 > 75) {   // nur ein Touch angetippt
      wuerfeln (0);
    } else
    if (antT1 < 35) {                 // zuerst Tip T1
      wuerfeln (1);
    } else
    if (antT2 < 35) {                 // zuerst Tip T2
      wuerfeln (6);
    } else {                          // Tip nicht eindeutig
      wuerfeln (0);
    }
    delay (1000);
    zeit1 = micros();
    zeit2 = micros();
    zeit3 = micros();
    zeit1Diff = 0;
    zeit2Diff = 0;
  }
  digitalWrite (LED_PIN, HIGH);
} else {
  zeit1 = micros();
  zeit2 = micros();
  zeit3 = micros();
  zeit1Diff = 0;
  zeit2Diff = 0;
  digitalWrite (LED_PIN, LOW);
```

```
  }
  delay(10);
}
```

Listing 9.1 The Evil Dice

Die allermeisten Programmbestandteile sind bereits in ähnlicher Form behandelt worden, sodass der Code keine dramatischen Neuerungen aufweist. Die eigentliche Schwierigkeit liegt darin, die Parameter der eigenen »Wischfertigkeit« anzupassen.

Die ersten Programmzeilen sind wie immer notwendigen `#includes`, `#defines` und Definition von globalen Variablen vorbehalten.

Für den Pegelwechsel aller LEDs auf LOW ist eine eigene Funktion `setLedLow()` vorgesehen, da dies von zwei Stellen des Programms nötig ist.

In der Funktion `displayZahl (int zahl)` werden die LEDs entsprechend dem Parameter `zahl`, der den Würfelwert enthält, über eine `switch`-Anweisung eingeschaltet.

Drei oder sieben Stränge?

Anstatt sieben Stränge für die LED-Steuerung zu verwenden, kann die gleiche Aufgabe auch durch nur drei Leitungen erfolgen. Der Stromverbrauch kann dann aber in Abhängigkeit von den verwendeten LEDs die maximal zulässige GPIO-Belastung übersteigen, sodass in der Folge eine Transistorschaltung (siehe Abschnitt 6.2) nötig wäre.

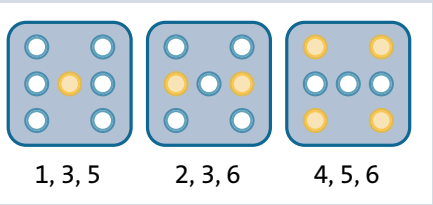

Abbildung 9.2 Smarter Würfel – Steuerung mit drei Leitungen

Die Funktion `wuerfeln (int art)` ermittelt das Ergebnis des Würfelvorgangs. Der Parameter `art` bestimmt, ob es sich um normales Würfeln oder einen »Sondervorgang« handelt. Sie können die Anzahl der Rollvorgänge (hier 30) und die Verzögerungszeit beim Auslaufen des Würfels (hier `delay(15 * i)`) Ihren Anforderungen entsprechend ändern.

In der Hauptschleife befindet sich die eigentliche Verarbeitungslogik. Zu Beginn werden die Touch-Sensoren ausgelesen. In Abhängigkeit davon, ob der in der Variablen `maxS` festgelegte Schwellenwert für eine Berührung unterschritten ist, erfolgt dann die weitere Auswertung.

Die Abfrage `if (zeit3Diff > 500000)` überprüft, ob der Touch-Vorgang insgesamt lang genug gedauert hat. Die Überprüfung erfolgt auf der Basis von Mikrosekunden (1.000 Mikrosekunden = 1 Millisekunde, 1.000.000 Mikrosekunden = 1 Sekunde; hier also 0,5 Sekunden).

Bei positivem Ergebnis berechnet der Sketch im Anschluss den jeweiligen prozentualen Anteil der Berührung von Touch1 und Touch2 an der gesamten Zeit des Touch-Vorgangs. Dieser wird als Nächstes ausgewertet und die Funktion `wuerfeln()` entsprechend parametrisiert aufgerufen.

Der Sketch wird ausgeführt

Wie der fertige Würfel genau aussieht, hängt natürlich von Ihrem Bastelgeschick ab. Dass die LEDs leuchten, wie sie sollen, zeigt Abbildung 9.3.

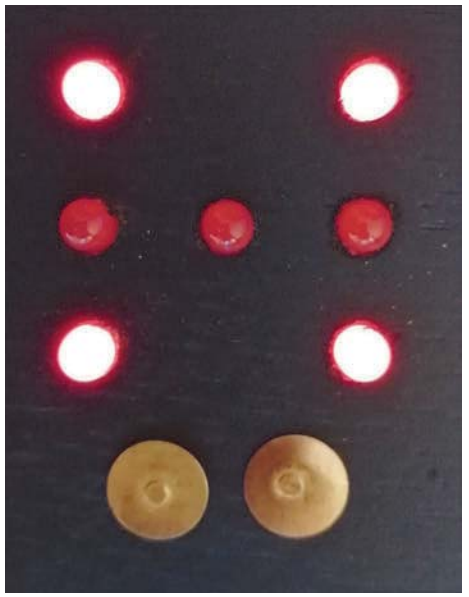

Abbildung 9.3 Ergebnis des smarten Würfels

Damit der Würfel auch seine Evil-Funktionen wunschgemäß ausführt, braucht es allerdings etwas Übung – die Funktion soll ja nicht zu einfach auszulösen sein. Passen Sie gegebenenfalls die Parameter an, und probieren Sie unterschiedliche Werte aus.

9.2 Die Maker-Uhr

Maker realisieren gerne eigene Lösungen auch für gängige alltägliche Güter. So liegt es nahe, diesen Ansatz auch auf eine individuelle Zeitanzeige, eine unkonventionelle

Digitaluhr, anzuwenden. In diesem Sinne soll der folgende Abschnitt eine Uhr zeigen, die die Zeit in binärer Form anzeigt. Bezugsquelle für die aktuelle Zeit soll das Web sein. Außerdem sollen gewisse Anzeigen-Einstellungen flexibel über eine App vorgenommen werden können.

Möglichkeiten einer Binäruhr

Bei einer Binäruhr wird die Uhrzeit mit binären Anzeigenelementen dargestellt. Dabei wird jede Dezimalziffer in das Dualsystem umgerechnet und angezeigt. Geläufig sind zwei Varianten: eine, in der jede Dezimalziffer (Stundenzehner, Stundeneiner usw.) in dualer Form dargestellt wird, und eine, in der die Darstellung ohne direkten Bezug zum Dezimalsystem als *BCD*-Codierung (*Binary Coded Decimal*) vorgenommen wird. Ein Beispiel dazu sehen Sie links in Abbildung 9.4; die rechte Darstellung zeigt eine einfache Umwandlung ins Binärsystem (Jede Stelle der Zahl hat den Wert der entsprechenden 2er-Potenz. Die der ersten Ziffer von rechts entsprechende Potenz ist $2^0 = 1$. Nehmen Sie jede Ziffer mal mit der entsprechenden Potenz und summieren Sie sie. Gehen Sie am besten von rechts nach links vor: Stunden-Zehner: 0001. Von rechts nach links bedeutet das $(1 \times 1) + (0 \times 2) + (0 \times 4) + (0 \times 8) = 1 + 0 + 0 + 0 = 1$).

Eine übersichtliche Darstellung der BCD-Codierung finden Sie in der Wikipedia: *https://de.wikipedia.org/wiki/BCD-Code*.

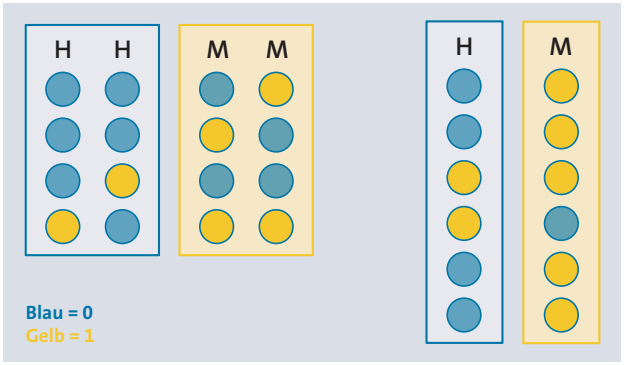

Abbildung 9.4 Prinzip einer Binäruhr

In beiden Fällen, die Sie in Abbildung 9.4 sehen, ist die Uhrzeit 12:59.

Die LED-Matrix

Für die Darstellung der einzelnen Binärwerte bietet sich eine LED-Matrix an. Diese Matrizes sind in den unterschiedlichsten Formaten erhältlich. Für dieses Beispiel kommt eine 4×5-Matrix zur Anwendung, die noch einen Bezug zum Dezimalsystem bietet (1. Variante). Die fünf Spalten sind für Stundenzehner, Stundeneiner, Minuten-

zehner, Minuteneiner und Sekunden vorgesehen. Die Sekunden sollen dabei in 15-Sekunden-Intervallen zusammengefasst werden, d. h., für Sekundenwerte 0–14 wird die unterste LED leuchten.

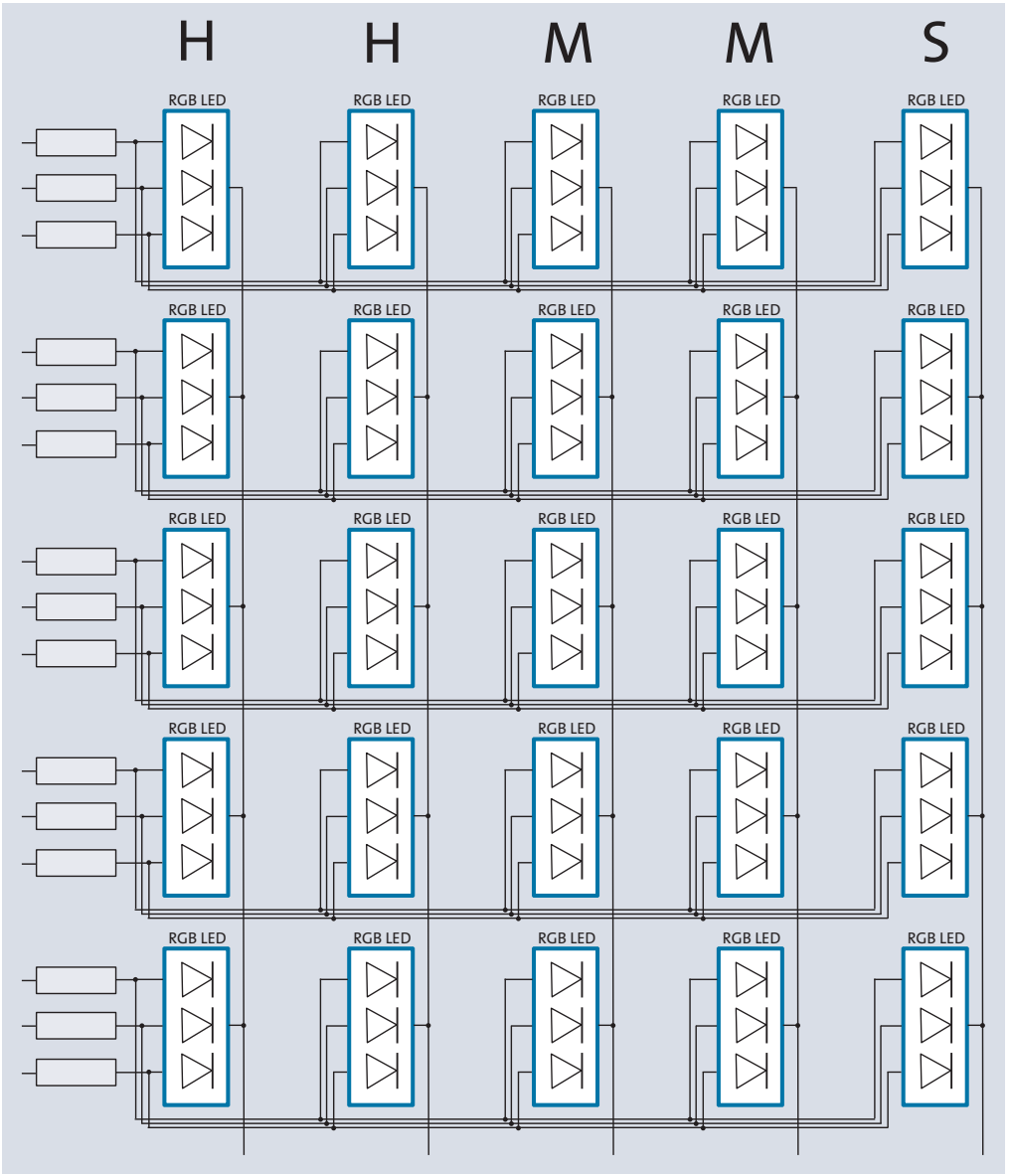

Abbildung 9.5 LED-Matrix – Schaltplan für eine Binäruhr

Die Matrix ist mit RGB-LEDs mit gemeinsamer Kathode realisiert. Dabei sind die Kathoden für Stundenzehner, Stundeneiner usw. miteinander verbunden und werden an einen GPIO-Pin des ESP32 geführt.

Miteinander verbunden sind auch die Anschlüsse für Rot, Grün und Blau der RGB-LED-Zeilen. Sie werden über einen Widerstand ebenfalls mit einem GPIO-Pin verbunden, wobei der Wert des Widerstands sich nach den Spezifikationen des Bauteils richtet. Jede Zeile repräsentiert einen Dualwert, unten beginnend mit 1, 2, 4 und 8.

Es bietet sich an, die LED-Matrix in Eigenentwicklung später als fertiges Bauteil auf einer Lochrasterplatine aufzubauen.

Auf der Oberseite der Lochrasterplatine werden die Verbindungen für die Kathoden geführt. Auf der Unterseite befinden sich die Stränge für die RGB-Anoden. Für den besseren Anschluss der Matrix an das ESP32-Dev-KitC V4 sind hinsichtlich der Verbindungen Header-Stiftleisten vorgesehen.

Abbildung 9.6 Bauteil für Binäruhr

Was die Binäruhr leisten soll

Das Programm für die Binäruhr muss vor allem Folgendes leisten:

- dauerhaftes Speichern der Einstellungen, die hinsichtlich der RGB-Farbauswahl getroffen wurden
- Aufbauen einer WLAN-Verbindung
- Besorgen der aktuellen Zeitangabe
- Aufbereiten und Erzeugen der Daten für die LED-Matrix; die Ausgabe selbst erfolgt dann über PWM

Der Code setzt sich also aus einigen Blöcken zusammen, die in ihren Grundzügen mit einer Ausnahme bereits angerissen wurden.

Neu und auf den ersten Blick etwas problematisch ist aber die aktuelle Zeitangabe. Anders als der PC oder ein Mikrocomputer verfügen Mikrocontroller von Haus aus nicht über die Möglichkeit, die aktuelle Zeit abzurufen. Da sich der ESP32 aber mit dem Internet verbinden kann, gibt es auch hierfür eine Lösung. Das Stichwort heißt *NTP* (*Network Time Protocol*).

NTP

Weltweit wird ein weitverzweigtes Netz von Rechnern betrieben, die die aktuelle Zeit auf der Basis von Atomuhren anbieten. Die Tiefe der Darstellung geht bis in den Nanosekundenbereich, der aber eher für wissenschaftliche Zwecke von Interesse ist. Die Rechner sind in einem Pool organisiert, sodass bei Ausfall eines Zugangs auf einen anderen ausgewichen werden kann. In der Bundesrepublik sind Anbieter beispielsweise die Physikalisch-Technische Bundesanstalt in Braunschweig (drei Zeitserver: *ptbtime1.ptb.de*, *ptbtime2.ptb.de*, *ptbtime3.ptb.de*), Universitäten (z. B. Regensburg mit der URL *ntp.ur.de*) und andere.

Hilfreiche Funktionen für den Zugriff auf einen NTP-Server über Arduino finden sich in der Bibliothek *NTPClient*, die Sie noch installieren müssen.

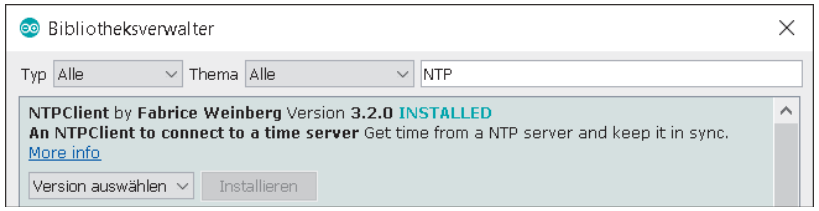

Abbildung 9.7 Bibliothek NTPClient einfügen

In der Bibliothek (Version: 3.2.0) enthaltene interessante Funktionen sind:

- `getFormattedTime()`: Gibt einen Wert vom Typ String zurück, der die Zeit im Format `hh:mm:ss` enthält.
- `getEpochTime()`: Gibt einen Wert vom Typ Unsigned Long Integer zurück, der die Zeit in Sekunden seit dem 01.01.1970 enthält.
- `getHours()`/`getMinutes()`/`getSeconds()`: Geben jeweils einen Integer-Wert mit der entsprechenden Zeitangabe zurück.

Der Sketch

Der Sketch aus Listing 9.2 ist aufgrund der Vielzahl der Aufgaben etwas umfangreicher. Zahlreiche Kommentare helfen Ihnen aber beim Verständnis.

```
#include <EEPROM.h>                       // Bibliothek Flash Memory
#include <WiFi.h>                         // für WLAN
#include <WiFiUdp.h>
#include <NTPClient.h>                    // für NTP

#define EEPROM_SIZE 15                    // Speichergröße festlegen

#define NTP_OFFSET 2 * 60 * 60            // in Sekunden
#define NTP_INTERVAL 60 * 1000            // in Millisekunden
#define NTP_ADDRESS "3.de.pool.ntp.org"   // "ptbtime1.ptb.de"

const char* ssid     = "dieRouterSSID";
const char* password = "dasRouterPW";
IPAddress lclIP (192,168,2,207);
IPAddress gateway (192,168,2,1);          // IP-Adresse des Routers
IPAddress subnet (255,255,255,0);
IPAddress primaryDNS (8, 8, 8, 8);        // optional
IPAddress secondaryDNS (8, 8, 4, 4);      // optional

WiFiServer serverWiFi(80);                // bietet Standard-Port 80
WiFiClient wifiClient;
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, NTP_ADDRESS, NTP_OFFSET, NTP_INTERVAL);

int eepromRGB [5][3];                     // Werte für [Zeitspalten]/[rgb]

// pins für die Spalten
int pinGND1 = 18;                         // GPIO21, Stundenzehner
int pinGND2 = 19;                         // GPIO1, Stundeneiner
int pinGND3 = 21;                         // GPIO1, Minutenzehner
int pinGND4 = 22;                         // GPIO22, Minuteneiner
int pinGND5 = 23;                         // GPIO23, Block Sekunden
int pinGNDar [5] = {pinGND1, pinGND2, pinGND3, pinGND4, pinGND5};
// Pins für die Zeilen
// 1. Dimension = 1. (unterste) Reihe;  2. Dimension:
// GPIO-Pin rot, grün, blau
int RGBLedAr [4][3] = {{15,2,0}, {13,12,14}, {27,26,25}, {33,32,5}};

int stunden;
int minuten;
int sekunden;
```



```
// Lesen des RGB-Farbstatus
void lesenEeprom() {
  // gesicherten Status aus dem Flash Memory auslesen
  int sp =0, rgb = 0, stat = 0;
  for (int i = 0; i < 15; i++) {
    stat = EEPROM.read(i);
    if (stat == 255) {
      eepromRGB [sp][rgb]= 1;
    } else {
      eepromRGB [sp][rgb]= 0;
    }
    rgb++;
    if (rgb > 2) {
      sp++;
      rgb = 0;
    }
  }
}

// Speichern des RGB-Farbstatus
// rx = gerade Zahl   => eeprom = 0   => aus
// rx = ungerade Zahl => eeprom = 255 => an
void speichernEeprom(int rx) {
  int i = rx / 2;
  int iRest = rx  % 2;
  if (iRest == 0) {
    EEPROM.write(i-1, 0);
  } else {
    EEPROM.write(i, 255);
  }
  EEPROM.commit();
  return;
}

// Holen der aktuellen Zeit
void holenNTP() {
  timeClient.update();                    // Aktualisieren des Zeitstempels
  stunden = timeClient.getHours();
  minuten = timeClient.getMinutes();
  sekunden = timeClient.getSeconds();
  return;
}
```

```
// Anzeige des Spaltenwertes
void displayWert(int w, int spalte){
  int r = eepromRGB [spalte][0];
  int g = eepromRGB [spalte][1];
  int b = eepromRGB [spalte][2];
  if (w & 1) {
    if (r) digitalWrite(RGBLedAr[0][0], HIGH);
    if (g) digitalWrite(RGBLedAr[0][1], HIGH);
    if (b) digitalWrite(RGBLedAr[0][2], HIGH);
    delay (2);
    digitalWrite(RGBLedAr[0][0], LOW);
    digitalWrite(RGBLedAr[0][1], LOW);
    digitalWrite(RGBLedAr[0][2], LOW);
  }
  if (w & 2) {
    if (r) digitalWrite(RGBLedAr[1][0], HIGH);
    if (g) digitalWrite(RGBLedAr[1][1], HIGH);
    if (b) digitalWrite(RGBLedAr[1][2], HIGH);
    delay (2);
    digitalWrite(RGBLedAr[1][0], LOW);
    digitalWrite(RGBLedAr[1][1], LOW);
    digitalWrite(RGBLedAr[1][2], LOW);
  }
  if (w & 4) {
    if (r) digitalWrite(RGBLedAr[2][0], HIGH);
    if (g) digitalWrite(RGBLedAr[2][1], HIGH);
    if (b) digitalWrite(RGBLedAr[2][2], HIGH);
    delay (2);
    digitalWrite(RGBLedAr[2][0], LOW);
    digitalWrite(RGBLedAr[2][1], LOW);
    digitalWrite(RGBLedAr[2][2], LOW);
  }
  if (w & 8) {
    if (r) digitalWrite(RGBLedAr[3][0], HIGH);
    if (g) digitalWrite(RGBLedAr[3][1], HIGH);
    if (b) digitalWrite(RGBLedAr[3][2], HIGH);
    delay (2);
    digitalWrite(RGBLedAr[3][0], LOW);
    digitalWrite(RGBLedAr[3][1], LOW);
    digitalWrite(RGBLedAr[3][2], LOW);
  }
  return;
}
```



```
// Ausgabe der Zeit
void displayZeit() {
  int s;
  switch (sekunden/15) {
    case 0: s = 1; break;
    case 1: s = 3; break;
    case 2: s = 7; break;
    default  : s = 15;
  }

  // Stundenzehner
  digitalWrite(pinGND1 , LOW);
  displayWert((stunden/10), 0);
  digitalWrite(pinGND1 , HIGH);
  // Stundeneiner
  digitalWrite(pinGND2 , LOW);
  displayWert((stunden%10), 1);
  digitalWrite(pinGND2 , HIGH);
  // Minutenzehner
  digitalWrite(pinGND3 , LOW);
  displayWert((minuten/10), 2);
  digitalWrite(pinGND3 , HIGH);
  // Minuteneiner
  digitalWrite(pinGND4 , LOW);
  displayWert((minuten%10), 3);
  digitalWrite(pinGND4 , HIGH);
  // Block Sekunden
  digitalWrite(pinGND5 , LOW);
  displayWert(s, 4);
  digitalWrite(pinGND5 , HIGH);
  return;
}

// sendet eine Antwort an den WLAN-Client
void wifiSend (WiFiClient client) {
  client.println("HTTP/1.1 200 OK");
  client.println("Content-type:text/html");
  client.println();
  // der Inhalt der HTTP-Antwort folgt auf den Header
  client.println();
}
```

```
// bearbeitet eine Anfrage von dem WLAN-Client
void wifiReceive (WiFiClient client) {
  //Serial.println("Neue Anfrage.");        // Ausgabe einer Meldung im
                                            // seriellen Monitor
  String currentLine = "";                  // String-Variable f.
                                            // eingehende Daten
  int cnt = 0;                              // Zähler while client.connected
  while (client.connected()) {              // while-Shleife, solange Client
    cnt++;                                  // Schleifenzähler um 1 erhöhen
    if (client.available()) {               // hat Client Bytes?
      char c = client.read();               // Lesen 1 Byte in Variable c
      // Serial.print(c);                   // Ausgabe des Bytes im SM
      if (c == '\n') {                      // ist Byte ein Newline-Zeichen?
        // das Ende der HTTP-Anfrage ist eine Blank-Zeile
        // und zwei Newline-Zeichen hintereinander
        if (currentLine.length() == 0) {    // Ende der HTTP-Anforderung
                                            // durch den Client
          wifiSend(client);
          break;                            // Beenden der while
                                            // client.connected-Schleife
        } else {                            // liegt ein Newline vor
          currentLine = "";                 // Variable currentLine löschen
        }
      } else if (c != '\r') {               // alles andere als ein
                                            // Wagenrücklauf-Zeichen
        currentLine += c;                   // Zeichen currentLine hinzufügen
      }
      // vollständiger Inhalt von currentLine z. B. GET /1 HTTP/1.1
      // war die Client-Anfrage "GET /" gefolgt von einer Zahl?
      if (currentLine[0] == 'G' &&
         currentLine.length() == 7 &&
         isDigit (currentLine[5] )) {
        int n = 0;
        String s1 = "";
        if (isDigit (currentLine[6] )) {    //2-stellig?
          s1 += currentLine[5];
          s1 += currentLine[6];
        } else {
          s1 += currentLine[5];
        }
        n = s1.toInt();
        Serial.print("n = ");
        Serial.println(n);
```



```
        speichernEeprom(n);
      }
    }                                       // end if client.available

    // erzwingt disconnect bei fehlendem client.available
    if (cnt > 10000) {
      break;
    }
  }                                         // end while client.connected
  client.stop();                            // Verbindung schließen
  //Serial.println("Client Disconnected.");
  //Serial.println();
}

void setup() {
  Serial.begin(115200);
  // EEPROM mit festgelegter Größe initialisieren
  EEPROM.begin(EEPROM_SIZE);
  // gesicherten Status aus dem Flash Memory auslesen
  lesenEeprom();

  // GPIO für die Spalten initialisieren
  for (int i = 0; i < 5; i++) {
    pinMode(pinGNDar [i], OUTPUT);
    digitalWrite(pinGNDar [i], HIGH);
  }

  // GPIO für die Zeilen initialisieren
  for (int i = 0; i < 4; i++) {
    pinMode(RGBLedAr[i][0], OUTPUT);
    digitalWrite(RGBLedAr[i][0], LOW);
    pinMode(RGBLedAr[i][1], OUTPUT);
    digitalWrite(RGBLedAr[i][1], LOW);
    pinMode(RGBLedAr[i][2], OUTPUT);
    digitalWrite(RGBLedAr[i][2], LOW);
  }

  if (!WiFi.config(lclIP, gateway, subnet, primaryDNS, secondaryDNS)) {
    Serial.println("STA failed to configure ");
  }
  Serial.print("Verbindungsaufbau zu ");    // Verbindungsaufbau
                                           // WLAN-Netzwerk
  Serial.println(ssid);
```

```
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi verbunden");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
  serverWiFi.begin();

  Serial.println("Start NTP Client!");
  timeClient.begin();
  holenNTP();
  Serial.println("setup beendet");
}

void loop() {
  WiFiClient client = serverWiFi.available();  // horcht auf Client-Anfragen
  if (client) {                                // fragt ein Client an?
    wifiReceive (client);                      // Anfrage aufbereiten
  }
  for (int i = 0; i < 20; i++) {
    displayZeit();
  }
  holenNTP();
  delay(1);                           // warten
}
```

Listing 9.2 Maker-Uhr

Zu Beginn stehen die `#includes`, die wie gehabt die notwendigen Bibliotheken einbinden.

```
#include <EEPROM.h>                   // Bibliothek Flash Memory
#include <WiFi.h>                     // für WLAN
#include <WiFiUdp.h>
#include <NTPClient.h>                // für NTP
```

Die EEPROM-Speichergröße wird auf 15 festgelegt, d. h. je ein Byte für jede Spalte und dort für jede RGB-Farbe (5 × 3).

```
#define EEPROM_SIZE 15                // Speichergröße festlegen
```


NTP verlangt noch einige Einstellungen. Basis für NTP ist die *GMT* (*Greenwich Mean Time*). Sie läuft der hiesigen Zeit (Berlin) um eine Stunde bzw. zwei Stunden hinterher, sodass die Basis um ein entsprechendes Offset zu erhöhen ist. Darüber hinaus werden ein Update-Intervall und der NTP-Time-Server festgelegt.

```
#define NTP_OFFSET 2 * 60 * 60              // in Sekunden
#define NTP_INTERVAL 60 * 1000              // in Millisesekunden
#define NTP_ADDRESS "3.de.pool.ntp.org"     // "ptbtime1.ptb.de"
```

Es folgen die Angaben für die WLAN-Verbindung. Zur Anwendung kommt an dieser Stelle eine feste IP-Adresse, damit diese in der Update-App für die Binäruhr-Einstellungen fest vorgegeben werden kann.

```
const char* ssid     = "dieRouterSSID";
const char* password = "dasRouterPW";
IPAddress lclIP (192,168,2,207);
IPAddress gateway (192,168,2,1);            //IP-Adresse des Routers
IPAddress subnet (255,255,255,0);
IPAddress primaryDNS (8, 8, 8, 8);          //optional
IPAddress secondaryDNS (8, 8, 4, 4);        //optional
```

Um die Funktionen der Bibliotheken nutzen zu können, sind einige Objekte nötig.

```
WiFiServer serverWiFi(80);                  // bietet Standard-Port 80
WiFiClient wifiClient;
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, NTP_ADDRESS, NTP_OFFSET, NTP_INTERVAL);
```

Danach werden einige globale Variablen, insbesondere für die Zuordnung der GPIO-Pins zu den Anschlüssen der LED-Matrix initialisiert. Passen Sie unter Umständen die GPIO-Belegung den eigenen Bedürfnissen an; dabei sollten Sie jedoch nicht aus den Augen verlieren, dass einige GPIOs nicht wahlfrei nutzbar sind!

Die Funktion `lesenEeprom()` liest die im EEPROM gespeicherten RGB-Farbwerte aus. Ist der eingelesene Wert 255, wird das korrespondierende Element der RGB-Farbtabelle auf 1 gesetzt, ansonsten auf 0. Aus den Einzelwerten der RGB-Farbtabelle (1 = an, 0 = aus) entscheidet sich im weiteren Programmverlauf, ob die entsprechende Farbe der Binäruhrspalte angezeigt wird oder nicht. Programmtechnisch sind hier sicherlich auch andere Lösungen denkbar, für hier soll das so genügen.

Die Funktion `speichernEeprom()` speichert den Status einer RGB-Farbe für eine Spalte im EEPROM. Status und zugehöriger Tabellenplatz (sprich RGB-Farbe/Spalte) ergeben sich aus dem übergebenen Parameter `rx`. Dieser kann den Wert von 1 bis 30 annehmen. In der Programmlogik verbirgt sich dahinter ein Zahlenpaar: Eine ungerade Zahl bedeutet »an«, eine gerade »aus« (z. B. 1 / 2: 1 Farbe Rot der RGB-LEDs in Spalte 1 »an«; 2 Farbe Rot der RGB-LEDs in Spalte 1 »aus«). Der Tabellenplatz der RGB-Farb-

tabelle und damit die Spalte der LED-Matrix wird durch eine Division durch 2 ermittelt.

```
int i = rx / 2;
```

Der Status berechnet sich mittels des Modulo-Operators. Ist der Rest der Division 0, also eine gerade Zahl, wird der Status auf 0, d. h. »aus« gesetzt.

```
int iRest = rx  % 2;
if (iRest == 0) {
    EEPROM.write(i-1, 0);
  } else {
    EEPROM.write(i, 255);
  }
```

Die Funktion `lesenEeprom()` aktualisiert dann die RGB-Farbtabelle, die die Grundlage für die Darstellung der RGB-Farben der Matrix darstellt.

Die Funktion `holenNTP()` holt den aktuellen Zeitstempel und weist den Variablen `stunden`, `minuten` und `sekunden` die neuen Werte zu.

Die Funktionen `displayZeit()` und `displayWert(int w, int spalte)` sind gewissermaßen die Kernfunktionen des Sketches. Sie sorgen dafür, dass die Zeit in binärer Form angezeigt wird, und steuern die LEDs der LED-Matrix an. Grundgedanke hierbei ist das Nutzen der PWM-Funktionalität, und zwar dergestalt, dass in einem Zyklus eine jede Farbe aller RGB-LEDs für eine kurze Zeit leuchtet, sofern der in der RGB-Farbtabelle gesetzte Status eine »1«, also »an«, aufweist. Damit eine Farbe einer RGB-LED aber angeschaltet wird, muss deren Anode auf logisch HIGH und die Kathode auf logisch LOW gesetzt werden.

Die Funktion `displayZeit()` wertet den in den Variablen `stunden`, `minuten` und `sekunden` gespeicherten Zeitstempel aus. Für jede Spalte (Stundenzehner, Stundeneiner usw.) ermittelt sie den anzuzeigenden Wert und ruft die Funktion `displayWert(int w, int spalte)` auf. Der Parameter `w` enthält den Wert (z. B. 5), der Parameter `spalte` die Spalte. Die Berechnung der Werte erfolgt durch einfache Division durch 10 für die Zehner und mit dem Modulo-Operator für die Einer. Vor dem Aufruf der Funktion wird die Spalte aktiviert, d. h., der GPIO, an den die Kathoden der RGB-LEDs in der Spalte angeschlossen sind, wird auf LOW gesetzt. Nach dem Aufruf wird der GPIO durch das Setzen auf HIGH wieder deaktiviert (z. B. für Stundenzehner).

```
// Stundenzehner
digitalWrite(pinGND1 , LOW);
displayWert((stunden/10), 0);
digitalWrite(pinGND1 , HIGH);
```

Die Ermittlung des Wertes für die Sekunden weicht von diesem Schema aber ab. Da die Anzeige der Sekunden in 15er-Blöcken erfolgen soll (0 bis 14 die unterste LED der

Sekundenspalte, 0 bis 30 die untersten zwei LEDs der Sekundenspalte usw.), muss der Wert anderweitig ermittelt werden. Dies geschieht durch eine Division durch 15; das Ergebnis ist der Sekundenbereich, in dem die Zeitangabe liegt. Anschließend wird das Ergebnis in eine Zahl umgesetzt, die die Funktion `displayWert(int w, int spalte)` auswerten kann.

```
int s;
  switch (sekunden/15) {
    case 0: s = 1; break;
    case 1: s = 3; break;
    case 2: s = 7; break;
    default  : s = 15;
  }
```

Die Funktion `displayWert(int w, int spalte)` schaltet nun die Anoden der RGB-LEDs auf `HIGH` (an) bzw. `LOW` (aus). Welche Spalte angesteuert werden soll, ergibt sich aus dem Parameter `spalte`. Die RGB-Farbtabelle liefert die entsprechenden Statuswerte für die Variablen.

```
int r = eepromRGB [spalte][0];
int g = eepromRGB [spalte][1];
int b = eepromRGB [spalte][2];
```

Danach unterliegt jede RGB-LED einer Spalte durch eine UND-Verknüpfung mit dem Parameter `w` einer Überprüfung, ob sie angeschaltet werden soll. Beispiel: Ist der Wert 3 (als binär 0011), führt eine UND-Verknüpfung mit 1 und 2 zu *true*, mit 4 und 8 jedoch zu *false*.

```
if (w & 1) {
```

true hat den neuen GPIO-Status GPIO `HIGH` zur Folge, sofern die Angaben der RGB-Farbtabelle dies so vorsehen.

```
if (r) digitalWrite(RGBLedAr[0][0], HIGH);
if (g) digitalWrite(RGBLedAr[0][1], HIGH);
if (b) digitalWrite(RGBLedAr[0][2], HIGH);
```

Das anschließende `delay(2)` sorgt dafür, dass der Schaltzustand für eine kurze Zeit erhalten bleibt, ehe er in den Ausgangszustand zurückfällt.

Die Funktion `wifiSend (WiFiClient client)` gibt bei einer HTML-Anfrage lediglich eine leere Antwort.

Die Funktion `wifiReceive (WiFiClient client)` greift bei einem über das Netz angestoßenen Wechsel der RGB-Farbtabelle. Wesentliche Aufgabe ist, die maßgebliche Zeile mit dem neuen Status herauszufiltern und den Status selbst zu isolieren. Anknüpfungspunkt ist eine Zeile, deren vollständiger Inhalt z. B. `GET /1 HTTP/1.1` ist. An

Position 5 beginnt der verschlüsselte Wert für den Status. Er kann einstellig oder zweistellig sein. Mit der String-Object-Funktion `s1.toInt()` wird der Wert in eine Integer-Variable überführt und mit der Funktion `speichernEeprom(n)` gespeichert.

```
if (currentLine[0] == 'G' &&
    currentLine.length() == 7 &&
    isDigit (currentLine[5] )) {
  int n = 0;
  String s1 = "";
  if (isDigit (currentLine[6] )) {    //2-stellig?
     s1 += currentLine[5];
     s1 += currentLine[6];
  } else {
     s1 += currentLine[5];
  }
  n = s1.toInt();
  Serial.print("n = ");
  Serial.println(n);
  speichernEeprom(n);
}
```

Die Funktion `setup()` weist prinzipiell mit einer kleinen Ausnahme keine Besonderheiten auf: Die Kathoden-GPIOs der LED-Matrix werden auf `HIGH`, die Anoden-GPIOs auf `LOW` gesetzt; so ist gewährleistet, dass alle LEDs initial ausgeschaltet sind.

Auch die `loop()`-Funktion ist wenig spektakulär. Einstellungsmöglichkeiten ergeben sich hinsichtlich der Anzahl der Schleifendurchläufe für die Zeitausgabe (hier 20).

```
for (int i = 0; i < 20; i++) {
  displayZeit();
}
```

Das `delay(1)` am Ende erlaubt dem ESP32, Selbstverwaltungsprozesse durchzuführen. Ob dies notwendig ist, kann experimentell verifiziert werden.

Der Sketch beginnt nach dem Hochladen mit ein paar Ausgaben im seriellen Monitor.

Beim ersten Start sollten alle EEPROM-Bytes den Wert 255 haben, sodass alle eingeschalteten RGB-LEDs einer Spalte weiß erscheinen.

Danach ist die aktuelle Uhrzeit 13:42 und 45–59 Sekunden.

Nun können über eine Browsereingabe die RGB-Farben für die einzelnen Spalten aus- oder wieder eingeschaltet werden (z. B. *http://192.168.2.207/4* für grüne Stundenzehner aus).

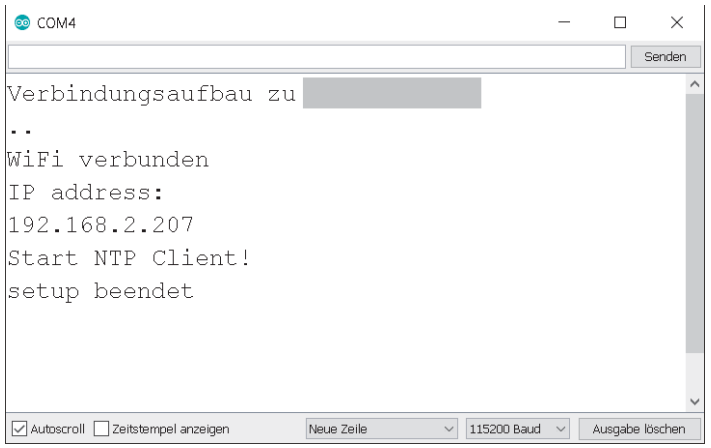

Abbildung 9.8 Startmeldungen der Binäruhr im seriellen Monitor

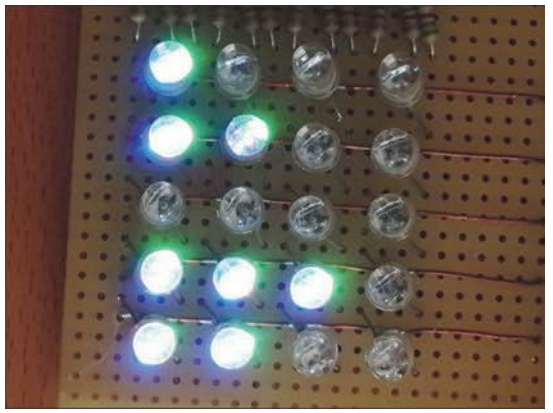

Abbildung 9.9 Binäruhr-Anzeige

Abbildung 9.10 Binäruhr-Anzeige mit geänderten RGB-Farben

RGB-Farben über eine App festlegen

Sie können die Farbeinstellungen auch smarter über eine App bestimmen. Da die Anzahl der kostenfreien Widgets bei Blynk beschränkt ist, kommt hierfür eher der MIT App Inventor in Frage. Als Tool für den Verbindungsaufbau scheidet allerdings Bluetooth aus, da die Funktionen für den ESP32 mehr Programmspeicherplatz benötigen, als neben dem bereits existierenden Code noch zur Verfügung steht. Das stellt jedoch kein Problem dar, weil der Baukasten für die App auch über ein Tool verfügt, das eine Verbindung mittels WLAN ermöglicht.

Die App selbst zu erstellen, ist eher weniger schwierig, sondern mehr eine zeitaufwändige Fleißarbeit. Dabei sind auch mehrere Lösungen denkbar, insbesondere ob nun das Schalten der Farben über einen Schaltknopf oder einen Schalter realisiert wird. Sinnvollerweise werden diese in einer Tabelle mit entsprechenden Überschriften (hier für die Farben Rot, Grün und Blau) sowie Spaltenbezeichnungen (hier für die einzelnen Spalten der RGB-Matrix) angeordnet.

Aus optischen Gründen könnte das erste Element eine Komponente *Label* aus der Gruppe *User Interface* sein. Für die Tabelle wird die Komponente *TableArrangement* aus der Gruppe *Layout* benötigt. Die Eigenschaften werden mit sechs Spalten (vier Spalten mit Werten und je eine Trennspalte zwischen Rot/Grün sowie Grün/Blau) und sechs Reihen spezifiziert.

Abbildung 9.11 TableArrangement für die Binäruhr im MIT App Inventor

Als Nächstes gilt es, die einzelnen Tabellenfelder zu füllen. Für die Spaltenbezeichnungen sind das fünf *Label*-Komponenten, drei für die Farbbezeichnungen und zwei für die Zwischenräume zwischen den »Farbspalten«. Die Eigenschaften müssen aber noch angepasst werden:

- »Farbspalten«:
  - Width = 10 percent
  - Text = Rot / Grün / Blau
  - TextAlignment = center
- Zwischenräume:
  - Width = 5 percent
  - Text löschen
  - TextAlignment = center

Für die Reihenbezeichnungen sind ebenfalls fünf *Label*-Komponenten nötig. Die Eigenschaften sind Width = 30 percent und Text = Stundenzehner usw.

Die Schaltvorgänge sollen an dieser Stelle mit Schaltern erfolgen. Benötigt wird also fünfzehnmal die Komponente *switch* aus der Gruppe *User Interface*. Es empfiehlt sich, den einzelnen Komponenten sprechende Namen zu geben (so z. B. für den Schalter, der die Farbe Rot für die Stundenzehner bestimmt, hzR). Bei den Eigenschaften wird Width auf »10 percent« gesetzt und der Inhalt des Textfeldes gelöscht.

Wer mag, kann noch die Eigenschaft TrackColourActive in Rot, Grün oder Blau ändern, sodass der aktivierte Schalter auch farblich ansprechend dargestellt wird.

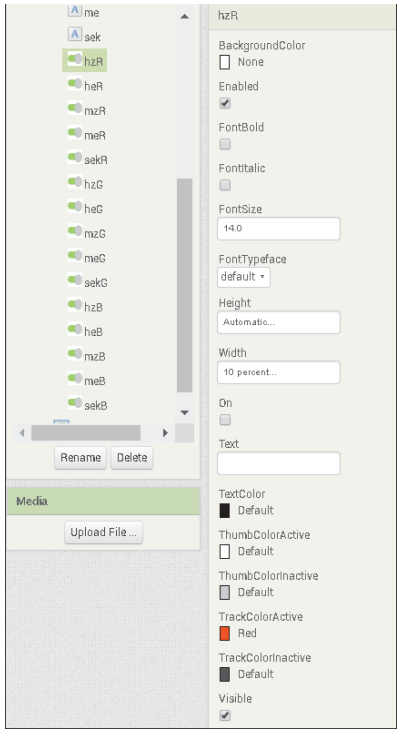

Abbildung 9.12 switch-Komponente im MIT App Inventor

Übertragen Sie schließlich noch die Komponente *WebViewer* aus der Gruppe *User Interface* in den Viewer.

Abbildung 9.13 App im Viewer

Wechseln Sie nun in die Blocks-Ansicht. Ziehen Sie auch hier die benötigten Blöcke aus der Blockliste in den Viewer, und setzen Sie diese dort zusammen. Erstellen Sie dabei für jede *switch*-Komponente eine Schaltstruktur.

Bestimmendes Element ist der HTTP-Request. Er enthält die IP-Adresse des ESP32, die in dem Sketch festgelegt wurde (hier `192.168.2.207`), und den Inhalt der Anfrage (hier `1` für »ein« bzw. `2` für »aus«), getrennt durch ein `/`. Eine Schritt-für-Schritt-Anleitung können Sie aus den folgenden Abbildungen ableiten, die den Aufbau der Struktur hinreichend verdeutlichen.

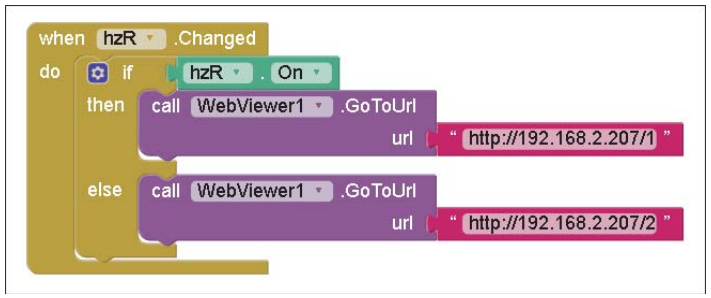

Abbildung 9.14 Blocks im MIT App Inventor