LESEPROBE

Eigene Zigbee-Geräte mit ESP32 entwickeln

Praxisüberblick mit Hands-on-Beispielen für ESP32 Arduino Core, Home Assistant und Zigbee2MQTT

Markus Edenhauser, MA MSc

Juli 2026
1. Auflage
Selbstverlag

Markus Edenhauser, MA MSc
Völser Straße 41
6020 Innsbruck
Österreich
pixeledi.eu/impressum

Du bist mein schönstes `Zigbee`-Signal. Mit dir bleibt selbst das wildeste Mesh im Leben stabil.

Inhaltsverzeichnis

0 Einführung **6**
0.1 Für wen dieses Buch gedacht ist . . . 6
0.2 Was du in diesem Buch lernst . . . 6
0.3 Verwendete Hardware . . . 7
0.4 Arduino IDE und PlatformIO . . . 8
0.5 Backend und Voraussetzungen . . . 8
0.6 Aufbau des Buches . . . 8
0.7 Codebeispiele und GitHub Repository . . . 9
0.8 KI Transparenz Hinweis . . . 9
0.9 Lizenz und Haftung . . . 10

1 Zigbee Grundlagen kompakt **11**
1.1 Was ist Zigbee? . . . 11
1.2 Was die Zigbee-Spezifikation technisch festlegt . . . 12
1.3 Zigbee im Smart Home . . . 15
1.4 Unterschiede zu WLAN und Bluetooth . . . 16
1.5 Coordinator, Router und End Device . . . 16
1.6 Das Zigbee Mesh Netzwerk . . . 17
1.7 Endpoints, Cluster, Attribute Einsteigerblick . . . 17
1.8 Zigbee2MQTT und Home Assistant Überblick . . . 18
1.9 Typische Zigbee Probleme und Ursachen . . . 18
1.10 Abgrenzung zu Matter over WiFi und Matter over Thread . . . 19
1.11 Warum es heute zwei Smart-Home-Standards gibt . . . 20
1.12 Zukunft von Zigbee und warum es weiterhin wichtig ist . . . 21

2 ESP32 Zigbee-Varianten und Boardauswahl **22**
2.1 ESP32-Zigbee-SoCs im Überblick . . . 22
2.2 Welche Variante passt zu welchem Projekt? . . . 23
2.3 Entwicklungsboards im Vergleich z. B. XIAO, DevKit . . . 24
2.4 Warum ich in diesem Buch oft XIAO verwende . . . 24
2.5 LiPo-Praxis: Laden vorhanden, Schutzfunktionen prüfen . . . 25
2.6 Antenne, Gehäuse und Einbauort . . . 25

3 Zigbee-Grundeinstellungen in Arduino IDE und PlatformIO **27**
3.1 Warum dieses Kapitel wichtig ist . . . 27
3.2 Grundprinzip: Zigbee wird über Build-Kontext aktiviert . . . 28
3.3 Arduino IDE: welche Zigbee-Einstellungen wirklich zählen . . . 28
3.4 PlatformIO: Referenzkonfiguration für XIAO ESP32-C6 . . . 29
3.5 End Device vs. Router: technische Folgen der Rollenwahl . . . 30
3.6 Partitionierung: warum `zigbee.csv` so wichtig ist . . . 30
3.7 Loglevel und Diagnosetiefe . . . 30
3.8 Serial, Build-Flags und Rollen-Guards zusammen denken . . . 31
3.9 Typische Konfigurationsfehler und schnelle Gegenprüfung . . . 32
3.10 Vor-dem-ersten-Flash Checkliste . . . 32

4 Zigbee2MQTT Setup und Home Assistant Integration 33
4.1 Voraussetzungen Home Assistant OS 33
4.2 Zigbee Adapter auswählen und einbinden 33
4.3 MQTT Mosquitto Broker installieren und prüfen 35
4.4 MQTT Benutzer und Zugriff konfigurieren 36
4.5 Zigbee2MQTT installieren und starten 36
4.6 Grundkonfiguration prüfen 37
4.7 Weboberfläche und Logs verstehen 38
4.8 Home Assistant MQTT Discovery aktivieren 38
4.9 Erster End-to-End Test mit einem Gerät 39
4.10 Typische Setup-Probleme und schnelle Diagnose 40
4.11 Optional Node-RED installieren und anbinden 40

5 CPU-Temperatur als erster Messwert 42
5.1 Zigbee-spezifische Einstellungen 42
5.2 Flash und Verbindung mit Zigbee2MQTT 43
5.3 Vollständiger Beispielcode 44
5.4 Technischer Ablauf im Code 45

6 Projekt Temperatur-Sensor 47
6.1 Hardware und Verdrahtung I2C 47
6.2 SHT3x auslesen nur Temperatur 48
6.3 Erweiterung auf Temperatur und Luftfeuchte 50
6.4 Darstellung in Zigbee2MQTT und Home Assistant 52
6.5 Reporting, Plausibilisierung und Messintervalle 52

7 SHT3x als Battery End Device 53
7.1 Stromverbrauch verstehen 53
7.2 Deep Sleep und Wakeup-Grundlagen 54
7.3 Umbau vom Netzgerät zum Sleepy End Device 54
7.4 Setup-Ablauf und Zigbee-Details 60
7.5 Flash-Hinweis bei aktivem Deep Sleep 60
7.6 Batterielaufzeit messen und gezielt optimieren mit PPK2 60
7.7 Typische Sleep-Probleme und Diagnose 61
7.8 Fazit 62

8 XIAO-log SHT40, BH1750 und Zigbee-Endgerät 63
8.1 Warum XIAO-log in diesem Kapitel 64
8.2 Zigbee-Rolle und Projektziel 64
8.3 Aufbau des Sketches 64
8.4 PlatformIO-Konfiguration 64
8.5 XIAO-log-spezifische Codebeschreibung 65
8.6 Sensorversorgung und I2C-Robustheit 66
8.7 SHT40- und BH1750-Behandlung 67
8.8 Reporting mit Bestätigung und Retry 67
8.9 Keep-Alive und Deep Sleep 68
8.10 Pairing und Verifikation 69

8.11 Einordnung im Projektverlauf 70

9 Projekt On/Off Switch Button 71
9.1 Hardware-Aufbau mit Taster 72
9.2 Button-Events über Zigbee senden 72
9.3 Schalten in Zigbee2MQTT und Home Assistant 73
9.4 Entprellung, kurze/lange Klicks 73
9.5 Variante mit Default-Response-Callback 73
9.6 Typische Fehler und schnelle Diagnose 74

10 SCD41 CO2-Sensor (NDIR) 75
10.1 Ziel des ersten Teilkapitels 76
10.2 PlatformIO-Konfiguration 76
10.3 Codeablauf 77
10.4 Erwartete Verifikation im Labor 81
10.5 SCD41 im Zigbee-Gerätemodus 81

11 DIY Türkontakt mit IAS Zone 86
11.1 Ziel des Projekts 91
11.2 Verdrahtung und Eingangslogik 91
11.3 Warum IAS Zone und was Zone hier bedeutet 92
11.4 Startablauf und Enrollment-Logik 92
11.5 Kernfunktion: Zone-Status setzen und melden 92
11.6 Warum in Zigbee2MQTT oft zwei Alarm-Flags erscheinen 93
11.7 Tamper-Logik 93
11.8 Laufzeitverhalten im loop() 94
11.9 Diagnose und typische Stolperstellen 95

12 1-Kanal-Relaismodul 230V mit Zigbee schalten 96
12.1 Nur für Fachpersonal 96
12.2 Hardwareeinordnung 96
12.3 Projektziel und Zigbee-Rolle 98
12.4 Codeaufbau: die wichtigen Bausteine 98
12.5 Verdrahtungshinweise 100
12.6 Typische Use Cases 100
12.7 Ausblick DIY-Variante mit Strommessung 100

13 CodeCell C6 Drive: Bewegungsdetektor mit Zigbee 101
13.1 Praktische Einordnung 102
13.2 Quellcode 102
13.3 Plattform und Build-Konfiguration 102
13.4 Zielbild des Geräts 104
13.5 CodeCell initialisieren 104
13.6 Zigbee-Gerätemodell für Bewegung 105
13.7 Endpoint registrieren und Zigbee starten 105
13.8 Loop 106
13.9 Nur bei Zustandswechsel senden 106

13.10LED als lokale Rückmeldung 107
13.11Ausblick 107

14 Zigbee-Ausblick: Nächste Gerätetypen mit ESP32 108
14.1 Warum dieser Fokus dich für eigene Umsetzungen fit macht . . . 108
14.2 Offizielle Beispielsammlung als nächster Schritt 108
14.3 Was davon neu ist und was nur eine Variation 108
14.4 Praktische Reihenfolge für eigene Erweiterungen 109

15 Maker Playbook: Tipps für stabile Zigbee-Projekte 110
15.1 10-Minuten-Healthcheck vor dem Bastelabend 110
15.2 Vor-dem-Flash Checkliste 110
15.3 24h-Stabilitätstest in drei Schritten 110
15.4 Vor-dem-Einbau Checkliste 110
15.5 Namens- und Strukturregeln, die später Zeit sparen 111
15.6 Fünf typische No-Gos im Maker-Alltag 111
15.7 Nächste sinnvolle Ausbauschritte 111

16 Troubleshooting 112
16.1 Warum zeigt Zigbee2MQTT nach einem Flash noch alte Exposes? 112
16.2 Wann ist ein kompletter Flash-Reset sinnvoll? 112
16.3 Warum pairt das Gerät, sendet aber keine Werte? 112
16.4 Warum verbindet ein ESP32 gar nicht oder nur instabil? 112
16.5 Warum verliert das Gerät im Betrieb die Verbindung? 113
16.6 Warum sind Sleep-Endgeräte oft als offline markiert? 113
16.7 Warum ist die Funkqualität schlecht? 113
16.8 Warum klappt Pairing nur einmal? 114
16.9 Warum ist das Mesh trotz vieler Geräte instabil? 114
16.10Warum schlägt der Build in `platformio.ini` fehl? 114

17 Abschließende Worte 115
17.1 Downloadlinks 115
17.2 Über den Autor 116
17.3 Weiteres vom Autor 116
17.4 Stichwortverzeichnis 118

Abbildungsverzeichnis

1 Rollenmodell im Netzwerk 16
2 Mesh mit Alternativroute 17
3 Datenpfad Zigbee2MQTT mit Home Assistant und Node-RED . 18
4 Zigbee und Matter Abgrenzung 19
5 XIAO ESP32-C6 Antennenkonfiguration 23
6 Build und Flash Schrittablauf 27
7 Debugging Problemklassen Build Flash Join Laufzeit 32
8 SONOFF Zigbee Adapter 34
9 Setup Überblick Infrastruktur 34
10 Zigbee2MQTT Home Assistant Integration 39
11 Erster End to End Test mit einem Gerät 40
12 SHT3x Sensor 47
13 Battery End Device Aufbau 53
14 XIAO-log Board 63
15 SCD41 Sensoraufbau 75
16 DIY Türkontakt Aufbau 86
17 Seeedstudio Relais für XIAO 96
18 CodeCellC6 101

0 Einführung

0.1 Für wen dieses Buch gedacht ist

Dieses Buch richtet sich an Entwickler, die mit Arduino bereits produktiv arbeiten und jetzt einen praktischen Schnelleinstieg in Zigbee suchen. Gemeint sind Leser, die nicht bei der ersten Blink-LED stehen, sondern bereits eigene kleine bis mittlere Projekte gebaut haben, seriellen Output sinnvoll lesen können und typische Build-, Flash- und Integrationsprobleme aus der Praxis kennen.

Du musst dafür kein Embedded-Spezialist sein, aber du solltest bereit sein, einen Schritt über reine Sketch-Routinen hinauszugehen. Der zentrale Wechsel in diesem Buch ist methodisch: weg von „läuft auf dem Tisch irgendwie“, hin zu reproduzierbaren, nachvollziehbaren Geräteprojekten mit klarer Struktur, sauberer Konfiguration und überprüfbarem Verhalten in Zigbee2MQTT und Home Assistant.

Nicht im Fokus steht ein kompletter Einstieg in Elektronikgrundlagen oder allgemeine Mikrocontroller-Basics. Das Buch setzt voraus, dass Begriffe wie GPIO, I2C, serieller Monitor und grundlegende Arduino-Projektstruktur nicht völlig neu sind. Der Mehrwert liegt stattdessen darin, diese Vorkenntnisse gezielt in den Zigbee-Kontext zu überführen und typische Stolperstellen im Übergang zu vermeiden.

Besonders passend ist das Buch für Leser, die sich genau diese Fragen stellen:

1. Warum joinen manche Geräte sofort sauber, andere aber nur sporadisch?
2. Warum erscheinen Exposes in Zigbee2MQTT nicht wie erwartet?
3. Warum ist ein Gerät im Labortest stabil, im Dauerbetrieb aber nicht?
4. Wie plant man ein Zigbee-Gerät so, dass Wartung und Erweiterung später nicht zum Chaos werden?

Wenn du aus Arduino IDE oder PlatformIO kommst und auf diese Fragen Antworten suchst, bist du exakt in der Zielgruppe. Dieses Buch schließt bewusst die Lücke zwischen funktionierendem Demo-Code und alltagstauglichem Zigbee-Gerätedesign.

0.2 Was du in diesem Buch lernst

Du lernst in diesem Buch nicht nur, wie ein Gerät grundsätzlich joined und Daten sendet. Der Fokus ist ein praxisnaher Schnelleinstieg mit klarer Schritt-für-Schritt-Umsetzung und einem sauberen Betriebsbezug. Ziel ist, dass du am Ende nicht nur Code kompilierst, sondern Geräte entwirfst, die im Alltag stabil laufen und sich später wartbar erweitern lassen.

Im praktischen Kern arbeitest du entlang einer klaren Projektlinie:

1. Einstieg über einfache Aktorik und reproduzierbare Join-Abläufe.
2. Übergang zu Sensorik mit nachvollziehbarer Messwertübertragung.

3. Ausbau zu Sleep- und Battery-End-Device-Mustern.
4. Event- und Sicherheitsnahe Geräte wie Button und Türkontakt.
5. Netzbetriebene Aktorik mit Relais und sicherem Integrationspfad.

Technisch lernst du mehrere Ebenen gleichzeitig sauber zu verknüpfen:

1. Gerätemodell: Rollen, Endpoints, Cluster, Attribute.
2. Übertragung: Reporting-Strategien statt blindem Intervall-Senden.
3. Betrieb: Zusammenspiel aus Zigbee2MQTT, Home Assistant und Logs.
4. Robustheit: Recovery-Verhalten, Re-Interview, Neu-Join und typische Fehlerpfade.
5. Energieverhalten: Sleep-Zyklen, Keep-Alive-Bezug und Auswirkungen auf Netzstabilität.

Ein zentraler Lernerfolg ist die Diagnosefähigkeit. Du lernst, Probleme nicht nur „wegzudebuggen“, sondern strukturiert einzugrenzen: Hardwarepfad, Netzwerkpfad, Gerätemodell und Automationspfad. Genau diese Trennung macht den Unterschied zwischen zufälligem Erfolg und reproduzierbarer Projektarbeit.

Am Ende sollst du eigene Zigbee-Geräte planen können, ohne bei jeder Designentscheidung raten zu müssen. Du weißt dann, welche Entscheidungen früh wichtig sind, welche später korrigierbar bleiben und wie du neue Geräte kontrolliert in ein bestehendes Smart-Home-Setup integrierst.

0.3 Verwendete Hardware

Eine vollständige und gepflegte Materialliste liegt im zugehörigen GitHub Repository. Dort findest du die jeweils aktuelle Zusammenstellung mit Boards, Sensoren, Zubehör und Variantenhinweisen für die einzelnen Projekte.

Auszugsweise folgt hier eine Auflistung, die ich im Verlauf ergänze.

Die Hardwarebasis bleibt bewusst flexibel, weil mehrere ESP32-Zigbee-Varianten sinnvoll einsetzbar sind. Für den praktischen Einstieg arbeiten wir zunächst mit sehr einfacher Peripherie wie LED und Taster, gehen danach zu SHT3x, SHT40 und BH1750 über und bauen später auf Battery-End-Device, Türkontakt und Relais-Aktorik auf.

Für die Beispiele eignen sich ESP32-H2, ESP32-C6 und ESP32-C5 sehr gut. Wichtig beim ESP32-H2: Er hat kein WiFi und ist für Zigbee und Thread ausgelegt.

Für die 230V-Relaisstrecke gilt ein klarer Rahmen. Der Fokus liegt auf Zigbee-Integration und sauberer Gerätestruktur, nicht auf elektrotechnischer Grundausbildung. Netzspannung erfordert geeignete Komponenten, sichere Verdrahtung und die Einhaltung lokaler Vorschriften.

LESEPROBE

3 Zigbee-Grundeinstellungen in Arduino IDE und PlatformIO

> Gute Zigbee-Firmware scheitert selten am C++-Code, aber oft an zwei falschen Build-Schaltern.

Dieses Kapitel ist bewusst keine Installationsanleitung für Arduino IDE oder PlatformIO. Stattdessen geht es um die technischen Grundeinstellungen, die für Zigbee-Projekte auf ESP32 entscheidend sind. Genau diese Einstellungen bestimmen, ob ein Sketch überhaupt mit Zigbee-Stack startet, im richtigen Rollenmodell läuft und im Backend stabil sichtbar wird.

Referenz für den XIAO ESP32-C6 im Arduino-Zigbee-Kontext: `https://wiki.seeedstudio.com/xiao_esp32c6_zigbee_arduino/`

3.1 Warum dieses Kapitel wichtig ist

Viele Fehler im Zigbee-Projektalltag sind keine Logikfehler im Sketch, sondern Konfigurationsfehler:

1. Falscher Zigbee-Modus (End Device statt Router oder umgekehrt).
2. Falsche Partitionstabelle.
3. Unvollständige Build-Flags.
4. Zu wenig Logtiefe bei der Inbetriebnahme.

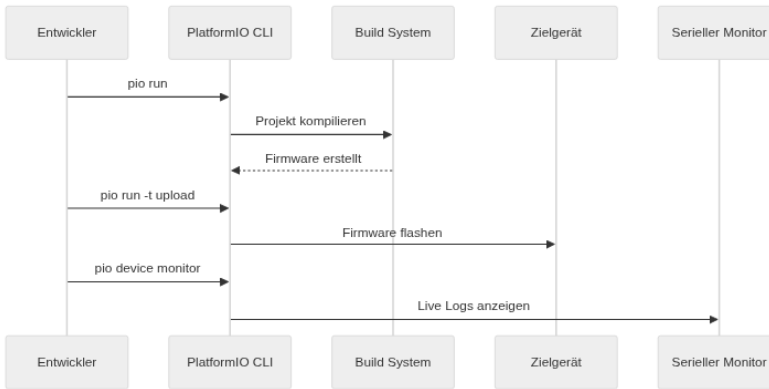

Abbildung 6: Build und Flash Schrittablauf

Das Ablaufdiagramm, das auch in GitHub in einer größeren Version verfügbar ist, zeigt einen exemplarischen Verbindungsablauf. Gleichzeitig wird deutlich, dass es entlang des gesamten Prozesses zahlreiche Stellen gibt, an denen Fehler auftreten können. Ein grundlegendes Verständnis dieses Ablaufs hilft dabei, Probleme gezielt einzugrenzen und effizient zu debuggen.

Das führt typischerweise zu schwer interpretierbaren Symptomen: Gerät startet,

joint aber nicht sauber, Exposes fehlen, oder das Verhalten passt nicht zur erwarteten Rolle.

3.2 Grundprinzip: Zigbee wird über Build-Kontext aktiviert

Im Arduino-ESP32-Core ist Zigbee nicht nur eine Bibliothek, die man einbindet. Der Stack erwartet passende Build- und Partitionsparameter. Diese Parameter müssen zur Rolle des Geräts passen:

1. `End Device (ED)` für Sleep-/Battery-nahe Sensorik, Buttons, kompakte Endknoten.
2. `Router (ZCZR-Kontext im Core)` für netzbetriebene Knoten mit Weiterleitungsrolle.

Im Sketch werden diese Annahmen oft per Compile-Guard abgesichert:

```
#ifndef ZIGBEE_MODE_ED
#error "Zigbee end device mode is not selected in Tools->Zigbee mode"
#endif
```

oder bei Router-Projekten:

```
#ifndef ZIGBEE_MODE_ZCZR
#error "Zigbee router mode is not selected in Tools->Zigbee mode"
#endif
```

Wenn diese Guards anschlagen, ist nicht der Sketch „kaputt“, sondern die Build-Konfiguration passt nicht zur Firmware-Rolle.

3.3 Arduino IDE: welche Zigbee-Einstellungen wirklich zählen

Hier ein Video von mir zu den wichtigsten Einstellungen: https://youtu.be/lvS98yui9JU

In der Arduino IDE gibt es mehrere Menüpunkte, aber für dieses Buch sind vor allem drei relevant:

1. `Tools - Board`: korrektes ESP32-Boardprofil (z. B. XIAO ESP32-C6).
2. `Tools - Zigbee mode`: passende Rolle (`Zigbee ED` oder `Zigbee Router` je nach Projekt).
3. `Tools - Partition Scheme`: Zigbee-kompatible Partitionierung.

Praktisch bedeutet das:

1. Die Boardauswahl steuert Pin-Mapping, Uploadparameter und Core-Kontext.
2. Der Zigbee-Modus steuert Rollen-Makros und Stack-Verhalten.
3. Die Partitionierung entscheidet, ob der Zigbee-Stack ausreichend Speicherlayout bekommt.

Wenn hier ein Punkt nicht passt, hilft auch ein sauberer Anwendungscode nur begrenzt.

Für den späteren Wechsel oder Parallelbetrieb mit PlatformIO ist wichtig: Diese Einstellungen lassen sich im jeweiligen Projekt in der `platformio.ini` explizit abbilden und dadurch reproduzierbar versionieren.

Typische Übersetzung Arduino IDE - PlatformIO:

1. `Tools - Board` - `board = seeed_xiao_esp32c6`
2. `Tools - Zigbee mode = Zigbee ED` - `build_flags = -DZIGBEE_MODE_ED`
3. `Tools - Partition Scheme = Zigbee ...` - `board_build.partitions = zigbee.csv`
4. Serial Monitor `115200` - `monitor_speed = 115200`

3.4 PlatformIO: Referenzkonfiguration für XIAO ESP32-C6

Für dieses Buch ist PlatformIO der Referenzpfad, weil alle relevanten Parameter explizit in `platformio.ini` stehen und versioniert werden können. Die oben gezeigten Einstellungen in der Arduino IDE können wie folgt in die `platformio.ini` Datei übernommen werden.

```
[env:seeed_xiao_esp32_c6]
platform = espressif32
board = seeed_xiao_esp32c6
framework = arduino
monitor_speed = 115200
upload_port = /dev/ttyACM1
monitor_port = /dev/ttyACM1
lib_deps =

board_build.partitions = zigbee.csv
board_build.filesystem = spiffs
build_flags =
           -DZIGBEE_MODE_ED
           -DCORE_DEBUG_LEVEL=5
```

Was diese Felder technisch bedeuten

Hier nochmal im Überblick was die

1. `board = seeed_xiao_esp32c6`: Aktiviert das korrekte Boardprofil für Pinbelegung, Upload und Core-Mapping.
2. `board_build.partitions = zigbee.csv`: Setzt eine Zigbee-taugliche Partitionstabelle.
3. `build_flags = -DZIGBEE_MODE_ED`: Definiert die Firmware-Rolle als End Device auf Build-Ebene.
4. `build_flags = -DCORE_DEBUG_LEVEL=5`: Erhöht die Logtiefe für Inbetriebnahme und Diagnose.

5. `monitor_speed = 115200`: Muss zum `Serial.begin(115200)` im Sketch passen, sonst werden Logs unlesbar.
6. `upload_port` / `monitor_port`: Fixieren den erwarteten Port für reproduzierbare Flash-/Monitorläufe.

3.5 End Device vs. Router: technische Folgen der Rollenwahl

Die Rollenwahl ist nicht nur semantisch, sondern wirkt sich auf den Betrieb aus:

1. End Device: für stromsparende Endknoten, häufig mit Sleep-Zyklen, ohne Routingpflicht.
2. Router: dauerhaft versorgt, aktiv im Netzrouting, relevanter für Mesh-Stabilität.

Für Maker-Projekte heißt das:

1. Sensor mit Deep Sleep - End Device.
2. Relaisknoten mit Dauerstrom - eher Router.
3. Mischentscheidungen führen oft zu Diagnosechaos, wenn Rolle und Einsatzprofil nicht zusammenpassen.

3.6 Partitionierung: warum `zigbee.csv` so wichtig ist

Der Zigbee-Stack benötigt ein passendes Speicherlayout. Mit einer ungeeigneten Partitionierung treten häufig diese Probleme auf:

1. Build-/Link-Fehler bei größeren Ständen.
2. Instabile Laufzeit trotz erfolgreichem Flash.
3. Unerwartete Restart-/Speicherprobleme.

Die Zigbee-Partitionierung ist daher kein optionales Feintuning, sondern Basisvoraussetzung.

3.7 Loglevel und Diagnosetiefe

`CORE_DEBUG_LEVEL=5` ist für Entwicklungsphasen sehr hilfreich:

1. Mehr Kontext beim Stack-Start.
2. Bessere Sicht auf Join-/Verbindungszustände.
3. Schnellere Einordnung von Timeouts und Fehlerpfaden.

Praxisregel:

1. In der Entwicklung eher höheres Loglevel.
2. In produktionsnahen Ständen reduzieren, um Rauschen und Laufzeitlast zu begrenzen.

Wichtig ist nicht „immer maximal“, sondern gezielt genug Information für reproduzierbares Debugging.

3.8 Serial, Build-Flags und Rollen-Guards zusammen denken

Stabile Inbetriebnahme entsteht durch die Kombination:

1. Korrekte `platformio.ini` oder IDE-Menüparameter.
2. Passende Compile-Guards im Sketch.
3. Lesbare serielle Ausgabe.
4. Klare Rollenentscheidung pro Projekt.

Wenn einer dieser Bausteine fehlt, wird Fehlersuche unnötig teuer.

LESEPROBE

7.4 Setup-Ablauf und Zigbee-Details

Im `setup()` wird zunächst die board-spezifische Antennenkonfiguration für den XIAO ESP32-C6 gesetzt. Die Zeilen mit `WIFI_ENABLE` und `WIFI_ANT_CONFIG` sind nur für diese Hardwarevariante relevant und schalten auf die externe Antenne statt auf die integrierte Keramikantenne. Danach folgen LED-Initialisierung und Sensor-Power-On, sodass der SHT3x vor dem ersten Read stabil versorgt ist.

Nach `Wire.begin()` und `sht.begin()` werden die Zigbee-Geräteparameter gesetzt: Hersteller/Modell, Temperaturgrenzen, Toleranz, Stromquelle als Batterie und die Erweiterung um den Feuchtepfad mit `addHumiditySensor(...)`. Damit ist das Gerätemodell vollständig beschrieben, bevor es mit `Zigbee.addEndpoint(&zbTempSensor)` registriert wird. Diese Reihenfolge ist fachlich wichtig, weil das Interview in Zigbee2MQTT nur dann das vollständige Profil sieht.

Der Stack-Start erfolgt mit `Zigbee.begin(&zigbeeConfig, false)` auf Basis der expliziten ED-Konfiguration. Wenn der Start scheitert, signalisiert der Sketch das per LED-Muster und startet neu. Danach wartet die Schleife `while (!Zigbee.connected())` auf eine reale Netzwerkverbindung, bevor Mess- und Reportlogik ausgeführt werden. So wird vermieden, dass Sensorwerte „ins Leere“ produziert werden, während das Gerät noch gar nicht im Zigbee-Netz eingebunden ist.

Ein zusätzlicher, praxisnaher Block unterscheidet zwischen Kaltstart und Wakeup aus Deep Sleep: Nur beim frischen Boot wird eine kurze Pairing-Wartezeit eingeräumt, beim normalen Wakeup-Zyklus entfällt dieser Zusatz-Delay. Dadurch bleibt der Betrieb energieeffizient, ohne den initialen Join-Prozess unzuverlässig zu machen.

7.5 Flash-Hinweis bei aktivem Deep Sleep

Wenn Deep Sleep bereits aktiv ist, kann die USB-Verbindung beim Flashen unzuverlässig wirken, weil das Board sehr schnell wieder in den Sleep-Pfad fällt. In diesem Fall hilft in der Praxis der klassische Bootloader-Weg: Boot-Taste gedrückt halten, Board starten oder resetten, Flash-Vorgang beginnen und die Taste erst nach sicherem Upload loslassen. Falls der Port weiterhin nicht reagiert, ist ein kurzes vollständiges Stromlosmachen oft der schnellste Reset des Zustands.

7.6 Batterielaufzeit messen und gezielt optimieren mit PPK2

Hier noch ein Tipp aus der Praxis für Laufzeitabschätzung und Optimierung: Für eine belastbare Aussage brauchst du nicht nur den konfigurierten Sleep-Intervallwert, sondern das reale Verhältnis aus aktiver Phase und Deep-Sleep-Phase. Der Durchschnittsstrom ergibt sich als gewichteter Mittelwert beider Phasen; daraus folgt die Laufzeit über Kapazität geteilt durch

Durchschnittsstrom. Diese Rechnung ist nur dann verlässlich, wenn die Stromwerte wirklich **gemessen** und nicht nur geschätzt wurden.

Genau dafür ist das Power Profiler Kit 2 (PPK2) ein sehr gutes Werkzeug (Ein Video von mir zu diesem Kit ist ein paar Absätze weiter unten verlinkt). Das PPK2 ist ein hochauflösendes Strommessgerät für Embedded-Geräte, mit dem du Stromverläufe zeitlich aufgelöst sehen kannst: Wakeup-Spike, Sensor-Read, Zigbee-Sendephase und anschließender Deep Sleep. Damit erkennst du sofort, ob die gewünschten Sleep-Ströme tatsächlich erreicht werden oder ob noch ein Verbraucher aktiv bleibt, zum Beispiel ein Sensorpfad, ein Pull-up oder eine nicht sauber abgeschaltete Peripherie.

Du kannst messen, ob dein Device nach `esp_deep_sleep_start()` wirklich auf den erwarteten Ruhestrom fällt und wie lang die aktive Phase pro Zyklus tatsächlich dauert. Erst mit diesen Messwerten kannst du die Reporting-Intervalle, Wakeup-Zeiten und Sensorversorgung sinnvoll optimieren, ohne die Zuverlässigkeit zu verschlechtern.

Energieoptimierung bedeutet daher nicht „blind seltener senden“, sondern kontrolliert einstellen und nachmessen. Praktisch erreichst du das über passende Sleep-Intervalle, sinnvolle Reporting-Deltas, kurze Aktivfenster, sauberes Abschalten nicht benötigter Peripherie und reduzierte Debug-Logs in produktionsnahen Ständen. Gleichzeitig darf die Stabilität nicht leiden: Wenn ein zu aggressives Tuning zu häufigen Rejoins oder verpassten Reports führt, ist der scheinbare Energiespareffekt schnell wieder verloren.

Weiterführende Videos:

- Power Profiler Kit 2 im Detail (Messaufbau und Auswertung):
  - https://www.youtube.com/watch?v=P5mJvBcywbw
- ESP32 Sleep-Modes im Überblick (unabhängig von Zigbee):
  - https://www.youtube.com/watch?v=v9uWip4GbYE

7.7 Typische Sleep-Probleme und Diagnose

Typische Fehlerbilder sind ein ausbleibendes Wakeup, erfolgreiche Wakeups ohne Report, fehlende Feuchtewerte trotz Sensorread, unplausible Batterieanzeigen oder ein instabiler Flash-Prozess bei aktivem Sleep. Für die Fehlersuche hilft eine feste Reihenfolge: zuerst Wakeup-Mechanik prüfen, dann I2C/Sensorread, danach Zigbee-Verbindungsstatus, anschließend Exposes/Interview und zuletzt Batteriemesspfad inklusive Datentypen und Skalierung.

Wichtig im Projektalltag: Wenn du ein bereits gepairtes Gerät vom alten Profil auf ein erweitertes Profil umstellst, musst du in Zigbee2MQTT ein neues Interview auslösen oder sauber neu joinen. Sonst bleiben oft alte Exposes aktiv, obwohl die Firmware bereits zusätzliche Daten liefert.

7.8 Fazit

In diesem Kapitel hast du den Übergang vom einfachen Sensordemo zum alltagstauglichen Battery End Device gemacht. Du hast gesehen, wie SHT3x-Messung, Zigbee-Reporting und Deep Sleep sauber zusammenspielen und warum echte Strommessungen für belastbare Laufzeitaussagen unverzichtbar sind. Im nächsten Kapitel lernen wir ein spezielles Modul kennen, mit dem sich besonders einfach und schnell ein eigener Zigbee Temperatursensor realisieren lässt.

8 XIAO-log SHT40, BH1750 und Zigbee-Endgerät

Dieses Projekt ist der freundlichste Shortcut zu echten Messwerten: XIAO draufstecken, flashen und freuen statt Kabel-Yoga.

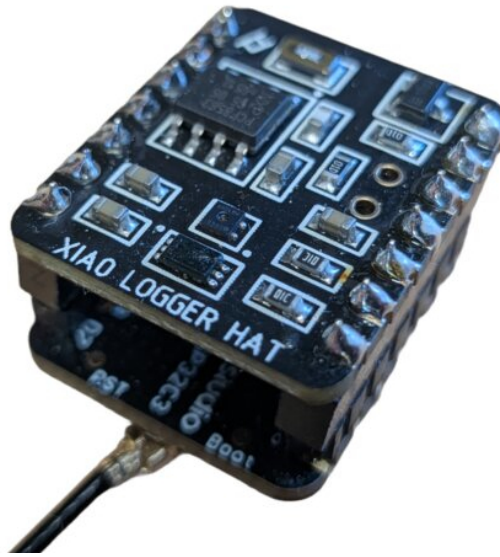

Abbildung 14: XIAO-log Board

In diesem Kapitel verwenden wir bewusst die fertige `XIAO-log`-Platine als Hardwarebasis. Ein XIAO ESP32-C5 oder XIAO ESP32-C6 wird aufgesteckt, die Firmware geflasht, und erste Messwerte sind ohne aufwendiges Verdrahten oder langes Fehlersuchen verfügbar.

Das Board kombiniert drei für den Einstieg sehr nützliche Bausteine:

1. `SHT40` für Temperatur und Luftfeuchte.
2. `BH1750` für Beleuchtungsstärke (Lux).
3. `RTC` als Option für lokale Zeitführung ohne Netzwerk.

Für dieses Kapitel ist der RTC-Teil nicht Gegenstand der Darstellung, weil wir den Schwerpunkt auf Zigbee-Endgerät, Messwerterfassung und zuverlässiges Reporting legen.

Referenz zum Board: `https://github.com/potblitd/XIAO-log`

8.1 Warum XIAO-log in diesem Kapitel

Mit `XIAO-log` lassen sich typische Fehlerquellen bereits früh ausschließen. Da der Sensor direkt auf der Platine sitzt, entfallen provisorische Verdrahtungen mit Jumperkabeln und unklare I2C-Probleme durch lose Steckkontakte. Gleichzeitig entsteht ein schneller, reproduzierbarer Aufbau, der sich sowohl für Tests als auch für den späteren Vergleich mit der Produktivhardware eignet.

Das ist besonders wertvoll, wenn parallel an Zigbee2MQTT-Exposes, Join-Verhalten und Sleep-Zyklen gearbeitet wird. So entstehen schneller klare Aussagen, ob ein Problem aus dem Funkpfad oder aus dem Anwendungscode kommt.

8.2 Zigbee-Rolle und Projektziel

Der Sketch ist als `Zigbee End Device` ausgelegt. Das Gerät wacht auf, liest Sensoren, meldet Werte an den Coordinator und geht wieder in den Deep Sleep. Dieses Muster ist für batterienahe Sensorknoten der richtige Standard, weil die aktive Funkzeit kurz bleibt.

Die erste Codezeile erzwingt genau das:

```
#ifndef ZIGBEE_MODE_ED
#error "Zigbee end device mode is not selected in Tools->Zigbee mode"
#endif
```

Damit bricht der Build sofort ab, wenn versehentlich ein falscher Zigbee-Modus gewählt wurde.

8.3 Aufbau des Sketches

Der Code ist klar in Funktionsblöcke getrennt:

1. Bibliotheken und Konfiguration (`Timing`, `Pins`, Endpunkte).
2. Sensor-Initialisierung mit robuster I2C-Prüfung.
3. Zigbee-Endpunkte für Klima- und Luxdaten.
4. Join-Ablauf mit Timeout.
5. Reporting mit ACK-Überwachung und Retry-Logik.
6. Rückkehr in Deep Sleep.

Diese Struktur ist sehr gut für produktionsnahe Geräte geeignet, weil sie Fehlerzustände sichtbar macht und trotzdem deterministisch bis zum Sleep durchläuft.

8.4 PlatformIO-Konfiguration

```
[env:seeed_xiao_esp32c6]
platform = https://github.com/pioarduino/platform-espressif32/
    releases/download/stable/platform-espressif32.zip
```

LESEPROBE

In diesem Kapitel wird das `Seeed Studio Single Relay` verwendet. Das Modul kann Lasten bis `10 A` bei `230 V AC` schalten. Andere Hersteller funktionieren ebenfalls, solange die elektrischen Kenndaten, die Isolationsanforderungen und die Ansteuerlogik vergleichbar sind.

Wichtig: 10 A sind ein Maximalwert unter definierten Bedingungen. Schaltvermögen hängt von Lastart (ohmsch, induktiv, kapazitiv), Einschaltstrom und thermischer Umgebung ab.

```
#include <Arduino.h>
#ifndef ZIGBEE_MODE_ZCZR
#error "Zigbee router mode is not selected in Tools->Zigbee mode"
#endif
#include "Zigbee.h"
/* Zigbee power outlet configuration */
#define ZIGBEE_OUTLET_ENDPOINT 1
static const uint8_t relayPin = 1;
static const bool RELAY_ACTIVE_LOW = false;
static const uint8_t ledPin = LED_BUILTIN;
uint8_t button = BOOT_PIN;
ZigbeePowerOutlet zbOutlet = ZigbeePowerOutlet(ZIGBEE_OUTLET_ENDPOINT
    );
/******************** Relay functions **************************/
void setRelay(bool value) {
  bool pinLevel = RELAY_ACTIVE_LOW ? !value : value;
  digitalWrite(relayPin, pinLevel);
  digitalWrite(ledPin, value ? LOW : HIGH);
}
void setup() {
  Serial.begin(115200);
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, RELAY_ACTIVE_LOW ? HIGH : LOW);
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);
  pinMode(button, INPUT_PULLUP);
  zbOutlet.setManufacturerAndModel("Espressif", "ZBPowerOutlet");
  zbOutlet.onPowerOutletChange(setRelay);
  Serial.println("Adding ZigbeePowerOutlet endpoint to Zigbee Core");
  Zigbee.addEndpoint(&zbOutlet);
  if (!Zigbee.begin(ZIGBEE_ROUTER)) {
    Serial.println("Zigbee failed to start!");
    Serial.println("Rebooting...");
    ESP.restart();
  }
  Serial.println("Connecting to network");
  while (!Zigbee.connected()) {
    Serial.print(".");
    delay(100);
  }
  Serial.println();
}
```

```
42
43 void loop() {
44   if (digitalRead(button) == LOW) { // Push button pressed
45     delay(100);
46     int startTime = millis();
47     while (digitalRead(button) == LOW) {
48       delay(50);
49       if ((millis() - startTime) > 3000) {
50         Serial.println("Resetting Zigbee to factory and rebooting in 1
               s.");
51         delay(1000);
52         Zigbee.factoryReset();
53       }
54     }
55     zbOutlet.setState(!zbOutlet.getPowerOutletState());
56   }
57   delay(100);
58 }
```

12.3 Projektziel und Zigbee-Rolle

Der Sketch baut aus dem ESP32 ein Zigbee-`PowerOutlet`-Gerät mit lokalem Relaisausgang. Anders als Battery-End-Device-Beispiele läuft dieses Gerät als Router-Rolle:

```
1 #ifndef ZIGBEE_MODE_ZCZR
2 #error "Zigbee router mode is not selected in Tools->Zigbee mode"
3 #endif
4 ...
5 if (!Zigbee.begin(ZIGBEE_ROUTER)) { ... }
```

Das ist hier sinnvoll, weil ein netzbetriebenes Relaismodul dauerhaft online ist und als aktiver Infrastrukturknoten betrieben werden kann.

12.4 Codeaufbau: die wichtigen Bausteine

Endpoint und Hardwarepins

Der Sketch definiert einen Outlet-Endpoint sowie Relais-, LED- und Button-Pin:

```
1 #define ZIGBEE_OUTLET_ENDPOINT 1
2 static const uint8_t relayPin = 1;
3 static const bool RELAY_ACTIVE_LOW = false;
4 static const uint8_t ledPin = LED_BUILTIN;
5 uint8_t button = BOOT_PIN;
```

Entscheidend ist `RELAY_ACTIVE_LOW`: Damit lässt sich die Logik an unterschiedliche Relay-Boards anpassen, ohne die restliche Steuerlogik umzuschreiben.