

Emergency Stop Button – ESP32

Technische Dokumentation

Juni 2026

Inhaltsverzeichnis

1 Ziel und Anforderungen

Ein physischer Notaus-Taster soll beim Drücken so schnell wie möglich einen API-Call absetzen. Der ESP32 befindet sich im Ruhezustand und wird durch den Tastendruck geweckt.

- Latenz Knopfdruck \rightarrow API-Call: $< \mathbf{250\ ms}$
- Stromversorgung: LiPo 2000 mAh (kabellos, batteriebetrieben)
- Möglichst lange Akkulaufzeit (Taster wird selten gedrückt, $\leq 1 \times /h$)
- Einfache, wartungsarme Architektur

2 Architekturentscheidung

2.1 Bewertete Optionen

Option	Latenz	Ø Strom	Gateway nötig	Komplexität
Deep Sleep + WiFi (RTC-Memory)	600–1300 ms	$\approx 0,02\ \text{mA}$	nein	niedrig
WiFi Light Sleep (DTIM=10)	150–250 ms	0,5–1 mA	nein	niedrig
BLE Light Sleep + Gateway	100–300 ms	0,5–2 mA	ja	hoch

2.2 Entscheidung: WiFi Light Sleep

WiFi Light Sleep erfüllt alle Anforderungen ohne zusätzliche Infrastruktur:

- Der ESP32 hält die WiFi-Verbindung aufrecht und schläft nur die CPU
- Ein GPIO-Interrupt weckt den ESP32 sofort beim Tastendruck
- Der API-Call geht direkt vom ESP32 – kein Container, kein Gateway

Warum nicht BLE? Bluetooth LE wäre minimal sparsamer ($\approx 0,5\ \text{mA}$ vs. $\approx 1\ \text{mA}$), erfordert aber ein dauerhaft laufendes Gateway-Gerät (Raspberry Pi, PC), das seinerseits Strom verbraucht und eine Fehlerquelle darstellt.

Warum nicht Deep Sleep? Deep Sleep braucht beim Aufwachen 600–1300 ms (WiFi-Reconnect), was die Latenzanforderung von 250 ms verfehlt.

3 WiFi Light Sleep – Funktionsprinzip

Im WiFi Light Sleep (Modem Sleep) schläft die CPU, während der WiFi-Stack aktiv bleibt. Der Router sendet alle 100 ms einen Beacon; mit DTIM=10 wacht der ESP32 automatisch alle $\approx 1000\ \text{ms}$ für 1–2 ms auf, um gepufferte Pakete abzuholen. Die Verbindungsassoziation bleibt dabei erhalten.

3.1 DTIM-Einstellung und Stromverbrauch

DTIM	Wakeup-Intervall	Ø Strom
1	100 ms	5–10 mA
3	300 ms	2–4 mA
10	1000 ms	0,5–1 mA

Empfohlen: DTIM=10 – sparsamste Option, Verbindung bleibt stabil.

3.2 Akkulaufzeit (2000 mAh LiPo)

Bei ≈ 1 mA Durchschnittsstrom (DTIM=10, selten gedrückt) und 80 % nutzbarer Kapazität:

$$t = \frac{1600 \text{ mAh}}{1 \text{ mA}} \approx 67\text{--}80 \text{ Tage}$$

Die **Latenz von 250 ms** hat Vorrang vor maximaler Akkulaufzeit – Deep Sleep scheidet daher aus (WiFi-Reconnect 600–1300 ms). Mit 2000 mAh und WiFi Light Sleep sind ca. 2,5 Monate Betrieb ohne Laden realistisch.

4 Latenzbudget

Schritt	Zeit
ESP32 Wakeup aus Light Sleep	1–5 ms
WiFi-Verbindung prüfen (bereits aktiv)	0 ms
HTTP-Request aufbauen	20–50 ms
TLS-Handshake (HTTPS)	50–150 ms
Server-Antwort	20–50 ms
Gesamt	$\approx 100\text{--}250$ ms

5 Hardware

5.1 Empfohlene Boards

Board	Lader	Preis	Bemerkung
FireBeetle ESP32-E (DFRobot)	integriert	8–12 €	Low-Power optimiert
LOLIN D32 (Wemos)	TP4054	5–8 €	günstig, bewährt
Adafruit HUZZAH32 Feather	MCP73831	≈ 20 €	gute Dokumentation

5.2 Akku-Spezifikation

Für den FireBeetle 2 wird ein **1S LiPo** mit **JST-PH-2,0-Stecker** und **BMS** benötigt:

Merkmal	Wert
Typ	LiPo / Li-Polymer, 1 Zelle (1S)
Nennspannung	3,7 V (voll: 4,2 V)
Stecker	JST PH, 2,0 mm Raster, 2-polig
Schutzschaltung	Ja (BMS/PCM) – Pflicht bei Deep-Sleep-Betrieb
Kapazität	2000 mAh

Achtung Polarität: JST-PH-Stecker sind nicht normiert – Polung vor dem Einstecken mit Multimeter prüfen. Sicherste Quelle: Akku direkt bei DFRobot kaufen.

5.3 Schaltung

```
ESP32 Taster
GPIO 9 ---- [Taster] ---- GND
              (interner Pull-Up aktiv: HIGH = offen, LOW = gedrueckt)

LiPo 2000mAh (JST PH 2.0, mit BMS) ---- BAT+ / BAT- des Boards
```

Kein weiteres Bauteil nötig – der interne Pull-Up des ESP32 reicht.

6 Software

6.1 Abhängigkeiten (Arduino IDE)

- Board-Package: `esp32` by Espressif (Boards Manager)
- Bibliotheken: `WiFi.h`, `HTTPClient.h` (im `esp32`-Package enthalten)

6.2 Konfiguration in `EmergencyStopButton.ino`

```
#define BUTTON_PIN 9
#define WIFI_SSID "DEIN_SSID"
#define WIFI_PASSWORD "DEIN_PASSWORT"
#define API_URL "https://deine-api.example.com/emergency-stop"
#define API_TOKEN "DEIN_API_TOKEN"
```

6.3 Ablauf

1. `setup()`: WiFi verbinden, `WIFI_PS_MAX_MODEM` aktivieren
2. `loop()`: `esp_light_sleep_start()` – CPU schläft, WiFi aktiv
3. Tastendruck löst GPIO-Interrupt aus – ESP32 wacht auf
4. `sendEmergencyStop()` sendet HTTP-POST an API
5. Warten bis Taster losgelassen, zurück zu Schritt 2

6.4 Kritische API-Funktion

```
void sendEmergencyStop() {
  HTTPClient http;
  http.begin(API_URL);
  http.addHeader("Content-Type", "application/json");
  http.addHeader("Authorization", "Bearer_" API_TOKEN);
  http.setTimeout(3000);
  String body = "{\"event\":\"emergency_stop\",\"device\":\"esp32-estop\"}";
  int httpCode = http.POST(body);
  http.end();
}
```

7 Deployment-Hinweise

- **HTTPS:** TLS-Handshake kostet 50–150 ms. Falls die Latenz kritisch ist und das Netz vertrauenswürdig, kann HTTP verwendet werden (dann \approx 50–100 ms Gesamt).
- **DTIM am Router:** Manche Router ignorieren den DTIM-Wunsch des Clients. Im Zweifelsfall DTIM=3 am Router einstellen.
- **API-Token:** Nicht im Quellcode einchecken – z.B. in eine separate `secrets.h` auslagern, die im `.gitignore` steht.
- **Watchdog:** Bei produktivem Einsatz einen Hardware-Watchdog aktivieren, damit der ESP32 sich bei WiFi-Verlust selbst neu startet.

8 Dateien

Datei	Beschreibung
EmergencyStopButton.ino	Arduino-Sketch (Hauptprogramm)
eStopESP32.tex	Diese Dokumentation