ESP32 Web-Speicheroszi

Aufgeräumte Projektdokumentation aus dem Chat: MicroPython-Webserver, Browser-Diagramm, Messintervall, Trigger und GitHub-Ablage.

Stand: 05.06.2026

1. Ziel des Projekts

• Ein ESP32-C6 soll als kleines Speicheroszilloskop arbeiten.

• Der ESP32 stellt einen Webserver bereit.

• Der Webclient im Browser startet Messungen, holt Messwerte und zeichnet ein Diagramm.

• Die Messdaten werden zusätzlich als CSV angezeigt und können kopiert oder weiterverarbeitet werden.

• Die Entwicklung soll mit GitHub Desktop versioniert werden.

2. Grundprinzip

Die Arbeitsteilung ist bewusst einfach gehalten: Der ESP32 misst und liefert Daten. Der Browser übernimmt Bedienoberfläche, CSV-Anzeige und Diagrammzeichnung. Dadurch bleibt der MicroPython-Teil überschaubar.

Teil	Aufgabe
ESP32-C6	WLAN verbinden, Webserver starten, ADC messen, CSV senden
Browser	Messparameter eingeben, /messen aufrufen, /werte laden, Diagramm zeichnen
GitHub	Projektstände sichern und nachvollziehbar machen

3. Entwicklungsstufen

Version	Kernfunktion
V1	Webserver, Anzahl Messwerte, Messung, CSV-Ausgabe, Diagramm im Browser
V2	Messintervall dt in Mikrosekunden und zeilenweises CSV-Senden
V3	Triggerlevel und Triggerart als Pegeltrigger
V4	Flankentrigger: steigende und fallende Flanke; ESP32-C6 mit GPIO4; dt=0-Korrektur empfohlen/eingebaut

4. Aktueller Bedienablauf

• ESP32 mit MicroPython starten.

• Im Thonny-Ausgabefenster die IP-Adresse des ESP32 ablesen.

- Im Browser die Adresse öffnen, zum Beispiel http://192.168.x.x.
- Anzahl Messwerte, Messintervall, Triggerlevel und Triggerart einstellen.
- Auf „Messung starten“ klicken.
- Der Browser zeigt Status, Diagramm und CSV-Daten an.

5. Web-Endpunkte

Endpunkt	Bedeutung
/	liefert die HTML-Bedienoberfläche
/messen?n=300&dt=100&trig=2000&mode=rising	startet eine Messung mit Parametern
/werte	liefert die gespeicherten Messwerte als CSV

6. Messparameter

Parameter	Beispiel	Bedeutung
n	300	Anzahl der Messwerte, begrenzt durch MAX_N
dt	100	Messintervall in Mikrosekunden; dt=0 bedeutet „so schnell wie möglich“
trig	2000	Triggerlevel im ADC-Bereich 0 bis 4095
mode	none	kein Trigger
mode	rising	steigende Flanke
mode	falling	fallende Flanke

7. Triggerlogik

Die letzte Ausbaustufe ist ein echter Flankentrigger. Das ist besser als ein reiner Pegeltrigger, weil die Messung nicht sofort startet, wenn das Signal bereits oberhalb oder unterhalb des Triggerlevels liegt.

• Steigende Flanke: Erst warten, bis ADC < Triggerlevel ist. Danach startet die Messung, sobald ADC >= Triggerlevel wird.

• Fallende Flanke: Erst warten, bis ADC > Triggerlevel ist. Danach startet die Messung, sobald ADC <= Triggerlevel wird.

• Ein Timeout verhindert, dass der ESP32 unbegrenzt in der Trigger-Warteschleife hängt.

Kernidee des Flankentriggers

```
# steigende Flanke
erst warten: ADC < trigger_level
dann starten: ADC >= trigger_level

# fallende Flanke
erst warten: ADC > trigger_level
dann starten: ADC <= trigger_level
```

8. CSV-Datenformat

Die Werte werden zeilenweise übertragen. Das spart RAM auf dem ESP32, weil kein riesiger CSV-String aufgebaut werden muss.

```
zeit_us;adc_wert
0;1876
100;1878
200;1875
300;1880
```

Bei dt=0 wird statt einer Zeit die Messpunktnummer als x-Wert verwendet. Dadurch bleibt das Diagramm auch bei schnellster Messung sinnvoll darstellbar.

```
for i in range(N):
    if DT_US == 0:
        zeit = i
    else:
        zeit = i * DT_US

    zeile = str(zeit) + ";" + str(werte[i]) + "\n"
    client.send(zeile.encode("utf-8"))
```

9. Hardware-Hinweise

• Aktuell wird ein ESP32-C6 verwendet. Deshalb ist GPIO34 nicht passend; dieser Pin existiert auf dem ESP32-C6 nicht wie beim klassischen ESP32.

• GPIO4 wird im aktuellen Projekt als ADC-Pin verwendet, sofern das verwendete Board diesen Pin als ADC herausführt.

• Der ADC-Bereich wird als 0 bis 4095 behandelt.

• Die WLAN-Zugangsdaten sollten in veröffentlichten Dateien nicht offen stehen. Für GitHub besser Platzhalter verwenden.

10. GitHub-Ablage

GitHub soll hier nicht kompliziert eingesetzt werden. Für den Anfang reicht der Arbeitsablauf mit GitHub Desktop:

```
Datei ändern
→ Summary schreiben
→ Commit to main
→ Push origin
```

Empfohlene Ordnerstruktur im Repository:

```
ESP32_Web_Speicheroszi/
├── main.py
├── README.md
├── versionen/
│   ├── ESP32-WEB-Speicheroszi-1.py
│   ├── ESP32-WEB-Speicheroszi-2.py
│   ├── ESP32-WEB-Speicheroszi-3.py
│   └── ESP32-WEB-Speicheroszi-4.py
└── notizen/
    └── changelog.txt
```

Wichtig: main.py ist immer die aktuelle Datei für den ESP32. Alte Stände liegen in versionen/.

11. Sinnvolle Commit-Texte

```
Basis-Webserver mit Diagramm
Messintervall hinzugefügt
CSV zeilenweise senden
Pegeltrigger hinzugefügt
Flankentrigger eingebaut
dt=0 im Diagramm korrigiert
Projektstand ESP32-C6 GPIO4 gesichert
```

12. Aktueller Projektstand

Der aktuelle Stand ist ein brauchbares kleines Browser-gesteuertes Messsystem. Es ist noch kein professionelles Oszilloskop, aber bereits ein sehr gutes Lern- und Experimentierprojekt: Webserver, ADC-Messung, Zeitbasis, Trigger, CSV und Canvas-Diagramm arbeiten zusammen.

Empfehlung: Den funktionierenden Stand jetzt in GitHub sichern und erst danach weitere Funktionen ergänzen.

13. Mögliche spätere Erweiterungen

- Hysterese für den Trigger, um Rauschen um das Triggerlevel besser zu beherrschen.
- Pre-Trigger-Speicher, damit auch Werte vor der Flanke sichtbar werden.
- CSV-Download als Datei im Browser.
- Mehrere Kanäle, falls später mehrere ADC-Eingänge verwendet werden.
- Automatische Skalierung umschaltbar: Min/Max oder fester ADC-Bereich 0 bis 4095.
- README.md für GitHub mit kurzer Bedienungsanleitung.

Hinweis zur Dokumentation

Dieses PDF ist eine aufgeräumte Zusammenfassung des Projektverlaufs. Es ersetzt nicht die Python-Dateien selbst. Die lauffähigen Programmstände sollten zusätzlich im GitHub-Repository und lokal auf dem PC gespeichert werden.