ESP32 Phasenanschnittsteuerung (AC-Dimmer)

1. Gemini

Dieses Programm verwandelt einen **ESP32** in einen intelligenten Phasenanschnitt-Dimmer (z. B. zur Steuerung von Helligkeit oder Leistung von 230V-Verbrauchern an einem 50Hz-Netz) mit integrierter WLAN-Steuerung.

Die Hauptfunktionen im Detail:

- **Nulldurchgangserkennung (Zero-Crossing):** Ein Interrupt (`SyncPin`) synchronisiert das Programm bei jeder Halbwelle der Netzspannung (alle 10 ms bei 50Hz), um den genauen Startzeitpunkt für die Phasenanschnitt-Verzögerung zu kennen.

- **Präzise Timer-Steuerung:** Ein Hardware-Timer berechnet die exakte Verzögerungszeit (abhängig vom Dimmwert in Mikrosekunden) und steuert den TRIAC-Trigger-Pin mit einem kurzen Zündimpuls an.

- **WLAN & Webserver:** Das Gerät verbindet sich mit einem WLAN und startet einen asynchronen Webserver auf Port `8080`. Über eine einfache HTML-Oberfläche (`http://<IP-Adresse>:8080`) lässt sich der Dimmwert stufenlos von **0 % (aus)** bis **100 % (volle Leistung)** einstellen.

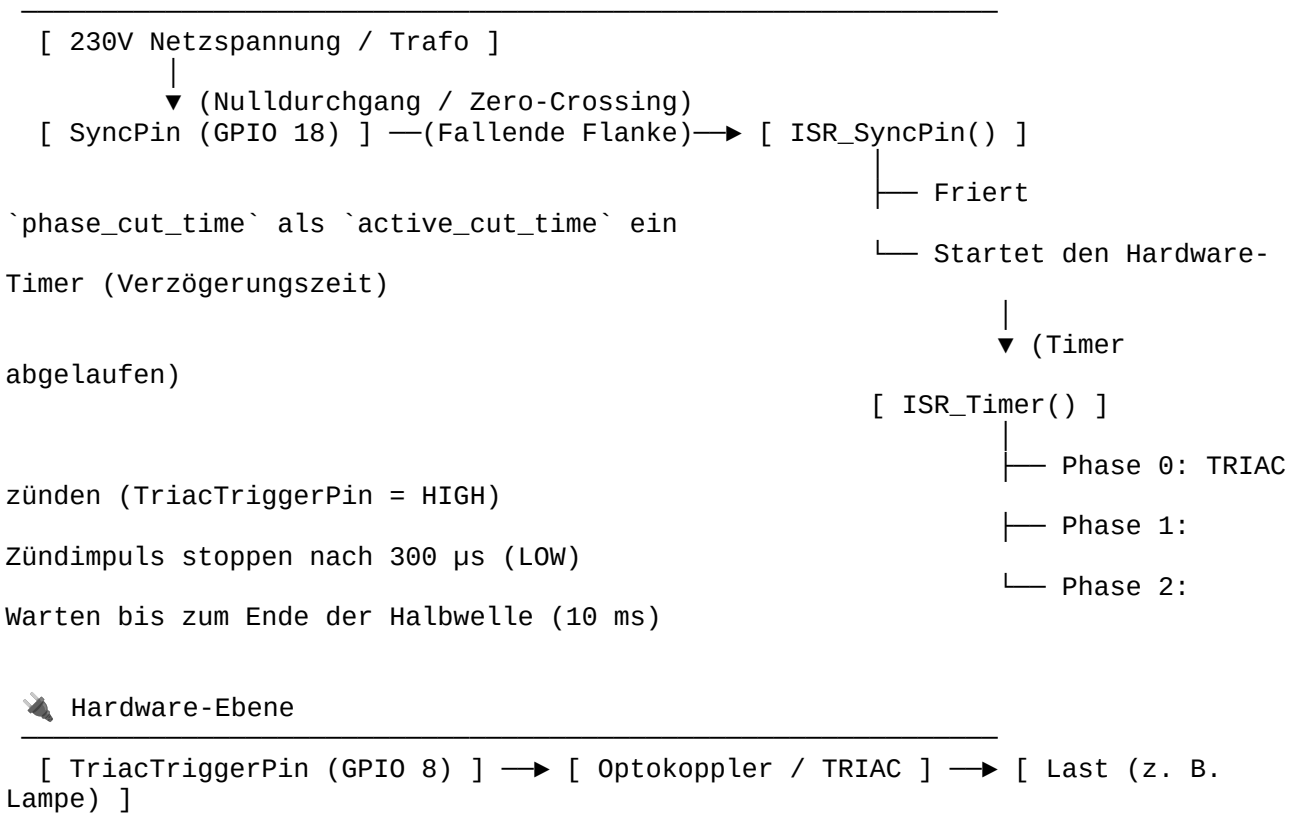
- **Hohe Stabilität (Thread-Safe):**

- Verwendet kritische Bereiche (`portMUX_TYPE`), um Datenkonflikte (Race Conditions) zwischen dem Webserver und den schnellen Interrupt-Routinen (ISR) zu verhindern.

- Speichert den aktuellen Zustand im **RTC-RAM**, damit er bei bestimmten Reset-Vorgängen erhalten bleibt.

Hier ist eine übersichtliche Struktur-Skizze deines Programms. Sie zeigt, wie die verschiedenen Komponenten (Webserver, Hardware-Interrupts und Timer) auf dem ESP32 zusammenarbeiten:

```
🌐 Web-Ebene (Nicht-Echtzeit / FreeRTOS)
──────────────────────────────────────────────────────────
[ Webbrowser / Client ]
        │ (HTTP GET /setdim?dimm=XX)
        ▼
[ AsyncWebServer ]
        │
        ├── 1. Liest den neuen Dimmwert (0–100 %)
        ├── 2. Berechnet die Phasenanschnitt-Zeit (`phase_cut_time` in µs)
        └── 3. Schützt die Variable mit einem Mutex (`pinMux`)


⚡ Echtzeit-Ebene (Interrupts im IRAM - Höchste Priorität)
```



Kurze Erklärung des Ablaufs:

1. **Entkopplung von Webserver und Echtzeit:** Der Webserver läuft im Hintergrund und ändert lediglich den globalen `dim_value`. Er stört die zeitkritische Netzzählung nicht.
2. **Die Nulldurchgangserkennung (`ISR_SyncPin`):** Sobald das Stromnetz den Nulldurchgang erreicht, wird die ISR sofort ausgeführt. Sie „friert“ den aktuellen Dimmwert für diese Halbwelle ein (verhindert Flackern durch Wertänderungen mitten im Takt) und startet einen Hardware-Timer.
3. **Die Phasensteuerung (`ISR_Timer`):** Der Timer wartet exakt so lange, wie es der Dimmwert vorschreibt (z. B. 5 ms bei 50%), schaltet dann den TRIAC für 300 Mikrosekunden ein, schaltet ihn wieder ab und wartet auf den nächsten Nulldurchgang.

Wichtige Optimierungen & gefundene Schwachstellen (gemini)

1. **Mutex-Inkonsistenz & Race Conditions (`phase_cut_time`)**
   Im Webserver (`/setdim`) schützt du das Schreiben von `phase_cut_time` mit `pinMux`. In der Timer-ISR (`ISR_Timer`) hast du dagegen `timerMux` verwendet. Da es sich hierbei um zwei völlig unabhängige Mutexe bzw. Spinlocks handelt, bieten sie keinen gegenseitigen Schutz vor gleichzeitigen Zugriffen.
   Timing-Konsistenz innerhalb einer Halbwelle

Wenn über den Webserver während einer laufenden Halbwelle ein neuer Dimmwert gesetzt wird, ändert sich `phase_cut_time` mitten im Timing-Zyklus. Das kann zu unsauberen Zündimpulsen oder Jitter führen.
Lösung: Es ist viel sicherer, den aktuellen Dimmwert (`phase_cut_time`) einmalig beim Nulldurchgang (`ISR_SyncPin`) in eine lokale Variable für die aktuelle Halbwelle (z. B. `active_cut_time`) zu übernehmen. Dadurch läuft jede Halbwelle mit einem stabilen, konsistenten Wert durch.

```
#undef DEBUG
#include <Arduino.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include "esp_log.h"

// Log-Tag für die ESP-Protokollierung
static const char *TAG = "KNER";

// WLAN-Konfiguration
const char *ssid = "wlan";
const char *password = "password";

// Webserver auf Port 8080 initialisieren
AsyncWebServer server(8080);

// Pin-Definitionen
const int TriacTriggerPin = 8; // Gemeinsamer Steuer-Pin für TRIAC und Optokoppler
const int SyncPin = 18; // Eingang für das Nulldurchgangs-Taktsignal (Zero-Crossing)

// Globale Variablen
RTC_DATA_ATTR volatile int dim_value = 50; // Dimmwert in % (0 = aus, 100 = volle Leistung)
volatile unsigned long phase_cut_time = 5000; // Phasenanschnitt-Verzögerung in Mikrosekunden (µs)
volatile unsigned long active_cut_time = 5000; // Für die aktuelle Halbwelle eingefrorener Wert
volatile int isr_phase = 0; // Aktueller Zustand der ISR-Zustandsmaschine
volatile bool allowSenseInterrupt = true; // Sperr-Flag gegen Mehrfach-Trigger

// Timer- und Mutex-Variablen
hw_timer_t *timer = NULL;
portMUX_TYPE pinMux = portMUX_INITIALIZER_UNLOCKED; // Einziger Mutex für kritische Variablen-Bereiche

// Konstanten für die 50Hz-Netzversorgung (1 Periode = 20ms, 1 Halbwelle = 10ms)
const unsigned long PULSE_WIDTH = 300UL; // Dauer des Zündimpulses in µs (300 µs)
const unsigned long HALF_WAVE_US = 10000UL; // Gesamte Dauer einer Halbwelle in µs (10 ms)

/*** @brief Interrupt Service Routine (ISR) für den Nulldurchgang (SyncPin)*/
void IRAM_ATTR ISR_SyncPin(){
if (allowSenseInterrupt){
allowSenseInterrupt = false;
portENTER_CRITICAL_ISR(&pinMux);
```

```
isr_phase = 0;
// Dimmwert für diese Halbwelle einmalig einfrieren, um Race Conditions zu verhindern
active_cut_time = phase_cut_time;
digitalWrite(TriacTriggerPin, LOW);

// Sonderfall 1: 100% Leistung (Keine Verzögerung, sofort zünden)
if (active_cut_time <= 10){
digitalWrite(TriacTriggerPin, HIGH);
timerRestart(timer);
timerAlarm(timer, PULSE_WIDTH, false, 0);
isr_phase = 1;
}
// Sonderfall 2: Fast 0% Leistung (Fast am Ende der Halbwelle)
else if (active_cut_time >= HALF_WAVE_US - PULSE_WIDTH){
digitalWrite(TriacTriggerPin, LOW);
timerRestart(timer);
timerAlarm(timer, HALF_WAVE_US, false, 0);
isr_phase = 3;
}
// Normaler Dimm-Betrieb: Warten bis zum Phasenanschnitt
else{
timerRestart(timer);
timerAlarm(timer, active_cut_time, false, 0);
isr_phase = 0;
}
portEXIT_CRITICAL_ISR(&pinMux);
}
}

/*** @brief ISR für den Hardware-Timer (Phasensteuerung / Zündimpuls-Management)*/
void IRAM_ATTR ISR_Timer(){
portENTER_CRITICAL_ISR(&pinMux);
if (isr_phase == 0){
// Schritt 1: Verzögerungszeit ist abgelaufen -> TRIAC zünden
digitalWrite(TriacTriggerPin, HIGH);
timerRestart(timer);
timerAlarm(timer, PULSE_WIDTH, false, 0);
isr_phase = 1;
}
else if (isr_phase == 1){
// Schritt 2: Zündimpuls-Dauer (300 µs) vorbei -> TRIAC-Trigger ausschalten (LOW)
digitalWrite(TriacTriggerPin, LOW);
// Berechne die verbleibende Restzeit bis zum Ende der 10ms-Halbwelle anhand des eingefrorenen
Wertes
unsigned long remaining_time = HALF_WAVE_US - active_cut_time - PULSE_WIDTH;
if (remaining_time < 100) remaining_time = 100;
timerRestart(timer);
timerAlarm(timer, remaining_time, false, 0);
isr_phase = 2;
}
else{
```

```
// Schritt 3: Halbwelle komplett beendet -> Timer stoppen, auf nächsten Nulldurchgang warten
digitalWrite(TriacTriggerPin, LOW);
timerStop(timer);
isr_phase = 0;
allowSenseInterrupt = true;
}
portEXIT_CRITICAL_ISR(&pinMux);
}

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

// Pins konfigurieren
pinMode(TriacTriggerPin, OUTPUT);
digitalWrite(TriacTriggerPin, LOW);

#ifdef DEBUG
pinMode(SyncPin, INPUT_PULLDOWN);
#else
pinMode(SyncPin, INPUT_PULLUP);
#endif

// WLAN-Verbindung herstellen
Serial.println("Connecting to WiFi...");
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
delay(1000);
Serial.print(".");
}
Serial.println("\nConnected to WiFi");
Serial.print("IP address: ");
Serial.println(WiFi.localIP());

// --- Webserver Routen ---
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
String html = "<html><body>";
html += "<h1>ESP32 TRIAC Control</h1>";
html += "<p>Dimmwert: " + String(dim_value) + " %</p>";
html += "<form action=\"/setdim\" method=\"GET\">";
html += "<label for=\"dimm\">Dimmwert (0-100 %): </label>";
html += "<input type=\"number\" id=\"dimm\" name=\"dimm\" min=\"0\" max=\"100\" value=\"" +
String(dim_value) + "\">";
html += "<input type=\"submit\" value=\"Set Dimmwert\">";
html += "</form>";
html += "</body></html>";
request->send(200, "text/html", html);
});

server.on("/setdim", HTTP_GET, [](AsyncWebServerRequest *request){
if (request->hasParam("dimm")) {
String dimmParam = request->getParam("dimm")->value();
```

```
int new_dim_value = dimmParam.toInt();

if (new_dim_value >= 0 && new_dim_value <= 100) {
portENTER_CRITICAL(&pinMux);
dim_value = new_dim_value;
phase_cut_time = round(HALF_WAVE_US * ((100 - dim_value) / 100.0));
portEXIT_CRITICAL(&pinMux);

Serial.print("Dimmwert auf ");
Serial.print(dim_value);
Serial.print(" % gesetzt. Delay: ");
Serial.print(phase_cut_time);
Serial.println(" µs");
} else {
Serial.println("Ungültiger Dimmwert! Bitte Wert zwischen 0 und 100 eingeben.");
}
}
request->send(200, "text/html", "ok");
request->redirect("/");
});

server.begin();
Serial.println("Server gestartet");

// Interrupt für den Nulldurchgang
attachInterrupt(digitalPinToInterrupt(SyncPin), ISR_SyncPin, FALLING);

// Hardware-Timer konfigurieren (1 MHz Takt = 1 Tick pro Mikrosekunde)
timer = timerBegin(1000000);
while (timer == NULL) {
Serial.println("Fehler: Timer ist NULL");
delay(500);
}

timerAttachInterrupt(timer, &ISR_Timer);
}

void loop(){
#ifdef DEBUG
delayMicroseconds(30000UL);
pinMode(SyncPin, INPUT_PULLUP);
pinMode(SyncPin, INPUT_PULLDOWN);
#endif
}
```

##
Ich habe das Programm überarbeitet. Zuerst die wichtigsten Probleme im Original, danach der Code.

Gefundene Probleme (Claude)

1. **Doppelte HTTP-Antwort**: `request->send(...)` und danach `request->redirect(...)` im Handler `/setdim`. Das ist ein Bug, es darf nur eine Antwort geben.
2. **Absturzgefahr beim Start**: `attachInterrupt()` lief vor `timerBegin()`. Kommt ein Nulldurchgang dazwischen, greift die ISR auf `timer == NULL` zu.
3. **Mögliches Einfrieren**: Das Flag `allowSenseInterrupt` wird nur vom Timer zurückgesetzt. Geht ein Timer-Alarm verloren, wird der Dimmer nie wieder freigegeben. Besser ist eine Zeitsperre per Zeitstempel in der ISR, das ist selbstheilend.
4. **Zu viele Zustände**: Die dritte Phase (Restzeit bis Halbwellenende) ist überflüssig, wenn die Sperre über den Zeitstempel läuft. Die Sonderfälle 0 % und 100 % sind jetzt sauber getrennt.
5. **Mutex unnötig**: Ein aligned 32-Bit-Wert ist auf dem ESP32 atomar. `portENTER_CRITICAL` in der ISR und im Webhandler kostet nur Jitter.
6. **`digitalWrite` in der ISR** ist langsam. `gpio_ll_set_level` ist inline und IRAM-sicher.
7. **`RTC_DATA_ATTR`** bringt ohne Deep Sleep nichts. Außerdem wurde `phase_cut_time` nach einem Wakeup nie aus `dim_value` berechnet.
8. **Eingabevalidierung**: `toInt()` liefert bei "abc" 0, das schaltet den Dimmer unbeabsichtigt aus.
9. **WLAN-Schleife blockiert endlos**. Der Dimmer startet jetzt zuerst und läuft auch ohne WLAN.
10. **Leistungskurve**: Ein linearer Zeitverlauf ist nicht linear in der Leistung. Optional wird per Bisektion eine Kompensation für ohmsche Lasten berechnet, außerhalb der ISR.
11. Unbenutzte Includes und `TAG` entfernt, HTML ohne String-Verkettung.

Optimierter Code

```
#include <Arduino.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include "hal/gpio_ll.h"
#include "esp_timer.h"

// ---------- Konfiguration ----------
#define SIMULATE_SYNC 0 // 1 = Nulldurchgang per Pull-Up/Down simulieren (Debug)
#define LINEARIZE_POWER 1 // 1 = Dimmwert entspricht ungefähr der Leistung (ohmsche Last)

const char *ssid = "wlan";
const char *password = "password";

constexpr uint8_t TRIAC_PIN = 8; // Zündausgang (TRIAC / Optokoppler)
constexpr uint8_t SYNC_PIN = 18; // Nulldurchgangs-Eingang
constexpr uint32_t HALF_WAVE_US = 10000; // 50 Hz -> 10 ms pro Halbwelle
constexpr uint32_t PULSE_US = 300; // Zündimpulsdauer
constexpr uint32_t MIN_SYNC_GAP_US = 7000; // Entprellung / Sperrzeit
constexpr uint32_t MAX_DELAY_US = HALF_WAVE_US - 2 * PULSE_US; // darüber = aus
constexpr uint32_t MIN_DELAY_US = 10; // darunter = sofort zünden

// ---------- Globale Zustände ----------
AsyncWebServer server(8080);
hw_timer_t *timer = nullptr;

volatile uint32_t g_delay_us = HALF_WAVE_US / 2; // Phasenanschnitt in µs
```

```
volatile int g_dim = 50; // Dimmwert in %
volatile uint8_t g_phase = 0; // 0 = warte auf Zündung, 1 = Impuls läuft
volatile int64_t g_lastSync = 0;

// ---------- Hilfsfunktionen ----------
static inline void IRAM_ATTR triac(bool on) {
gpio_ll_set_level(&GPIO, TRIAC_PIN, on ? 1 : 0);
}

// Dimmwert (%) -> Verzögerung (µs). Läuft nur im normalen Kontext, nie in der ISR.
uint32_t dimToDelayUs(int dim) {
if (dim <= 0) return HALF_WAVE_US; // aus
if (dim >= 100) return 0; // volle Leistung
float target = dim / 100.0f;
#if LINEARIZE_POWER
// Leistung P(a) = 1 - a/PI + sin(2a)/(2PI), a = Zündwinkel; Umkehrung per Bisektion
float lo = 0.0f, hi = PI;
for (int i = 0; i < 20; i++) {
float mid = 0.5f * (lo + hi);
float p = 1.0f - mid / PI + sinf(2.0f * mid) / (2.0f * PI);
if (p > target) lo = mid; else hi = mid;
}
return (uint32_t)(lo / PI * HALF_WAVE_US);
#else
return (uint32_t)roundf(HALF_WAVE_US * (1.0f - target));
#endif
}

void setDim(int dim) {
g_dim = dim;
g_delay_us = dimToDelayUs(dim); // 32-Bit-Schreibzugriff ist atomar
Serial.printf("Dimmwert %d %% -> Delay %lu us\n", dim, (unsigned long)g_delay_us);
}

// ---------- ISRs ----------
void IRAM_ATTR isrSync() {
int64_t now = esp_timer_get_time();
if (now - g_lastSync < MIN_SYNC_GAP_US) return; // Störimpuls / Doppeltrigger
g_lastSync = now;

triac(false);
g_phase = 0;

uint32_t d = g_delay_us;
if (d >= MAX_DELAY_US) return; // praktisch aus: nichts zünden
if (d < MIN_DELAY_US) d = MIN_DELAY_US;

timerRestart(timer);
timerAlarm(timer, d, false, 0);
}

void IRAM_ATTR isrTimer() {
if (g_phase == 0) { // Verzögerung vorbei -> zünden
triac(true);
g_phase = 1;
timerRestart(timer);
timerAlarm(timer, PULSE_US, false, 0);
} else { // Impuls vorbei -> Gate aus
triac(false);
}
```

```
}

// ---------- Webseite ----------
static const char INDEX_HTML[] PROGMEM = R"rawliteral(
<!DOCTYPE html><html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>ESP32 TRIAC</title></head><body>
<h1>ESP32 TRIAC Control</h1>
<p>Dimmwert: %DIM% %</p>
<form action="/setdim" method="GET">
<label for="dimm">Dimmwert (0-100 %): </label>
<input type="number" id="dimm" name="dimm" min="0" max="100" value="%DIM%">
<input type="submit" value="Setzen">
</form></body></html>
)rawliteral";

void setupWeb() {
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
request->send(200, "text/html", INDEX_HTML,
[](const String &var) -> String {
return var == "DIM" ? String(g_dim) : String();
});
});

server.on("/setdim", HTTP_GET, [](AsyncWebServerRequest *request) {
if (!request->hasParam("dimm")) {
request->send(400, "text/plain", "Parameter 'dimm' fehlt");
return;
}
String s = request->getParam("dimm")->value();
s.trim();
bool numeric = s.length() > 0 && s.length() <= 3;
for (size_t i = 0; numeric && i < s.length(); i++) numeric = isDigit(s[i]);
int v = numeric ? s.toInt() : -1;

if (v < 0 || v > 100) {
request->send(400, "text/plain", "Ungueltiger Dimmwert (0-100)");
return;
}
setDim(v);
request->redirect("/");
});

server.begin();
}

// ---------- Setup / Loop ----------
void setup() {
Serial.begin(115200);

pinMode(TRIAC_PIN, OUTPUT);
digitalWrite(TRIAC_PIN, LOW);
pinMode(SYNC_PIN, SIMULATE_SYNC ? INPUT_PULLDOWN : INPUT_PULLUP);

// Dimmer zuerst starten (Timer VOR dem Interrupt!), läuft auch ohne WLAN
timer = timerBegin(1000000); // 1 MHz = 1 Tick pro µs
while (timer == nullptr) {
Serial.println("Fehler: Timer ist NULL");
delay(500);
}
```

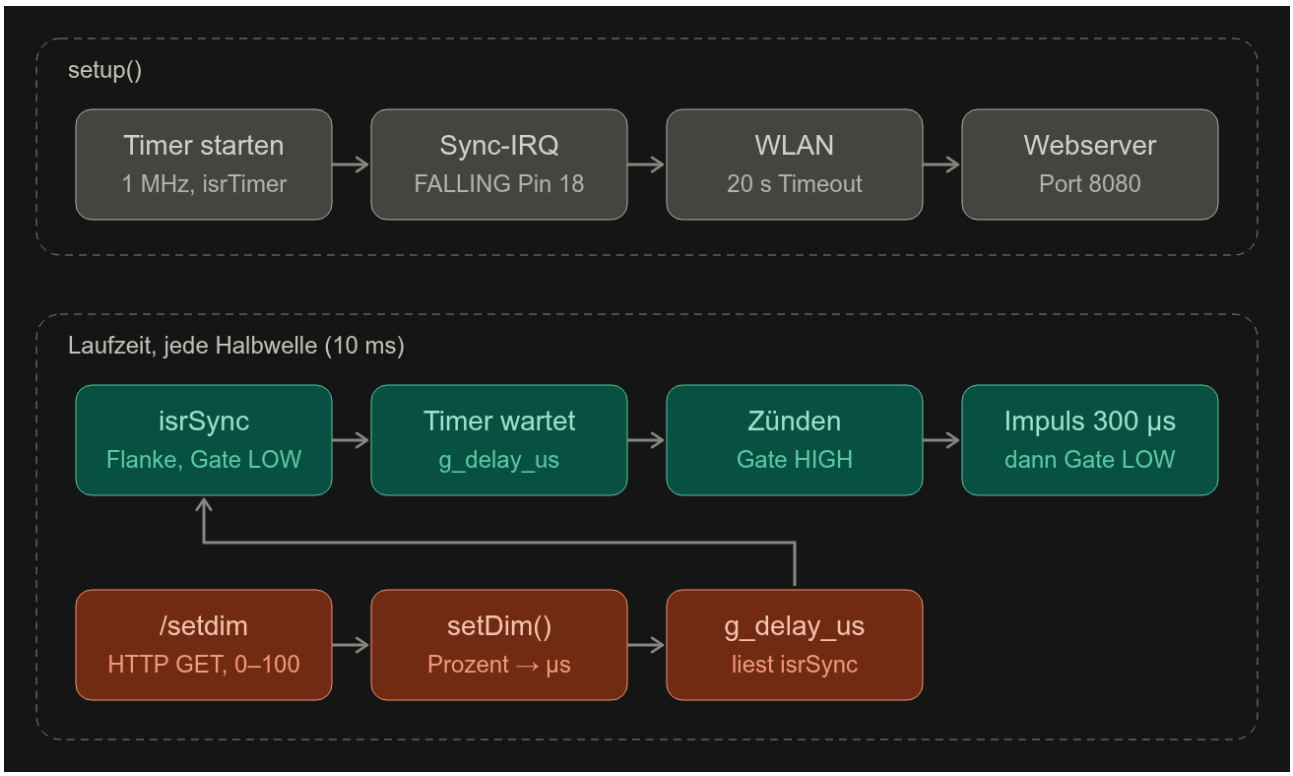
```
timerAttachInterrupt(timer, &isrTimer);
setDim(g_dim);
attachInterrupt(digitalPinToInterrupt(SYNC_PIN), isrSync, FALLING);

// WLAN mit Timeout
WiFi.mode(WIFI_STA);
WiFi.setAutoReconnect(true);
WiFi.setSleep(false); // weniger Latenz/Jitter
WiFi.begin(ssid, password);
Serial.print("Connecting to WiFi");
uint32_t t0 = millis();
while (WiFi.status() != WL_CONNECTED && millis() - t0 < 20000) {
delay(500);
Serial.print(".");
}
if (WiFi.status() == WL_CONNECTED) {
Serial.print("\nIP address: ");
Serial.println(WiFi.localIP());
} else {
Serial.println("\nWLAN nicht verbunden, Dimmer laeuft trotzdem weiter");
}

setupWeb();
Serial.println("Server gestartet");
}

void loop() {
#if SIMULATE_SYNC
delay(10);
pinMode(SYNC_PIN, INPUT_PULLUP);
pinMode(SYNC_PIN, INPUT_PULLDOWN); // erzeugt eine fallende Flanke
#else
delay(1000);
#endif
}
```

Hinweise zur Hardware

- **Offset des Nulldurchgangs**: Optokoppler-Detektoren melden die Flanke meist etwas vor oder nach dem echten Nulldurchgang (oft 100 bis 500 µs). Das lässt sich am Oszilloskop messen und in `isrSync` zum Delay addieren.
- **Dimmwert speichern**: Soll er einen Neustart überleben, nimm `Preferences` und speichere ihn verzögert, zum Beispiel 5 Sekunden nach der letzten Änderung, damit der Flash nicht verschleißt.
- **Zeitbasis des Sync-Signals**: Der Code geht von einer Flanke pro Halbwelle aus (Brückengleichrichter am Detektor). Kommt nur eine Flanke pro Vollwelle, musst du `HALF_WAVE_US` und die Zündlogik anpassen oder beide Halbwellen separat behandeln.
- **Sicherheit**: Netzspannung ist lebensgefährlich. Das ESP32 muss sauber galvanisch getrennt sein, zum Beispiel mit Optokoppler MOC3021 oder ähnlichem. Bei induktiven Lasten brauchst du ein Snubber-Glied.

Wenn du mir sagst, welches Board du nutzt (Pin 8 und 18 deuten auf C3 oder S3 hin) und welche Last du dimmst, kann ich die Parameter noch genauer abstimmen.

Hier die Skizze: oben der Start, darunter der Ablauf pro Halbwelle und die Verbindung zum Webserver.



Der Ablauf in kurzen Worten:

- **Grau (`setup()`)**: Timer und Sync-Interrupt werden zuerst gestartet, damit der Dimmer auch ohne WLAN läuft. Danach folgen WLAN und Webserver.
- **Türkis (Laufzeit)**: Bei jeder fallenden Flanke an Pin 18 sperrt `isrSync` das Gate und startet den Timer mit `g_delay_us`. Läuft der Timer ab, zündet `isrTimer` den TRIAC. Nach 300 µs setzt er das Gate wieder auf LOW. Der TRIAC hält dann selbst bis zum nächsten Nulldurchgang.
- **Koralle (Web)**: `/setdim` rechnet den Prozentwert in Mikrosekunden um und schreibt ihn in `g_delay_us`. Das ist die einzige Verbindung zwischen Webserver und ISR. Die nächste Halbwelle übernimmt den neuen Wert.

ChatGPT

Verbessert wurden:

- Timer-Zustände und Schutz vor veralteten Alarmen.
- Synchronisierung zwischen Webserver und Interrupts.
- Sofortiges Verwerfen geplanter Zündungen bei 0 %.
- Leistungslinearisierung, WLAN-Wiederverbindung und Eingabeprüfung.
- Sync-Simulation über einen eigenen Hardware-Timer.

Die Timer-Nutzung folgt der Espressif-API.

```
// Arduino-ESP32 3.x
// Libraries: ESPAsyncWebServer + AsyncTCP
// 50 Hz mains, one FALLING edge per zero crossing.
// TRIAC gate output is active HIGH.
```

```
#include <Arduino.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <esp_arduino_version.h>
#include <math.h>

#ifndef LINEARIZE_POWER
#define LINEARIZE_POWER 1
#endif

#ifndef SIMULATE_SYNC
// Test only, with the load disconnected.
// Simulated crossings are NOT synchronized with mains.
#define SIMULATE_SYNC 0
#endif

#if ESP_ARDUINO_VERSION_MAJOR < 3
#error "This sketch requires Arduino-ESP32 3.x."
#endif

// ---------- Configuration ----------

constexpr char WIFI_SSID[] = "wlan";
constexpr char WIFI_PASSWORD[] = "password";

constexpr uint8_t TRIAC_PIN = 8;
constexpr uint8_t SYNC_PIN = 18;

constexpr uint32_t HALF_WAVE_US = 10000; // 50 Hz
constexpr uint32_t PULSE_US = 300;
constexpr uint32_t END_MARGIN_US = 100;

constexpr uint32_t LATEST_FIRE_US =
HALF_WAVE_US - PULSE_US - END_MARGIN_US;

constexpr uint32_t MAX_DELAY_US =
LATEST_FIRE_US - END_MARGIN_US; // Additional ISR latency margin

constexpr uint32_t SYNC_FILTER_US = 8000;

// Matches the original sketch. Set to 0 to start switched off.
constexpr int INITIAL_DIM = 50;

static_assert(TRIAC_PIN != SYNC_PIN,
"TRIAC and SYNC pins must be different.");

static_assert(PULSE_US + 2 * END_MARGIN_US < HALF_WAVE_US,
"Pulse duration and margins are too long.");

static_assert(SYNC_FILTER_US < HALF_WAVE_US,
"Sync filter must be shorter than a half-wave.");
```

```
// ---------- Shared state ----------

AsyncWebServer server(8080);

hw_timer_t *phaseTimer = nullptr;
portMUX_TYPE stateMux = portMUX_INITIALIZER_UNLOCKED;

// Access shared state under stateMux after initialization.
int dimValue = INITIAL_DIM;
uint32_t requestedDelayUs = MAX_DELAY_US / 2;

enum class Phase : uint8_t {
Idle,
WaitFire,
Pulse
};

Phase phase = Phase::Idle;

uint64_t nextAlarmUs = 0;
uint64_t lastSyncUs = 0;
bool haveSync = false;

// ---------- Interrupt handlers ----------

// Call only while holding stateMux.
// The timer runs continuously; alarm times are absolute.
void ARDUINO_ISR_ATTR armAlarm(uint64_t atUs) {
nextAlarmUs = atUs;
timerAlarm(phaseTimer, atUs, false, 0);
}

void ARDUINO_ISR_ATTR onZeroCross() {
portENTER_CRITICAL_ISR(&stateMux);

const uint64_t now = timerRead(phaseTimer);

// Reject closely spaced duplicate edges.
if (haveSync && now - lastSyncUs < SYNC_FILTER_US) {
portEXIT_CRITICAL_ISR(&stateMux);
return;
}

haveSync = true;
lastSyncUs = now;

digitalWrite(TRIAC_PIN, LOW);
phase = Phase::Idle;

if (dimValue > 0) {
```

```
if (requestedDelayUs == 0) {
// Full power: start the pulse immediately.
digitalWrite(TRIAC_PIN, HIGH);
phase = Phase::Pulse;
armAlarm(timerRead(phaseTimer) + PULSE_US);
} else {
phase = Phase::WaitFire;
armAlarm(now + requestedDelayUs);
}
}

portEXIT_CRITICAL_ISR(&stateMux);
}

void ARDUINO_ISR_ATTR onTimer() {
portENTER_CRITICAL_ISR(&stateMux);

const uint64_t now = timerRead(phaseTimer);

// Ignore inactive or stale callbacks that arrive before the
// currently scheduled alarm.
if (phase == Phase::Idle || now < nextAlarmUs) {
portEXIT_CRITICAL_ISR(&stateMux);
return;
}

if (phase == Phase::WaitFire) {
// Skip a firing event that has become too late.
if (now - lastSyncUs > LATEST_FIRE_US) {
digitalWrite(TRIAC_PIN, LOW);
phase = Phase::Idle;
} else {
digitalWrite(TRIAC_PIN, HIGH);
phase = Phase::Pulse;
armAlarm(timerRead(phaseTimer) + PULSE_US);
}
} else {
// End the gate pulse.
digitalWrite(TRIAC_PIN, LOW);
phase = Phase::Idle;
}

portEXIT_CRITICAL_ISR(&stateMux);
}

// ---------- Dimming ----------

// Calculate only in task context, never inside an ISR.
uint32_t dimToDelayUs(int value) {
if (value <= 0) {
return HALF_WAVE_US;
```

```
}

if (value >= 100) {
return 0;
}

#if LINEARIZE_POWER
// Resistive-load normalized power:
// P(a) = 1 - a/pi + sin(2a)/(2pi)
// Invert using bisection.
constexpr float pi = 3.14159265358979323846f;
const float target = value / 100.0f;

float lo = 0.0f;
float hi = pi;

for (unsigned i = 0; i < 20; ++i) {
const float angle = (lo + hi) * 0.5f;

const float power =
1.0f - angle / pi +
sinf(2.0f * angle) / (2.0f * pi);

if (power > target) {
lo = angle;
} else {
hi = angle;
}
}

const uint32_t result =
lroundf((lo + hi) * 0.5f / pi * HALF_WAVE_US);
#else
const uint32_t result =
(HALF_WAVE_US * (100U - value) + 50U) / 100U;
#endif

// Leave room for pulse duration and interrupt latency.
return result > MAX_DELAY_US ? MAX_DELAY_US : result;
}

void setDim(int value) {
value = value < 0 ? 0 : (value > 100 ? 100 : value);

// Keep floating-point work outside the critical section.
const uint32_t delayUs = dimToDelayUs(value);

portENTER_CRITICAL(&stateMux);

dimValue = value;
requestedDelayUs = delayUs;
```

```
if (value == 0) {
// Cancel any pending firing and immediately lower the gate.
// An already conducting TRIAC turns off at current zero.
phase = Phase::Idle;
digitalWrite(TRIAC_PIN, LOW);
}

portEXIT_CRITICAL(&stateMux);
}

int readDim() {
portENTER_CRITICAL(&stateMux);
const int value = dimValue;
portEXIT_CRITICAL(&stateMux);

return value;
}

// Strict integer parsing: reject partial numbers and extra characters.
bool parseDim(const String &text, int &value) {
if (text.length() == 0 || text.length() > 3) {
return false;
}

int result = 0;

for (size_t i = 0; i < text.length(); ++i) {
const char c = text[i];

if (c < '0' || c > '9') {
return false;
}

result = result * 10 + (c - '0');
}

if (result > 100) {
return false;
}

value = result;
return true;
}

// ---------- Web page ----------

// Encode literal percent signs so they do not conflict with
// ESPAsyncWebServer's %PLACEHOLDER% template syntax.
static const char PAGE[] PROGMEM = R"HTML(
<!doctype html>
```

```
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>ESP32 TRIAC</title>
</head>
<body>
<h1>ESP32 TRIAC Control</h1>
<p>Dimming: %DIM% &#37;</p>

<form action="/setdim" method="post">
<label for="dimm">Dimming (0–100 &#37;): </label>
<input
type="number"
id="dimm"
name="dimm"
min="0"
max="100"
step="1"
value="%DIM%"
required
>
<button type="submit">Apply</button>
</form>
</body>
</html>
)HTML";

// ---------- Setup ----------

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

pinMode(TRIAC_PIN, OUTPUT);
digitalWrite(TRIAC_PIN, LOW);
pinMode(SYNC_PIN, INPUT_PULLUP);

setDim(INITIAL_DIM);

// Initialize the phase timer before enabling sync interrupts.
phaseTimer = timerBegin(1000000); // 1 tick = 1 microsecond

if (phaseTimer == nullptr) {
Serial.println(
"Phase timer initialization failed; output remains LOW."
);

while (true) {
delay(1000);
}
}
```

```
timerAttachInterrupt(phaseTimer, &onTimer);

#if SIMULATE_SYNC
// A separate hardware timer supplies simulated zero crossings.
hw_timer_t *syncTimer = timerBegin(1000000);

if (syncTimer == nullptr) {
Serial.println(
"Simulation timer initialization failed; output remains LOW."
);

while (true) {
delay(1000);
}
}

timerAttachInterrupt(syncTimer, &onZeroCross);
timerAlarm(syncTimer, HALF_WAVE_US, true, 0);
#else
attachInterrupt(
digitalPinToInterrupt(SYNC_PIN),
onZeroCross,
FALLING
);
#endif

server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
// Both template substitutions use the same snapshot.
const int snapshot = readDim();

request->send_P(
200,
"text/html; charset=utf-8",
PAGE,
[snapshot](const String &key) -> String {
return key == "DIM" ? String(snapshot) : String();
}
);
});

server.on("/setdim", HTTP_POST,
[](AsyncWebServerRequest *request) {
int value;

if (!request->hasParam("dimm", true) ||
!parseDim(request->getParam("dimm", true)->value(), value)) {
request->send(
400,
"text/plain; charset=utf-8",
"dimm must be an integer from 0 to 100."
```

```
);
return;
}

setDim(value);

// POST/Redirect/GET prevents resubmission on page refresh.
AsyncWebServerResponse *response =
request->beginResponse(303);

response->addHeader("Location", "/");
response->addHeader("Cache-Control", "no-store");
request->send(response);
});

server.onNotFound([](AsyncWebServerRequest *request) {
request->send(
404,
"text/plain; charset=utf-8",
"Not found"
);
});

// Connect without blocking the dimmer startup.
WiFi.mode(WIFI_STA);
WiFi.setAutoReconnect(true);
WiFi.setSleep(false);
WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

server.begin();

Serial.println(
"Dimmer active; Wi-Fi connection running in the background."
);
}

// ---------- Main loop ----------

void loop() {
static bool wasConnected = false;
static uint32_t lastRetryMs = 0;

const bool connected = WiFi.status() == WL_CONNECTED;

if (connected && !wasConnected) {
Serial.print("Web interface: http://");
Serial.print(WiFi.localIP());
Serial.println(":8080/");
}

if (!connected && millis() - lastRetryMs >= 30000U) {
```

```
lastRetryMs = millis();
WiFi.reconnect();
}

wasConnected = connected;
delay(100);
}
```