TETRIS FÜR ESP32-S3
PROJEKT-DOKUMENTATION
Embedded Systems Building Blocks - EIT-DU Studium

Autor: Mathias Lampert
Datum: Dezember 2025
Mikrocontroller: ESP32-S3 DevKitC-1 N16R8
Entwicklungszeit: ~131 Stunden
Code-Umfang: ~2500 Zeilen Code + ~2000 Zeilen Kommentare

1. PROJEKTZIEL UND AUSGANGSSITUATION

1.1 URSPRÜNGLICHE PLANUNG

Das Projekt sollte ursprünglich ein minimalistisches Tetris-Spiel für den ESP32-S3 werden mit folgenden Anforderungen:

GEPLANTES GAMEPLAY:
- Nur horizontale Bewegung (Links/Rechts) - KEINE Rotation
- Spielfeld: 10×20 Raster
- Automatischer Steinfall
- 3 Tasten: Links, Rechts, Start/Pause
- Einfaches Punktesystem

GEPLANTE AUSGABE-OPTIONEN:
- OLED-Display (SSD1306, 128×64, I²C) ODER
- Flutter-App über BLE (Bluetooth Low Energy) ODER
- LED-Matrix (als optionale Erweiterung)

GEPLANTE LERNZIELE:
- Objektorientierte Programmierung in C (Structs + Funktionszeiger)
- BLE-Kommunikation zwischen ESP32 und Smartphone
- Echtzeitsteuerung auf Embedded-Hardware
- Speicher- und Ressourcenmanagement

1.2 TATSÄCHLICHE UMSETZUNG

Im Verlauf der Entwicklung wurde das Projekt deutlich erweitert und professionalisiert. Die finale Version ist ein vollwertiges Tetris-Spiel:

UMGESETZTES GAMEPLAY:
- ☑ Vollständige Rotation aller 7 Tetromino-Formen (außer O-Block)
- ☑ Spielfeld: 16×24 Pixel (angepasst an Hardware)
- ☑ 4 Tasten: Links, Rechts, Rotieren, Schneller fallen
- ☑ 10 Geschwindigkeits-Level (400ms → 50ms Fall-Intervall)
- ☑ Multi-Zeilen-Clearing (1-4 Zeilen gleichzeitig)
- ☑ Punktesystem mit persistent gespeichertem Highscore
- ☑ Splash-Screen mit scrollendem "TETRIS"-Text
- ☑ Game-Over-Animation (3× rot blinken)
- ☑ Emergency-Reset via 4-Button-Kombination

UMGESETZTE AUSGABE:
- ☑ LED-Matrix: 6× CJMCU 8×8 WS2812 Panels (384 LEDs, 16×24 Pixel)
- ☑ OLED-Display: SSD1306 für HUD (Score/Lines/Level/Highscore)
- ☑ Gleichzeitige Nutzung beider Displays
- ✗ BLE/Flutter: Nicht implementiert (Fokus auf lokale Performance)

ZUSÄTZLICHE FEATURES:
- ☑ Tetris-Theme-Song über passiven Buzzer (68 Noten, Endlos-Loop)
- ☑ 60 FPS flimmerfreies Rendering (Differential Rendering)
- ☑ Interrupt-basierte Button-Steuerung (ISR + Event-Queue)
- ☑ Smart Block-Spawning (versucht mehrere Positionen)
- ☑ NVS-basierte Highscore-Speicherung (überlebt Stromausfall)

TECHNISCHE ENTSCHEIDUNGEN:
- ➜ Modulare C-Struktur statt OOP mit Funktionszeigern (wartbarer)
- ➜ FreeRTOS-Tasks für parallele Ausführung (GameLoop + Audio)
- ➜ Hardware-Peripherals: RMT für WS2812-Timing + Audio-PWM
- ➜ LVGL für Display-Grafikausgabe

2. VERWENDETE HARDWARE

2.1 MIKROCONTROLLER: ESP32-S3 DEVKITC-1 N16R8

Hersteller: diymore (2PCS Packung)
Prozessor: Xtensa LX7 Dual-Core @ 240 MHz
Flash-Speicher: 16 MB
PSRAM: 8 MB
USB: Native USB (für Programmierung und Debugging)

VORTEILE FÜR PROJEKT:
- Genug Speicher für LED-Framebuffer (384 LEDs × 3 Bytes = 1152 Bytes)
- FreeRTOS für parallele Tasks (Spiel + Musik gleichzeitig)
- RMT-Hardware-Peripheral (kritisch für WS2812-Timing)
- NVS (Non-Volatile Storage) für Highscore-Persistierung
- Ausreichend GPIOs für alle Komponenten

GPIO-BELEGUNG:
  GPIO 38: WS2812 LED-Matrix (RMT Channel 0)
  GPIO 17: Passiver Buzzer (RMT Channel 1, PWM für Tonhöhen)
  GPIO 18: Button Links (Pull-up, Interrupt auf fallende Flanke)
  GPIO 19: Button Rechts (Pull-up, Interrupt auf fallende Flanke)
  GPIO 20: Button Rotieren (Pull-up, Interrupt auf fallende Flanke)
  GPIO 21: Button Schneller (Pull-up, Interrupt auf fallende Flanke)
  GPIO 4:   I²C SDA (OLED-Display)
  GPIO 5:   I²C SCL (OLED-Display)

2.2 LED-MATRIX: 6× CJMCU-8×8 WS2812 PANELS

Bezeichnung: U 64 LED Matrix Panel CJMCU-8x8 Modul
LED-Typ: WS2812 5050 RGB (individuell addressierbar)
LEDs pro Panel: 64 (8 Zeilen × 8 Spalten)
Anzahl Panels: 6 Stück
Gesamt-Layout: 2 Zeilen × 3 Spalten = 16 Pixel hoch × 24 Pixel breit
Gesamt-LEDs: 384 (16 × 24)
Spannung: 5V (über externe Stromversorgung)
Protokoll: WS2812 (800 kHz, Timing-kritisch)

VERKABELUNG (SERPENTINEN-MUSTER):
  Panel-Reihenfolge: 1 -> 2 -> 3 -> 4 -> 5 -> 6

  [Panel 6] ← [Panel 5]
                  ↑
  [Panel 3] → [Panel 4]
     ↑
  [Panel 2] ← [Panel 1]

  Innerhalb jeder 8×8-Matrix: Serpentinen-Muster
  - Zeile 0: Links → Rechts
  - Zeile 1: Rechts → Links
  - Zeile 2: Links → Rechts
  - usw.

HERAUSFORDERUNG:
  Komplexes Mapping von logischen Koordinaten (y, x) zu LED-Index nötig!
  Beispiel: Pixel an Position (5, 10) entspricht LED Nummer 234

2.3 OLED-DISPLAY: SSD1306

Typ: OLED (Monochrom)
Auflösung: 128×64 Pixel (rechteckig)
Controller: SSD1306
Kommunikation: I²C (Adresse 0x3C)
Spannung: 3.3V
Größe: ca. 1.5 Zoll diagonal

VERWENDUNGSZWECK:
- Echtzeit-HUD während des Spiels:
  - SCORE: aktueller Punktestand
  - HIGH: gespeicherter Highscore
- Game-Over-Screen mit Score-Vergleich

GRAFIKBIBLIOTHEK:
- LVGL (Light and Versatile Graphics Library) v8.4
- ESP-IDF-Integration über espressif__esp_lvgl_port

2.4 EINGABE: 4 TASTER + BUZZER

TASTER (4 Stück):

```
  Typ:        Standard Taster (Normally Open)
  Spannung:   3.3V (intern Pull-up)
  Debouncing: 50ms Software-Debounce in ISR

  Funktion:
    - Links:     Block nach links bewegen
    - Rechts:    Block nach rechts bewegen
    - Rotieren:  Block um 90° im Uhrzeigersinn drehen
    - Schneller: Manuelles schnelleres Fallen

PASSIVER BUZZER:
  GPIO:           17
  Frequenzbereich: 100 Hz - 5 kHz
  Implementierung: RMT-basierte PWM (variable Frequenz)
  Verwendung:     Tetris-Theme-Song (Korobeiniki)
  Noten:          68 Noten, Gesamt-Länge ~30 Sekunden
  Loop:           Endlos-Wiederholung im Hintergrund-Task


2.5 STROMVERSORGUNG
-------------------
ESP32:        USB-Stromversorgung (5V, ~200 mA)
LED-Matrix:   Externe 5V-Stromversorgung (bis 23 A bei voller Helligkeit!)
              Tatsächlicher Verbrauch: ~2 A bei reduzierter Helligkeit
OLED-Display: Über ESP32 3.3V (ca. 20 mA)

WICHTIG: LED-Matrix und ESP32 teilen sich GND (gemeinsame Masse)!


================================================================================
3. SOFTWARE-ARCHITEKTUR
================================================================================

3.1 ENTWICKLUNGSUMGEBUNG
------------------------
Framework:      ESP-IDF v5.x (Espressif IoT Development Framework)
Toolchain:      Xtensa GCC Compiler
Build-System:   CMake
Programmiersprache: C (C99 Standard)
Betriebssystem: FreeRTOS (integriert in ESP-IDF)

EXTERNE ABHÄNGIGKEITEN (ESP Component Registry):
  • espressif__led_strip ^2.5.5        (WS2812-Treiber)
  • espressif__esp_lcd_SSD1306 ^1.0.1   (OLED-Treiber)
  • espressif__esp_lvgl_port ^2.4.6    (LVGL-Integration)
  • lvgl__lvgl ^8.4.0                  (GUI-Bibliothek)


3.2 MODULARE CODE-STRUKTUR (13 MODULE)
--------------------------------------
Modul               Dateien                  Beschreibung
--------------------------------------------------------------------------------
Main                main.c                   App-Einstiegspunkt, Initialisierung
Controls            Controls.c/h             Button-ISR, Event-Queue, Debouncing
GameLoop            GameLoop_NEW.c/h         Spielschleife, State Machine
Blocks              Blocks.c/h               Tetromino-Formen, Rotation
Grid                Grid.c/h                 Spielfeld, Kollision, Zeilen löschen
MatrixNummer        MatrixNummer.c/h         LED-Mapping-Daten (Lookup-Tabelle)
LedMatrixInit       LedMatrixInit.c/h        LED-Mapping-Initialisierung
BlockColors         Colors.c                 RGB-Farbtabelle für 7 Block-Typen
Score               Score.c/h                Punkteberechnung, NVS-Highscore
SpeedManager        SpeedManager.c/h         Geschwindigkeits-Level-Verwaltung
DisplayInit         DisplayInit.c/h          OLED I²C + LVGL Setup
Splash              Splash.c/h               Startbildschirm-Animationen
ThemeSong           ThemeSong.c/h            Tetris-Theme (Musik-Task)
                    musical_score_encoder.c/h RMT-Encoder für Tonhöhen
Globals             Globals.h                Konstanten, Makros, Typdefinitionen

GESAMTSTATISTIK:
  14 C-Dateien (ohne main.c)
  13 Header-Dateien
  ~2500 Zeilen Code (ohne Kommentare)
  ~2000 Zeilen Kommentare (Doxygen-Style)


3.3 DATENSTRUKTUREN
-------------------

TETRISBLOCK (aktuell fallender Stein):
  typedef struct {
      uint8_t shape[4][4];  // 1 = Blockteil, 0 = leer
```

```
      int x;               // Position: linke obere Ecke (Grid-Koordinaten)
      int y;               // Position: obere Kante
      uint8_t color;       // Farbindex (0-6: I, J, L, O, S, T, Z)
  } TetrisBlock;

MATRIX (LED-Mapping):
  typedef struct {
      uint16_t LED_Number[LED_HEIGHT][LED_WIDTH];  // [16][24]
  } MATRIX;

  Lookup-Tabelle: logische Position → physikalische LED-Nummer
  Beispiel: ledMatrix.LED_Number[5][10] = 234

SPIELFELD (fixierte Blöcke):
  uint8_t grid[GRID_HEIGHT][GRID_WIDTH];  // [16][24]
  Werte:
    0   = leer
    1-7 = fixierter Block (Farbindex + 1)


3.4 FREERTOS-TASK-ARCHITEKTUR
------------------------------
Task              Priorität   Stack   Funktion
--------------------------------------------------------------------------------
GameLoopTask      5 (hoch)    4096    Spielschleife (Input, Physics, Rendering)
ThemeTask         3 (mittel)  2048    Hintergrundmusik (Endlos-Loop)
LVGL Timer Task   1 (niedrig) auto    UI-Updates für OLED-Display

TASK-KOMMUNIKATION:
  • Controls → GameLoop: FreeRTOS Queue (Button-Events aus ISR)
  • GameLoop → Display: Direkte Funktionsaufrufe (display_update_score)
  • Keine Kommunikation zwischen GameLoop und Theme (beide unabhängig)


3.5 STATE MACHINE (SPIELABLAUF)
--------------------------------

┌──────────────────────────────────────────────────────────────────────────┐
│                              GAME STATES                                 │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│   [WAIT]                                                                 │
│     │  (Splash scrollt endlos "TETRIS")                                  │
│     │  Warte auf beliebigen Button-Press                                 │
│     ↓                                                                    │
│   [Button gedrückt]                                                      │
│     │  → Queue leeren + Button-Release abwarten                          │
│     ↓                                                                    │
│   [RUNNING] ←──────────────────────────┐                                 │
│     │  • Input-Verarbeitung (5ms Poll) │                                 │
│     │  • Automatischer Fall (Level-abhängig)                             │
│     │  • Rendering (60 FPS)            │                                 │
│     │  • Kollisionsprüfung             │                                 │
│     │                                  │                                 │
│     ├─ Block fixiert → Neuen Block spawnen                               │
│     │                                  │                                 │
│     └─ Kein Platz für neuen Block      │                                 │
│        ↓                               │                                 │
│   [GAME_OVER]                          │                                 │
│     │  • Highscore aktualisieren (falls besser)                          │
│     │  • Game-Over auf Display anzeigen │                                │
│     │  • 3× rot blinken (Animation)     │                                │
│     │  • Flag setzen: game_over_flag = 1│                                │
│     ↓                                   │                                │
│   [Neustart-Sequenz]                    │                                │
│     │  1. Queue drainieren              │                                │
│     │  2. Splash 2s (Inputs ignoriert)  │                                │
│     │  3. 500ms kontinuierlich Queue drainieren                          │
│     │  4. Button-Release abwarten       │                                │
│     │  5. Interactive Splash (warte auf Button)                          │
│     │  6. Button-Release abwarten       │                                │
│     │  7. Game starten ─────────────────┘                                │
│     │                                                                    │
│   EMERGENCY RESET (4 Buttons gleichzeitig):                              │
│     [ANY STATE] → 1s halten → [Hard Reset] → [Neustart-Sequenz]          │
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘


3.6 TIMING-PARAMETER
--------------------
```

Parameter	Wert (ms)	Beschreibung
LOOP_INTERVAL_MS	5	GameLoop-Polling (responsive Input)
RENDER_INTERVAL_MS	16	60 FPS Rendering (flimmerfrei)
FALL_INTERVAL_MS	400	Initial-Geschwindigkeit (Level 0)
FALL_INTERVAL_MS_MIN	50	Maximum-Geschwindigkeit (Level 10)
DEBOUNCE_MS	50	Button-Debounce-Zeit
SPLASH_DURATION_MS	2000	Minimum Splash-Anzeige
GAME_OVER_BLINK_ON_MS	300	Blink-Animation: rot
GAME_OVER_BLINK_OFF_MS	300	Blink-Animation: schwarz
GAME_OVER_BLINK_COUNT	3	Anzahl Wiederholungen

4. IMPLEMENTIERUNGS-HIGHLIGHTS

4.1 DIFFERENTIAL RENDERING (60 FPS OHNE FLACKERN)

PROBLEM:
  Naive Implementierung: led_strip_clear() + alles neu zeichnen pro Frame
  → Sichtbares Flackern (384 LEDs brauchen ~5ms zum Updaten)
  → CPU-Last: 1152 Bytes schreiben pro Frame

LÖSUNG:
  1. Vorherige Position des dynamischen Blocks merken
  2. Nur diese Pixel zurück auf statische Farben setzen
  3. Neuen Block an neuer Position zeichnen
  4. Nur geänderte Pixel werden aktualisiert

DATENSTRUKTUR:

```
  static int prev_dynamic_count = 0;
  static int prev_dynamic_pos[GRID_WIDTH * GRID_HEIGHT][2];  // [y, x]
```

ALGORITHMUS:

```
  // Frame N: Restauriere Frame N-1
  for (i = 0; i < prev_dynamic_count; i++) {
      ry = prev_dynamic_pos[i][0];
      rx = prev_dynamic_pos[i][1];
      if (grid[ry][rx] > 0) {
          set_pixel_to_static_color(ry, rx);  // Aus grid[][]
      } else {
          set_pixel_to_black(ry, rx);  // Leer
      }
  }

  // Frame N: Zeichne aktuellen Block + merke Positionen
  prev_dynamic_count = 0;
  for (by, bx in current_block.shape) {
      if (shape[by][bx]) {
          set_pixel_to_block_color(gy, gx);
          prev_dynamic_pos[prev_dynamic_count++] = {gy, gx};
      }
  }

  led_strip_refresh();  // Nur 1× pro Frame!
```

RESULTAT:
  Flimmerfreies Rendering auch bei Level 10 (50ms Fall-Intervall)

4.2 INTERRUPT-BASIERTE BUTTON-STEUERUNG

PROBLEM:
  Polling allein: Verzögerung durch Loop-Timing (5ms Intervall)
  Pure ISR: Prellen führt zu Mehrfach-Events
  Blockierendes Debouncing: Unresponsive Steuerung

LÖSUNG (HYBRID-ANSATZ):
  1. GPIO-Interrupt bei fallender Flanke (Button-Press)
  2. ISR pusht Event in FreeRTOS Queue (non-blocking)
  3. ISR führt Software-Debounce aus (50ms Mindestabstand)
  4. GameLoop pollt Queue mit xQueueReceive (nicht blockierend)
  5. Für Halten-Funktion: check_button_pressed() (direkter GPIO-Read)

ARCHITEKTUR:

```
  [Button] → [GPIO IRQ] → [ISR (IRAM)] → [Queue (8 Events)] → [GameLoop]
                                ↓
                          [50ms Debounce]
```

CODE (VEREINFACHT):

```
  // ISR (läuft in IRAM, sehr schnell)
  static void IRAM_ATTR gpio_isr_handler(void* arg) {
      gpio_num_t gpio = (gpio_num_t)arg;
      uint32_t now = xTaskGetTickCountFromISR() * portTICK_PERIOD_MS;

      if (now - last_event_time < DEBOUNCE_MS) return;  // Zu schnell
      last_event_time = now;

      xQueueSendFromISR(s_button_queue, &gpio, NULL);
  }

  // GameLoop
  gpio_num_t button;
  if (controls_get_event(&button)) {  // Non-blocking
      // Event verarbeiten
  }

  // Für Halten
  if (check_button_pressed(BTN_LEFT)) {
      // Block nach links (während Button gehalten)
  }

VORTEILE:
  • Kein Event-Verlust (Queue mit 8 Events)
  • Responsive (Latenz <10ms)
  • Keine Prelleffekte
  • Niedriger CPU-Overhead


4.3 LED-MATRIX-MAPPING
----------------------
HERAUSFORDERUNG:
  6 Panels in komplexem Serpentinen-Muster verkabelt
  Innerhalb jedes Panels: Serpentine (Zeile 0: →, Zeile 1: ←, ...)
  Panel-Übergreifend: 1→2→3→4←5←6

LÖSUNG:
  Vollständige Lookup-Tabelle in MatrixNummer.c mit allen 384 LEDs

BEISPIEL (Matrix-Panel 1, unten links):
  Zeile 0 (y=0): x=0→7  entspricht LED 0→7
  Zeile 1 (y=1): x=0→7  entspricht LED 15→8 (rückwärts!)
  Zeile 2 (y=2): x=0→7  entspricht LED 16→23
  ...

INITIALISIERUNG:
  void LedMatrixInit(uint8_t height, uint8_t width,
                     uint16_t matrix[height][width]) {
      // Füllt matrix[][] mit LED-Nummern basierend auf
      // Verkabelungs-Pattern (hardcoded in MatrixNummer.c)
      for (y = 0; y < height; y++) {
          for (x = 0; x < width; x++) {
              matrix[y][x] = calculate_led_number(y, x);
          }
      }
  }

VERWENDUNG:
  int led_num = ledMatrix.LED_Number[5][10];  // Lookup
  led_strip_set_pixel(led_strip, led_num, r, g, b);

DEBUGGING-AUFWAND:
  ~8 Stunden (Verkabelung musste durch Trial-and-Error ermittelt werden)


4.4 NVS-HIGHSCORE-PERSISTIERUNG
-------------------------------
HERAUSFORDERUNG:
  Highscore muss Stromausfall überleben
  ESP32-NVS (Non-Volatile Storage) muss initialisiert werden
  Fehlerbehandlung bei Korruption nötig

IMPLEMENTIERUNG:
  // Initialisierung (in main.c)
  esp_err_t ret = nvs_flash_init();
  if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
      ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
      ESP_ERROR_CHECK(nvs_flash_erase());
      ret = nvs_flash_init();
  }
```

```
  // Highscore laden (in score_load_highscore)
  nvs_handle_t nvs_handle;
  nvs_open("storage", NVS_READONLY, &nvs_handle);
  nvs_get_u32(nvs_handle, "highscore", &highscore);
  nvs_close(nvs_handle);

  // Highscore speichern (in score_update_highscore)
  if (score > highscore) {
      nvs_open("storage", NVS_READWRITE, &nvs_handle);
      nvs_set_u32(nvs_handle, "highscore", score);
      nvs_commit(nvs_handle);  // KRITISCH: Commit nicht vergessen!
      nvs_close(nvs_handle);
  }

FEHLERBEHANDLUNG:
  Lese-Fehler → Highscore = 0 (Default)
  Schreib-Fehler → Log-Ausgabe, aber Spiel läuft weiter


4.5 AUDIO-SYSTEM (RMT-BASIERT)
------------------------------
HERAUSFORDERUNG:
  Passiver Buzzer benötigt PWM mit variabler Frequenz
  Software-PWM (vTaskDelay) ist zu ungenau
  Musik darf Spiel nicht blockieren

LÖSUNG:
  RMT-Hardware-Peripheral (Remote Control Transceiver) für präzise PWM
  Separater FreeRTOS-Task für Musik (Priorität niedriger als GameLoop)

TONHÖHEN-ARRAY (Tetris-Theme):
  const uint16_t tetris_theme_notes[] = {
      659, 494, 523, 587, 523, 494, 440, 440, 523, 659, 587, 523, 494, ...
  };  // Frequenzen in Hz (68 Noten)

  const uint16_t tetris_theme_durations[] = {
      40, 20, 20, 40, 20, 20, 40, 20, 20, 40, 20, 20, 60, ...
  };  // Dauer in 10ms-Einheiten

IMPLEMENTIERUNG:
  void play_tone(uint16_t frequency, uint16_t duration_10ms) {
      // RMT-Konfiguration mit Carrier-Frequenz = frequency
      rmt_carrier_config_t carrier_cfg = {
          .frequency_hz = frequency,
          .duty_cycle = 0.5,  // 50% Duty-Cycle
      };
      rmt_apply_carrier(rmt_channel, &carrier_cfg);
      rmt_enable(rmt_channel);

      vTaskDelay(pdMS_TO_TICKS(duration_10ms * 10));

      rmt_disable(rmt_channel);
      vTaskDelay(pdMS_TO_TICKS(10));  // Fade-Out (verhindert Knacken)
  }

  void theme_task(void *pvParameters) {
      while (1) {
          for (int i = 0; i < THEME_LENGTH; i++) {
              play_tone(notes[i], durations[i]);
          }
      }  // Endlos-Loop
  }

TASK-START:
  void StartTheme(void) {
      xTaskCreate(theme_task, "ThemeTask", 2048, NULL, 3, NULL);
  }


================================================================================
5. HERAUSFORDERUNGEN UND LÖSUNGEN
================================================================================

5.1 TOP 10 TECHNISCHE PROBLEME
------------------------------

PROBLEM #1: LED-FARBFEHLER BEI WS2812
Symptom:     Zufällige LEDs zeigen falsche Farben (grün statt rot)
Ursache:     Interrupts unterbrechen Software-Bit-Banging
Lösung:      ESP-IDF led_strip-Komponente mit RMT-Hardware
Debugging:   4 Stunden (mehrere Bibliotheken getestet)
```

```
PROBLEM #2: GAME-OVER-LOOP (UNENDLICHER NEUSTART)
Symptom:     Nach Game Over startet Spiel sofort wieder neu
Ursache:     Button-Event aus Blink-Animation verbleibt in Queue
Lösung:      Comprehensive Queue-Drain + 500ms kontinuierliches Draining
Code-Fix:
             while (controls_get_event(&ev)) {}
             uint32_t start = get_time();
             while (get_time() - start < 500) {
                 while (controls_get_event(&ev)) {}
                 vTaskDelay(10);
             }
Debugging:   6 Stunden (mehrere Ansätze probiert)

PROBLEM #3: ROTATION NAHE RAND = CRASH
Symptom:     ESP32 Reset bei Rotation am linken/rechten Spielfeldrand
Ursache:     Negative Array-Indizes in grid_check_collision()
Lösung:      Bounds-Check VOR Array-Zugriff
Code-Fix:
             if (gx < 0 || gx >= GRID_WIDTH) return true;
             if (gy < 0) continue;  // Oberhalb OK
             if (gy >= GRID_HEIGHT) return true;
             // Erst jetzt: grid[gy][gx] zugreifen
Debugging:   3 Stunden

PROBLEM #4: HIGHSCORE VERSCHWINDET NACH STROMAUSFALL
Symptom:     Gespeicherter Highscore ist nach Reboot weg
Ursache:     nvs_commit() vergessen nach nvs_set_u32()
Lösung:      nvs_commit() hinzugefügt
Code-Fix:
             nvs_set_u32(nvs_handle, "highscore", score);
             nvs_commit(nvs_handle);  // ← ESSENTIELL!
Debugging:   2 Stunden

PROBLEM #5: FLACKERNDE LEDS BEI HOHER GESCHWINDIGKEIT
Symptom:     LED-Matrix flackert bei Level 10 (50ms Fall-Intervall)
Ursache:     led_strip_clear() + vollständiges Neuzeichnen
Lösung:      Differential Rendering (siehe 4.1)
Debugging:   8 Stunden (Optimierung mehrfach iteriert)

PROBLEM #6: AUDIO-KNACKEN BEI NOTENWECHSEL
Symptom:     Hörbares Knacken beim Übergang zwischen Tönen
Ursache:     RMT-Carrier abrupt gestoppt (Phasensprung)
Lösung:      10ms Fade-Out zwischen Noten
Debugging:   3 Stunden

PROBLEM #7: DISPLAY-INITIALISIERUNG SCHLÄGT FEHL
Symptom:     ESP32 bootet nicht wenn Display nicht verbunden
Ursache:     I²C-Timeout blockiert Boot-Sequenz
Lösung:      Fehlerbehandlung + Graceful Degradation
Code-Fix:
             if (display_init() != ESP_OK) {
                 g_disp = NULL;  // Display deaktiviert
             }
             // Spiel läuft trotzdem weiter
Debugging:   2 Stunden

PROBLEM #8: BUTTON-PRELLEN (MEHRFACH-EVENTS)
Symptom:     Einzelner Button-Press löst mehrere Aktionen aus
Ursache:     Hardware-Prellen (mechanische Kontakt-Bounces)
Lösung:      50ms Software-Debounce in ISR (siehe 4.2)
Debugging:   6 Stunden (verschiedene Debounce-Zeiten getestet)

PROBLEM #9: QUEUE-OVERFLOW (EVENTS VERLOREN)
Symptom:     Button-Presses werden manchmal ignoriert
Ursache:     Queue nur 4 Events groß → Overflow bei schnellem Drücken
Lösung:      Queue-Size auf 8 erhöht
Code-Fix:
             s_button_queue = xQueueCreate(8, sizeof(gpio_num_t));
Debugging:   1 Stunde

PROBLEM #10: SPIEL FRIERT EIN NACH LANGER SPIELZEIT
Symptom:     Nach ~30 Minuten friert Spiel ein (keine Reaktion)
Ursache:     Stack-Overflow im GameLoop-Task
Lösung:      Stack-Size von 2048 auf 4096 Bytes erhöht
Code-Fix:
             xTaskCreate(game_loop_task, "GameLoopTask", 4096, ...);
Debugging:   4 Stunden (schwer reproduzierbar)
```

5.2 ZEITAUFWAND NACH PHASEN

Phase	Stunden	Anteil
Hardware-Setup & Verkabelung	12	9%
LED-Matrix-Mapping-Debugging	8	6%
WS2812-Integration (RMT)	4	3%
Display-Integration (SSD1306+LVGL)	6	5%
Button-System (ISR + Queue)	8	6%
Collision Detection & Physics	10	8%
Rendering-Optimierung	12	9%
Spiellogik (Spawn, Clear, Score)	15	11%
Audio-System (RMT + Theme)	10	8%
Score/Highscore/NVS	6	5%
Splash-Screen-Animationen	8	6%
Testing & Bugfixing	20	15%
Code-Refactoring	10	8%
Dokumentation & Kommentare	12	9%
GESAMT	~131	100%

ANMERKUNG:
  Initial geschätzt: ~40 Stunden
  Tatsächlich: ~131 Stunden (3.3× länger!)
  Hauptgrund: Hardware-Debugging + unvorhergesehene Bugs

6. VERGLEICH: PLANUNG VS. UMSETZUNG

FEATURE	GEPLANT	UMGESETZT	ÄNDERUNGSGRUND
Rotation	❌ Nein	☑ Ja	Gameplay zu simpel
Spielfeld-Größe	10×20	16×24	Hardware-bedingt
Display	SSD1306	SSD1306	Verfügbare Hardware
Ausgabe	OLED ODER LED	OLED + LED	Beide gleichzeitig
BLE/Flutter	☑ Ja	❌ Nein	Fokus Performance
Geschwindigkeits-Level	Konstant	10 Level	Mehr Challenge
Punktesystem	Optional	Vollständig	Spiel-Tiefe
Highscore-Speicherung	❌ Nein	☑ NVS	Persistenz wichtig
Audio	Optional	Vollständig	Atmosphäre
Splash-Screen	❌ Nein	☑ Ja	Professionell
Multi-Line-Clear	❌ Nein	☑ Ja	Tetris-Standard
Emergency-Reset	❌ Nein	☑ Ja	Debugging-Hilfe
Code-Struktur	OOP (Fn-Ptr)	Modular	Einfacher wartbar
Button-Count	3	4	Rotation + Schneller

LERNZIELE:
☑ OOP in C   → Umgesetzt als modulare Struktur (wartbarer)
❌ BLE-Kommunikation   → Nicht umgesetzt (Fokus lokal)
☑ Echtzeitsteuerung   → 60 FPS Rendering, 5ms Input-Polling
☑ Ressourcenmanagement → Differential Rendering, NVS, optimierte Tasks
☑ ZUSÄTZLICH GELERNT: → RMT-Hardware, ISR-Programmierung, LVGL

7. ERGEBNISSE UND ERKENNTNISSE

7.1 TECHNISCHE ERFOLGE

☑ 60 FPS flimmerfreies Rendering (384 LEDs)
☑ <10ms Input-Latenz (Interrupt-basiert)
☑ Parallele Audio-Wiedergabe ohne Performance-Verlust
☑ Robuste Fehlerbehandlung (Display optional, NVS-Korruption)
☑ Smart Block-Spawning (versucht mehrere Positionen)
☑ Professionelle Animationen (Splash, Game-Over, Line-Clear)
☑ Persistent Storage (Highscore überlebt Stromausfall)

7.2 WICHTIGSTE ERKENNTNISSE

1. HARDWARE-FIRST:
   Hardware zuerst testen bevor Software-Entwicklung!
   LED-Verkabelung hat 8 Stunden gekostet weil erst spät getestet.

2. INCREMENTAL DEVELOPMENT:
   Kleine Schritte, nach jedem Schritt testen.
   Bugs sind einfacher zu finden wenn nur 1 Feature hinzugefügt wurde.

3. TIMING IST KRITISCH:

   Embedded Systems sind timing-sensitiv (WS2812, Debouncing).
   Hardware-Peripherals (RMT) sind essentiell für präzises Timing.

4. DOKUMENTATION PARALLEL:
   Kommentare sofort beim Coden schreiben, nicht am Ende.
   Nach 2 Wochen versteht man eigenen Code nicht mehr.

5. REALISTISCHE ZEITPLANUNG:
   Initial geschätzt: 40 Stunden
   Tatsächlich: 131 Stunden (3.3×)
   Debugging kostet ~15% der Gesamtzeit!

6. FEHLERBEHANDLUNG WICHTIG:
   Graceful Degradation (Spiel läuft ohne Display)
   Robuste Initialisierung (NVS-Korruption)

7. USER-FEEDBACK WERTVOLL:
   Emergency-Reset kam durch Feature-Request von Testern
   Button-Debouncing wurde mehrfach angepasst nach Feedback

7.3 SCHWÄCHEN DER AKTUELLEN IMPLEMENTIERUNG

❌ Kein BLE (ursprünglich geplant) → könnte nachgerüstet werden
❌ Keine Hold-Funktion (Block für später speichern)
❌ Keine Next-Block-Vorschau auf Display
❌ Keine T-Spin-Erkennung (fortgeschrittene Tetris-Mechanik)
❌ LED-Matrix fest kodiert (nicht konfigurierbar)
❌ Musik kann nicht pausiert werden
❌ Keine Replay-Funktion (Spiel aufzeichnen/abspielen)

7.4 MÖGLICHE ERWEITERUNGEN

GAMEPLAY:
□ Hard-Drop (Block sofort ganz nach unten)
□ Hold-Funktion (Block speichern)
□ Next-Block-Vorschau (auf Display zeigen)
□ Ghost-Piece (zeigt wo Block landen wird)
□ Combo-System (Punkte-Multiplikator)

TECHNISCH:
□ BLE-Implementierung (Smartphone-Steuerung)
□ WiFi-Highscore-Server (globale Rangliste)
□ SD-Karten-Logging (Replay-Funktion)
□ Mehrere Soundeffekte (Line-Clear, Level-Up, Game-Over)
□ Konfigurierbare LED-Matrix-Größe

HARDWARE:
□ Größere LED-Matrix (32×48 Pixel)
□ Touchscreen-Steuerung (statt Tasten)
□ RGB-Status-LEDs
□ Externes EEPROM für mehr Highscore-Slots

==
8. FAZIT
==

Das Projekt hat die ursprünglichen Ziele deutlich übertroffen und wurde von
einem "minimalistischen Tetris" zu einem vollwertigen, spielbaren Spiel mit
professionellen Features entwickelt.

TECHNISCHE HÖHEPUNKTE:
- 60 FPS flimmerfreies Rendering auf 384 LEDs
- Interrupt-basierte Button-Steuerung mit <10ms Latenz
- Parallele Audio-Wiedergabe (FreeRTOS-Tasks)
- Robuste Fehlerbehandlung und Graceful Degradation
- Persistent Storage für Highscore (NVS)

LERNEFFEKT:
  Das Projekt vermittelte tiefe Kenntnisse in:
- Embedded Systems Programming (ESP-IDF, FreeRTOS)
- Hardware-Integration (WS2812, I²C, GPIO, RMT)
- Interrupt-Programmierung (ISR + Event-Queues)
- Performance-Optimierung (Differential Rendering)
- Debugging auf Embedded-Hardware (ohne Debugger)

ZEITAUFWAND:
  ~131 Stunden (3.3× länger als initial geschätzt)
  Hauptgrund: Hardware-Debugging + unvorhergesehene Bugs

  Aber: Jede Stunde war lehrreich!

SPIELBARKEIT:
  Das Spiel ist vollständig spielbar und bietet ein authentisches Tetris-
  Erlebnis. Die Steuerung ist responsive, das Rendering flimmerfrei, und
  die Musik schafft Atmosphäre. Mehrere Testpersonen haben das Spiel gespielt
  und positives Feedback gegeben.

EMPFEHLUNG FÜR ÄHNLICHE PROJEKTE:
  1. Hardware zuerst vollständig testen
  2. Incremental Development (kleine Schritte)
  3. Zeitplanung × 3 rechnen
  4. Dokumentation parallel schreiben
  5. User-Feedback früh einholen
  6. Fehlerbehandlung von Anfang an einbauen

AUSBLICK:
  Das Projekt bietet viele Möglichkeiten für Erweiterungen (siehe 7.4).
  Besonders interessant wären:
  • BLE-Implementierung für Smartphone-Steuerung
  • WiFi-Highscore-Server für globale Rangliste
  • Größere LED-Matrix (32×48 Pixel)

Das Projekt zeigt eindrucksvoll, dass moderne Embedded Systems wie der ESP32-S3
leistungsfähig genug sind, um komplexe Anwendungen mit professionellen Features
zu realisieren - und das zu einem Bruchteil der Kosten traditioneller Gaming-
Hardware.

==
ANHANG: REFERENZEN UND RESSOURCEN
==

HARDWARE-QUELLEN:
- ESP32-S3: https://www.espressif.com/en/products/socs/esp32-s3
- WS2812 Datasheet: https://cdn-shop.adafruit.com/datasheets/WS2812.pdf
- SSD1306 Datasheet: https://www.displayfuture.com/Display/datasheet/
  controller/SSD1306.pdf

SOFTWARE-REFERENZEN:
- ESP-IDF: https://docs.espressif.com/projects/esp-idf/en/latest/
- FreeRTOS: https://www.freertos.org/
- LVGL: https://lvgl.io/
- led_strip: https://components.espressif.com/components/
  espressif/led_strip