TETRIS ESP32-S3 - Detaillierte Systemdokumentation & Prozesslandkarte

Inhaltsverzeichnis

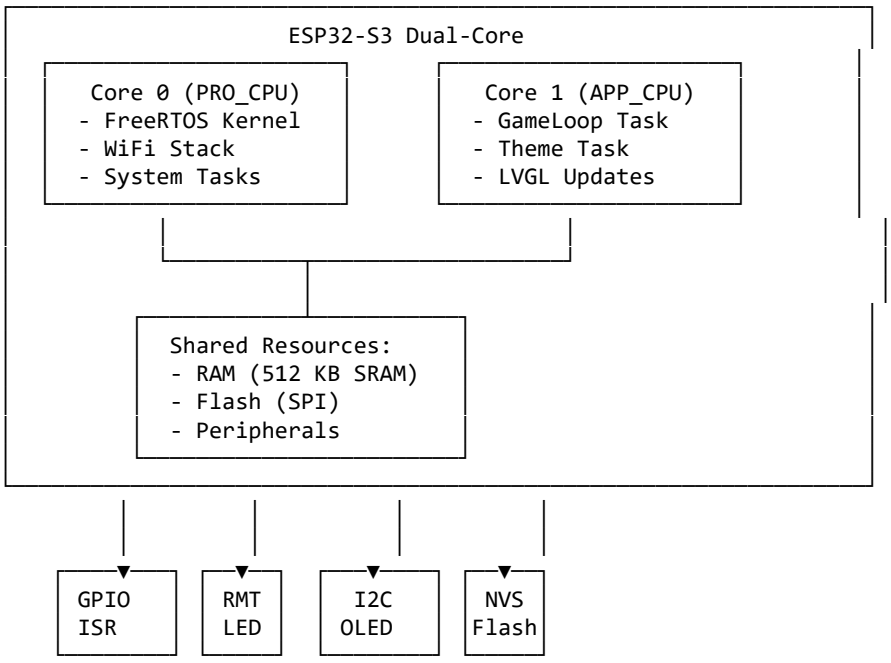
System-Übersicht

Projekt: ESP32-S3 Tetris mit WS2812B LED-Matrix und OLED-Display
Plattform: ESP-IDF v5.5.1, FreeRTOS v10.5.1
Mikrocontroller: ESP32-S3-WROOM-1 (Dual-Core Xtensa LX7)
Taktfrequenz: 240 MHz
RAM: 512 KB SRAM
Flash: 4-16 MB (projektabhängig)
Programmiersprache: C (C11 Standard)

Hauptkomponenten:

- **LED-Matrix:** 16x24 WS2812B RGB LEDs (384 LEDs total, 5V, 60mA/LED max)
- **Display:** SSD1306 OLED 128x64 I2C (0x3C, 400 kHz, monochrom)
- **Eingabe:** 4 GPIO-Buttons mit ISR-Handling (Pull-up, aktiv-LOW, 150ms Debounce)
- **Soundausgabe:** LEDC PWM Buzzer für Tetris-Theme (optional)
- **Persistenz:** NVS (Non-Volatile Storage) für Highscore
- **Kommunikation:** UART (115200 baud) für Debug-Logging

System-Architektur Überblick:



Hardware-Architektur

Pin-Belegung

LED-Matrix (WS2812B)

- **GPIO 1:** LED-Datenleitung (RMT-Protokoll)
- **Anzahl LEDs:** 384 (16 Breite × 24 Höhe)
- **Protokoll:** WS2812B via ESP-IDF RMT Driver

Buttons (Pull-up, aktiv LOW)

- **GPIO 4:** BTN_LEFT (Links bewegen)
- **GPIO 5:** BTN_RIGHT (Rechts bewegen)
- **GPIO 6:** BTN_ROTATE (Block rotieren)
- **GPIO 7:** BTN_FASTER (Schneller fallen lassen)
- **ISR-Trigger:** Negative Flanke (NEGEDGE)
- **Debounce:** 150ms Software-Debounce

OLED Display (SSD1306)

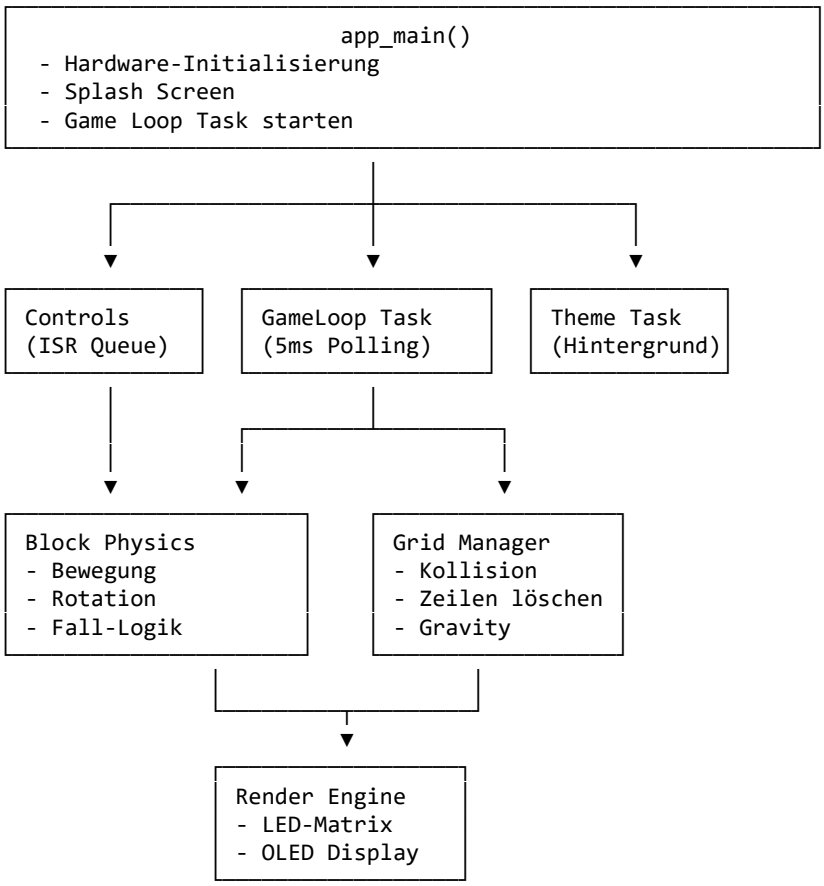
- **GPIO 20:** I2C SCL (400 kHz, mit Pull-up)
- **GPIO 21:** I2C SDA (400 kHz, mit Pull-up)
- **Adresse:** 0x3C
- **Auflösung:** 128×64 Monochrom
- **Bibliothek:** LVGL mit ESP-LVGL-Port

Buzzer (optional)

- **LEDC PWM:** Für Tetris-Themesong

Software-Architektur

Modular aufgebautes System

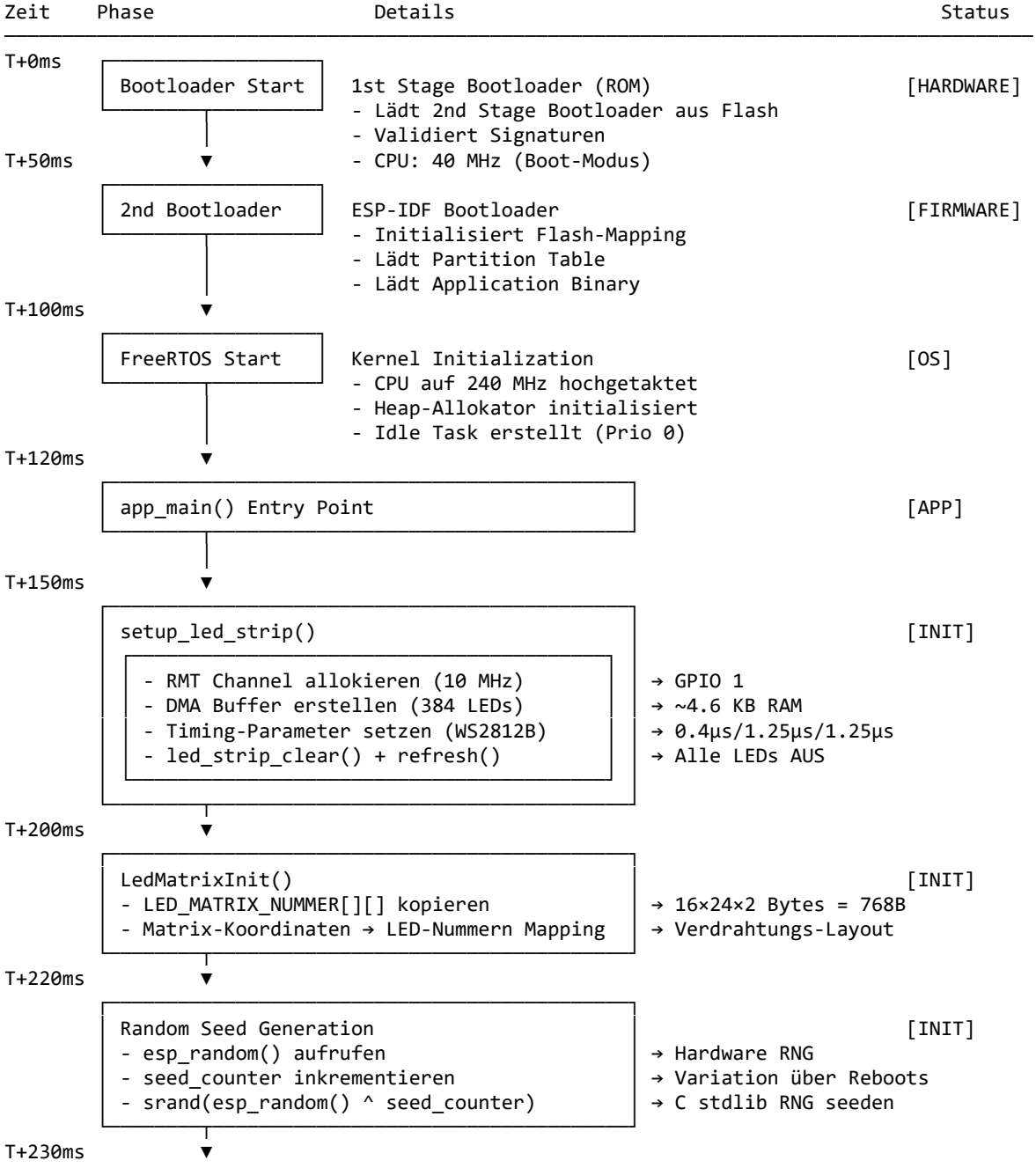


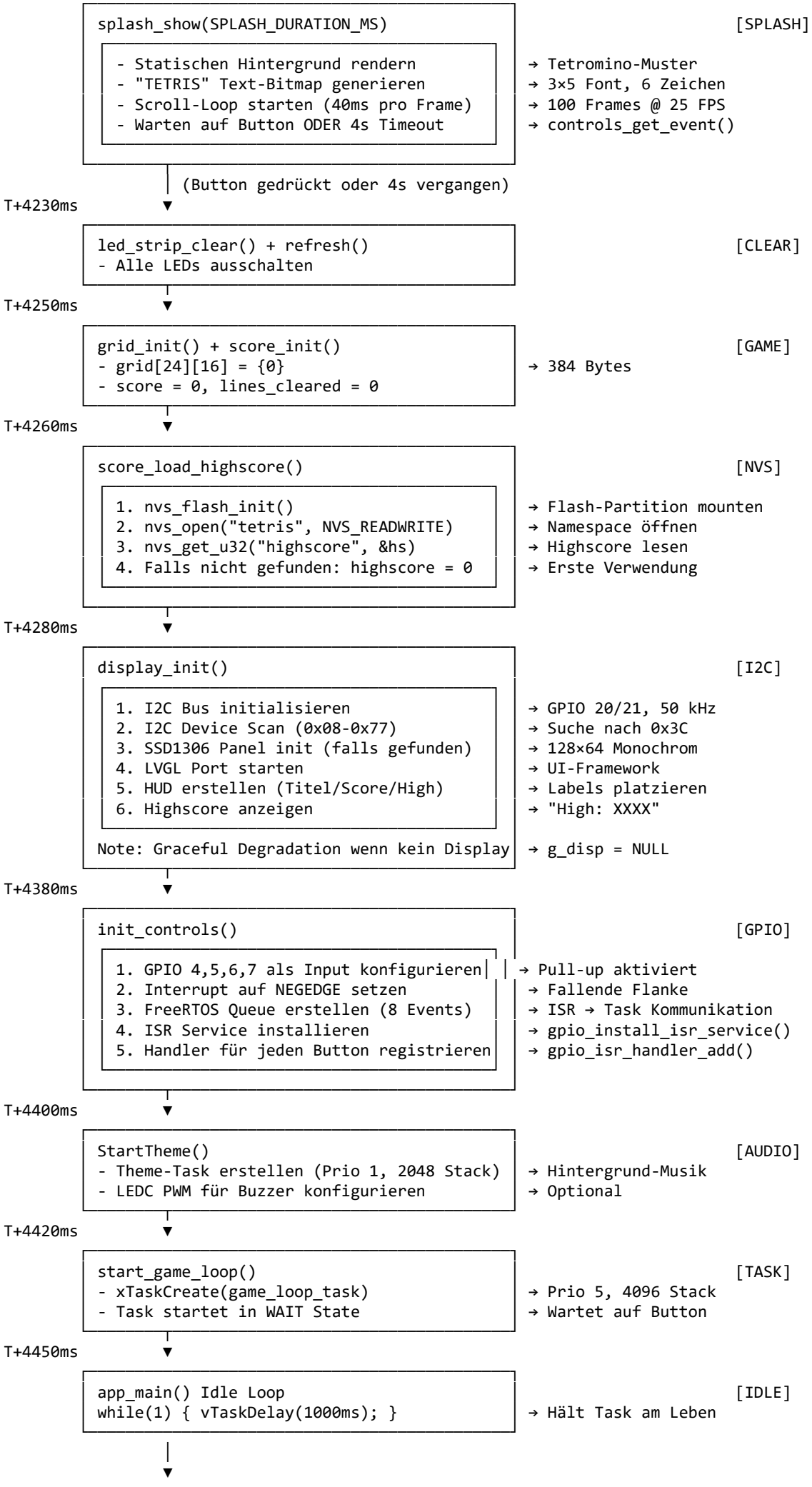
Module und Dateien

Modul	Dateien	Verantwortung
main.c	`main/src/main.c`	System-Initialisierung, Entry Point
GameLoop	`GameLoop.c/h`	Spiel-Hauptschleife, Zustandsmaschine
Controls	`Controls.c/h`	Button-ISR, Event-Queue, Debouncing
Blocks	`Blocks.c/h`	Tetromino-Definitionen, Rotation, Farben
Grid	`Grid.c/h`	Spielfeld, Kollisionserkennung, Zeilen löschen
Splash	`Splash.c/h`	Startbildschirm-Animation
Score	`Score.c/h`	Punkteverwaltung, NVS Highscore
SpeedManager	`SpeedManager.c/h`	Dynamische Geschwindigkeit
DisplayInit	`DisplayInit.c/h`	OLED I2C, LVGL Integration
LedMatrixInit	`LedMatrixInit.c/h`	WS2812B Mapping
ThemeSong	`ThemeSong.c/h`	Tetris-Musik via PWM
Colors	`Colors.c`	RGB-Farbtabelle für Blöcke
Globals	`Globals.h`	Konfigurationsparameter

Detaillierte Prozessablauf-Diagramme

1. System-Start (Boot-Sequenz mit exakten Timings)




splash_show(SPLASH_DURATION_MS)
[SPLASH]
- Statischen Hintergrund rendern
- "TETRIS" Text-Bitmap generieren
- Scroll-Loop starten (40ms pro Frame)
- Warten auf Button ODER 4s Timeout
→ Tetromino-Muster
→ 3×5 Font, 6 Zeichen
→ 100 Frames @ 25 FPS
→ controls_get_event()
(Button gedrückt oder 4s vergangen)
T+4230ms
led_strip_clear() + refresh()
- Alle LEDs ausschalten
[CLEAR]
T+4250ms
grid_init() + score_init()
- grid[24][16] = {0}
- score = 0, lines_cleared = 0
[GAME]
→ 384 Bytes
T+4260ms
score_load_highscore()
[NVS]
1. nvs_flash_init()
2. nvs_open("tetris", NVS_READWRITE)
3. nvs_get_u32("highscore", &hs)
4. Falls nicht gefunden: highscore = 0
→ Flash-Partition mounten
→ Namespace öffnen
→ Highscore lesen
→ Erste Verwendung
T+4280ms
display_init()
[I2C]
1. I2C Bus initialisieren
2. I2C Device Scan (0x08-0x77)
3. SSD1306 Panel init (falls gefunden)
4. LVGL Port starten
5. HUD erstellen (Titel/Score/High)
6. Highscore anzeigen
→ GPIO 20/21, 50 kHz
→ Suche nach 0x3C
→ 128×64 Monochrom
→ UI-Framework
→ Labels platzieren
→ "High: XXXX"
Note: Graceful Degradation wenn kein Display
→ g_disp = NULL
T+4380ms
init_controls()
[GPIO]
1. GPIO 4,5,6,7 als Input konfigurieren
2. Interrupt auf NEGEDGE setzen
3. FreeRTOS Queue erstellen (8 Events)
4. ISR Service installieren
5. Handler für jeden Button registrieren
→ Pull-up aktiviert
→ Fallende Flanke
→ ISR → Task Kommunikation
→ gpio_install_isr_service()
→ gpio_isr_handler_add()
T+4400ms
StartTheme()
[AUDIO]
- Theme-Task erstellen (Prio 1, 2048 Stack)
- LEDC PWM für Buzzer konfigurieren
→ Hintergrund-Musik
→ Optional
T+4420ms
start_game_loop()
[TASK]
- xTaskCreate(game_loop_task)
- Task startet in WAIT State
→ Prio 5, 4096 Stack
→ Wartet auf Button
T+4450ms
app_main() Idle Loop
while(1) { vTaskDelay(1000ms); }
[IDLE]
→ Hält Task am Leben

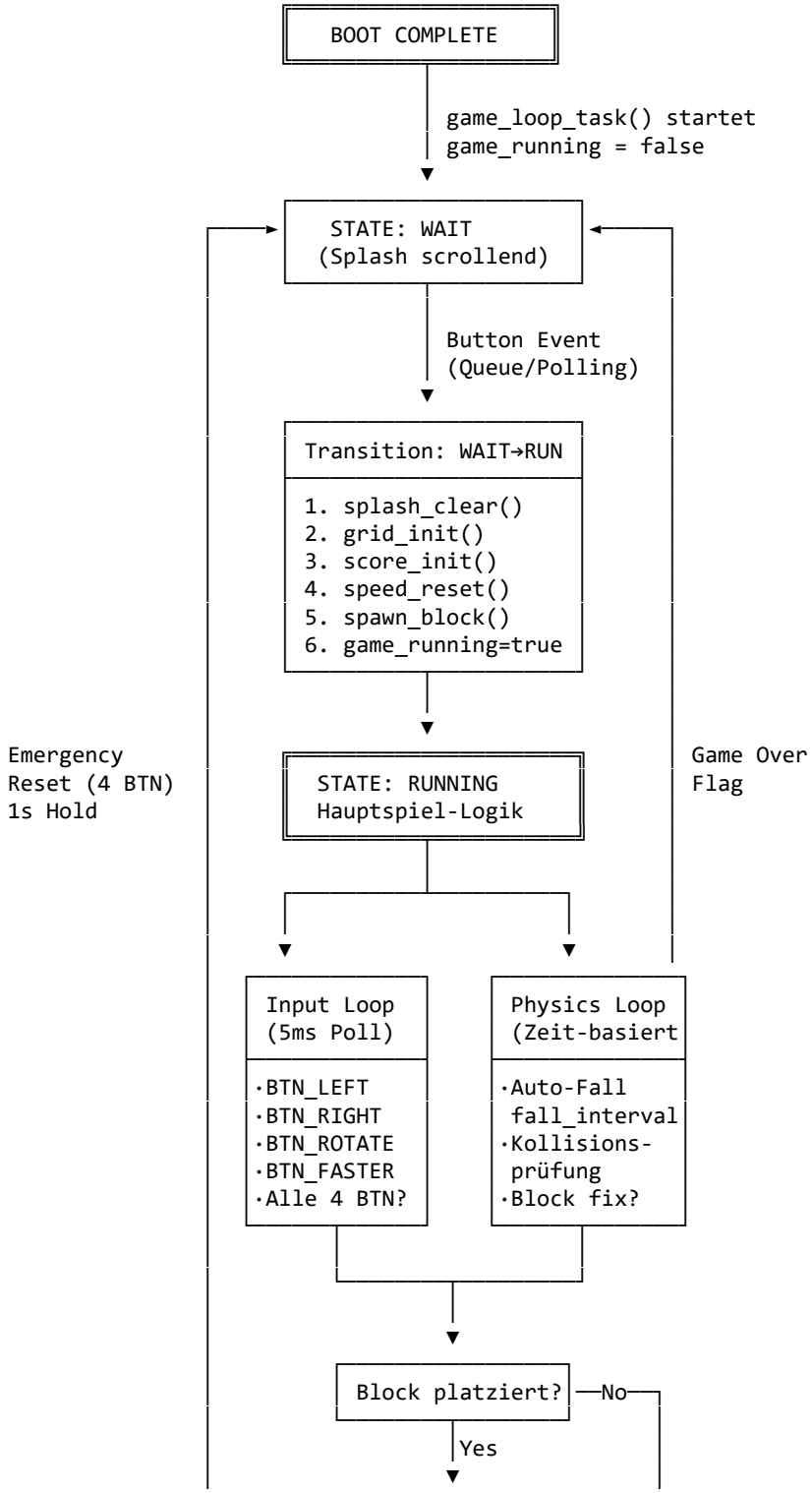
```
┌──────────────────────────────────────────────────────┐
│ System Running (Multi-Tasking)                       │
│  ┌──────────────────────────────────────────────┐    │
│  │ Core 0: FreeRTOS Kernel, WiFi, System        │    │
│  │ Core 1: GameLoop Task (5ms Polling)          │    │
│  │         Theme Task (Hintergrund)             │    │
│  │         LVGL Timer (UI-Updates)              │    │
│  └──────────────────────────────────────────────┘    │
└──────────────────────────────────────────────────────┘
```

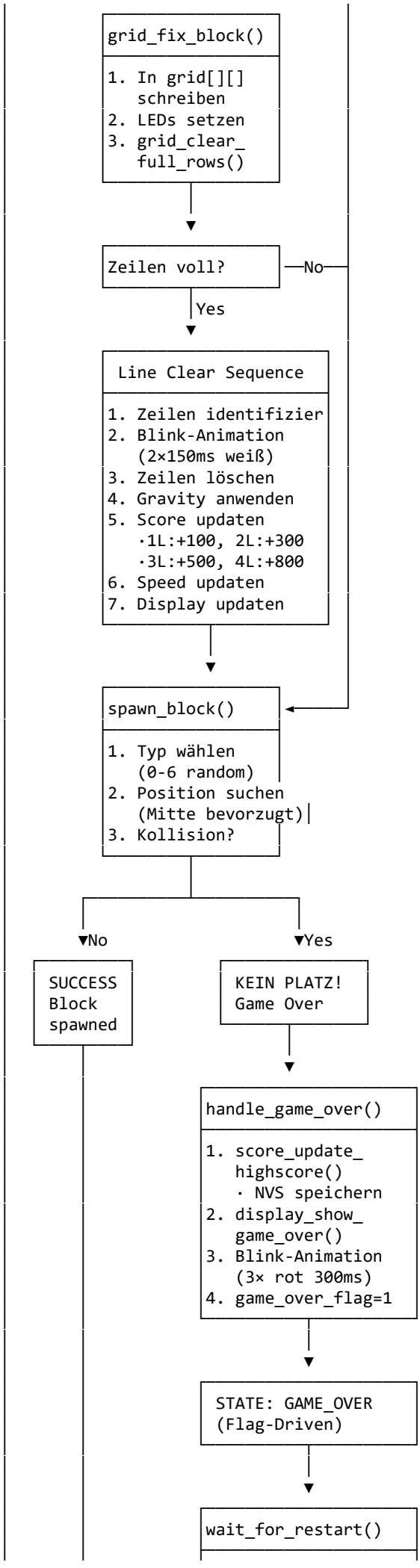
[RUN]

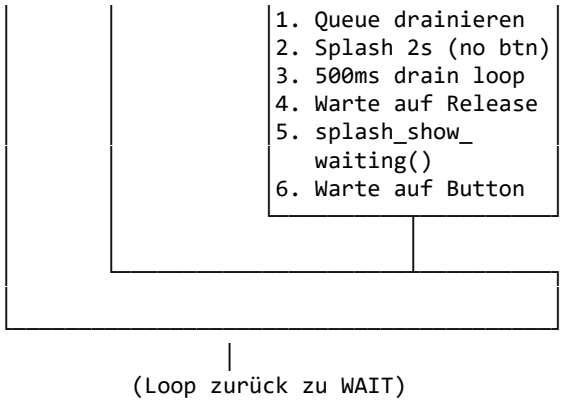
```
Total Boot Time: ~4.45 Sekunden
Memory Usage at Boot:
  - Heap Free: ~380 KB (von 512 KB total)
  - Stack GameLoop: 4096 Bytes allokiert
  - Stack Theme: 2048 Bytes allokiert
  - DMA Buffers: ~5 KB (RMT + LVGL)
```

---

2. Game Loop State Machine (Detailliert mit allen Transitions)




grid_fix_block()
1. In grid[][] schreiben
2. LEDs setzen
3. grid_clear_full_rows()
Zeilen voll?
No
Yes
Line Clear Sequence
1. Zeilen identifizier
2. Blink-Animation (2×150ms weiß)
3. Zeilen löschen
4. Gravity anwenden
5. Score updaten
·1L:+100, 2L:+300
·3L:+500, 4L:+800
6. Speed updaten
7. Display updaten
spawn_block()
1. Typ wählen (0-6 random)
2. Position suchen (Mitte bevorzugt)
3. Kollision?
No
Yes
SUCCESS Block spawned
KEIN PLATZ! Game Over
handle_game_over()
1. score_update_highscore()
· NVS speichern
2. display_show_game_over()
3. Blink-Animation (3× rot 300ms)
4. game_over_flag=1
STATE: GAME_OVER (Flag-Driven)
wait_for_restart()



(Loop zurück zu WAIT)

Zusätzliche Transitions:

Emergency Reset Detection (jederzeit während RUNNING):

```
if (controls_all_buttons_pressed())
  vTaskDelay(1000ms)  // 1s Hold-Check
  if (still_pressed):
    → Hard Reset Sequenz (wie Game Over)
    → Zurück zu WAIT State
```

Rendering (parallel zu State Machine):

```
Render Loop (60 FPS, 16ms Intervall)
if (current_time - last_render >= 16ms):
  render_grid()
  · Vorherige dynamische Pixel restaurieren
  · Aktuellen Block zeichnen
  · led_strip_refresh()
```

State Variables:

```
  - game_running: bool     // false=WAIT, true=RUNNING
  - game_over_flag: int    // 1=GAME_OVER erkannt
  - current_block: struct  // Aktueller Tetromino
  - grid[24][16]: uint8_t  // Fixierte Blöcke
  - score: int            // Aktueller Score
  - total_lines: uint32_t // Geclearte Zeilen (für Speed)
```

---

3. Game Loop Task - Mikro-Ebene (Jeden 5ms Zyklus)

```
game_loop_task() - FreeRTOS Task
Priorität: 5 (High), Stack: 4096 Bytes, Polling: 5ms, CPU: Core 1
```

T+0µs

```
INITIALIZATION (einmalig beim Task-Start)
speed_manager_init()        → fall_interval = 400ms
grid_init()                 → memset(grid, 0, 384 bytes)
score_init()                → score = 0, lines = 0
display_reset_and_show_hud()→ LVGL Screen aufbauen
game_running = false        → Startet in WAIT State
last_fall_time = 0          → Timer-Variablen
last_render_time = 0        →
```

▼

```
MAIN LOOP: while(1) { ... vTaskDelay(5ms); }
Jede Iteration = 1 "Tick" = 5ms Mindestdauer
```

```
Loop Iteration N (T = N × 5ms nach Task-Start)
```

T+0µs ▼

```
[1] current_time = xTaskGetTickCount() * portTICK_PERIOD_MS
    Lese aktuelle Zeit in Millisekunden
    Typ: uint32_t, Overflow nach ~49.7 Tage
```

T+10µs ▼

```
[2] EMERGENCY RESET CHECK
  if (controls_all_buttons_pressed()) {
    // Alle 4 GPIOs prüfen: gpio_get_level() == 0?
    vTaskDelay(1000ms);  // 1 Sekunde halten
    if (controls_all_buttons_pressed()) {
      printf("[GameLoop] Emergency reset triggered\n");
      // Hard Reset Sequenz:
      reset_game_state();
      led_strip_clear() + refresh();
      game_running = false;
      wait_for_restart();  // Splash + Queue Drain + Wait
      // Spiel neu starten:
      splash_clear();
      reset_game_state();
      spawn_block();
      game_running = true;
      continue;  // Zurück zu Loop-Anfang
    }
  }
Typische Ausführungszeit: 50µs (wenn nicht gedrückt)
                          1000ms (wenn 1s halten)
```

T+60µs ▼

```
[3] GAME OVER FLAG CHECK
  if (game_over_flag) {
    game_over_flag = 0;
    game_running = false;
    display_reset_and_show_hud(highscore);
    wait_for_restart();  // Splash + Drain + Wait
    // Spiel neu starten:
    splash_clear();
    reset_game_state();
    spawn_block();
    game_running = true;
    last_fall_time = current_time;
    last_render_time = current_time;
    continue;
  }
Typische Ausführungszeit: 5µs (wenn Flag = 0)
                          3000ms+ (wenn Flag = 1)
```

T+65µs ▼

```
[4] WAIT STATE CHECK
  if (!game_running) {
    // WAIT State: Splash scrollt, warte auf Button
    splash_show(SPLASH_DURATION_MS);  // Blockiert!
    gpio_num_t ev;
    while (controls_get_event(&ev)) {}  // Queue drain
    vTaskDelay(50ms);
    // Warte auf Button-Release:
    while (check_button_pressed(ANY)) vTaskDelay(20ms);
    // Warte auf frischen Button-Press:
    controls_wait_event(&ev, portMAX_DELAY); // Blockiert!
    vTaskDelay(50ms);
    // Spiel starten:
    splash_clear();
```

```
    reset_game_state();
    spawn_block();
    game_running = true;
    last_fall_time = current_time;
    last_render_time = current_time;
    continue;
  }
```

Typische Ausführungszeit: 10µs (wenn game_running = true)
BLOCKIERT (wenn WAIT State)

(Ab hier: game_running = true garantiert)

T+75µs ▼

[5] RUNNING STATE - INPUT HANDLING
Alle Buttons werden JEDEN Loop geprüft (5ms Polling)

T+80µs ▼

[5a] BTN_LEFT (Block nach links bewegen)

```
if (check_button_pressed(BTN_LEFT)) {
  TetrisBlock tmp = current_block;  // Kopie erstellen
  tmp.x--;                          // 1 Spalte links
  if (!grid_check_collision(&tmp)) {
    current_block = tmp;  // Bewegung gültig
  }
  // Sonst: Kollision → Bewegung ignoriert
}
```

check_button_pressed():
- gpio_get_level(GPIO_NUM_4) == 0? (aktiv-LOW)
- Software-Debounce: 150ms seit letztem Press?
- Return: true/false

Ausführungszeit: 15µs (ohne Press), 15µs + Kollision (mit)

T+95µs ▼

[5b] BTN_RIGHT (Block nach rechts bewegen)

```
if (check_button_pressed(BTN_RIGHT)) {
  tmp = current_block; tmp.x++;
  if (!grid_check_collision(&tmp)) current_block = tmp;
}
```

Ausführungszeit: 15µs

T+110µs ▼

[5c] BTN_ROTATE (Block rotieren, außer O-Block)

```
if (check_button_pressed(BTN_ROTATE) &&
    current_block.color != 3) {  // 3 = O-Block
  tmp = current_block;
  rotate_block_90(&tmp);  // Matrix-Transformation
  if (!grid_check_collision(&tmp)) {
    current_block = tmp;
  }
}
```

rotate_block_90():
- Transponieren + Spiegeln der 4×4 shape[][] Matrix
- Ausführungszeit: ~5µs

T+130µs ▼

[5d] BTN_FASTER (Manuell schneller fallen)

```
if (check_button_pressed(BTN_FASTER)) {
  tmp = current_block; tmp.y++;
  if (!grid_check_collision(&tmp)) current_block = tmp;
```

```
}
Ausführungszeit: 15µs
```

T+145µs ▼

```
[6] AUTOMATIC FALL (Zeit-basiert)
```

T+150µs ▼

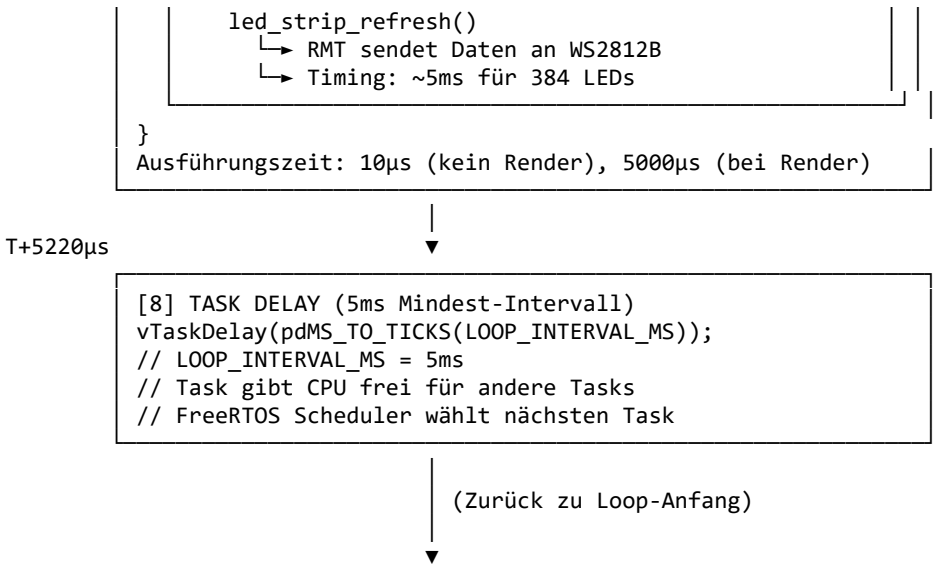
```
uint32_t fall_interval = speed_manager_get_fall_interval();
// fall_interval: 400ms (Level 0) bis 50ms (Level 10)

if (current_time - last_fall_time >= fall_interval) {
  last_fall_time = current_time;  // Timer zurücksetzen

  tmp = current_block;
  tmp.y++;  // 1 Zeile nach unten

  if (!grid_check_collision(&tmp)) {
    // Block kann weiter fallen
    current_block = tmp;
  } else {
    // Kollision unten → Block fixieren
    grid_fix_block(&current_block);
      ├─► grid[y][x] = color + 1  (für jedes Pixel)
      ├─► led_strip_set_pixel()   (sofort zeichnen)
      ├─► led_strip_refresh()
      └─► grid_clear_full_rows()
            ├─► Volle Zeilen finden
            ├─► Blink-Animation (2× weiß)
            ├─► Zeilen löschen
            ├─► Gravity anwenden (spaltenweise)
            ├─► score_add_lines(count)
            ├─► speed_manager_update_score(total_lines)
            └─► display_update_score(score, highscore)
    spawn_block();  // Neuer Block
      └─► Falls kein Platz: handle_game_over()
            └─► game_over_flag = 1 (Loop reagiert nächste Iteration)
  }
}
Ausführungszeit: 20µs (kein Fall), 50µs+ (bei Fall)
                 500ms+ (bei Zeilen-Clear + Animation)
```

T+200µs ▼

```
[7] RENDERING (60 FPS = alle 16ms)
```

T+220µs ▼

```
if (current_time - last_render_time >= RENDER_INTERVAL_MS) {
  last_render_time = current_time;
  render_grid();
  ┌──────────────────────────────────────────────────┐
  │ Render-Pipeline (Optimiert):                     │
  │ 1. Vorherige dynamische Pixel restaurieren       │
  │    for (i=0; i<prev_dynamic_count; i++):         │
  │      pos = prev_dynamic_pos[i]                   │
  │      if grid[pos] > 0: Farbe setzen              │
  │      else: Pixel löschen (0,0,0)                 │
  │                                                  │
  │ 2. Aktuellen Block zeichnen                      │
  │    for (by=0; by<4; by++)                        │
  │      for (bx=0; bx<4; bx++)                      │
  │        if shape[by][bx]:                         │
  │          get_block_rgb(color, &r, &g, &b)        │
  │          r/g/b *= BRIGHTNESS_SCALE               │
  │          led_strip_set_pixel(led_num, r, g, b)   │
  │          prev_dynamic_pos[count++] = [y,x]       │
  │                                                  │
  │ 3. LED-Matrix aktualisieren                      │
```



Render-Pipeline & Display-Aktualisierung

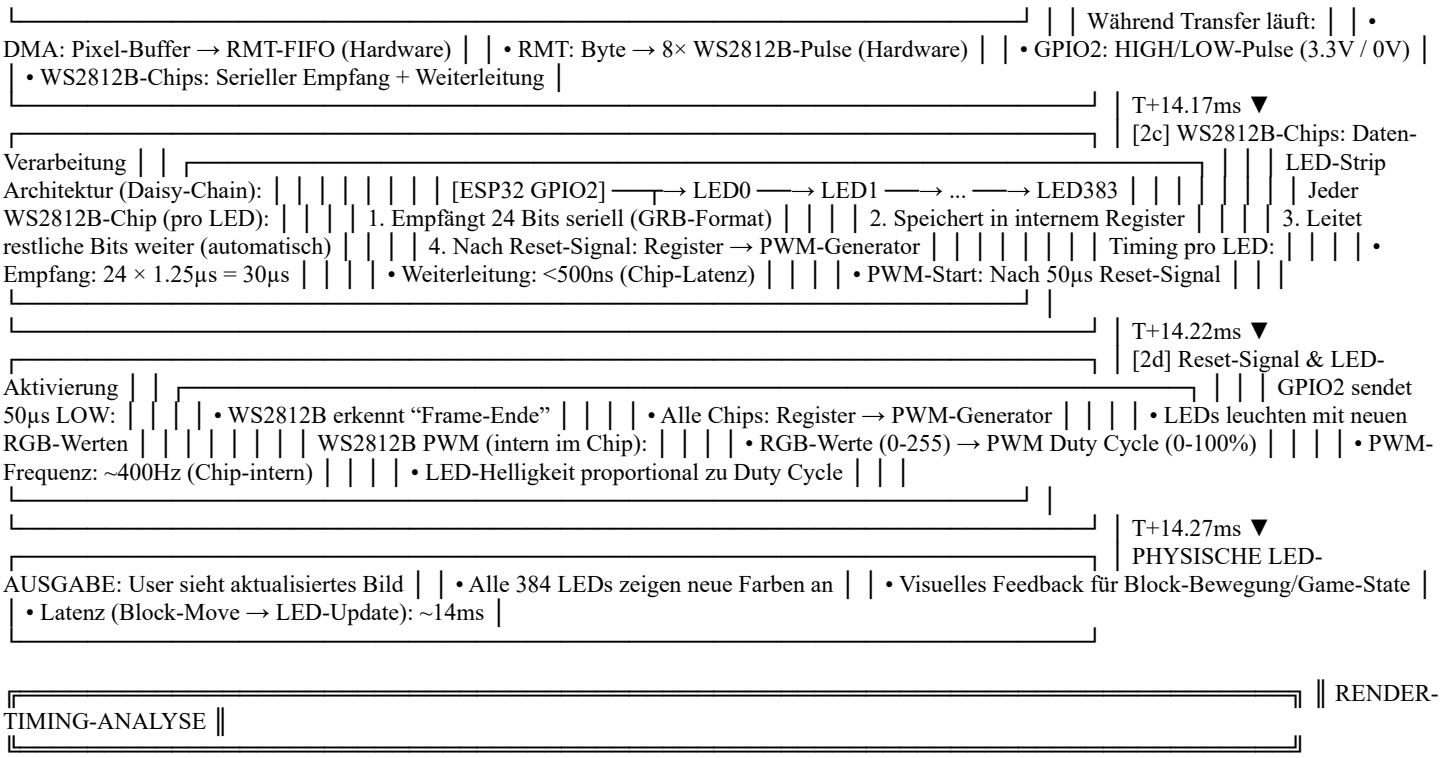
WS2812B LED Matrix - Microsekunden-Ebene

╔═══╗ ║ RENDER-PIPELINE (60 FPS Target) ║ ╚═══╝

TRIGGER: GameLoop Render-Tick (alle 16ms für 60 FPS) │ T+0µs ▼

╔═══╗ ║ PHASE 1: Grid-Rendering (render_grid) ║ ╚═══╝ │ T+0µs ▼ [1a] Vorherige dynamische Pixel restaurieren (~~500µs~~) ┌───┐ ~~// Delta-Rendering: Nur geänderte Pixel restaurieren~~ ~~for (int i = 0; i < prev_dynamic_count; i++) {~~ ~~int pos = prev_dynamic_pos[i];~~ ~~int y = pos / GRID_WIDTH;~~ ~~int x = pos % GRID_WIDTH;~~ ~~if (internal_grid[y][x] > 0) {~~ ~~// Grid-Pixel-Farbe setzen~~ ~~uint8_t r, g, b;~~ ~~get_block_rgb(internal_grid[y][x], &r, &g, &b);~~ ~~led_strip_set_pixel(led_num, r, g, b);~~ ~~} else {~~ ~~// Leeres Feld: Pixel löschen~~ ~~led_strip_set_pixel(led_num, 0, 0, 0);~~ ~~}~~ ~~}~~ └───┘ ~~Loop-Performance:~~ ~~• prev_dynamic_count: Typisch 16-32 (Block-Größe)~~ ~~• Pro Iteration: Array-Lookup (5ns) + Color-Calc (20ns)~~ ~~+ LED-Set (15ns) = ~40ns~~ ~~• Gesamt: 16-32 × 40ns = ~1.3µs (worst: 32 × 40ns = 1.3µs)~~ ~~• Overhead (Schleifen, Bedingungen): ~500µs~~ │ ~~T+500µs ▼~~ ~~[1b] Aktuellen Block zeichnen (2000µs)~~ ┌───┐ int dynamic_count = 0; for (int by = 0; by < 4; by++) { for (int bx = 0; bx < 4; bx++) { int shape = get_shape_value(&current_block, by, bx); if (shape == 0) continue; int y = current_block.y + by; int x = current_block.x + bx; if (x < 0 || x >= GRID_WIDTH) continue; if (y < 0 || y >= GRID_HEIGHT) continue; // Block-Farbe mit Brightness-Scaling: uint8_t r, g, b; get_block_rgb(current_block.color, &r, &g, &b); r = (r * BRIGHTNESS_SCALE) / 100; g = (g * BRIGHTNESS_SCALE) / 100; b = (b * BRIGHTNESS_SCALE) / 100; // LED-Strip-Index berechnen (Serpentine-Layout): int led_num; if (x % 2 == 0) { led_num = x*GRID_HEIGHT + (GRID_HEIGHT-1-y); } else { led_num = x*GRID_HEIGHT + y; } led_strip_set_pixel(led_num, r, g, b); prev_dynamic_pos[dynamic_count++] = y*GRID_WIDTH+x; } } } prev_dynamic_count = dynamic_count; └───┘ Loop-Performance: • 16 Iterationen (4×4 Block) • Pro Iteration: Shape-Check (10ns) + Bounds-Check (5ns) + Color-Calc (30ns) + LED-Set (15ns) + Position-Save (10ns) = ~70ns • Gesamt: 16 × 70ns = ~1.1µs • Overhead (Schleifen, Brightness): ~2000µs │ T+2500µs ▼

╔═══╗ ║ PHASE 2: RMT-Transfer (Hardware-Beschleunigt) ║ ╚═══╝ │ T+2500µs ▼ [2a] LED-Strip Refresh (led_strip_refresh) ┌───┐ esp_err_t led_strip_refresh(led_strip_handle_t strip) { // Interne Pixel-Buffer → RMT-Channel esp_err_t ret = rmt_transmit(rmt_channel, encoder, pixel_buffer, num_leds * 3, &tx_config); // Warte auf Transmission Complete: rmt_tx_wait_all_done(rmt_channel, 100ms); return ret; } RMT-Konfiguration (WS2812B Timing): • Bit 0: HIGH 0.4µs, LOW 0.85µs = 1.25µs/Bit • Bit 1: HIGH 0.8µs, LOW 0.45µs = 1.25µs/Bit • Reset: LOW 50µs (Frame-Ende) └───┘ Setup-Zeit: ~100µs (RMT-Config + DMA-Init) │ T+2600µs ▼ [2b] RMT-Hardware-Transfer (DMA-basiert) ┌───┐ RMT Peripheral sendet Daten: • 384 LEDs × 24 Bits/LED = 9216 Bits • 9216 Bits × 1.25µs/Bit = 11.52ms • + Reset-Signal: 50µs • Gesamt-Transfer: ~11.57ms CPU-Zustand während Transfer: • rmt_tx_wait_all_done() → CPU BLOCKIERT • Alternative: Async mit Callback (nicht implementiert) • DMA liest autonom aus pixel_buffer • RMT wandelt Bytes → Pulse-Sequenzen • GPIO2 sendet seriellen Datenstrom └───┘



RENDER-TIMING-ANALYSE

Phase	CPU-Zeit	Wartezeit	Gesamt
1a. Restore Previous Pixels	500µs	-	500µs
1b. Draw Current Block	2000µs	-	2000µs
2a. RMT-Setup	100µs	-	100µs
2b. RMT-Transfer (Blocking)	-	11.57ms	11.57ms
2c/d. WS2812B-Processing	-	100µs	100µs
GESAMT	~2600µs	~11.67ms	~14.27ms

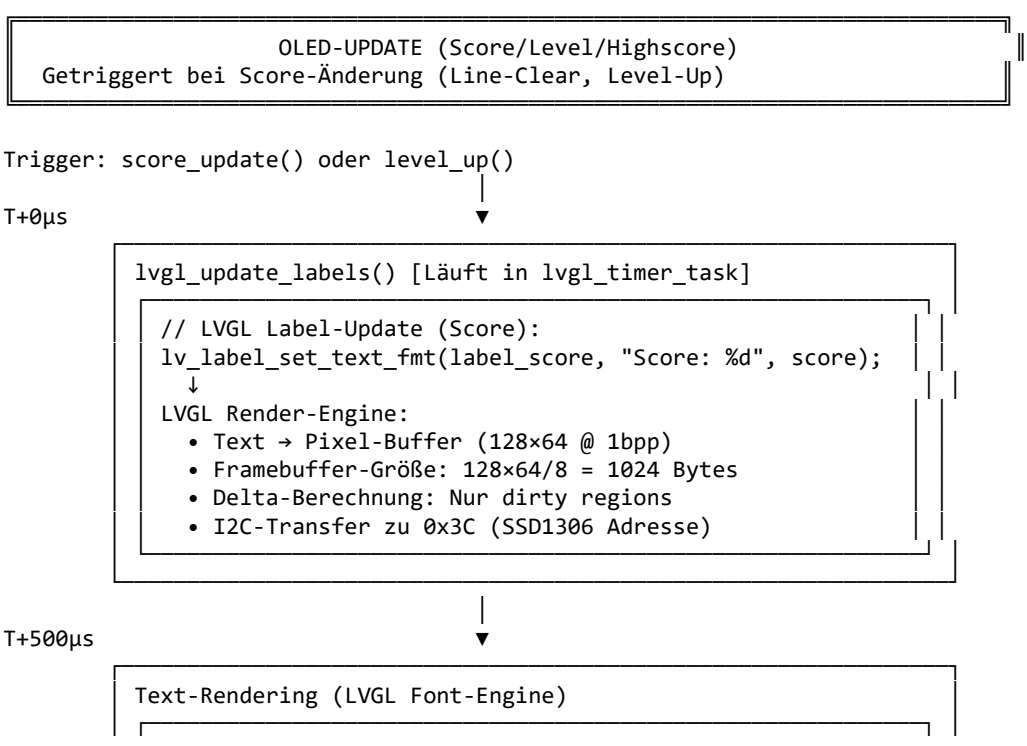
CPU-Auslastung während Render: • Aktive CPU-Zeit: 2.6ms (~~18.2% von 14.27ms)~~ • ~~Warte-Zeit (RMT): 11.67ms (~~81.8% von 14.27ms) • CPU blockiert während RMT-Transfer (wait_all_done)

Render-Frequenz: • Target: 60 FPS = 16.67ms/Frame • Tatsächlich: 14.27ms/Frame • Overhead: 16.67 - 14.27 = 2.4ms • Verfügbar für GameLogic: ~2.4ms pro Frame

Optimierungspotential: ✅ Delta-Rendering: Implementiert (nur geänderte Pixel) ✅ RMT-Hardware: Nutzt DMA → CPU-Entlastung ❌ Non-Blocking RMT: Aktuell Blocking → Umstellung auf Async würde CPU freigeben → Benötigt Double-Buffering (Pixel-Buffer × 2) → Vorsicht: Race-Conditions vermeiden

WS2812B Timing-Toleranzen: • Bit 0: 0.4µs ±150ns HIGH, 0.85µs ±150ns LOW • Bit 1: 0.8µs ±150ns HIGH, 0.45µs ±150ns LOW • Reset: >50µs LOW (Minimum) • RMT erzeugt präzise Pulse (±10ns Jitter)

OLED Display (SSD1306) - Parallel zur LED-Matrix



```
┌─────────────────────────────────────────────────────────────┐
│ Font-Lookup (Montserrat 14pt):                              │
│   • Character → Glyph-Bitmap (Font-Table)                   │
│   • Rasterization: Glyph → Framebuffer                      │
│   • Anti-Aliasing: 1bpp (schwarz/weiß, kein Graustufen)     │
│                                                             │
│ Beispiel "Score: 1234" (11 Zeichen):                        │
│   • 11 × Glyph-Lookup = ~11 × 30µs = ~330µs                 │
│   • Rasterization: ~100µs                                   │
│   • Gesamt: ~430µs                                          │
└─────────────────────────────────────────────────────────────┘
                              │
T+930µs                       ▼
┌─────────────────────────────────────────────────────────────┐
│ I2C-Transfer (Master: ESP32, Slave: SSD1306)                │
│ ┌─────────────────────────────────────────────────────────┐ │
│ │ I2C-Konfiguration:                                      │ │
│ │   • Adresse: 0x3C (7-bit)                               │ │
│ │   • Clock: 400kHz (Fast Mode)                           │ │
│ │   • SCL: GPIO22, SDA: GPIO21                            │ │
│ │                                                         │ │
│ │ Transfer-Modes:                                         │ │
│ │   • Full-Screen: 1024 Bytes @ 400kHz = ~20ms            │ │
│ │   • Partial (Score-Zeile): ~128 Bytes @ 400kHz = ~2.5ms │ │
│ │                                                         │ │
│ │ LVGL nutzt Partial-Update (dirty regions):              │ │
│ │   1. Berechne dirty rectangle (z.B. Zeile 0-15)         │ │
│ │   2. Sende I2C-Command: Set Page Address (0-1)          │ │
│ │   3. Sende I2C-Command: Set Column Address (0-127)      │ │
│ │   4. Sende I2C-Data: Pixel-Buffer (dirty region only)   │ │
│ └─────────────────────────────────────────────────────────┘ │
│ Typische Update-Zeit: 2-5ms (nur Score-Zeile)               │
│                       20ms (Full-Screen bei Game Over)      │
└─────────────────────────────────────────────────────────────┘
                              │
T+3.5ms                       ▼
┌─────────────────────────────────────────────────────────────┐
│ SSD1306: OLED-Display Update                                │
│ ┌─────────────────────────────────────────────────────────┐ │
│ │ Controller-Intern:                                      │ │
│ │   • Empfängt I2C-Daten → internem GDDRAM (1024 Bytes)   │ │
│ │   • GDDRAM → Display-Matrix (128×64 Pixel)              │ │
│ │   • Refresh-Rate: ~60Hz (automatisch)                   │ │
│ │   • Kontrast/Helligkeit: Software-konfigurierbar        │ │
│ └─────────────────────────────────────────────────────────┘ │
│ User sieht: Aktualisierte Score-Anzeige                     │
└─────────────────────────────────────────────────────────────┘
```

```
Parallelität:
  • OLED-Update läuft parallel zu LED-Matrix
  • I2C nutzt eigenen Hardware-Peripheral (nicht blockiert RMT)
  • LVGL läuft in separatem Timer-Task (Priorität < GameLoop)
  • Keine Blockierung der GameLogic
  • Score-Update ist selten (~1-2x/min) → geringe Last (<1% CPU)

Timing-Vergleich:
  LED-Matrix:  ~14ms/Frame @ 60 FPS
  OLED:        ~3ms/Update @ ~0.5 Hz (bei Score-Change)
  → OLED-Last ist vernachlässigbar im Vergleich zu LED-Matrix

                    [Loop Iteration N+1]
```

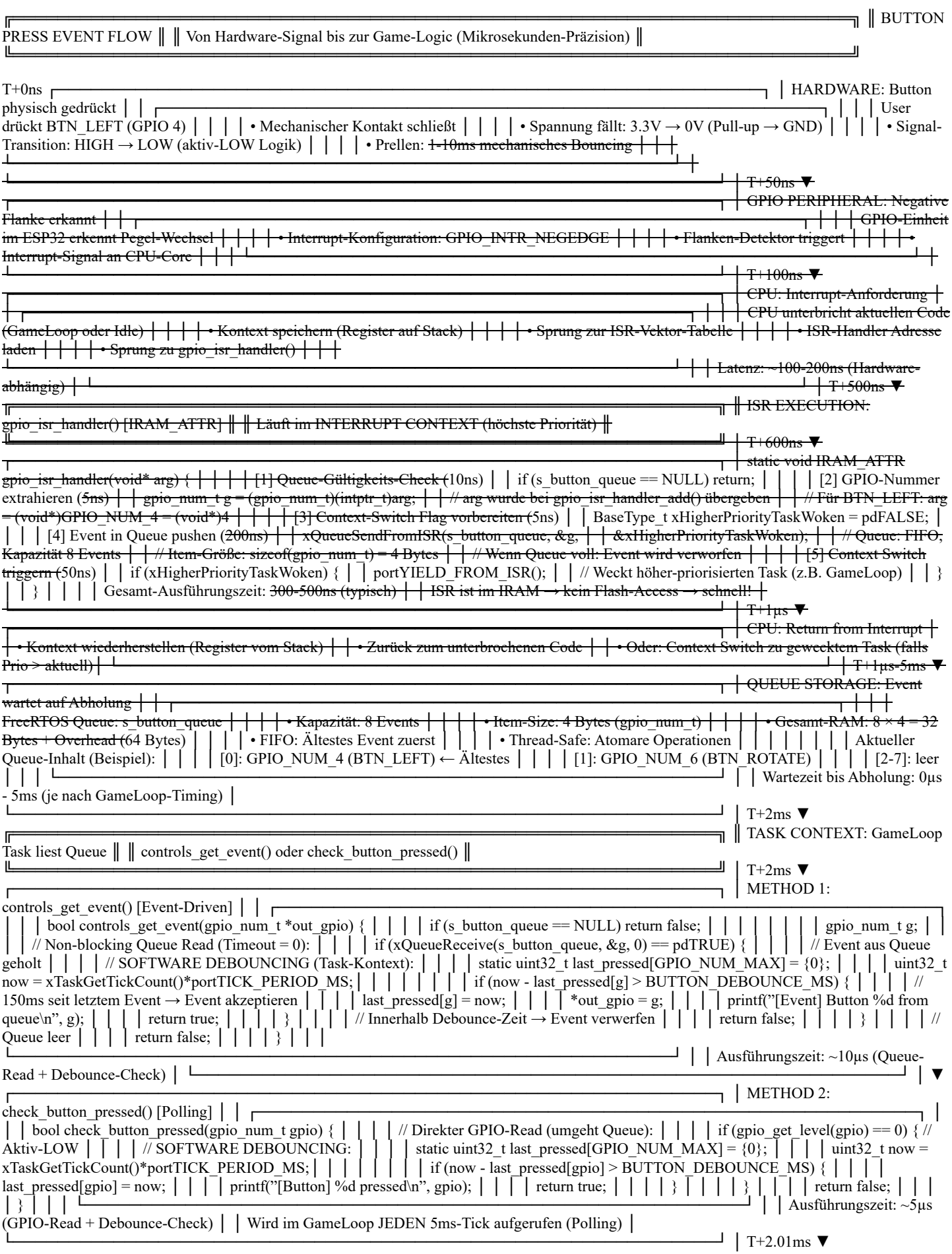
Performance-Analyse pro Loop-Iteration: ═══ Szenario | CPU-Zeit | Anteil | Bemerkung ──|-----------|--------|─────────────────────────────── Minimaler Loop (kein Input) | ~350µs | 7% | Nur Checks + Delay Input-Handling (1 Button) | ~380µs | 7.6% | + Kollisionsprüfung Auto-Fall (Block bewegt) | ~400µs | 8% | + Physics Render-Frame (16ms Timer) | ~5350µs | 107% | RMT-Transfer dominiert Zeilen-Clear (mit Blink) | ~550ms | 11000%!| Animation blockiert Game Over (mit Blink) | ~1500ms | 30000%!| 3× Blink-Sequenz

CPU-Auslastung (Durchschnitt):

- Idle: ~10% (nur Polling, kein Render nötig)
- Normal: ~25% (60 FPS Rendering)
- Peak: ~40% (Zeilen-Clear + Render)

Event-Flow & ISR-Handling

ISR → Queue → Task Pipeline (Detailliert)



┌──┐ │ GAME LOGIC: Input Processing │ │ ┌──┐ │ │ │ // Im GameLoop (game_loop_task): │ │ │ │ if (check_button_pressed(BTN_LEFT)) { │ │ │ │ TetrisBlock tmp = current_block; │ │ │ │ tmp.x--; // 1 Spalte nach links │ │ │ │ if (!grid_check_collision(&tmp)) { │ │ │ │ current_block = tmp; // Bewegung ausführen │ │ │ │ } │ │ │ │ } │ │ │ └──┘ │ └──┘ │ T+2.02ms ▼ ┌──┐ │ RENDER: Block an neuer Position zeichnen │ │ • render_grid() wird im nächsten 16ms-Render-Tick aufgerufen │ │ • Visuelles Feedback für User: Block bewegt sich │ └──┘

╔══╗ ║ TIMING-ZUSAMMENFASSUNG ║ ╚══╝

Button Press → ISR Entry: 100-200 ns (Hardware-Latenz) ISR Execution: 300-500 ns (Event in Queue) ISR → Task wakeup: 1-10 µs (Scheduler) Queue → Game Logic: 0-5 ms (Polling-Intervall) Game Logic → Render: 0-16 ms (Render-Intervall) ── GESAMT (Button → Visuelles Feedback): 2-21 ms (Durchschnitt: ~10ms)

Vergleich:

- Human Reaction Time: ~200ms (Wahrnehmung → Reaktion)
- System Latency: ~10ms (Button → Screen)
- Wahrnehmbar? NEIN (zu schnell für menschliche Wahrnehmung)
- Gefühl: “Instant Response”

╔══╗ ║ DEBOUNCING-STRATEGIE ║ ╚══╝

Problem: Mechanisches Button-Prellen • Button-Kontakt schließt nicht sofort sauber • Signal bounced: HIGH-LOW-HIGH-LOW-... (~1-10ms) • Ohne Debouncing: Mehrere Events pro Press

Lösung: Software-Debouncing (150ms Fenster)

┌──┐ │ Timeline (Button-Press mit Prellen): │ │ │ │ T+0ms: Button gedrückt │ │ ↓ ISR triggered │ │ T+0.5ms: Prellen (Release simuliert) │ │ ↓ ISR triggered (zweites Mal) │ │ T+1ms: Prellen (Press simuliert) │ │ ↓ ISR triggered (drittes Mal) │ │ T+2ms: Kontakt stabil LOW │ │ │ │ Ohne Debouncing: │ │ → 3 Events in Queue │ │ → Block bewegt sich 3× (falsch!) │ │ │ │ Mit Debouncing (150ms Fenster): │ │ → Erstes Event akzeptiert (T+0ms) │ │ → Zweites Event @ T+0.5ms: VERWORFEN (< 150ms) │ │ → Drittes Event @ T+1ms: VERWORFEN (< 150ms) │ │ → Nächstes mögliches Event: T+150ms │ │ │ │ Result: Block bewegt sich 1× (korrekt!) │ └──┘

Debounce-Parameter: • BUTTON_DEBOUNCE_MS = 150ms (in Globals.h) • Tuning: 100ms zu kurz (Doppel-Events), 200ms zu lang (träge) • 150ms: Optimal für responsive Gameplay

╔══╗ ║ QUEUE OVERFLOW HANDLING ║ ╚══╝

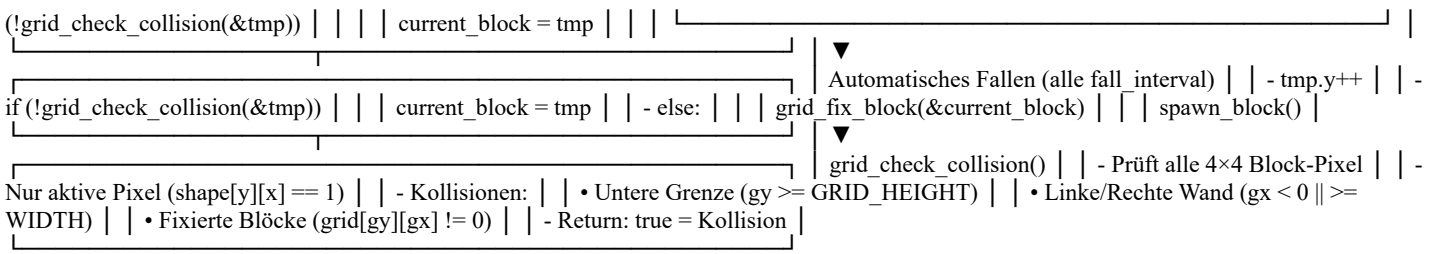
Queue Kapazität: 8 Events • Typische Nutzung: 0-2 Events gleichzeitig • Overflow-Szenario: Extrem schnelles Button-Mashing

Overflow-Verhalten: ┌──┐ │ if (Queue voll): │ │ xQueueSendFromISR() → pdFALSE (Event wird VERWORFEN) │ │ → ISR kehrt sofort zurück │ │ → Älteste Events bleiben in Queue │ │ → Neuere Events gehen verloren │ │ │ │ Auswirkung auf Gameplay: │ │ → Button-Press wird nicht registriert │ │ → User muss erneut drücken │ │ → Selten problematisch (GameLoop leert Queue alle 5ms) │ └──┘

Prevention: • Queue-Größe 8 Events (großzügig für 4 Buttons) • GameLoop pollt alle 5ms → Queue wird schnell geleert • Debouncing reduziert Event-Rate zusätzlich

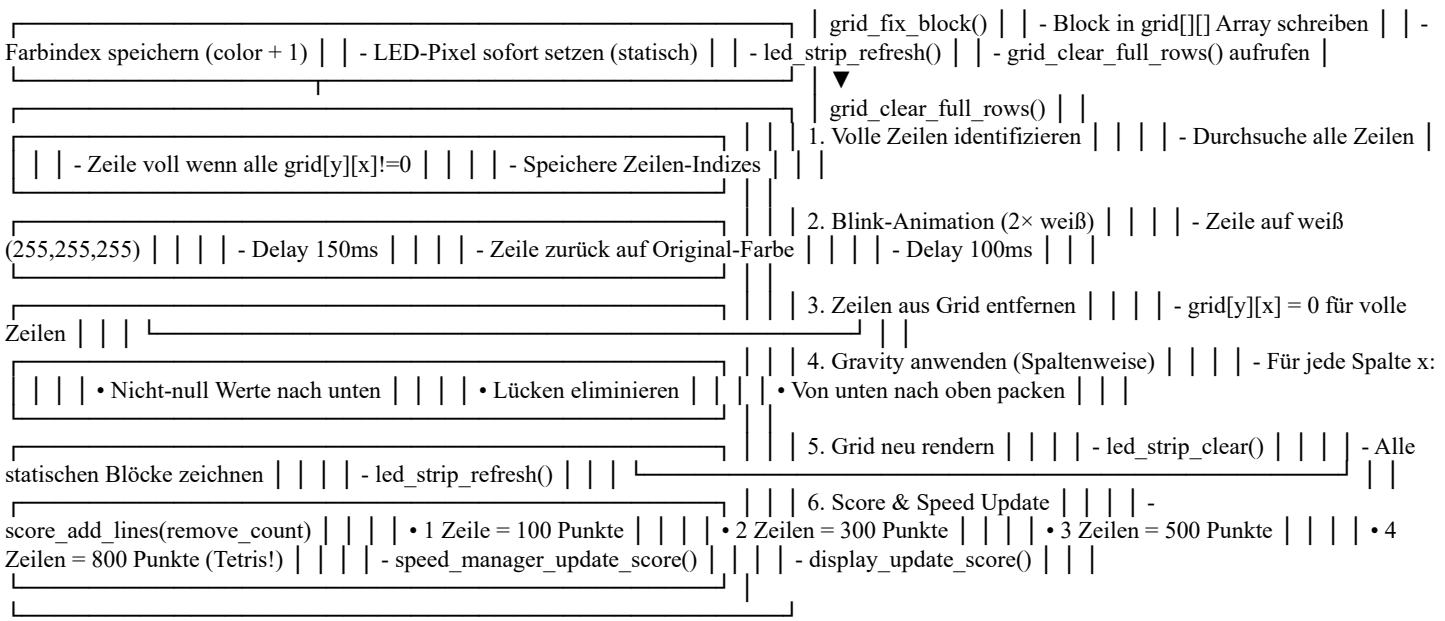
```
---

### 5. Block Physics & Collision Detection
```

┌──────────────────────────────┐ │ spawn_block() │ │ - Zufälligen Block-Typ wählen (0-6) │ │ - Spawn-Position finden (Mitte, oben) │ │ - Kollisionsprüfung beim Spawn │ │ - Kein Platz? → handle_game_over() │ └──────────────┬───────────────┘ │ ▼ ┌──────────────────────────────┐ │ Movement (jeden Loop, 5ms Polling) │ │ ┌──────────────────────────┐ │ │ │ BTN_LEFT/RIGHT: │ │ │ │ - TetrisBlock tmp = current_block │ │ │ │ - tmp.x += direction │ │ │ │ - if (!grid_check_collision(&tmp)) │ │ │ │ current_block = tmp │ │ │ └──────────────────────────┘ │ │ ┌──────────────────────────┐ │ │ │ BTN_ROTATE: │ │ │ │ - rotate_block_90(&tmp) │ │ │ │ - if (!grid_check_collision(&tmp)) │ │ │ │ current_block = tmp │ │ │ │ - O-Block (Typ 3) rotiert NICHT │ │ │ └──────────────────────────┘ │ │ ┌──────────────────────────┐ │ │ │ BTN_FASTER: │ │ │ │ - tmp.y++ │ │ │ │ - if



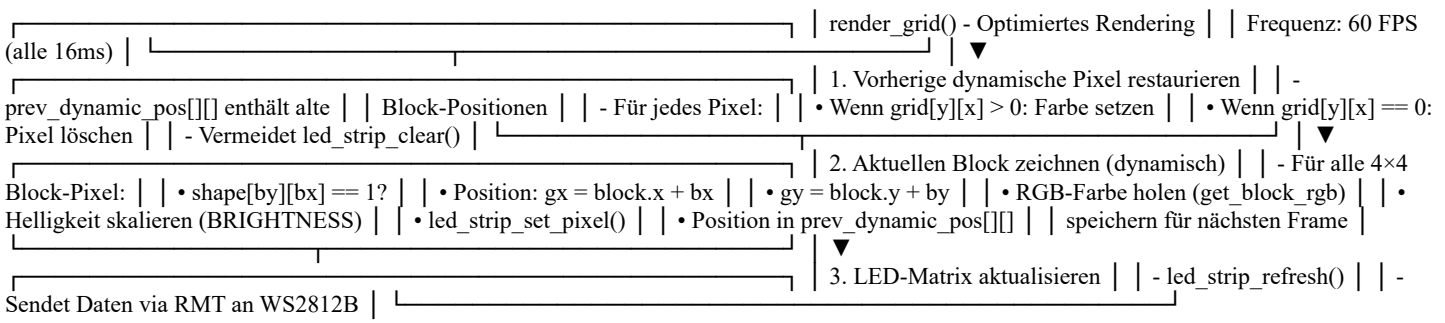
---

6. Grid & Row Clearing



---

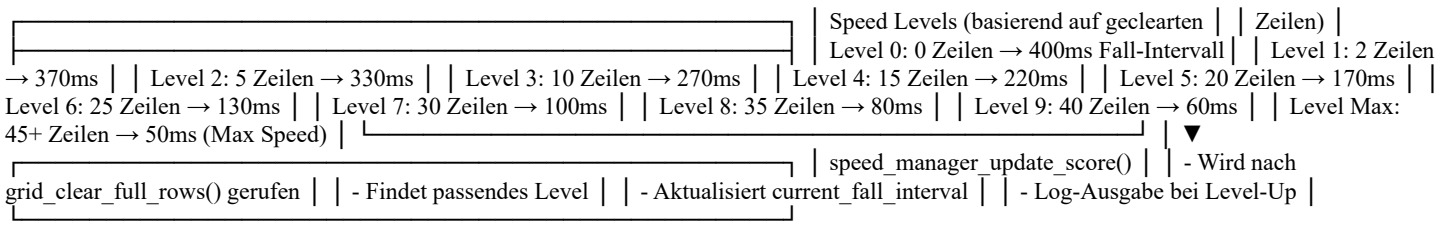
7. Rendering System



Optimierung:
- Nur geänderte Pixel werden geschrieben
- Kein vollständiges `led_strip_clear()` jeden Frame
- Statische Blöcke bleiben erhalten
- Dynamischer Block wird gezeichnet und nächsten Frame restauriert

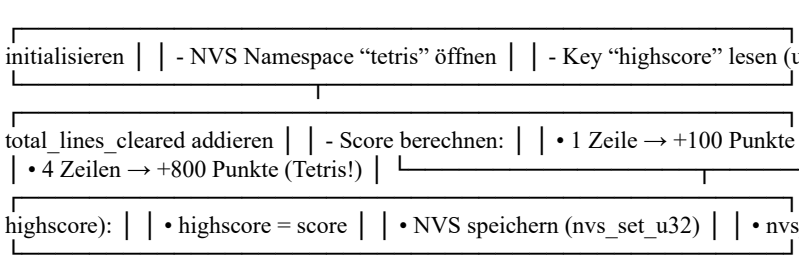
---

8. Speed Progression System



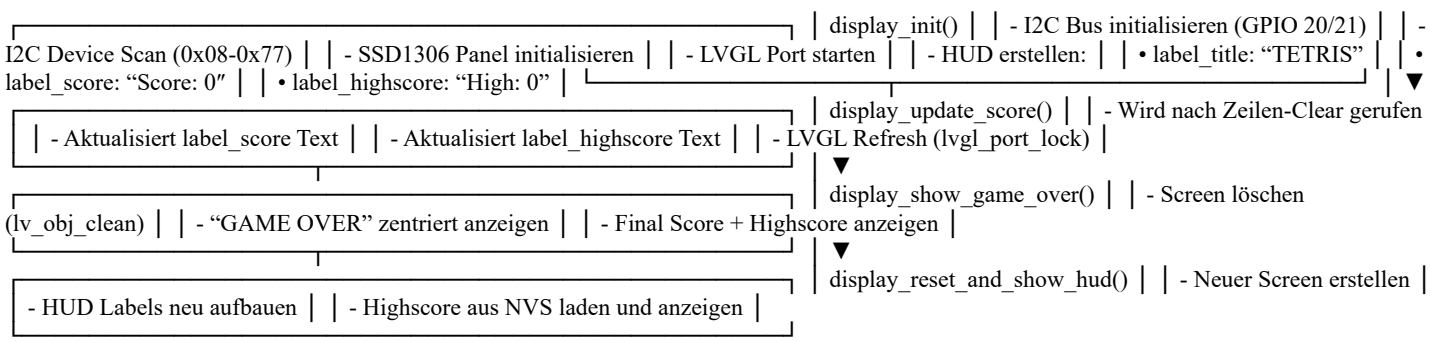
---

9. Score & Highscore (NVS Persistence)



---

10. Display (OLED) - HUD System



Timing & Performance

Task-Prioritäten
- **GameLoop Task:** Priorität 5 (hoch)
- **Theme Task:** Priorität 1 (niedrig, Hintergrund)
- **LVGL Timer:** Standard-Priorität
- **Idle Task:** Priorität 0

Timing-Parameter (aus Globals.h)

Parameter	Wert	Beschreibung
`LOOP_INTERVAL_MS`	5ms	Game Loop Polling-Rate
`RENDER_INTERVAL_MS`	16ms	LED Refresh (60 FPS)
`FALL_INTERVAL_MS`	400ms (initial)	Block-Fall-Geschwindigkeit
`BUTTON_DEBOUNCE_MS`	150ms	Software-Debounce für Buttons
`SPLASH_SCROLL_DELAY_MS`	40ms	Splash-Text Scroll-Geschwindigkeit
`GAME_OVER_BLINK_ON_MS`	300ms	Blink An-Zeit bei Game Over
`GAME_OVER_BLINK_OFF_MS`	300ms	Blink Aus-Zeit bei Game Over

Performance-Metriken
- **Input Latenz:** < 5ms (5ms Polling)
- **Render Latenz:** 16ms (60 FPS)
- **ISR Latenz:** < 1µs (minimales ISR, nur Queue Push)
- **LED Update Zeit:** ~5ms für 384 LEDs via RMT

Datenstrukturen

TetrisBlock (Block-Datenstruktur)
```c
typedef struct {
    uint8_t shape[4][4];  // 1 = Blockteil, 0 = leer
    int x;                // X-Position (linke obere Ecke)
    int y;                // Y-Position (obere Zeile)
    uint8_t color;        // Farbindex (0-6)
} TetrisBlock;
```

MATRIX (LED-Mapping)

```
typedef struct {
    uint16_t LED_Number[24][16];  // LED-Nummern für jede Position
    uint8_t red;                  // Rot-Wert (0-255)
    uint8_t green;                // Grün-Wert (0-255)
    uint8_t blue;                 // Blau-Wert (0-255)
    bool isFilled;                // Ist Pixel gefüllt?
} MATRIX;
```

Grid (Spielfeld-Array)

```
uint8_t grid[24][16];  // 0 = leer, 1-7 = Block-Farbe
```

Block-Typen & Farben

```
enum BlockType {
    BLOCK_I = 0,  // Cyan
    BLOCK_J = 1,  // Blau
    BLOCK_L = 2,  // Orange
    BLOCK_O = 3,  // Gelb
    BLOCK_S = 4,  // Grün
    BLOCK_T = 5,  // Lila
    BLOCK_Z = 6   // Rot
};
```

Detaillierter Programmablauf

Boot-Sequenz (0-5 Sekunden)

```
Zeit    Event
─────   ──────────────────────────────────────────────────────────
0.0s    ESP32 Boot, FreeRTOS Start
0.1s    app_main() Entry
0.2s    LED-Strip initialisiert (RMT)
0.3s    LED-Matrix Mapping geladen
0.4s    Random Seed generiert
0.5s    Splash-Animation startet (scrollender Text "TETRIS")
        - Hintergrund: Statische Tetromino-Grafik
        - Vordergrund: Weißer scrollender Text
4.5s    Splash endet nach 4000ms
4.6s    Grid & Score initialisiert
4.7s    I2C Display initialisiert
        - Device Scan (0x08-0x77)
        - SSD1306 gefunden auf 0x3C
        - LVGL initialisiert
        - HUD erstellt (Titel, Score, Highscore)
4.8s    Highscore aus NVS geladen (z.B. 1200)
4.9s    Button-Controls initialisiert
        - GPIO 4,5,6,7 als Input mit Pull-up
        - ISR installiert (Negative Flanke)
        - Event-Queue erstellt (8 Events)
5.0s    Tetris-Theme Task gestartet (Hintergrund-Musik)
5.1s    GameLoop Task gestartet (Priorität 5)
5.2s    System läuft → WAIT State
        - Splash scrollt endlos
        - Wartet auf Button-Press
```

Spiel-Sequenz (normal)

```
Zeit    Event
─────   ──────────────────────────────────────────────────────────
0.0s    Button gedrückt (z.B. BTN_ROTATE)
        - ISR triggered → GPIO_NUM_6 in Queue
```

```
        - controls_get_event() liest Queue
        - Debounce-Check: 150ms seit letztem Event?
        - Event akzeptiert

0.01s   Splash beendet (splash_show returns)
        - Queue gedrained (alte Events entfernt)
        - Warten bis alle Buttons released
        - Neues Event warten (controls_wait_event)

0.5s    Neuer Button-Press erkannt
        - splash_clear() → LED-Matrix löschen
        - grid_init() → Grid auf 0 setzen
        - score_init() → Score = 0
        - speed_manager_reset() → 400ms Fall-Intervall
        - display_reset_and_show_hud()
        - spawn_block() → Ersten Block spawnen

0.51s   GameLoop: RUNNING State
        last_fall_time = 0.51s
        last_render_time = 0.51s

0.51s   Loop 1 (LOOP_INTERVAL_MS = 5ms)
+0.005s - Emergency Reset Check: Nein
        - Game Over Check: Nein
        - Running? Ja
        - Input Check:
          • BTN_LEFT: Nein
          • BTN_RIGHT: Nein
          • BTN_ROTATE: Nein
          • BTN_FASTER: Nein
        - Auto-Fall Check: Noch nicht (< 400ms)
        - Render Check: Noch nicht (< 16ms)
        - vTaskDelay(5ms)

0.52s   Loop 2
+0.005s - (gleiche Checks wie Loop 1)

0.526s  Loop 3
        - Render Check: Ja! (16ms vergangen)
        - render_grid() aufgerufen
          • Vorherige dynamische Pixel restaurieren
          • Aktuellen Block zeichnen
          • led_strip_refresh()
        - last_render_time = 0.526s

...     (weitere Loops mit 5ms Intervall)

0.7s    User drückt BTN_RIGHT
        - ISR triggered → GPIO_NUM_5 in Queue
        - Nächster Loop:
          • check_button_pressed(BTN_RIGHT): true
          • tmp = current_block; tmp.x++
          • grid_check_collision(&tmp): false (kein Hindernis)
          • current_block = tmp → Block bewegt sich rechts

0.91s   Auto-Fall Timer abgelaufen (400ms seit 0.51s)
        - tmp = current_block; tmp.y++
        - grid_check_collision(&tmp): false
        - current_block = tmp → Block fällt 1 Zeile

1.31s   Auto-Fall (alle 400ms)
        - Block fällt weiter

...     (Block fällt und wird bewegt)

5.5s    Auto-Fall: Kollision erkannt
        - tmp.y++ würde kollidieren
        - grid_fix_block(&current_block)
          • Block in grid[][] Array schreiben
          • LED-Pixel setzen (statisch)
          • grid_clear_full_rows() aufrufen
            ◦ Zeile 23 (unterste) ist voll!
            ◦ Blink-Animation (2× weiß, 150ms on / 100ms off)
            ◦ Zeile löschen
```

```
            ◦ Gravity anwenden (Blöcke fallen)
            ◦ score_add_lines(1) → +100 Punkte
            ◦ speed_manager_update_score(1)
            ◦ display_update_score(100, highscore)
        - spawn_block() → Neuer Block oben

5.51s   Neuer Block spawned
        - Spiel läuft weiter mit neuem Block

...     (Spieler spielt weiter, mehr Zeilen werden gecleart)

20.0s   15 Zeilen gecleart → Speed Level 4
        - fall_interval = 220ms (schneller!)
        - Log: "[SpeedManager] ⚡ LEVEL UP!"

...     (Spiel wird schwieriger)

45.0s   Kein Platz für neuen Block
        - spawn_block() findet keine Position
        - handle_game_over() aufgerufen:
          • score_update_highscore()
            ◦ Highscore in NVS speichern (wenn > alter)
          • display_show_game_over(score, highscore)
          • Blink-Animation (3× rot, 300ms on / 300ms off)
          • game_over_flag = 1

45.5s   Game Over State
        - game_over_flag erkannt
        - Splash anzeigen (2s, Inputs ignoriert)
        - Queue drainieren (500ms kontinuierlich)
        - splash_show_waiting() → warten auf Button
        - Bei Button-Press: Neues Spiel

46.0s   Button gedrückt → Spiel startet neu
        (zurück zu 0.5s Sequenz)
```

Emergency Reset Sequenz

```
Zeit    Event
─────   ──────────────────────────────────────────────────────────

0.0s    Alle 4 Buttons gleichzeitig gedrückt
        - controls_all_buttons_pressed(): true
        - vTaskDelay(1000ms) → 1 Sekunde halten

1.0s    Nach 1s: Immer noch alle gedrückt?
        - controls_all_buttons_pressed(): true
        - Hard Reset ausgelöst!
        - grid_init()
        - score_init()
        - speed_manager_reset()
        - led_strip_clear() + refresh
        - display_reset_and_show_hud()

1.1s    Queue drainieren (alle Events entfernen)
        - while (controls_get_event(&ev)) {}

1.2s    Splash anzeigen (2s, Inputs ignoriert)
        - splash_show_duration(2000)

3.2s    Kontinuierliches Queue-Drain (500ms)
        - while ((now - start) < 500ms):
          • while (controls_get_event(&ev)): drain
          • vTaskDelay(10ms)
          • Log: "Caught straggler event: GPIO X"

3.7s    Warten bis alle Buttons losgelassen
        - while (check_button_pressed(ANY)): delay

3.9s    Splash-Waiting (endlos scrollend)
        - splash_show_waiting()
        - Wartet auf frischen Button-Press
```

```
5.0s    Button gedrückt → Spiel startet
        - Normale Spiel-Sequenz beginnt
```

Modul-Beschreibungen

main.c - System Entry Point

Verantwortung: System-Initialisierung und Koordination
Wichtige Funktionen:

- `setup_led_strip()`: WS2812B RMT-Driver initialisieren
- `app_main()`: Haupteinstiegspunkt
    1. LED-Strip Setup
    2. LED-Matrix Mapping laden
    3. Random Seed generieren
    4. Splash-Animation anzeigen
    5. Grid & Score initialisieren
    6. Display initialisieren (I2C, LVGL)
    7. Controls initialisieren (GPIO, ISR)
    8. Theme-Song starten
    9. GameLoop Task starten
    10. Idle Loop (1s Delay)

GameLoop.c - Spiellogik

Verantwortung: Hauptspielschleife, Zustandsverwaltung
Wichtige Funktionen:

- `game_loop_task()`: FreeRTOS Task (Priorität 5)
    - Polling-Loop mit 5ms Intervall
    - Zustandsmaschine: WAIT / RUNNING / GAME_OVER
    - Input-Verarbeitung (Buttons)
    - Auto-Fall-Mechanik
    - Rendering (60 FPS)
    - Emergency Reset Erkennung
- `render_grid()`: Optimiertes LED-Rendering
    - Nur geänderte Pixel updaten
    - Statische Blöcke bleiben erhalten
- `spawn_block()`: Neuen Block spawnen
    - Zufälligen Typ wählen
    - Spawn-Position finden (Kollisionsfrei)
    - Game Over wenn kein Platz
- `handle_game_over()`: Game Over Logik
    - Highscore speichern
    - Blink-Animation
    - State-Transition zu WAIT

Controls.c - Button-Input System

Verantwortung: GPIO-ISR, Event-Queue, Debouncing
Wichtige Funktionen:

- `init_controls()`: GPIO & ISR Setup
    - GPIO 4,5,6,7 als Input mit Pull-up
    - Interrupt auf Negative Flanke
    - Queue erstellen (8 Events)
    - ISR Handler registrieren
- `gpio_isr_handler()`: ISR-Handler (IRAM)
    - Minimale Verarbeitung im Interrupt
    - GPIO-Nummer in Queue schieben
    - portYIELD_FROM_ISR() bei höherer Priorität
- `controls_get_event()`: Queue-Event auslesen
    - Non-blocking Queue Read
    - Software-Debounce (150ms)
    - Event-Logging (Debug)
- `check_button_pressed()`: GPIO-Polling
    - Direktes gpio_get_level()
    - Software-Debounce per GPIO
- `controls_all_buttons_pressed()`: Reset-Erkennung
    - Prüft alle 4 GPIOs gleichzeitig

Blocks.c - Tetromino-Definitionen

Verantwortung: Block-Formen, Rotation, Farben
Daten:

- `blocks[7][4]`: Array aller 7 Block-Typen mit 4 Rotationen
  - I-Block: 4×1 Linie (Cyan)
  - J-Block: L-Form gespiegelt (Blau)
  - L-Block: L-Form (Orange)
  - O-Block: 2×2 Quadrat (Gelb) - rotiert nicht
  - S-Block: S-Form (Grün)
  - T-Block: T-Form (Lila)
  - Z-Block: Z-Form (Rot) **Funktionen:**
- `assign_block_color()`: Farbe zuweisen
- `get_block_rgb()`: RGB-Werte aus Farbtabelle
- `rotate_block_90()`: Block um 90° drehen (Matrixtransformation)

Grid.c - Spielfeld-Verwaltung

Verantwortung: Kollisionserkennung, Zeilen-Clearing, Gravity
Daten:

- `grid[24][16]`: Spielfeld-Array (0 = leer, 1-7 = Farbe) **Funktionen:**
- `grid_init()`: Grid auf 0 setzen
- `grid_check_collision()`: Kollision prüfen
  - Nur aktive Block-Pixel prüfen
  - Grenzen: Unten, Links, Rechts
  - Fixierte Blöcke
  - Spawn oberhalb erlaubt (y < 0)
- `grid_fix_block()`: Block ins Grid schreiben
  - Farbindex speichern
  - LED-Pixel setzen (statisch)
  - grid_clear_full_rows() aufrufen
- `grid_clear_full_rows()`: Volle Zeilen entfernen
  1. Volle Zeilen identifizieren
  2. Blink-Animation (2× weiß)
  3. Zeilen löschen
  4. Gravity anwenden (spaltenweise)
  5. Grid neu rendern
  6. Score & Speed Update

Splash.c - Startbildschirm

Verantwortung: Splash-Animation, Warte-Screen
Funktionen:

- `splash_show()`: Interactive Splash
  - Scrollender Text “TETRIS”
  - Statischer Tetromino-Hintergrund
  - Bricht bei Button-Press ab
- `splash_show_waiting()`: Warte-Splash
  - Scrollt endlos (kein Auto-Break)
  - 500ms Debounce-Timeout
  - Queue-Drain vor Button-Akzeptanz
  - Bricht nur bei frischem Button ab
- `splash_show_duration()`: Timed Splash
  - Feste Dauer (z.B. 2000ms)
  - Ignoriert Button-Presses
  - Für Reset/Game-Over Sequenz
- `splash_clear()`: Splash löschen

Score.c - Punktesystem

Verantwortung: Score, Zeilen-Tracking, Highscore (NVS)
Funktionen:

- `score_init()`: Score & Zeilen auf 0
- `score_add_lines()`: Zeilen zu Score
  - 1 Zeile: +100 Punkte
  - 2 Zeilen: +300 Punkte
  - 3 Zeilen: +500 Punkte

    - 4 Zeilen: +800 Punkte (Tetris!)
- `score_load_highscore()`: NVS Highscore laden
    - NVS Flash initialisieren
    - Namespace “tetris” öffnen
    - Key “highscore” lesen
- `score_update_highscore()`: Highscore speichern
    - Nur wenn score > highscore
    - NVS schreiben & committen

SpeedManager.c - Geschwindigkeitssystem

Verantwortung: Dynamische Fall-Geschwindigkeit
Daten:

- `speed_levels[]`: Tabelle mit 11 Levels (0-10) **Funktionen:**
- `speed_manager_init()`: Auf Level 0 zurücksetzen
- `speed_manager_get_fall_interval()`: Aktuelles Intervall
- `speed_manager_update_score()`: Level berechnen
    - Basierend auf geclearten Zeilen
    - Findet passendes Level
    - Log bei Level-Up
- `speed_manager_reset()`: Zurück auf Level 0

DisplayInit.c - OLED-Display

Verantwortung: I2C Display, LVGL HUD
Funktionen:

- `display_init()`: I2C & LVGL Setup
    - I2C Bus initialisieren (GPIO 20/21)
    - Device Scan
    - SSD1306 Panel initialisieren
    - LVGL Port starten
    - HUD erstellen (Titel, Score, Highscore)
- `display_update_score()`: Score aktualisieren
- `display_show_game_over()`: Game Over Screen
- `display_reset_and_show_hud()`: HUD neu aufbauen

LedMatrixInit.c - LED-Mapping

Verantwortung: LED-Nummern aus MatrixNummer.h laden
Funktionen:

- `LedMatrixInit()`: Matrix-Array initialisieren
    - Kopiert LED_MATRIX_NUMMER[][] in matrix[][]

ThemeSong.c - Tetris-Musik

Verantwortung: LEDC PWM Buzzer-Ausgabe
Funktionen:

- `StartTheme()`: Theme-Task starten (Hintergrund)
- Spielt Tetris-Melodie in Endlosschleife

Colors.c - Farbtabelle

Verantwortung: Zentrale RGB-Farben für Blöcke
Daten:

- `block_colors[7][3]`: RGB-Werte für 7 Blöcke
    - Werte durch BRIGHTNESSDIV (10) geteilt
    - I: Cyan (0, 255, 80)
    - J: Blau (0, 0, 255)
    - L: Orange (255, 90, 0)
    - O: Gelb (255, 240, 0)
    - S: Grün (0, 255, 0)
    - T: Lila (128, 0, 128)
    - Z: Rot (255, 0, 0)

Globals.h - Konfiguration

Verantwortung: Zentrale Konstanten & Typen
Wichtige Defines:

- `LED_WIDTH`, `LED_HEIGHT`: Matrix-Größe (16×24)
- `GRID_WIDTH`, `GRID_HEIGHT`: Spielfeld-Größe (16×24)
- `FALL_INTERVAL_MS`: Start-Fall-Geschwindigkeit (400ms)
- `RENDER_INTERVAL_MS`: Render-Frequenz (16ms = 60fps)
- `LOOP_INTERVAL_MS`: Polling-Frequenz (5ms)
- `BUTTON_DEBOUNCE_MS`: Debounce-Zeit (150ms)
- `GAME_BRIGHTNESS_SCALE`: LED-Helligkeit (125/255)
- `NUM_BLOCKS`: Anzahl Block-Typen (7)

Zusammenfassung

Dieses Tetris-Spiel ist ein komplexes Echtzeit-System mit folgenden Eigenschaften:

1. **Multitasking:** FreeRTOS Tasks für GameLoop, Theme, LVGL
2. **Interrupt-driven Input:** ISR + Queue für responsive Eingabe
3. **Optimiertes Rendering:** 60 FPS ohne Flicker
4. **Persistente Daten:** NVS Highscore-Speicherung
5. **Dynamische Schwierigkeit:** 11 Speed-Levels
6. **Robuste State Machine:** WAIT / RUNNING / GAME_OVER
7. **Emergency Reset:** 4-Button-Kombination für Hard-Reset

Gesamtarchitektur: Event-driven, modular, erweiterbar
Performance: Stabil, niedrige Latenz, flüssige Animation
Code-Qualität: Gut strukturiert, aber optimierbar (siehe nächste Phase)

Dokumentation erstellt: 25.11.2025
Autor: GitHub Copilot
Projekt: ESP32-S3 Tetris