ESP32-S3 Tetris - Systemdokumentation

1. System-Übersicht

Hardware: ESP32-S3 (Dual-Core @ 240 MHz), 512 KB RAM
Software: ESP-IDF v5.5.1, FreeRTOS v10.5.1
Komponenten:

- LED-Matrix: 16×24 WS2812B (384 LEDs) via RMT/GPIO 1
- OLED: SSD1306 128×64 I2C @ 0x3C (GPIO 20/21)
- Buttons: 4× GPIO (4,5,6,7) mit ISR + Debounce
- Buzzer: LEDC PWM (optional)
- Storage: NVS für Highscore

2. Pin-Belegung

Komponente	Pin(s)	Funktion
LED-Matrix	GPIO 1	WS2812B Daten (RMT)
BTN_LEFT	GPIO 4	Block links (Pull-up, aktiv-LOW)
BTN_RIGHT	GPIO 5	Block rechts
BTN_ROTATE	GPIO 6	Block rotieren
BTN_FASTER	GPIO 7	Schneller fallen
OLED SCL	GPIO 20	I2C Clock (400 kHz)
OLED SDA	GPIO 21	I2C Data

3. Software-Architektur

3.1 Modul-Übersicht

Modul	Verantwortung
`main.c`	Initialisierung, Entry Point
`GameLoop.c`	Hauptschleife, State Machine, Rendering (60 FPS)
`Controls.c`	ISR, Event-Queue, Debouncing (150ms)
`Blocks.c`	Tetromino-Definitionen, Rotation, Farben
`Grid.c`	Spielfeld, Kollision, Zeilen-Clear, Gravity
`Score.c`	Punktesystem, NVS Highscore
`SpeedManager.c`	Dynamische Fall-Geschwindigkeit (11 Levels)
`DisplayInit.c`	OLED I2C, LVGL HUD
`Splash.c`	Startbildschirm-Animation
`ThemeSong.c`	Tetris-Musik via PWM
`Globals.h`	Zentrale Konfiguration

3.2 Task-Struktur

Task	Priorität	Stack	Intervall	Funktion
GameLoop Task	5 (hoch)	4096 B	5 ms	Spiellogik, Input, Render
Theme Task	1 (niedrig)	2048 B	-	Hintergrundmusik

Task	Priorität	Stack	Intervall	Funktion
LVGL Timer	Standard	-	-	UI-Updates

4. Programmablauf

4.1 Boot-Sequenz (0-4.5s)

Phase 1: Hardware Init (0-0.4s)

- ESP32 Bootloader Start (ROM → 2nd Stage Bootloader)
- FreeRTOS Kernel Init, CPU auf 240 MHz
- `app_main()` Entry Point
- RMT Channel allokieren (10 MHz, GPIO 1)
- LED-Strip Setup: DMA Buffer (384 LEDs × 3 Bytes = 1152 B)
- LED-Matrix Mapping laden: `LED_MATRIX_NUMMER[24][16]` (768 B)
- Random Seed generieren: `esp_random() ^ seed_counter`
- Alle LEDs ausschalten: `led_strip_clear() + refresh()`

Phase 2: Splash Screen (0.5-4.5s)

- Statischer Hintergrund: Tetromino-Muster rendern
- “TETRIS” Text-Bitmap: 3×5 Font, 6 Zeichen
- Scroll-Animation: 40ms pro Frame (25 FPS)
- Abbruch: Button-Press ODER 4s Timeout
- User-Interaktion möglich ab 0.5s

Phase 3: System Init (4.5-5.0s)

- Grid initialisieren: `grid[24][16] = {0}` (384 Bytes)
- Score & Lines zurücksetzen: `score = 0, lines_cleared = 0`
- **NVS Highscore laden:**
  - `nvs_flash_init()` → Flash-Partition mounten
  - `nvs_open("tetris", NVS_READWRITE)` → Namespace öffnen
  - `nvs_get_u32("highscore", &hs)` → Highscore lesen
  - Falls nicht vorhanden: `highscore = 0` (erste Verwendung)
- **I2C Display Init:**
  - I2C Bus: GPIO 20 (SCL), GPIO 21 (SDA), 400 kHz
  - Device Scan: 0x08-0x77 (suche 0x3C)
  - SSD1306 Panel Init: 128×64 Monochrom
  - LVGL Port starten + HUD erstellen
  - Labels: “TETRIS” (Titel), “Score: 0″, “High: XXXX”
  - Graceful Degradation: Falls kein Display → `g_disp = NULL`
- **Button Controls Init:**
  - GPIO 4,5,6,7: Input-Mode mit Pull-up
  - Interrupt: Negative Flanke (NEGEDGE)
  - FreeRTOS Queue: 8 Events (4 Bytes pro Event)
  - ISR Service: `gpio_install_isr_service()`
  - Handler registrieren: `gpio_isr_handler_add()` für jeden Button
- **Theme Task:** Hintergrundmusik starten (Priorität 1, 2048 B Stack)
- **GameLoop Task:** Hauptspiel-Task (Priorität 5, 4096 B Stack)

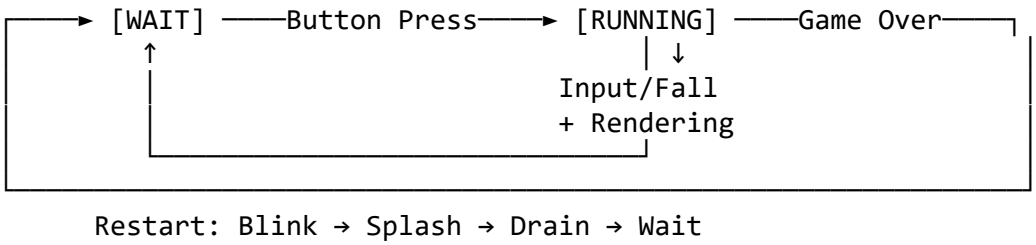
Phase 4: WAIT State (ab 5.0s)

- Splash scrollt endlos (keine Auto-Break)
- Warte auf Button-Press (beliebige Taste)
- Queue wird kontinuierlich überwacht
- Bei Button: Transition zu RUNNING State

4.2 Game Loop (5ms Polling)

```
Emergency Reset Check → Game Over Check → Input Handling
        ↓                       ↓                  ↓
  All 4 BTN?          game_over_flag?   BTN_LEFT/RIGHT/ROTATE/FASTER
        ↓                       ↓                  ↓
   Reset Game          Restart Sequence     Block bewegen/rotieren
                                                   ↓
                    Auto-Fall (Zeit-basiert: 400ms initial)
                                ↓
                    Kollision unten?
                  ↙                  ↘
              Ja                      Nein
               ↓                       ↓
          Fix Block              Block fällt
               ↓
         Clear Rows? → Score + Speed Update
               ↓
        Spawn neuer Block
               ↓
        Kein Platz? → Game Over
               ↓
          Rendering (16ms @ 60 FPS)
               ↓
     Delta-Rendering + RMT (14ms)
```

4.3 State Machine



5. Rendering & Timing

5.1 LED-Matrix Rendering (60 FPS)

Phase	Dauer	Details
Delta-Rendering	0.5ms	Vorherige Block-Pixel restaurieren
Block zeichnen	2.0ms	Aktueller Block mit Brightness-Skalierung
RMT-Setup	0.1ms	DMA-Vorbereitung
RMT-Transfer	11.5ms	384 LEDs × 24 Bits @ 1.25µs/Bit
Gesamt	**14ms**	**Innerhalb 16.67ms @ 60 FPS**

WS2812B Timing: Bit 0: 0.4µs HIGH + 0.85µs LOW | Bit 1: 0.8µs HIGH + 0.45µs LOW
Optimierung: ✅ Delta-Rendering, ✅ RMT+DMA, ❌ Blocking-RMT

5.2 OLED Display (bei Score-Änderung)

Phase	Dauer	Details
LVGL Update	0.5ms	Text-Formatierung + Framebuffer
Font-Rendering	0.4ms	Glyph-Bitmap (Montserrat 14pt)
I2C-Transfer	2.5ms	Partial-Update (Score-Zeile @ 400kHz)

Phase	Dauer	Details
Gesamt	**3.5ms**	**Nur bei Zeilen-Clear (~1-2x/min)**

Parallelität: OLED blockiert LED-Matrix nicht (separater I2C-Peripheral)

6. Input-System (ISR + Queue)

6.1 Event-Flow

```
Button Press → ISR (300-500ns) → Queue → GameLoop (5ms Poll) → Game Logic
     ↓              ↓                ↓              ↓                ↓
 Flanke LOW   GPIO Event    8-Event FIFO   Debounce Check   Block bewegen
```

Latenz: Button → Visuelles Feedback: ~10ms (nicht wahrnehmbar)
Debounce: 150ms Software-Filter (Prellen eliminiert)

6.2 Emergency Reset

- **Trigger:** Alle 4 Buttons gleichzeitig 1s halten
- **Aktion:** Hard Reset → Splash → Warte auf Button → Neues Spiel

7. Spielmechanik

7.1 Block-Spawn & Initialisierung

Spawn-Prozess:

1. Zufälliger Block-Typ wählen: `rand() % 7` → 0-6
2. Spawn-Position berechnen:
   - **X-Position:** Mitte bevorzugt: `x = (GRID_WIDTH - 4) / 2` = 6
   - **Y-Position:** Oberhalb Grid: `y = -2` (für 4-Zeilen-Blöcke)
   - I-Block (4×1 vertikal) spawned bei y=-3 für vollständige Sichtbarkeit
3. Block-Struktur initialisieren:
   - `shape[4][4]` aus `blocks[type][rotation]` kopieren
   - `color` aus Block-Typ ableiten (0=Cyan, 1=Blau, etc.)
   - Initial-Rotation: 0 (Standardausrichtung)
4. **Kollisionsprüfung beim Spawn:**
   - Prüfe ob Spawn-Position frei ist
   - Falls blockiert: `handle_game_over()` → Game Over Sequenz
   - Falls frei: Block wird aktiv und fällt

Spawn-Frequenz: Neuer Block nach Fixierung (Kollision unten) + Zeilen-Clear

7.2 Block-Typen (Tetrominos)

Typ	Form	Farbe	RGB	Rotation
I	4×1 Linie	Cyan	0,255,80	2 Zustände
J	L gespiegelt	Blau	0,0,255	4 Zustände
L	L-Form	Orange	255,90,0	4 Zustände
O	2×2 Quadrat	Gelb	255,240,0	**Nicht**
S	S-Form	Grün	0,255,0	2 Zustände
T	T-Form	Lila	128,0,128	4 Zustände
Z	Z-Form	Rot	255,0,0	2 Zustände

7.2 Kollisionserkennung

- **Grenzen:** Unten (y ≥ 24), Links (x < 0), Rechts (x ≥ 16)
- **Fixierte Blöcke:** `grid[y][x] != 0`
- **Spawn-Zone:** Oberhalb Grid erlaubt (y < 0) für hohe Blöcke

7.3 Zeilen-Clear (Detailliert)

Schritt 1: Volle Zeilen identifizieren

- Durchsuche Grid von unten nach oben (y=23 bis y=0)
- Zeile gilt als voll wenn: `for (x=0; x<16; x++) grid[y][x] != 0`
- Speichere Zeilen-Indizes in Array: `int full_rows[4]` (max. 4 gleichzeitig)
- Anzahl zählen: `remove_count` (1-4 möglich)

Schritt 2: Blink-Animation

- **Loop 2×:** (Visuelles Feedback für User)
  - Alle vollen Zeilen auf weiß setzen: `led_strip_set_pixel(led, 255, 255, 255)`
  - `led_strip_refresh()` → LEDs leuchten weiß
  - `vTaskDelay(150ms)` → Warte
  - Original-Farben restaurieren: `get_block_rgb(grid[y][x])`
  - `led_strip_refresh()` → LEDs zurück auf Original
  - `vTaskDelay(100ms)` → Warte
- Gesamt-Dauer: 2× (150ms + 100ms) = 500ms

Schritt 3: Zeilen aus Grid löschen

- Für jede volle Zeile: `for (x=0; x<16; x++) grid[y][x] = 0`
- LEDs der Zeilen ausschalten: `led_strip_set_pixel(led, 0, 0, 0)`

Schritt 4: Gravity anwenden (spaltenweise)

- **Für jede Spalte x (0-15):**
  - Nicht-null Werte nach unten "packen"
  - Lücken eliminieren (entstanden durch gelöschte Zeilen)
  - Von unten nach oben durchgehen:
    - Finde nächsten nicht-null Wert oberhalb Lücke
    - Verschiebe nach unten: `grid[target_y][x] = grid[source_y][x]`
    - Alte Position leeren: `grid[source_y][x] = 0`
- Ergebnis: Blöcke fallen in Lücken, keine "schwebenden" Blöcke

Schritt 5: Grid komplett neu rendern

- `led_strip_clear()` → Alle LEDs aus
- Für alle Grid-Positionen: Falls `grid[y][x] != 0` → LED setzen
- `led_strip_refresh()` → Zeige neuen Grid-Zustand

Schritt 6: Score & Speed Update

- `score_add_lines(remove_count)`:
  - 1 Zeile: +100 Punkte
  - 2 Zeilen: +300 Punkte (Bonus!)
  - 3 Zeilen: +500 Punkte (Großer Bonus!)
  - 4 Zeilen: +800 Punkte (TETRIS! - Maximaler Bonus)
- `total_lines_cleared += remove_count` → Globaler Zähler
- `speed_manager_update_score(total_lines_cleared)`:
  - Findet passendes Speed-Level basierend auf Zeilen
  - Aktualisiert `current_fall_interval`
  - Log bei Level-Up: "[SpeedManager] ⚡ LEVEL UP! Level X"
- `display_update_score(score, highscore)`:
  - LVGL Label aktualisieren: `lv_label_set_text_fmt()`

- I2C Partial-Update (nur Score-Zeile, ~2.5ms)

Performance: Zeilen-Clear dauert ~500ms (Animation blockiert GameLoop)

7.4 Speed-Progression

Level	Zeilen gecleart	Fall-Intervall
0	0	400 ms
1	2	370 ms
2	5	330 ms
3	10	270 ms
4	15	220 ms
5	20	170 ms
6	25	130 ms
7	30	100 ms
8	35	80 ms
9	40	60 ms
10	45+	50 ms (Max)

8. Persistenz (NVS)

- **Namespace:** “tetris”
- **Key:** “highscore” (uint32_t)
- **Speichern:** Bei Game Over, wenn `score > highscore`
- **Laden:** Bei Boot → HUD anzeigen

9. Wichtige Timing-Parameter

Parameter	Wert	Beschreibung
`LOOP_INTERVAL_MS`	5 ms	GameLoop Polling-Rate
`RENDER_INTERVAL_MS`	16 ms	LED Refresh (60 FPS)
`FALL_INTERVAL_MS`	400 ms	Start-Fall-Geschwindigkeit
`BUTTON_DEBOUNCE_MS`	150 ms	Software-Debounce
`GAME_BRIGHTNESS_SCALE`	49%	LED-Helligkeit (125/255)

10. Performance-Metriken

10.1 Loop-Iteration (5ms)

Szenario	CPU-Zeit	Bemerkung
Minimaler Loop	350 µs	Nur Checks + Delay
Input-Handling	380 µs	+ Kollisionsprüfung
Auto-Fall	400 µs	+ Physics
Render-Frame	5350 µs	RMT-Transfer dominiert
Zeilen-Clear	550 ms	Animation blockiert
Game Over	1500 ms	3× Blink-Sequenz

10.2 CPU-Auslastung

- **Idle:** ~10% (nur Polling)
- **Normal:** ~25% (60 FPS Rendering)

- **Peak:** ~40% (Zeilen-Clear + Render)

11. Datenstrukturen & Speicherverwaltung

11.1 TetrisBlock (Dynamischer Block)

```
typedef struct {
    uint8_t shape[4][4];  // 1=Blockteil, 0=leer (16 Bytes)
    int x, y;             // Position: linke obere Ecke (8 Bytes)
    uint8_t color;        // Farbindex 0-6 (1 Byte)
} TetrisBlock;           // Gesamt: 25 Bytes
```

Verwendung:

- Globale Variable: `TetrisBlock current_block` (im GameLoop)
- Temporäre Kopie: `TetrisBlock tmp` (für Kollisionsprüfung vor Bewegung)
- `shape[4][4]`: Binäre Matrix, 1=Pixel gefüllt, 0=leer
- Beispiel I-Block vertikal: `{0,1,0,0}, {0,1,0,0}, {0,1,0,0}, {0,1,0,0}`

11.2 Grid (Spielfeld-Array)

```
uint8_t grid[24][16];  // 0=leer, 1-7=Block-Farbe (384 Bytes)
```

Speicher-Layout:

- Zeilen (y-Achse): 0 (oben) bis 23 (unten)
- Spalten (x-Achse): 0 (links) bis 15 (rechts)
- Werte: 0=leer, 1-7=fixierte Block-Farbe (color+1)
- Beispiel: `grid[23][0] = 5` → unterste Zeile, linke Ecke, T-Block (Lila)

11.3 LED-Matrix Mapping (MATRIX Struktur)

```
typedef struct {
    uint16_t LED_Number[24][16];  // LED-Index für jede Grid-Position
    uint8_t red, green, blue;     // RGB-Wert (0-255)
    bool isFilled;                // Ist Pixel aktiv?
} MATRIX;                         // Array: matrix[24][16]
```

Verdrahtungs-Layout: Serpentine (Zick-Zack)

- Gerade Spalten (x=0,2,4...): LEDs von unten nach oben
- Ungerade Spalten (x=1,3,5...): LEDs von oben nach unten
- Berechnung: `led_num = (x % 2 == 0) ? x*24 + (23-y) : x*24 + y`
- Beispiel: Position [0,0] (oben links) → LED 23, Position [1,0] → LED 24

11.4 Block-Definitionen (Konstante Arrays)

```
// 7 Block-Typen × 4 Rotationen × 4×4 Matrix = 448 Bytes (Flash)
const uint8_t blocks[7][4][4][4];

// Beispiel: L-Block (Typ 2), Rotation 0
blocks[2][0] = {
    {0,1,0,0},
```

```
    {0,1,0,0},
    {0,1,1,0},
    {0,0,0,0}
};
```

11.5 Farbtabelle (RGB-Werte)

```
const uint8_t block_colors[7][3] = {
    {0, 255, 80},    // I: Cyan
    {0, 0, 255},     // J: Blau
    {255, 90, 0},    // L: Orange
    {255, 240, 0},   // O: Gelb
    {0, 255, 0},     // S: Grün
    {128, 0, 128},   // T: Lila
    {255, 0, 0}      // Z: Rot
};
```

Brightness-Skalierung: RGB-Werte × `GAME_BRIGHTNESS_SCALE` (125/255 = 49%)

11.6 Speicher-Übersicht (Heap-Allokation)

Komponente	Größe	Typ	Bemerkung
Grid	384 B	Global	Spielfeld-Array
LED-Matrix Mapping	768 B	Global	LED_Number Array
RMT DMA Buffer	~1152 B	Heap	384 LEDs × 3 Bytes (GRB)
LVGL Framebuffer	1024 B	Heap	OLED 128×64 @ 1bpp
Button Queue	64 B	Heap	8 Events × 4 Bytes + Overhead
GameLoop Stack	4096 B	Heap	FreeRTOS Task
Theme Stack	2048 B	Heap	FreeRTOS Task
Gesamt	**~9.5 KB**	-	Von 512 KB RAM verfügbar

Freier Heap nach Boot: ~380 KB (von 512 KB total)

12. Fehlerbehebung

Problem	Ursache	Lösung
LEDs flackern	RMT-Timing falsch	WS2812B-Timing prüfen (Globals.h)
Doppelte Button-Events	Debounce zu kurz	`BUTTON_DEBOUNCE_MS` auf 150ms
Display reagiert nicht	I2C Adresse falsch	Device Scan (0x3C prüfen)
Highscore nicht gespeichert	NVS nicht initialisiert	`nvs_flash_init()` aufrufen
Block spawned außerhalb	Spawn-Kollision nicht geprüft	`spawn_block()` Logik prüfen

13. Game Over & Restart-Sequenz

13.1 Game Over Trigger

Auslöser: Neuer Block kann nicht gespawned werden (Kollision bei Spawn)

Ablauf:

1. `spawn_block()` findet keine freie Position
2. `handle_game_over()` wird aufgerufen

3. Game Over Flag setzen: `game_over_flag = 1`
4. Nächster GameLoop-Tick reagiert auf Flag

13.2 Game Over Sequenz

Phase 1: Highscore speichern

```
if (score > highscore):
    highscore = score
    nvs_set_u32(nvs_handle, "highscore", highscore)
    nvs_commit(nvs_handle)
    ESP_LOGI(TAG, "New highscore saved: %d", highscore)
```

Phase 2: Display aktualisieren

- Screen löschen: `lv_obj_clean(lv_scr_act())`
- "GAME OVER" zentriert anzeigen (große Schrift)
- Final Score: `lv_label_set_text_fmt(label, "Score: %d", score)`
- Highscore: `lv_label_set_text_fmt(label, "High: %d", highscore)`
- I2C Full-Screen Transfer (~20ms)

Phase 3: Blink-Animation

- 3× Blink-Sequenz in ROT (Game Over Farbe)
- Loop 3×:
  - Alle LEDs rot: `led_strip_set_pixel(i, 255, 0, 0)`
  - `led_strip_refresh()` + `vTaskDelay(300ms)` → ON
  - Alle LEDs aus: `led_strip_clear()` + `refresh()`
  - `vTaskDelay(300ms)` → OFF
- Gesamt-Dauer: 3× (300ms + 300ms) = 1800ms

Phase 4: Restart-Vorbereitung

- `game_over_flag = 1` → Loop erkennt
- `game_running = false` → Transition zu WAIT State
- HUD zurücksetzen: `display_reset_and_show_hud(highscore)`

13.3 Restart-Prozess (wait_for_restart)

Schritt 1: Splash anzeigen (2s, keine Inputs)

- `splash_show_duration(2000)` → 2 Sekunden ignorieren alle Buttons
- Verhindert sofortigen Neustart durch noch gedrückte Buttons

Schritt 2: Queue kontinuierlich leeren (500ms)

```
uint32_t start = xTaskGetTickCount() * portTICK_PERIOD_MS;
while ((current_time - start) < 500) {
    gpio_num_t ev;
    while (controls_get_event(&ev)) {
        ESP_LOGI(TAG, "Caught straggler event: GPIO %d", ev);
    }
    vTaskDelay(10ms);
}
```

- Alle noch in Queue wartende Events entfernen
- Verhindert "Ghost-Inputs" vom vorherigen Spiel

Schritt 3: Warten auf Button-Release

```
while (check_button_pressed(BTN_LEFT) ||
       check_button_pressed(BTN_RIGHT) ||
```

```
       check_button_pressed(BTN_ROTATE) ||
       check_button_pressed(BTN_FASTER)) {
    vTaskDelay(20ms);
}
```

- Blockiert bis alle Buttons losgelassen
- Garantiert frischen Button-Press

Schritt 4: Splash-Waiting (endlos)

- `splash_show_waiting()` → Splash scrollt ohne Auto-Break
- Wartet auf frischen Button-Press
- Bei Button: Neues Spiel startet

Schritt 5: Spiel-Neustart

- `splash_clear()` → LEDs löschen
- `reset_game_state()`:
  - `grid_init()` → Grid auf 0
  - `score_init()` → Score = 0, lines = 0
  - `speed_manager_reset()` → Level 0, 400ms Fall-Intervall
- `spawn_block()` → Erster Block
- `game_running = true` → RUNNING State

14. Optimierungen & Best Practices

14.1 Implementierte Optimierungen

Delta-Rendering:

- Nur geänderte Pixel werden neu geschrieben
- Vorherige Block-Position in `prev_dynamic_pos[][]` gespeichert
- Spart ~70% der LED-Operationen pro Frame
- Kein `led_strip_clear()` jeden Frame nötig

Hardware-Beschleunigung:

- RMT Peripheral für WS2812B-Timing (präzise Pulse)
- DMA für LED-Daten-Transfer (CPU-Entlastung während Transfer)
- I2C Hardware-Peripheral (parallel zu RMT, keine Blockierung)

ISR + Queue Pattern:

- Minimale Verarbeitung im Interrupt (300-500ns)
- Event-Queue für asynchrone Verarbeitung
- GameLoop pollt Queue im Task-Kontext
- Software-Debounce außerhalb ISR (150ms)

FreeRTOS Task-Prioritäten:

- GameLoop Prio 5 (höchste) → Responsive Input
- Theme Prio 1 (niedrig) → Stört Gameplay nicht
- LVGL Standard-Prio → UI-Updates bei Bedarf

14.2 Bekannte Limitierungen

Blocking RMT-Transfer:

- `rmt_tx_wait_all_done()` blockiert CPU für 11.5ms
- Alternative: Async RMT mit Callback (nicht implementiert)
- Benötigt Double-Buffering (2× Pixel-Buffer)
- Race-Conditions vermeiden (Rendering während Transfer)

Animation blockiert GameLoop:

- Zeilen-Clear: 500ms (2× Blink)
- Game Over: 1800ms (3× Blink)
- Input während Animation ignoriert
- Könnte mit separatem Animation-Task gelöst werden

Keine Pause-Funktion:

- Spiel läuft kontinuierlich (nur WAIT/RUNNING/GAME_OVER)
- Emergency Reset als Workaround (4 Buttons 1s)
- PAUSED State könnte hinzugefügt werden

14.3 Erweiterungsmöglichkeiten

1. Async RMT (Non-Blocking LED-Updates)

- Double-Buffering: `pixel_buffer[2][1152]`
- Callback bei Transfer-Complete: `rmt_tx_done_callback()`
- CPU-Zeit während Transfer für Physics nutzen
- Implementierungsaufwand: Medium
- Performance-Gewinn: 11.5ms CPU-Zeit pro Frame

2. Next-Block-Preview

- Nächster Block auf OLED anzeigen
- Block in `next_block` Variable speichern
- 4×4 Mini-Grid auf OLED rendern
- Implementierungsaufwand: Niedrig
- Gameplay-Verbesserung: Hoch (strategisches Planen)

3. Pause-Funktion

- Neuer State: `PAUSED`
- Button-Kombination: BTN_ROTATE + BTN_FASTER (1s)
- Pausiert Fall-Timer + Input (außer Resume)
- OLED zeigt “PAUSED” an
- Implementierungsaufwand: Niedrig

4. Multiplayer (2 Matrizen)

- Zweite LED-Matrix: GPIO 2 (separates RMT Channel)
- Synchronisierte Spiellogik (gleicher Block-Seed)
- Zeilen-Clear sendet “Müll-Zeilen” an Gegner
- OLED zeigt beide Scores
- Implementierungsaufwand: Hoch

5. Custom Themes & Farbpaletten

- Mehrere `block_colors[][]` Tabellen
- Theme-Wechsel via Menü
- Alternative Musik-Tracks
- Persistenz in NVS
- Implementierungsaufwand: Niedrig

6. Statistiken & Achievements

- Tracking: Blöcke gespawned, Tetris-Count, Max Combo
- NVS für Langzeit-Statistiken
- OLED Statistik-Screen
- Achievements: “100 Zeilen”, “10 Tetris”, etc.
- Implementierungsaufwand: Medium

15. Debugging & Logging

15.1 ESP-IDF Logging

Log-Levels verwendet:

- `ESP_LOGI()`: Informationen (Boot, Spawn, Level-Up, Highscore)
- `ESP_LOGW()`: Warnungen (NVS-Fehler, Display nicht gefunden)
- `ESP_LOGE()`: Fehler (I2C-Fehler, RMT-Fehler)

Beispiel-Ausgaben:

```
I (5234) GameLoop: [Button] GPIO 4 pressed
I (5678) SpeedManager: ⚡ LEVEL UP! Level 3
I (8901) Score: New highscore saved: 1200
W (1234) DisplayInit: No display found, running without OLED
```

Serial Monitor: 115200 baud, UART via USB

15.2 Performance-Monitoring

FreeRTOS Task-Stats:

```
vTaskList(buffer);  // Task-Namen, Status, Stack-Usage
vTaskGetRunTimeStats(buffer);  // CPU-Auslastung pro Task
```

Heap-Monitoring:

```
esp_get_free_heap_size();        // Aktuell freier Heap
esp_get_minimum_free_heap_size(); // Niedrigster Heap-Stand
```

Typical Values:

- Free Heap nach Boot: ~380 KB
- Min Free Heap während Spiel: ~370 KB
- Stack Highwater GameLoop: ~2.8 KB verwendet (von 4 KB)

Projekt: ESP32-S3 Tetris
Autoren: Rupp Arian, Lampert Mathias
Version: 1.0
Datum: November 2025
ESP-IDF: v5.5.1
Hardware: ESP32-S3-WROOM-1, WS2812B Matrix 16×24, SSD1306 OLED