Hanoi University of Science and Technology
School of Information and Communication Technology

Project Report: Game Tetris using STM32CubeIDE hardware and software

IT4210E - Embedded Systems

Group Members:

Group 5 - Class: 161346
Luong Ngoc Vu Long - 20235967
Tran Sy Nguyen - 20235985
Nguyen Vu Anh Khoa - 20235957

Lecturers:

Prof. Ngô Lam Trung

January 27, 2026

Contents

1 Introduction

The "Embedded Tetris" project aims to replicate the classic arcade experience on the STM32F429I-DISCO development board. By leveraging the high-performance ARM Cortex-M4 microcontroller, this project demonstrates the capability of modern embedded systems to handle real-time game logic, fluid graphical rendering, and multi-tasking.

1.1 Project Goals

The primary objectives of this project are:

- **System Integration:** To successfully integrate FreeRTOS with the TouchGFX framework, creating a responsive and stable application environment where the GUI task and background logic run concurrently.
- **Game Fidelity:** To implement the full functionality of the classic Tetris game, including:
  - **Tetromino Logic:** Spawning of all seven standard Tetromino types (I, J, L, O, S, T, Z) with randomized generation.
  - **Movement and Rotation:** Precise control for horizontal movement, soft dropping, and piece rotation with collision detection.
  - **Matrix Management:** A 10x20 grid system for tracking settled pieces, performing line detection, and implementing line clearing.
  - **Scoring and Progression:** A scoring system based on line clears and a leveling system that increases game speed to provide progressive difficulty.
  - **User Experience:** Integration of a "Hold" piece mechanism, "Next" piece preview, and high score tracking for a complete gameplay experience.

1.2 Project Scope

This project is confined to the following boundaries:

- **Hardware:** Strictly targeted for the STM32F429I-DISCO kit (STM32F429ZIT6 MCU).
- **Input:** Control is limited to the on-board user button and external push buttons (via GPIO), or the capacitive touch screen.
- **Output:** Visual elements are rendered on the built-in 2.4" QVGA display, and audio is output via a piezo buzzer connected to a PWM pin.
- **Software:** The codebase is built using STM32CubeIDE, utilising the HAL library for hardware abstraction and FreeRTOS for task scheduling.

2 Hardware Design

2.1 Development Board Specifications

The project utilizes the STM32F429I-DISCO kit, which is built around the STM32F429ZIT6 microcontroller. Key specifications utilized in this project include:

- **Core:** ARM Cortex-M4 with FPU, running at 168 MHz (configured via PLL).
- **Flash Memory:** 2 MB (Storing code and const assets like fonts/images).
- **RAM:** 256 KB Internal SRAM + 64 Mbits External SDRAM (used for Frame Buffers).
- **Display:** 2.4" QVGA TFT LCD via LTDC interface.

2.2 Peripheral Configuration

The hardware configuration is generated using STM32CubeMX (inside STM32CubeIDE).

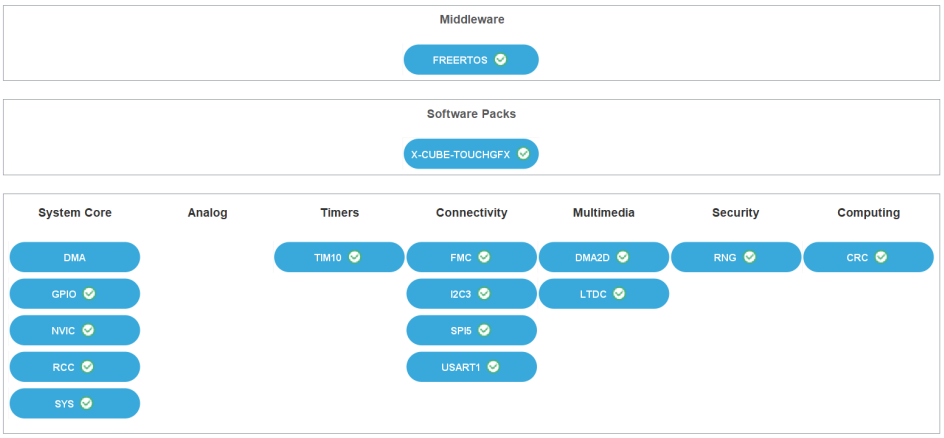

Figure 1: STM32CubeMX System and Peripheral Initialization Overview

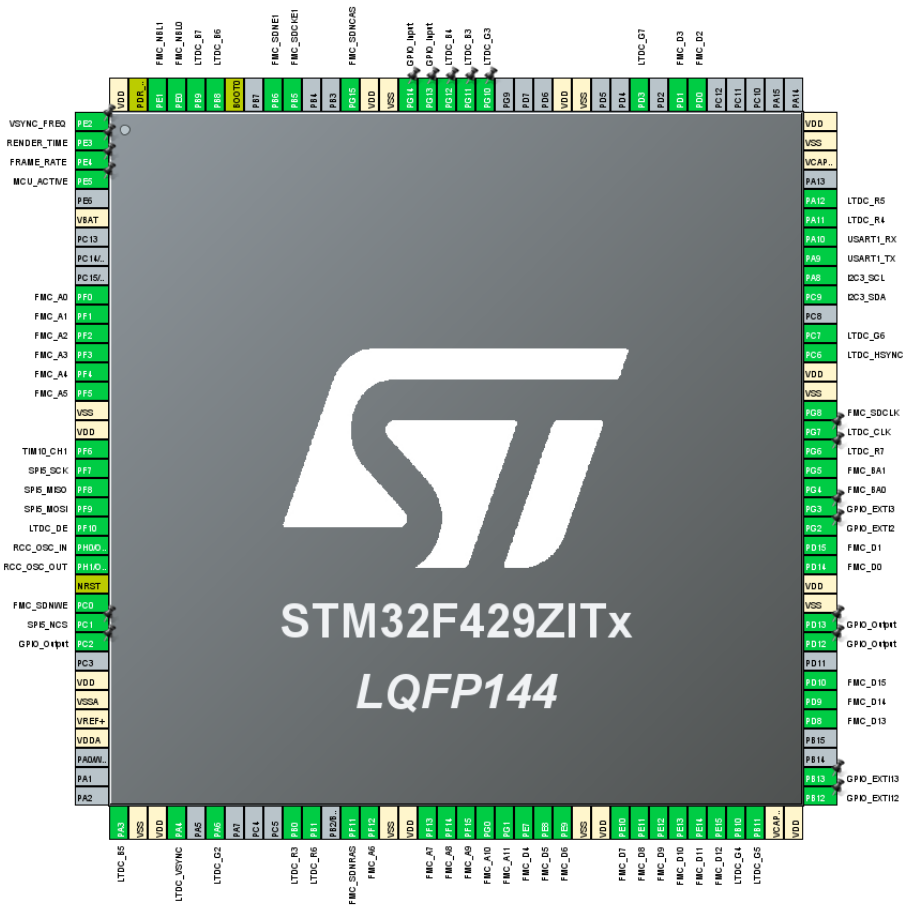

Figure 2: STM32F429ZIT6 Pinout and GPIO Configuration

2.2.1 Display Subsystem (LTDC & DMA2D)

The Liquid Crystal Display (LCD) is driven by the LTDC (LCD-TFT Display Controller) peripheral.

- **Interface:** RGB565 (16-bit color depth).
- **Resolution:** 240 x 320 pixels.
- **Memory:** Two frame buffers are allocated in the external SDRAM to support double buffering, preventing screen tearing.
- **DMA2D:** Chrom-Art Accelerator is enabled to handle image copying and blending efficiently, offloading the CPU.

2.2.2 Audio Subsystem (PWM)

Sound generation is achieved using Pulse Width Modulation (PWM) connected to a Piezo Buzzer.

- **Timer:** TIM10.
- **Channel:** Channel 1.
- **Prescaler:** 167 (resulting in a 1 MHz timer clock).

- **Logic:** The frequency of the PWM signal controls the musical pitch (Note), and the duty cycle controls the volume.

2.2.3 Input Controls (GPIO & EXTI)

The game is controlled via four external push buttons connected to GPIO pins configured as external interrupts (EXTI).

Function	Pin	Port	Mode
Rotate Piece	Pin 12	GPIOB	EXTI Rising/Falling
Move Right	Pin 13	GPIOB	EXTI Rising/Falling
Soft Drop	Pin 2	GPIOG	EXTI Rising/Falling
Move Left	Pin 3	GPIOG	EXTI Rising/Falling

Table 1: GPIO Pin Mapping for Game Controls

2.3 Random Number Generator (RNG)

To ensure fair and unpredictable gameplay, the hardware Tetromino generation relies on the STM32's built-in Random Number Generator (RNG) peripheral. The peripheral requires a precise 48 MHz clock (PLL48CLK) and is configured to provide entropy for the piece spawning logic. Detailed implementation challenges regarding the clock tree configuration are discussed in Section 4.1.

3 Software Design

3.1 Model-View-Presenter (MVP) Architecture

The software architecture is centered around the Model-View-Presenter (MVP) pattern provided by the TouchGFX framework. This pattern ensures a strict separation of concerns between the visual interface, the underlying game logic, and the synchronization layer that connects them.

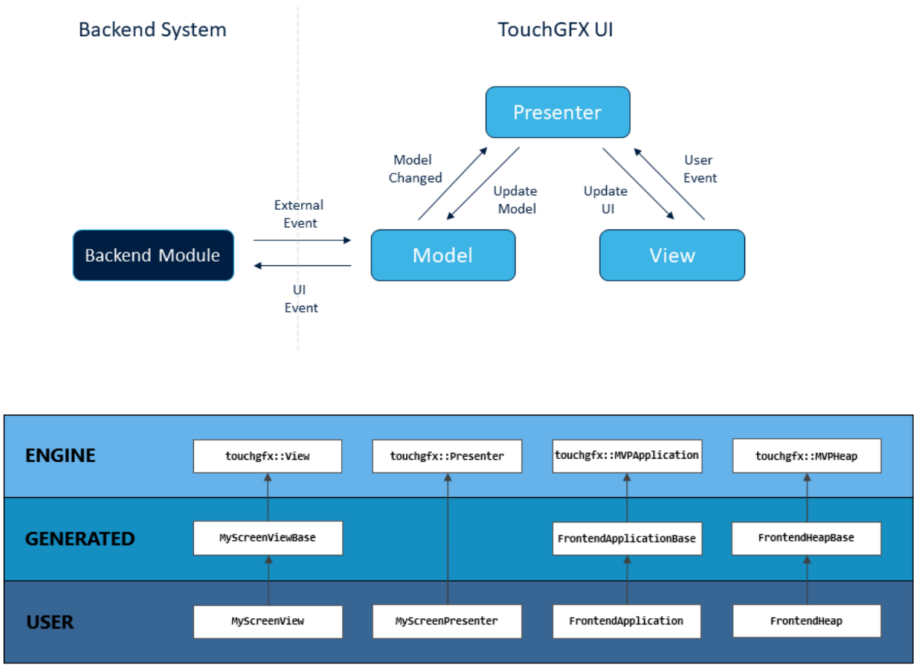

Figure 3: TouchGFX MVP Architecture and Class Hierarchy

- **View:** Responsible for the visual representation and layout.
- **Model:** Contains the core game state, rules, and mathematical logic.
- **Presenter:** Acts as the mediator, handling events from the View and updating the Model, or reflecting Model changes back to the View.

3.2 The View: User Interface

The View layer is responsible for the visual presentation of the game and handling user interactions. Built on the TouchGFX framework, it strictly follows the Model-View-Presenter (MVP) pattern, ensuring that the graphical interface is decoupled from the core game logic.

3.2.1 Main Menu

The Main Menu (‘MainView‘) serves as the entry point for the application. It is designed to be visually engaging while providing clear navigation options.

Visual Elements:

- **Background:** The screen uses a deep "Oxford Blue" color (#0A1128) to set a retro atmosphere.
- **Dynamic Animation:** To make the menu alive, semi-transparent Tetromino blocks fall in the background at varying speeds. This is implemented in the ‘handleTickEvent‘ function, which updates the vertical position of each block every frame.

- **Interactive Components:** The menu features "New Game" and "High Scores" buttons. A High Score Modal pops up to display the top records when requested.

Code Implementation: The following code snippet demonstrates the initialization of the animated background blocks in ‘MainViewView::setupScreen‘:

```
// Source: TouchGFX/gui/src/mainview_screen/MainViewView.cpp

void MainViewView::setupScreen()
{
  MainViewViewBase::setupScreen();
  SoundEngine_PlayTrack(TRACK_MENU);

  // 1. Background (Oxford Blue #0A1128)
  background.setPosition(0, 0, 240, 320);
  background.setColor(touchgfx::Color::getColorFromRGB(0x0A, 0x11, 0x28));
  add(background);

  // ... (Logo and Buttons setup omitted) ...

  // 2. Decoration: Random floating blocks
  touchgfx::BitmapId blocks[] = {
    BITMAP_BLOCK_I_ID, BITMAP_BLOCK_J_ID, BITMAP_BLOCK_L_ID,
    BITMAP_BLOCK_O_ID, BITMAP_BLOCK_S_ID, BITMAP_BLOCK_T_ID, BITMAP_BLOCK_Z_ID
  };

  for (int i = 0; i < 10; i++)
  {
    backgroundBlocks[i].setBitmap(touchgfx::Bitmap(blocks[i % 7]));
    int x = getRandom(228);
    int y = getRandom(400) - 100; // Random start position
    backgroundBlocks[i].setXY(x, y);
    backgroundBlocks[i].setAlpha(40); // Subtle transparency
    backgroundBlockSpeeds[i] = 1 + getRandom(3); // Random speed
    insert(&background, backgroundBlocks[i]);
  }
}

```

Listing 1: Main Menu Background Animation Setup

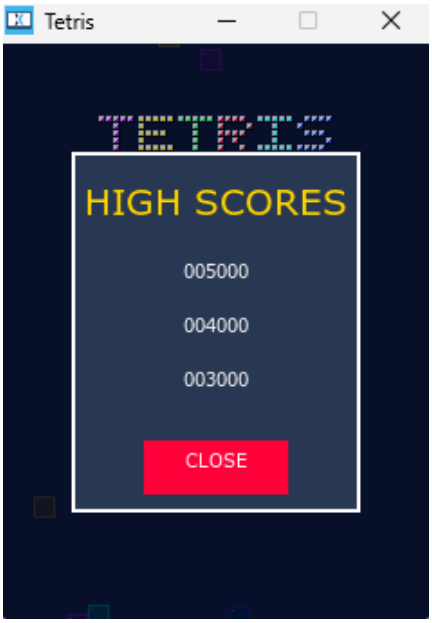

Figure 4: Screenshot of Main Menu with High Score Medal

3.2.2 Game Screen

The Game Screen (‘GameView‘) is the core interface where gameplay occurs. It is optimized to render the game state at 60 FPS by selectively updating only the graphical elements that have changed.

Screen Layout: The layout is divided into three main sections to maximize the 240x320 resolution:

- **Central Matrix:** A 120x240 pixel container displaying the game grid. It renders the fixed blocks, the falling active piece, and the "Ghost Piece" (a visual guide showing where the piece will land).
- **Left Sidebar:** Displays the "HOLD" piece, current Level, and Lines cleared.
- **Right Sidebar:** Displays the "NEXT" piece preview, Score, and the Goal for the next level.

Rendering Logic: The ‘updateBoard‘ function acts as the main rendering loop. It synchronizes the visual state with the data from the Presenter. To optimize performance, the system uses a pool of ‘touchgfx::Image‘ widgets (‘fixedBlocks‘, ‘fallingBlocks‘, ‘ghostBlocks‘) rather than creating new objects dynamically.

```
1   // Source: TouchGFX/gui/src/gameview_screen/GameViewView.cpp
2
3   void GameViewView::updateBoard()
4   {
5     // 1. Synchronize Grid (Fixed Blocks)
6     for (int y = 0; y < MATRIX_ROWS; y++)
7     {
8       for (int x = 0; x < MATRIX_COLS; x++)
9       {
10        signed char type = presenter->getGridValue(x, y);
11        if (type >= 0 && type < Tetris::COUNT)
```

```
12          {
13            fixedBlocks[y][x].setBitmap(touchgfx::Bitmap(blockBitmaps[
    type]));
14            fixedBlocks[y][x].setVisible(true);
15          }
16          else
17          {
18            fixedBlocks[y][x].setVisible(false);
19          }
20          fixedBlocks[y][x].invalidate(); // Mark for redraw
21        }
22      }
23
24      // 2. Draw Active Piece and Ghost Piece
25      Tetris::TetrominoType currentType = presenter->
    getCurrentPieceType();
26      if (currentType != Tetris::NONE)
27      {
28        // Draw Ghost Piece
29        int ghostY = presenter->getGhostY();
30        if (ghostY > presenter->getCurrentY())
31        {
32          drawPiece(currentType,
33          presenter->getCurrentX(),
34          ghostY,
35          presenter->getCurrentRotation(),
36          ghostBlocks, 2, 2);
37        }
38
39        // Draw Falling Piece
40        drawPiece(currentType,
41        presenter->getCurrentX(),
42        presenter->getCurrentY(),
43        presenter->getCurrentRotation(),
44        fallingBlocks, 2, 2);
45      }
46     }
47
```

Listing 2: GameView Board Rendering Logic

Figure 5: Screenshot of GameView with Active Gameplay

3.3 The Presenter: UI-Model Sychnorization

The Presenter acts as the "middleman" in the MVP architecture, bridging the gap between the Game Model and the View. It ensures that the Model remains completely unaware of the UI implementation details, while the View remains passive and data-driven.

Responsibilities:

- **Data Formatting:** The Presenter retrieves raw data from the Model (e.g., integer scores) and formats it into structures the View can easily consume (e.g., sorted leaderboards).
- **Event Handling:** It listens for UI events (like button clicks) and forwards actionable commands to the Model.
- **State Synchronization:** Implementing the ‘ModelListener‘ interface, it receives notifications when the game state changes (e.g., board updates, game over) and instructs the View to repaint.

Code Implementation: A key example of the Presenter’s logic is the ‘getScoreboard‘ method in ‘GameViewPresenter‘. It combines the persistent high scores from the Model with the current session’s score and sorts them to display a dynamic ranking on the "Game Over" screen.

```
1    // Source: TouchGFX/gui/src/gameview_screen/GameViewPresenter.cpp
2
3    void GameViewPresenter::getScoreboard(ScoreInfo* buffer)
4    {
5      // 1. Retrieve raw data from Model
6      int highScores[3];
7      model->getHighScores(highScores);
8      int currentScore = model->getScore();
9
```

```
      // 2. Populate buffer
      for(int i=0; i<3; i++) {
        buffer[i].score = highScores[i];
        buffer[i].isCurrent = false;
      }
      buffer[3].score = currentScore;
      buffer[3].isCurrent = true;

      // 3. Sort Descending (Bubble Sort for small dataset)
      for(int i=0; i<3; i++)
      {
        for(int j=0; j<3-i; j++)
        {
          if (buffer[j].score < buffer[j+1].score)
          {
            ScoreInfo temp = buffer[j];
            buffer[j] = buffer[j+1];
            buffer[j+1] = temp;
          }
        }
      }
    }

```

Listing 3: Scoreboard Sorting Logic in Presenter

3.4 The Model: Game Logic

The Model class encapsulates the entire state of the Tetris game, operating independently of the View and Presenter. This ensures that the game rules—including gravity, collisions, and scoring—remain consistent regardless of the visual representation. At its core, the game field is represented by a 2D array, `grid[20][10]`, where each cell stores a value corresponding to the type of Tetromino occupying it, or `-1` if the cell is empty.

3.4.1 The Game Loop (Tick)

The `tick()` function serves as the "heartbeat" of the game. It is triggered periodically by the TouchGFX framework based on the current game speed. This loop manages the progression of the game through several key phases:

- **Gravity:** The system tracks time using a `tickCounter`. When it reaches the `dropSpeed` threshold, the current piece is moved down one row.
- **Collision Detection:** Before any movement or rotation, the `isCollision()` method checks the proposed coordinates against the grid boundaries and existing blocks.
- **Landing and Locking:** If a downward move results in a collision, the piece is "locked" into the static grid, and a new piece is spawned.
- **Line Clearing:** Immediately after locking a piece, the system scans the grid. If a row is full, it is cleared, and blocks above it are shifted down.

```
void Model::tick()
{
    if (isGameOver || isPaused) return;

```

```
    // 1. Gravity Logic
    tickCounter++;
    if (tickCounter >= dropSpeed)
    {
        tickCounter = 0;
        step(); // Attempt to move piece down
    }

    // ... Input Processing (See Section 3.4.2) ...

    // 2. Notify UI of changes
    if (modelListener != 0)
    {
        modelListener->modelStateChanged();
    }
}

```

Listing 4: Game Loop Core Logic (Simplified)

3.4.2 Input Processing

User interaction is handled asynchronously to ensure responsiveness. Inputs generated via GPIO interrupts (buttons) or the Touch interface are converted into commands and sent to the Model through a FreeRTOS message queue (`inputQueueHandle`).

During each execution of the `tick()` function, the Model drains this queue and executes the corresponding operations:

- **Queue Integration:** The code uses `osMessageQueueGet` to retrieve pending commands without blocking the GUI task.
- **Command Flow:**
  - `'U'`: Rotate the piece.
  - `'L' / 'R'`: Move the piece left or right.
  - `'D'`: Soft Drop (one step down).
  - `'H'`: Hard Drop (snap to bottom).
  - `'S'`: Hold the current piece.

```
// Processing Input Queue from FreeRTOS
#ifndef SIMULATOR
uint8_t key = 0;
while (osMessageQueueGet(inputQueueHandle, &key, NULL, 0) == osOK)
{
    switch (key)
    {
        case 'U': rotate(); break;
        case 'R': moveRight(); break;
        case 'D': step(); break;
        case 'L': moveLeft(); break;
        case 'H': hardDrop(); break;
        case 'S': holdPiece(); break;
    }
}
```

```
#endif

```

Listing 5: Input Processing Logic in Model

3.4.3 Collision Detection & Mechanics

To prevent pieces from moving through walls or other blocks, the ‘isCollision‘ method checks the target coordinates of every block in the active Tetromino against the grid boundaries and existing occupied cells.

When a piece lands (collision detected on ‘step()‘), the ‘lockPiece()‘ method transfers the active piece into the static ‘grid‘ array and immediately triggers ‘checkLines()‘.

3.4.4 Line Clearing Algorithm

The ‘checkLines()‘ method scans the grid from bottom to top. If a row is fully occupied (no ‘-1‘ values), it is cleared, and all rows above it are shifted down. This method also calculates the score based on the number of lines cleared simultaneously (100, 300, 500, or 800 points).

```
// Source: TouchGFX/gui/src/model/Model.cpp

void Model::checkLines()
{
  int clearedInThisStep = 0;

  for (int y = 19; y >= 0; y--)
  {
    bool isFull = true;
    for (int x = 0; x < 10; x++)
    {
      if (grid[y][x] == -1) // -1 means empty
      {
        isFull = false;
        break;
      }
    }

    if (isFull)
    {
      clearedInThisStep++;
      // Shift everything above down
      for (int shiftY = y; shiftY > 0; shiftY--)
      {
        for (int x = 0; x < 10; x++)
        {
          grid[shiftY][x] = grid[shiftY - 1][x];
        }
      }
      // Clear top line
      for (int x = 0; x < 10; x++) grid[0][x] = -1;

      // Re-check this same Y index since it now contains new data
      y++;
    }
```

```
36        }
37
38        // Update Score and Level
39        if (clearedInThisStep > 0)
40        {
41          int points[] = {0, 100, 300, 500, 800};
42          score += points[clearedInThisStep] * level;
43        }
44      }
45
```

Listing 6: Line Clearing Logic

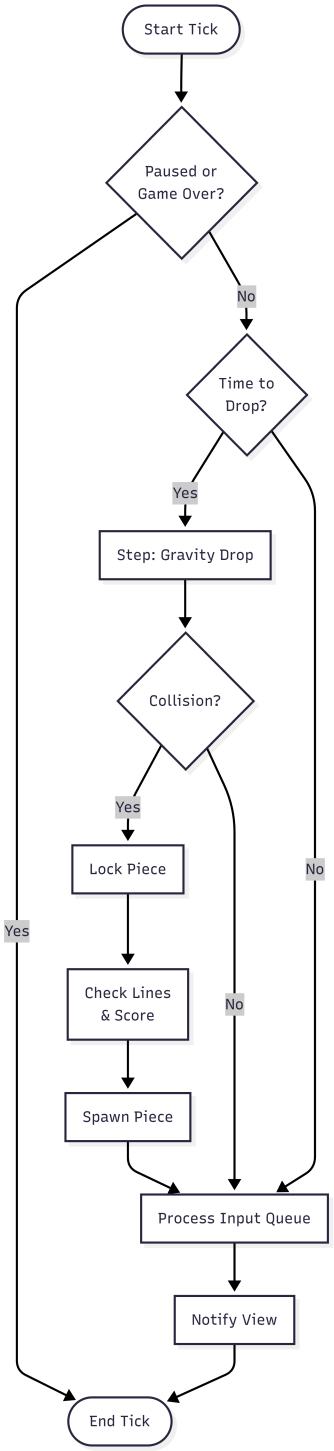

Figure 6: Flowchart of the Game Logic (Tick Execution)

3.5 System Integration (RTOS & Audio)

The application logic rests upon a robust real-time foundation powered by FreeRTOS. This architecture ensures deterministic behavior by managing hardware resources and asynchronous background tasks effectively, allowing the STM32F429 to handle graphics rendering and audio synthesis concurrently without blocking.

3.5.1 FreeRTOS Task Management

To ensure smooth operation and responsiveness, the system is divided into three primary tasks. The scheduling policy is preemptive, ensuring that time-critical operations (like UI updates) take precedence over background activities.

The task allocation is detailed in Table 2:

Task Name	Priority	Stack Size	Functionality
`GUI_Task`	Normal	8192 words	Executes TouchGFX engine and MVP logic.
`DefaultTask`	Normal	128 words	Handles physical button polling (debouncing).
`SoundTask`	Low	256 words	Processes background music and SFX.

Table 2: FreeRTOS Task Configuration

- **GUI_Task:** Being the heaviest consumer of CPU cycles, this task drives the display updates. It integrates with the TouchGFX framework to render the Tetris grid and animations.
- **SoundTask:** Configured with *Low Priority* to prevent audio processing from stuttering the graphical interface. It utilizes 'osDelay' to manage note duration, yielding CPU time back to the scheduler during silence or sustained notes.

3.5.2 Audio Engine Implementation

The audio subsystem is implemented using the STM32's hardware timer (TIM10) in Pulse Width Modulation (PWM) mode. The system generates sound by manipulating the frequency of the square wave output on pin **PF6**.

Frequency Calculation: The Timer is clocked at 1MHz ($1\mu s$ per tick). To generate a specific musical note frequency (f_{note}), the Auto-Reload Register (ARR) is dynamically updated using the formula:

$$\text{ARR} = \frac{f_{timer}}{f_{note}} - 1 = \frac{1,000,000}{f_{note}} - 1 \tag{1}$$

To maintain a consistent volume, the Pulse width (CCR) is always set to 50% of the ARR ($CCR = ARR/2$).

3.5.3 Source Code Implementation

The implementation consists of a low-level driver for frequency generation and a high-level RTOS task for sequencing the melody (e.g., Mario Theme).

1. Low-Level PWM Driver: This function directly manipulates the hardware timer registers to change the pitch.

```
1    /* Sets the PWM frequency for the passive buzzer */
2    void Buzzer_SetFrequency(uint32_t freq)
3    {
4      if (freq == 0) {
5        // Stop PWM generation for silence/rests
```

```
        __HAL_TIM_SET_COMPARE(&htim10, TIM_CHANNEL_1, 0);
        return;
      }

      // Calculate period based on 1MHz Timer Clock
      uint32_t period = (1000000 / freq) - 1;

      // Update Auto-Reload Register (Frequency)
      __HAL_TIM_SET_AUTORELOAD(&htim10, period);

      // Update Capture Compare Register (50% Duty Cycle)
      __HAL_TIM_SET_COMPARE(&htim10, TIM_CHANNEL_1, period / 2);
    }

```

Listing 7: Buzzer Frequency Control Function

2. RTOS Music Task: The `SoundTask` iterates through predefined arrays of notes and durations. It uses ‘osDelay‘ to maintain precise timing without blocking the CPU.

```
    void SoundEngineTask(void *argument)
    {
      // Ensure PWM is started on Channel 1
      HAL_TIM_PWM_Start(&htim10, TIM_CHANNEL_1);

      uint32_t i = 0;
      // Calculate total notes in the melody
      const uint32_t len = sizeof(mario_notes)/sizeof(mario_notes[0]);

      for (;;) // Infinite Loop
      {
        if (mario_notes[i] == 0) {
          Buzzer_Stop(); // Handle Rests
        } else {
          Buzzer_SetFrequency(mario_notes[i]); // Play Note
        }

        // Non-blocking delay for note duration
        osDelay(mario_durations[i]);

        // Loop the melody
        i++;
        if (i >= len) i = 0;
      }
    }

```

Listing 8: RTOS Task for Background Music

4 System Integration & Challenges

4.1 Clock Tree Configuration for RNG

A significant challenge during hardware integration was the 48 MHz clock requirement for the RNG. The STM32F429 reaches its maximum performance at a SysClk of 180

MHz, but the Main PLL cannot generate both 180 MHz and 48 MHz simultaneously due to integer divider limitations.

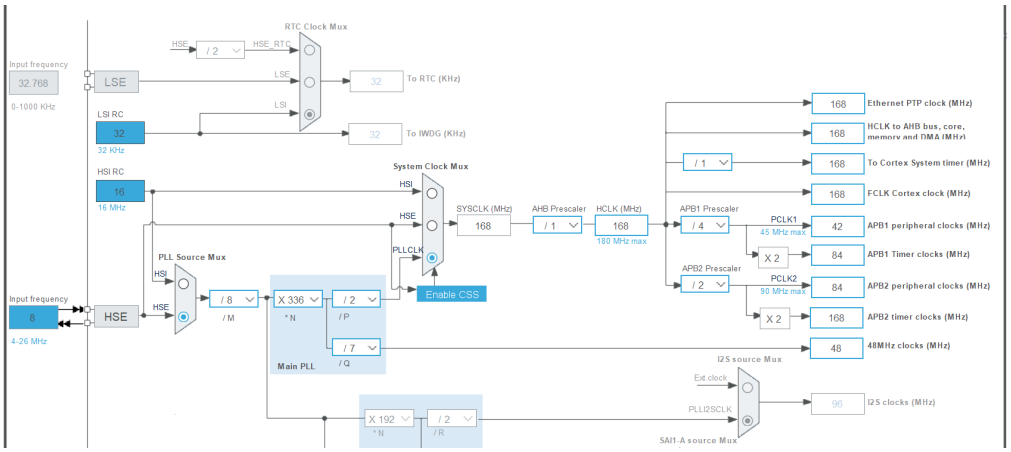

Figure 7: Final Clock Configuration (168 MHz SysClk, 48 MHz PLL48CLK)

- **The Issue:** At 180 MHz (VCO = 360 MHz), the PLLQ divider would need to be 7.5 to reach 48 MHz, which is not supported. Deviations from 48 MHz lead to peripheral initialization failure and "Clock Error" flags.
- **The Solution:** We adjusted the system clock to 168 MHz (PLLM=8, PLLN=336, PLLP=2, PLLQ=7).
- **VCO Calculation:** $(8\ \text{MHz}/8) \times 336 = 336\ \text{MHz}$.
- **Resulting Clocks:** SysClk $= 336\ \text{MHz}/2 = 168\ \text{MHz}$; RNG Clock $= 336\ \text{MHz}/7 = 48\ \text{MHz}$ (Exact).

4.2 Input Debouncing

Directly reading GPIO pins caused erratic behavior due to mechanical switch bounce. We implemented software debouncing using FreeRTOS Timers.

- When a button press interrupt occurs, a timer starts (e.g., 50ms).
- The input is only registered if the signal remains stable until the timer callback executes.

5 Results and Conclusion

5.1 Project Outcomes

The project successfully delivers a playable Tetris clone on the STM32F429I-DISCO.

- Performance: The game runs at a stable 60 FPS.
- Audio: Background music plays smoothly without interrupting game logic.
- Gameplay: All standard Tetris rules (rotation, wall kicks, line clears, leveling) are implemented.

5.2 Future Improvements

- Implement "Ghost Piece" visualization (indicated in design but currently simplified).
- Save high scores to internal Flash memory to persist after power loss.
- Add support for using the on-board Gyroscope (SPI5) for experimental tilt controls.