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 26, 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).

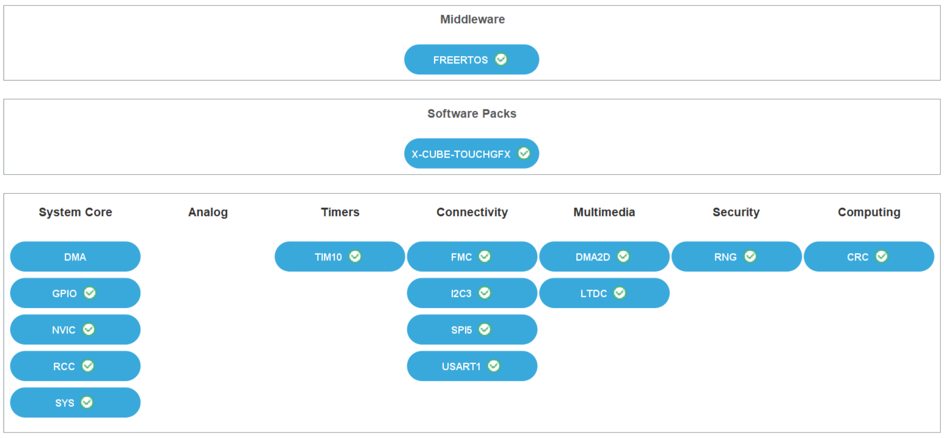


Figure 1: STM32CubeMX System and Peripheral Initialization Overview

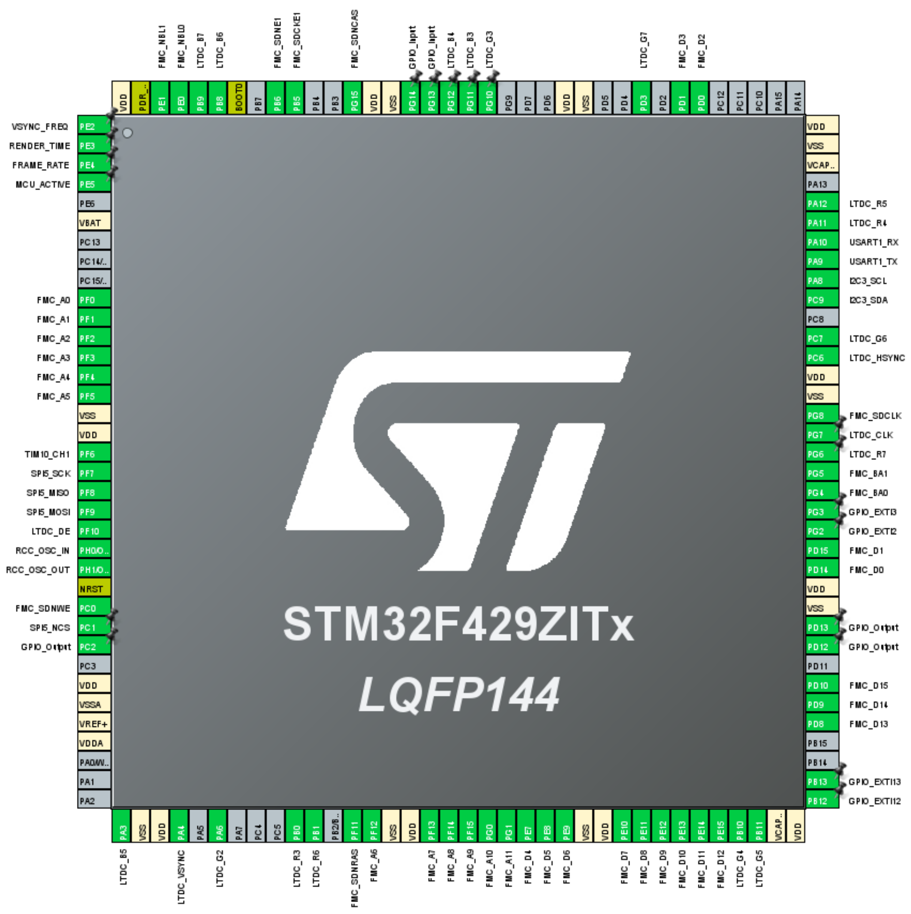


Figure 2: STM32F429ZIT6 Pinout and GPIO Configuration

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.

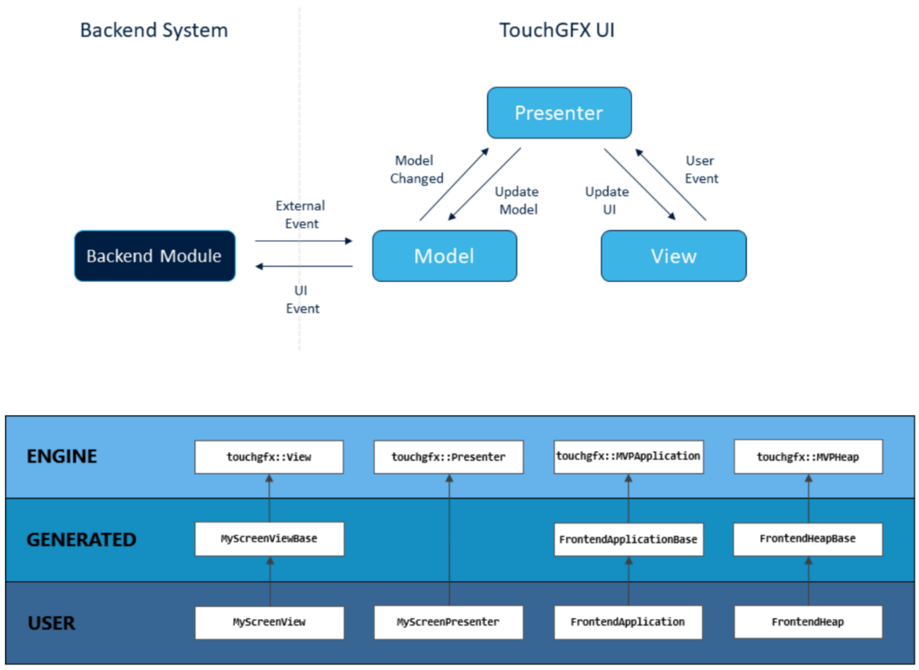


Figure 3: TouchGFX MVP Architecture and Class Hierarchy

- **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:

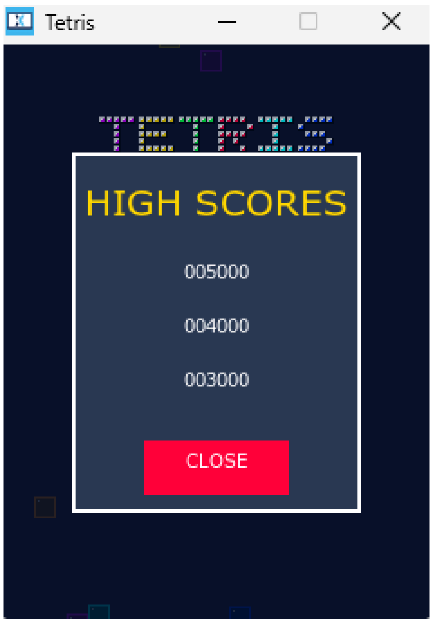
- **Background:** The screen uses a deep "Oxford Blue" color (#0A1128) to set a retro atmosphere.
- **Dynamic Animation:** To make the menu alive, semi-transparent Tetromino blocks fall in the background at varying speeds. This is implemented in the ‘handleTickEvent‘ function, which updates the vertical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



HIGH SCORES

005000

004000

003000

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.

```
  // Source: TouchGFX/gui/src/gameview_screen/GameViewView.cpp

  void GameViewView::updateBoard()
  {
    // 1. Synchronize Grid (Fixed Blocks)
    for (int y = 0; y < MATRIX_ROWS; y++)
    {
      for (int x = 0; x < MATRIX_COLS; x++)
      {
        signed char type = presenter->getGridValue(x, y);
        if (type >= 0 && type < Tetris::COUNT)
      {
        fixedBlocks[y][x].setBitmap(touchgfx::Bitmap(blockBitmaps[type]));
        fixedBlocks[y][x].setVisible(true);
      }
      else
      {
        fixedBlocks[y][x].setVisible(false);
      }
      fixedBlocks[y][x].invalidate(); // Mark for redraw
    }
  }

  // 2. Draw Active Piece and Ghost Piece
  Tetris::TetrominoType currentType = presenter->getCurrentPieceType();
  if (currentType != Tetris::NONE)
  {
    // Draw Ghost Piece
    int ghostY = presenter->getGhostY();
    if (ghostY > presenter->getCurrentY())
    {
      drawPiece(currentType,
      presenter->getCurrentX(),
      ghostY,
      presenter->getCurrentRotation(),
      ghostBlocks, 2, 2);
    }

    // Draw Falling Piece
    drawPiece(currentType,
    presenter->getCurrentX(),
    presenter->getCurrentY(),
    presenter->getCurrentRotation(),
    fallingBlocks, 2, 2);
  }
}

```

Listing 2: GameView Board Rendering Logic

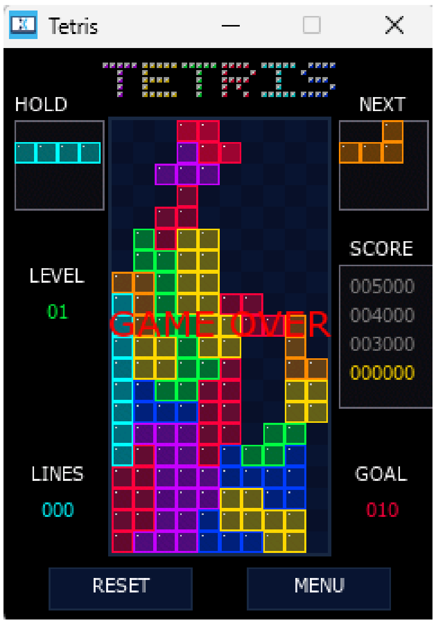


Figure 5: Screenshot of GameView with Active Gameplay

3.3 The Presenter: Logic-UI Synchronization

The Presenter layer serves as the bridge between the View and the Model. It listens for notifications from the Model (via ‘ModelListener‘) and triggers updates in the View, ensuring the visual layer stays perfectly synchronized with the logical state. This project utilizes two distinct presenters corresponding to the two main screens.

3.3.1 MainViewPresenter

The ‘MainViewPresenter‘ manages the logic for the Main Menu screen. Its responsibilities are streamlined to focus on navigation and data retrieval for the menu interface:

- **Navigation:** Handles the transition event when the user selects "Start Game", requesting the application to switch to the Game Screen.
- **Data Retrieval:** Fetches stored high scores from the Model (‘getHighScores‘) to display the leaderboard on the menu, enticing the player to beat the top records.

3.3.2 GameViewPresenter

The ‘GameViewPresenter‘ is the core mediator for the active gameplay session. It handles a high frequency of events and data flow:

- **Input Proxying:** It receives user input events from the View (e.g., button presses for left, right, rotate) and forwards them to the Model via methods like ‘handleLeft()‘ or ‘handleRotate()‘.
- **State Synchronization:** It implements the ‘modelStateChanged()‘ method. When the game tick updates the Model (gravity, collisions), this method is called to trigger ‘view.updateBoard()‘, ensuring the screen redraws at 60 FPS.

- **Scoreboard Logic:** It implements complex logic for the Game Over screen. The ‘getScoreboard()‘ method retrieves the top 3 high scores, mixes in the current player’s score, and sorts them dynamically. This allows the player to see their rank immediately upon finishing a game (‘ScoreInfo‘ structure).

```
  // Example: Synchronization in GameViewPresenter.cpp
  void GameViewPresenter::modelStateChanged()
  {
    view.updateBoard(); // Signal the View to redraw based on new
Model data
  }

```

3.4 The Model: Game Logic

The Model layer maintains the internal state of the Tetris grid and implements the rules of the game. It is independent of the GUI framework, allowing for easier testing and logic updates.

3.4.1 Grid Representation

The 10x20 game board is represented as a 2D array: ‘signed char grid[20][10]‘, where each cell stores a value corresponding to a Tetromino type or remains -1 if empty.

3.4.2 Game Loop (‘tick‘)

The ‘tick()‘ function is the entry point for game updates, called every frame (60Hz). It handles gravity, collision detection via ‘isCollision()‘, and line clearing logic.

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

    tickCounter++;
    if (tickCounter >= dropSpeed)
    {
      tickCounter = 0;
      step(); // Gravity drop
    }
    // ... Input queue processing ...
  }

```

3.5 System Integration (RTOS & Audio)

The application logic sits atop a real-time foundation that manages hardware resources and asynchronous background tasks.

3.5.1 FreeRTOS Task Management

Three primary tasks ensure smooth operation:

- **GUI_Task:** Executes the TouchGFX engine and MVP logic (High Priority).

- **DefaultTask:** Handles button debouncing and system monitoring.
- **SoundTask:** Processes audio requests in the background (Low Priority).

3.5.2 Audio Engine

To maintain performance, audio is handled via a Producer-Consumer pattern. The Game Model (Producer) sends ‘TrackID‘ requests to a queue, which are processed by the ‘SoundTask‘ (Consumer) to drive the PWM-based buzzer.

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.

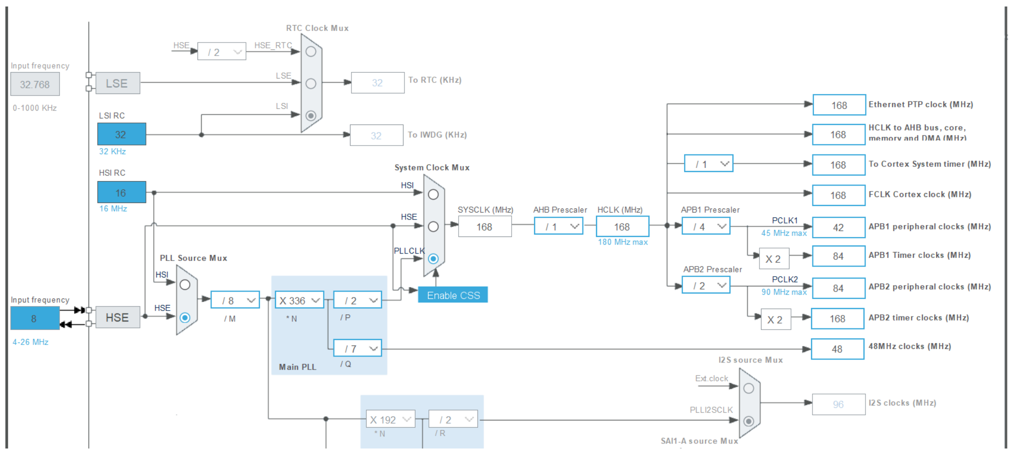


Figure 6: Final Clock Configuration (168 MHz SysClk, 48 MHz PLL48CLK)

- **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:** $(8\text{ MHz}/8) \times 336 = 336$ MHz.
- **Resulting Clocks:** SysClk = 336 MHz/2 = 168 MHz; RNG Clock = 336 MHz/7 = 48 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.