Project Report: Design and Development of the 2048 Game on STM32F429 Embedded System

Student Name

January 20, 2026

Abstract

This document presents the design, development, and validation process of a portable game console based on the STM32F429 microcontroller. The project integrates the management of analog peripherals, the use of a real-time operating system (FreeRTOS), and the deployment of an advanced graphical interface via TouchGFX.

Contents

1 Introduction

This project focuses on the complete realization of a "2048" video game embedded system on the **STM32F429I-DISCO** platform. The challenge consisted of interfacing algorithmic game logic (C++), a fluid graphical interface (TouchGFX), and analog input peripherals (Joystick), all managed by a real-time system (FreeRTOS).

The educational objective was twofold: to master the STMicroelectronics development chain (CubeMX, CubeIDE, TouchGFX Designer) and to understand the constraints related to embedded systems, particularly memory management and input/output responsiveness.

This report traces the development timeline, the chosen hardware and software architecture, as well as the solutions applied to the technical problems encountered.

2 Hardware Architecture

The choice of the STM32F429I-DISCO development board is significant. It possesses the necessary resources to support a resource-intensive graphical application.

2.1 The STM32F429ZIT6 Microcontroller

The core of the system is an ARM Cortex-M4 clocked at 180 MHz. This processor has a Floating Point Unit (FPU), although our game logic relies mainly on integers. Its major assets for this project are its dedicated graphics peripherals:

- **LTDC (LCD-TFT Display Controller):** Allows sending image data directly to the screen without permanently loading the CPU.
- **DMA2D (Chrom-ART Accelerator):** This hardware peripheral accelerates graphical operations (memory copy, pixel format conversion, color blending). It is essential for TouchGFX to maintain a fluid framerate (60 FPS).

2.2 Memory Management and Display

The display is provided by a 2.4" QVGA TFT LCD screen.

- **The need for SDRAM:** A 240x320 pixel image with a 16-bit color depth (RGB565) occupies about 153 KB ($240 \times 320 \times 2$ bytes). To avoid flickering, we use a "Double Buffering" technique (two images in memory), which requires over 300 KB. As internal RAM is limited, the use of the external 64 Mbit SDRAM connected via the FMC interface is indispensable.

2.3 User Interface (Inputs)

Interaction is provided by a 2-axis analog joystick. Unlike digital buttons (on/off), the joystick provides a variable voltage proportional to the position.

- **Connection:** X-Axis on pin **PC1**, Y-Axis on **PC4**.
- **Conversion:** These signals are processed by the microcontroller's ADC1 (Analog-to-Digital Converter).

3 Software Architecture

The software architecture is based on a clear separation between the hardware, the real-time kernel, and the graphical application.

3.1 The MVP Model (Model-View-Presenter)

The project follows the **Model-View-Presenter** pattern imposed by the TouchGFX framework. This architecture allows decoupling the business logic from the display.

Figure 1: MVP communication diagram in TouchGFX

1. **Model (C++):** This is the brain of the application. It contains the 4x4 game matrix (‘board[4][4]‘), the current score, and the game state (In Progress, Won, Lost). It does not "know" how this data is displayed.
2. **View (C++):** Manages the display. The `GameScreenView` class handles placing tiles on the screen and triggering animations. It makes no decisions regarding game rules.
3. **Presenter (C++):** Acts as an intermediary. When the Model changes (e.g., a tile moves), it notifies the Presenter, which then asks the View to update.

3.2 Integration with FreeRTOS

The system uses a real-time OS to manage concurrency.

- **GUI Task:** Automatically managed by TouchGFX, it has a low priority but runs frequently to refresh the screen.
- **Input Task (StartDefaultTask):** A dedicated task polls the ADC values. When a movement is detected, it sends a message via an `osMessageQueue` to the graphical interface.

4 Game Engine Implementation

The logic of the 2048 game relies on matrix manipulations. This part was coded entirely in C++ in `Model.cpp`.

4.1 Merge and Compress Algorithm

The critical function is `compressAndMerge()`. It must move tiles in a given direction and merge identical ones.

We implemented this logic generically to handle all 4 directions (Up, Down, Left, Right) using coordinate transformations, thus avoiding duplicating the code four times.

Algorithm 1 Simplified Logic for Move Left

for each row i from 0 to 3 **do**
    Step 1: Compression (move zeros to the end)
    Move all non-zero elements to the left
    Step 2: Fusion
    for each column j from 0 to 2 **do**
        if $board[i][j] == board[i][j+1]$ AND $board[i][j] \neq 0$ **then**
            $board[i][j] \leftarrow board[i][j] \times 2$
            $board[i][j+1] \leftarrow 0$
            Increase score
        end if
    end for
    Step 3: New Compression to fill the gaps created
end for

4.2 Random Generation

To respect the official rules, the appearance of new tiles follows a probabilistic distribution:

- 90% chance to appear as a "2".
- 10% chance to appear as a "4".

This function uses `rand()` and searches for a random empty cell in the matrix.

5 Joystick Signal Processing

Integrating the joystick required software processing to convert an unstable raw analog value into a reliable game command.

5.1 ADC Reading and Deadzone

The STM32 ADC returns a 12-bit value (0 to 4095). At rest, the joystick, subject to return springs, never returns exactly to the theoretical center value (2048). It oscillates mechanically around this position.

To prevent the game from detecting ghost movements, we defined a software **Deadzone**:

- **Low Threshold:** 1000
- **High Threshold:** 3000

Any value between 1000 and 3000 is considered "Neutral".

5.2 Hysteresis and State Machine

An initial problem was sensitivity: a single thumb push sometimes triggered two very rapid movements. To correct this, we implemented a hysteresis mechanism via state flags (`neutral_X`, `neutral_Y`).

The principle is as follows: 1. A command (e.g., RIGHT) is sent only if the ADC value exceeds the threshold (e.g., > 3000) **AND** the `neutral` flag is true. 2. Once the command is sent, the `neutral` flag is set to `false`. 3. The system forbids any new command until the joystick has physically returned to the deadzone (between 1000 and 3000), which resets the flag to `true`.

Figure 2: Graphical representation of the deadzone and hysteresis

6 Technical Challenges and Solutions

6.1 The Linker Error (Unity Build)

A major problem was encountered when integrating the HAL ADC drivers. The error *"Undefined Reference to HAL_ADC_Init"* blocked compilation. The STM32CubeIDE (Eclipse-based) failed to correctly link the object files of the manually added drivers.

The solution adopted was the "Unity Build". Instead of letting the linker search for separately compiled files, we included the source ‘.c‘ files directly in ‘main.c‘ via preprocessor directives:

```
/* USER CODE BEGIN Includes */
#include "stm32f4xx_hal_adc.c"
#include "stm32f4xx_hal_adc_ex.c"
/* USER CODE END Includes */
```

Listing 1: Unity Build Implementation

This forces the compiler to treat these drivers as part of the same translation unit as the main program, instantly resolving dependencies.

6.2 Graphics Optimization

Early in development, tile rendering suffered from slight slowdowns during rapid movements. We optimized this by using partial screen invalidation (redrawing only the tiles that changed) rather than redrawing the entire screen every frame.

7 Testing and Validation

A validation phase was necessary to ensure the robustness of the game.

7.1 Test Protocols

- **Saturation Test:** Intentionally filling the grid without making merges to verify the triggering of the "Game Over" screen.
- **Victory Test:** Simulating a matrix close to victory (e.g., two 1024 tiles side by side) to validate the appearance of the "You Win" popup.
- **Joystick Stress Test:** Performing rapid circular movements to verify that the hysteresis system correctly filters parasitic inputs.

7.2 Results

The system proved to be stable. The use of SDRAM allows for constant fluidity without flickering. Memory consumption remains within the limits defined by the STM32F429 linker script.

8 Conclusion

This project allowed us to put theoretical embedded systems concepts into practice: interrupt management, ADC, inter-task RTOS communication, and LCD screen control. We moved from simple input/output management to a complex and reactive graphical application. The final product is a functional game, faithful to the original, demonstrating the STM32F429's capability to simultaneously manage game logic and graphical refreshing.