



# **EMBEDDED SYSTEM REPORT**

## **GAME SPACE INVADER**

**Course:** Embedded System - IT4210E

**Instructors:** Ph.D. Ngo Lam Trung

**Students:** Pham Duc Cuong - 20235904

Ngo Minh Ngoc - 20235984

Le Phuong Linh - 20235964

Ha Noi, January 2026

## **Contents**

<b>Introduction.....</b>	<b>3</b>
<b>Chapter 1: Project Overview.....</b>	<b>4</b>
1.1 Motivation of Topic Selection .....	4
1.2 Project Objectives.....	4
1.3 Implementation Idea .....	5
1.4 Research Methodology .....	5
<b>Chapter 2: Hardware System.....</b>	<b>6</b>
2.1 System architecture .....	6
2.2 Overview of the STM32F429I-DISC1 Discovery Kit .....	7
<b>Chapter 3: Program development and deployment.....</b>	<b>8</b>
3.1 System Configuration .....	8
3.2 User Interface Design using TouchGFX .....	11
3.3 Software Architecture (Model–View–Presenter) .....	13
3.4 System Interaction Flow .....	15
3.5 Source Code Construction Process .....	17
<b>Chapter 4: Demo .....</b>	<b>30</b>

## Introduction

Information technology has been rapidly developing and plays an essential role in modern life. It has significantly transformed various aspects of society, including communication, entertainment, education, and work. Today, we live in a world where billions of devices are connected to the Internet, forming a complex and extensive information network. However, beyond familiar large-scale devices such as personal computers and smartphones, information

technology is also deeply embedded in compact and integrated systems known as **embedded systems**.

Embedded systems are systems that integrate microprocessors, sensors, and communication modules to perform dedicated and specific functions. Typical applications of embedded systems include mobile devices, smart medical equipment, Internet of Things (IoT) home appliances, and many other applications in both daily life and industrial environments.

The continuous advancement of technology has strongly driven the development of embedded systems. Microprocessors are becoming increasingly powerful, compact, and energy-efficient; memory and storage technologies are rapidly evolving; and open-source software platforms and embedded development tools are widely adopted. These technological advancements have created favorable conditions for developing increasingly complex and diverse embedded applications.

Motivated by the attractiveness of this field and the desire to gain hands-on experience in solving practical embedded system problems, our group selected the topic **“Development of the Space Invaders Game Using the STM32F429I-DISC1 Kit”** as our course project.

This project consists of four main parts:

1. Project overview
2. Hardware system
3. Program development and deployment
4. Overall evaluation

Although our group has made considerable efforts to complete the project within the given timeframe and knowledge scope, certain limitations and shortcomings during implementation and testing are unavoidable. We sincerely appreciate any comments and suggestions from the instructor to further improve the project.

Finally, we would like to express our sincere gratitude to **PhD Ngô Lam Trung** for his valuable guidance and support throughout the completion of this course and project.

## **Chapter 1: Project Overview**

### **1.1 Motivation of Topic Selection**

Control devices and embedded systems are becoming increasingly widespread and play an essential role in modern life. In reality, almost every aspect of daily activities is influenced by various types of control systems. It is easy to find control devices in machinery, tools, computer systems... The development of embedded systems presents both significant challenges and valuable learning opportunities, contributing to a deeper understanding of how intelligent electronic devices operate. In this course project, our group decided to choose the

topic “**Development of the Space Invaders Game Using the STM32F429I-DISC1 Kit**” for the following reasons:

- **Space Invaders** is a classic and well-known game. Implementing this game on an embedded system is expected to provide novelty and technical challenges during the research and development process.
- **Practical application:** The STM32F429I-DISC1 is a popular and powerful development board widely used in embedded system applications. Developing the Space Invaders game on this platform helps us gain a deeper understanding of hardware operation and effectively utilize the board’s capabilities.
- **Learning and skill development:** This project requires applying knowledge of microcontrollers, data structures, peripheral communication, and embedded programming. Through both theoretical study and laboratory practice, we have opportunities to improve hardware programming skills and enhance logical and creative thinking.
- **Final product development:** The ultimate goal of this project is to develop a complete and stable Space Invaders game on an embedded system that can be practically experienced. This helps us understand the embedded software development process and prepares us for future real-world projects.

Based on these reasons, within the scope of this course project, our group aims to implement an embedded system that provides valuable learning and practical experience, while contributing to the application of embedded technology in everyday life.

## 1.2 Project Objectives

During the Embedded Systems course, based on the context and requirements mentioned above, our group set the following objectives for the project:

1. Apply fundamental knowledge in designing and developing simple embedded systems.
2. Develop a product that can be considered “intelligent.”
3. **User Interface (UI) development:** Design an intuitive and user-friendly interface that allows users to easily interact with and play the Space Invaders game.
4. **Game rule processing:** Define and implement the core rules of the Space Invaders game.
5. **Peripheral control:** Use the STM32F429I-DISC1 kit to control peripherals such as the display, keypad, and other necessary devices, enabling intuitive and flexible user interaction.
6. **Testing and debugging:** Perform cross-testing to ensure correctness and stability of the game, detect errors, and fix them to guarantee reliable system operation.

## 1.3 Implementation Idea

**Space Invaders** is a classic shoot-’em-up video game in which the player controls a spaceship to defend against waves of enemies descending from the top of the screen. In this project, the player uses a **joystick** to control the movement of the spaceship and to perform firing actions. The joystick-based control mechanism enables intuitive and responsive interaction between the user and the embedded system. The game emphasizes real-time interaction, simple control logic, and clearly defined game rules.

Due to its intuitive gameplay structure and moderate system resource requirements, Space Invaders is well suited for implementation on embedded platforms. The use of a joystick as an input device provides an effective way to evaluate real-time input processing, peripheral interfacing, graphical display control, and basic game state management in embedded systems.

The game is implemented on the STM32F429I-DISC1 development kit using the **TouchGFX** graphics library. The STM32F429I-DISCOVERY board leverages the high-performance STM32F429 microcontroller, enabling users to develop feature-rich applications with advanced graphical user interfaces.

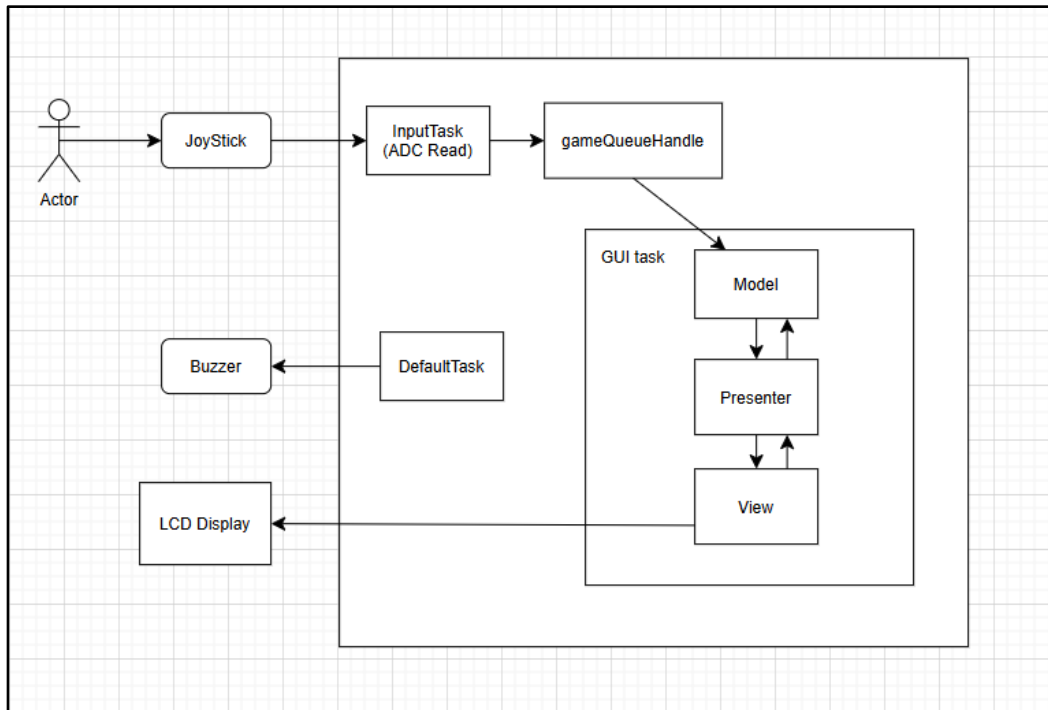
## 1.4 Research Methodology

The research methodology of this project includes the following steps:

1. Studying fundamental knowledge of the development board, system configuration, programming, and embedded implementation.
2. Using **STM32CubeIDE** for coding, debugging, and compilation.
3. Preparing the STM32F429I-DISC1 kit, joystick and required hardware components.
4. Investigating the STM32F429I-DISC1 kit, its components, and peripherals through datasheets and relevant technical documents available on the Internet, YouTube and AI-supported such as ChatGPT, Gemini, ect.

# Chapter 2: Hardware System

## 2.1 System architecture



*Our System Architecture*

- **Actor:** The user directly interacts with the system.
- **Joystick:** An input device that provides analog signals (voltage) changing according to the control direction.
- **LCD Display:** An output device that displays the user interface (GUI).
- **Buzzer:** An audio output device used for notifications or alerts.
- **InputTask (ADC Read):** This is a separate task in the RTOS. Its job is to read values from the Joystick's Analog-to-Digital Converter (ADC). Separating this task allows continuous data sampling without freezing the user interface.
- **gameQueueHandle:** This is a Message Queue. After reading data, InputTask sends the data (e.g., X, Y coordinates or movement commands) into this queue. This is a safe inter-task communication method in RTOS, helping to avoid data conflicts.

The system uses the Model–View–Presenter architecture, which is very common in graphics libraries such as TouchGFX:

- **Model:** Receives data from gameQueueHandle. It holds the game/application logic and updates the latest data state.
- **Presenter:** Acts as an “intermediary.” It takes data from the Model and decides what should be displayed on the View, while also receiving user interactions from the View and forwarding them back to the Model.

- **View:** Responsible for rendering graphics and displaying them directly on the LCD Display.

Buzzer Integration:

- **DefaultTask:** The system's default task. In this diagram, it is responsible for controlling the Buzzer.

## 2.2 Overview of the STM32F429I-DISC1 Discovery Kit

The STM32F429I-DISC1 Discovery Kit is a development and evaluation board designed by STMicroelectronics to support applications based on the STM32F429ZIT6 microcontroller, which integrates an ARM Cortex-M4 core with Floating Point Unit (FPU). The board is intended for learning, prototyping, and developing embedded systems that require high performance, rich peripherals, and graphical capabilities.

The STM32F429ZIT6 MCU operates at a maximum frequency of 180 MHz, providing 2 MB of Flash memory and 256 KB of SRAM, along with DSP instructions and hardware floating-point support. To enhance multimedia and graphical applications, the board is equipped with an external 8 MB SDRAM, a 2.4-inch QVGA color LCD with touch panel, and a dedicated LCD controller interface.

With its rich hardware resources, integrated debugging support, and compatibility with STM32CubeIDE, HAL, and LL libraries, the STM32F429I-DISC1 Discovery Kit provides a comprehensive platform for studying ARM Cortex-M4 architecture and developing real-time embedded applications.

# Chapter 3: Program development and deployment

## 3.1 System Configuration

### a) Configuring GPIO PG2 and PG3

In the MX\_GPIO\_Init() function, configure the PG2 and PG3 pins in Input mode - pull-up

```

/*Configure GPIO pins : PG2 PG3 for input*/
GPIO_InitStruct.Pin = GPIO_PIN_2 | GPIO_PIN_3;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

```

## b) ADC Configuration (GPIO PC3)

1. Enable HAL\_ADC\_MODULE\_ENABLED in /Core/Inc/stm32f4xx\_hal\_conf.h
2. Copy the ADC library files to the correct project folder
3. Import the ADC source files into the project
4. Add the ADC configuration functions to the stm32f4xx\_hal\_msp.c file:

```

106 void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
107 {
108     GPIO_InitTypeDef GPIO_InitStruct = {0};
109     if(hadc->Instance==ADC1)
110     {
111         /* USER CODE BEGIN ADC1_MspInit 0 */
112         /* USER CODE END ADC1_MspInit 0 */
113         /* Peripheral clock enable */
114         __HAL_RCC_ADC1_CLK_ENABLE();
115         __HAL_RCC_GPIOC_CLK_ENABLE();
116         __HAL_RCC_GPIOA_CLK_ENABLE();
117         /**ADC1 GPIO Configuration
118         PC3 -----> ADC1_IN13
119         */
120         GPIO_InitStruct.Pin = GPIO_PIN_3;
121         GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
122         GPIO_InitStruct.Pull = GPIO_NOPULL;
123         HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
124
125         GPIO_InitStruct.Pin = GPIO_PIN_5;
126         HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
127     }
128 }
129
130 void HAL_ADC_MspDeInit(ADC_HandleTypeDef* hadc)
131 {
132     if(hadc->Instance==ADC1)
133     {
134         __HAL_RCC_ADC1_CLK_DISABLE();
135         /**ADC1 GPIO Configuration
136         PC3 -----> ADC1_IN13
137         */
138         HAL_GPIO_DeInit(GPIOC, GPIO_PIN_3);
139     }
140 }

```

5. Add the MX\_ADC1\_Init() function to main.c:

```

681 static void MX_ADC1_Init(void)
682 {
683     /* USER CODE BEGIN ADC1_Init 0 */
684     /* USER CODE END ADC1_Init 0 */
685     ADC_ChannelConfTypeDef sConfig = {0};
686     /* USER CODE BEGIN ADC1_Init 1 */
687     /* USER CODE END ADC1_Init 1 */
688     /** Configure the global features of the ADC (Clock,
689     Resolution, Data Alignment and number of conversion)
690     */
691     hadc1.Instance = ADC1;
692     hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
693     hadc1.Init.Resolution = ADC_RESOLUTION_12B;
694     hadc1.Init.ScanConvMode = DISABLE;
695     hadc1.Init.ContinuousConvMode = DISABLE;
696     hadc1.Init.DiscontinuousConvMode = DISABLE;
697     hadc1.Init.ExternalTrigConvEdge =
698     ADC_EXTERNALTRIGCONVEDGE_NONE;
699     hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
700     hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
701     hadc1.Init.NbrOfConversion = 1;
702     hadc1.Init.DMAContinuousRequests = DISABLE;
703     hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
704     if (HAL_ADC_Init(&hadc1) != HAL_OK)
705     {
706         Error_Handler();
707     }
708 }

```

### c) Audio Hardware Configuration

For audio output, a Pulse Width Modulation (PWM) signal is used to generate sound through a buzzer/speaker.

- Based on the microcontroller datasheet, the GPIO pins occupied by the LCD display and joystick interface were identified and excluded.
- Pin PB4 was selected as the audio output pin since it was not shared with other peripherals.
- The selected pin was configured as a PWM output using a timer peripheral.
- The PWM signal is used to generate audio signals for background music and in-game sound effects.

The audio data used in the game was generated based on publicly available open-source

Reference: DotNet MXL Parsing for Arduino (GitHub repository).

<https://github.com/MrRedBeard/DotNet-MXL-Parsing-for-Arduino/tree/master>

```

void Buzzer_Init(void)
{
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    // A. Cấp xung nhịp (Clock) cho Timer 3 và Port B
    __HAL_RCC_TIM3_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    // B. Cấu hình chân PB4 làm chân PWM (Alternate Function)
    GPIO_InitStruct.Pin = GPIO_PIN_4;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;          // Chế độ thay thế đẩy-kéo
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF2_TIM3;       // Quan trọng: Chọn chức năng TIM3 cho PB4
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    // C. Cấu hình Timer 3
    // Giả sử Clock Timer là 90MHz (chuẩn trên F429).
    // Chia 90 để được bộ đếm 1MHz (1 micro-giây mỗi nhịp đếm)
    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 90 - 1;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = 499; // Ban đầu để 0
    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
    {
        // Lỗi khởi tạo
    }

    // D. Cấu hình kênh PWM (Channel 1)
    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 250; // Độ rộng xung ban đầu = 0 (Tắt)
    sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
    {
        // Lỗi cấu hình kênh
    }

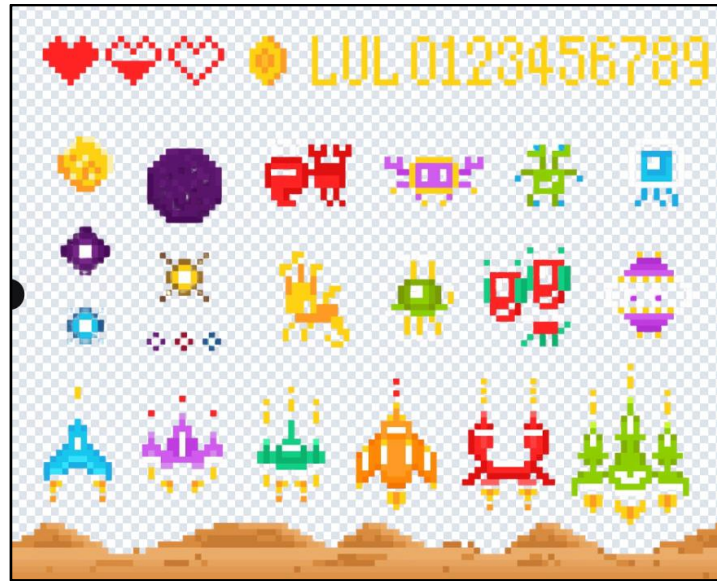
    // E. Bắt đầu Timer (nhưng chưa phát tiếng vì Pulse = 0)
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
}

```

## 3.2 User Interface Design using TouchGFX

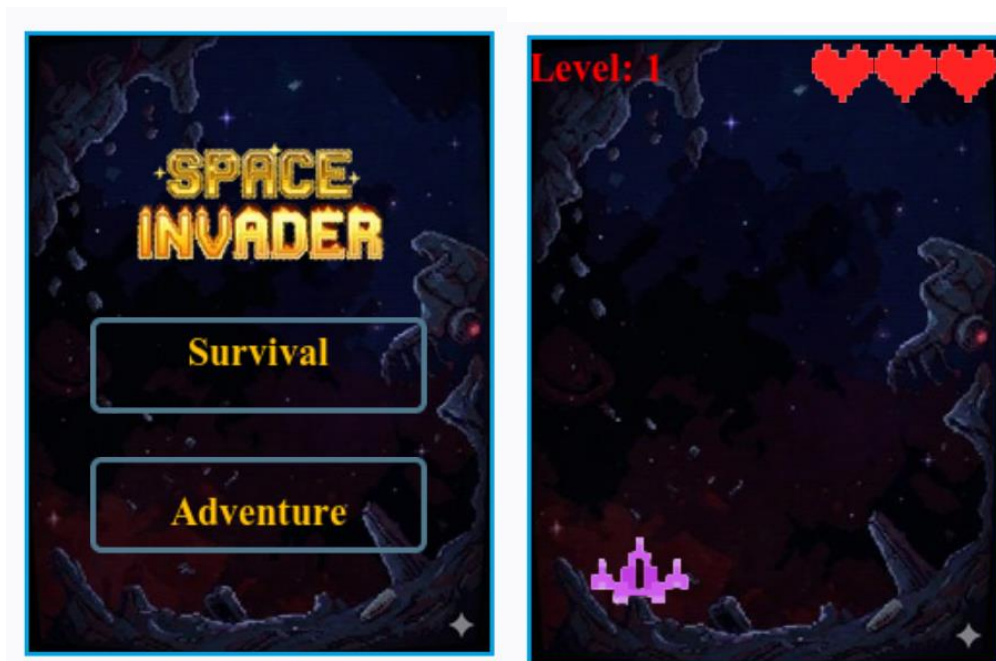
### 3.2.1 Collecting resources for elements

- Before designing the user interface, the images used in the game were prepared to ensure compatibility with the embedded system. Image source: [https://drive.google.com/drive/folders/1MLa1-b-47IN3NC4NBckhJkVg2\\_sGGV1x?usp=sharing](https://drive.google.com/drive/folders/1MLa1-b-47IN3NC4NBckhJkVg2_sGGV1x?usp=sharing)
- Image assets were resized according to the LCD screen resolution to prevent distortion and to reduce memory usage (32x32 pixels), the backgrounds of the images were removed to create transparent sprites, allowing more flexible rendering and layout of the user interface.



### 3.2.2 Game Interface design using TouchGFX

After the image processing was completed, the assets were imported into TouchGFX Designer. Based on these assets, the user interface was designed, including the main screens of the game such as the menu screen and the gameplay screen. Interface components such as images and buttons were arranged appropriately, and user interactions were configured through TouchGFX to support game control and state transitions.





*Our design in TouchGFX and the prototype*

### 3.3 Software Architecture (Model–View–Presenter)

The project is built upon the standard **TouchGFX MVP architecture**. This architecture facilitates a clear separation between the User Interface (UI), control logic, and the underlying hardware system:

- **Model:** Manages global data and communicates with the Operating System/Hardware
- **View:** Renders graphics, captures events from the touch screen, and executes the game logic.
- **Presenter:** Acts as an intermediary, coordinating information between the Model and the View

#### a) Model (Data Layer and Hardware) - Model.cpp, Model.hpp

**Main Role:**

- Acts as the sole bridge between the interface (GUI) and the rest of the embedded system (Backend/Hardware).
- Stores data states that exist throughout the application lifecycle (Global State).

**Specific Functions in Code:**

- **Hardware Communication:** Uses CMSIS-RTOS (osMessageQueueGet) to receive external control signals (e.g., physical buttons, sensors, or other tasks) via gameQueueHandle.
- **Periodic Processing:** The tick() function is called continuously according to the system beat to check the message Queue. When a signal is received (e.g., move ship), it notifies the modelListener (which is the current Presenter).

- Data Management: Stores the highScore variable. Provides saveScore to update the record and getHighScore to retrieve the score.

## **b) View (Interface Layer and Game Logic) - Screen1View.cpp, Screen2View.cpp, Screen3View.cpp**

### **Main Role:**

- Displays graphical elements (Ship, bullets, monsters, score, health hearts...).
- In this game project, the View handles the majority of the game operation logic (Game Physics) to ensure smooth display performance.

### **Specific Functions in Code:**

- Game Loop: The handleTickEvent() function runs every frame. This is where bullet and monster movement, collision detection (checkCollision), health deduction, and monster spawning are handled.
- UI Update: Redraws ship position (updateShipPosition), updates the health bar (updateHealthUI), and displays the score.
- Interaction: When the game ends or screens change, the View calls Presenter functions (e.g., presenter->gameOver(), presenter->saveScore()).

## **c) Presenter (Coordinator Layer) - Screen1Presenter.cpp, Screen2Presenter.cpp, Screen3Presenter.cpp**

### **Main Role:**

- Acts as the "interpreter" between the Model and the View. The View never directly calls the Model and vice versa.
- Decides the screen transition flow.

### **Specific Functions in Code:**

- Receiving Commands from Model: When the Model receives a hardware signal (e.g., moveShip), it notifies the Presenter, and the Presenter calls view.updateShipPosition().
- Receiving Commands from View: When the View reports a game over or needs to save a score, the Presenter calls the Model (model->saveScore) or calls FrontendApplication to switch screens.

## **d) Screen Structure**

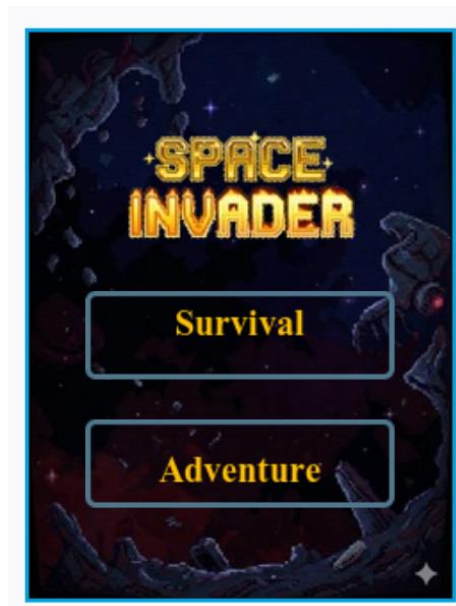
Based on the code files, the project consists of 3 distinct screens, managed by FrontendApplication:

- **Screen 2 (Menu / Transition):**

- Current code is quite empty (Screen2View.cpp, Screen2Presenter.cpp), serving only as a skeleton.
  - Acts as a waiting screen, main menu, or transition screen between play sessions.
- **Screen 1 (Game Play - Level 1):**
    - The main game screen (vertical shooter style).
    - Logic: Purple ship (purpleplane), shooting bullets, aliens falling from the top.
    - Contains logic to switch levels (nextLevel) when enough monsters are defeated.
    - On loss -> Switch to Screen 2 (via the gameOver function).
  - **Screen 3 (Game Play - Level 2 / Advanced Mode):**
    - An advanced game screen or a different game mode.
    - Logic: More complex than Screen 1; monsters appear from both left/right sides (alienSpeedX), monsters can shoot back (enemyBulletPool), and includes a High Score save feature.
    - On loss -> Displays the scoreboard (Score/HighScore) -> Waits for user -> Switches back to Menu (Screen 2).

### 3.4 System Interaction Flow

The user can choose between 2 game-play modes: Survival and Adventure:



- Selecting Button 1 in the TouchGFX interface navigates the user to Screen1, which handles the gameplay logic for the Survival mode via Screen1View.cpp

- Selecting Button 2 navigates to Screen3, where the Adventure mode logic is implemented in Screen3View.cpp

The interaction flow in the project is divided into 3 main scenarios:

#### **a) Control Flow from Hardware (Hardware to UI)**

This is the most critical flow, demonstrating how physical signals control the game.

- **Step 1 (Hardware & RTOS):** In main.c, button presses or peripheral tasks send data to the gameQueueHandle queue via the osMessageQueuePut function. For example: When the user presses the left button, the character 'L' is pushed into the Queue.
- **Step 2 (Model - Backend):** The Model class in TouchGFX runs continuously. The Model::tick() function performs queue checks using osMessageQueueGet.
- **Step 3 (Model to Presenter):** If a message is received (e.g., 'L', 'R', 'U', 'D'), the Model calls the notification function via the Interface: modelListener->moveShip(msg).
- **Step 4 (Presenter to View):** The Presenter of the current screen (e.g., Screen1Presenter) receives the command and forwards it to the View: view.updateShipPosition(dir).
- **Step 5 (View - UI):** The View (such as Screen1 View) recalculates the new coordinates for the purpleplane object, checks screen boundaries, and calls invalidate() to request the hardware to redraw the new frame.

#### **b) Game Logic and Collision Processing Flow (Internal Game Logic)**

This flow takes place entirely within the View layer to ensure frame processing speed.

- **Step 1 (Game Loop):** Every time the screen refreshes (typically at 60Hz), the `handleTickEvent()` function in the View is called.
- **Step 2 (Update & Collision):**
  - Automatically moves bullets and monsters (Aliens).
  - Calculates collisions using the `checkCollision` function.
- **Step 3 (Event Trigger):**
  - **If a bullet hits a monster:** Increment the `enemiesDefeated` variable (Screen 1) or `currentScore` (Screen 3).
  - **If a monster hits the ship:** Decrease `playerHealth` and update the heart UI via `updateHealthUI()`.

### c) End-of-Game and Data Persistence Flow (Game Over & Data Persistence)

Describes how data from the Game is saved back into the system.

- **Step 1 (View):** When `playerHealth <= 0`, the View calls `presenter->gameOver()` or `handleGameOver()`.
- **Step 2 (Presenter to Model):** The Presenter receives the command and requests the Model to save the score: `model->saveScore(currentScore)`.
- **Step 3 (Model):** The Model checks if the current score is higher than the old `highScore`; if so, it updates the memory with the new value.
- **Step 4 (Navigation):** The Presenter uses `FrontendApplication` to transition between screens (e.g., returning to the Menu Screen 2).

## 3.5 Source Code Construction Process

### 3.5.1 main.c file:

The program operates based on an RTOS (Real-Time Operating System) mechanism. Instead of running a simple `while(1)` loop to handle everything, `main()` initializes the hardware, creates "Tasks," and then hands over control to the operating system's Scheduler.

#### a) Main Threads/Tasks

This file defines and initializes three parallel processing threads:

<b>GUI_Task</b>	<ul style="list-style-type: none"> <li>- The most resource-intensive task, responsible for rendering the TouchGFX graphical interface onto the screen.</li> </ul>
-----------------	---

	<ul style="list-style-type: none"> <li>- Execution function: TouchGFX_Task (called from an external library).</li> </ul>
<b>InputTask</b>	<ul style="list-style-type: none"> <li>- <b>Function:</b> Reads control signals (e.g., Joystick).</li> <li>- <b>Logic:</b></li> <li>- Uses the StartInputTask function.</li> <li>- Reads Analog values from ADC1 (Channel 13 for the X-axis and Channel 5 for the Y-axis).</li> <li>- Compares voltage values to determine direction: <ul style="list-style-type: none"> <li>• Left ('L'), Right ('R') based on X-axis values.</li> <li>• Up ('U'), Down ('D') based on Y-axis values.</li> </ul> </li> <li>- If movement is detected, it sends the corresponding character into the message queue gameQueueHandle.</li> </ul>
<b>DefaultTask</b>	<ul style="list-style-type: none"> <li>- Currently empty (an infinite loop doing nothing).</li> </ul>

#### b) Inter-thread Communication Mechanism

<b>gameQueueHandle</b>	<ul style="list-style-type: none"> <li>- A Message Queue.</li> <li>- Transfers data from inputTask to other tasks (typically so TouchGFX can recognize which button the user pressed to update the game screen).</li> </ul>
------------------------	---

#### c) Initialized Hardware Peripherals

The main function calls a series of MX\_...\_Init functions to configure the hardware:

- **FMC & SDRAM**
  - Configures the external RAM (SDRAM).
- **LTDC & DMA2D**
  - **LTDC:** Integrated LCD controller that pushes image data to the screen.
  - **DMA2D:** Hardware graphics accelerator (Chrom-ART) that enables extremely fast image rendering and copying without taxing the CPU.
- **SPI5 & ILI9341:**
  - SPI communication used to send configuration commands to the display driver IC (ILI9341).
- **ADC1 (MX\_ADC1\_Init, Get\_ADC\_Value):**
  - Analog-to-Digital Converter. Used to read potentiometer/joystick values.
- **I2C3:**

- Typically used for the capacitive touch controller, though only basic configuration is present in this file.

### 3.5.2 Model.cpp file:

Role: Acts as a bridge between the hardware (via RTOS Queue) and the user interface, communicate with the real-time operating system (RTOS), and store the application's state

#### a) Connection to the Operating System (RTOS)

<b>gameQueueHandle</b>	<p>An external variable representing a Message Queue in CMSIS-RTOS. It is used to receive data from other tasks or from hardware (such as buttons or sensors).</p> <pre>extern osMessageQueueId_t gameQueueHandle;</pre>
------------------------	--

#### b) tick() Function (Heart of Model)

<b>tick()</b>	<ul style="list-style-type: none"> <li>- Called continuously by the framework (typically every time the screen frame refreshes).</li> <li>- Checks if there are any messages (msg) in the <code>gameQueueHandle</code> queue.</li> <li>- If a message is successfully retrieved (<code>osOK</code>), it forwards that command to the interface via <code>modelListener-&gt;moveShip(msg)</code>. This indicates that the system is controlling a ship based on the signals received.</li> </ul>
---------------	---

#### c) Score Management (Business Logic)

These functions handle the simple business logic of the application:

<b>saveScore(int newScore)</b>	Checks if the new score is higher than the current record ( <code>highScore</code> ) and updates it if necessary.
<b>getHighScore()</b>	Returns the highest saved score.

#### d) Interaction

Component	Role
Hardware/Other Task	Send data into the gameQueueHandle.
Model::tick()	Retrieves data from the queue.
ModelListener	Sends the processed signal from the Model to the Presenter.
Presenter	Updates the interface (View) so the user sees the ship move.

#### 3.5.3 Screen1Presenter.cpp file:

This class implements the coordination logic for Screen1 by forwarding ship movement requests to the View and managing the transition to Screen2 when the game ends.

##### 1. Action Control (Movement Logic)

<b>moveShip(uint8_t dir)</b>	<ul style="list-style-type: none"><li>- This is a bridge function.</li><li>- When the user interacts with the UI (e.g., presses an arrow button), the View calls this function in the Presenter.</li><li>- The Presenter then calls back <code>view.updateShipPosition(dir)</code> to request the UI to redraw the ship's position.</li></ul>
------------------------------	---

##### 2. Screen Navigation (Scene Transition)

<b>gameOver()</b>	<ul style="list-style-type: none"><li>- This function handles the end of the game.</li></ul>
-------------------	--

	<ul style="list-style-type: none"> <li>- It uses <code>FrontendApplication</code> to switch from Screen 1 to Screen 2 without a transition effect (<code>NoTransition</code>).</li> <li>- This is the standard approach in TouchGFX for changing screens via C++ source code.</li> </ul>
--	--

### 3.5.3 Screen1View.hpp file:

The `Screen1View.hpp` file serves as the class defining the control variables, methods and state management for an action game level

#### a) Object Management (Object Pooling)

Instead of constantly creating and deleting objects (which consumes resources), this file utilizes a "Pool" mechanism for management:

<b>bulletPool[MAX_BULLETS]</b>	An array storing bullets (maximum 20).
<b>alienPool[MAX_ALIENS]</b>	An array storing enemies/aliens (maximum 30).
<b>heartIcons[3]</b>	An array containing heart icons to display the player's health.

#### b) Game State Variables

This class stores all current data for the game level:

<b>Health (playerHealth)</b>	Manages the remaining health points.
<b>Level (currentLevel, enemiesToNextLevel)</b>	Tracks the current level and the number of enemies required to be defeated to level up.

<b>Speed (currentAlienSpeed, spawnSpeedRange)</b>	Adjusts game difficulty via movement speed and enemy spawn rates.
<b>Timers (tickCounter, alienSpawnCounter, invulnerabilityCounter)</b>	Coordinates frame-based events (ticks), such as the invulnerability period after taking damage or timing for new enemy spawns.

#### c) Core logic methods - declaration - further explanation in file Screen1View.cpp

<b>handleTickEvent()</b>	The "heart" of the game, called continuously every frame to update logic (movement, collision detection, and counter updates).
<b>checkCollision()</b>	A function to check for collisions between two objects based on coordinates (x, y) and dimensions (w, h).
<b>updateShipPosition(uint8_t dir)</b>	Changes the player's ship position based on control direction.
<b>nextLevel() &amp; updateHealthUI()</b>	Updates the display state when the player levels up or loses health.
<b>getRandomNumber(...)</b>	A custom random number generator (used to determine enemy spawn positions).

#### d) UI Components

<b>levelTextBuffer</b>	A buffer used to display the level number on the screen (utilizing Unicode format for TouchGFX).
<b>currentAlienBitmapId</b>	Stores the image ID of the current enemy (allowing appearance changes across different levels).

### 3.5.4 Screen1View.cpp file:

#### a) Functional Overview

The Screen1View class is responsible for:

- Initializing the game (health, level, enemies).
- Handling the game loop (bullet movement, enemy movement, collision detection).

- Managing UI display (health, level, win/lose messages).
- Receiving input to control the player's aircraft.

## b) Main Components

A. **setupScreen:** Initializes the game's initial state (seed, health, level), sets up object pooling for bullets and enemies, and updates the UI to display the level and player health.

## B. Game Loop (**handleTickEvent**)

This is the core function that processes real-time game logic:

<b>Counter management</b>	<ul style="list-style-type: none"> <li>- Increments <code>tickCounter</code> and <code>randomSeed</code>, and decreases <code>levelUpTimer</code> and <code>gameWinTimer</code> to coordinate display states.</li> </ul>
<b>Automatic shooting</b>	<ul style="list-style-type: none"> <li>- Uses the condition <code>tickCounter % 10 == 0</code> to create a firing rhythm.</li> <li>- Iterates through the <code>bulletPool[i]</code> array to find an element with <code>!isVisible()</code> for reuse.</li> <li>- Uses <code>bulletPool[i].setXY()</code> to place the bullet at the nose of the aircraft and <code>setY(getY() - bulletSpeed)</code> to make the bullet move upward.</li> </ul>
<b>Alien spawning</b>	<ul style="list-style-type: none"> <li>- Increases <code>alienSpawnCounter</code> and compares it with <code>nextSpawnTime</code> to determine when a new alien appears.</li> <li>- Uses <code>getRandomNumber(0, maxX)</code> to determine a random horizontal spawn position.</li> <li>- Activates an alien by setting <code>alienPool[i].setVisible(true)</code>.</li> </ul>
<b>Temporary invulnerability handling</b>	<ul style="list-style-type: none"> <li>- Checks whether <code>invulnerabilityCounter &gt; 0</code>.</li> <li>- Uses <code>purpleplane.setVisible(!purpleplane.isVisible())</code> to create a blinking effect when the player is damaged.</li> </ul>

<b>Collision logic</b>	<ul style="list-style-type: none"> <li>- <b>Bullet hits alien:</b> Calls the <code>checkCollision()</code> function inside a nested loop between <code>MAX_ALIENS</code> and <code>MAX_BULLETS</code>. If a hit occurs, increments <code>enemiesDefeated</code> and checks the condition to call <code>nextLevel()</code>.</li> <li>- <b>Alien hits player:</b> Checks for a collision between <code>purpleplane</code> and <code>alienPool[i]</code>. If it occurs, decreases <code>playerHealth</code>, calls <code>updateHealthUI()</code>, and sets <code>invulnerabilityCounter = 100</code>.</li> <li>- <b>Alien goes off-screen:</b> Checks if <code>alienPool[i].getY() &gt; 320</code>. If true, the player's health is reduced in the same way as when being hit.</li> </ul>
------------------------	--

### C. Level System (`nextLevel`)

This function adjusts environmental parameters when the player reaches the required score:

<b>Level up</b>	<ul style="list-style-type: none"> <li>- Increments <code>currentLevel</code> and resets <code>enemiesDefeated = 0</code>.</li> </ul>
<b>Difficulty increase</b>	<ul style="list-style-type: none"> <li>- Updates <code>currentAlienBitmapId</code> (<code>BITMAP_ALIEN2_ID</code>, <code>ALIEN3_ID</code>, etc.).</li> <li>- Increases <code>currentAlienSpeed</code>.</li> <li>- Decreases <code>spawnSpeedRange</code> so aliens appear more frequently.</li> </ul>
<b>Victory</b>	<ul style="list-style-type: none"> <li>- When <code>currentLevel &gt; 4</code>, activates <code>txtGameWin.setVisible(true)</code> and starts the <code>gameWinTimer</code> countdown to end the level.</li> </ul>

#### D. Ship Movement (**updateShipPosition**)

This function handles controller input:

- Uses an `if (dir == 'L'/'R'/'U'/'D')` structure to determine direction.
- Updates coordinates using `purpleplane.setX()` and `purpleplane.setY()` with a movement step of `step = 5`.
- Enforces boundary limits using comparison conditions such as `purpleplane.getX() > 0` or `purpleplane.getX() < (240 - width)` to prevent the ship from leaving the screen.

#### E. Utilities (**checkCollision** & **updateHealthUI**)

<b>checkCollision</b>	Accepts 8 parameters (x, y, w, h of two objects) and returns a <code>bool</code> value based on the AABB (Axis-Aligned Bounding Box) collision algorithm.
<b>updateHealthUI</b>	Iterates through the <code>heartIcons[i]</code> array and uses <code>setVisible(true/false)</code> based on a comparison between the loop index and <code>playerHealth</code> to display the correct number of hearts on the screen.

#### 3.5.5 Screen3Presenter.cpp file:

The Presenter class serves as the intermediary, coordinating the game logic (ship movement, score processing) and screen navigation for Screen3

##### a) Data Processing (Communication with the Model)

The Presenter retrieves data from the Model to provide to the View, or vice versa:

<b>saveScore(int score)</b>	Receives the score and requests the Model to save it.
<b>getHighScore()</b>	Requests the highest score from the Model to display it on the screen.

##### b) UI Control (Communication with the View)

<b>moveShip(uint8_t dir)</b>	This is a navigation function. When an event occurs (e.g., a button press or a hardware signal), the Presenter calls <b>view.updateShipPosition(dir)</b> to execute the ship's movement on the screen.
------------------------------	--

### c) Screen Transitions

<b>gotoMenu()</b>	<ul style="list-style-type: none"> <li>- This function handles scene switching.</li> <li>- It accesses the <b>FrontendApplication</b>.</li> <li>- It uses the command <b>gotoScreen2ScreenNoTransition()</b> to immediately switch back to Screen2 (typically the Menu screen) without any transition effects.</li> </ul>
-------------------	---

### d) Interactions

Component	Role
<b>Model</b>	Stores scores ( <b>saveScore</b> , <b>getHighScore</b> ).
<b>View</b>	Displays the ship graphics ( <b>updateShipPosition</b> ).
<b>Presenter</b>	Receives user commands, requests the Model to process data, and instructs the View to update the graphics.

## 3.5.6 Screen3View.hpp file:

### a) Configuration Constants (Constants)

The file defines maximum limits for in-game entities to manage static memory:

<b>MAX_ALIENS_3 = 30</b>	Maximum number of aliens.
<b>MAX_ENEMY_BULLETS = 20</b>	Maximum number of enemy bullets on screen.

<b>MAX_BULLETS_3 = 30</b>	Maximum number of player bullets.
---------------------------	-----------------------------------

### b) Control Methods (Methods)

These functions handle the game lifecycle and core logic:

<b>Screen lifecycle</b>	<code>setupScreen()</code> (initialization), <code>tearDownScreen()</code> (cleanup), and most importantly <code>handleTickEvent()</code> (updates game logic each frame).
<b>Player logic</b>	<code>takeDamage()</code> (receive damage), <code>updateShipPosition()</code> (move the ship), and <code>updateHealthUI()</code> (update health display).
<b>Game state</b>	<code>handleGameOver()</code> (handles losing condition).
<b>Utilities</b>	<code>checkCollision()</code> (checks collisions between objects) and <code>getRandomNumber()</code> (generates random numbers).

### c) Game State Variables (Game State Variables)

Store the current operating parameters:

<b>currentScore, playerHealth</b>	The player's score and health.
<b>isGameOver</b>	Flag indicating the game-over state.

<b>tickCounter, spawnTimer</b>	Timing counters to control enemy spawning or events.
<b>invulnerabilityCounter</b>	Timer for invulnerability (usually after being hit).

#### d) UI Object & Pool Management (UI Objects & Pools)

Uses the “Object Pooling” technique to efficiently manage on-screen graphics:

<b>Enemies</b>	<b>alienPool</b> along with a velocity array ( <b>alienSpeedX</b> ) and reload timers for shooting ( <b>alienReloadTimer</b> ).
<b>Bullets</b>	Two separate arrays for enemy bullets ( <b>enemyBulletPool</b> ) and player bullets ( <b>bulletPool</b> ).
<b>UI</b>	<b>heartIcons</b> (array of heart icons displaying health) and buffers ( <b>scoreBuffer</b> , <b>yourScoreBuffer</b> ) for displaying scores in Unicode format.

### 3.5.7 Screen3View.cpp file:

#### a) Initialization and Setup — **setupScreen()** Method

This is the starting point when the screen is loaded

<b>Basic Parameters</b>	Resets state variables such as <b>currentScore</b> , <b>playerHealth</b> , <b>isGameOver</b> , and <b>tickCounter</b> ...
<b>Alien Initialization</b>	Uses a loop to set images (from the <b>ALIEN_BITMAPS</b> array) for objects in the <b>alienPool</b> .

<b>Bullet Initialization</b>	Sets images and initial hidden states for <code>bulletPool</code> (player bullets) and <code>enemyBulletPool</code> (enemy bullets).
<b>Initial UI</b>	Hides end-game notification labels ( <code>txtGameLose</code> , <code>txtHighScore</code> ) and prepares the <code>heartIcons</code> array to manage health display.

#### b) Ship Movement — `updateShipPosition(uint8_t dir)` Method

This function receives control commands (usually from buttons or touch input) to move the purpleplane ship

<b>Movement Logic</b>	Checks the value of <code>dir</code> ('L', 'R', 'U', 'D') to modify X or Y coordinates.
<b>Boundary Limits</b>	Uses <code>SCREEN_WIDTH</code> and <code>SCREEN_HEIGHT</code> constants to ensure the ship does not fly outside the display area.

#### c) Game Loop — `handleTickEvent()` Method

This is the most critical function, called continuously by the system to update the game state

<b>Game Over Check</b>	If <code>isGameOver</code> is true, the function increments <code>gameOverTickCount</code> and automatically returns to the menu after 180 ticks.
<b>Invulnerability Effect</b>	Processes the <code>invulnerabilityCounter</code> . When this value is > 0, the ship flashes by toggling the <code>setVisible</code> attribute.

<b>Auto-Firing</b>	Every 15 ticks ( <code>tickCounter % 15 == 0</code> ), a new bullet is retrieved from the <code>bulletPool</code> and placed at the ship's position.
<b>Spawning</b>	When <code>spawnTimer &gt; 60</code> , the function calculates a random position and activates aliens from the <code>alienPool</code> .

#### d) Collision and Combat Logic — Located within `handleTickEvent()`

This logic is nested within the update loop to check interactions between objects:

<b>Movement and Alien Counter-attack</b>	Updates the X-position of aliens and uses <code>alienReloadTimer</code> to decide when an alien fires a bullet from the <code>enemyBulletPool</code> .
<b>Bullet - Alien Collision</b>	Uses the <code>checkCollision</code> function to check if player bullets hit monsters. If a hit occurs, <code>currentScore</code> increases and <code>txtScore</code> is updated.
<b>Enemy - Player Collision</b>	Checks for both aliens crashing into the ship or red bullets hitting the ship. If triggered, the <code>takeDamage()</code> function is called.

#### e) Health Handling and Termination — `takeDamage()` & `handleGameOver()`

<b>takeDamage()</b>	Subtracts from <code>playerHealth</code> , calls <code>updateHealthUI()</code> to update heart images, and sets <code>invulnerabilityCounter = 60</code> for temporary player protection
<b>handleGameOver()</b>	Marks <code>isGameOver = true</code> , saves the high score via <code>presenter-&gt;saveScore()</code> , and displays the end-game UI components

#### f) Helper Methods

<b>getRandomNumber(int min, int max)</b>	A random number generation algorithm based on <code>randomSeed</code> to create unpredictability in the game.
<b>checkCollision(int x1, int y1, ...)</b>	Applies the <b>AABB (Axis-Aligned Bounding Box)</b> algorithm to check if two rectangles overlap.
<b>updateHealthUI()</b>	Iterates through the <code>heartIcons</code> array and toggles their visibility based on the current <code>playerHealth</code> count.

## **Chapter 4: Demo**

Following a successful build process, we have effectively deployed the functional demo on hardware. Below is the video demonstrating the project's operation:

<https://drive.google.com/file/d/1EFGtjL8a2mDxzmVIGfWEI-x5tTHuiXoP/view?usp=sharing>

