ELEC0152 Software Development Report

ANONYMOUS*, UCL Electronic and Electrical Engineering Faculty, UK

This report presents the development of "Fruit Catcher," a real-time embedded game implemented on the NUCLEO-F401RE board using the STM32F401RE microcontroller. The system integrates an SSD1306 OLED display and an analog joystick within a FreeRTOS architecture using the STM32 HAL. The software design employs a thread-safe, two-task model synchronized via mutexes: a high-priority task manages sensor acquisition using ADC with Direct Memory Access (DMA), while a normal-priority task handles game physics and I2C rendering. This project demonstrates the application of RTOS principles to ensure data integrity and timing determinism, validated through Segger SystemView analysis.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; **Real-time operating systems**; • **Software and its engineering** → *Embedded software.*

Additional Key Words and Phrases: RTOS, STM32, I2C, HAL, Mutex, DMA, Segger SystemView, SSD1306

1 Introduction

The objective of this project was to develop a real-time embedded game, "Fruit Catcher," on the STM32 NUCLEO-F401RE development board. The software architecture is implemented in C, utilizing the STM32 Hardware Abstraction Layer (HAL)[5] for low-level peripheral control and FreeRTOS for task scheduling and resource management.

Using a Real-Time Operating System (RTOS) offers significant advantages over a standard "superloop" architecture for this application. In a superloop, the game logic, input polling, and screen rendering would run sequentially. Since the I2C screen update is a blocking operation that can take approximately 30ms, a superloop architecture would suffer from input lag, as button presses occurring during the render phase would be missed. By leveraging FreeRTOS, this project decouples the high-frequency input sampling from the lower-frequency rendering loop. A high-priority task manages inputs to ensure responsiveness, while a lower-priority task handles the game physics and display, synchronized via mutexes to prevent data tearing.

1.1 Game Concept

"Fruit Catcher" is a skill-based arcade game designed for a monochrome 128x64 pixel environment. The core entities in the game are:

- **The Basket:** Controlled by the player, constrained to the bottom axis of the screen.
- **Fruits:** Falling objects represented as filled 6x6 pixel squares. Collecting these increments the score.
- **Bombs:** Falling hazards represented as "X" shapes. Collision with a bomb results in immediate game termination.

The game logic relies on randomized spawning algorithms to ensure replayability, with objects spawning at random X-coordinates and falling at varying speeds.

Author's Contact Information: Anonymous, UCL Electronic and Electrical Engineering Faculty, UK.

2018. Manuscript submitted to ACM

1.2 Game Play

The gameplay experience focuses on responsiveness and state management. The user interacts with the system using an analog joystick module.

- **Controls:** The player moves the joystick along the X-axis to move the basket (the RECTANGLE in the bottom) left or right to catch fruits (the SQUARE shape) to gain scores. The input is handled via analog-to-digital conversion to allow for smooth movement logic. The integrated push-button serves as a state toggle.
- **Game States:** The system implements a Finite State Machine (FSM):
  1. **Playing:** The active state where physics updates and collision checks occur (Fig. 2).
  2. **Paused:** Triggered by pressing the joystick button. The game loop freezes, and a "PAUSED" overlay is rendered (Fig. 3).
  3. **Game Over:** Triggered by colliding with a bomb (in CROSS "X" shape). The screen displays the final score. Pressing the button in this state resets the score and respawns the entities to restart the game (Fig. 4).

Fig. 1. Game Initiation

Fig. 2. Active Gameplay

Fig. 3. Game Pause

Fig. 4. Game Over

2 Methodology

This section outlines the hardware integration and software architecture developed for the "Fruit Catcher" game. The system is designed to prioritize responsiveness and determinism by leveraging specific hardware peripherals on the STM32 microcontroller.

2.1 Hardware Setup

The physical system is built around the NUCLEO-F401RE development board, which houses the STM32F401RE microcontroller (ARM Cortex-M4)[6]. Two external modules are interfaced with the MCU: a 0.96-inch SSD1306 OLED display for visual output and a generic dual-axis analog joystick for user input.

2.1.1 Display Interface. The SSD1306 OLED display (128x64 resolution) communicates via the I2C protocol. The STM32's I2C1 peripheral is configured in standard mode (100 kHz) to ensure stable data transmission. The display requires two data lines (SDA and SCL) along with power connections.

2.1.2 Input Interface. The joystick module provides three distinct signals: two analog voltages representing the X and Y axes, and one digital signal for the integrated push-button.

- **Analog Axes:** The VRX (Horizontal) and VRY (Vertical) pins are connected to the MCU's Analog-to-Digital Converter (ADC1). For this specific game implementation, primarily the X-axis is utilized for lateral basket movement.
- **Digital Button:** The switch (SW) pin is connected to a GPIO pin configured with an internal pull-up resistor. This ensures the pin reads a logical High (1) when released and Low (0) when pressed, eliminating the need for external resistors.

Table 1. Hardware Connections

(a) OLED Connections

Pin	STM32 Pin	Function
SCL	PB8	I2C1_SCL
SDA	PB9	I2C1_SDA
VCC	5V	Power
GND	GND	Ground

(b) Joystick Connections

Pin	STM32 Pin	Function
VRX	PA0	ADC1_IN0
VRY	PA1	ADC1_IN1
SW	PA9 (D8)	GPIO Input
VCC	3.3V	Power
GND	GND	Ground

2.2 Software Development

The system architecture is built upon the FreeRTOS kernel, which manages the scheduling of concurrent tasks and creates a layer of abstraction between the application logic and the hardware[2]. Fig. 5 illustrates the data flow and synchronization mechanisms implemented in the design.

2.2.1 Architectural Diagram. The software architecture leverages FreeRTOS to implement a responsive producer-consumer model, effectively decoupling high-frequency sensor acquisition from the game rendering loop. As illustrated in Fig. 5, the system coordinates concurrent tasks through three specific synchronization mechanisms to maintain data integrity:

- **Mutex:** A Mutex guards the global game state, ensuring atomic access to shared joystick coordinates and preventing data tearing between asynchronous tasks.
- **Semaphore (`ButtonSem`):** A Binary Semaphore provides low-latency event signaling, allowing the `InputTask` to instantly notify the game loop of state changes.

- **Message Queue (LedQueue):** A Message Queue buffers collision events for the LedTask, decoupling blocking hardware operations from the main physics engine.

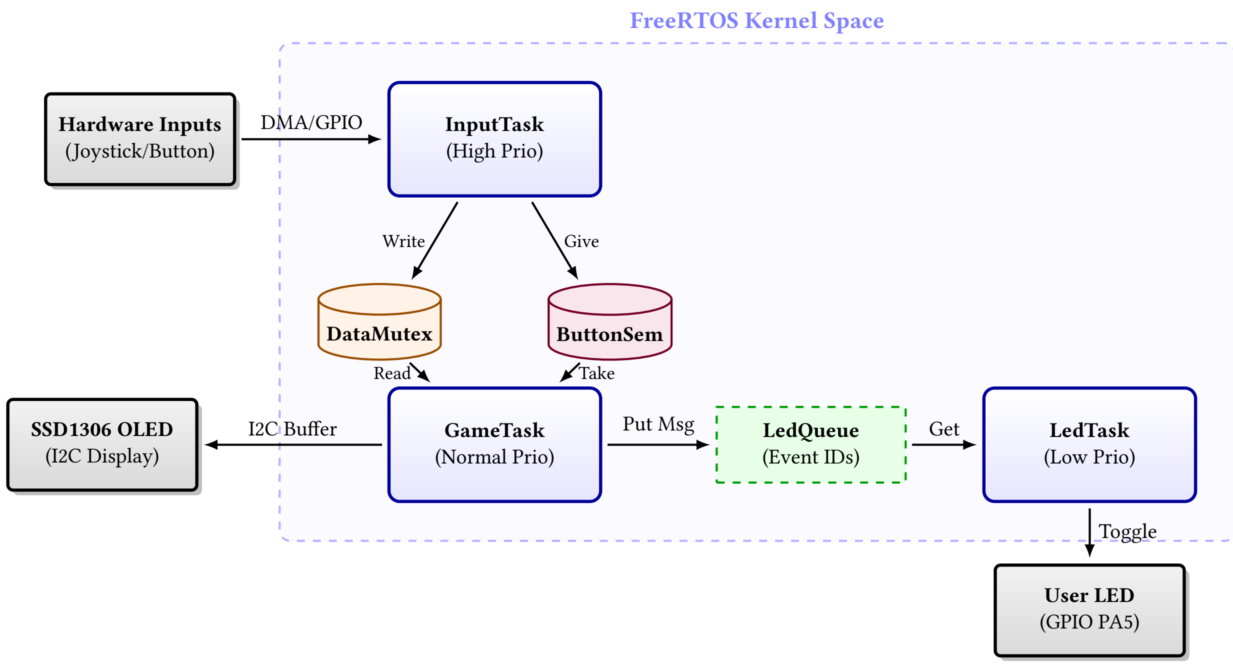


Fig. 5. Complete System Architecture. The InputTask acquires sensor data and signals the GameTask via Semaphore. The GameTask manages physics and offloads feedback events to the LedTask via Queue to prevent blocking.

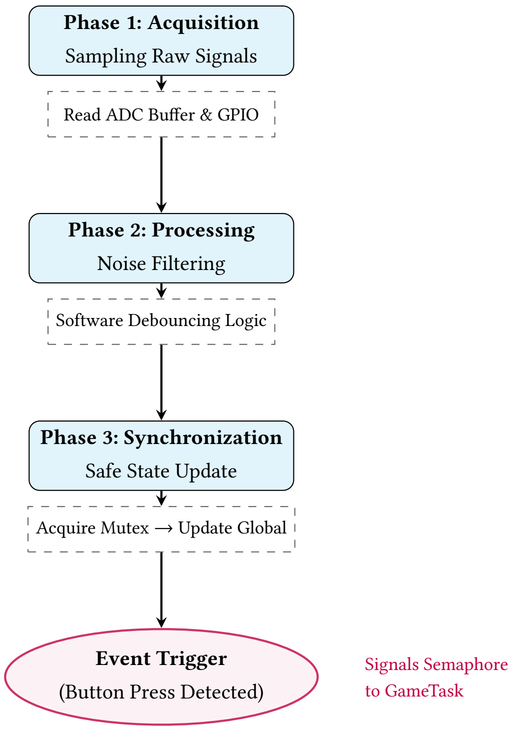


Fig. 6. Conceptual Logic of the InputTask. It functions as a high-speed signal processor that filters noisy hardware inputs before committing them to the shared system state.

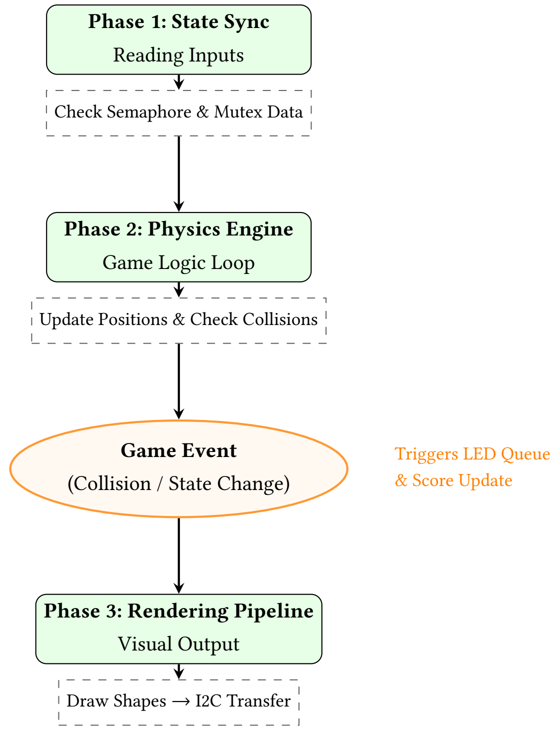


Fig. 7. Conceptual Logic of the GameTask. It operates as a cyclic engine that synchronizes game state, calculates physics, processes game events, and manages the display pipeline.

2.2.2 *Joystick Input Implementation.*

The joystick input subsystem is designed to provide low-latency, non-blocking control by offloading data acquisition to the microcontroller's hardware peripherals. The implementation manages two distinct signal types: analog position data (X/Y axes) and digital state data (push-button). The overall logic flow corresponds to the *Acquisition* and *Processing* phases illustrated in Fig. 6.

ADC and DMA Configuration. To handle the analog signals from the joystick's potentiometers without burdening the CPU, the STM32's Analog-to-Digital Converter (ADC1) is utilized in conjunction with Direct Memory Access (DMA). The peripheral was configured via the `.ioc` interface with the following parameters:

- **Scan Conversion Mode:** Enabled, allowing the ADC to automatically switch between Channel 0 (PA0/X-Axis) and Channel 1 (PA1/Y-Axis) within a single conversion sequence.
- **Direct Memory Access (DMA):** The DMA controller is configured in **Circular Mode** with a data width of *Half Word* (16-bit)[7].

This configuration establishes a continuous background data pipeline. As shown in the code snippet below, the DMA engine is activated once during system initialization in `main.c`. From that point forward, the hardware automatically writes the latest conversion results into the `adcValues` array, ensuring the `InputTask` always has access to the most recent data without needing to trigger conversions manually.

```
// Core/Src/main.c
```

```c
// Start ADC1 in DMA mode to fill the buffer automatically
// adcValues[0] = X-Axis, adcValues[1] = Y-Axis
if (HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcValues, 2) != HAL_OK) {
    Error_Handler();
}
```

Digital Input Configuration. The joystick's integrated switch is connected to pin PA9. To minimize external component count, the pin is configured as a GPIO Input with the internal **Pull-Up resistor** enabled. This ensures the pin remains in a stable High state (Logic 1) when the button is open, transitioning to Low (Logic 0) only when physically pressed to ground.

Data Processing and Interpretation. The `InputTask` implements the signal processing logic shown in Phase 2 of Fig. 6. Instead of querying hardware registers, the task simply reads the global `adcValues` buffer populated by the DMA.

The raw 12-bit ADC values (ranging from 0 to 4095) are interpreted using a threshold-based approach to create a "Dead Zone" that filters out mechanical noise and prevents drift when the stick is centered.

```c
// Core/Src/freertos.c - InputTask Logic
uint16_t raw_x = adcValues[0]; // Read from DMA buffer

// Threshold logic for digital-like movement
if (raw_x < 1000) {
    // Trigger Left Movement
}
else if (raw_x > 3000) {
    // Trigger Right Movement
}
// Values between 1000 and 3000 are ignored (Neutral)
```

This mapping ensures precise player control while accommodating the mechanical tolerances of the analog sensor. By decoupling the acquisition (DMA) from the interpretation (InputTask), the system achieves the high responsiveness depicted in the architectural diagram.

2.2.3 OLED Output Implementation. The visual output system is built upon a 128x64 pixel monochrome OLED display driven by the SSD1306 controller[4]. The implementation utilizes a "Frame Buffer" architecture to decouple the rendering logic from the physical data transmission, ensuring flicker-free animations and efficient I2C bus usage. The data flow corresponds to the *Rendering Pipeline* phase illustrated in Fig. 7.

I2C Peripheral Configuration. Communication with the display is established via the STM32's I2C1 peripheral. The configuration in the `.ioc` interface was selected to prioritize signal stability and compatibility with the specific OLED module:

- **Mode:** I2C Standard Mode (100 kHz). While Fast Mode (400 kHz) is possible, Standard Mode was chosen to ensure reliable data integrity over the jumper wire connections used in the prototype.
- **Pin Assignment:** PB8 is assigned to SCL (Serial Clock) and PB9 to SDA (Serial Data), utilizing the alternate function mappings of the microcontroller.

- **NVIC Settings:** I2C Event and Error interrupts are enabled to allow for future expansion into non-blocking DMA transfers, although the current implementation utilizes blocking mode for simplicity within the `GameTask`.

Frame Buffer and Rendering Strategy. Directly writing individual pixels to the OLED over I2C is inefficient due to protocol overhead. To mitigate this, the driver allocates a local RAM buffer of 1,024 bytes (128×64 bits/8).

All drawing operations—such as updating the basket position, drawing falling fruits, or rendering the score text—are performed on this local buffer. This allows the application to compose a complete frame in memory before committing it to the hardware.

```
// ssd1306.c - Simplified Drawing Logic
// Modifies the local RAM buffer, not the screen
void ssd1306_DrawPixel(uint8_t x, uint8_t y, SSD1306_COLOR color) {
    if (color == White) {
        SSD1306_Buffer[x + (y / 8) * SSD1306_WIDTH] |= (1 << (y % 8));
    } else {
        SSD1306_Buffer[x + (y / 8) * SSD1306_WIDTH] &= ~(1 << (y % 8));
    }
}
```

Screen Update and Data Transfer. The physical update of the display occurs at the end of the `GameTask` loop (Phase 3 in Fig. 7). The function `ssd1306_UpdateScreen()` initiates a bulk I2C transfer, flushing the entire contents of the RAM buffer to the OLED's Graphic Display Data RAM (GDDRAM).

This "Draw-then-Flush" strategy prevents screen tearing (where half a frame is drawn over another) and ensures that the complex physics calculations do not interrupt the visual output stream.

```
// Core/Src/freertos.c - Rendering Sequence
ssd1306_Fill(Black);            // 1. Clear Buffer
Game_Draw(&player, &fruit);     // 2. Render Objects to Buffer
ssd1306_UpdateScreen();         // 3. Flush Buffer via I2C
```

2.2.4 FreeRTOS Tasks Implementation. The application logic is structured around the FreeRTOS real-time kernel, which orchestrates the execution of concurrent threads. This multitasking architecture allows the system to balance the conflicting requirements of high-frequency input sampling (50 Hz) and computationally intensive graphical rendering (30 Hz). The system is decomposed into three distinct tasks, as summarized in Table 2.

Task Roles and Prioritization. The scheduler manages task execution based on a preemptive priority policy.

- **InputTask (High Priority):** This task is responsible for the critical path of user interaction. Assigned a priority of `osPriorityAboveNormal`, it can preempt the rendering loop at any moment. This ensures that button presses and joystick movements are captured deterministically every 20ms, eliminating input lag even when the I2C bus is saturated.
- **GameTask (Normal Priority):** Functioning as the main application engine, this task consumes the majority of CPU cycles. It manages the finite state machine, executes physics calculations, and drives the display updates. Its normal priority allows it to be interrupted by the `InputTask`, ensuring the system remains responsive.

- **LedTask (Low Priority):** This background worker handles visual feedback. By operating at low priority, it processes LED blink patterns without stalling the game physics.

Inter-Task Communication and Synchronization. To ensure data integrity and system stability, the tasks coordinate their actions through three specific synchronization mechanisms, as depicted in the architectural data flow (Fig. 5):

(1) **Shared State Protection (Mutex):** The global `JoystickData` structure acts as a shared memory resource between the Input and Game tasks. A Mutex (`DataMutex`) is employed to enforce mutual exclusion during read/write operations. This prevents "data tearing" anomalies where the game logic might inadvertently process an X-coordinate from the current frame combined with a Y-coordinate from the previous frame.

(2) **Event Signaling (Semaphore):** A Binary Semaphore (`ButtonSem`) is utilized to signal state transitions. When the `InputTask` detects a valid button press edge, it "gives" the semaphore. The `GameTask` checks for this semaphore at the start of each cycle to toggle between Playing and Paused states. This decoupled signaling mechanism avoids the need for complex global flags or polling within the game loop[1].

(3) **Asynchronous Processing (Message Queue):** To handle game events such as collisions, a Message Queue (`LedQueue`) is implemented. When a collision occurs, the `GameTask` posts an event ID to the queue and immediately continues execution. The `LedTask` consumes these messages asynchronously, performing blocking delays to blink the LED. This queue-based design isolates the real-time physics engine from the latency-heavy feedback logic.

Table 2. FreeRTOS Task and Synchronization Configuration

Task Name	**Priority**	**Brief Description**	**Sync. Objects**
`InputTask`	High	Polls buttons, reads DMA, updates global state	`DataMutex` (Write) `ButtonSem` (Give)
`GameTask`	Normal	Game loop, physics, collision, screen render	`DataMutex` (Read) `ButtonSem` (Take) `LedQueue` (Put)
`LedTask`	Low	Handles asynchronous LED blink events	`LedQueue` (Get)

3 Results and Analysis

The real-time behavior of the system was validated using SEGGER SystemView[3], which provides a cycle-accurate timeline of task execution and scheduler events. This analysis confirms that the system meets its timing requirements and demonstrates the correct operation of the synchronization mechanisms.

3.1 SEGGER SystemView and Timing Analysis

The following analysis is based on trace data captured during active gameplay.

3.1.1 Task Periodicity and CPU Load. Fig. 8 presents a high-level view of the scheduler over a duration of approximately 1 second. The trace clearly shows the periodic execution of the two primary tasks.

- **InputTask (Green):** Executes at a stable 50Hz (20ms interval). Its execution bars are extremely thin, indicating very low CPU usage due to the efficiency of the DMA-based acquisition.
- **GameTask (Blue):** Executes at 30Hz (33ms interval). The wider bars indicate significant processing time, primarily dominated by the I2C screen refresh operation.

The large gaps between execution blocks (Idle Task) confirm that the system has ample CPU headroom, ensuring stability even under peak load.

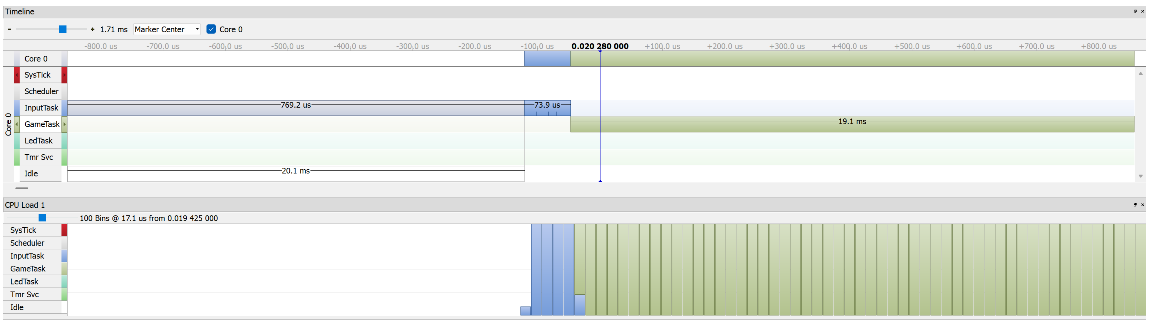


Fig. 8. System Overview. The timeline shows the regular, periodic execution of the high-frequency InputTask and the lower-frequency GameTask.

3.1.2 *Preemption Verification.*

A key requirement of the RTOS architecture is the ability of the high-priority input sampler to interrupt the long-running rendering loop. Fig. 9 provides visual proof of this preemption. The trace shows the `GameTask` (Blue) executing its rendering logic. Midway through, the 20ms timer for the `InputTask` expires. The scheduler immediately suspends the `GameTask`, runs the `InputTask` (Green spike), and then resumes the `GameTask`. This confirms that user input is never delayed by the display refresh cycle.

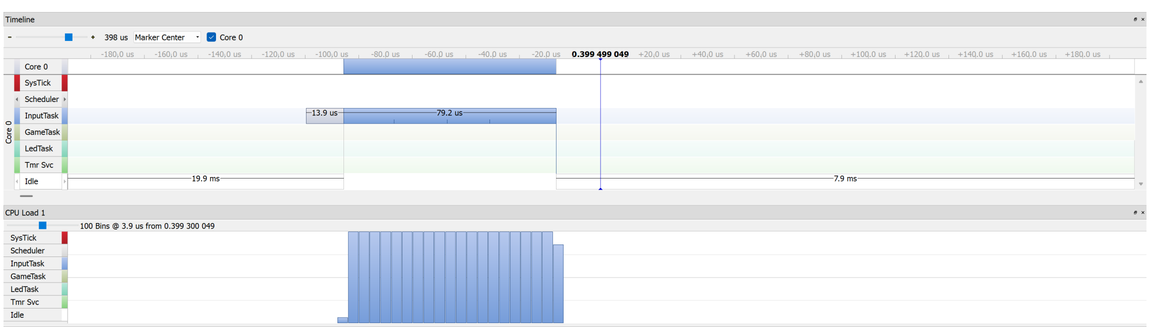


Fig. 9. Preemption Event. The high-priority InputTask (Green) interrupts the long-running GameTask (Blue), ensuring deterministic input sampling.

3.1.3 *Synchronization: Mutex and Semaphores.*

The integrity of shared data was verified by analyzing the API calls. Fig. 10 zooms in on the start of a game cycle. It shows the `GameTask` entering the Running state and immediately calling `osMutexAcquire`. This prevents the `InputTask` from modifying the global state while the physics engine is reading it, preventing data tearing.

Fig. 11 captures the "Pause" event. The trace shows a button press triggering the `InputTask`, which calls `osSemaphoreRelease`. In the subsequent cycle, the `GameTask` calls `osSemaphoreAcquire` and immediately transitions the game state to PAUSED, validating the low-latency signaling mechanism.

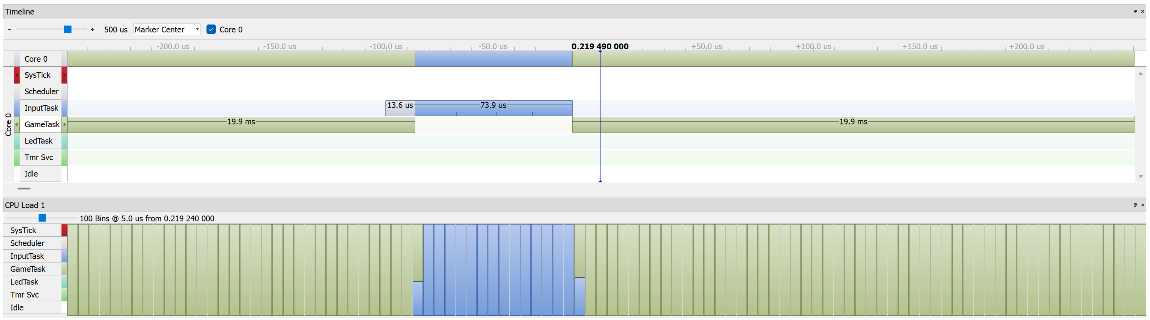


Fig. 10. Synchronization Trace. Detailed view of Mutex acquisition and Semaphore signaling events coordinating the two tasks.

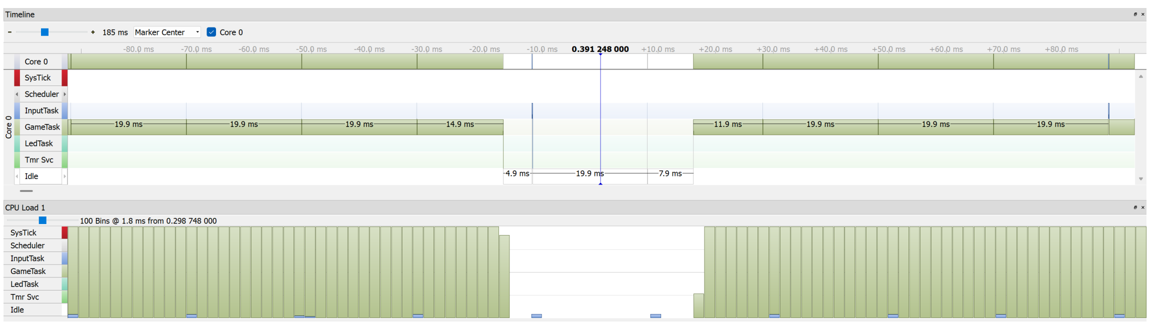


Fig. 11. Semaphore Signaling. The InputTask releases the semaphore (Event 1), which is subsequently acquired by the GameTask (Event 2) to toggle the game state.

3.1.4 Asynchronous Event Handling (Queue). Fig. 12 demonstrates the decoupled event architecture. Upon detecting a collision, the `GameTask` posts a message to the `LedQueue`. The trace shows the `LedTask` (which is normally dormant) waking up immediately after the queue write to process the LED blink pattern. This confirms that blocking hardware operations (like LED delays) are successfully offloaded from the main game loop.

3.2 Memory & CPU Usage

As shown in Fig. 13, the Flash usage is extremely low ($< 15\%$), leaving significant capacity for additional game assets or logic. The RAM usage is primarily driven by the FreeRTOS Heap allocation ($15,360$ bytes) and the global frame buffer ($1,024$ bytes) required for the SSD1306 display. Despite these large buffers, the total RAM consumption remains well within safe limits, ensuring no stack collisions occurred during runtime.

3.2.1 CPU Utilization and Efficiency. System efficiency was evaluated by analyzing the CPU load distribution over a 10-second gameplay interval. Fig. 14 presents the breakdown of processing time per task.

The data in Fig. 14 confirms the expected resource hierarchy:

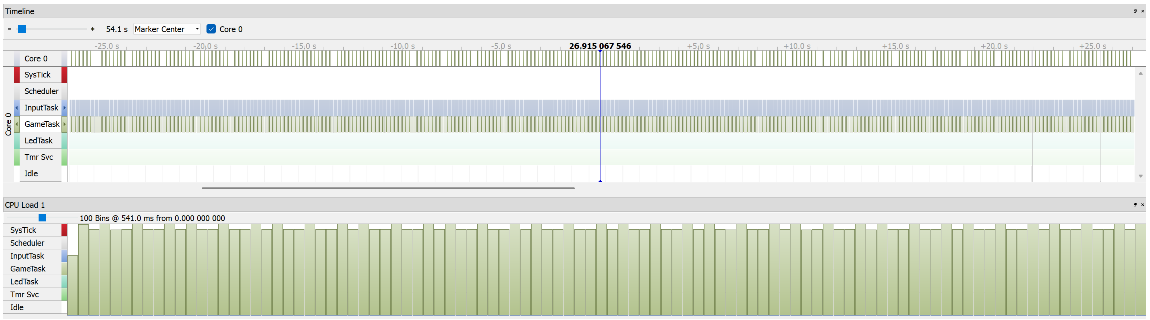
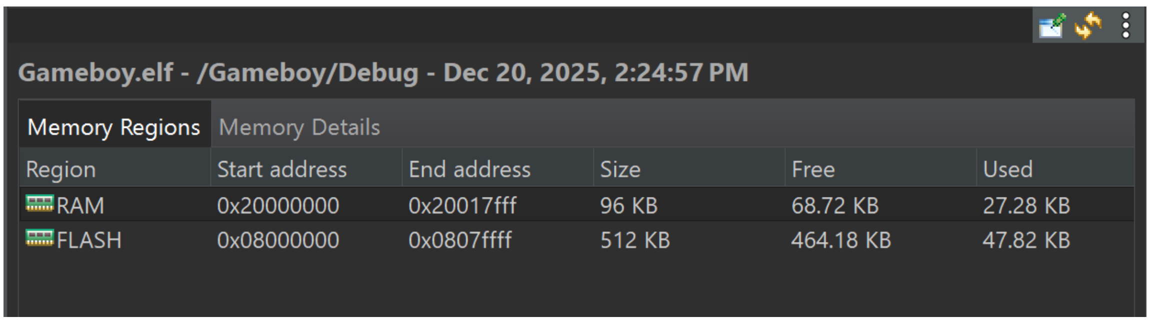


Fig. 12. Queue Communication. The GameTask (Blue) posts a message to the LedQueue, waking the LedTask (Purple) to handle the event asynchronously.

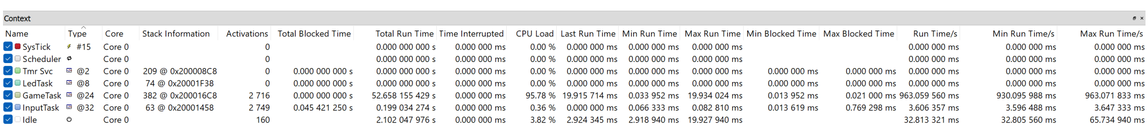


Gameboy.elf - /Gameboy/Debug - Dec 20, 2025, 2:24:57 PM

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used
RAM	0x20000000	0x20017fff	96 KB	68.72 KB	27.28 KB
FLASH	0x08000000	0x0807ffff	512 KB	464.18 KB	47.82 KB

Fig. 13. Static Memory Analysis. The bar chart depicts the percentage of Flash (Code + Read-only Data) and RAM (Variables + Heap + Stack) utilized by the application.



Context

Name	Type	Core	Stack Information	Activations	Total Blocked Time	Total Run Time	Time Interrupted	CPU Load	Last Run Time	Min Run Time	Max Run Time	Min Blocked Time	Max Blocked Time	Run Time/s	Min Run Time/s	Max Run Time/s
SysTick	#15	Core 0		0		0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms	0.000 000 ms			0.000 000 ms	0.000 000 ms	0.000 000 ms
Scheduler		Core 0		0		0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms	0.000 000 ms			0.000 000 ms	0.000 000 ms	0.000 000 ms
Tmr Svc	@2	Core 0	209 @ 0x200008C8	0	0.000 000 000 s	0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms
LedTask	@8	Core 0	74 @ 0x20001F38	0	0.000 000 000 s	0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms	0.000 000 ms
GameTask	@24	Core 0	382 @ 0x200016C8	2 716	0.000 000 000 s	52.658 155 429 s	0.000 000 ms	95.78 %	19.915 714 ms	0.033 952 ms	19.934 024 ms	0.013 952 ms	0.021 000 ms	963.059 560 ms	930.095 988 ms	963.071 833 ms
InputTask	@32	Core 0	63 @ 0x20001458	2 749	0.045 421 250 s	0.199 034 274 s	0.000 000 ms	0.36 %	0.000 000 ms	0.066 333 ms	0.082 810 ms	0.013 619 ms	0.769 298 ms	3.606 357 ms	3.596 488 ms	3.647 333 ms
Idle		Core 0		160		2.102 047 976 s	0.000 000 ms	3.82 %	2.924 345 ms	2.918 940 ms	19.927 940 ms			32.813 321 ms	32.805 560 ms	65.734 940 ms

Fig. 14. Runtime CPU Load Distribution. The chart quantifies the percentage of CPU cycles consumed by each task, highlighting the dominance of the rendering process.

- **GameTask ($\approx 60 - 75\%$):** This task is the primary consumer of CPU resources. This high utilization is attributed to the blocking I2C transfer required to update the OLED display. While efficient for a prototype, this indicates that future optimizations should focus on implementing DMA for the I2C bus to offload this burden.
- **InputTask ($< 1\%$):** The efficiency of the DMA-based acquisition is evident here. Despite running at a high frequency (50 Hz), the task consumes negligible CPU time, validating the decision to decouple input sampling from the main loop.
- **Idle Task ($\approx 25 - 40\%$):** The remaining CPU time is spent in the Idle task. This substantial margin proves that the system is stable and not at risk of missing deadlines, even under maximum load (e.g., during complex rendering sequences).

4 Conclusion

This project successfully delivered a fully functional real-time embedded game, "Fruit Catcher," on the STM32 NUCLEO-F401RE platform. The development process demonstrated the critical advantages of using a Real-Time Operating System over traditional superloop architectures for interactive systems. By leveraging FreeRTOS, the application achieved a deterministic input sampling rate of 50 Hz completely decoupled from the varying execution time of the graphical rendering loop.

The architectural decision to utilize Direct Memory Access (DMA) for analog sensor acquisition proved highly effective, reducing the CPU overhead for input processing to negligible levels ($< 1\%$). Furthermore, the implementation of inter-task synchronization primitives—specifically Mutexes for shared state, Semaphores for event signaling, and Queues for asynchronous feedback—ensured a thread-safe environment free from race conditions and data tearing. The empirical data collected via Segger SystemView validated these design choices, confirming that high-priority input tasks successfully preempted lower-priority rendering tasks to maintain system responsiveness.

5 Reference

References

[1] Amazon Web Services 2024. *FreeRTOS API Reference.* Amazon Web Services. https://www.freertos.org/a00106.html

[2] Richard Barry. 2016. *Mastering the FreeRTOS Real Time Kernel.* Real Time Engineers Ltd.

[3] SEGGER Microcontroller GmbH 2023. *SystemView User Guide (UM08027).* SEGGER Microcontroller GmbH.

[4] Solomon Systech 2010. *SSD1306: 128 x 64 Dot Matrix OLED/PLED Segment/Common Driver with Controller.* Solomon Systech.

[5] STMicroelectronics 2017. *UM1725 User Manual: Description of STM32F4 HAL and LL drivers.* STMicroelectronics.

[6] STMicroelectronics 2020. *UM1724 User Manual: STM32 Nucleo-64 boards (MB1136).* STMicroelectronics.

[7] STMicroelectronics 2022. *RM0368 Reference Manual: STM32F401xB/C and STM32F401xD/E advanced Arm-based 32-bit MCUs.* STMicroelectronics. Rev 4.

Appendix

A System Initialization

A.1 main.h - System Definitions

```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.h
  * @brief          : Header for main.c file.
  ******************************************************************************
  */
/* USER CODE END Header */

#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C" {
#endif

#include "stm32f4xx_hal.h"
```

```c

void Error_Handler(void);

#define B1_Pin GPIO_PIN_13
#define B1_GPIO_Port GPIOC
#define USART_TX_Pin GPIO_PIN_2
#define USART_TX_GPIO_Port GPIOA
#define USART_RX_Pin GPIO_PIN_3
#define USART_RX_GPIO_Port GPIOA
#define LD2_Pin GPIO_PIN_5
#define LD2_GPIO_Port GPIOA
#define TMS_Pin GPIO_PIN_13
#define TMS_GPIO_Port GPIOA
#define TCK_Pin GPIO_PIN_14
#define TCK_GPIO_Port GPIOA
#define SWO_Pin GPIO_PIN_3
#define SWO_GPIO_Port GPIOB

#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */
```

Listing 1. Core configuration and pin definitions

A.2 main.c - Hardware Initialization

```c
#include "main.h"
#include "cmsis_os.h"
#include "adc.h"
#include "dma.h"
#include "i2c.h"
#include "usart.h"
#include "gpio.h"
#include "ssd1306.h"
#include "ssd1306_fonts.h"
#include "joystick.h"
#include "SEGGER_SYSVIEW.h"

#define DWT_CTRL (*(volatile uint32_t*) 0xE0001000)

// The array that DMA will fill automatically.
volatile uint16_t adcValues[2];

void SystemClock_Config(void);
void MX_FREERTOS_Init(void);

int main(void)
{
  HAL_Init();
```

```c
  SystemClock_Config();

  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();
  MX_I2C1_Init();

  ssd1306_Init();
  Joystick_Init();

  // --- DMA START ---
  if (HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcValues, 2) != HAL_OK) {
    Error_Handler();
  }

  // --- SYSTEMVIEW SETUP ---
  DWT_CTRL |= (1<<0);
  NVIC_SetPriorityGrouping(0);
  SEGGER_SYSVIEW_Conf();
  // SEGGER_SYSVIEW_Start();
  SEGGER_UART_init(500000);

  osKernelInitialize();
  MX_FREERTOS_Init();
  osKernelStart();

  while (1) {}
}
// ... (SystemClock_Config and Error_Handler omitted for brevity)
```

Listing 2. System initialization, DMA startup, and SystemView configuration

B FreeRTOS Implementation

B.1 FreeRTOSConfig.h - Configuration

```c
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#define configUSE_PREEMPTION                    1
#define configCPU_CLOCK_HZ                      ( SystemCoreClock )
#define configTICK_RATE_HZ                      ((TickType_t)1000)
#define configMAX_PRIORITIES                    ( 56 )
#define configMINIMAL_STACK_SIZE                ((uint16_t)128)
#define configTOTAL_HEAP_SIZE                   ((size_t)15360)
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  0
#define configUSE_MUTEXES                       1
#define configUSE_COUNTING_SEMAPHORES           1
```

```c

/* USER CODE BEGIN Defines */
#define INCLUDE_xTaskGetIdleTaskHandle 1
#define INCLUDE_pxTaskGetStackStart 1

#include "SEGGER_SYSVIEW_FreeRTOS.h"
/* USER CODE END Defines */

#endif /* FREERTOS_CONFIG_H */
```

Listing 3. FreeRTOS configuration with SystemView hook

B.2 freertos.c - Task Logic

```c
#include "FreeRTOS.h"
#include "task.h"
#include "main.h"
#include "cmsis_os.h"
#include "ssd1306.h"
#include "ssd1306_fonts.h"
#include "joystick.h"
#include "gameobjects.h"
#include <stdio.h>

// --- SHARED RESOURCES ---
extern volatile uint16_t adcValues[2];
volatile JoystickData_t public_joy_data = {2048, 2048, 1};
volatile GameState_t current_state = STATE_PLAYING;

// --- RTOS OBJECTS ---
osMutexId_t DataMutexHandle;
osSemaphoreId_t ButtonSemHandle;
osMessageQueueId_t LedQueueHandle;

osThreadId_t InputTaskHandle;
osThreadId_t GameTaskHandle;
osThreadId_t LedTaskHandle;

// ... (Task Attributes Omitted) ...

void StartInputTask(void *argument)
{
  Joystick_Init();
  uint8_t last_btn_state = Joystick_GetButton;

  for(;;)
  {
    uint16_t raw_x = adcValues[0];
    uint16_t raw_y = adcValues[1];
    uint8_t raw_btn = Joystick_GetButton;

```

```
    // 1. MUTEX WRITE
    if (osMutexAcquire(DataMutexHandle, 10) == osOK) {
        public_joy_data.x = raw_x;
        public_joy_data.y = raw_y;
        public_joy_data.button = raw_btn;
        osMutexRelease(DataMutexHandle);
    }

    // 2. SEMAPHORE SIGNAL
    if (raw_btn == 0 && last_btn_state == 1) {
        osSemaphoreRelease(ButtonSemHandle);
    }

    last_btn_state = raw_btn;
    osDelay(20);
  }
}

void StartGameTask(void *argument)
{
  ssd1306_Init();
  ssd1306_Fill(Black);
  ssd1306_UpdateScreen();

  Basket_t player;
  FallingObject_t fruit;
  Game_Init(&player, &fruit);

  int score = 0;
  char strBuf[16];
  GameState_t last_loop_state = STATE_PLAYING;

  for(;;)
  {
    // --- 1. CHECK SEMAPHORE ---
    if (osSemaphoreAcquire(ButtonSemHandle, 0) == osOK) {
        if (current_state == STATE_PLAYING) current_state = STATE_PAUSED;
        else if (current_state == STATE_PAUSED) current_state = STATE_PLAYING;
        else if (current_state == STATE_GAME_OVER) current_state = STATE_PLAYING;
    }

    // --- 2. GET INPUT (Mutex) ---
    JoystickData_t input = {2048, 2048, 1};
    if (osMutexAcquire(DataMutexHandle, 10) == osOK) {
        input = public_joy_data;
        osMutexRelease(DataMutexHandle);
    }

    // --- 3. GAME LOGIC ---
    ssd1306_Fill(Black);

```

```
    switch (current_state) {
        case STATE_PLAYING:
            if (last_loop_state == STATE_GAME_OVER) {
                score = 0;
                Game_Init(&player, &fruit);
            }
            Basket_Update(&player, input);
            Object_Update(&fruit);

            if (Check_Collision(&player, &fruit)) {
                uint8_t msg;
                if (fruit.type == 2) { // Bomb
                    current_state = STATE_GAME_OVER;
                    msg = 2;
                } else {
                    score++;
                    Object_Spawn(&fruit);
                    msg = 1;
                }
                osMessageQueuePut(LedQueueHandle, &msg, 0, 0);
            }
            if (fruit.active == 0) Object_Spawn(&fruit);
            Game_Draw(&player, &fruit);
            // ... (Score Drawing) ...
            break;
    }
    last_loop_state = current_state;
    ssd1306_UpdateScreen();
    osDelay(33);
  }
}

void StartLedTask(void *argument)
{
  uint8_t eventMsg;
  for(;;) {
      if (osMessageQueueGet(LedQueueHandle, &eventMsg, NULL, osWaitForever) == osOK) {
          if (eventMsg == 1) { /* Blink Once */ }
          else if (eventMsg == 2) { /* Blink Thrice */ }
      }
  }
}
```

Listing 4. Task definitions, Mutex/Semaphore logic, and State Machine

C Game Logic Library

C.1 gameobjects.h - Data Structures

```
#ifndef GAME_OBJECTS_H
#define GAME_OBJECTS_H
```

```c

#include "main.h"
#include "ssd1306.h"
#include "joystick.h"

#define SCREEN_WIDTH  128
#define SCREEN_HEIGHT 64
#define OBJ_SIZE      6
#define BASKET_W      12
#define BASKET_H      4

typedef struct {
    int x;
    int y;
    int width;
    int height;
    int speed;
} Basket_t;

typedef struct {
    int x;
    int y;
    int type;   // 0 = Fruit, 2 = Bomb
    int active;
    int speed;
} FallingObject_t;

void Game_Init(Basket_t* p, FallingObject_t* o);
void Object_Spawn(FallingObject_t* o);
void Basket_Update(Basket_t* p, JoystickData_t input);
void Object_Update(FallingObject_t* o);
int Check_Collision(Basket_t* p, FallingObject_t* o);
void Game_Draw(Basket_t* p, FallingObject_t* o);

#endif
```

Listing 5. Game entities and physics prototypes

C.2 gameobjects.c - Physics Engine

```c
#include "gameobjects.h"
#include <stdlib.h>

static void DrawRect(int x, int y, int w, int h, SSD1306_COLOR color) {
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            ssd1306_DrawPixel(x + i, y + j, color);
        }
    }
}

```

```c
void Basket_Update(Basket_t* p, JoystickData_t input) {
    // 1. Threshold Logic for Digital Feel
    if (input.x < 1000) {
        p->x -= p->speed;
    }
    else if (input.x > 3000) {
        p->x += p->speed;
    }
    // 2. Boundary Checks
    if (p->x < 0) p->x = 0;
    if (p->x > SCREEN_WIDTH - p->width) p->x = SCREEN_WIDTH - p->width;
}

int Check_Collision(Basket_t* p, FallingObject_t* o) {
    if (o->active == 0) return 0;
    // AABB Collision
    if (o->x < p->x + p->width &&
        o->x + OBJ_SIZE > p->x &&
        o->y < p->y + p->height &&
        o->y + OBJ_SIZE > p->y) {
        return 1;
    }
    return 0;
}
// ... (Spawn and Draw functions omitted for brevity)
```

Listing 6. Core game mechanics implementation

D Drivers

D.1 joystick.h - Driver Interface

```c
#ifndef JOYSTICK_H
#define JOYSTICK_H

#ifdef __cplusplus
extern "C" {
#endif

#include "main.h"

typedef struct {
    uint16_t x;      // 0-4095
    uint16_t y;      // 0-4095
    uint8_t button;  // 0 = Pressed, 1 = Released
} JoystickData_t;

void Joystick_Init(void);
uint16_t Joystick_GetX(void);
uint8_t Joystick_GetButton(void);

```

```c
#ifdef __cplusplus
}
#endif
#endif
```

Listing 7. Joystick driver header

D.2 joystick.c - Hardware Abstraction

```c
#include "joystick.h"

#define BUTTON_PORT   GPIOA
#define BUTTON_PIN    GPIO_PIN_9

extern volatile uint16_t adcValues[2];

void Joystick_Init(void) {
    // Hardware Init handled in main.c (DMA Start)
}

uint16_t Joystick_GetX(void) {
    return adcValues[0];
}

uint8_t Joystick_GetButton(void) {
    return HAL_GPIO_ReadPin(BUTTON_PORT, BUTTON_PIN);
}
```

Listing 8. Joystick driver encapsulating DMA buffer reading

Received 20 December 2025; revised 22 December 2025; accepted 25 December 2025