Videojuego portátil inspirado en consolas retro

Documento de arquitectura de software

Autor:
Lic. Jezabel Danon (jezabel.danon@gmail.com)

27/07/2025
Versión A

Este documento fue creado durante el curso de Ingeniería de Software entre el 26 de junio de 2025 y el 21 de agosto de 2025.

Historial de cambios

Versión	Fecha	Descripción	Autor	Revisores
A	27/07/2025	Creación del documento	Lic. Jezabel Danon	

Índice

1. Introducción

1.1. Propósito

1. Este documento describe la arquitectura de software para el sistema embebido *Videojuego portátil inspirado en consolas retro*.
2. Está dirigido a los desarrolladores que se ocupen del análisis, diseño e implementación del software, así como también a quienes desarrollen el testing, validaciones y/o verificaciones del mismo.

1.2. Ámbito del sistema

1. El software a desarrollar controlará la interfaz de usuario de la consola de juegos y procesará las entradas y salidas de los periféricos.
2. Las entradas del sistema serán:
   - Botones
   - Joystick analógico
   - Acelerómetro
3. Las salidas del sistema serán:
   - Pantalla
   - Salida de audio
   - Motor de vibraciones

1.3. Definiciones, Acrónimos y Abreviaturas

1. UART: Universal Asynchronous Receiver/Transmitter.
2. RTOS: Sistema Operativo de Tiempo Real (Real Time Operating System).
3. HAL: Hardware Abstraction Layer (STM32Cube HAL/LL).
4. API: Application Programming Interface.
5. PWM: Pulse Width Modulation
6. PCM: Pulse Code Modulation.
7. N/A: No Aplica.

1.4. Referencias

1. Plan de proyecto del trabajo práctico final para la *Carrera de Especialización en Sistemas Embebidos* (RETRO_GAME-PP-v5).
2. Especificaciones de requisitos de software: RETRO_GAME-RS-vA.
3. Especificaciones de requisitos de hardware: RETRO_GAME-RH-vA.
4. Especificaciones de casos de uso: RETRO_GAME-CU-vA.
5. B. Douglass — *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*.

1.5. Visión general del documento

1. Este documento incluye la definición de los patrones arquitectura aplicados.
2. Se incluye la definición de los componentes de software, sus responsabilidades e interfaces internas y externas.

2. Arquitectura

2.1. Patrones

Para este software se emplearán los siguientes patrones de arquitectura:

1. Arquitectura en capas (Layered Architecture).
2. Capa de abstracción de hardware (HAL).
3. Observar y reaccionar.
4. Segmentación de procesos.

2.1.1. Arquitectura en capas

1. Este patrón es utilizado cuando se desea separar la funcionalidad del software por niveles de abstracción.
2. Entre sus características principales se encuentran:
   - Acceso únicamente a la/s capa/s inmediatamente anterior/es.
   - Mantenimiento de una API *hacia arriba* estable.
3. En este proyecto se emplearán las siguientes capas:
   a) Capa de lógica del juego: implementa la lógica específica de gameplay, el renderizado de escenas y la interpretación de las entradas como acciones del juego.
   b) Capa de lógica del sistema: incluye los componentes que gestionan el estado global del sistema, la persistencia, los menús de usuario, la carga de recursos y la gestión de entradas.
   c) Capa de drivers: contiene los controladores de hardware y sus rutinas de inicialización.
   d) Capas provistas por la plataforma y middlewares de terceros:
      - RTOS: Se utilizará FreeRTOS como sistema operativo de tiempo real.
      - CMSIS + STM HAL: bibliotecas oficiales para manejo del hardware del microcontrolador.

2.1.2. Capa de abstracción de hardware (HAL)

1. Este patrón se utiliza cuando se desea abstraer el manejo del hardware de las funcionalidades propias de la aplicación.
2. Se utilizará la capa de abstracción HAL provista por STM, además de incorporar una capa de drivers que la utilizará directamente para lograr una abstracción mayor hacia las capas de aplicación.

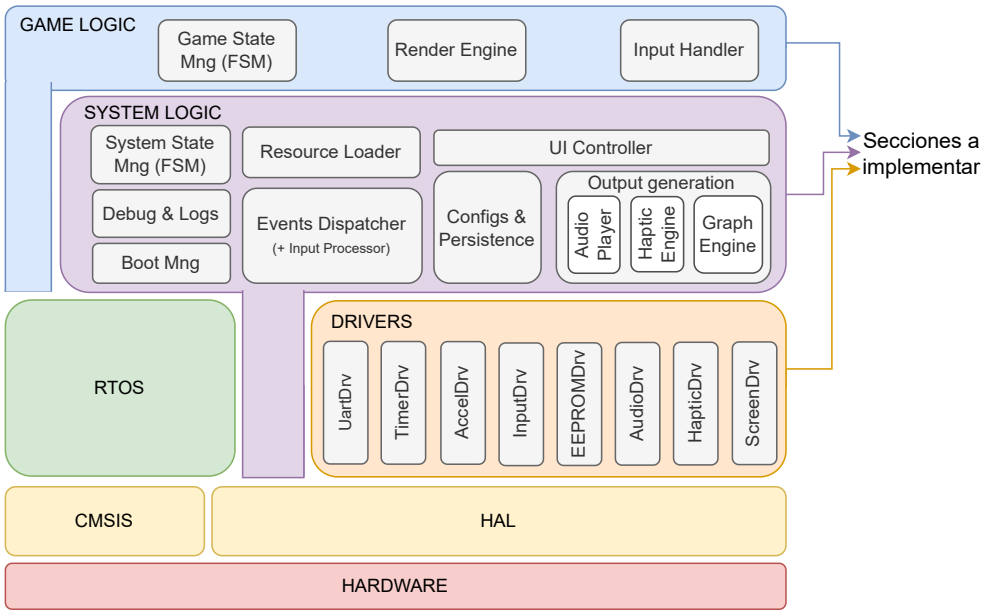

Figura 1: Diagrama de capas del sistema.

2.1.3. Observar y reaccionar

1. Este patrón se utiliza cuando un conjunto de sensores se monitorean y despliegan una respuesta de manera rutinaria.

2. Este patrón describe el comportamiento del sistema en términos de procesos observadores de entradas, procesos de despliegue de salidas y un proceso central de análisis de la información y reacción apropiada.

3. Se aplicará este patrón de arquitectura en ambas capas de aplicación:

   a) Para la capa de lógica del sistema, el proceso observador será el componente *Event Dispatcher*, los procesos centrales serán *System State Manager* y *UI Controller*, mientras que los procesos de despliegue serán: *Audio Player*, *Haptic Engine* y *Graph Engine*

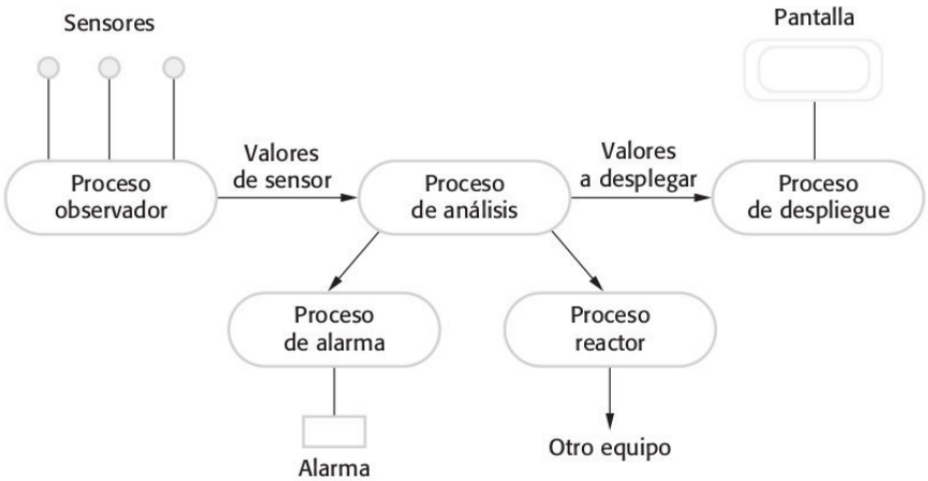

Figura 2: Patrón observar y reaccionar. Imagen del apunte de la materia Ingeniería de Software.

b) Para la capa de lógica del juego, el proceso observador será el *Input Handler*, el proceso central será el *Game State Manager* y el proceso de despliegue será el *Render Engine*.

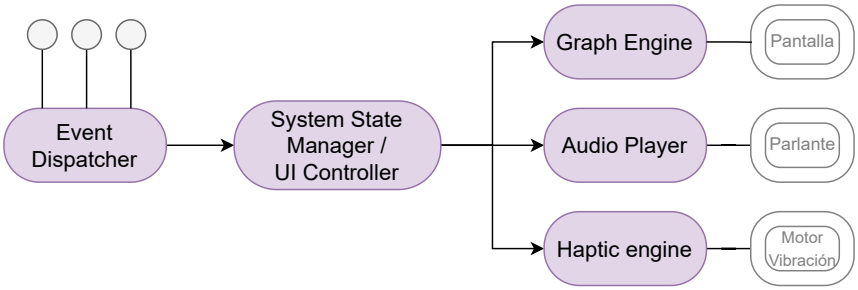

Figura 3: Observar y reaccionar: patrón aplicado a la capa del sistema.

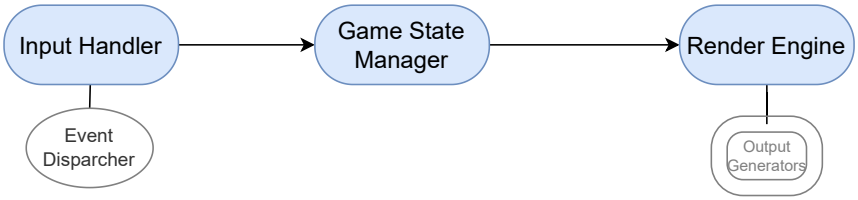

Figura 4: Observar y reaccionar: patrón aplicado a la capa del juego.

2.1.4. Segmentación de procesos

1. Este patrón se usa al transformarse datos de una representación a otra antes de que puedan procesarse.
2. Es empleado para transformar y trasladar datos a lo largo de varias etapas desacopladas mediante búferes.
3. Será implementado para la lectura de datos, el envío de datos hacia los actuadores y las comunicaciones internas entre procesos por medio de colas y un proceso ruteador de eventos *Event Dispatcher*.

Figura 5: Patrón segmentación de procesos. Imagen del apunte de la materia Ingeniería de Software.

2.2. Componentes

2.2.1. Drivers

1. Driver de acelerómetro (`AccelDrv`): lectura de acelerómetro por I^2C.
2. Driver de memoria (`EEPROMDrv`): lectura y escritura de memoria EEPROM por SPI.

3. Driver de pantalla (`ScreenDrv`): driver de la pantalla, gestion de ventana y transferencias DMA-SPI de píxeles.
4. Driver de audio (`AudioDrv`): reproduce efectos de sonido mediante modulación por ancho de pulso (PWM).
5. Driver de vibración (`HapticDrv`): comanda patrones de vibración del controlador DRV2605L mediante I^2C.
6. Driver de UART (`UARTDrv`): comunicación serie con DMA para depuración.
7. Driver de entradas (`InputDrv`): lectura de botones (GPIO + EXTI) y joystick (ADC DMA continuo), aplica antirrebote y dead-zone.
8. Driver de timers (`TimerDrv`): generación de interrupciones de tiempo.

2.2.2. Lógica del sistema

1. Boot Manager (`BootMgr`): inicialización del RTOS, creación de tareas y colas, verificación de drivers.
2. Cargador de recursos (`ResLoader`): carga de recursos (imágenes, sonidos) en RAM desde memoria externa.
3. Configuración y persistencia (`ConfigPersist`): guarda y carga datos de las partidas del juego en memoria externa. Verificación de partida guardada.
4. Despachador de eventos (`EventDispatcher`): recibe eventos de bajo nivel (inputs, ticks), los normaliza y redistribuye a colas de alto nivel. Recibe y redistribuye eventos y mensajes entre componentes.
5. Manejador de la interfaz de usuario (`UIController`): gestión de la interfaz de usuario del sistema en los estados no interactivos del juego (menús, splash, opciones de pausa). Funciones principales:
   - Construcción de las pantallas de interfaz (Splash, Menú principal, Menú de pausa).
   - Gestión del foco y navegación de menús.
   - Interpretación de eventos de entrada normalizados y publicación de eventos de transición de estado.
   - Coordinación con servicios de generación de salidas para brindar retroalimentación visual, sonora y háptica.
   - Coordinación con `ConfigPersist` para comprobar si hay partidas guardadas válidas.
   - Acceso a recursos gráficos (iconos, textos, imágenes) mediante el `ResLoader`.
6. Motor gráfico (`GraphEngine`): renderizado gráfico, gestión de buffers de pantalla.
7. Reproductor de audio (`AudioPlayer`): reproducción de efectos de sonido.
8. Motor de retroalimentación táctil (`HapticEngine`): gestión de la activación de patrones de vibración.
9. System State Manager (`SysMng`): implementación de máquina de estados principal del sistema, gestión de transiciones entre distintos modos de operación (`SYS_SPLASH, SYS_MAIN_MENU, SYS_IN_GAME, SYS_PAUSED`).
10. Debug & Logs (`LogSink`): gestión de comunicaciones por UART y trazas de depuración.

2.2.3. Lógica del juego

1. Game State Manager (`GameMng`): control del ciclo principal del juego, construcción del modelo gráfico del juego a partir del estado del gameplay. Manejo de los estados internos del juego (`GME_RUNNING, GME_FROZEN, GME_STOPPED`).
2. Motor de renderizado (`RenderEngine`): generación del contenido visual del juego para cada cuadro, coordinación con servicios de generación de salidas para brindar retroalimentación visual, sonora y háptica. Opera únicamente durante `GME_RUNNING`.
3. Gestión de entradas (`InputHandler`): traducción de los eventos normalizados de entradas del `EventDispatcher` a acciones de gameplay. Opera únicamente durante `GME_RUNNING`.

2.3. Interfaces

2.3.1. Interfaces externas

1. `InputDrv`: botones (GPIO + EXTI) y joystick (ADC + DMA).
2. `AccelDrv`: acelerómetro MPU-6500 I^2C (400 kHz)
3. `EEPROMDrv`: EEPROM 25LC256 SPI (10 MHz).
4. `ScreenDrv`: ST7735 TFT, 128x160 px. SPI + DMA (48 MHz).
5. `AudioDrv`: LM386 + audio mono 1 W (PWM / GPIO).
6. `HapticDrv`: controlador háptico DRV2605L I^2C (400 kHz).
7. `UARTDrv`: UART + DMA (115200bps 8N1).

2.3.2. Interfaces internas

1. `InputDrv`
   a) Entradas: no requiere.
   b) Salidas: datos crudos (`BTN_X_PRESS, {JY_X, JY_Y}`).
2. `AccelDrv`
   a) Entradas: no requiere.
   b) Salidas: datos crudos (`{ACC_X, ACC_Y}`).
3. `EEPROMDrv`
   a) Escritura:
      1) Entradas: data serializada, dirección de guardado.
      2) Salidas: guardado exitoso/fallido (bool o enum).
   b) Lectura:
      1) Entradas: dirección de lectura.
      2) Salidas: data serializada.
4. `ScreenDrv`
   a) Entradas: paquete de dibujo con bitmap RGB565 y region de pantalla a escribir (struct: `drawPacket{x1,x2,y1,y2,bitmap}`).
   b) Salidas: escritura de datos finalizada correctamente/fallida (bool o enum).

5. `AudioDrv`
   a) Entradas: frecuencia fija o muestras PCM, duración (struct: `{*pcm, durtionMs}`).
   b) Salidas: envío de datos finalizado correctamente/fallido (bool o enum).
6. `HapticDrv`
   a) Entradas: id del patrón de vibración a reproducir.
   b) Salidas: envío de datos finalizado correctamente/fallido (bool o enum).
7. `UARTDrv`
   a) Envío:
      1) Entradas: buffer de datos a enviar.
      2) Salidas: envío de datos finalizado correctamente/fallido (bool o enum).
   b) Recepción:
      1) Entradas: buffer de recepción de datos.
      2) Salidas: recepción de datos finalizado correctamente/fallido (bool o enum).
8. `BootMng`
   a) Entradas: status de drivers inicializados (bool o enum).
   b) Salidas: señal de inicialización completa (`SYS_READY`).
9. `ResLoader`
   a) Entradas: id del asset a cargar (unsigned int).
   b) Salidas: asset serializado, señal de asset disponible (`RES_READY`).
10. `ConfigPersist`
    a) Existencia de partida guardada:
       1) Entradas: no requiere.
       2) Salidas: partida encontrada/no encontrada (bool o enum).
    b) Guardar partida:
       1) Entradas: partida serializada.
       2) Salidas: guardado de datos finalizado correctamente/fallido (bool o enum).
    c) Cargar partida:
       1) Entradas: no requiere.
       2) Salidas: partida cargada/fallo en carga (bool o enum).
11. `EventDispatcher`
    a) Procesamiento de entradas:
       1) Entradas: datos crudos de entradas (enum, struct: `BTN_X_PRESS, {JY_X, JY_Y}, {ACC_X, ACC_Y}`).
       2) Salidas: eventos normalizados de entradas (enum, struct: `BTN_START, KEY_UP, TILT_RIGHT, JYcoords{}`).
    b) Despachador de eventos:
       1) Entradas: eventos varios (enum, struct).
       2) Salidas: eventos reenviados (enum, struct).

c) Activar/desactivar trazas de depuración:

1) Entradas: señal de traza activada/desactivada (bool o enum).
2) Salidas: no emite.

12. `UIController`

a) Entradas: eventos varios (enum, struct).

b) Salidas:

- eventos de transición de estado del sistema y del juego (enum).
- datos para reproducción de audio (struct: `{id, duration}`).
- datos para generación de vibración (enum: `SEND_PATTERN_1, SEND_PATTERN_2`).
- lista de objetos gráficos (struct: `drawList{}`).

13. `GraphEngine`

a) Entradas: lista de objetos gráficos (struct: `drawList{}`).

b) Salidas:

- listas de dibujo para el driver de pantalla (struct: `drawPacket{x1,x2,y1,y2,bitmap}`).
- envío de datos finalizado correctamente/fallido (bool o enum).

14. `AudioPlayer`

a) Entradas: datos de audio (struct: `audioCue{id, duration}`).

b) Salidas:

- audio para el driver (struct: `{*pcm, durtionMs}`).
- envío de datos finalizado correctamente/fallido (bool o enum).

envío de datos finalizado correctamente/fallido (bool o enum).

15. `HapticEngine`

a) Entradas: datos de vibración (enum: `hapticCue: SEND_PATTERN_1, SEND_PATTERN_2`).

b) Salidas:

- id del patrón de vibración a reproducir por el driver de vibración.
- envío de datos finalizado correctamente/fallido (bool o enum).

16. `SysMng`

a) Entradas: eventos de transición de estado del sistema (enum).

b) Salidas:

- eventos de transición del juego (enum).
- señales de cambio de estado confirmado para renderizado de menús (struct).

17. `LogSink`

a) Recepción de comandos:

1) Entradas: comando recibido por UART.
2) Salidas: señal de traza activada/desactivada (bool o enum).

b) Reenvío de logs de depuración:

1) Entradas: eventos del `EventDispatcher`.
2) Salidas: eventos en buffer de datos a enviar por UART.

18. `InputHandler`
    a) Entradas: eventos normalizados de entradas (enum, struct).
    b) Salidas:
        - acciones de gameplay (enum).
        - señal de detección de pausa de juego.
19. `RenderEngine`
    a) Entradas:
        - modelo gráfico de juego (struct: `gameModel{}`).
        - lista de hechos relevantes del juego (array: `gameSignal[]`).
    b) Salidas:
        - datos para reproducción de audio (struct: `audioCue{audio_id, duration}`).
        - datos para generación de vibración (enum: `hapticCue: SEND_PATTERN_1, SEND_PATTERN_2`).
        - lista de objetos gráficos para el motor de gráficos (struct: `drawList{}`).
20. `GameMng`
    a) Actualización del estado del juego:
        1) Entradas: eventos de transición del estado del juego (enum).
        2) Salidas: no requiere.
    b) Procesamiento de acciones:
        1) Entradas: acciones de gameplay (enum).
        2) Salidas:
            - lista de hechos relevantes del juego (array: `gameSignal[]`).
            - modelo gráfico de juego (struct: `gameModel{}`).

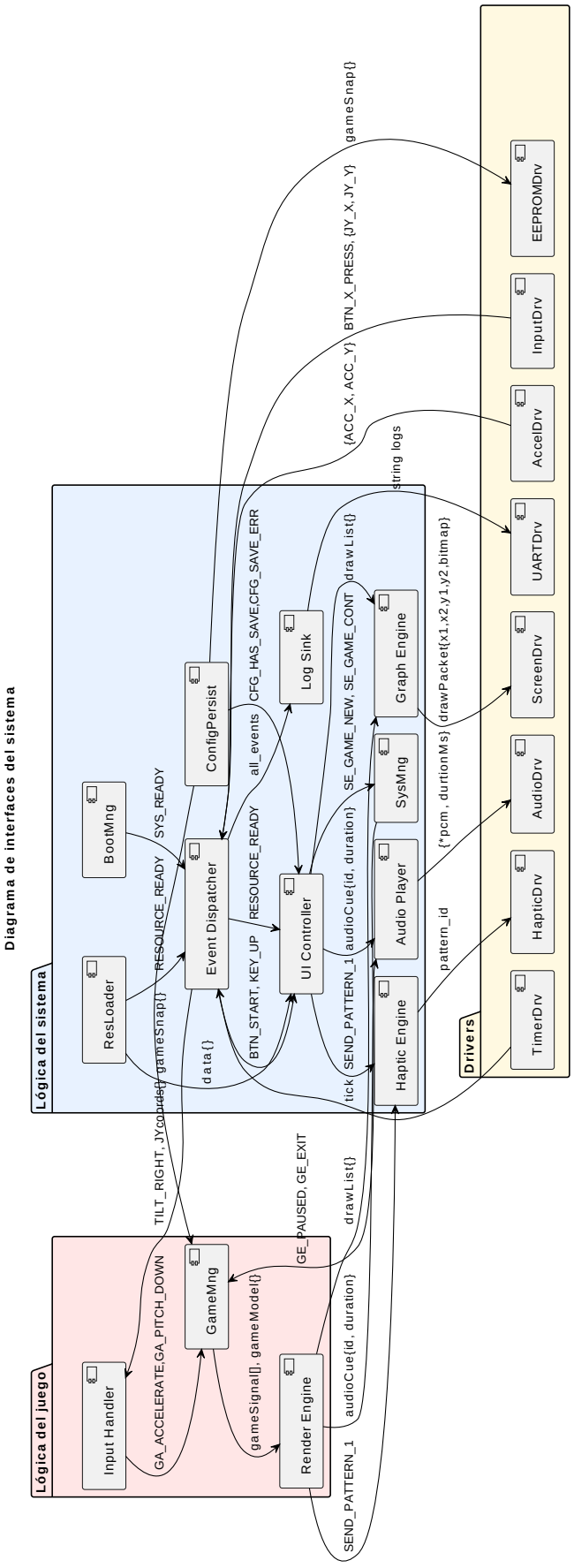
Diagrama de interfaces del sistema
Lógica del sistema
Lógica del juego
Drivers
BootMng
ResLoader
Event Dispatcher
ConfigPersist
Log Sink
UI Controller
Graph Engine
SysMng
Audio Player
Haptic Engine
Input Handler
GameMng
Render Engine
TimerDrv
HapticDrv
AudioDrv
ScreenDrv
UARTDrv
AccelDrv
InputDrv
EEPROMDrv
SYS_READY
RESOURCE_READY
RESOURCE_READY
gameSnap{}
data{}
all_events
CFG_HAS_SAVE,CFG_SAVE_ERR
BTN_START, KEY_UP
RESOURCE_READY
audioCue{id, duration}
SE_GAME_NEW, SE_GAME_CONT
drawList{}
SEND_PATTERN_1
tick
pattern_id
{*pcm, durtionMs}
drawPacket{x1,x2,y1,y2,bitmap}
string logs
{ACC_X, ACC_Y}
BTN_X_PRESS, {JY_X, JY_Y}
gameSnap{}
TILT_RIGHT, JYcopied
GA_ACCELERATE,GA_PITCH_DOWN
gameSignal[], gameMode{}
GE_PAUSED, GE_EXIT
drawList{}
audioCue{id, duration}
SEND_PATTERN_1