Videojuego portátil inspirado en consolas retro

Documento de diseño de software (SDD)

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

09/07/2025
Versión A

Basado en el estándar IEEE 1016-2017.

Historial de cambios

Versión	Fecha	Descripción	Autor	Revisores
A	09/07/2025	Creación del documento	Lic. Jezabel Danon	

Índice

1. Introducción

1.1. Propósito

1. Este documento describe el diseño del 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

1. El sistema está compuesto por un dispositivo embebido portátil que integra entradas físicas (botones, joystick, acelerómetro), salidas audiovisuales (pantalla, parlante, vibración) y una unidad de procesamiento encargada de coordinar la ejecución de un demo de juego simple.

2. El presente diseño de software abarca los módulos encargados de:

   - Control de entrada/salida digital y analógica.
   - Gestión del flujo del juego.
   - Renderizado gráfico en pantalla.
   - Generación de audio y control de vibración.
   - Manejo de estados internos y lógica de juego.

3. Este documento se enfoca exclusivamente en la descripción estructural y funcional del software embebido. No se incluyen aspectos de diseño físico, elección final de componentes electrónicos ni pruebas de verificación, los cuales son abordados en documentos específicos (como las especificaciones de hardware).

4. El diseño cubre las capas de abstracción de hardware (drivers), lógica de control del juego, gestión de recursos gráficos y sonoros, así como el manejo de entradas y salidas del sistema.

5. Quedan fuera de este documento los detalles de bajo nivel sobre protocolos específicos de comunicación (como I^2C o SPI), que se encuentran documentados en los manuales de componentes o en la especificación de hardware.

6. Tampoco se incluyen aspectos de programación de la interfaz de usuario en cuanto a estilo o experiencia visual final, los cuales podrán ser definidos en una etapa posterior del desarrollo.

1.3. Definiciones y acrónimos

1. IEEE: Instituto de Ingenieros Eléctricos y Electrónicos.

2. ERS: Especificación de Requisitos de Software.

3. RTOS: Sistema Operativo de Tiempo Real (Real Time Operating System).

4. CMSIS: Arm's Common Microcontroller Software Interface Standard. Se compone por un conjunto de APIS, componentes de software, herramientas y flujos de trabajo provisto por el desarrollador de la arquitectura del microcontrolador a utilizar.

5. HAL: Hardware Abstraction Layer. Conjunto de bibliotecas de software provistas por STMicroelectronics para simplificar la interacción con el hardware de microcontroladores STM32.

6. UI: Interfaz de usuario (User Interface).

7. UART: Universal Asynchronous Receiver/Transmitter.
8. I2C: Inter-Integrated Circuit.
9. SPI: Serial Peripheral Interface.
10. ADC: Analogic to Digital Converter.
11. PWM: Pulse Width Modulation.
12. GPIO: General-purpose Input/Output.
13. ST-Link: programador y depurador integrado en placas STM32.
14. 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. Documento de arquitectura: RETRO_GAME-AR-vA.

2. Resumen de la arquitectura

1. El sistema software para el sistema embebido *Videojuego portátil inspirado en consolas retro* se basa en una arquitectura modular y estratificada, descrita en profundidad en el documento de arquitectura *RETRO_GAME-AR*.
2. El software se organiza en tres capas principales:
   - **Capa de drivers**: incluye los controladores de bajo nivel responsables de inicializar y acceder a los periféricos físicos (pantalla TFT por SPI, audio por PWM, EEPROM por I^2C, entradas físicas por GPIO/ADC y temporizadores).
   - **Lógica del sistema**: contiene los servicios de propósito general como el gestor de estados del sistema, el manejador de eventos, el cargador de recursos gráficos y sonoros, el subsistema de persistencia y el motor de interfaz de usuario. Esta capa actúa como intermediaria entre los drivers y la lógica del juego, manteniendo la consistencia y coordinando flujos.
   - **Lógica del juego**: implementa el comportamiento específico del gameplay. Incluye su propia máquina de estados, el motor de renderizado de escenas y la lógica que interpreta eventos de entrada como acciones dentro del juego.
3. El sistema corre sobre un RTOS de tipo cooperativo con planificación por prioridad fija. Las tareas concurrentes están organizadas por funcionalidad (drivers, UI, lógica del juego) y se comunican mediante colas, semáforos y señales. La arquitectura favorece un desacoplamiento fuerte entre capas, facilitando el testeo modular y la reutilización de servicios comunes.
4. Este documento de diseño detallado expande la descripción de cada uno de los módulos presentados en esta arquitectura, definiendo su comportamiento interno, sus estructuras de datos y sus interfaces.

3. Diseño detallado de módulos y servicios

A continuación se presenta el listado de componentes por capa junto a una breve descripción de sus responsabilidades.

Capa	Servicio	Nombre	Responsabilidad principal
Drivers	`AccelDrv`	Driver de acelerómetro	Lectura por I^2C
	`EEPROMDrv`	Driver de memoria EEPROM	Lectura y escritura por SPI
	`ScreenDrv`	Driver de pantalla	Control ST7735, ventana activa, transferencia de píxeles por SPI/DMA
	`AudioDrv`	Driver de audio	Reproducción de efectos de sonido mediante modulación por ancho de pulso (PWM)
	`HapticDrv`	Driver de vibración	Comunicación con DRV2605L por I^2C, ejecución de patrones
	`UARTDrv`	Driver de UART	Comunicación USART2 (DMA); base para consola de debug
	`InputDrv`	Driver de entradas	Lectura de botones (EXTI) y joystick (ADC DMA), empaqueta eventos crudos
	`TimerDrv`	Driver de timers	Generación de interrupciones de tiempo
Lógica del sistema	`BootMgr`	Gestor de arranque	Inicializa drivers, crea tareas FreeRTOS, transfiere a menú principal
	`ResLoader`	Cargador de recursos	Carga de recursos (imágenes, sonidos) en RAM desde memoria externa
	`ConfigPersist`	Configuración y persistencia	Guarda/carga snapshot de juego; verificación de partida guardada
	`EventDispatcher`	Despachador de eventos	Normaliza entradas y publica eventos a tareas superiores; redistribución de mensajes entre componentes
	`UIController`	Gestor de interfaz de usuario	Gestión de la UI en estados no interactivos del juego
	`GraphEngine`	Motor gráfico	Renderizado gráfico, gestión de buffers de pantalla
	`AudioPlayer`	Reproductor de sonido	Mezcla y reproduce música o efectos
	`HapticEngine`	Motor háptico	Gestión de la activación de patrones de vibración
	`LogSink`	Debug y logs	Gestión de comunicaciones por UART y trazas de depuración
	`SysMng`	Gestión de estados del sistema	Implementación de máquina de estados principal del sistema, gestión de transiciones (`SYS_SPLASH`, `SYS_MAIN_MENU`, `SYS_IN_GAME`, `SYS_PAUSED`)
Lógica del juego	`GameMng`	Gestión de estado del juego	Control del bucle de juego, construcción del modelo gráfico; implementación de los estados internos del juego (`GME_RUNNING`, `GME_FROZEN`, `GME_STOPPED`)

Capa	Servicio	Nombre	Responsabilidad principal
	`RenderEngine`	Motor de renderizado	Generación del contenido visual del juego para cada cuadro; coordinación con servicios de generación de salidas
	`InputHandler`	Gestión de entradas	Traducción de eventos normalizados a acciones de gameplay

3.1. Drivers

3.1.1. `AccelDrv` — Driver de acelerómetro

Resumen: Controla la comunicación con el sensor MPU-6500 a través del bus I^2C. Permite obtener valores de aceleración crudos en los tres ejes para su posterior uso en lógica de juego o UI.

Notas adicionales:

- Se debe configurar la dirección de I2C y registrar la interrupción de "data ready".
- Solo se utilizará el acelerómetro (y solo 2 ejes); se puede deshabilitar el giroscopio para ahorrar consumo.
- Requiere setup inicial del registro PWR_MGMT y escala de sensibilidad.
- Retorna: datos x DMA (ACC_X, ACC_Y) evento para EventDispatcher (DRV_OK, DRV_ERR)

3.1.2. `EEPROMDrv` — Driver de memoria EEPROM

Resumen: Brinda una interfaz para operaciones de lectura y escritura en la memoria EEPROM SPI 25LC256, utilizada para snapshots y configuraciones.

Notas adicionales:

- Debe controlar el pin `CS` manualmente.
- Implementar verificación de página y latencia de escritura.
- Usar checksum CRC para validar snapshots.
- Recibe: [data,] direccion
- Retorna [data,] ok/fail, evento para EventDispatcher (DRV_OK, DRV_ERR, info de guardado y carga)

3.1.3. `ScreenDrv` — Driver de pantalla

Resumen: Controla la pantalla TFT de 1.8çon controlador ST7735R mediante SPI. Permite enviar bloques de píxeles desde un búfer de imagen RAM.

Notas adicionales:

- Usa DMA para transferencias rápidas de píxeles.
- Es mayormente el driver TFT_ST7735 ya desarrollado.
- Considerar la implementación de "ventana activa"para actualizar solo regiones.
- Retorna: evento para EventDispatcher (DRV_OK, DRV_ERR, alguna info de renderizado o flush??)

3.1.4. `AudioDrv` — Driver de audio

Resumen: Genera salida de audio mediante modulación PWM y amplificador externo (LM386). Se alimenta con datos desde el `AudioPlayer`.

Notas adicionales:

- Usar PWM de alta frecuencia (al menos 20 kHz).
- Controlar el volumen mediante ajuste de duty o por hardware.
- Entradas: frecuencia fija o muestras PCM, duración ({*pcm, durtionMs})
- Retorna evento para EventDispatcher (DRV_OK, DRV_ERR, info de envio de datos)

3.1.5. `HapticDrv` — Driver de vibración

Resumen: Controla el módulo DRV2605L por I^2C para generar patrones de vibración con motor ERM.

Notas adicionales:

- El DRV2605L tiene biblioteca interna de patrones ROM.
- Requiere escritura de patrón y activación del disparo.
- Considerar cola de vibraciones para evitar pérdidas.
- Entradas: id del patrón de vibración a reproducir.
- Retorna evento para EventDispatcher (DRV_OK, DRV_ERR, info de envio de datos)

3.1.6. `UARTDrv` — Driver de UART

Resumen: Maneja la comunicación serie con la PC a través de USART2, principalmente para logging y comandos de consola.

Notas adicionales:

- Configurar USART2 con DMA para recepción continua.
- Reconocimiento de comando de on/off.
- Se conecta vía ST-Link como puerto virtual.
- Retorna eventos para EventDispatcher (DRV_OK, DRV_ERR, CMD_LOGON, CMD_LOGOFF)

3.1.7. `InputDrv` — Driver de entradas

Resumen: Lee botones físicos mediante interrupciones (GPIO/EXTI) y joystick analógico mediante ADC. Entrega eventos crudos al despachador.

Notas adicionales:

- Debounce por software.
- ADC con conversión continua + DMA para eje X/Y (??)
- Agrupar estado en estructura con bitmask de botones + valores analógicos.
- Retorna eventos para EventDispatcher (DRV_OK, DRV_ERR, BTN_X_PRESS, JY_X, JY_Y)

3.1.8. `TimerDrv` — Driver de temporizadores

Resumen: Proporciona temporización periódica (ej. tick de frame) mediante timers de hardware. Puede generar interrupciones o callbacks.

Notas adicionales:

- Base para lógica temporal en el juego y animaciones.
- Se usará probablemente TIM2 o TIM3.
- Considerar usar HAL Timer + callbacks con prioridad baja.

3.2. Componentes de sistema

3.2.1. `BootMgr` — Gestor de arranque

Resumen: Se ejecuta al inicio del sistema tras el bootloader. Se encarga de inicializar los servicios básicos, crear las tareas de FreeRTOS y mostrar la pantalla inicial (splash).

Notas adicionales:

- Debería inicializar los drivers y gestionar que todos funciones OK.
- Llamar a ConfigPersist para que revise partida guardada para renderizar el menú ppal.
- Crea colas, inicializa módulos y transfiere el control a `SysMng` (`SYS_READY`).
- Puede verificar integridad del sistema o mostrar fallos iniciales.

3.2.2. `ResLoader` — Cargador de recursos

Resumen: Permite leer recursos binarios como sprites, fuentes o jingles desde EEPROM externa o memoria Flash, y ponerlos en RAM para uso por los motores gráficos o de audio.

Notas adicionales:

- Solo lectura. Desacoplado del guardado.
- Puede usar índice predefinido o direcciones fijas:
  - mapa de la memoria EEPROM, secciones predefinidas para assets, profile/configs, juego
  - tabla de id-location para assets predefinidos/precargados en memoria
- Debe notificar cuando el recurso esté listo vía evento.
- Entradas: id del asset a cargar (unsigned int).
- Retorna eventos para EventDispatcher (RES_READY) y asset serializado

3.2.3. `ConfigPersist` — Persistencia y configuración

Resumen: Gestiona el guardado y restauración del estado del juego (snapshot) y configuraciones persistentes. Se comunica con `EEPROMDrv`.

Notas adicionales:

- Es responsable del mapeo de direcciones y formato de snapshot.
- Requiere validación con checksum/CRC.
- Debería poder distinguir entre guardado válido y vacío.
- Mapa de la memoria EEPROM, secciones predefinidas para assets, profile/configs, juego
- Regiones de memoria para los snapshots y flag HAS_SAVE .
- Generacion del campo flag HAS_SAVE y guardado en region config (acceso mas simple que el snap).
- Retorno de eventos al EventDispatcher (CFG_SAVE_DONE, CFG_SAVE_ERR, CFG_HAS_SAVE)

3.2.4. `EventDispatcher` — Despachador de eventos

Resumen: Recibe eventos crudos desde `InputDrv`, los normaliza (antirrebote, umbral, zona muerta) y los distribuye a tareas lógicas superiores mediante colas o flags.

Notas adicionales:

- Es el único punto entre entradas físicas y lógica del sistema/juego.
- Convierte joystick a ejes discretos (ej. izquierda, centro, derecha) y boton X a BTN_START.
- Procesamiento de inclinación (si es necesario)
- Guarda la bandera traceEnabled. Si true, duplica cada mensaje que rutea hacia queueLogSink.
- Todos los eventos deben pasar por aca para redistribuir a log.
- Tabla origen/evento –>destinatario

3.2.5. `UIController` — Interfaz de usuario

Resumen: Controla pantallas como menú principal, pausa, confirmaciones o mensajes fuera del gameplay. Interpreta entradas y publica acciones internas.

Notas adicionales:

- Dibuja menus con LVGL.
  - Si pongo los lv_port en GraphEngine, entonces UIController estaria tecnicamente enviado la data al GraphEngine
  - Envia lista de objetos gráficos (struct: drawList{})) a GraphEngine (mas o meno lo hace la libreria LVGL)
- Integración con sonido y vibración (??)
  - datos para reproducción de audio (audioCue {id, duration}).
  - datos para generación de vibración (hapticCue enum: SEND_PATTERN_1, SEND_PATTERN_2).

3.2.6. `GraphEngine` — Motor gráfico

Resumen: Proporciona primitivas 2D para dibujar en pantalla: sprites, textos, rectángulos. Coordina el doble búfer y compone la display-list para `ScreenDrv`.

Notas adicionales:

- Seria un wrapper de las cosas que voy a usar de LVGL
- Si pongo los lv_port en GraphEngine, entonces UIController estaria tecnicamente enviado la data al GraphEngine
- Aca hay que armar todas las primitivas que podrian usar LVGL por abajo para renderizar los graficos del juego a partir de la RenderList o drawList (Objetos lógicos: layer, spriteID, posX, posY)
- Ideal tener una cola o display-list reutilizable.
- Ofrece API común a UI y juego.
- Prioridad baja en renderizado; puede reducir FPS en pausa.

3.2.7. `AudioPlayer` — Motor de audio

Resumen: Mezcla efectos de sonido y música, y los entrega a `AudioDrv`. Permite silenciar, repetir, interrumpir o atenuar sonidos.

Notas adicionales:

- Controla el buffer circular y su tasa.
- Soporta volúmenes diferenciados.
- Puede tener canal reservado para jingles o música de fondo.

3.2.8. `HapticEngine` — Motor háptico

Resumen: Programa el driver DRV2605L con patrones de vibración en respuesta a eventos del sistema o juego.

Notas adicionales:

- Puede implementar una cola de eventos con prioridad.
- Compatible con patrones integrados del DRV2605L.
- Deshabilitado automáticamente durante muting o en pausa.

3.2.9. `LogSink` — Debug y logs

Resumen: Canaliza mensajes del sistema o juego hacia la UART, para ser interpretados por consola o herramientas externas.

Notas adicionales:

- Usa `UARTDrv` como backend.
- Incluye flag para activar/desactivar salida.
- Posibilidad de implementar comandos de texto para debug.

3.2.10. `SysMng` — Gestión de estados del sistema

Resumen: Implementa la FSM principal del sistema embebido. Administra transiciones globales como encendido, menú principal, pausa y juego activo.

Notas adicionales:

- Encapsula los modos `SYS_SPLASH`, `SYS_MENU`, `SYS_IN_GAME`, `SYS_PAUSED`.
- Activa/desactiva módulos según contexto.
- Dispara los estados de juego al iniciar o salir.

3.3. Componentes del juego

3.3.1. `GameMng` — Gestor de estado del juego

Resumen: Controla el bucle principal del gameplay. Define y administra los estados internos del juego como ejecución activa, pausa y finalización.

Notas adicionales:

- Implementa una FSM propia independiente de `SysMng`.
- Responsable de invocar a `UpdateEngine`, `RenderEngine` y servicios de salida.
- Emite eventos como `PLAY_SFX`, `VIBRATE_HIT`, `EXIT_TO_MENU`.
- Debe poder congelar el juego sin perder el estado (ej. en pausa o guardado).
- Genera gameModel que encapsule subt-structs: Player, World
  - Contiene todo lo que hace falta — cada frame — para actualizar la lógica y dibujar la escena
  - Lo que no puede calcularse del gameModel a partir de otras variables, es lo que se va a guardar en el snapshot
  -
- Genera gameSignal[]: vector / cola de "hechos relevantes"

3.3.2. `RenderEngine` — Motor de renderizado del juego

Resumen: Construye por cada cuadro la lista de elementos visuales (sprites, textos) que serán enviados al motor gráfico. Define qué se ve en pantalla en función del estado del juego.

Notas adicionales:

- Responsable de composición visual: fondo, entidades, HUD.
- Define capas de render (orden de dibujo).
- Coordina con `GraphEngine` para emitir comandos gráficos.
- Puede incluir un modo debug visual.
- transformacion gameSignal: Tabla de mapeo:
  - SIG_STALL_WARN → audio=BEEP1, haptic=TICK
  - SIG_WP_REACHED → audio=CHIME, haptic=BUZZ

3.3.3. `InputHandler` — Procesador de entradas del juego

Resumen: Recibe eventos de entrada desde `EventDispatcher` y los traduce a acciones dentro del juego, como mover la nave o disparar.

Notas adicionales:

- Ejecuta lógica específica de controles (reglas por juego).
- Filtra eventos duplicados o irrelevantes según contexto.
- Idealmente desacoplado del hardware y del tipo exacto de entrada.
- Puede incluir combinaciones o combos (ej. doble click).

4. Diagramas estáticos relevantes

4.1. Relaciones entre módulos

4.2. Organización de tareas en FreeRTOS

5. Modelos dinámicos

5.1. Statecharts

5.2. Diagrama de secuencia 1 - Encendido hasta menú

Descripción narrativa de cada paso.

5.3. Diagrama de secuencia 2 - Pausa y opciones

6. Aspectos de sincronización y concurrencia

6.1. Mapa de tareas FreeRTOS

Tabla con tareas, prioridad, stack, frecuencia.

6.2. Sincronización y comunicación

Colas, semáforos, exclusiones mutuas usadas.

7. Modelo de datos

7.1. Estructuras persistentes

Formato del snapshot de partida, layout EEPROM, checksum.

7.2. Estructuras en RAM

Buffers de audio, frame buffer, colas de eventos.

7.3. Datos en EEPROM

7.4. Buffers y estructuras temporales en RAM

8. Manejo de errores y logs

Política de códigos de error, niveles de log, rutas de fallos críticos.

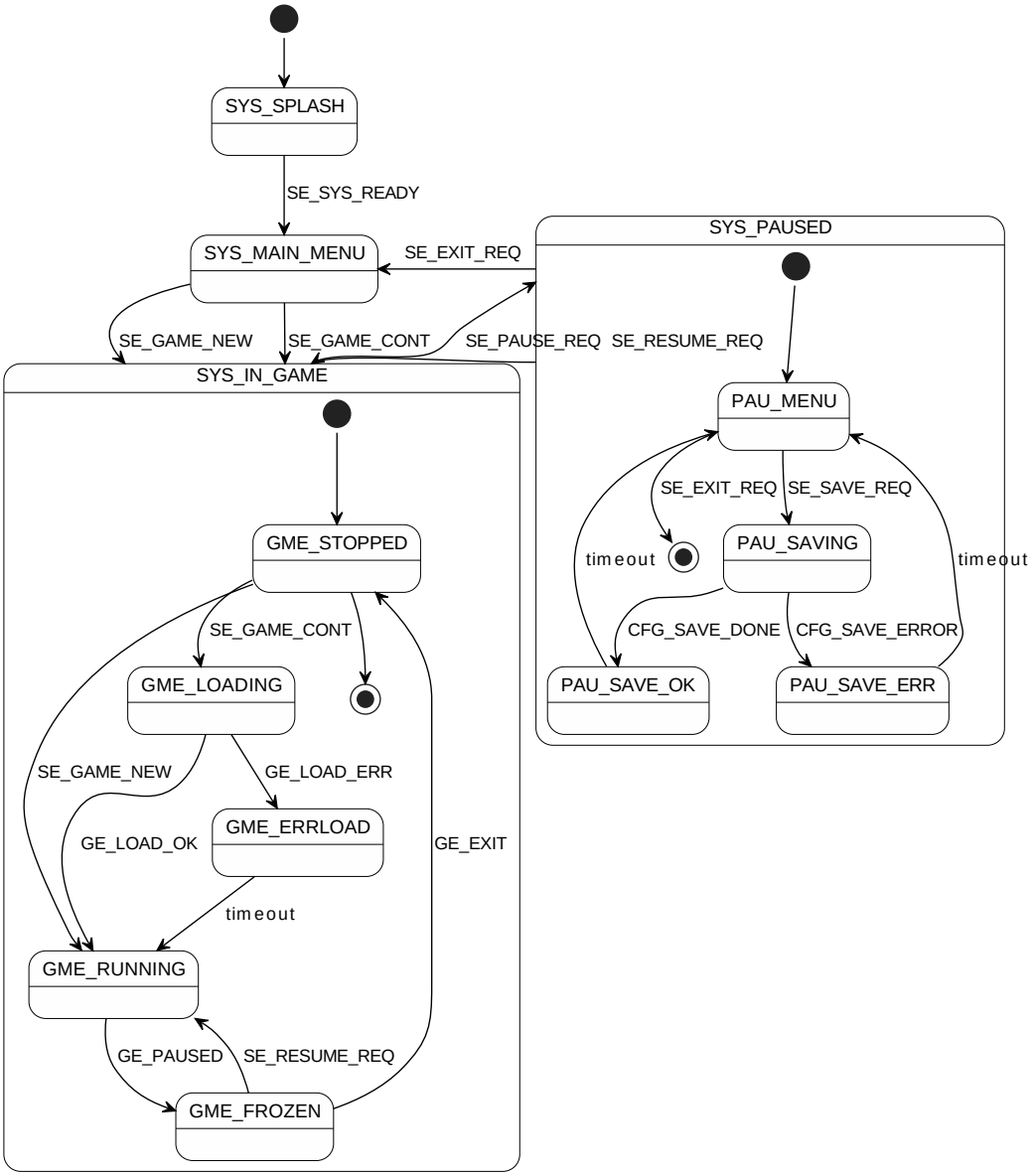

Figura 1: Diagrama de estados del sistema.

8.1. códigos de error definidos

8.2. Ruta de logs por UART

A. Apéndice A - Glosario completo de servicios

Tabla resumen: Nombre, capa, descripción de 1 línea.

B. Apéndice B - Diagramas adicionales

Coloca aquí otros diagramas UML o listas de mensajes.