Proyecto Final Fundamentos de Sistemas Embebidos: Consola de videojuegos retro

Molina Véjar Aarón Gael

28 de noviembre del 2025

1. Introducción

La implementación de un sistema embebido abarca desde la definición de un problema que se busca resolver y el planteamiento de una solución basada en las distintas técnicas y conocimientos científicos que abarcan los sistemas embebidos. Es importante el planteamiento inicial para así poder abordar desde un lugar mas robusto la resolución a los problemas, en este caso la finalidad no es sino una consola de entretenimiento para un usuario final que solo busca distraerse por un rato sin necesidad de dificultades técnicas, simplemente conectar y jugar (plug and play).

En este reporte se abordará desde los antecedentes que nos permiten encontrar una solución lo suficientemente óptima para cumplir con el objetivo final, el desarrollo que permitió la implementación abarcando a grandes rasgos los conceptos técnicos mas importantes, para finalizar con una conclusión respecto a los resultados obtenidos con esta implementación.

2. Antecedentes

2.1. Interfaz gráfica

2.1.1. Arquitectura de software

Se optó por usar de base un sistema operativo sin interfaz gráfica como lo es Raspberry Pi OS Lite, esto para aprovechar de manera eficiente los recursos gráficos y conseguir la adaptación a consola de videojuegos. Se garantiza de esta manera que el usuario tenga la experiencia completa de una consola de videojuegos sin que se confunda con una computadora de propósito general. Como señalan Barr y Massa, en un entorno restringido, el desarrollador debe gestionar la memoria y el procesamiento para garantizar que el sistema opere según las especificaciones. Al operar en modo kiosco, eliminamos la latencia introducida por la inclusión de escritorio.[1]

Se seleccionó la librería SDL2 como middleware para el manejo de gráficos, audio y entradas en lugar de toolkits como Qt o GTK, o librerías de bajo nivel como OpenGL puro. SDL proporciona una capa de abstracción delgada sobre el hardware de video y audio esto permite trabajar la interfaz en C++ mientras la librería gestiona las diferencias entre los drivers. A diferencia de frameworks orientados a widgets, SDL2 utiliza framebuffers lo cual lo convierte en pieza ideal para una interfaz de consola de videojuegos.

SDL2 incluye una base de datos comunitaria de mapeo de controles “gamecontrollerdb.txt”, lo que simplificó drásticamente la implementación de controles para la construcción total de la consola.

2.1.2. Controlador de video KMS/DRM

Se configuró la aplicación para utilizar el controlador de video KMSDRM mediante la variable de entorno SDL_VIDEODRIVER= kmsdr, ubicada en el script launch.sh, evitando de esta manera elm uso de drivers x11.

La capa DRM proporciona varios servicios a los controladores gráficos, muchos de ellos impulsados por las interfaces de aplicación que proporciona a través de libDRM, la biblioteca que envuelve la mayoría de los ioctls DRM. Entre ellas se encuentra vblank gestión de eventos, gestión de memoria, gestión de salida, framebuffer gestión, envío de mandos y esgrima, apoyo de suspensión/reanudación, y Servicios DMA.[2]

2.1.3. Diseño de navegación

El subsistema de entrada se diseñó utilizando un mecanismo de sondeo (polling) sincronizado con el ciclo de refresco de la pantalla. Aunque las interrupciones son más eficientes para eventos esporádicos, el sondeo es preferible cuando se requiere una respuesta continua y de baja latencia en interfaces de usuario.[1] Se implementó una clase InpuManager que abstrae los eventos físicos en acciones lógicas. Esta arquitectura permite separar la lógica de la interfaz al menú de la implementación del hardware de entrada. Esto permite que el sistema fuera compatible a la perfección con un gamepad sin duplicar la lógica de navegación en cada pantalla.

2.1.4. Gestión de ciclo de vida

La interfaz gráfica no ejecuta los emuladores como hijos dentro de su propio proceso, en su lugar escribe un comando en un archivo temporal y se cierra, cediendo el control al script maestro launcher.sh.

La arquitectura del sistema se basa en un bucle infinito de gestión de ciclo de vida, un patrón estándar en el desarrollo de firmware descrito por Barr [1]. El uso de un script maestro (launcher.sh) para orquestar la ejecución secuencial de la interfaz y el emulador garantiza que cada proceso tenga acceso exclusivo a los recursos críticos del sistema.

2.1.5. Estrategia de unificación Mednafen

Se seleccionó Mednafen para trabajar como motor de emulación unificado para tres plataformas distintas: NES, SNES Y GBA, descartando el uso de emuladores independientes para cada sistema. Esta decisión fue crucial para una arquitectura limpia y eficiente con una alto rendimiento y facilidad de integración,

Mednafen es un emulador multisistema portátil, que utiliza OpenGL y SDL, basado en argumentos (líneas de comandos). Mednafen tiene la capacidad de reasignar funciones de teclas rápidas y entradas del sistema virtual a un teclado, un joystick o ambos simultáneamente. Se admiten estados de guardado, al igual que el rebobinado en tiempo real. Las instantáneas de pantalla pueden tomarse, en formato PNG, con solo pulsar un botón. Mednafen puede grabar películas audiovisuales en formato de archivo QuickTime, con varios códecs sin pérdida compatibles.[3]

En comparación a la implementación de tres emuladores distintos para la emulación de cada plataforma de manera separada, la implementación de un binario multi-sistema reduce la complejidad del mantenimiento y gestión y dependencias, en lugar de gestionar tres configuraciones, tres mapas de controles y tres sets de dependencias, se unifican en administrar un archivo de configuración: mednafen.cfg Mednafen está diseñado para operar nativamente sobre SDL y OpenGL lo cual lo hace ideal para implementar en entornos sin ventanas gráficas, permitiendo la ejecución directa sobre el framebuffer o KMS.

2.2. Gestión de recursos de video: transición de contexto

Se realizó una implementación secuencial en lugar de concurrente. La interfaz no lanza el emulador como un subproceso, sino que delega la ejecución al script maestro launcher.sh y termina su propia ejecución antes de iniciar el emulador.

En sistemas embebidos Linux que utilizan KMS/DRM para gráficos sin escritorio, el proceso activo se convierte en el DRM-Master[2]. Si la interfaz gráfica ocupara todo el tiempo dicho recurso, a los emuladores se les sería negado utilizar el control de la salida de video, resultando en conflictos de acceso. Al cerrar la interfaz gráfica antes de lanzar el juego se libera la totalidad de memoria RAM consumida por las texturas y recursos del menú, optimizando en gran medida los tiempos de latencia y el rendimiento durante la emulación.

En la figura 9 se puede observar parte de la implementación secuencial, pues muestra como se escribe el comando en un archivo temporal y se cierra la interfaz exit(0) para liberar el framebuffer (KMSDRM) antes de que inicie el juego.

2.2.1. Integración de firmware propietario

Para la emulación de GBA fue necesario realizar una integración manual del archivo BIOS original en el directorio de configuración de Mednafen. Aunque existen emuladores de alto nivel que simulan la BIOS, el núcleo de Mednafen prioriza la precisión del ciclo[3] y requiere la BIOS original para inicializar correctamente el entorno virtualizado de la consola original. Esto soluciona diversos problemas como crasheos y utilidad en diversas ROMS. Se realiza una copia del binario propietario al directorio /home/pi/.mednafen/, para esto también es necesario otorgarle permisos de lectura.

2.2.2. Abstracción de entrada para emuladores

Se configuraron los controles con ayuda del motor de emulación mediante mapeo físico, independientemente de la capa de abstracción de la interfaz gráfica. La implementación de lectura de eventos del dispositivo por el emulador mediante SDL evita overhead al pasar señales a través de capas intermedias de software. Esto es importante para mantener una respuesta de control o input las baja. Se aprovechan las funciones exclusivas del emulador como guardar estado o salir.

2.3. Configuración del sistema operativo y arranque automático

2.3.1. Transformación a consola de videojuegos

Se realizó una configuración con un objetivo claro: transformar la raspberry pi en una consola de videojuegos, por lo que debía de operar en modo "kiosko", eliminando el acceso a la terminal de comandos y un inicio de sesión de usuario tradicional. Se realizaron los cambios correspondientes por un servicio de sistema personalizado retro-console.service gestionado por systemd.[4][5] En un dispositivo de consumo el usuario final no debe interactuar con el sistema operativo base. Al utilizar systemd como gestor de procesos, se garantiza que la aplicación principal se reinicie automáticamente en caso de fallos, cumpliendo con principios de alta disponibilidad en sistemas embebidos. Se configura el servicio Type=simple y eliminamos dependencias de red bloqueantes como NetworkManager-wait-online para reducir considerablemente el tiempo de inicio desde el encendido hasta la interfaz de usuario, priorizando la experiencia inmediata sobre servicios secundarios.

2.3.2. Implementación de arranque silencioso

Se modificaron parámetros del kernel de Linux y el firmware de la GPU para suprimir toda salida de texto y logotipos del fabricante durante la secuencia de arranque. Esto es importante debido a la experiencia de usuario y a que este mismo no espera encontrarse con cosas "raras" al momento de jugar a su consola de videojuegos favorita, por lo que es fundamental mantener un inicio limpio.

Según Barry, el proceso de arranque en plataformas embebidas debe ser optimizado para iniciar la aplicación principal lo más rápido posible. Mediante la configuración de servicios de systemd y la personalización de los parámetros del kernel, se logró ocultar la complejidad del sistema operativo al usuario final, presentando una interfaz coherente desde el encendido, alineándose con las expectativas de los dispositivos electrónicos de consumo modernos.[1][6]

Se redirigió la consola del sistema a una terminal virtual oculta con console=tty3 y se desactivaron los mensajes de estado del gestor de servicios systemd.show_stattus=false, asegurando que el framebuffer se mantenga limpio para la animación de carga. Gracias a nuestro servicio de arranque maestro systemd en el archivo /etc/systemd/system/retro-console.service, podemos manejar de manera eficiente el arranque mediante la secuencia exacta:

- Mantiene la imagen de carga por 10 segundos

- Cierra la animación
- Lanza el script de la consola launcher.sh
- Silencia cualquier salida de texto residual

2.3.3. Personalización gráfica y transiciones

La implementación de Plymouth fue necesaria para hacer uso de ella como multiplexor de arranque para gestionar el framebuffer durante la carga del kernel. Se desarrolló un tema personalizado "BrigadaMMP" y se configuró una transición temporizada en el servicio de arranque. La transición entre el arranque y la aplicación de usuario suele generar parpadeos o pantallazos negros debido al cambio de modos de video, Plymouth mantiene la imagen en memoria hasta que la aplicación gráfica está lista.

Se implementó una pausa artificial dentro para poder observar el diseño de experiencia programado para el usuario, no obedece a un requerimiento de tiempo técnico ocupado por el sistema debido a que este inicia antes, sin embargo, es necesario mantener unos segundos extra en pantalla para que el usuario pueda visualizar la marca del producto antes de ingresar al menú, además evitando transiciones abruptas o extrañas.

Se implementa en el script maestro launcher.sh la técnica de "Zeoring", una técnica de manipulación directa del framebuffer, esto para conseguir que la pantalla se vea negra. En sistemas Linux embebidos, el kernel expone la memoria como un dispositivo de caracteres en /dev/fb0[9], en el script maestro vuelca un flujo continuo de bytes nulos sobre el dispositivo de video, a nivel de hardware esto sobre escribe cada pixel en la pantalla con el color negro, eliminando instantáneamente cualquier texto residual que pudiera permanecer en la memoria. Entre el cierre del menú y la apertura del emulador existe un milisegundo donde la memoria puede contener basura o mostrar durante unos instantes la terminal de texto. Esta técnica fuerza un borrado a negro instantáneo asegurando una transición visual agradable y profesional entre aplicaciones.[7]

3. Desarrollo

Para dar flexibilidad a cada explicación se hará uso de dos fuentes principales: setup.sh y código fuente C++. Esto debido a la practicidad para explicar modificaciones o instalaciones sobre el sistema operativo base y de funcionalidades explicitas de código necesarias de destacar. Así mismo se procura mantener la impresión de código lo más breve y conciso posible con fines explicativos, para leer el código completo y una demostración del funcionamiento en vivo véase sección 3.7

3.1. Dependencias, bibliotecas básicas y compilación

Para las dependencias básicas es necesario destacar la siguiente lista de herramientas y bibliotecas que se usaron a lo largo de la implementación, sin ellas es imposible avanzar en nuestro proyecto de la manera en la que se implementó.

El comando utilizado dentro de setup.sh es el siguiente, a continuación se muestra el cuadro 1 con las especificaciones correspondientes

3.1.1. Código o parte de la implementación

```
sudo apt-get install -y build-essential git cmake libsdl2-dev libsdl2-image-dev
libsdl2-ttf-dev libsdl2-mixer-dev libssl-dev plymouth plymouth-themes mednafen pmount
```

Figura 1: Comando completo de instalación de todas las dependencias necesarias

Cuadro 1: Bibliotecas y herramientas utilizadas en el desarrollo

Biblioteca o librería	**Descripción**
build-essential	Metapaquete que instala el compilador GNU C++ (g++), make y otras herramientas
cmake	Requerido como dependencia para la compilación de herramientas configuración del entorno de desarrollo.
libsdl2-dev	Librería de desarrollo multimedia. Provee la capa de abstracción para el acceso al hardware de video
libsdl2-image-dev	Extensión de SDL2 utilizada para la carga y decodificación de formatos de imagen
libsdl2-ttf-dev	Extensión de SDL2 motor de renderizado de fuentes tipográficas
libsdl2-mixer-dev	Extensión de SDL2 para la gestión y mezcla de audio multicanal
libssl-dev	Librería de criptografía (OpenSSL) que provee algoritmos de hashing.
plymouth	Gestor de arranque gráfico que opera en el espacio de usuario temprano
plymouth-themes	Colección de scripts y recursos gráficos base para Plymouth
mednafen	Motor de emulación multi-sistema unificado (NES, SNES, GBA)
pmount	Herramienta de administración de volúmenes que permite montar medios extraíbles (USB)

3.2. Interfaz gráfica, menús y transiciones

Para la interfaz gráfica, como se mencionó anteriormente se implementó en C++ con la biblioteca SDL2 que nos permitió implementar una interfaz fluida y aprovechando todos los recursos gráficos al máximo posible.

```
void MenuScreen::render(SDL_Renderer* renderer, TextRenderer* textRenderer) {
    SDL_SetRenderDrawColor(renderer, BG_COLOR.r, BG_COLOR.g, BG_COLOR.b, BG_COLOR.a);
    SDL_RenderClear(renderer);

    renderTopBar(renderer, textRenderer);
```

Figura 2: Fragmento de renderizado en MenuScreen.cpp

La implementación del menú principal consta de tres secciones principales: Biblioteca, Favoritos y ROMS, para estas tres secciones se implementan mecanismos diferentes de interacción, todos ellos codificados dentro de MenuScreen.cpp, de esta manera se logra un aislamiento completo en la comunicacion y navegación de la consola.

Para conseguir transiciones entre menús fue necesario realizar modificaciones en la implementación secuencial que permitiera ocultar lineas de comando que pueden provocar una mala experiencia de usuario,

```
        if (currentTab == Tab::ROMS) {
            btnAddROM->update(mx,my);
            btnDeleteROM->update(mx,my);
            btnRescan->update(mx,my);

            if (input.isMouseButtonPressed(SDL_BUTTON_LEFT)){
                if (btnAddROM->isClicked(mx,my)) btnAddROM->click();
                if (btnDeleteROM->isClicked(mx,my)) btnDeleteROM->click();
                if (btnRescan->isClicked(mx,my)) btnRescan->click();
        }
```

Figura 3: Fragmento del mecanismo de interfaz en para la sección ROM, código extraido de la función handleInput en MenuScreen.cpp

para esto consulte la sección 3.5 para una explicación detallada respecto a las configuraciones aplicadas en el SO para la navegación amigable.

3.3. Emuladores y accesibilidad

Los emualdores se encuentran unificados en el motor principal de emulación implementado: Mednafen, esta unificación funciona y es implementada de la siguiente manera:

```
                case Console::NES:
                    emulatorPath = EMULATOR_FCEUX;
                    break;
                case Console::SNES:
                    emulatorPath = EMULATOR_SNES9X;
                    break;
                case Console::GBA:
                    emulatorPath = EMULATOR_MGBA;
                    break;
                default:
                    return;
            }
            launchEmulator(emulatorPath, rom->getFilepath());
        }
    });
```

Figura 4: Fragmento de llamada a launchEmulador en MenuScren.cpp

Primero la función launchEmulator es invocada desde onClick, función definida dentro de MenuScreen para seleccionar el tipo de emulador invocado, en este caso se unificó en Mednafen por lo que encuentra los paths originales del emulador en config.h [Figura 5]

```
        const std::string EMULATOR_FCEUX = "/usr/games/mednafen";
        const std::string EMULATOR_SNES9X = "/usr/games/mednafen";
        const std::string EMULATOR_MGBA = "/usr/games/mednafen";
```

Figura 5: Paths originales para Mednafen en Config.h

A continuación se crea el comando en launchEmulator y es lanzado en caso de no pertenecer a mGBA, en este caso no se envía dentro de esta opción explicita sino que se redirige directamente a la opción genérica adoptando el comando de mednafen, de igual manera esta implementación se debe a las diversas implementaciones probadas a lo largo del desarrollo, dejando esta segunda opción para un posible escalamiento.

```
std::string command;

if (emulatorFixed.find("mgba") != std::string::npos) {
    command = emulatorFixed + " -f \"" + romPathAbs + "\"";
}
else {
    command = emulatorFixed + " \"" + romPathAbs + "\"";
}

std::cout << "[System] Preparando lanzamiento: " << command << std::endl;
```

Figura 6: Generación del comando de lanzamiento en launch emulator

3.4. Funcionalidades destacadas: cartuchos externos

Se implementó la funcionalidad de cartuchos externos para la consola y así expandir en gran medida la utilidad de nuestra consola, pues no simplemente se limita a las roms precargadas sino que mediante una usb puedes añadir las ROMS que quieras a gusto personal. Para esto se implementó una arquitectura de "watchdog.ªcompañada de un automontaje mediante el script automount.sh

```
while true; do
    if [ -e "$DEVICE" ]; then
        if ! mountpoint -q "/media/$LABEL"; then
            echo "¡Dispositivo detectado! Intentando montar..."

            pmount "$DEVICE" "$LABEL"
```

Figura 7: Automontaje de usb mediante script para la detección efectiva

```
LOG="/tmp/wd.log"
echo "Iniciando vigilancia..." > $LOG

while true; do
    if mountpoint -q /media/usb_drive; then

        if pgrep -f "mednafen" > /dev/null; then

            echo "Alerta: USB conectada durante juego. Matando..." >> $LOG
            pkill -9 -f "mednafen"
```

Figura 8: Fragmento de watchdog implementado para la detección de cartucho usb en tiempo de ejecución del emulador

Todo este sistema secuencial no sería posible sin nuestro scritp maestro que dirige toda la secuencia implementada: launcher.sh, script que arranca desde el inicio en nuestra configuración final de SO y realiza toda la orquestación de nuestra consola, para la implementación de esta característica de cartucho fue necesario hacer uso de este script maestro secuencial.

```
/home/pi/proyecto_final/automount_usb.sh & /dev/null 2>&1 &
/home/pi/proyecto_final/usb_watchdog.sh > /dev/null 2>&1 &
clear
# Bucle
while true; do
    ./build/retro-console > /dev/null 2>&1
```

Figura 9: Fragmento de watchdog implementado para la detección de cartucho usb en tiempo de ejecución del emulador

3.5. Sistema operativo y temas personalizados

Se hizo uso de plymouth para configurar todas las personalizaciones que nos permitieron transformar nuestra raspberry en una consola de videojuegos real. En esta implementación de llevó a cabo la ocultación de lineas de comando y una pantalla de carga personalizada mostrando la marca del producto.

```
if [ -d "plymouth" ]; then
    sudo cp -r plymouth/* /usr/share/plymouth/themes/
    sudo plymouth-set-default-theme -R console
fi
```

Figura 10: Implementación de tema personalizado mediante stup.sh, conserva todas las configuraciones realizadas con plymouth y las monta en el sistema del usuario.

El uso de Systemd fue escencial para lograr otpimización de recursos y configuracion limpiea del sistema, a continucación se muestra un fragmento de setup.sh que sintetiza esta configuración general

```
sed "s|/home/pi/proyecto_final|$PROJECT_DIR|g" retro-console.service.bak
> /tmp/retro-console.service
sudo cp /tmp/retro-console.service /etc/systemd/system/retro-console.service
```

Figura 11: Implementación de sysdtemd para configuraciones de arranque y silenciamiento de linea de comandos desde setup.sh

El tercer componente de la personalización y optimización de nuestro sistema operativo se realizó mediante cmdline, esta implementación resultó sumamente cuidadosa puesto que dentro de este archivo se encuentran configuraciones muy sensibles que pueden causar daños irreversibles al hardware, este comando fue sumamente úitil para desaparecer gran parte de lineas de comandos en la pantalla tanto en el inicio como en transiciones y apagado de la consola.

Una implementación interesante fue la de launcher.sh para eliminar definitivamente las lienas de comandos mostradas en transiciones mediante la técnica zoering.

```
ROOT_PART=$(echo "$CURRENT_CMDLINE" | grep -o 'root=PARTUUID=[^ ]*')

if [ -z "$ROOT_PART" ]; then
    echo "ADVERTENCIA No se pudo detectar PARTUUID.
    Se omitirá la modificación de cmdline.txt por seguridad."
else
    echo "   -> PARTUUID detectado: $ROOT_PART"

    # C. nueva línea conservando el UUID original
    NEW_CMDLINE="console=serial0,115200 console=tty3 ${ROOT_PART} rootfstype=ext4
    fsck.repair=yes rootwait quiet splash plymouth.ignore-serial-consoles
    logo.nologo vt.global_cursor_default=0 loglevel=3
    systemd.show_status=false"
```

Figura 12: Configuración de cmdline mediante setup.sh

```
/home/pi/proyecto_final/automount_usb.sh & /dev/null 2>&1 &
/home/pi/proyecto_final/usb_watchdog.sh > /dev/null 2>&1 &
clear
# Bucle
while true; do
    ./build/retro-console > /dev/null 2>&1
```

Figura 13: Técnica zoering en launcher.sh

3.6. Resultados finales

3.6.1. Arranque de la consola

La consola arranca directamente a la pantalla de carga, luciendo la marca del producto de una manera profesional:

Figura 14: Pantalla de carga

Posteriormente avanza con una transición suave sin ningun elemento de por medio a el menú principal, mismo donde puede tranquilamente seleccionar el juego de su elección y ejecutarlo o marcarlo como favorito

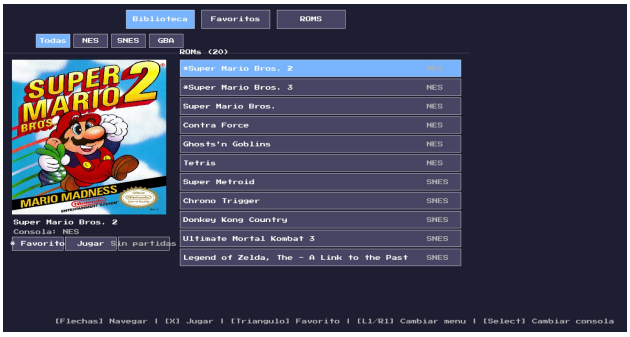

Figura 15: Menú principal

Dentro de la sección ROMS puede acceder a cargar nuevos juegos mediante una usb

Figura 16: Menú para cargar nuevas roms mediantte usb

Nota: La consola unicamente funciona de manera cien por ciento funcional con un control de PlayStation 4 Dualshock mediante cable usb.

Mientras se juega una rom se puede salir al menu principal con el botón PS del control, se guarda automáticamente el progreso al salir y si se inserta una usb mientras corre el juego el sistema te arroja automáticamente al menú principal.

La consola se apaga de manera segura mediante la combinación de presión los dos joysticks al mismo tiempo dos veces seguidas de manera rápida.

3.7. Ligas de interés

- Repositorio de GitHub para código fuente: https://github.com/aaronmolvej/proyecto_final/tree/main
- Video-demostración: https://youtu.be/lLOmO3dR3u0

4. Conclusiones

La implementación de una consola retro parece una misión sencilla, sin embargo existen diversos retos antes de lograr el resultado final bien pulido, desde la arquitectura del software, la implementación de bibliotecas necesarias para el desarrollo, algoritmos y técnicas de optimización de recursos, creación de la interacción completa con el usuario y la consola, las personalizaciones y optimizaciones de arranque, navegación y apagado de la consola y la integración completa entre emuladores para conseguir la compatibilidad adecuada.

Al finalizar este proyecto conseguimos implementar la mayoría de las funcionalidades solicitadas con éxito, el usuario puede sentirse cómodo usando esta consola y pasar horas de diversión sin preocuparse por nada mas que jugar y divertirse. A pesar de todos los desafíos presentados se logró una implementación completa.

5. Referencias

Referencias

[1] M. Barr y A. Massa, *Programming Embedded Systems with C and GNU Development Tools*, 2.ª ed. Sebastopol, CA: O'Reilly Media, 2007.

[2] "Linux GPU Driver Developer's Guide." The Linux Kernel documentation, The Linux Kernel Organization, visitado 27 de nov. de 2025. dirección: `https://docs.kernel.org/gpu/`

[3] "Mednafen Documentation," Mednafen, visitado 27 de nov. de 2025. dirección: `https://mednafen.github.io/`

[4] "systemd System and Service Manager," Freedesktop.org, visitado 27 de nov. de 2025. dirección: `https://systemd.io/`

[5] "systemd.service(5) — Linux manual page," The Linux Kernel Organization, visitado 27 de nov. de 2025. dirección: `https://man7.org/linux/man-pages/man5/systemd.service.5.html`

[6] "Linux Boot Process and initrd," en *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*, Waltham, MA: Morgan Kaufmann, 2012.

[7] "The Frame Buffer Device." The Linux Kernel Documentation, The Linux Kernel Organization, visitado 27 de nov. de 2025. dirección: `https://docs.kernel.org/driver-api/frame-buffer.html`