UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de herramientas que permitan el uso eficiente de audio en el desarrollo de aplicaciones homebrew para la Nintendo 3DS

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Pablo Moreira Flors

Tutores: Miguel Ángel Mateo Pla
Lenin Guillermo Lemus Zúñiga

Curso 2020-2021

Resum

La Nintendo 3DS és una videoconsola portàtil desenvolupada per l'empresa de videojocs Nintendo. Va eixir al mercat en 2011 com a successora de la videoconsola Nintendo DS, incorporant grans millores de potència de processament, tractament de gràfics i reproducció d'àudio, que donen més llibertat i creativitat als desenvolupadors de videojocs.

Interesats en les possibilitats que esta nova consola oferia, una comunitat de desenvolupadors va decidir crear un conjunt d'eines lliures, denominades *homebrew*, per a crear aplicacions per a esta consola. Les eines s'ofereixen com alternativa al kit de desenvolupament comercial oferit per Nintendo. D'esta manera, qualsevol persona interessada podria experimentar amb la Nintendo 3DS creant els seus propis videojocs i utilitats sense necessitat de *hardware* específic o sense haver de comprar costoses llicències. A dia d'avui, una gran quantitat de projectes *homebrew* han sigut desenvolupats gràcies a estes eines lliures i estan disponibles com programes de codi obert.

A pesar de les millores introduïdes a esta nova videoconsola, encara té factors limitants a l'hora de desenvolupar una aplicació, un molt important és la quantitat de memòria disponible: 64 MB. En la memòria cal allotjar el codi executable de l'aplicació, la memòria de les pantalles, les primitives de gràfiques, els fitxers d'àudio i altres dades de propòsit general. Per aquest motiu, la consola inclou hardware específic que permet utlilitzar formats de compressió per a les dades gràfiques i d'audio.

Mentres que el kit de desenvolupament oficial ofereix eines per a la generació i carga d'imatge i àudio comprimit, el kit *homebrew* a soles presenta eines relacionades amb la compressió d'imatge, deixant al desenvolupador la responsabilitat de generar i carregar àudio comprimit pel seu compte.

El principal objectiu d'aquest TFG és facilitar l'ús de formats d'àudio comprimit en el desenvolupament d'aplicacions que fan ús del kit *homebrew* per a Nintendo 3DS. Per a això es desenvoluparà una eina de PC que genere la informació d'àudio comprimit i una biblioteca de funcions que permeta utilitzar aquesta informació comprimida.

L'eina de PC, amb nom ***cwavtool***, permetrà generar fitxers de audio comprimit suportats pel hardware de la consola Nintendo 3DS a partir d'altres formats d'àudio. La biblioteca de funcions per a la consola Nintendo 3DS, amb nom ***libcwav***, permetrà carregar i reproduir a la consola els fitxers d'àudio generats amb l'eina anterior.

El format escollit per als fitxers comprimits és el conegut com *Binary CTR Wave file format* o *BCWAV*. Este format és un format binari, compacte i senzill, usat en una gran varietat d'aplicacions comercials de la consola Nintendo 3DS i que, a diferència d'altres fitxers d'àudio més utlitzats en els ordinadors personals com el format *Waveform audio file format* (*WAV*) o el format *OGG*, permet emmagatzemar àudio comprimit ADPCM (*Adaptive Differential Pulse Code Modulation*) reduint el consum de memòria. Com és un format de fitxer existent, hi ha disponible documentació prèvia i algunes eines que facilitaran la implementació de les eines proposades.

Paraules clau: Nintendo 3DS, *homebrew*, compressió d'audio, biblioteca de funcions, eina d'ordinador personal

Resumen

La Nintendo 3DS es una videoconsola portátil desarrollada por la empresa de videojuegos Nintendo. Salió al mercado en 2011 como sucesora de la videoconsola Nintendo DS, incorporando grandes mejoras de potencia de procesamiento, tratamiento de gráficos y reproducción de audio, que permiten mayor libertad y creatividad a los desarrolladores de videojuegos.

Interesados en las posibilidades que esta nueva consola ofrecía, una comunidad de desarrolladores decidió crear un conjunto de herramientas libres, denominadas *homebrew*, para crear aplicaciones para dicha consola. Estas herramientas se ofrecen como alternativa al kit de desarrollo comercial ofrecido por Nintendo. De esta manera, cualquier persona interesada podría experimentar con la Nintendo 3DS creando sus propios juegos y utilidades sin necesidad de hardware específico o sin tener que comprar costosas licencias. A día de hoy, una gran cantidad de proyectos *homebrew* han sido desarrollados gracias a estas herramientas libres y están disponibles como programas de código abierto.

Pese a las mejoras introducidas a esta nueva videoconsola, sigue teniendo factores limitantes a la hora de desarrollar una aplicación, uno muy importante es la cantidad de memoria disponible: 64 MB. En la memoria hay que alojar el código ejecutable de la aplicación, la memoria de las pantallas, las primitivas de gráficos, los archivos de audio y otros datos de propósito general. Por este motivo, la consola incluye hardware específico que permite usar formatos de compresión para los datos gráficos y de audio.

Mientras que el kit de desarrollo oficial ofrece herramientas para la generación y carga de imagen y audio comprimido, el kit *homebrew* solamente presenta herramientas relacionadas con la compresión de imagen, dejando al desarrollador la responsabilidad de generar y cargar audio comprimido por su cuenta.

El principal objetivo de este TFG es facilitar el uso de formatos de audio comprimido en el desarrollo de aplicaciones usando el kit *homebrew* para Nintendo 3DS. Para ello se desarrollará una herramienta de PC que genere la información de audio comprimida y una biblioteca de funciones que permita utilizar esa información comprimida.

La herramienta de PC, con nombre ***cwavtool***, permitirá generar archivos de audio comprimido soportados por el hardware de la consola Nintendo 3DS a partir de otros formatos de audio. La biblioteca de funciones para la consola Nintendo 3DS, con nombre ***libcwav***, permitirá cargar y reproducir en la consola los archivos de audio generados con la herramienta anterior.

El formato escogido para los archivos comprimidos es el conocido como *Binary CTR Wave file format* o *BCWAV* . Este formato es un formato binario, compacto y sencillo, usado en una gran variedad de aplicaciones comerciales de la consola Nintendo 3DS y que, a diferencia de otros archivos de audio más utilizados en los ordenadores personales como el *Waveform audio file format* (*WAV*) o el *OGG*, permite almacenar audio comprimido ADPCM (*Adaptive Differential Pulse Code Modulation*) reduciendo el consumo de memoria. Al ser un formato de archivo existente, hay disponible documentación previa y algunas herramientas que facilitarán la implementación de las herramientas propuestas.

Palabras clave: Nintendo 3DS, *homebrew*, compresión de audio, biblioteca de funciones, herramienta de ordenador personal

Abstract

The Nintendo 3DS is a portable game console developed by the video game company Nintendo. It was released in 2011 as the successor to the Nintendo DS game console, incorporating vast improvements in processing power, graphics processing and audio playback, giving game developers more freedom and creativity.

Interested in the possibilities this new console offered, a community of developers decided to make a set of free tools, called homebrew, in order to create applications for said console. These tools are offered as an alternative to the commercial development kit offered by Nintendo. This way, anyone interested could experiment with the Nintendo 3DS by creating their own games and utilities without the need for specific hardware nor having to buy expensive licenses. To this date, a large number of homebrew projects have been developed thanks to these free tools and are available as open source applications.

Despite the improvements made to this new game console, it is still a console with limiting factors for application development, a very important one is the amount of available memory: 64 MB. The executable code of the application, the screen buffers, the graphic primitives, the audio files and other general-purpose data have all to fit in memory. For this reason, the console includes specific hardware that allows the use of compression formats for graphics and audio data.

While the official development kit offers tools for generating and loading compressed audio and image data, homebrew tools only give tools related to image compression, leaving the responsibility of generating and loading compressed audio to the developer.

The main objective of this TFG is to facilitate the usage of compressed audio formats in the homebrew application development for the Nintendo 3DS. In order to do this, a PC tool to generate compressed audio information and a function library to allow the use of this compressed information will be developed.

The PC tool, named ***cwavtool***, will generate compressed audio files supported by the Nintendo 3DS hardware from other audio formats. The function library for the Nintendo 3DS console, named ***libcwav***, will allow the audio files generated with the previous tool to be loaded and played on the console.

The format chosen for the compressed files is known as *Binary CTR Wave* or *BCWAV*. This format is a binary, compact and simple format, used in a great variety of commercial applications for the Nintendo 3DS console and, unlike other common audio files used in personal computers such as the *Waveform audio (WAV)* file format or the *(OGG)* file format, *BCWAV* allows to store compressed ADPCM (Adaptive Differential Pulse Code Modulation) audio which reduces memory consumption. As it is an existing file format, there is previous documentation and some tools that will make the implementation of the proposed tools easier.

Key words: Nintendo 3DS, homebrew, audio compression, function library, personal computer tool

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

La Nintendo 3DS es una videoconsola portátil creada por la compañía de videojuegos Nintendo. A día de hoy, se han vendido 75,94 millones de unidades [1], demostrando su éxito en el mercado de las videoconsolas. Además, posee una gran biblioteca de videojuegos desarrollados por la propia compañía, terceras empresas y desarrolladores *indie*.

1.1 Motivación

Tras la salida al mercado de la Nintendo 3DS y como ha sucedido con otras consolas de la compañía, un grupo de desarrolladores se dedicó a investigar el *hardware* y el *software* de la videoconsola con la finalidad de crear un kit de desarrollo (*SDK*) alternativo al ofrecido de forma oficial por Nintendo, motivados por el movimiento del *software* de código abierto. Actualmente, este *SDK homebrew*[1] es alojado y mantenido por la organización DevkitPro.

A diferencia del *SDK* oficial, el kit *homebrew* permite crear un archivo ejecutable capaz de funcionar en una consola comercial, sin necesidad de *hardware* específico adicional. Gracias a esto, cualquier persona interesada puede utilizar el *SDK homebrew* para crear sus propias aplicaciones y juegos para la consola, pudiendo así introducirse en el mundo del desarrollo para videoconsolas y obtener valiosos conocimientos.

Por otro lado, cabe destacar que la disponibilidad de recursos humanos y temporales para el desarrollo de este kit *homebrew* ha sido mucho menor comparado con el kit oficial. Además, mientras que la compañía es conocedora de todos los detalles técnicos de su consola, la comunidad de desarrolladores *homebrew* ha tenido que dedicar recursos a investigar los detalles del *hardware* y *software* para poder crear el kit alternativo. Como resultado, las herramientas y bibliotecas de funciones disponibles son más limitadas.

Gracias al esfuerzo conjunto de todos los usuarios, el *SDK homebrew* ha ido creciendo progresivamente, por ejemplo, facilitando el uso de imágenes y gráficos en las aplicaciones. Sin embargo, uno de los sectores menos desarrollados es el tratado de audio.

[1] "Hecho en casa", es decir, creado por los usuarios en vez de por la compañía.

1.2 Objetivos

Como se ha comentado en la sección anterior, el mantenimiento y desarrollo del kit *homebrew* ha sido posible gracias a la comunidad de usuarios que han dedicado sus recursos para ello. Como miembro de dicha comunidad, he decidido contribuir a la mejora del kit, en este caso, creando un conjunto de herramientas para la generación y reproducción de audio de forma eficiente.

Gracias a este conjunto de herramientas, aquellos usuarios que deseen crear sus propias aplicaciones no deberán preocuparse por los detalles más técnicos y de bajo nivel requeridos para la reproducción de audio, y podrán centrar sus esfuerzos en otros ámbitos del desarrollo de su videojuego o aplicación.

Por tanto, los objetivos de este proyecto son los siguientes:

Objetivos principales:

- Crear una herramienta para ordenador capaz generar archivos de audio compatibles con el *hardware* de audio de la Nintendo 3DS.
- Crear una biblioteca de funciones para la Nintendo 3DS que se pueda incluir en proyectos *homebrew* y permita reproducir los archivos generados por la herramienta.
- Documentar y explicar el funcionamiento tanto de la herramienta como de la biblioteca de funciones.
- Crear una aplicación *homebrew* básica que utilice la biblioteca de funciones con tal de que el usuario tenga un ejemplo a la hora de incorporarla en su proyecto.

Objetivos secundarios:

- Crear un repositorio *git* donde el código de la herramienta y de la biblioteca de funciones esté disponible al público, usando una licencia de código abierto.
- Distribuir ejecutables compilados de la herramienta, con tal de que el usuario no tenga que compilarla por su cuenta.
- Distribuir la biblioteca de funciones mediante un paquete capaz de ser instalado mediante el gestor de paquetes *devkitPro pacman*.

1.3 Estructura de la memòria

A continuación se describirá brevemente el contenido de cada capítulo de la memoria.

El primer capítulo se dedica a realizar una corta introducción a la familia de consolas portables de juegos Nintendo 3DS y la comunidad de *homebrew*, esto es, la comunidad de desarrolladores que existe en torno a esta consola dedicados a crear aplicaciones y juegos de código libre.

En el segundo capítulo, se describe la consola Nintendo 3DS: sus características principales, algunos detalles técnicos importantes para entender el presente TFG y el estado del arte de la reproducción de audio en el desarrollo *homebrew*.

En el tercer capítulo, se realizará un análisis del problema de reproducción de audio en la 3DS y se expondrán posibles soluciones al mismo. La solución elegida se basa en un conjunto de herramientas (programa y bibliotecas de funciones) para la generación de

audio comprimido en un ordenador (PC o similar) y su posterior reproducción en una consola Nintendo 3DS.

Tras la presentación del diseño general, en el capítulo cuarto nos centraremos en detallar la fase de diseño e implementación de la herramienta para ordenador ***cwavtool*** para la generación de archivos de audio comprimido.

En el quinto capítulo, se describirá la fase de diseño e implementación de la biblioteca de funciones para Nintendo 3DS ***libcwav*** para la reproducción de los archivos de audio generados.

En el sexto capítulo, se explicará el diseño de una aplicación de ejemplo disponible para desarrolladores que quieran usar las herramientas, que además servirá para realizar las pruebas necesarias de las herramientas implementadas.

En el séptimo capítulo, se mostrará cómo se ha distribuido el conjunto de herramientas implementado y se pondrán ejemplos reales de su uso en aplicaciones para Nintendo 3DS.

Finalmente, en el último capítulo, se realizará una conclusión del trabajo realizado, y se expondrán posibles mejoras que se podrán realizar en trabajos futuros.

CAPÍTULO 2
Estado del arte

La Nintendo 3DS es una videoconsola portátil creada por Nintendo y lanzada al mercado en el año 2011. Fue diseñada como sucesora directa de la consola Nintendo DS, manteniendo compatibilidad binaria con muchos juegos y reincorporando algunas de las principales características que marcaron el éxito de su predecesora, como el formato plegable, dos pantallas independientes o el uso de cartuchos para la carga de videojuegos. Además, se incluyeron mejoras importantes como *hardware* más eficiente, la capacidad de mostrar imágenes 3D sin gafas en la pantalla superior o la inclusión de un *stick* analógico para mayor precisión en el control.

Dado que la consola Nintendo 3DS posee componentes muy similares a aquellos encontrados en la Nintendo DS, es retro-compatible con los videojuegos de Nintendo DS, lo que amplió en gran medida la cantidad de aplicaciones y juegos disponibles en el mercado. Sin embargo, la Nintendo 3DS eliminó el puerto para cartuchos de GameBoy Advance, siendo solamente posible jugar a sus juegos si están disponible para la consola virtual en la tienda en línea.

Tras la salida al mercado de la Nintendo 3DS, Nintendo comenzó a crear modelos alternativos que reducían costes o que mejoraban alguna de las características del producto. Aunque cada uno de estos modelos contiene características diferentes, en lo que concierne a este proyecto, diferenciaremos solamente entre dos familias de modelos.

Por un lado, contamos con la familia de modelos Nintendo 3DS original: Nintendo 3DS, Nintendo 3DS XL (mayor pantalla) y Nintendo 2DS (sin funcionalidad 3D), lanzados al mercado entre 2011 y 2014. Por otro lado, disponemos de la familia de modelos "nuevos"de Nintendo 3DS: New Nintendo 3DS, New Nintendo 3DS XL y New Nintendo 2DS XL,lanzados al mercado entre 2015 y 2017, cuya principal diferencia a los modelos originales es la mejora de prestaciones y la adición de nuevos botones.

En la figura 2.1 se pueden observar los diferentes modelos según el orden en el que salieron al mercado.

Figura 2.1: Modelos familia Nintendo 3DS [2]

2.1 *Hardware* de Nintendo 3DS

Para poder entender a qué problema nos enfrentamos, es necesario conocer los detalles técnicos del sistema. De esta manera, podremos hacernos una idea de para qué tipo de máquina se va a diseñar la solución y qué limitaciones presenta.

2.1.1. Especificaciones técnicas

Dado que la Nintendo 3DS se trata de una consola privativa, los detalles del *hardware* no han sido revelados por la compañía. Sin embargo, gracias al esfuerzo conjunto de la comunidad *homebrew*, ha sido posible recopilar la siguiente lista de características principales:

- **Unidad procesadora**:
  - **Familia original**: *ARM11 MPCore* de 2 núcleos cada uno con una unidad coprocesadora *VFPv2* (268MHz). 1 núcleo *ARM946* (134MHz). 1 núcleo ARM7. [3, 4]
  - **Familia *new***: *ARM11 MPCore* de 4 núcleos cada uno con una unidad coprocesadora *VFPv2* (804MHz) y 2MB de caché L2 adicional. 1 núcleo *ARM946* (134MHz). 1 núcleo ARM7. [3, 4]
- **Gráficos**: GPU *DMP PICA200* (268MHz) con 6MB de VRAM interna [3]. Pantalla superior LCD TN [5] de 400x240 píxeles capaz de mostrar imágenes 3D sin gafas, pantalla inferior LCD TN de 320x240 píxeles[1].
- **RAM**:
  - **Familia original**: 128MB [6]
  - **Familia *new***: 256MB [6]
- **Audio**: *DSP CEVA TeakLite* (134MHz). 24 canales con frecuencia de muestreo máxima de 32728Hz. [3]
- **Almacenamiento interno**: 1GB aprox. memoria flash usable. [7]
- **Almacenamiento externo**: Ranura para tarjeta SD compatible con *SDHC*.
- **Interfaces**:
  - **Todos los modelos**: Conectividad 802.11 WiFi [8], ranura de cartuchos de juego de Nintendo DS y Nintendo 3DS, micrófono monoaural, emisor/receptor de infrarrojos [9], botones y pantalla inferior táctil.
  - **Familia *new***: Conectividad NFC.

Como se puede observar, las especificaciones técnicas son muy inferiores a cualquier consola de sobremesa u ordenador personal actual. Esto es debido a que todos los componentes deben ser eficientes energéticamente, al funcionar la consola mediante una batería recargable, y las restricciones de espacio y peso para que la consola sea manejable.

[1] En algunas unidades de New Nintendo 3DS, se pueden encontrar pantallas LCD IPS [5].

2.1.2. Recursos disponibles en aplicaciones

De las especificaciones descritas en el apartado anterior, solamente una parte de los recursos están disponibles para el desarrollador. Esto es debido a que parte de ellos están reservados para el correcto funcionamiento del sistema.

Respecto al procesador, solo un núcleo ARM11 en el caso de la familia original y 2 núcleos ARM11 en el caso de la familia *new* están dedicados a la ejecución de aplicaciones, ya que el resto de núcleos ARM11 están reservados para el sistema operativo [10]. Por otro lado, el núcleo ARM9 se encarga del acceso al almacenamiento, las operaciones de seguridad (criptografía) [11] y como procesador principal en modo retro-compatibilidad con Nintendo DS. Finalmente, el núcleo ARM7 solo está activo en modo retro-compatibilidad para Nintendo DS y GameBoy Advance [4].

Respecto a la memoria, una gran parte de ella está reservado para el sistema operativo, el menú HOME y otras aplicaciones (*applets*) que funcionan con la aplicación principal suspendida en segundo plano [11]. En el caso de la familia original, existen dos modos de memoria: el modo 64MB, usado por la gran mayoría de videojuegos comerciales, y el modo 80MB que limita la ejecución de *applets* en segundo plano y es menos usado. En el caso de la familia *new*, están disponibles los modos 64MB (retro compatibilidad familia original) y 124MB [12].

2.2 *Software* de Nintendo 3DS

La Nintendo 3DS ha contado a lo largo de su ciclo de vida con una gran librería de videojuegos y utilidades. Esta librería se puede considerar como el conjunto de aplicaciones de la consola.

Por otro lado, es necesaria la existencia de un programa cargador con una interfaz de usuario, para que las aplicaciones puedan ser lanzadas. En el caso de la Nintendo 3DS, este se trata del menú HOME, que no es más que otra aplicación.

Todas estas aplicaciones, junto con el sistema operativo, puede definirse como el *software* de la consola.

A continuación se realizará una explicación de cada tipo de *software* necesario para el desarrollo de este TFG, en un nivel mas técnico.

2.2.1. Aplicaciones

Existen dos tipos de aplicaciones oficiales para la Nintendo 3DS. Por un lado, pueden ser obtenidas de forma digital a través de la tienda en línea e instaladas en la tarjeta SD (formato *.cia* o *CTR*[2] *Importable Archive*) [13], o almacenadas en un cartucho de juego de solo lectura disponible en tiendas físicas (formato *.cci* o *CTR Cart Image*) [14]. Ambos tipos pueden ser lanzados directamente desde el menú HOME de la consola.

Sin embargo, los formatos de archivo oficiales requieren permisos elevados para poder ser utilizados, por lo que la comunidad *homebrew* tuvo que diseñar su propio formato de aplicación, conocido como el formato *.3dsx* [15], que puede ser ejecutado a través de el cargador de aplicaciones *homebrew* (*The Homebrew Launcher*) [16]. Gracias a los avances en la investigación del sistema operativo de la Nintendo 3DS, hoy en día también es posible utilizar el formato *.cia* para la distribución de aplicaciones *homebrew* junto con el formato *3dsx*.

[2]Las siglas CTR, cuyo significado es incierto, denotan el nombre en clave de la Nintendo 3DS.

Independientemente del formato de aplicación, todos ellos cuentan con un segmento de código ejecutable para el núcleo ARM11 y un contenedor de archivos (llamado *romfs*) utilizado para almacenar recursos audiovisuales o cualquier otro tipo de ficheros necesarios por la aplicación.

También es importante destacar un mecanismo diseñado por la comunidad *homebrew* que permite la modificación y extensión de funcionalidad de aplicaciones oficiales. Este mecanismo se conoce como "*plugins 3GX*", que son trozos de código inyectados en la memoria del proceso de una aplicación mientras está en ejecución. La mayoría de los "*plugins*" utilizan la biblioteca de funciones *CTRPluginFramework*, que expone una gran variedad de funciones al programador para que pueda expandir la funcionalidad de la aplicación anfitriona a su gusto [17]. Dado que se trata de código inyectado y la aplicación anfitriona ocupa casi toda la memoria disponible, tan solo se pueden utilizar 5MB de memoria del sistema, donde se debe incluir el propio ejecutable del "*plugin*".

En la figura 2.2 se puede observar el cargador *The Homebrew Launcher* instalado como aplicación *.cia* en el menú HOME de la consola.

Figura 2.2: The Homebrew Launcher en el menú HOME.

2.2.2. Sistema operativo

A diferencia de anteriores consolas portátiles de Nintendo, la Nintendo 3DS cuenta con un sistema operativo encargado de otorgar una capa de abstracción con el *hardware*, en vez de otorgar control total a la aplicación en ejecución [11].

El sistema operativo está compuesto por un *microkernel*, encargado de tareas básicas como el mapeo de memoria o la gestión de hilos [18], y un conjunto de módulos (siendo cada módulo un proceso independiente en un núcleo ARM11 dedicado solo para estos) que ofrecen un API de servicios a otras aplicaciones o módulos a través de un protocolo IPC (*Inter-Process Communication*) [19].

Cuando una aplicación requiere de un recurso ofrecido por un servicio, ésta debe negociar con el *microkernel* y el módulo correspondiente el acceso al canal de comunicación a través de IPC. Gracias a este diseño, es posible controlar el acceso a los servicios, limitando aquellos que no son necesarios por la aplicación y que supondrían un compromiso de la seguridad del sistema.

Pese a la existencia de esta API de servicios, es necesaria una capa de abstracción adicional, ya que la API no está diseñada para ser utilizada por el programador directamente, requiriendo métodos de sincronización y gestión de memoria avanzados. En el caso del SDK oficial, Nintendo ofrece un *framework* C++ para que el programador no tenga que ocuparse de la comunicación con la API. En el caso del SDK *homebrew*, DevkitPro ofrece la biblioteca de funciones *libctru*, escrita en C, que facilita el acceso a dichos servicios [20].

En la figura 2.3 se puede observar la estructura de una aplicación de Nintendo 3DS y sus canales de comunicación con los servicios y el *hardware*.

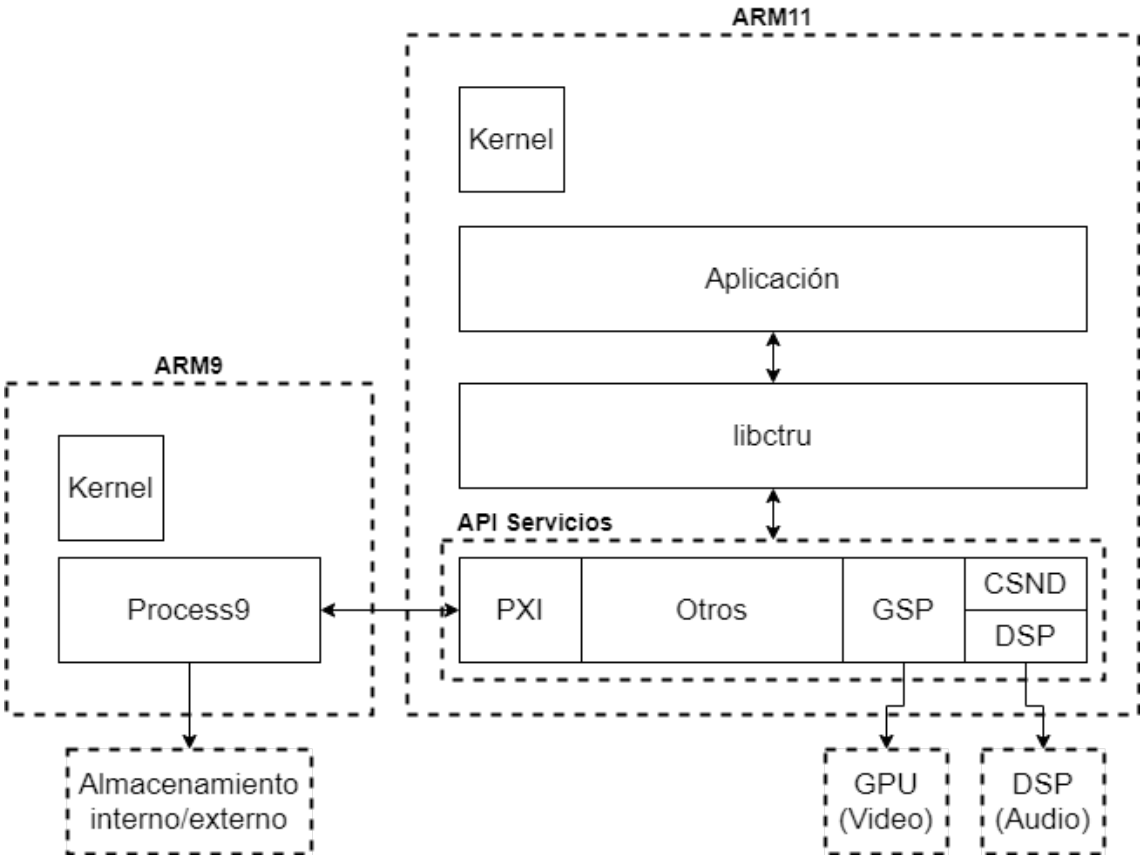

Figura 2.3: Estructura de comunicación aplicación y sistema operativo (simplificado).

2.3 Audio en la Nintendo 3DS

La reproducción de audio es un aspecto muy importante en el desarrollo de videojuegos y una manera adicional de aportar información al jugador. Tanto es así que la capacidad de reproducción de audio ha sido un factor imprescindible desde las primeras videoconsolas lanzadas al mercado. Desde entonces, esta funcionalidad ha ido evolucionando según la tecnología ha mejorado con los años, siendo incluso directamente el objetivo de muchos desarrollos e investigaciones.

Un ejemplo claro lo podemos encontrar en la GameBoy original. Esta consola portátil de 1989 revolucionó el mundo de las videoconsolas portátiles y ya incorporaba la capacidad de reproducir audio. La GameBoy contaba con cuatro canales de audio [21]: dos de ellos se trataban de un generador de onda cuadrada y otro generaba ruido blanco pseudo-aleatorio. El último canal era especial y podía reproducir una onda a partir de una tabla de 32 valores (muestras) de 4 *bits* definida por el usuario, donde cada muestra representaba el valor de amplitud de la onda en ese instante (técnica *Pulse Code Modulation* o *PCM* [22]).

Los primeros tres canales de audio de la GameBoy solo permitían la reproducción de sonidos muy básicos y, pese a que el cuarto canal ya ofrecía el concepto de muestra *PCM*, solo se podían definir 32 de ellas que eran repetidas en bucle. Por este motivo, no era posible la reproducción de sonidos complejos a no ser que se utilizasen ciertas técnicas como la modulación de volumen, que consumían gran parte o todo el tiempo de procesamiento disponible [23].

Volviendo a la actualidad, a diferencia de la GameBoy, la Nintendo 3DS es capaz de reproducir una secuencia de muestras de longitud arbitraria almacenada en memoria. Cada una de estas muestras puede tener una resolución de hasta 16 *bits*, haciendo posible la reproducción de casi cualquier sonido sin pérdida de calidad. En la siguiente sección se dará una explicación más técnica del *hardware* de audio de la Nintendo 3DS, incluyendo los diferentes formatos de muestra soportados.

2.3.1. Características del *hardware* de audio

Como se ha especificado en la sección 2.1.1, la Nintendo 3DS cuenta con un *DSP CEVA TeakLite* (134MHz) de 24 canales con frecuencia de muestreo máxima de 32728Hz. Cada uno de estos canales puede ser configurado independientemente, para poder reproducir sonido de forma paralela.

Una de las características más importantes del *hardware* de audio, es la capacidad de reproducir muestras codificadas en *Adaptive Differential Pulse Code Modulation (ADPCM)*, además de la codificación estándar *PCM* de 8 y 16 *bits* (*PCM8* y *PCM16* respectivamente).

Los algoritmos *ADPCM* permiten que las muestras sean almacenadas en 4 bits, usando un predictor y unos parámetros iniciales para intentar reconstruir la señal analógica original [24]. En el caso de la Nintendo 3DS, los algoritmos *ADPCM* disponibles son:

IMA-ADPCM: Es un algoritmo desarrollado por Interactive Multimedia Association, donde el audio está codificado como una secuencia de muestras de 4 *bits*. Para la decodificación, es necesario conocer la primera muestra descomprimida, además del índice inicial en la tabla de decodificación [25]. Este formato de compresión permite que un canal mono *PCM16* se pueda almacenar con un tamaño 25 % menor que el original.

DSP-ADPCM: Es un algoritmo diseñado por Nintendo para sus videoconsolas, incluyendo la Nintendo GameCube, Nintendo Wii y Nintendo 3DS. En este caso, el audio está codificado como bloques de 8 *bytes*, donde cada bloque contiene 14 muestras de 4 *bits* y una cabecera de 8 *bits*. Para la decodificación, es necesario conocer las dos primeras muestras descomprimidas, además de un par índice-escala y 16 coeficientes usados por el predictor [26]. Este formato de compresión permite que un canal mono *PCM16* se almacene con un tamaño 28.5 % menor que el original, pero produce audio de mayor calidad respecto a la codificación *IMA-ADPCM*.

2.3.2. Servicios de audio

La Nintendo 3DS cuenta con 2 servicios responsables de la grabación y reproducción de audio. Ambos están disponibles desde el entorno de desarrollo *homebrew* y tienen sus particularidades, pudiendo incluso reproducir audio al mismo tiempo.

DSP: Es el servicio de audio principal, que permite mayor libertad de ajustes de los canales de audio y la utilización de la codificación *DSP-ADPCM*. En cambio, requiere la carga de un *firmware* propietario usado por el *hardware DSP*. Este servicio es el utilizado por las aplicaciones comerciales para la reproducción de audio [27].

CSND: Servicio de audio secundario, más limitado, pero que permite utilizar la codificación *IMA-ADPCM*. Este servicio es usado por *applets* en segundo plano para reproducir efectos de sonido (por ejemplo, el teclado en pantalla), dado que la aplicación en primer plano puede estar usando el servicio *DSP*. CSND no requiere un *firmare* adicional, dado que negocia internamente con el servicio *DSP* para la reproducción de audio [28].

2.4 Audio en el entorno de desarrollo *homebrew*

De igual forma que el resto de servicios, los servicios de audio *DSP* y *CSND* requieren un proceso de inicialización y gestión del canal de comunicaciones para funcionar correctamente. De esto se ocupa la biblioteca de funciones *libctru*, que elimina la necesidad de que el desarrollador tenga que realizar dichas operaciones.

2.4.1. Audio en *libctru*

La biblioteca de funciones *libctru* permite usar los dos servicios de audio descritos anteriormente mediante las siguientes interfaces:

ndsp: Utiliza el servicio *DSP* para la gestión de audio, permitiendo ajustar diversos parámetros de audio para cada canal, incluyendo el volumen, ajustes de audio envolvente y filtros aplicados a las muestras. Por motivos legales, el usuario debe extraer el *firmware DSP* de una aplicación oficial (por ejemplo, el menú HOME) y colocarlo en la tarjeta SD. Esta interfaz se encarga además de la gestión de la pausa o detención del audio en el caso de suspender o cerrar la aplicación.

csnd: Utiliza el servicio *CSND* para la gestión de audio, con funciones limitadas para cada canal, como el ajuste del volumen o la reproducción en bucle. Dado que el servicio *csnd* es rara vez utilizado, *libctru* no implementa la gestión de pausa o detención del audio al suspender o cerrar la aplicación, siendo responsabilidad del desarrollador.

2.4.2. Bibliotecas de funciones de audio

Pese a la capa de abstracción ofrecida por *libctru* con la API de servicios, las interfaces *ndsp* y *csnd* aún requieren ciertas operaciones a bajo nivel por parte del usuario. Por ejemplo, sigue siendo necesario gestionar los canales de audio disponibles (24 en total) o manejar direcciones de memoria y otras configuraciones para indicar a la interfaz donde están situadas las muestras de audio. En comparación, la biblioteca gráfica Citro2D ofrecida por DevkitPro, que permite la muestra de gráficos 2D, provee una mayor abstracción al tener sus propias funciones de carga, transformación y visualización de *sprites*, sin necesidad de manejar direcciones de memoria en ningún momento [29].

Por otro lado, sí que existen bibliotecas de funciones externas adaptadas a la Nintendo 3DS (conocidas como *portlibs*) para el manejo del audio. Por ejemplo: *3ds-libogg*, para la

reproducción de archivos *.ogg*; *3ds-flac*, para la reproducción de audio *flac*; o *3ds-mikmod*, para la reproducción de archivos *.mod* [30]. Sin embargo, ninguna de ellas permite la reproducción acelerada por *hardware*, dado que utilizan audio sin comprimir o requieren de la decodificación del audio comprimido por parte del procesador principal.

También hay disponibles otros proyectos realizados por usuarios ajenos DevkitPro que facilitan la reproducción de audio, como por ejemplo *m3diaLib* [31] o *SDL-3DS* [32], pero, igual que las *portlibs*, no permiten la reproducción de audio comprimido *ADPCM*.

2.5 Solución oficial para la reproducción de audio

Pese a los conocimientos reducidos de las soluciones que utiliza Nintendo en su kit de desarrollo oficial, la comunidad *homebrew* es consciente de algunos detalles de sus herramientas, gracias a la investigación del *hardware* y del *software* de la Nintendo 3DS.

Una de las herramientas conocidas, es un conjunto de utilidades para ordenador y bibliotecas de funciones para la Nintendo 3DS que permiten al desarrollador crear diferentes formatos de archivos orientados al desarrollo de videojuegos, y cargarlos posteriormente en la videoconsola. Por ejemplo, para modelos 3D existe el archivo *Binary CTR Model (bcmdl)* [33] o para imágenes, el formato *Binary CTR Layout Image (bclim)* [34].

2.5.1. Formato BCWAV

El formato de archivo *Binary CTR Wave (BCWAV)* es un formato binario cuya funcionalidad es almacenar sonido para ser reproducido en la Nintendo 3DS [35]. Sus características principales son las siguientes:

- Almacenamiento de audio de cualquier longitud y número de canales, siempre y cuando todos los canales tengan la misma longitud.
- Soporte de los siguientes formatos: *PCM8*, *PCM16*, *IMA-ADPCM*, *DSP-ADPCM*.
- Provisión de los parámetros necesarios para la decodificación *ADPCM*.
- Posibilidad de indicar un punto de bucle opcional, especificando el principio y el final del rango a repetir. De esta manera, una vez se alcance el punto final del rango de bucle, se reanudará la reproducción desde el principio del rango de bucle, sin detenerse nunca.

Como se puede observar, a diferencia de otras soluciones *homebrew*, este formato si que permite aprovechar las opciones de codificación comprimida que ofrecen los servicios de audio. Además, se trata de un formato compacto, con una pequeña cabecera con la información del audio almacenado, y una sección con las muestras de audio. Dado que no se almacenan meta-datos adicionales, a diferencia de otros formatos como el *mp3*, su uso de memoria es altamente eficiente.

2.6 Uso de archivos *BCWAV* en el entorno *homebrew*

Pese a que el formato *BCWAV* se trata de un archivo oficial, existen algunas herramientas capaces de reproducir y generar este tipo de archivo.

En primer lugar, los usuarios interesados en la investigación de los datos de juegos oficiales, han creado herramientas capaces de reproducir este tipo de archivo. Dicha tarea ha sido sencilla, ya que la codificación *DSP-ADPCM*, presente desde la Nintendo GameCube, ha sido conocida y documentada desde entonces. El resto de codificaciones soportadas por el archivo son estándar y no conllevan tampoco dificultad para reproducir. La biblioteca de funciones *vgmstream* junto con el reproductor de sonido *foobar2000* son capaces de reproducir los archivos *BCWAV* desde un ordenador [36].

En segundo lugar, aquellos usuarios dedicados en la modificación y expansión de videojuegos oficiales (conocidos en inglés como *ROM hacks*) han creado herramientas para la generación de archivos *BCWAV* con finalidad de reemplazar sonido y música en juegos oficiales. Sin embargo, estas herramientas solo suelen soportar la codificación *DSP-ADPCM* (el más usado en videojuegos oficiales) y en la mayoría de los casos, solo están disponibles para el sistema operativo Windows. La herramienta *Citric Composer*, escrita en C#, es capaz de generar archivos *BCWAV* con codificación *DSP-ADPCM* [37].

Finalmente, la comunidad de desarrolladores de aplicaciones *homebrew* desarrollaron una herramienta de línea de consola, conocida como *bannertool*, con funcionalidad para generar archivos *BCWAV* con codificación *PCM8* y *PCM16* [38]. Esto es debido a que uno de los metadatos contenidos en las aplicaciones distribuidas en *.cia* (véase sección 2.2.1) es un pequeño modelo 3D que contiene un efecto de sonido, reproducido cuando la aplicación es seleccionada en el menú HOME.

CAPÍTULO 3
Análisis

Una vez expuesto el estado del arte, procederemos a analizar las limitaciones de la reproducción del audio en la Nintendo 3DS usando el entorno *homebrew* y sus posibles soluciones. Estas soluciones permitirán implementar un conjunto de herramientas que faciliten el uso eficiente de las capacidades de de audio de la consola en el desarrollo *homebrew*.

3.1 Limitación de memoria y compresión de audio

Pese a que el modelo original de Nintendo 3DS posee un modo de memoria de 80MB, este modo no es muy utilizado al limitar el uso de *applets* en el menú HOME y porque conlleva un mayor tiempo de carga al iniciar la aplicación (la consola se reinicia en un modo especial para poder reservar dicha cantidad de memoria). Es por esto que, si no es absolutamente necesario, todas las aplicaciones se inician en el modo de memoria de 64MB. Como queremos dar soporte a todos los modelos de Nintendo 3DS, consideraremos el modo 64MB como el factor limitante de memoria.

Procedamos a hacer cálculos: ¿cuánto espacio de memoria necesitaría una aplicación que requiere la carga de 10 minutos de efectos de sonido y música? Si asumimos que no se desea perder calidad de audio, entonces las muestras de sonido serán almacenadas con PCM16 a 32728 Hz. Al realizar los cálculos, obtenemos el siguiente resultado:

$$Memoria\ (MB) = 10min * \frac{60s}{1min} * \frac{32728\ muestras}{1s} * \frac{2\ bytes}{1\ muestra} * \frac{1\ megabyte}{10^6\ bytes} \simeq 39MB$$

Esta cantidad de memoria representa el 60 % de la cantidad total disponible y, considerando que el código de la aplicación puede llegar a ocupar 5MB[1], obtenemos que tan sólo quedan restantes 20MB para almacenar los datos de gráficos y de propósito general (*heap*).

Sin embargo, gracias a la codificación *ADPCM* acelerada por *hardware* (no requiere tiempo de procesamiento adicional), podemos disminuir la cantidad de espacio que ocupan nuestros 10 minutos de audio, eso sí, con una pequeña pérdida de calidad. Si se utilizase la codificación DSP-ADPCM, que requiere 8 bytes por cada 14 muestras, nuestros 10 minutos de audio ocuparían:

$$Memoria\ (MB) = 10min * \frac{60s}{1min} * \frac{32728\ muestras}{1s} * \frac{8\ bytes}{14\ muestras} * \frac{1\ megabyte}{10^6\ bytes} \simeq 11MB$$

[1] Basándose en el análisis del videojuego oficial *Mario Kart 7*

En este caso, la cantidad de memoria necesaria para el almacenamiento del audio es un 17 % de la memoria total de la consola, lo que supone una gran mejora respecto a no usar ningún tipo de compresión en el audio.

Viendo los resultados anteriores, podemos llegar a la conclusión que para poder desarrollar un videojuego para Nintendo 3DS, la compresión de audio acelerada por *hardware* es completamente necesaria.

3.2 Almacenamiento de audio comprimido

El siguiente problema a analizar es cómo poder cargar audio comprimido en la Nintendo 3DS que haya sido generado en un entorno de desarrollo para ordenador. Dado que ningún archivo de audio convencional, como *WAV* o *OGG*, soporta la compresión *DSP-ADPCM*, es necesario buscar soluciones alternativas.

3.2.1. Solución 1: Archivos *RAW*

Una solución sencilla es simplemente guardar las muestras comprimidas en un archivo binario sin ninguna información adicional. Posteriormente en la Nintendo 3DS, este archivo se carga directamente en memoria y se especifican los parámetros necesarios para su reproducción manualmente desde el código del programa.

Esta solución, pese a ser fácil de implementar, puede suponer una mayor carga de trabajo si se desean reemplazar los archivos de audio o añadir nuevos, ya que requiere también la modificación del código de la aplicación cada vez que esto sucede. Además, es imposible cargar archivos de forma dinámica desde un medio de almacenamiento, ya que es necesario conocer los parámetros de reproducción, que no vienen incluidos en el archivo.

3.2.2. Solución 2: Formato de archivo nuevo

Como mejora a la solución anterior, es posible incorporar los parámetros de reproducción en una pequeña cabecera junto con las muestras comprimidas. De esta manera, la aplicación de Nintendo 3DS será capaz de saber todos los parámetros necesarios sin necesidad de que sean especificados por código.

Dado que se requiere la especificación de una cabecera estándar, esta solución es equivalente a la creación de un formato de archivo completamente nuevo con unas especificaciones determinadas para que pueda ser entendido por la biblioteca de funciones responsable de reproducirlo. Como ventaja, al ser nosotros los diseñadores del formato, podemos incluir cualquier modificación o característica nueva si lo viéramos necesario. Sin embargo y como desventaja, partiríamos desde cero al no contar con documentación y herramientas previas que facilitasen la implementación.

3.2.3. Solución 3: Formato de archivo existente

Como última alternativa, se puede usar un formato de archivo existente para el almacenamiento de audio comprimido. Es aquí donde entra en juego el formato de archivo *BCWAV*, explicado en el apartado 2.5.1.

El formato *BCWAV* permite guardar cualquier número de canales de audio, soportando las codificaciones de audio comprimido de la Nintendo 3DS aceleradas por *hardware*.

Además, permite la especificación de puntos de bucle, siendo muy útil para el desarrollo de videojuegos.

Por otro lado, el formato *BCWAV* está completamente documentado en la página web `http://www.3dbrew.org` y existen herramientas capaces de generar estos archivos a partir de otros más comunes, como el formato *WAV* o *OGG*, lo que facilitaría la implementación de nuestro conjunto de herramientas para la generación y reproducción de audio comprimido.

3.2.4. Solución escogida para el proyecto

Tras exponer las soluciones anteriores, debemos analizar cual de ellas se ajusta más a los objetivos de este proyecto.

En primer lugar, podemos descartar la solución 1 (ver Sección 3.2.1) , dado que para la biblioteca de funciones es conveniente otorgar flexibilidad y transparencia al programador en la carga de archivos, cosas que son muy difíciles de lograr si se deben especificar los parámetros de audio para cada archivo. Por tanto, debemos elegir entre las soluciones 2 y 3. Dado que este proyecto debe realizarse en un tiempo limitado, podemos sacrificar el hecho de poder añadir cualquier funcionalidad al formato de archivo, dejando la solución 3 (ver Sección 3.2.3) como mejor opción. De este modo, se puede aprovechar el material existente para facilitar la implementación del conjunto de herramientas.

Por tanto, el conjunto de herramientas deberá funcionar con archivos *BCWAV* y ser compatible con todas sus funcionalidades, incluyendo los diferentes formatos de audio comprimido y los puntos de bucle opcionales.

3.3 Estructura de un archivo *BCWAV*

En la sección 2.5.1 ya se realizó una pequeña descripción del formato *BCWAV*, sin embargo, si se desea implementar herramientas que utilicen este tipo de archivo, es necesario conocer su estructura interna de una forma más detallada.

Los archivos *BCWAV* cuentan internamente con 3 bloques de datos diferentes. El primero de ellos, contiene los metadatos del archivo, es decir, la información general sobre el propio archivo. El segundo incluye los metadatos del audio almacenado y, finalmente, el tercer bloque alberga los datos de audio en sí mismo, es decir, los datos de las muestras.

En la figura 3.1 se puede observar un ejemplo del contenido de un archivo *BCWAV* visualizado con el editor de ficheros hexadecimal *HxD*.

A continuación se describirán las diferentes estructuras de datos encontrados en el archivo, incluyendo los diferentes bloques.

3.3.1. Referencias, referencias con tamaño y tablas de referencias

Las primeras estructuras a analizar son los diferentes tipos de referencia. Dado que el archivo *BCWAV* es un formato compacto, aquellos datos de la cabecera no necesarios son eliminados. Para poder indicar al programa que procesa el archivo la posición de un conjunto de datos que puede o no existir, se utiliza una referencia. Si esta referencia tiene un valor negativo, quiere decir que ese conjunto de datos no existe en el archivo.

Por otro lado, como la cantidad de canales de audio almacenados puede ser arbitraria, se utiliza una tabla de referencias para indicar la posición de la información de cada canal dentro de la cabecera. Los diferentes tipos de referencias son los siguientes:

- **Referencia**: Indica la posición de un conjunto de datos. Cada referencia tiene un tamaño de 8 *bytes*. Los primeros 2 *bytes* son un identificador que indica el tipo de dato referenciado. Los siguientes 2 *bytes* son de relleno y, finalmente, los últimos 4 *bytes* son un valor de 32 *bits* con signo que representan el *offset* al conjunto de datos. Este *offset* es relativo a otra posición dentro del archivo dependiendo de qué conjunto de datos es referenciado.
- **Referencia con tamaño**: Se trata de una referencia con 4 *bytes* adicionales al final, indicando el tamaño del conjunto de datos referenciado.
- **Tabla de referencias**: Como su nombre indica, se trata de una tabla que almacena una secuencia de referencias. Los primeros 4 *bytes* indican la cantidad de referencias, y el resto de la estructura son las referencias en sí. En este caso, el *offset* de cada referencia siempre es relativo a la posición de la cantidad de referencias.

A partir de ahora, cuando se mencione que una referencia "es relativa a una posición" querrá decir que el valor del *offset* de la referencia es relativo a dicha posición.

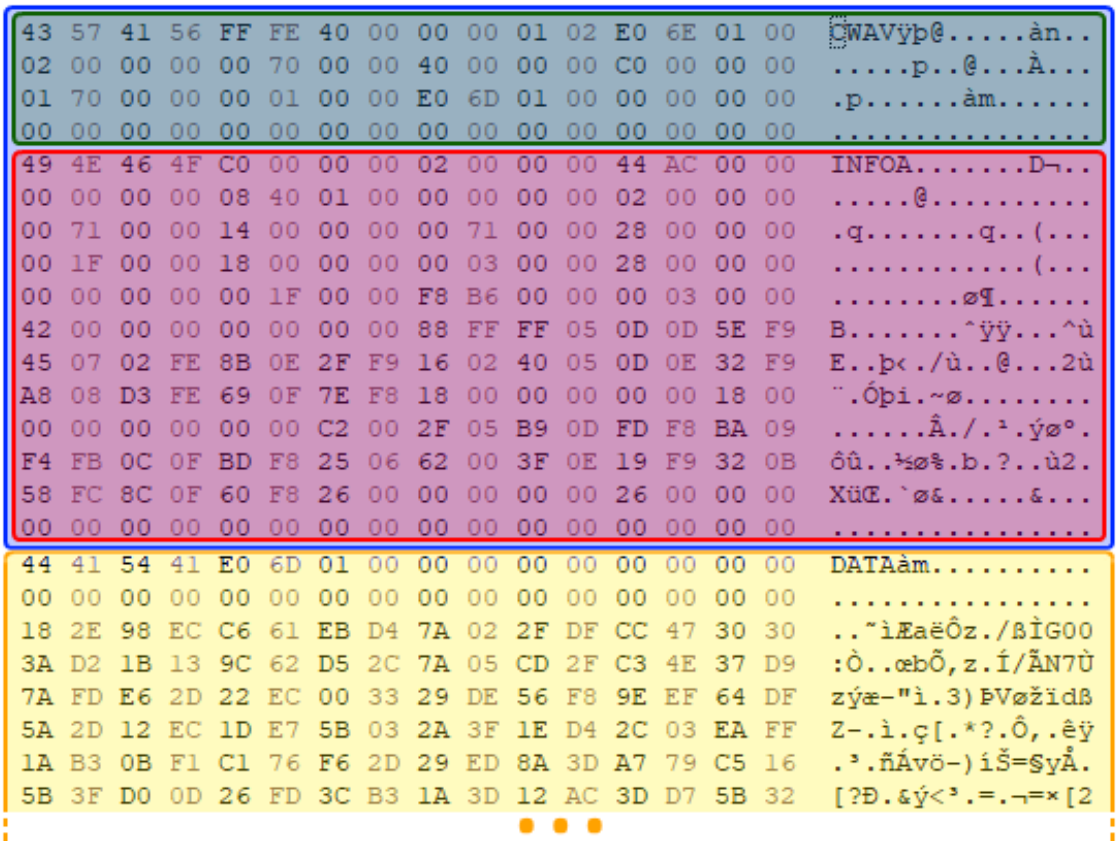

Figura 3.1: Ejemplo de archivo BCWAV. Azul: cabecera, verde: información general, rojo: información de audio, amarillo: muestras de audio.

3.3.2. Bloque de información general (*CWAV*)

Este bloque es el primero que se encuentra en el archivo e indica información genérica sobre él. Se puede identificar porque los primeros 4 *bytes* contienen el valor *ASCII*: *CWAV*. En la figura 3.1, este bloque está representado de color verde.

Los datos almacenados en este bloque son: el tamaño del archivo, la versión del mismo, el tamaño de este bloque, el orden de *bytes* (*endianness*), la cantidad de bloques almacenados y dos referencias con tamaño para el bloque de información de audio (*INFO*) y

el bloque de muestras (*DATA*) relativos al principio del archivo. El resto de *bytes* de este bloque son de relleno para asegurar el correcto alineamiento de los siguientes datos.

3.3.3. Bloque de información de audio (*INFO*)

Este bloque es el siguiente al bloque de información general, y contiene diversas configuraciones y parámetros para el audio almacenado. Se puede identificar porque los primeros 4 *bytes* contienen el valor *ASCII*: *INFO*. En la figura 3.1, este bloque está representado de color rojo.

Los datos almacenados en este bloque son: el tamaño del mismo, el tipo de codificación de las muestras, si el punto de bucle está activado o no, la frecuencia de muestreo, la posición donde comienza y termina el bucle (en el caso de estar desactivado, el primer valor es siempre 0 y el último es siempre la posición de la última muestra del canal) y una tabla de referencias a la información de cada canal.

Posterior a la tabla de referencias, se encuentran las propias entradas referenciadas por dicha tabla sobre la información de cada canal. Cada una de estas entradas contiene: una referencia al comienzo de las muestras del canal relativa a la posición del bloque de muestras (*DATA*) + 8 y una referencia a los parámetros de decodificación *ADPCM* relativa a la referencia anterior (en el caso de que no se use dicha codificación, el valor de la referencia es -1).

A continuación de las entradas de información a cada canal, se encuentran las diferentes entradas a los parámetros de decodificación *ADPCM* para cada canal (si existen). Cada entrada puede contener los parámetros *IMA-ADPCM* o los parámetros *DSP-ADPCM*, dependiendo del valor de identificación almacenado en la referencia.

En el caso de la entrada de parámetros *IMA-ADPCM*, contiene tanto el contexto para el punto inicial como para el punto de bucle. Cada contexto contiene la primera muestra en *PCM16* y el índice de 8 *bits* en la tabla de decodificación.

En el caso de la entrada de parámetros *DSP-ADPCM*, contiene los coeficientes de decodificación (16 valores de 16 *bits*) y el contexto tanto para el punto inicial como para el punto de bucle. Cada contexto contiene las dos primeras muestras en *PCM16* y un par índice-escala de 8 *bits* para el predictor.

3.3.4. Bloque de muestras (*DATA*)

Es el bloque final del archivo, que contiene las muestras de audio en si. Se puede identificar porque los primeros 4 *bytes* contienen el valor *ASCII*: *DATA*. En la figura 3.1, este bloque está representado de color amarillo.

El propósito de este bloque es almacenar las muestras de audio de cada canal, además del tamaño del propio bloque. Los canales están almacenados de forma secuencial, y la posición de inicio de cada uno de ellos viene determinada por la referencia encontrada en el bloque de información de audio (*INFO*).

Este bloque ocupa el resto del archivo, siendo el más grande de todos.

3.4 Generación de audio comprimido

Una vez entendido como funciona un archivo *BCWAV*, es necesario analizar la forma de generar estos archivos a partir de otros formatos de audio más comunes.

Esto se debe lograr mediante una herramienta de ordenador, preferiblemente de línea de comandos para que pueda ser incorporada en un archivo *GNU Make* [39] y escrita en C o C++ utilizando funciones del sistema UNIX, para que sea fácilmente portable a múltiples sistemas operativos.

Como se ha expuesto en la sección 2.6, la herramienta de ordenador *bannertool* es capaz de convertir archivos *WAV* y *OGG* a formato *BCWAV*, pero solo soporta las codificaciones *PCM8* y *PCM16*, y por otro lado, la herramienta *Citric Composer* solo puede generar archivos con codificación *DSP-ADPCM*.

Como se puede observar, ninguna de estas herramientas soporta todas las funcionalidades de los archivos *BCWAV*, por tanto, una solución posible es crear nuestra propia herramienta ampliando la funcionalidad de alguna de las anteriores.

Pese a que *Citric Composer* puede generar archivos con codificación *DSP-ADPCM* (siendo esta codificación la más difícil de implementar), no cumple con ninguno de los requisitos necesarios. En primer lugar, se trata de una herramienta de interfaz gráfica, haciendo imposible añadir su ejecución a un archivo *GNU Make*. Además, utiliza el sistema *.NET Framework* para su funcionamiento, dificultando la portabilidad a otros sistemas operativos. Tampoco es posible aprovechar su código de generación de muestras *DSP-ADPCM*, ya que está escrito en C#.

Por otro lado, la herramienta *bannertool* es una candidata perfecta. En primer lugar, utiliza el sistema UNIX de funciones y el repositorio *git* donde está alojada ya cuenta con una sistema de compilación basado en *GNU Make* que permite el soporte de diferentes sistemas operativos. En segundo lugar, se trata de una herramienta de linea de comandos y, al estar pensada para el desarrollo *homebrew*, está diseñada teniendo en mente que va a ser incorporado a un sistema *GNU Make* de compilación.

Por este motivo, se ha decidido utilizar como base la aplicación *bannertool* y ampliar su funcionalidad para soportar las codificaciones *ADPCM* y los puntos de bucle opcionales.

A continuación se analizarán las diferentes funcionalidades que son necesarias para la creación de un archivo *BCWAV*, y como están implementadas en *bannertool* (o si es necesario implementarlas o utilizar recursos externos).

3.4.1. Entrada de parámetros a través de la línea de comandos.

La herramienta de ordenador a implementar debe aceptar parámetros a través de la linea de comandos. Es necesario indicar el archivo de entrada, el archivo de salida, la codificación de salida y el punto inicial y final del bucle de forma opcional.

Tras ejecutar la orden *bannertool* en la linea de comandos, los parámetros disponibles para la generación de archivos *BCWAV* son los siguientes (figura 3.2).

```
makecwav - Creates a CWAV file from a WAV.
  -i/--input: WAV file to convert.
  -o/--output: File to output the created CWAV to.
  -l/--loop: Optional. Whether or not the audio should loop (false/true).
  -s/--loopstartframe: Optional. Sample frame to return to when looping.
  -f/--loopendframe: Optional. Sample frame to loop at.
```

Figura 3.2: Parámetros para la generación de *BCWAV* de *bannertool*.

Como se puede observar, *bannertool* no soporta seleccionar el formato de destino del archivo *BCWAV*, ni siquiera seleccionar codificación *PCM8* o *PCM16*, ya que utiliza la codificación especificada en el archivo *WAV* o *OGG* de entrada. Por este motivo, se deberá

añadir un nuevo parámetro para poder seleccionar entre las diferentes codificaciones soportadas por *BCWAV*.

Además, dado que *bannertool* es utilizado para la generación de metadatos incorporados en aplicaciones *.cia*, contiene más funcionalidades y parámetros que deberán ser eliminados para la creación de nuestra herramienta.

3.4.2. Procesado de archivos de entrada

Pese a no estar documentado en la descripción de los parámetros de *bannertool*, esta herramienta soporta tanto archivos *WAV* como *OGG*. Por un lado, la carga y procesamiento de *WAV* es código original escrito por el desarrollador de la aplicación, mientas que por otro lado, los archivos *OGG* son tratados mediante la librería *stb_vorbis* [40]. En ambos casos, es posible obtener la cantidad de canales de audio, cantidad de muestras de sonido por canal y las muestras de audio en sí, datos necesarios para la conversión a las diferentes codificaciones soportadas.

Una vez obtenidas las muestras, es necesario preprocesarlas para adaptarlas a un formato fácil de manipular similar al formato de destino. Pese a que tanto los formatos de entrada soportados como el formato *BCWAV* pueden almacenar más de un canal de audio, el orden de las muestras es diferente.

En el caso de los archivos de entrada, las muestras están almacenadas de forma intercalada: $muestra_{11}muestra_{21}...muestra_{i1}muestra_{12}muestra_{22}...muestra_{i2}...muestra_{ij}$ donde i es el número de canales y j el número de muestras por canal.

Sin embargo, en el caso del formato *BCWAV*, cada canal está almacenado de forma secuencial: $muestra_{11}muestra_{12}...muestra_{1j}muestra_{21}muestra_{22}...muestra_{2j}...muestra_{ij}$ donde i es el número de canales y j el número de muestras por canal.

En algunos casos, el desintercalado de las muestras se puede realizar a la vez que se convierten a la codificación de destino, mejorando la eficiencia del programa.

Hay que recordar también que ambos formatos de entrada soportan el almacenamiento tanto de muestras *PCM8* como de muestras *PCM16*, por lo que hay que tenerlo en cuenta a la hora de realizar la conversión.

3.4.3. Generación de muestras de audio PCM8

La generación de muestras de audio *PCM8* es un proceso trivial. Cada muestra ocupa 8 bits (1 byte), por lo que si el formato de entrada es *PCM8*, simplemente se han de desintercalar las muestras, mientras que si el formato de entrada es *PCM16* (16 bits), se deben des-intercalar las muestras y dividir el valor de cada una entre 256.

3.4.4. Generación de muestras de audio PCM16

Del mismo modo que las muestras *PCM8*, la generación de muestras *PCM16* también es un proceso trivial. En este caso, cada muestra ocupa 16 bits (2 bytes). Si el formato de entrada es *PCM16*, simplemente se deben des-intercalar las muestras, mientras que si el formato de entrada es *PCM8*, se han de des-intercalar las muestras y multiplicar el valor de cada una por 256.

3.4.5. Generación de muestras de audio IMA-ADPCM

A diferencia de las codificaciones *PCM8* y *PCM16*, las codificaciones tipo *ADPCM* requieren el uso de un algoritmo complejo, con tal de obtener la mejor secuencia de valores de muestras. La implementación de un algoritmo ADPCM desde cero cae fuera del ámbito de trabajo de este proyecto, y se ha decidido utilizar alguna herramienta externa.

En el caso de *IMA-ADPCM*, la disponibilidad de bibliotecas de funciones es mayor, ya que es un estándar no propietario diseñado por Interactive Media Association. Tras revisar diferentes implementaciones de un codificador *IMA-ADPCM*, se ha llegado a la conclusión de que la implementación realizada por David Bryant en su proyecto *adpcm-xq* es la mejor alternativa [41].

Esta biblioteca de funciones permite la generación de una secuencia de muestras *IMA-ADPCM* y sus parámetros de decodificación a partir de una secuencia *PCM16*. Además, cuenta con técnicas de reducción de ruido y análisis de muestras posteriores con tal de mejorar la calidad del audio generado.

Dado que *adpcm-xq* funciona con muestras *PCM16* de entrada, es necesario convertir los formatos de entrada *PCM8* previamente, además de realizar la des-intercalación de las muestras. También es necesario alinear la cantidad de muestras en un canal de audio, además de los puntos de bucle opcionales, a un múltiplo de 8 para el correcto funcionamiento del algoritmo. Esto se puede lograr insertando silencio al final de cada canal o copiando las muestras justo después del punto de bucle.

3.4.6. Generación de muestras de audio DSP-ADPCM

De la misma forma que la codificación *IMA-ADPCM*, la codificación *DSP-ADPCM* requiere el uso de un algoritmo complejo para obtener las muestras comprimidas. Por este motivo, también se ha decidido usar una herramienta externa.

Dado que la codificación *DSP-ADPCM* es propietaria de Nintendo, no hay gran variedad de bibliotecas de funciones disponibles. Sin embargo, es posible encontrar alguna desarrollada por la comunidad *homebrew*, como por ejemplo *gc-dspadpcm-encode*, desarrollada por Jack Andersen [42].

Esta biblioteca de funciones permite obtener los mejores coeficientes para la decodificación a partir de una secuencia de muestras *PCM16*, además de la codificación de bloques de 14 muestras *PCM16* a un bloque de 8 bytes en formato *DSP-ADPCM* (cada bloque consta de 14 muestras de 4 bits y una cabecera de 1 byte).

Además de la desintercalación y la conversión previa a *PCM16* de aquellos datos de entrada que estén en formato *PCM8*, es necesario alinear la cantidad de muestras en cada canal de audio a un múltiplo de 14, de la misma manera que el proceso de codificación *IMA-ADPCM*.

3.4.7. Construcción del archivo *BCWAV* de salida

La herramienta *bannertool* ya es capaz de construir un archivo *BCWAV* a partir de las muestras de audio. Sin embargo, debe modificarse para que utilice las muestras generadas en la codificación apropiada en vez de obtener las muestras directamente del archivo de entrada.

Por otro lado, es necesario añadir los parámetros adicionales para la decodificación de las muestras *ADPCM* en la cabecera del archivo *BCWAV*.

Tras indicar todos los valores correctos en la cabecera y añadir las muestras, los datos pueden escribirse al archivo especificado en la línea de comandos.

3.5 Reproducción de audio en la Nintendo 3DS

El siguiente proceso a analizar es la reproducción de los archivos *BCWAV* generados. Una forma conveniente de lograr esto es proveer al desarrollador una biblioteca de funciones, donde la carga y procesado de los archivos sea transparente. Además, se pueden proveer funciones para operaciones básicas sobre el audio, abstrayendo la comunicación con los servicios de audio.

Dado que el desarrollo de aplicaciones *homebrew* puede realizarse tanto en C como en C++, y se desea dar compatibilidad a la mayor cantidad de proyectos posible, la biblioteca de funciones será escrita en C.

Además, la biblioteca de funciones deberá soportar tanto el servicio de audio *CSND* como en servicio de audio *DSP*, con tal de que pueda ser utilizada independientemente del tipo de aplicación que se desee crear.

A continuación se analizarán los diferentes aspectos que esta biblioteca de funciones deberá cumplir.

3.5.1. Carga y procesamiento de archivos *BCWAV*

Una de las funciones más importantes de la biblioteca de funciones es la carga de archivos *BCWAV* desde un medio de almacenamiento. Como se ha mencionado en la sección 2.2.1, las aplicaciones de Nintendo 3DS pueden contar con un contenedor de archivos genérico para el almacenamiento de recursos audiovisuales (*romfs*). La biblioteca de funciones deberá ser capaz de cargar en memoria los archivos de audio de este contenedor, además de la tarjeta SD.

Por otro lado, es posible que ni el contenedor *romfs* ni la tarjeta SD estén disponibles. En este caso, será responsabilidad del usuario de posicionar el archivo *BCWAV* en memoria, y la biblioteca de funciones deberá ser capaz de aceptar la dirección de memoria donde el archivo ha sido cargado.

Una vez el archivo de audio ha sido cargado en memoria, es necesario procesar la cabecera de este para obtener la dirección de las muestras, además de los diferentes parámetros, como la cantidad de canales de audio, los puntos de bucle opcionales y los parámetros de decodificación *ADPCM*.

Tras obtener toda la información necesaria del archivo, esta se puede guardar en un "objeto" para que el desarrollador pueda realizar operaciones sobre él. Dichas operaciones serán posibles sobre cada uno de los canales de audio almacenados en el archivo *BCWAV*, que serán referenciados como "canal *BCWAV*" a partir de ahora.

3.5.2. Operaciones sobre los canales *BCWAV* cargados

Una vez cargados los canales en memoria, es posible realizar operaciones sobre ellos.

La operación mas importante, es la reproducción de un canal. Dado que los archivos *BCWAV* pueden almacenar una cantidad arbitraria de canales de audio, en el momento de iniciar la reproducción el desarrollador deberá indicar cuál de los canales desea reproducir, con la posibilidad de elegir dos canales de audio para la reproducción en estéreo.

Otra operación importante, es la detención de un canal en reproducción. De la misma manera que en la reproducción, se debe indicar qué canal de audio de un archivo se desea detener, o si se desean detener todos los canales.

Pese a que otra operación importante es la pausa y continuación de un canal, no es posible su implementación en la biblioteca de funciones al no estar dicha función disponible en el servicio *CSND*. Una solución a este problema será expuesto en la sección 3.5.4.

Otras funcionalidades adicionales, como por ejemplo el control del volumen, la velocidad de reproducción y la distribución entre el oído izquierdo o derecho (*panning*) también pueden ser añadidas, pero solo surtirán efecto a la hora de iniciar la reproducción del sonido. Esto es debido de nuevo a las limitaciones del servicio *CSND*.

La última operación necesaria, es la liberación de los recursos utilizados cuando se desee disponer del archivo de audio. Esto incluye la liberación del bloque de memoria utilizado dependiendo del método de carga del archivo.

3.5.3. Asignación de canales de audio *DSP* y *CSND*

Otro de los principales objetivos de esta biblioteca de funciones es proveer abstracción con la capa de servicios de la Nintendo 3DS. De este modo, el desarrollador debe solo preocuparse de la carga de archivos y realizar las operaciones sobre ellos. Sin embargo y como discutiremos a continuación, esto no será siempre posible.

Una solución para lograr la reproducción de un canal de audio almacenado en un archivo *BCWAV*, es asignarle un canal de audio libre del servicio *DSP* o *CSND*. Mediante las llamadas a los servicios, es posible iterar cada canal de audio y comprobar su estado de reproducción. Este proceso tiene un coste temporal lineal, pero es despreciable al contar los servicios con un número de canales pequeño.

Una vez encontrado un canal libre, será posible ajustar los parámetros de codificación *ADPCM*, indicar los puntos de bucle si los hay y finalmente, iniciar la reproducción. Por otro lado, la detención de un canal *BCWAV* será posible deteniendo el canal *DSP* o *CSND* asignado.

Pese a ser una solución sencilla, presenta dos inconvenientes. Por un lado, es importante saber que los servicios de audio no notifican la detención de un canal de audio al finalizar su reproducción. Esto es un problema en la siguiente secuencia de eventos (dado dos canales de audio *BCWAV* A y B, y el primer canal de audio *CSND* X):

1. Se inicia la reproducción de A, al que se le asigna el primer canal de audio libre X.
2. El canal de audio X finaliza la reproducción del sonido, pero no lo notifica a A.
3. Se inicia la reproducción de B, al que se le asigna el primer canal de audio libre, que de nuevo es X al haber finalizado anteriormente.
4. Se decide detener la reproducción de A.

Esta situación presenta un problema, ya que el canal *BCWAV* A desconoce que se ha terminado de reproducir, pero detecta que su canal de audio asignado X sigue en reproducción. Al intentar realizar la detención del canal X, esto resultaría erróneamente en la detención del canal *BCWAV* B.

Una solución disponible es iterar sobre todos los canales de audio *BCWAV* registrados y comprobar si su canal de audio *DSP* o *CSND* asignado ha cesado la reproducción,

desasignándolo si se da el caso. Este proceso se realizaría cada vez que se inicie la reproducción de un canal, teniendo un coste temporal lineal adicional proporcional a la cantidad de canales *BCWAV* cargados.

Por otro lado, el segundo inconveniente viene dado al intentar reproducir varias veces el mismo canal *BCWAV*. Si esto se intentase con la solución actual, la segunda reproducción asignaría un nuevo canal *DSP* o *CSND* al canal *BCWAV*, dejando "huérfano" el canal asignado anteriormente a este.

Una posible solución es mantener una lista de canales asignados a un canal *BCWAV*. Gracias a esta lista, se podría iniciar la reproducción de un mismo canal una cantidad arbitraria de veces. Sin embargo, mantener una lista de un tamaño arbitrario requiere mayor coste temporal y espacial, y tampoco tiene sentido iniciar la reproducción de un sonido a partir de una cierta cantidad.

Otra solución es utilizar un búfer cíclico, donde se van insertando los nuevos canales *DSP* o *CSND* asignados. De esta manera, el desarrollador puede indicar una cantidad máxima de veces que puede estar un canal *BCWAV* en reproducción, eliminando la necesidad de crear una lista de tamaño variante. En este caso, cuando el búfer esté lleno, y se desee reproducir otra vez el canal, se detendrá el canal más antiguo y se reemplazará con el nuevo.

3.5.4. Otras consideraciones sobre los servicios de audio

Como se ha mencionado en la sección 2.4.1, el servicio *CSND* presenta limitaciones en las operaciones posibles con sus canales de audio.

La primera limitación de *CSND* es la incapacidad de pausar un canal en reproducción. Por este motivo, se ha decidido no incluir la función de pausa directamente en la biblioteca de funciones. Sin embargo, esta decisión eliminaría la posibilidad de pausar un canal si se está utilizando el servicio *DSP*.

Mientras que la implementación *ndsp* de *libctru* posee una interfaz conocida como *ndspChn* (*ndsp channel*) que permite realizar múltiples operaciones sobre canales de audio *DSP*, no tiene sentido añadir toda esta funcionalidad a nuestra biblioteca de funciones ya que *CSND* no permite muchas de estas operaciones.

Como solución al problema de pausado y de las operaciones adicionales de *ndsp*, la biblioteca de funciones puede devolver los canales de audio *DSP* o *CSND* que han sido asignados tras la reproducción de un canal *BCWAV*, y el desarrollador podrá utilizar dichos canales retornados para llamar directamente a las funciones de la interfaz *ndspChn* y realizar las operaciones adicionales que desee.

La segunda limitación relacionada con *CSND* está relacionada con la implementación en *libctru* de las comunicaciones con dicho servicio. A diferencia de *ndsp*, la interfaz *csnd* carece de comunicación con el servicio *APT* [43], responsable de enviar las notificaciones de suspensión cuando se abre el menú HOME o se cierra la tapa de la consola. Como resultado, aquellos canales de audio *CSND* que estén en reproducción cuando se abra el menú HOME seguirán escuchándose en vez de detenerse.

Para solucionar este problema, la biblioteca de funciones deberá ser capaz de recibir las notificaciones enviadas por el servicio *APT*. Afortunadamente para nosotros, la interfaz *apt* implementada en *libctru* permite registrar una función por parte del desarrollador que será llamada cuando se reciba una notificación de *APT*. Gracias a esta funcionalidad, la biblioteca de funciones puede registrar una función que detendrá todos los canales *BCWAV* cuando se reciba la notificación de suspensión de la aplicación, en el caso de que se esté usando el servicio *CSND*.

Esta solución tiene una pequeña pega, ya que la interfaz *apt* de *libctru* solo está disponible en *applets* y aplicaciones convencionales, pero no en "*plugins 3GX*". Para dar compatibilidad a este último caso, es posible añadir una función para que el desarrollador pueda realizar las notificaciones *APT*, detectando la suspensión de la aplicación manualmente.

Un último punto a considerar es la diferencia entre direcciones de memoria físicas y direcciones de memoria virtuales [44]. Mientras que las aplicaciones utilizan direcciones de memoria virtuales pertenecientes a su propio proceso, los servicios *DSP* y *CSND* requieren el uso de direcciones de memoria físicas, que son reenviadas directamente al *hardware* de sonido. Esto es necesario ya que dicho *hardware* no cuenta con un *MMU* para realizar la conversión de direcciones virtuales a físicas por su cuenta.

Normalmente, la conversión de memoria virtual a física es fácil de realizar (se trata de una simple operación matemática), siempre y cuando la dirección a convertir esté posicionada en la región de memoria apropiada, conocida como *LINEAR memory*. Es por este motivo que el desarrollador deberá utilizar las funciones `linearAlloc()` y `linearFree()` para asegurar que los archivos *BCWAV* son cargados en la región de memoria apropiada.

Sin embargo, hay una excepción a esta conversión sencilla, y es cuando la biblioteca de funciones es utilizada dentro de un "*plugin 3GX*". Al tratarse de código y memoria inyectados en un proceso en ejecución, no puede usarse la región de memoria *LINEAR*, ya que está siendo utilizada por la propia aplicación anfitriona. Por este motivo, el cargador de "*plugins*" ofrece utilidades para realizar la conversión de cualquier dirección de memoria, sea o no de la región *LINEAR*.

Por tanto, la biblioteca de funciones deberá ofrecer la posibilidad al desarrollador de reemplazar la función para la conversión de direcciones de memoria por defecto con una propia, con tal de dar compatibilidad a los "*plugins 3GX*" (en este caso, se utilizaría la función ofrecida por el cargador de "*plugins*").

3.6 Aplicación de ejemplo de uso de la biblioteca de funciones

Al tratarse de una biblioteca de funciones completamente nueva, es una buena práctica otorgar al desarrollador una aplicación de ejemplo donde se muestren todas las funcionalidades.

Por una parte, se deberá incluir al menos un archivo *BCWAV* de cada tipo de codificación disponible, y algunos de ellos con la funcionalidad de punto de bucle activada. Al iniciar la aplicación el usuario deberá poder elegir entre el servicio *CSND* y *DSP* y a continuación, poder iniciar la reproducción y detención de los archivos disponibles. También sería conveniente mostrar aquellos canales del servicio elegido que están siendo utilizados.

De esta manera, cualquier desarrollador interesado podrá usar el código como base en su propia aplicación, o simplemente servirle para entender el funcionamiento de las funciones disponibles.

Por otro lado, esta aplicación también nos será útil para realizar casos de prueba de aquellas funciones que se vayan implementando durante el desarrollo de la biblioteca de funciones.

3.7 Distribución del conjunto de herramientas

Una vez desarrolladas la herramienta de ordenador y la biblioteca de funciones, es necesario distribuirlas a los desarrolladores para que puedan utilizarlas en sus proyectos.

Respecto a los nombres del *software* a desarrollar, se ha decidido llamar la herramienta de ordenador como ***cwavtool***, y la biblioteca de funciones como ***libcwav***.

Dado que casi todo el *software homebrew* para la Nintendo 3DS se trata de programas de código abierto, se ha decidido seguir el mismo principio y distribuir el código fuente de forma abierta y gratuita.

También es necesario elegir una licencia que otorgará libertades y restricciones a aquellos desarrolladores interesados. Tras analizar varias opciones, se ha decidido utilizar las siguientes licencias de código abierto:

- En el caso de *libcwav*, se ha decidido seguir los pasos de la biblioteca de funciones *libctru* y utilizar la licencia zLib [45]. Esta licencia permite al desarrollador usar y modificar libremente el código para cualquier finalidad, siempre y cuando no se represente incorrectamente el autor original del código, se indiquen las versiones modificadas como tal y no se elimine o modifique la licencia. También se elimina cualquier garantía y responsabilidad en caso de daños por parte de los autores del código.

- En el caso de *cwavtool*, se ha mantenido la licencia original de *bannertool*, siendo esta la licencia MIT [46]. Esta licencia permite al desarrollador usar y modificar libremente el código para cualquier finalidad, siempre y cuando no elimine la notificación de *copyright* de los autores originales. También se elimina cualquier garantía y responsabilidad en caso de daños por parte de los autores del código.

Una vez elegidas las licencias, es posible distribuir el código de *libcwav* y *cwavtool*. Una página web muy utilizada para distribuir software es GitHub, siendo esta muy apropiada para este proyecto, ya que se va a utilizar el sistema de control de versiones *git*. Además, la página web permite a otros desarrolladores reportar problemas encontrados o sugerir mejoras.

Además de distribuir el código fuente, es interesante distribuir una versión funcional, ya compilada, de *cwavtool* para los sistemas operativos más utilizados y una forma de instalar *libcwav* de forma sencilla. En ambos casos se puede recurrir a la creación de un paquete *pacman*, de la misma manera que devkitPro distribuye su *software homebrew* [47].

Una vez con las herramientas instaladas en el ordenador, el desarrollador interesado podrá incluir *libcwav* en su proyecto añadiéndola al archivo *GNU Make* para ser enlazada con el resto del código y generar los archivos con *cwavtool*, que serán añadidos posteriormente en el *romfs* de la aplicación o en la tarjeta SD.

3.8 Análisis de recursos

Para finalizar la sección de análisis del problema, es necesario estimar qué recursos temporales y materiales serán necesarios para la implementación de la solución.

En el caso de los recursos temporales, dada la naturaleza del proyecto y los conocimientos previos sobre el desarrollo en un entorno *homebrew* para la Nintendo 3DS, es difícil calcular cuánto tiempo será requerido para la investigación y entendimiento de los diferentes puntos expuestos en esta sección de análisis.

Por otro lado, mientras que la funcionalidad principal de *cwavtool* y *libcwav* se realizará de forma independiente, es muy probable que se necesite realizar un desarrollo en paralelo de ciertos aspectos de la herramienta de ordenador y de la biblioteca de funciones.

También deberá incluirse el tiempo necesario para la implementación de una aplicación de pruebas para la Nintendo 3DS que verifique el correcto funcionamiento de las herramientas.

Dicho esto, se estima que el tiempo total de investigación y desarrollo de la herramienta de ordenador será de 100 horas, mientras que el tiempo dedicado a los mismos procesos para la biblioteca de funciones será de otras 125 horas. Utilizando 50 horas para implementar la aplicación de ejemplo y dejando un colchón de 50 horas para realizar las pruebas, la documentación y la distribución del código, este proyecto en su totalidad requerirá de 325 horas de trabajo. Además, la redacción de esta memoria se estima en 35 horas adicionales.

Respecto a los recursos materiales, es necesario un ordenador para realizar la implementación de tanto la herramienta como de la biblioteca de funciones. El sistema operativo es irrelevante, ya que el entorno de desarrollo soporta los sistemas operativos más utilizados (Windows, la diferentes distribuciones de Linux y Mac).

Pese a que existen emuladores de Nintendo 3DS, como el emulador *Citra* [48], es importante comprobar el funcionamiento de las herramientas en *hardware* real, por lo que también es necesaria una consola Nintendo 3DS capaz de ejecutar los diferentes tipos de aplicaciones *homebrew* soportados por la biblioteca de funciones. Además, dicho emulador, que es el más avanzado de todos los disponibles, no soporta el servicio de audio *CSND*, por lo que no sería posible comprobar el soporte de dicho servicio. Por este motivo, la capacidad de procesamiento del ordenador también es irrelevante, ya que no se requiere el cálculo de operaciones complejas en ningún momento.

CAPÍTULO 4
cwavtool: Diseño e implementación

En este capítulo se describirá el proceso de diseño e implementación de la herramienta para ordenador *cwavtool*. Como se ha comentado en el capítulo de análisis, esta herramienta será una modificación de la herramienta de ordenador existente *bannertool*, escrita en C++.

Para poder compilar esta herramienta en Windows, es necesaria la instalación del entorno de desarrollo *Mingw-w64* [49]. Gracias a la inclusión del conjunto de *scripts buildtools* creado por el mismo desarrollador que *bannertool* [50], es posible compilar la aplicación fácilmente utilizando *GNU Make*.

4.1 Compilación inicial de *bannertool*

El primer paso es realizar un *fork* (bifurcación) del repositorio *bannertool* encontrado en GitHub. Una vez clonado este nuevo repositorio en la máquina local, se procede a abrir una ventana de comandos para ejecutar la orden `make` que compilará la aplicación para nuestro sistema operativo.

```
D:\Documentos\3ds\cwavtool>make
fatal: No tags can describe '5f297e49c8c72610caedd615958b960ec2bb0ab3'.
Try --always, or create some tags.
Building for NATIVE32...
build/windows-i686/source/pc/stb_image.o
build/windows-i686/source/pc/stb_vorbis.o
build/windows-i686/source/cmd.o
build/windows-i686/source/main.o
build/windows-i686/source/3ds/cbmd.o
build/windows-i686/source/3ds/cwav.o
build/windows-i686/source/3ds/lz11.o
build/windows-i686/source/pc/wav.o
output/windows-i686/bannertool.exe
output/bannertool.zip
/bin/sh: línea 2: zip: orden no encontrada
make: *** [buildtools/make_base:689: output/bannertool.zip] Error 127
```

Figura 4.1: Compilación de *bannertool* sin modificaciones.

En la figura 4.1 se puede observar el proceso de compilación de *bannertool*. Pese haber sucedido dos errores, estos están relacionados con la incorrecta configuración del reposi-

torio *git* y por no contar el sistema con la orden *zip*. Ninguno de estos errores es relevante para la compilación de prueba, el archivo ejecutable (*.exe*) ha sido generado en el subdirectorio *output* y funciona sin problemas.

4.2 Eliminación de código fuente no necesario

El primer paso que conlleva modificación de código es la eliminación de aquellas funciones de *bannertool* que no son necesarias en *cwavtool*. Esto es todo aquel trozo de código que no esté relacionado con la generación de *BCWAV*.

El archivo `cmd.cpp` contiene el código relacionado con el procesamiento de los parámetros de línea de comandos, además de diversas funciones para la lectura y generación de diversos archivos. Todas las funciones excepto las involucradas con la generación de *BCWAV* (principalmente `convert_to_cwav()` y `cmd_make_cwav()`) han sido eliminadas.

Finalmente, se han eliminado todos los archivos que no están relacionados con la generación de *BCWAV* y lectura de archivos de audio.

4.3 Modificación de parámetros de línea de comandos

A continuación es necesario editar la función para el procesado de las opciones de línea de comandos `cmd_process_command()`, además de las diversas funciones para mostrar los diversos parámetros de entrada disponibles.

Los parámetros de entrada necesarios para la generación de *BCWAV* son los siguientes:

- ***-i / --input***: Archivo de entrada *WAV* o *OGG*.
- ***-o / --output***: Archivo de salida *BCWAV*.
- ***-e / --encoding***: Codificación de audio de salida.
- ***-ls / --loopstartframe***: Número de muestra donde comienza el bucle.
- ***-le / --loopendframe***: Número de muestra donde finaliza el bucle.

Estos parámetros están sujetos a las siguientes restricciones:

- Tanto *-i* como *-o* son argumentos obligatorios, y deben ser un *path* válido.
- El parámetro *-e* debe ser uno de los siguientes valores: *pcm8*, *pcm16*, *imaadpcm* o *dspadpcm*. En el caso de no especificarse, por defecto se usa *pcm16*.
- Tanto *-ls* como *-le* son opcionales, si no están presentes no se utilizará la funcionalidad de bucle. En el caso que se desee utilizar la funcionalidad de bucle, deben estar ambos presentes e indicar un número de muestra válido. Si se indica "*end*" en *-le*, el punto final de bucle será la última muestra.

Después de realizar las modificaciones y compilar de nuevo el programa, al ejecutar la orden *cwavtool*, se muestran los nuevos parámetros, como se puede observar en la figura 4.2.

```
D:\Documentos\3ds\cwavtool>output\windows-i686\cwavtool.exe
cwavtool v1.0.0
Usage: output\windows-i686\cwavtool.exe <args>
Available arguments:
  -i/--input: WAV/OGG input file.
  -o/--output: CWAV output file.
  -e/--encoding: Optional. Encoding of the created CWAV (pcm8/pcm16/imaadpcm/dspadpcm).
  -ls/--loopstartframe: Optional. Sample to return to when looping.
  -le/--loopendframe: Optional. Sample to loop at or "end".
```

Figura 4.2: Ejecución y parámetros de *cwavtool*.

4.4 Lectura de archivos de entrada

Una vez procesados los parámetros de entrada a través de la linea de comandos, se puede proceder a cargar el archivo de audio de entrada en memoria para obtener las muestras. Este proceso se realiza en la función `convert_to_cwav()`, que tiene como argumentos los parámetros de entrada, entre ellos, el archivo de audio *WAV* o *OGG*.

Esta función está implementada en *bannertool*, pero es necesario entender como funciona y ampliarla para las nuevas funcionalidades que se desean añadir. Para poder pasar información a la siguiente función, hay presente un *struct*[1] para almacenar los diferentes parámetros de entrada y características del audio. Como ampliación al *struct*, se han añadido campos relacionados con la codificación de destino del audio (que serán explicados en su sección correspondiente). En la figura 4.3 se puede observar el *struct* final, cuyo nombre es `CWAV`.

```
typedef struct {
    u32 channels;  // Cantidad de canales en el archivo de entrada.
    u32 sampleRate; // Muestras por segundo en el archivo de entrada.
    u32 bitsPerSample; // Bits por muestra (8 para PCM8 y 16 para PCM16).

    u32 encoding; // Codificación del archivo de salida.

    IMAADPCMInfo* imainfos; // Parámetros de decodificación IMA-ADPCM (inicio).
    IMAADPCMInfo* imainfosloop; // Parámetros de decodificación IMA-ADPCM (bucle).

    DSPADPCMInfo* dspinfos; // Parámetros de decodificación DSP-ADPCM (inicio y bucle).

    bool loop; // Bucle opcional activado o no.
    u32 loopStartFrame; // Muestra de inicio del bucle.
    u32 loopEndFrame; // Muestra final del bucle.

    u32 dataSize; // Tamaño de la secuencia de muestas (en bytes).
    void* data; // Puntero a la secuencia de muestras.
} CWAV;
```

Figura 4.3: Definición del *struct* para intercambio de datos entre funciones en *cwavtool*.

Una vez rellenado el *struct*, se llama a la función `cwav_build()` que se encargará de construir el archivo *BCWAV* en memoria.

Es importante destacar que el campo `loopEndFrame` del *struct* marcará la última muestra a convertir, se utilice el punto de bucle opcional o no. De esta manera, todas las muestras a partir de este punto serán ignoradas (ya que en el caso de bucle, nunca se repro-

[1] Estructura de datos de C/C++ que permite almacenar datos de diferente tipo, que suelen estar relacionados. Similar en concepto a una clase en programación orientada a objetos.

ducirían, y en el caso de no haber bucle, este valor siempre equivale a la posición de la última muestra).

4.5 Preprocesamiento y codificación a PCM8 y PCM16

El primer paso a realizar tras cargar las muestras en memoria es la conversión a la codificación de destino. Para poder convertir entre codificaciones, se ha implementado la función `cwav_convert_target_format()`, encargada de convertir la codificación del archivo de entrada a la de destino.

Como se comenta en la sección 3.4.2, las muestras en los archivos *WAV* y *OGG* están intercaladas. Por este motivo, es necesario realizar la desintercalación independientemente de la codificación de destino. También es importante recordar que para la conversión a las codificaciones *ADPCM*, es necesario convertir las muestras de audio a *PCM16* previamente.

La comprobaciones mostradas en la tabla 4.1 se han realizado para llamar a la función correspondiente de conversión de audio.

Tabla 4.1: Funciones a utilizar para cambiar el formato de audio.
Para *IMA-ADPCM* y *DSP-ADPCM* se utiliza como codificación intermedia *PCM16*.

Cod. entrada	Cod. salida	Función de conversión
PCM8	*PCM8*	`cwav_convert_pcm8_pcm8()`
PCM16	*PCM8*	`cwav_convert_pcm16_pcm8()`
PCM16	*PCM16*	`cwav_convert_pcm16_pcm16()`
PCM8	*PCM16*	`cwav_convert_pcm8_pcm16()`
PCM8	*IMA-ADPCM*	`cwav_convert_pcm8_pcm16()` y después `cwav_convert_pcm16_imaadpcm()`
PCM16	*IMA-ADPCM*	`cwav_convert_pcm16_pcm16()` y después `cwav_convert_pcm16_imaadpcm()`
PCM8	*DSP-ADPCM*	`cwav_convert_pcm8_pcm16()` y después `cwav_convert_pcm16_dspadpcm()`
PCM8	*DSP-ADPCM*	`cwav_convert_pcm16_pcm16()` y después `cwav_convert_pcm16_dspadpcm()`

Todas las funciones de conversión usadas en la tabla 4.1 toman como argumento un *struct* `CWAV` y realizan operaciones sobre las muestras. Los campos `dataSize` y `data` son actualizados con el puntero a las nuevas muestras y el nuevo tamaño en *bytes*.

Es importante recordar que las codificaciones *ADPCM* requieren que la cantidad de muestras sea un múltiplo de cierto número. Para poder realizar el proceso de alineación, se ha creado la función `cwav_align_pcm16()` que toma como argumento el número al que se deben alinear las muestras.

A continuación se realizará una descripción del funcionamiento de cada función de conversión.

4.5.1. Funciones de conversión *PCM8* a *PCM8* y *PCM16* a *PCM16*

Tanto la función `cwav_convert_pcm8_pcm8()` como `cwav_convert_pcm16_pcm16()` tienen los mismos objetivos, siendo estos la desintercalación de las muestras de entrada hasta el punto de bucle final (`loopEndFrame`). La única diferencia entre la versión de *PCM8* y la versión de *PCM16* es el tipo de dato usado para las muestras (`s8` para *PCM8* y `s16` para *PCM16*).

El primer paso realizado en estas funciones es la creación de un bloque de memoria para copiar las nuevas muestras desintercaladas. Esto se consigue llamando a la función `malloc()` con el tamaño (en *bytes*) de la cantidad de muestras por cantidad de canales (`loopEndFrame * channels`). En el caso de la versión de *PCM16*, se multiplica este valor por dos, ya que cada muestra ocupa dos *bytes*. El resultado de `malloc()` se guarda en la variable temporal `newData`.

En la figura 4.4 se puede observar el algoritmo de desintercalación. Para recapitular, recordemos que es necesario pasar las muestras de la forma intercalada: $muestra_{11}muestra_{21}...muestra_{i1}muestra_{12}muestra_{22}...muestra_{i2}...muestra_{ij}$ a la forma secuencial: $muestra_{11}muestra_{12}...muestra_{1j}muestra_{21}muestra_{22}...muestra_{2j}...muestra_{ij}$ donde i es el número de canales y j el número de muestras por canal.

Primero se inicializa el canal actual a 0, a continuación, se realiza un bucle desde la primera muestra hasta la última, siendo esta `loopEndFrame * channels`. Dentro del bucle, se obtiene la muestra actual (línea 4) y se almacena en el nuevo bloque de memoria (línea 5). Para obtener la nueva posición, primero se obtiene la posición de la primera muestra del canal correspondiente (`samplesPerChannel * currChannel`) y posteriormente se añade la posición de dicha muestra dentro del propio canal (`i / cwav->channels`). Finalmente, se incrementa el valor del canal actual y se reinicia a 0 en el caso de ser mayor o igual a la cantidad de canales.

```
1. u32 currChannel = 0;
2. u32 samplesPerChannel = cwav->loopEndFrame;
3. for (u32 i = 0; i < cwav->loopEndFrame * cwav->channels; i++) {
4.     s8 sample = ((s8*)(cwav->data))[i];
5.     newData[samplesPerChannel * currChannel + (i / cwav->channels)] = sample;
6.     currChannel++;
7.     if (currChannel >= cwav->channels)
8.         currChannel = 0;
9. }
```

Figura 4.4: Algoritmo de desintercalación de muestras (*PCM8*).

El código del algoritmo de desintercalación mostrado en la figura 4.4 es el utilizado en la función de conversión *PCM8* a *PCM8*. Para la versión de *PCM16* a *PCM16* simplemente cambia el tipo de dato utilizado en la línea 4 de `s8` a `s16`.

Una vez realizada la desintercalación, se libera el bloque de memoria original con `free()` y se asigna el nuevo bloque (`newData`) al campo `data` del struct `CWAV`. Finalmente, se actualiza el tamaño total de las muestras (campo `dataSize`).

4.5.2. Funciones de conversión *PCM8* a *PCM16* y *PCM16* a *PCM8*

Las siguientes funciones a implementar son la funciones de conversión *PCM8* a *PCM16* (`cwav_convert_pcm8_pcm16()`) y *PCM16* a *PCM8* (`cwav_convert_pcm16_pcm8()`). Estas funciones tienen exactamente la misma estructura que las explicadas en la sección anterior. La única diferencia es una línea adicional para cambiar el tamaño de cada muestra, y el cálculo del tamaño del bloque de memoria a reservar.

En el caso de la función de conversión *PCM8* a *PCM16*, el tamaño de bloque de memoria de destino es `loopEndFrame * channels * 2`. Para convertir cada muestra individual, se ha añadido `sample *= 256;` entre las líneas 4 y 5 en el algoritmo de desintercalación (figura 4.4).

En el caso de la función de conversión *PCM16* a *PCM8*, el tamaño de bloque de memoria de destino es `loopEndFrame * channels`. Para convertir cada muestra individual, se ha añadido `sample /= 256;` entre las líneas 4 y 5 en el algoritmo de desintercalación (figura 4.4).

4.5.3. Función de alineación de *PCM16*

Para poder realizar la conversión a *ADPCM*, es necesario primero asegurar que tanto la cantidad de muestras como los puntos de bucle estén alineados a un múltiplo de un número (8 en el caso de *IMA-ADPCM* y 14 en el caso de *DSP-ADPCM*).

En el caso de no estar activado el punto de bucle, alinear las muestras a un múltiplo de un valor es un proceso trivial, simplemente se debe insertar muestras con valor 0 (silencio) al final de cada canal, hasta alcanzar el siguiente múltiplo.

En el caso de estar activado el punto de bucle, el proceso requiere un poco más de trabajo, ya que se necesita copiar las muestras desde punto de bucle inicial al punto de bucle final, hasta que el punto de bucle final esté alineado. Sin embargo, esto no asegura que el punto de bucle inicial también esté alineado.

Para alinear el punto de bucle inicial, se ha aprovechado el hecho de que mover el punto de bucle una cantidad pequeña de muestras supone una diferencia inaudible, por lo que simplemente también se ha al siguiente múltiplo del valor sin ninguna operación adicional.

El algoritmo implementado en la función `cwav_align_pcm16()` procede de la siguiente manera:

1. Se obtiene la cantidad de muestras por canal y se calcula la diferencia con el siguiente múltiplo del valor requerido. El resultado se almacena en `remainingSamples`.

2. En el caso de que el valor de `remainingSamples` sea 0 y la cantidad de muestras por canal equivalga a `loopEndFrame`, se salta al paso 5. (La segunda condición siempre es verdadera, ya que las funciones de conversión a *PCM16* previas aseguran que este hecho sea cierto.)

3. Se reserva un nuevo bloque de memoria teniendo en cuenta el tamaño de las muestras al ser alineadas al siguiente múltiplo del valor requerido.

4. Para cada canal de audio, se realizan las siguientes acciones:

   a) Se copian las muestras desde la posición original (`samples * currChannel`) a la nueva posición (`(loopEndFrame + remainingsamples) * currChannel`) con una longitud de `loopEndFrame`.

   b) En el caso de no estar activado el bucle opcional, se rellena el final del canal con `remainingsamples` muestras de silencio.

   c) En el caso de estar activado el bucle opcional, se copian las muestras desde `loopStartFrame + j` hasta `loopEndFrame + j` para $0 \leq j < remainingsamples$.

5. Se traslada `loopEndFrame` la cantidad `remainingsamples` de muestras para reflejar el nuevo final del canal (también se traslada `loopStartFrame` la misma cantidad en el caso de estar activado el punto de bucle opcional).

6. Se alinea el punto de bucle inicial (`loopStartFrame`) al siguiente múltiplo del valor requerido.

Una vez realizadas todas las operaciones, se libera el bloque de memoria original y se asigna el nuevo bloque con las muestras alineadas al campo `data` del *struct*. También se actualiza el campo `dataSize` con el nuevo valor.

Debido a las operaciones realizadas, se puede dar el caso de que los valores de rango de bucle (`loopStartFrame` y `loopEndFrame`) terminen siendo el mismo. Si se da este caso, se aborta la conversión y se muestra un mensaje de error al usuario.

4.5.4. Función de conversión *PCM16* a *IMA-ADPCM*

Como se ha mencionado en la sección 3.4.5, el proceso de codificación *IMA-ADPCM* requiere de un algoritmo complejo, por lo que se ha decidido utilizar la biblioteca de funciones externa *adpcm-xq*, que incluye diversas técnicas para mejorar la calidad de audio de salida.

En primer lugar, se ha creado un subdirectorio en el proyecto de *cwavtool* donde se han copiado los archivos `adpcm-lib.h` y `adpcm-lib.c`. La biblioteca de funciones expone 4 funciones al desarrollador, aunque para este proyecto solo son necesarias 3 de ellas:

1. `adpcm_create_context()`: Toma como argumentos la cantidad de canales, la cantidad de muestras posteriores a analizar, el tipo de reducción de ruido y los deltas iniciales para el predictor. Como valor de retorno, devuelve un puntero al contexto creado.

2. `adpcm_encode_block()`: Toma como argumentos el contexto creado con la función anterior, el bloque de memoria de muestras de salida (puntero y tamaño) y el bloque de memoria de muestras de entrada (puntero y tamaño). Esta función es la encargada de la conversión de muestras *PCM16* a muestras *IMA-ADPCM*.

3. `adpcm_free_context()`: Toma como argumento el contexto a liberar.

Sin embargo, las funciones expuestas al desarrollador necesitan ser modificadas o ampliadas para el formato de codificación usado en los archivos *BCWAV*.

Por un lado, el cálculo de las deltas iniciales no está añadido en la biblioteca de funciones, pero si que existe un programa de ejemplo proporcionado por el mismo desarrollador de donde se puede extraer una función para calcular dichas deltas, que ha sido añadido a la biblioteca de funciones con nombre `adpcm_calculate_initial_deltas()`. Esta función toma como entrada la cantidad de canales y el bloque de memoria de muestras *PCM16* (puntero y tamaño) y devuelve los deltas iniciales.

Por otro lado, la biblioteca de funciones añade los parámetros de decodificación *IMA-ADPCM* al principio de cada canal en las muestras de salida. Sin embargo, en los archivos *BCWAV* estos están almacenados en la cabecera del archivo. Por tanto, se ha modificado la función `adpcm_encode_block()` para que almacene los parámetros en un *struct* nuevo llamado `IMAADPCMInfo` en vez de añadirlos al bloque de memoria de muestras de salida.

Cabe destacar que pese a que las funciones disponibles toman un argumento en el que se especifica el número de canales, el algoritmo implementado asume que las muestras están intercaladas y produce las muestras de salida también intercaladas. Por este motivo, se debe convertir cada canal individualmente (el argumento de número de canales siempre será 1).

Una vez lista la biblioteca de funciones, se puede comenzar la implementación del código de conversión en la función `cwav_convert_pcm16_imaadpcm()`. En primer lugar, se llama a la función de alineación implementada anteriormente para asegurar que las muestras estén alineadas a un múltiplo de 8. A continuación, se reserva un bloque de

memoria con tamaño `dataSize / 4` (esto es debido a que cada muestra ocupa ahora medio *byte* en vez de dos).

Posteriormente y para cada canal, se calculan las deltas iniciales y se crea un contexto nuevo, con `adpcm_calculate_initial_deltas()` y `adpcm_create_context()` respectivamente. Una vez creado el contexto es necesario dividir el canal dependiendo si está activada la función de bucle, ya que se necesitan parámetros de decodificación independientes para la primera muestra y la muestra donde comienza el punto de bucle.

En el caso de que no esté activado el bucle, se llama a la función `adpcm_encode_block()` desde la primera muestra a la última, y se guardan los parámetros de decodificación salientes. Además, estos parámetros se copian a los parámetros de la muestra del punto de bucle por el mero hecho de que este es el comportamiento de los archivos *BCWAV* oficiales (pese a que a la hora de la reproducción, no se utilizan).

En el caso de que esté activado el bucle, se divide el canal de audio en dos bloques. El primer bloque contiene las muestras $[0,\ loopStartFrame)$ y el segundo bloque las muestras $[loopStartFrame,\ loopEndFrame)$. Estos bloques son posteriormente codificados mediante dos llamadas a la función `adpcm_encode_block()`. De este modo, es posible generar parámetros de decodificación tanto para el punto inicial como para el punto de bucle.

Una vez convertidas las muestras, se pueden liberar los contextos creados con la función `adpcm_free_context()` y, finalmente, se actualizan los campos `data`, `dataSize`, `imainfos` y `imainfosloop` del *struct* `CWAV` con los nuevos datos generados.

4.5.5. Función de conversión *PCM16* a *DSP-ADPCM*

Como se menciona en la sección 3.4.6, la codificación *DSP-ADPCM* requiere el uso de un algoritmo complejo, por lo que se ha utilizado la biblioteca de funciones externa *gc-dspadpcm-encode*.

Esta biblioteca de funciones consta solamente de un archivo de código C llamado *grok.c*, que ha sido copiado a un nuevo subdirectorio en el proyecto de *cwavtool*. Dado que *gc-dspadpcm-encode* carece de un archivo de cabecera (*.h*), se ha creado uno donde se han añadido las dos funciones relevantes para la codificación:

1. `DSPCorrelateCoefs()`: A partir de un bloque de memoria de muestras *PCM16* (puntero y tamaño) obtiene los 16 coeficientes de decodificación (cada uno siendo un valor de 16 *bits*).
2. `DSPEncodeFrame()`: Toma como entrada un bloque de 16 muestras *PCM16* y los coeficientes obtenidos anteriormente para generar un paquete de 14 muestras *DSP-ADPCM* de 8 *bytes* (14 muestras de 4 *bits* y una cabecera de 1 *byte*). Las dos primeras muestras del bloque de entrada deben ser las dos últimas muestras del bloque anterior. En el caso de ser el primer paquete, estas tienen un valor de 0.

A diferencia de la biblioteca de funciones de codificación *IMA-ADPCM*, no es necesario la modificación de ninguna característica en este caso. Sin embargo, si que es necesario la creación de un *struct* para el almacenamiento de los parámetros de decodificación, nombrado `DSPADPCMInfo`. Este *struct* almacenará los coeficientes globales y los contextos para el punto inicial y el punto de bucle.

Tras tener preparada la biblioteca de funciones externa, puede dar comienzo la implementación de la función de codificación `cwav_convert_pcm16_dspadpcm()`. En primer lugar, es necesario utilizar la función de alineación implementada anteriormente para

asegurar que las muestras estén alineadas a un múltiplo de 14. Posteriormente, se reserva un bloque de memoria con el tamaño final de las muestras *DSP-ADPCM*. El tamaño puede calcularse de la siguiente forma: $totalBytes = (muestrasTotalesDsp/14) * 8$.

Posteriormente para cada canal se realizan los siguientes pasos:

1. Se obtienen los coeficientes del canal utilizando `DSPCorrelateCoefs()` y se guardan en la variable `coefs`.
2. Se inicializa tanto el bloque de entrada de 16 muestras (llamado `convSamps`) como las dos primeras muestras del contexto de decodificación (llamadas `hist1` y `hist2`) a 0.
3. Se divide el canal de entrada en paquetes de 14 muestras. Para cada uno de estos paquetes:
   a) Se copian las muestras del paquete de entrada a partir de la tercera posición del bloque `convSamps`. De esta manera se mantienen las dos últimas muestras del bloque anterior, copiadas en el paso *e)*, en las dos primeras posiciones.
   b) Se realiza la llamada a la función `DSPEncodeFrame()` con el bloque de entrada `convSamps` y los coeficientes obtenidos anteriormente. La función retorna el paquete de salida *DSP-ADPCM* de 8 *bytes* llamado `block`.
   c) En el caso de que el bloque actual sea el bloque inicial, se actualiza el campo `index` del contexto de decodificación con el primer *byte* el paquete de salida `block`.
   d) En el caso de que el bloque actual sea el bloque de punto de bucle, se actualiza el campo `index` del contexto de decodificación con el primer *byte* el paquete de salida `block` y los campos `hist1` y `hist2` con las dos primeras muestras del bloque de entrada `convSamps`.
   e) Se copian las dos últimas muestras del bloque de entrada `convSamps` (que han sido modificadas por la llamada a la función de codificación) a las primeras dos muestras del mismo bloque.
   f) Se añade el paquete *DSP-ADPCM* `block` al bloque de memoria de muestras de salida.

En este punto, las muestras ya han sido convertidas al formato *DSP-ADPCM* y solo falta actualizar los campos `data`, `datasize` y `dspinfos` del *struct* `CWAV` con los nuevos datos generados.

4.6 Generación del archivo *BCWAV* de salida

Con las muestras convertidas a la codificación apropiada, podemos proceder a construir el archivo *BCWAV* de salida. Este proceso ya lo realiza la función `cwav_build()` de *bannertool*, sin embargo, debe ser modificada para que guarde los parámetros de decodificación *ADPCM* y no intente volver a desintercalar las muestras de audio.

El primer paso es entender el funcionamiento de la función conforme está presente en *bannertool*. Al principio de ella, se calcula el espacio total que va a ocupar el archivo para poder reservar la cantidad de memoria apropiada. Este espacio es la suma de los bloques *CWAV*, *INFO* y *DATA*.

Mientras que el el tamaño del bloque *CWAV* es estático, el tamaño del bloque *INFO* depende de la cantidad de canales y la codificación de las muestras. Es aquí donde aparece el primer cambio: se ha añadido al tamaño del bloque el tamaño de los parámetros

ADPCM. El tamaño de cada bloque principal es después alineado al siguiente múltiplo de 32.

A continuación, se rellenan los valores estáticos de los bloques *CWAV* y *INFO* a partir de los datos del *struct* `CWAV`. Posteriormente, se crean las entradas de información para cada canal de audio.

La segunda modificación, es la incorporación de las entradas para los parámetros de decodificación *ADPCM*, respetando las reglas que sigue cada *offset* de cada referencia descritas en la sección 3.3. De nuevo, todos estos parámetros se obtienen de los datos del *struct* `CWAV`, que ha sido rellenado anteriormente.

Para finalizar, se ha reemplazado la desintercalación que se realizaba para almacenar las muestras en el bloque *DATA* con un simple copiado de memoria, aprovechando el hecho de que las funciones de conversión ya generan las muestras en el formato correcto.

Una vez el bloque de memoria se ha rellenado con el archivo *BCWAV*, este es escrito al archivo de salida especificado en los parámetros de linea de comandos y la aplicación *cwavtool* termina su ejecución, liberando todos los recursos de memoria utilizados.

4.7 Ejemplo de uso y pruebas

Una vez compilado el programa, obtenemos el archivo ejecutable *cwavtool.exe*. Para poder comprobar el correcto funcionamiento del programa, se ha utilizado un archivo de sonido *WAV* obtenido de *freesound.org* con licencia *CC0*, es decir, de dominio público.

Este archivo se trata de un pequeño bucle y, utilizando el editor de audio *Audacity* [51], se ha encontrado la posición de bucles inicial y final (22979 y 336696 respectivamente).

En la figura 4.5 se puede observar el resultado de ejecutar la herramienta *cwavtool* con los puntos de bucle especificados y codificación *DSP-ADPCM*, produciendo el archivo de salida *loop.bcwav*.

```
D:\Documentos\3ds\cwavtool>output\windows-i686\cwavtool.exe
-i loop.wav -o loop.bcwav -e dspadpcm -ls 22979 -le 336696
Created CWAV "loop.bcwav".

D:\Documentos\3dshack\cwavtool>
```

Figura 4.5: Ejecución de *cwavtool* con un archivo de prueba.

Posteriormente, se ha utilizado el reproductor *foobar2000* junto a *vgmstream* para comprobar que el archivo funcione correctamente. Como se puede observar en la figura 4.6, el programa consigue abrir el archivo correctamente y es reproducido sin problemas, pudiendo comprobar también el correcto funcionamiento del punto de bucle.

A continuación, se ha procedido a crear el mismo archivo en las diferentes codificaciones disponibles, comprobando que es posible reproducirlo en todos los casos. Respecto al tamaño, se ha podido observar que mientras que el mismo archivo de entrada ocupa 658 *kilobytes* al codificarse en *PCM16*, este ocupa 189 *kilobytes* al codificarse en *DSP-ADPCM*. Esto supone un tamaño 28.72 % menor, tal y como se esperaba conforme a lo descrito en la sección 2.3.1 (teniendo en cuenta que la cabecera del archivo no es comprimida).

Para finalizar, se han probado otros archivos de *freesound.org* con mayor cantidad de canales de audio, y en todos los casos, ha sido posible su conversión a *BCWAV* con la herramienta sin ningún problema.

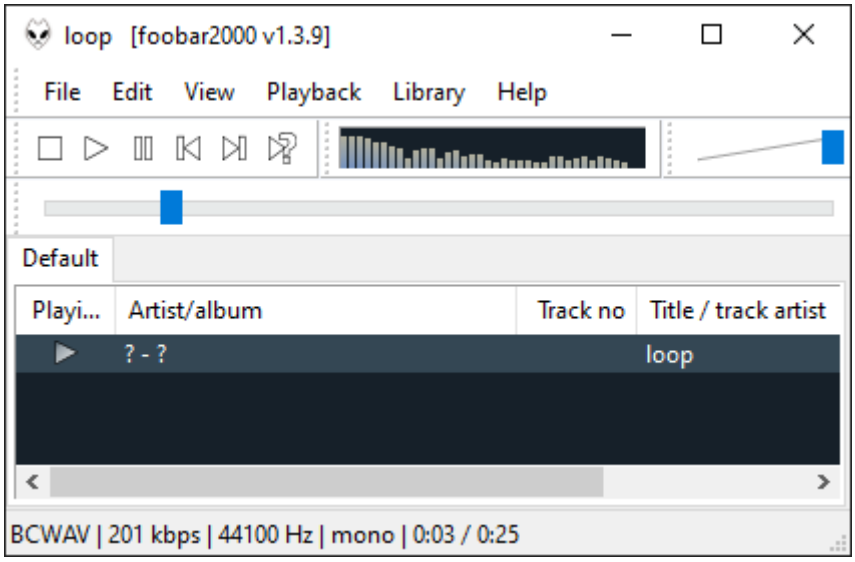

Figura 4.6: Aplicación *foobar2000* reproduciendo un archivo *BCWAV* generado con *cwavtool*.

CAPÍTULO 5
libcwav: Diseño e implementación

En este capítulo se describirá el proceso de diseño e implementación de la biblioteca de funciones para la Nintendo 3DS llamada *libcwav*. Esta biblioteca de funciones estará escrita en C para dar compatibilidad con todos los proyectos *homebrew* que se pueden desarrollar para la consola.

El primer paso a realizar es instalar el entorno de desarrollo de DevkitPro. Esto incluye el conjunto de herramientas *devkitARM* y la biblioteca de funciones de usuario *libctru*. Una vez instaladas, es posible compilar los diferentes tipos aplicaciones de Nintendo 3DS.

5.1 Preparación de la biblioteca de funciones

Tras instalar el entorno de desarrollo para la compilación de código para la Nintendo 3DS, es necesaria la preparación inicial de la biblioteca de funciones. Por suerte, la organización DevkitPro provee una plantilla para aquellos desarrolladores que quieran construir su propia biblioteca de funciones. Esta plantilla se puede descargar desde su repositorio de GitHub [52].

Una vez descargada la plantilla, se confirma que al ejecutar la orden `make`, se genera un archivo *.a*. Este tipo de archivos pueden ser enlazados con aplicaciones para añadir la biblioteca de funciones al archivo ejecutable final.

Por otro lado, se ha modificado el archivo *GNU Make* para poder realizar la instalación de la biblioteca de funciones. Al ejecutar el comando `make install`, los archivos de cabecera relevantes y el archivo *.a* son copiados a la carpeta correspondiente del entorno de DevkitPro. También se ha añadido la orden `make uninstall` para realizar el proceso contrario.

En el directorio de archivos de inclusión del proyecto, se ha creado un subdirectorio que albergará todos aquellos archivos de cabecera que no sean necesarios para el desarrollador (y por tanto, no serán instalados junto al resto), llamado `internal`.

Cuando una aplicación necesite utilizar *libcwav*, solo deberá añadir la carpeta de la biblioteca de funciones a la variable `$(LIBDIRS)` y el argumento de compilación `-lcwav` a la variable `$(LIBS)` del archivo *GNU Make* del proyecto de la aplicación. De este modo, la cadena de compilación de DevkitPro procederá a enlazar de forma estática la biblioteca de funciones en el ejecutable final.

5.2 Estructura interna de la biblioteca de funciones

Tal y como se explica en la sección 2.3.2 la Nintendo 3DS cuenta con dos servicios independientes para la reproducción de audio. Dado que *libcwav* debe soportar ambos servicios, se debe implementar código que sea compatible tanto con *ndsp* como con *csnd*.

Para evitar que todo el código de la biblioteca de funciones esté un mismo archivo, dificultando su lectura y mantenimiento, se ha decidido realizar una división en dos partes. En un primer lugar, se ha creado el archivo `cwav.c`, donde se añadirá el código de las funciones expuestas al desarrollador, además de otras funciones de carga y procesamiento de archivos *BCWAV*. Cuando este código necesite comunicarse con los servicios de audio, llamará a las funciones responsables de ello, que serán añadidas en el archivo `cwav_env.c`.

Todas las funciones en el archivo `cwav_env.c` serán las responsables de comunicarse con el servicio de audio que se ha seleccionado. De esta manera, no será necesario preocuparse por detalles específicos de cada servicio en el archivo `cwav.c`.

5.3 Recreación de las estructuras de los archivos *BCWAV*

Uno de los pasos más tempranos del desarrollo de la biblioteca de funciones es la definición en C de las diferentes estructuras encontradas en los archivos *BCWAV* (véase sección 3.3).

Para lograr esto, se ha utilizado toda la información encontrada en la página web `https://3dbrew.org` relacionada con los archivos *BCWAV* [35]. Las estructuras se han convertido en *structs* mientras que los diferentes valores se han convertido en *enums*[1].

Todas estas estructuras definidas ayudarán a la lectura y procesamiento de archivos *BCWAV* más adelante, y han sido añadidas en un archivo nombrado `cwav_defs.h` dentro del subdirectorio `internal`.

5.4 Estructuras de datos adicionales

Dado que el formato *BCWAV* contiene muchas referencias entre sus diferentes estructuras de datos, se ha decidido crear una estructura adicional que contenga los punteros en memoria a dichas estructuras, en vez de tener que calcular los *offset* de las referencias cada vez que se quiera acceder a un campo.

Por otro lado, también es necesario la creación de una estructura con diferentes configuraciones de reproducción que podrá controlar el desarrollador, como puede ser el volumen o el tono. Esta estructura será la que se podrá utilizar con las funciones expuestas, y representará el archivo *BCWAV* como un objeto sobre el que se podrán realizar las operaciones (concepto de *handle* o manejador [53]).

La primera estructura será representada como un *struct* llamado `cwav_t`, y contendrá los punteros a los bloques principales (*CWAV*, *INFO* y *DATA*), una tabla de punteros a las diferentes entradas de información de cada canal y otra tabla de punteros a las diferentes entradas de parámetros de decodificación *ADPCM*. Además, contendrá el búfer cíclico descrito en la sección 3.5.3 para cada uno de los canales almacenados y la posición actual de dicho búfer. En la figura 5.1 se puede observar la implementación de esta estructura realizada en el archivo `cwav_defs.h`.

[1] En C, un *enum* o enumeración es un tipo de dato que permite asignar nombres a valores numéricos.

```c
typedef struct cwav_s
{
    void* fileBuf;  // Puntero al archivo BCWAV en memoria.
    cwavHeader_t* cwavHeader; // Puntero al bloque CWAV.
    cwavInfoBlock_t* cwavInfo; // Punter al bloque INFO.
    cwavDataBlock_t* cwavData; // Puntero al bloque DATA.
    cwavchannelInfo_t** channelInfos; // Tabla de punteros a la información de cada canal.
    cwavIMAADPCMInfo_t** IMAADPCMInfos; // Tabla de punteros a la información IMA-ADPCM de cada canal.
    cwavDSPADPCMInfo_t** DSPADPCMInfos; // Tabla de punteros a la información DSP-ADPCM de cada canal.
    int** playingChanIds; // Búfer cíclico para cada canal (tamaño = channelcount * totalMultiplePlay).
    u8 channelcount; // Cantidad de canales del archivo.
    u8 totalMultiplePlay; // Cantidad de reproducciones simultaneas posibles.
    u8 currMultiplePlay; // Índice actual en el búfer cíclico.
} cwav_t;
```

Figura 5.1: Estructura de datos interna (*cwav_t*) implementada en C.

La segunda estructura también será representada como un *struct*, llamado `CWAV`, y será expuesto al desarrollador para que pueda realizar operaciones sobre él. Uno de los campos será un puntero de tipo `void` que apuntará a la estructura interna `cwav_t`. De esta manera, no será necesario exponer al desarrollador la implementación de la estructura interna. Esta estructura será implementada de forma progresiva, y estará presente en el archivo de cabecera principal (`cwav.h`).

5.5 Carga de archivos *BCWAV* en memoria

La carga de archivos *BCWAV* debe poder realizarse de dos métodos diferentes.

5.5.1. Carga de archivos a partir de un bloque de memoria

El primer método de carga puede realizarse a partir de un puntero a un bloque de memoria presente en la región *LINEAR*. Esto ha sido implementado mediante la llamada a la función `cwavLoad()`, que toma como argumentos el *struct* `CWAV` de salida, el puntero al archivo almacenado en memoria y la cantidad máxima de reproducciones simultáneas.

5.5.2. Carga de archivos a partir de un medio de almacenamiento

El segundo método de carga podrá realizarse a partir del nombre del archivo almacenado en la tarjeta SD o el *romfs*. Para ello, se ha implementado una función, con nombre `cwavFileLoad()`, que acepte como argumentos el *struct* `CWAV` de salida, el nombre del archivo y la cantidad máxima de reproducciones simultáneas.

Esta función, utilizará las llamadas POSIX de manejo de archivos (`fopen()`, `fread()`, etc) para leer los contenidos del archivo a un bloque de memoria que será reservado usando `linearAlloc()`. Una vez realizada la lectura en memoria, se llamará a la función `cwavLoad()` para continuar con la carga del archivo.

Sin embargo, se debe tener en cuenta que en el entorno de desarrollo de "*plugins 3GX*", las funciones `linearAlloc()` y `linearFree()` no están definidas. Por tanto, si se tratase de compilar y enlazar un "*plugin*", se mostraría un mensaje de error indicando que estas funciones no existen.

Para poder solucionar este problema, se han marcado ambas funciones como un símbolo *weak*[2], de modo que en el caso de no estar definidas, simplemente no serán llamadas, fallando la lectura en memoria y retornando un valor de error.

5.6 Procesado de archivos *BCWAV*

Una vez con el archivo *BCWAV* en memoria, es necesario procesar todas sus estructuras y rellenar los diferentes campos del *struct* `cwav_t` para su futuro uso.

El primer paso que realiza la función `cwavLoad()` es la inicialización del *struct* `CWAV` de entrada. Esto incluye la creación e inicialización del *struct* `cwav_t` interno. A continuación, se realiza la llamada a la función `cwav_initialize()`.

La función `cwav_initialize()` es la encargada de comprobar que el bloque de memoria sea un archivo *BCWAV* válido y utiliza la función `cwav_parseInfoBlock()` para procesar el bloque de información (*INFO*) del archivo. En esta función se comprueba que la codificación sea compatible con el servicio de audio actual y se resuelven todas las referencias internas a partir de la información descrita en la sección 3.3. Las referencias son resueltas mediante la construcción de punteros a las diferentes estructuras de datos, que son almacenados en los campos apropiados del *struct* `cwav_t`.

Tras finalizar la función `cwav_parseInfoBlock()`, el control se devuelve a la función `cwav_initialize()` que, para finalizar, procesa el bloque de muestras (*DATA*) y construye el búfer cíclico para almacenar los canales de los servicios de audio que serán asignados.

Una vez que los *structs* `cwav_t` y `CWAV` han sido rellenados correctamente, el archivo es registrado en la lista de archivos *BCWAV* cargados mediante la función `cwav_Register()`. Esta lista es necesaria para poder solucionar el primer inconveniente descrito en la sección 3.5.3.

En este momento, el archivo *BCWAV* ya ha sido completamente cargado y procesado. El desarrollador podrá utilizar el *struct* `CWAV` como *handle* para realizar las diferentes operaciones disponibles.

Si en algún paso se diese algún error de carga o procesamiento, el campo `loadStatus` del *struct* `CWAV` sería rellenado con el valor de error correspondiente, mientras que en el caso contrario, este campo indicaría que el proceso se ha completado satisfactoriamente.

5.7 Liberación de archivos *BCWAV*

Es importante que el desarrollador tenga maneras de liberar aquellos archivos *BCWAV* que hayan sido cargados con `cwavLoad()` o `cwavFileLoad()` y no requieran ser utilizados más. Esto se debe realizar incluso si el valor almacenado en el campo `loadStatus` indique un error de carga, ya que la función `cwavLoad()` no libera los recursos incluso en caso de error.

Para la liberación de recursos, se han creado las funciones `cwavFileFree()` y `cwavFree()`. La primera de ellas debe utilizarse si se ha cargado el archivo desde un medio de almacenamiento, ya que se encarga de liberar el bloque de memoria en la región *LINEAR* mediante la llamada a la función `linearFree()` (no sin antes realizar la llamada a `cwavFree`).

[2]En el compilador *GCC*, un símbolo *weak* es aquel que puede ser sobrescrito por otro símbolo con el mismo nombre. Si una función *weak* no está definida en tiempo de enlazado, las llamadas a dicha función son eliminadas.

La segunda función es llamada internamente por la primera, o si el archivo *BCWAV* ha sido cargado independientemente por el desarrollador.

Las operaciones realizadas por la función `cwavFree()` son: la detención de todos los canales de audio, la eliminación del archivo de la lista de *BCWAV* cargados mediante la llamada a `cwav_DeRegister()` y la liberación de todos los bloques reservados, incluyendo las tablas de punteros a las estructuras internas del archivo, el búfer cíclico y el propio *struct* `cwav_t`.

Una vez realizada la liberación, se modifica el valor del campo `loadStatus` para indicar que el *struct* `CWAV` ya no se puede volver a utilizar.

5.8 Reproducción de canales *BCWAV*

A continuación, debe implementarse la operación más importante de un archivo *BCWAV*, la reproducción de sus canales de audio.

La función encargada de esto, con nombre `cwavPlay()`, acepta como argumentos un *struct* `CWAV` y dos canales de audio que se deseen reproducir, correspondiendo el primer canal al oído izquierdo y el segundo canal al oído derecho. En el caso de solo querer reproducir un canal, el valor del segundo canal será -1.

Como se describió en la sección 3.5.3, el primer paso a realizar en la reproducción es asegurar que los canales asignados en todos los *BCWAV* cargados son correctos. Para ello, se realiza la llamada a la función `cwav_UpdatePlayingStatus()`, que itera sobre todos los archivos registrados y actualiza el estado de reproducción de los canales asignados, utilizando `cwavEnvChannelIsPlaying()`.

A continuación, se incrementa la posición del búfer cíclico, que será la posición donde se almacenará el canal de audio *DSP* o *CSND* que será asignado. En el caso de que el canal almacenado en la posición actual esté en reproducción, se detiene mediante una llamada a `cwav_stopImpl()`.

Posteriormente y para cada uno de los canales *BCWAV* recibidos como argumentos, se itera sobre todos los canales de audio del servicio seleccionado y se utilizan las funciones `cwavEnvIsChannelAvailable()` y `cwavEnvChannelIsPlaying()` para comprobar la disponibilidad de cada canal. Una vez encontrado un canal libre, este se asigna en la posición del búfer cíclico actual.

En el caso de que la codificación actual sea *ADPCM*, es necesaria la llamada a la función `cwavEnvSetADPCMState()` para inicializar los parámetros de decodificación del canal asignado.

Para finalizar, se obtienen los datos e informaciones importantes para la reproducción: la codificación, el puntero a las muestras inicial y punto de bucle (si existe), la cantidad de muestras totales (en el caso de estar el bucle activado, la cantidad de muestras desde el principio hasta el punto de bucle y la cantidad de muestras del rango de bucle), las muestras por segundo, el volumen, el tono y el *panning*. Todos estos parámetros y el canal *DSP* o *CSND* son utilizados en la llamada a la función `cwavEnvPlay()`, encargada de la comunicación con el servicio de audio correspondiente.

Una vez la función `cwavPlay()` finalice la ejecución, se retornará un *struct* con nombre `cwavPlayResult` donde estarán almacenados el estado de reproducción y los canales de audio *DSP* y *CSND* que han sido asignados, con el fin de que el desarrollador pueda realizar operaciones sobre ellos.

5.8.1. Reproducción de canales mediante el servicio *DSP*

Para poder reproducir audio con el servicio *DSP*, se utiliza la implementación de *libctru* llamada *ndsp*. Esta implementación separa las funciones genéricas de inicialización del servicio, que deberán haber sido previamente llamadas por el desarrollador, y las funciones relacionadas con las operaciones sobre los canales de audio.

Para poder iniciar la reproducción de un canal, es necesario crear y rellenar una estructura llamada `ndspWaveBuf` (búfer de audio *ndsp*). Esta estructura incluye información del audio a reproducir, incluyendo el puntero a las muestras, la cantidad de ellas y los parámetros de decodificación *ADPCM*.

Dado que cada canal puede utilizar dos búfers de audio *ndsp* (el primero para el rango desde la primera muestra al punto de bucle y el segundo para el rango de bucle) y el servicio *DSP* posee 24 canales, la biblioteca de funciones reserva espacio en memoria para 48 búfers de audio simultáneos. Para obtener uno de ellos se utiliza la función `cwavEnvGetNdspWaveBuffer()`.

Si se está utilizando la codificación *DSP-ADPCM*, `cwavEnvSetADPCMState()` utiliza la función `ndspChnSetAdpcmCoefs()` para indicar los coeficientes de decodificación del canal y, posteriormente, actualiza los contextos de decodificación para cada búfer. En el caso de no estar el bucle activado, solo se actualiza el primer búfer (punto inicial) mientras que si el bucle está activado, se actualizan ambos búfers (punto inicial y punto de bucle).

La función `cwavEnvPlay()` se encarga de calcular el volumen del canal mediante los argumentos de volumen y *panning* y, a continuación realiza la llamada a las funciones `ndspChnSetFormat()`, `ndspChnSetRate()` y `ndspChnSetMix()` para actualizar la codificación de las muestras, la frecuencia de muestras por segundo y la mezcla de volumen calculada, respectivamente.

Finalmente, los búfers son actualizados con las posiciones en memoria de las muestras, y se activa el bucle de ellos si lo es necesario. Una vez realizado todo este proceso, basta con llamar a la función `ndspChnWaveBufAdd()` para añadir los búfers a la cola e iniciar su reproducción.

En el caso de el bucle esté desactivado solo se añadirá un búfer que finalizará su reproducción, mientras que si el bucle está activado, se añadirán ambos búfers. En este caso, el primer búfer finalizará su reproducción y el segundo continuará después, repitiéndose hasta que el desarrollador lo detenga manualmente.

5.8.2. Reproducción de canales mediante el servicio *CSND*

Para la reproducción de audio mediante el servicio *CSND*, se utiliza la implementación de *libctru* con nombre *csnd*. Sin embargo y como veremos a continuación, serán necesarias algunas modificaciones para que este servicio sea compatible con el entorno de ejecución de "*plugins 3GX*".

Como se menciona en la sección 3.5.4, la reproducción de audio requiere que se utilice la región de memoria *LINEAR* para que la conversión a la dirección de memoria física sea posible. Sin embargo, en el entorno de ejecución de "*plugins*" esta región no está disponible y las funciones de conversión de direcciones de memoria dejan de funcionar correctamente.

Para solucionar este problema, debe utilizarse la función de conversión de memoria proporcionada por el cargador de "*plugins*" que puede convertir cualquier dirección

virtual a la correspondiente dirección física, sin necesidad de estar en una región de memoria específica.

El inconveniente a esta solución es que la conversión de dirección virtual a dirección física se realiza dentro del propio código de *libctru*, por lo que debe crearse una copia de la función de reproducción de audio de *csnd* dentro de la propia biblioteca de funciones para que pueda ser modificada.

Una vez realizada la copia de la función `csndPlaySound()`, se ha modificado para que utilice la función de conversión proporcionada por el usuario a través de la llamada a `cwavSetVAToPACallback()`. Otra modificación necesaria es la eliminación del código que obtiene los parámetros de decodificación *ADPCM* a partir del puntero del bloque de memoria de muestras, ya que en el archivo *BCWAV* estos están situados en otro lugar. Esta nueva función modificada tiene como nombre `csndPlaySoundFixed()`.

Si se está utilizando la codificación *IMA-ADPCM*, `cwavEnvSetADPCMState()` utiliza la función `CSND_SetAdpcmState()` para actualizar los parámetros de decodificación del canal para cada bloque de muestras (bloque inicial o bloque de bucle).

Para iniciar la propia reproducción, la función `cwavEnvPlay()` simplemente realiza la llamada a la nueva función modificada `csndPlaySoundFixed()`, indicando los diferentes parámetros de audio (si se debe repetir en bucle el segundo bloque, la codificación, la frecuencia de muestreo, el volumen, el tono, el *panning*, los punteros a los bloques de muestras y cantidad total de ellas).

5.9 Detención de canales *BCWAV*

Para poder realizar la detención de los canales en reproducción, se ha implementado una función con nombre `cwavStop()` que acepta como argumentos el *struct* `CWAV` a detener y dos canales de audio, uno para el oído izquierdo y otro para el oído derecho.

En el caso de que solamente se quiera detener un canal, se utilizará el valor -1 en el argumento para el oído derecho y, en el caso de querer detener todos los canales, se utilizará el valor -1 en ambos argumentos.

Dado que el búfer cíclico es transparente al usuario y la función `cwavPlay()` necesita poder detener canales en una posición del búfer determinada, se ha decidido crear la función `cwav_stopImpl()`, que toma los mismos argumentos que `cwavStop()` además de la posición del búfer cíclico con el canal *DSP* o *CSND* a detener.

De esta manera, la función `cwavStop()` solo deberá iterar sobre todas las posiciones del búfer cíclico y realizar la llamada a la función `cwav_stopImpl()`. Esta función en cambio se encargará de realizar las llamadas correspondientes a la función `cwavEnvStop()` con el canal *DSP* o *CSND* apropiado, según el argumento de posición en el búfer.

Respecto a la comunicación con los servicios de audio dentro de `cwavEnvStop()`, en el caso del servicio *CSND* se realizará la llamada a la función `CSND_SetPlayState()` con el valor 0 para detener la reproducción, mientras que en el caso del servicio *DSP* se utilizará la función `ndspChnReset()` para reiniciar el estado del canal y posteriormente se marcarán los búfers de audio correspondientes como libres.

5.10 Otras funcionalidades de la biblioteca de funciones

Además de las funcionalidades básicas que se han descrito anteriormente, se han implementado otras funciones adicionales que mejorarán el uso de la biblioteca de funciones o son necesarias para solventar algunos problemas descritos en la sección 3.5

5.10.1. Comprobación del estado de reproducción de canales *BCWAV*

Se ha implementado la función `cwavIsPlaying()` para poder comprobar el estado de reproducción de un archivo *BCWAV*. Para ello, la función itera sobre todas las posiciones del búfer cíclico y comprueba si hay un canal *DSP* o *CSND* asignado. En el caso de encontrar uno, se realiza la llamada a `cwavEnvChannelIsPlaying()` para asegurar que dicho canal sigue en reproducción.

En el caso del servicio *DSP*, la función `cwavEnvChannelIsPlaying()` obtiene los búfers de audio relacionados con el canal y comprueba sus estados, retornando verdadero si el búfer está en la cola de reproducción o si se está reproduciendo. En el caso del servicio *CSND*, dicha función realiza la llamada a `csndIsPlaying()` que comprueba si el canal especificado está en reproducción.

Pese a que se haya encontrado un canal en reproducción, la función `cwavIsPlaying()` sigue comprobando el estado de reproducción de todos los canales en el búfer cíclico. De esta forma se actualiza el estado de todos ellos.

5.10.2. Selección de servicio de audio

La selección del servicio de audio a utilizar se realiza mediante la llamada a la función `cwavUseEnvironment()`. Por defecto, se utilizará el servicio *DSP*.

Dado que la reproducción de audio requiere inicializar algunas estructuras (como los búfers de audio del servicio *DSP*) y no tiene sentido cambiar de servicio durante la ejecución de la aplicación, la llamada a esta función solo podrá realizarse antes de cargar el primer archivo *BCWAV*. A partir de ese momento, no será posible cambiar de servicio de audio.

5.10.3. Tratamiento de notificaciones del servicio *APT*

Como se ha explicado en la sección 3.5.4, la implementación de las comunicaciones con el servicio de audio *CSND* no tiene en cuenta las notificaciones del servicio *APT* para la suspensión de la aplicación.

Para poder detener los canales en el caso de que se utilice el servicio *CSND* y se suspenda la aplicación, se han implementado las dos funciones siguientes.

La primera función, con nombre `cwavDoAptHook()`, realiza la llamada a la función de *libctru* `aptHook()`, que causará que la función `cwavNotifyAptEvent()` sea ejecutada cuando se dé una notificación del servicio *APT*.

La segunda función, con nombre `cwavNotifyAptEvent()`, podrá ser llamada manualmente por el desarrollador en el caso de que el servicio *APT* no esté disponible (“*plugins 3GX*”). Esta se encargará de detener todos los canales *BCWAV* que estén registrados en la lista de archivos cargados cuando la aplicación vaya a cerrarse o suspenderse.

5.10.4. Estado de reproducción de todos los canales de audio

Para poder realizar pruebas sobre la asignación de canales de audio *DSP* y *CSND*, se ha implementado la función `cwavGetEnvironmentPlayingChannels()` que retorna un mapa de bits en el que se indica el estado de reproducción de cada canal.

Esta función realiza las llamadas a las funciones `cwavEnvIsChannelAvailable()` y `cwavEnvChannelIsPlaying()` para cada canal disponible del servicio de audio que se esté utilizado y finalmente, retorna un valor de 32 *bits* donde cada *bit* representa el estado de reproducción de cada canal.

5.10.5. Reproducción de sonidos *DirectSound* mediante *CSND*

Una de las funcionalidades del servicio *CSND* es la reproducción de sonidos con ciertos modificadores (como por ejemplo, ignorar el control de volumen de la consola o forzar la salida de audio por los altavoces). Este tipo de sonidos se han bautizado como *DirectSounds* (sonidos directos), ya que solo necesitan la especificación de algunos parámetros en una dirección de memoria compartida. Para poder reproducir un canal *BCWAV* como sonido directo, se debe utilizar la función `cwavPlayAsDirectSound()`.

La implementación completa de este tipo de sonidos sigue en desarrollo y ha requerido una gran cantidad de investigación ajena al ámbito de trabajo de este TFG, por lo que se ha decidido no incluirla en la memoria. Además, es necesaria la modificación de la biblioteca de funciones *libctru* con las últimas investigaciones para poder utilizar esta funcionalidad, que está desactivada por defecto.

5.11 Documentación del archivo de cabecera principal

Tras implementar todas la funciones de la biblioteca, es necesaria la documentación de todas las funcionalidades para que el desarrollador interesado pueda entender el funcionamiento de cada una de ellas.

5.11.1. Documentación de la estructura de datos principal

La estructura de datos `CWAV` es el corazón de la biblioteca de funciones ya que, sobre dicha estructura, se pueden realizar todas las operaciones implementadas.

Como se puede observar en la figura 5.2, se ha documentado esta estructura indicando la funcionalidad de cada campo y si se puede modificar o no.

5.11.2. Listado de los posibles valores de error

Durante la ejecución de las funciones de la biblioteca, puede darse el caso de que el desarrollador haya intentado cargar un archivo *BCWAV* incorrecto o haber realizado alguna acción que haya generado algún error. Para poder dar retroalimentación, las diferentes funciones pueden retornar un valor de resultado diferente dependiendo de la situación.

La lista de valores de error se ha implementado como un *enum* y contiene todos los posibles valores que pueden ser retornados por las funciones `cwavLoad()` y `cwavPlay()`, como se puede observar en la figura 5.3.

```c
/// CWAV structure, some values can be read [R] or written [W] to.
typedef struct CWAV_s
{
    // Pointer to internal cwav data, should not be used.
    void*           cwav;
    // [RW] Pointer to the buffer where the CWAV was loaded (used by cwavFileLoad and cwavFileFree).
    void*           dataBuffer;
    // [R] Value from the cwavStatus_t enum. Set when the CWAV is loaded.
    cwavStatus_t    loadStatus;
    // [RW] Value in the range [-1.0, 1.0]. -1.0 for left ear and 1.0 for right ear. Default: 0.0
    float           monoPan;
    // [RW] Value in the range [0.0, 1.0]. 0.0 muted and 1.0 full volume. Default: 1.0
    float           volume;
    // [RW] Changes the playback speed. Default: 1.0 (no pitch change)
    float           pitch;
    // [R] The sample rate of the audio data.
    u32             sampleRate;
    // [R] Number of CWAV channels stored in the file.
    u8              numChannels;
    // [R] Whether the file is looped or not.
    u8              isLooped;
} CWAV;
```

Figura 5.2: Estructura principal *CWAV* y su documentación en C.

5.11.3. Documentación de las funciones

Como paso final a documentar, se han añadido bloques de comentarios sobre las definiciones de las funciones indicando su descripción general, una explicación de los argumentos y valor de retorno y posibles detalles a considerar.

Estos bloques de comentarios se han creado utilizando el formato *doxygen* [54]. De esta manera, los principales entornos de desarrollo (*IDE*) podrán mostrar la información de las funciones mientras el desarrollador va escribiendo sus argumentos.

En la figura 5.4 se puede observar la documentación de la función `cwavPlay()`.

```
/// Posible status values.
typedef enum
{
    // General status values.
    CWAV_NOT_ALLOCATED = 0, ///< CWAV is not allocated and cannot be used.
    CWAV_SUCCESS = 1, ///< Operation succeded.
    CWAV_INVALID_ARGUMENT = 2, ///< An invalid argument was passed to the function call.

    // Load status values.
    CWAV_FILE_OPEN_FAILED = 3, ///< Failed to open the specified file.
    CWAV_FILE_READ_FAILED = 4, ///< The file failed to be read into memory.
    CWAV_UNKNOWN_FILE_FORMAT = 5, ///< The specified file is not a valid CWAV file.
    CWAV_INVAID_INFO_BLOCK = 6, ///< The INFO block in the CWAV file is invalid or not supported.
    CWAV_INVAID_DATA_BLOCK = 7, ///< The DATA block in the CWAV file is invalid or not supported.
    CWAV_UNSUPPORTED_AUDIO_ENCODING = 8, ///< The audio encoding is not supported.

    // Play status values.
    CWAV_INVALID_CWAV_CHANNEL = 9, ///< The specified channel is not in the CWAV.
    CWAV_NO_CHANNEL_AVAILABLE = 10, ///< No DSP/CSND channels available to play the sound.

} cwavStatus_t;
```

Figura 5.3: Posibles valores de error listados en una enumeración.

```
/**
 * @brief Plays the specified channels in the bcwav file.
 * @param cwav The CWAV to play.
 * @param leftChannel The CWAV channel to play on the left ear.
 * @param rigtChannel The CWAV channel to play on the right ear.
 * @return A cwavPlayResult struct with the status code and which audio channels were assigned.
 *
 * To play a single channel in mono for both ears, set rightChannel to -1.
*/
cwavPlayResult cwavPlay(CWAV* cwav, int leftChannel, int rightChannel);
```

Figura 5.4: Documentación de la función *cwavPlay()*.

CAPÍTULO 6
Aplicación de ejemplo: Diseño, implementación y pruebas

Tras haber completado tanto la herramienta *cwavtool* como la biblioteca de funciones *libcwav*, podríamos dar este proyecto como concluido. Sin embargo, para facilitar el entendimiento de este kit de generación y reproducción de audio, es conveniente proveer al desarrollador de una aplicación de Nintendo 3DS de ejemplo para que pueda comprobar su correcto funcionamiento y poder utilizar el código como ejemplo de uso.

Además, esta aplicación de ejemplo ha sido de ayuda en la prueba de todas las funcionalidades de la biblioteca de funciones mientras ha sido desarrollada, y la confirmación de que los archivos *BCWAV* generados funcionan correctamente.

A continuación se describirán los pasos realizados durante la implementación.

6.1 Preparación de la aplicación de ejemplo

De igual forma que DevkitPro provee a los desarrolladores con una plantilla para desarrollar bibliotecas de funciones, también se puede encontrar una plantilla de una aplicación de Nintendo 3DS vacía.

El primer paso realizado ha sido la descarga de dicha plantilla, que ha sido colocada en el subdirectorio `example_libcwav` del proyecto *libcwav*, de esta manera la aplicación de ejemplo será distribuida en el mismo repositorio que la biblioteca de funciones.

Tras ejecutar la orden `make` se ha podido comprobar que la aplicación se compila sin problemas y genera el archivo `example_libcwav.3dsx`. Este archivo puede ser transferido a la Nintendo 3DS y ejecutado desde el *homebrew launcher*.

6.2 Preparación de los archivos *BCWAV*

Para poder probar las diferentes funcionalidades de la biblioteca de funciones, se han obtenidos diferentes sonidos de *freesound.org* con licencia CC0 que serán codificados de diferentes maneras. También se ha reutilizado el bucle de sonido de la sección 4.7.

La lista de sonidos final, incluyendo sus propiedades, es la descrita en la tabla 6.1.

Tabla 6.1: Lista de archivos utilizada en la aplicación de ejemplo.

Nombre de Archivo	Codificación	Nº Canales	Bucle
beep_dsp_adpcm.bcwav	*DSP-ADPCM*	1	No
bell_stereo_dsp_adpcm.bcwav	*DSP-ADPCM*	2	No
bell_stereo_ima_adpcm.bcwav	*IMA-ADPCM*	2	No
bwooh_ima_adpcm.bcwav	*IMA-ADPCM*	1	No
loop_dsp_adpcm.bcwav	*DSP-ADPCM*	1	Si
loop_ima_adpcm.bcwav	*IMA-ADPCM*	1	Si
loop_pcm16.bcwav	*PCM16*	1	Si
meow_pcm8.bcwav	*PCM8*	1	No

Todos estos archivos han sido copiados a la carpeta *romfs*, que será incorporada al ejecutable *.3dsx* final.

6.3 Programación de la aplicación

La aplicación se ha diseñado de una forma en la que el usuario pueda seleccionar el servicio de audio que desee para la reproducción de los archivos. Una vez seleccionado el servicio de audio a utilizar, es notificado a la biblioteca de funciones mediante `cwavUseEnvironment()` y, a continuación, se realizan las llamadas a `ndspInit()` o `csndInit()` para inicializar *DSP* o *CSND* respectivamente. En el caso de utilizar el servicio *CSND* también es necesaria la llamada a `cwavDoAptHook()`.

A continuación se realiza la carga de los archivos *BCWAV* a memoria mediante el acceso al *romfs* de la aplicación. Para demostrar la carga manual, se han utilizado la funciones `linearAlloc()` y `cwavLoad()`.

Una vez inicializados todos los recursos, la aplicación entra en su bucle principal esperando la entrada del usuario. Este, mediante la cruceta de control, podrá cambiar el archivo *BCWAV* actual, iniciar la reproducción con el botón A (mediante la llamada a `cwavPlay()`) o detenerla con el botón B (mediante la llamada a `cwavStop()`).

En la pantalla superior se muestra el *BCWAV* seleccionado actualmente, su estado de reproducción y el estado de todos los canales de audio del servicio seleccionado. Mediante esta información se puede comprobar que la asignación de canales funciona correctamente.

Una vez que el usuario desee salir de la aplicación mediante el botón START, se liberan todos los archivos *BCWAV* cargados mediante `cwavFree()` y `linearFree()` y se finaliza el servicio de audio.

6.4 Ejecución y pruebas

Tras realizar la compilación, se ha copiado el archivo resultante a la tarjeta SD de una Nintendo 3DS con el *homebrew launcher* instalado. A continuación, se ha realizado la ejecución de la aplicación.

Durante la ejecución, ha sido posible comprobar los siguientes aspectos de la herramienta y la biblioteca de funciones, dando todos ellos resultados positivos:

- Generación de archivos *BCWAV* válidos y su carga y procesamiento en *hardware* real.
- Compatibilidad con los servicios de audio *CSND* y *DSP*.

- Asignación de los canales de audio ofrecidos por los servicios.
- Reproducción y detención de los canales de audio del archivo *BCWAV*.
- Compatibilidad con codificaciones *PCM8*, *PCM16*, *DSP-ADPCM* y *IMA-ADPCM*.
- Compatibilidad con aquellos archivos que tienen la reproducción en bucle activada.
- Detención de todos los canales de audio en el caso de suspensión de la aplicación con el servicio *CSND*.
- Liberación de recursos sin efectos secundarios.

En la figura 6.1 se puede observar el menú principal de la aplicación de ejemplo. En este caso, está seleccionado el archivo *meow_pcm8.bcwav* y se están utilizando los dos primeros canales del servicio *DSP* (*bit* 0 y *bit* 1).

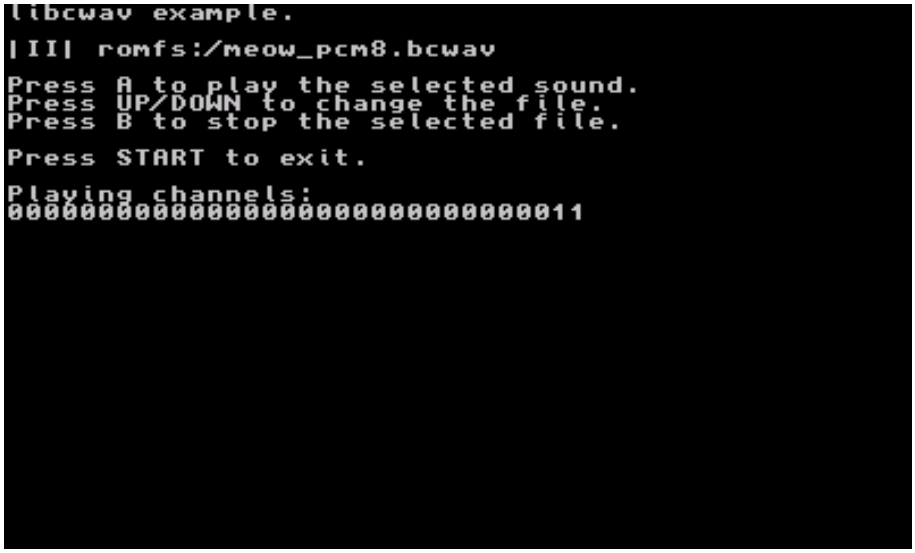

Figura 6.1: Captura de pantalla del menú principal de la aplicación de ejemplo.

En el siguiente vídeo, puede observarse la aplicación de ejemplo en funcionamiento en *hardware* real (obtenido mediante una capturadora de vídeo y audio [55]). Enlace: https://www.youtube.com/watch?v=Nk1Lg5BitWo

CAPÍTULO 7
Distribución de las herramientas y su uso en aplicaciones de Nintendo 3DS

Tras haber finalizado todos los componentes del conjunto de herramientas, es necesaria su distribución para que la comunidad de desarrolladores *homebrew* pueda utilizarlas en sus proyectos. En este capítulo, se describirá los detalles de distribución y se pondrán ejemplos de aplicaciones que ya han hecho uso de este proyecto.

7.1 Distribución de las herramientas

El primer paso para la distribución de las herramientas ha sido la creación de un repositorio *git* público en la página web GitHub. Dependiendo de si se trata de *cwavtool* o *libcwav*, se han seguido diferentes pasos.

En el caso de *cwavtool*, dado que se trata de un *fork* de la aplicación *bannertool*, la creación del repositorio se ha realizado desde la propia página web GitHub utilizando su función de realizar un *fork* de otro proyecto. A continuación se ha clonado el repositorio resultante en la máquina local. De esta manera, la lista de *commits* y los usuarios que han contribuido se mantienen en el nuevo repositorio.

En el caso de *libcwav*, se ha creado un repositorio vacío en GitHub donde se han realizado todas las operaciones de *commit*.

Sobre estos repositorios he ha realizado el desarrollo de ambas herramientas, además de la creación de los archivos *README.md* que dan una pequeña descripción del uso e instalación. También se ha añadido a estos archivos las licencias correspondientes descritas en la sección 3.7.

Mientras que en el caso de *libcwav* se ha mantenido el método de instalación mediante el comando `make install`, se han proporcionado ejecutables precompilados de la aplicación *cwavtool*. Estos ejecutables están disponibles para Windows de 32 y 64 *bits* (compilados mediante el uso de *MinGW-w64*), para Linux de 32 y 64 *bits* (compilados mediante el uso de *wsl* [56]) y para Mac de 64 bits.

Los ejecutables de *cwavtool* están disponibles en la sección *releases* del repositorio en GitHub.

Para poder acceder a los repositorios de ambas herramientas, se pueden utilizar los siguientes enlaces:

- *cwavtool*: https://github.com/mariohackandglitch/cwavtool
- *libcwav*: https://github.com/mariohackandglitch/libcwav

7.2 Uso de las herramientas en aplicaciones reales de Nintendo 3DS

A fecha de publicación de esta memoria, se conocen dos proyectos de Nintendo 3DS que han hecho uso del conjunto de herramientas implementado. A continuación se dará una pequeña explicación de dichos proyectos.

7.2.1. Aplicación *Yet Another Mario Kart Clone*

Esta aplicación es un videojuego *homebrew* desarrollado en C++ junto a dos compañeros para la asignatura de 4º de carrera de ingeniería informática: Arquitectura y entornos de desarrollo para videoconsolas.

Esta aplicación se trata de un videojuego de carreras que pretende replicar la experiencia de juego ofrecido por la saga oficial de juegos *Mario Kart*. En este caso, el videojuego simplemente consta de una carrera en la que se deberán completar tres vueltas en el menor tiempo posible.

El conjunto de herramientas ha tenido un rol importante en el desarrollo de este videojuego, ya que tanto la música de fondo como los diferentes efectos de sonido se han implementado mediante el uso de *cwavtool* y *libcwav*.

Además, dado que el videojuego fue desarrollado en paralelo a la biblioteca de funciones, sirvió para tener un punto de referencia sobre los aspectos importantes que deben incorporarse para que esta sea lo más útil posible en un caso de desarrollo real.

Por otro lado, el videojuego demuestra la capacidad de poder modificar características de reproducción (volumen y tono, por ejemplo) tanto antes de la reproducción como durante la propia reproducción, gracias a que la biblioteca de funciones retorna los canales de audio que han sido asignados. Esto se ha conseguido implementando una clase `Sound` que se encarga de las comunicaciones con la biblioteca de funciones y los servicios de audio.

En el siguiente enlace https://www.youtube.com/watch?v=qJSIHESXtfs, se puede observar una demostración del videojuego en funcionamiento. Toda la música y efectos de sonido son reproducidos gracias *libcwav* y están codificados en *DSP-ADPCM*. Tras comenzar la carrera, el volumen de la música se reduce a 0 para que el resto de sonidos se puedan escuchar con claridad.

Este proyecto está disponible en GitHub en el siguiente repositorio: https://github.com/mariohackandglitch/YAMKC_3DS

7.2.2. Biblioteca de funciones *CTRPluginFramework* para "*plugins 3GX*"

Como se menciona en la sección 2.2.1, los "*plugins 3GX*" son fragmentos de código que permiten extender la funcionalidad de otras aplicaciones. Para facilitar al desarrollador la programación, se creó la biblioteca de funciones *CTRPluginFramework*, que ofrece una gran diversidad de utilidades, como por ejemplo, la muestra en pantalla de menús interactivos, el acceso a los archivos de la tarjeta SD o la posibilidad de ejecutar tareas en paralelo.

Sin embargo, una funcionalidad que esta biblioteca de funciones no otorgaba, era la reproducción de sonido sobre la aplicación anfitriona. Es aquí donde entran en juego las herramientas realizadas en este proyecto. Gracias a que el servicio de audio *CSND* no es utilizado durante la ejecución de la aplicación, es posible su uso para la reproducción de audio. Además, la posibilidad de reproducción de audio comprimido *IMA-ADPCM* hace que el uso de memoria sea más eficiente, aspecto que es muy importante en este entorno tan limitado.

Para la reproducción de sonido, esta biblioteca de funciones expone la clase `Sound`, que sirve como interfaz de comunicaciones con *libcwav*. Además, se encarga internamente de la notificación manual de eventos *APT*, por lo que el desarrollador no debe de preocuparse de realizarlo por su cuenta.

En el siguiente enlace https://www.youtube.com/watch?v=-1JUb6u_vrg, se puede observar un vídeo de demostración de un "*plugin 3GX*" ejecutándose sobre el videojuego oficial *Mario Kart 7*. A los diez segundos del vídeo, se puede observar un menú mostrado por *CTRPluginFramework* donde es posible navegar entre diferentes opciones. Todos los efectos de sonido que se escuchan durante la navegación de este menú son reproducidos mediante *libcwav*, demostrando así que es posible la reproducción en paralelo con la aplicación anfitriona, que se encarga de la música de fondo.

CAPÍTULO 8
Conclusiones

Al comienzo de este proyecto, se expuso la necesidad de crear un conjunto de herramientas que facilitasen la reproducción eficiente de audio en el desarrollo de aplicaciones *homebrew* de Nintendo 3DS. Gracias a la compresión acelerada por *hardware*, ha sido posible crear una biblioteca de funciones capaz de reproducir archivos de audio, que han sido previamente generados en un ordenador.

Además, se ha creado con éxito una aplicación de Nintendo 3DS que demuestra la funcionalidad correcta de la generación de archivos y su reproducción en la consola gracias a las herramientas implementadas. Esta aplicación puede incluso servir como plantilla para cualquier desarrollador interesado que, junto con la documentación proporcionada, podrá dar uso de las herramientas en sus proyectos. De esta manera, han sido completados todos los objetivos principales planteados.

Respecto a los objetivos secundarios, también ha sido posible crear dos repositorios *git* en la página web GitHub donde se ha publicado el código de ambas herramientas desarrolladas bajo una licencia de código abierto.

De esta manera, otros miembros de la comunidad *homebrew* podrán reportar fallos encontrados o sugerir mejoras. En el repositorio de la herramienta de ordenador, también se han añadido ejecutables precompilados para los principales sistemas operativos.

Un objetivo secundario que no ha sido posible cumplir ha sido crear un paquete instalable mediante el gestor de paquetes *devkitPro pacman*. Esto ha sido debido a la falta de tiempo para investigar el funcionamiento de dicho gestor e implementar la generación del paquete para ambas herramientas. Aunque la existencia de estos paquetes facilitaría la instalación de las herramientas, el método actual sigue siendo sencillo y solo requiere unos pocos pasos adicionales: clonar repositorio, compilar e instalar mediante línea de órdenes.

Este proyecto se ha creado con la esperanza de que los elementos desarrollados sean útiles para la comunidad *homebrew* en el desarrollo de aplicaciones. Pese a que las herramientas se han publicado recientemente, su utilidad ya se ha demostrado en dos desarrollos diferentes, como se ha explicado en la sección 7.2, pero se espera que todavía más desarrolladores se animen a darles una oportunidad en sus propios proyectos.

8.1 Trabajos futuros

Pese a que este proyecto se puede considerar que ofrece una solución completa, siempre existe margen de mejora, lo que no es una excepción en el conjunto de herramientas desarrolladas. Algunas de las opciones de mejora se han recopilado en esta sección.

El primer trabajo que se podría realizar en el futuro, es completar los objetivos secundarios planteados en este proyecto, es decir, distribuir un paquete instalable a partir del gestor *devkitPro pacman* para facilitar la instalación de las herramientas.

También es deseable una mejora relacionada con la reproducción de "sonidos directos" mediante el servicio *CSND* (ver sección 5.10.5). Implementar esta mejora requeriría más investigación en los parámetros de interfaz *CSND*, desconocidos en gran medida en la comunidad *homebrew*. Los resultados de dicha investigación tendrían que ser reflejados e implementados en la biblioteca de funciones *libctru*.

Por otro lado, el principal inconveniente de las herramientas desarrolladas es la necesidad de cargar en memoria el archivo de audio en su totalidad. Mientras que para sonidos cortos esta solución es aceptable, no lo sería para sonidos mucho más largos. En este caso, sería conveniente cargar en memoria solo aquel fragmento del archivo que se esté reproduciendo actualmente.

Aunque esta funcionalidad podría implementarse manteniendo el uso del formato *BCWAV* para el almacenamiento del audio, existe otro formato de archivo similar conocido como archivo *BCSTM* (*Binary CTR Stream*)[57] que, como su nombre indica, facilita la "transmisión" de los datos de audio a memoria gracias a la división de las muestras en diferentes bloques. Cuando es necesaria la reproducción de un bloque, este se carga en memoria a partir del medio de almacenamiento y, cuando finaliza la reproducción, la memoria ocupada por dicho bloque es liberada. De esta manera, solo una porción del archivo está presente en memoria a la vez, lo que supone una disminución de las necesidades de memoria. Sin embargo, sería necesario gestionar la carga de datos a partir del medio de almacenamiento (por ejemplo, usando una técnica de doble o triple búfer), que añadiría un coste temporal adicional de procesamiento.

El uso de este nuevo tipo de archivo supondría la ampliación de las herramientas implementadas en este proyecto, o la creación de herramientas completamente nuevas orientadas solo a la reproducción de archivos de mayor duración.

8.2 Conclusiones personales

Es importante para mí destacar los beneficios que este proyecto ha aportado a mi experiencia como programador.

En mi opinión, el más importante ha sido el diseño de código con un estándar de calidad necesario para que otros desarrolladores puedan utilizarlo sin complicaciones, intentando ponerme en su piel y entender qué funcionalidades son las más importantes y como serán utilizadas.

Además, las limitaciones de procesamiento y memoria del dispositivo de destino han contribuido a plantear la implementación del código teniendo en cuenta siempre la manera óptima de llegar a la solución y que esta gaste la menor cantidad de recursos posibles, un hecho que en entornos con muchos más recursos suele omitirse.

Este proyecto ha requerido resolver un problema específico mediante el análisis de varias alternativas que me ha llevado a seleccionar una solución concreta, siendo este proceso de selección y equilibrio entre parámetros un concepto que nos ha sido recalcado en todo momento en la rama de computación.

Además, el tratado de datos visto en la asignatura SAR (Sistemas de Almacenamiento y Recuperación de información) y los conceptos de digitalización de audio vistos en la asignatura PER (Percepción) han servido de base para trabajar con comodidad en el manejo de datos de audio. Por otro lado, la asignatura de LPPL (Lenguajes de Programa-

ción y Procesadores de Lenguajes) introdujo la utilización de archivos *GNU Make* para la generación de ejecutables, técnica que también se ha utilizado en este proyecto.

Respecto al resto de asignaturas cursadas en la carrera, la asignatura FSO (Fundamentos de Sistemas Operativos) ha ayudado en gran medida a entender y realizar la comunicación con los servicios ofrecidos por el sistema operativo de la Nintendo 3DS así como el manejo de las estructuras que componen los archivos binarios. También ha sido importante los conocimientos sobre el manejo del lenguaje C, utilizado en las prácticas de varias asignaturas, como SGI (introducción a los sistemas gráficos interactivos) y LPPL.

También ha sido necesaria la adquisición de conocimientos que no han podido verse con detalle en la carrera, para los que he necesitado un proceso de autoaprendizaje. Por ejemplo, el concepto de biblioteca de funciones y como se enlaza al ejecutable final, el manejo de punteros a memoria y operaciones sobre ellos, algunos conceptos de digitalización de audio (algoritmos *ADPCM*) o técnicas de documentación, distribución y mantenimiento de código.

En conclusión, este TFG ha servido para consolidar y poner en práctica conocimientos obtenidos en la carrera, además de obtener conocimientos y habilidades adicionales. Tengo la convicción que todo esto me será muy útil en mi futuro profesional.

Bibliografía

1. NINTENDO COMPANY, LTD. *Consolidated Sales Transition by Region*. 2021-03-31. Disponible también desde: https://www.nintendo.co.jp/ir/finance/historical_data/xls/consolidated_sales_e2103.xlsx.
2. WIKIPEDIA. *Tabla comparación familia Nintendo 3DS*. 2021-06-05. Disponible también desde: https://es.wikipedia.org/wiki/Familia_Nintendo_3DS#Comparaci%C3%B3n.
3. 3DBREW.ORG. *Hardware*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/Hardware.
4. 3DBREW.ORG. *ARM7 Registers*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/ARM7_Registers.
5. NEKOMICHI. *TN and IPS Displays*. 2021-06-05. Disponible también desde: https://gbatemp.net/threads/guide-tn-and-ips-displays-which-does-my-3ds-have.409925/.
6. 3DBREW.ORG. *New 3DS*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/New_3DS.
7. 3DBREW.ORG. *Flash Filesystem*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/Flash_Filesystem.
8. STRICKLAND, Jonathan. *How the Nintendo 3DS Works*. 2021-06-05. Disponible también desde: https://electronics.howstuffworks.com/nintendo-3ds3.htm.
9. 3DBREW.ORG. *IR Services*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/IR_Services.
10. 3DBREW.ORG. *Glossary: Appcore*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/Glossary#appcore.
11. DERREK; PLUTOO y SMEALUM. *Breaking the 3DS*. 2015-12. Disponible también desde: https://smealum.github.io/3ds/32c3/#/.
12. 3DBREW.ORG. *Extended Header*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/NCCH/Extended_Header.
13. 3DBREW.ORG. *CIA*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/CIA.
14. 3DBREW.ORG. *CCI*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/CCI.
15. 3DBREW.ORG. *3DSX Format*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/3DSX_Format.
16. SMEALUM; YELLOWS8; PLUTOO et al. *The Homebrew Launcher*. 2021-06-05. Disponible también desde: https://smealum.github.io/3ds/.

17. NANQUITAS. *CTRPluginFramework - Blank Plugin - Now with Action Replay*. 2018-01-23. Disponible también desde: https://gbatemp.net/threads/ctrpluginframework-blank-plugin-now-with-action-replay.487729/.

18. 3DBREW.ORG. *Kernel*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/Kernel.

19. 3DBREW.ORG. *Services API*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/Services_API.

20. SMEALUM et al. *libctru - CTR user library*. 2021-06-05. Disponible también desde: https://libctru.devkitpro.org/.

21. RETRO GAME MECHANICS EXPLAINED. *Generation I Pokémon Cries Explained*. 2021-06-05. Disponible también desde: https://www.youtube.com/watch?v=gDLpbFXnpeY.

22. BURKE, Alex. *¿Qué es el sonido PCM?* 2021-06-05. Disponible también desde: https://techlandia.com/sonido-pcm-sobre_128723/.

23. RETRO GAME MECHANICS EXPLAINED. *Pikachu's Cry in Pokémon Yellow Explained*. 2021-06-05. Disponible también desde: https://www.youtube.com/watch?v=fooSxCuWvZ4.

24. BENVENUTO, Nevio; BERTOCCI, Guido; DAUMER, William R. y SPARRELL, Duncan K. Report: The 32-kb/s ADPCM coding standard. *AT T Technical Journal*. 1986, vol. 65, n.º 5, págs. 12-22. Disp. desde DOI: 10.1002/j.1538-7305.1986.tb00375.x.

25. IMA DIGITAL AUDIO FOCUS AND TECHNICAL WORKING GROUPS. Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems. 1992. Disponible también desde: http://www.cs.columbia.edu/~hgs/audio/dvi/IMA_ADPCM.pdf.

26. JACKOALAN; ANTIDOTE; PARAX et al. *DSPADPCM (.dsp audio) Encoding Made Easy!* 2021-06-10. Disponible también desde: https://gbatemp.net/threads/dspadpcm-dsp-audio-encoding-made-easy.390305/.

27. 3DBREW.ORG. *DSP Services*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/DSP_Services.

28. 3DBREW.ORG. *CSND Services*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/CSND_Services.

29. FINCS et al. *Citro2D*. 2021-06-05. Disponible también desde: https://github.com/devkitPro/citro2d.

30. DEVKITPRO. *portlibs*. 2021-06-05. Disponible también desde: https://devkitpro.org/wiki/portlibs.

31. STUNTHACKS. *m3diaLib*. 2021-06-05. Disponible también desde: https://github.com/m3diaLib-Team/m3diaLib-CTR.

32. NOP90. *SDL-3DS*. 2021-06-05. Disponible también desde: https://github.com/nop90/SDL-3DS.

33. 3DBREW.ORG. *CGFX*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/CGFX.

34. GBATEK. *3DS Files - Video Layout Images (CLIM/FLIM)*. 2021-06-05. Disponible también desde: http://problemkaputt.de/gbatek.htm#3dsfilesvideolayoutimagesclimflim. La información encontrada en esta página web puede estar obsoleta.

35. 3DBREW.ORG. *BCWAV*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/BCWAV.

36. VGMSTREAM. *vgmstream*. 2021-06-22. Disponible también desde: https://vgmstream.org/.
37. GOTA7. *Citric Composer*. 2021-06-22. Disponible también desde: https://gota7.github.io/Citric-Composer/.
38. STEVEICE10. *bannertool*. 2021-06-22. Disponible también desde: https://github.com/Steveice10/bannertool.
39. FREE SOFTWARE FOUNDATION, INC. *GNU Make*. 2020-01-19. Disponible también desde: https://www.gnu.org/software/make/.
40. BARRETT, Sean et al. *stb*. 2021-06-20. Disponible también desde: https://github.com/nothings/stb.
41. BRYANT, David. *adpcm-xq*. 2021-06-22. Disponible también desde: https://github.com/dbry/adpcm-xq.
42. ANDERSEN, Jack. *gc-dspadpcm-encode*. 2021-06-22. Disponible también desde: https://github.com/jackoalan/gc-dspadpcm-encode.
43. 3DBREW.ORG. *NS and APT Services*. 2021-06-05. Disponible también desde: https://www.3dbrew.org/wiki/NS_and_APT_Services.
44. ANKIT_BISHT. *Logical and Physical Address in Operating System*. 2021-06-23. Disponible también desde: https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/.
45. OPENSOURCE.ORG. *The zlib/libpng License (Zlib)*. 2021-06-28. Disponible también desde: https://opensource.org/licenses/Zlib.
46. OPENSOURCE.ORG. *The MIT License*. 2021-06-28. Disponible también desde: https://opensource.org/licenses/MIT.
47. DEVKITPRO.ORG. *devkitPro pacman*. 2021-06-28. Disponible también desde: https://devkitpro.org/wiki/devkitPro_pacman.
48. CITRA EMULATOR PROJECT. *Citra*. 2021-06-28. Disponible también desde: https://citra-emu.org/.
49. MINGW-W64.ORG. *mingw-w64*. 2021-06-28. Disponible también desde: http://mingw-w64.org/doku.php.
50. STEVEICE10. *buildtools*. 2021-06-28. Disponible también desde: https://github.com/Steveice10/buildtools.
51. AUDACITY. *Audacity*. 2021-06-30. Disponible también desde: https://www.audacityteam.org/.
52. DEVKITPRO. *3DS Library Template*. 2021-07-01. Disponible también desde: https://github.com/devkitPro/3ds-examples/tree/master/templates/library.
53. WIKIPEDIA. *Handle*. 2021-07-01. Disponible también desde: https://es.wikipedia.org/wiki/Handle.
54. DOXYGEN. *Documenting the code*. 2021-07-03. Disponible también desde: https://www.star.bnl.gov/public/comp/sofi/doxygen/docblocks.html.
55. MERKI. *Capture Cards*. 2021-07-04. Disponible también desde: https://www.merki.net/capture-card/?lang=en.
56. VELASCO, Rubén. *Aprende a usar WSL, el Subsistema de Windows 10 para Linux*. 2021-07-04. Disponible también desde: https://www.softzone.es/windows-10/como-se-hace/subsistema-windows-linux/.
57. 3DBREW.ORG. *BCSTM*. 2021-07-04. Disponible también desde: https://www.3dbrew.org/wiki/BCSTM.