LibWiiEsp: Manual de instalación y uso

Ezequiel Vázquez de la Calle

Versión 0.1 del manual

Índice

Índice de figuras

Índice de cuadros

Resumen

Debido a las diferencias existentes entre el desarrollo para un PC ordinario y una videoconsola, en este caso, para Nintendo Wii, se hace patente la necesidad de recoger todos esos detalles que, siendo sencillos de solventar, pueden suponer más de un quebradero de cabeza a un programador sin ningún tipo de experiencia en la programación para Wii. Además, la propia construcción de *LibWiiEsp* implica una serie de consideraciones a tener en cuenta a la hora de sacarle el máximo rendimiento. El objetivo de este documento es recoger todos los pormenores que un programador debe tener en cuenta a la hora de crear un videojuego para la consola, utilizando *LibWiiEsp* como herramienta.

1. Instalación del entorno de desarrollo

Para comenzar el desarrollo de un videojuego para Wii utilizando como herramienta *LibWiiEsp*, es necesario instalar en el sistema todas las dependencias que la biblioteca necesita.

Esta sección del manual detalla paso a paso la instalación de estas dependencias en un sistema *GNU/Linux* de 32 bits, siendo el proceso prácticamente el mismo para sistemas de 64 bits (únicamente cambia la versión de las herramientas base, que deberá ser en este caso la adecuada para 64 bits). Para sistemas Windows también es posible desarrollar con *LibWiiEsp*, pero este manual no cubre este tipo de arquitecturas.

En primer lugar, hay que crear una carpeta accesible para todos los usuarios del sistema, donde irán emplazadas las herramientas y dependencias de *LibWiiEsp*. Lo ideal es crear un directorio dentro de `/opt` y otorgarle a éste los permisos necesarios, pero es posible realizar la instalación en nuestro directorio `/home`, de tal manera que sólo nuestro usuario pueda acceder a estas herramientas.

Suponiendo que se escoge la primera opción, se crea el directorio en `/opt` y se le asignan permisos para que todos los usuarios del sistema puedan hacer uso de lo que en él se almacene:

```
sudo mkdir /opt/devkitpro
sudo chmod 777 /opt/devkitpro
```

A continuación, hay que descargar las herramientas que sirven de base para *LibWiiEsp*, que son el conjunto de compiladores, bibliotecas y binarios *DevKitPPC*, y la biblioteca de bajo nivel *libOgc*, en su versión 1.8.4, junto con la modificación de *libFat* 1.0.7 compatible con ella. Ambos recursos se encuentran disponibles en la forja del proyecto, en el paquete *Dependencias*. Se deben descargar en el directorio que acabamos de crear:

```
cd /opt/devkitpro
wget http://forja.rediris.es/frs/download.php/2316/devkitPPC_r21-i686-linux.tar.bz2
wget http://forja.rediris.es/frs/download.php/2315/libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

El siguiente paso es descomprimir ambos ficheros y, si no queremos tener ocupado espacio innecesariamente, eliminarlos. Las instrucciones para ejecutar estas acciones son:

```
tar -xvjf devkitPPC_r21-i686-linux.tar.bz2
tar -xvzf libogc-1.8.4-and-libfat-1.0.7.tar.gz
rm devkitPPC_r21-i686-linux.tar.bz2 libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

Después de esto, es necesario establecer algunas variables de entorno para que el sistema sepa dónde localizar estas herramientas que acabamos de instalar. Las dos primeras consisten en la ruta hasta el

directorio base *devkitpro*, y hasta el directorio donde se encuentran las herramientas como tal, concretamente *devkitpro/devkitPPC*. La tercera variable de entorno indica la dirección IP de nuestra Nintendo Wii dentro de la red local, dato necesario para ejecutar correctamente las aplicaciones que se desarrollen sin que haga falta copiar el ejecutable en la tarjeta SD de la consola cada vez que se genere uno nuevo.

Para establecer estas variables de entorno, basta con editar el archivo de configuración */.bashrc* del usuario con el que vayamos a desarrollar utilizando *LibWiiEsp*. Si por alguna causa fuera necesario que todos los usuarios del sistema tengan que usar estas herramientas, el archivo de configuración a editar sería */etc/bashrc*, o su equivalente en el sistema (por ejemplo, para una distribución *Ubuntu 10.10* de 32 bits, el archivo es */etc/bash.bashrc*).
Suponiendo que sólo nuestro usuario va a desarrollar utilizando *LibWiiEsp*, las instrucciones para establecer estas variables de entorno serían:

```
echo export DEVKITPRO=/opt/devkitpro >> ~/.bashrc
echo export DEVKITPPC=$DEVKITPRO/devkitPPC >> ~/.bashrc
echo export WIILOAD=tcp:192.168.X.X >> ~/.bashrc
```

Un detalle, en estas instrucciones, *192.168.X.X* se debe sustituir por la dirección IP que tiene asignada la consola Nintendo Wii en la red local.
A continuación, podemos recargar el fichero de configuración */.bashrc* para tener listas las varibles de entorno mediante la orden:

```
source ~/.bashrc
```

Llegados a este punto, el último paso para que el entorno de desarrollo funcione correctamente con *LibWiiEsp* es instalar la biblioteca propiamente dicha. Existen dos maneras de realizar esto, o bien podemos descargar la versión estable publicada en la forja, o también compilando el código disponible desde ésta. Para el primer caso, basta con descargar el paquete comprimido desde la página principal de la Forja del proyecto (http://forja.rediris.es/projects/libwiiesp), copiarlo al directorio */opt/devkitpro*, y descomprimirlo allí.

Para compilar la biblioteca a partir de los fuentes, debemos ejecutar las siguientes líneas, siempre que se hayan instalado previamente las dependencias:

```
svn co https://forja.rediris.es/svn/libwiiesp/trunk libwiiesp
cd libwiiesp
make install
```

Y después de la instalación de la biblioteca, ya tenemos preparado nuestro sistema para comenzar con el desarrollo de videojuegos en dos dimensiones para Nintendo Wii.

2. Estructura de un proyecto con LibWiiEsp

Una vez instalado correctamente el entorno de desarrollo para utilizar *LibWiiEsp*, el siguiente paso es crear la estructura de archivos y directorios necesarios para trabajar con un nuevo proyecto para Nintendo Wii. Por supuesto, queda en manos del programador la decisión final sobre esta estructura de directorios, pero, debido a todo lo que hay que tener en cuenta a la hora de desarrollar con *LibWiiEsp* (sobretodo, en lo referente a la compilación y enlazado de los archivos que componen un proyecto), voy a presentar en esta sección un ejemplo de proyecto vacío que quedaría listo para comenzar el desarrollo. Además de tratar la estructura de directorios, también mostraré y explicaré el código de un archivo *makefile* que pueda trabajar con esta organizción para el proyecto.

2.1. Estructura de directorios

En primer lugar, hay que crear un directorio base donde almacenar todo lo relativo a nuestro proyecto. La localización de este directorio en el sistema es indiferente, de tal manera que, para el ejemplo, lo haremos en nuestro directorio */home* con las instrucciones:

```
mkdir ~/juego
cd ~/juego
```

Una vez dentro, la siguiente estructura de directorios sería más que suficiente para albergar todos los componentes del proyecto:

- *doc*: Documentación del proyecto.
- *lib*: Bibliotecas externas que se vayan a utilizar en el proyecto. Aquí se debe guardar cada biblioteca en un directorio separado. En cada directorio debe existir un archivo *makefile* el cual compile la biblioteca, y genere un archivo de enlazado estático *.a*, además los archivos de cabecera de la biblioteca externa tienen que estar justo bajo este directorio principal de la biblioteca externa.
- *media*: En este directorio se colocarán todos los archivos que contengan recursos multimedia que vayan a ser empleados en el proyecto. De momento, sólo están soportados imágenes *bitmap* de 24 bits de color directo, fuentes de texto soportadas por *FreeType2*, efectos de sonido en formato *PCM*, y pistas de música *mp3*.
- *src*: Aquí van los archivos fuente del proyecto.
- *xml*: Archivos de datos del proyecto. Como mínimo, aquí se encontrarán el archivo que describe los recursos multimedia que se cargarán en la galería de medias, el archivo del soporte de idiomas, y el archivo de configuración del programa.

2.2. El archivo *makefile*

La estructura de directorios, y los detalles que la acompañan (como las restricciones de estructuración a la hora de utilizar bibliotecas externas en el proyecto), se han definido así para trabajar con un archivo de compilación *makefile* concreto.

Este archivo de compilación implica una serie de modificaciones considerables en comparación con uno que genere un ejecutable para PC en entornos *GNU/Linux*, por lo que va a detallarse su funcionamiento sección a sección. El objetivo de este *makefile* es generar un ejecutable con extensión *.dol*, que puede lanzarse en una videoconsola Nintendo Wii. A continuación, se muestra un sencillo ejemplo de *makefile* compatible con la estructura de directorios planteada en el punto anterior:

```
# Informacion configurable
LOCALLIBS = tinyxml bullet
PROJECT = Wii Pang
TARGET = boot
BUILD = build
SOURCE = src
DEPSDIR = $(BUILD)

# Reglas de compilacion
.SUFFIXES:
include $(DEVKITPPC)/wii_rules
include $(DEVKITPPC)/libwiiesp_rules

# Generar una lista con todos los ficheros objeto del proyecto
CPPFILES = $(notdir $(wildcard $(SOURCE)/*.cpp))
OFILES = $(CPPFILES:.cpp=.o)
OBJS = $(addprefix $(CURDIR)/$(BUILD)/,$(OFILES))

# Variables para la compilacion
INCLUDE = $(foreach dir,$(LOCALLIBS),-Ilib/$(dir)) -I$(LIBOGC_INC)
LIBPATHS = -L$(LIBOGC_LIB) -L$(CURDIR)/$(BUILD)
VPATH = $(SOURCE)$(foreach dir,$(LOCALLIBS),:lib/$(dir))
LIBS = $(LIBWIIESP_LIBS) -ltinyxml -lbullet
OUTPUT = $(CURDIR)/$(TARGET)
LD = $(CXX)

# Flags para la compilacion
CXXFLAGS        = -g -ansi -Wall $(MACHDEP) $(INCLUDE) -std=c++0x
OPTIONS         = -MMD -MP -MF
LDFLAGS         = -g $(MACHDEP) -Wl,-Map,$(notdir $@).map

# Objetivos
.PHONY: all $(BUILD) libs $(LOCALLIBS) listo run clean

all: $(BUILD) libs $(OUTPUT).dol listo

$(BUILD):
        @[ -d $(BUILD) ] || mkdir -p $(BUILD)

libs: $(LOCALLIBS)
        @echo Bibliotecas externas listas

$(LOCALLIBS):
        $(MAKE) --no-print-directory --silent -C lib/$@
        mv lib/$@/*.a $(BUILD)

$(CURDIR)/$(BUILD)/%.o: $(CURDIR)/$(SOURCE)/%.cpp
        $(CXX) -MMD -MP -MF $(DEPSDIR)/$*.d $(CXXFLAGS) -c $< -o $@

listo:
        $(RM) $(SOURCE)/-g $(OUTPUT).elf.map

run:
        wiiload $(TARGET).dol

```

```
clean:
        for dir in $(LOCALLIBS); do $(MAKE) clean -C lib/$$dir; done
        $(RM) -fr $(BUILD) $(OUTPUT).elf $(OUTPUT).dol *~ $(SOURCE)/*~

DEPENDS :=      $(OBJS:.o=.d)

$(OUTPUT).dol: $(OUTPUT).elf

$(OUTPUT).elf: $(OBJS)

-include $(DEPENDS)
```

En el primer bloque que aparece en el ejemplo, se definen una serie de variables que indican los directorios que forman parte del proyecto. *LOCALLIBS* debe recoger los nombres, separados por un espacio, de los directorios principales de cada biblioteca externa que se vaya a emplear en la compilación. La variable *PROJECT* indica el nombre completo del proyecto, y *TARGET* el nombre, sin extensión, del ejecutable que se generará. Para que el programa pueda ser ejecutado con *HomeBrew Channel*, se recomienda que se nombre *boot*. Por último, *BUILD* indica el directorio que se creará cuando se ordene la compilación del proyecto para almacenar todos los ficheros objeto del proyecto, *SOURCE* es el directorio donde están todos los archivos fuente, y *DEPSDIR* es donde se generarán los archivos que indican las dependencias entre módulos del sistema, que debe ser el mismo directorio que *BUILD*.

A continuación, se deben importar las reglas de compilación para Nintendo Wii, tanto las necesarias para utilizar las herramientas de *DevKitPPC*, como las propias de *LibWiiEsp*. Para ello, antes hay que limpiar las reglas implícitas existentes, lo cual se consigue con la instrucción *.SUFFIXES:*.

El bloque de tres instrucciones que viene a continuación se encarga de almacenar en variables una lista de todos los archivos *.cpp* que se encuentran en el directorio de fuentes, y otra lista con la ruta absoluta de los ficheros objeto que se generarán al compilar (es decir, archivos con extensión *.o* en la carpeta indicada por la variable *BUILD*).

El cuarto bloque de instrucciones recoge los directorios donde se encuentran los archivos de cabeceras externas (los almacena en la variable *INCLUDE*), los directorios en los cuales hay que buscar las bibliotecas de enlazado estático (guardados en la variable *LIBPATHS*), establece el *VPATH*, crea la lista de bibliotecas a enlazar estáticamente (tomando la variable *LIBWIIESP_LIBS* para contar con todo lo que necesita un ejecutable generado con *LibWiiEsp*, pero se le debe añadir además lo necesario para enlazar también las bibliotecas externas), y por último, indica la ruta absoluta hasta el ejecutable sin extensión y define el enlazador que se utilizará.

El bloque siguiente establece los flags de compilación necesarios para generar un ejecutable de extensión *.elf*, que posteriormente se transformará a otro con extensión *.dol*. Y justo después comienzan los objetivos del *makefile*, que se describen a continuación:

- *all*: Crea un ejecutable *.dol* a partir del código fuente que se encuentre en el directorio indicado en la variable *SOURCE*. Es el objetivo por defecto al ejecutar *make*.
- *$(BUILD)*: Crea el directorio donde se crearán todos los ficheros objetos del proyecto.
- *libs* y *$(LOCALLIBS)*: Se encargan de compilar las bibliotecas externas, de empaquetarlas en ficheros de enlace estático y mover éstos al directorio de los ficheros objeto.

- *$(CURDIR)/$(BUILD)/%.o*: Genera un fichero objeto en el directorio *$(BUILD)* por cada módulo que se encuentre en el directorio de los ficheros fuentes.
- *listo*: Objetivo que limpia un archivo de extensión *.elf.map*.
- *run*: Lanza la utilidad *wiiload* para enviar el ejecutable a la Nintendo Wii. Ésta debe tener abierto el *HomeBrew Channel* y estar conectada a la red local con la misma dirección IP que se indicó en la variable de entorno *WIILOAD*, de otro modo no ejecutará el programa.
- *clean*: Objetivo que limpia de archivos temporales el proyecto. Elimina la carpeta donde se almacenan los ficheros objeto, y ejecuta también los objetivos *clean* de cada biblioteca externa.

La última parte de este ejemplo se encarga de comprobar que se cumplan las dependencias de los módulos a compilar, de generar un ejecutable *.elf* a partir de los módulos objeto del proyecto, y en último lugar, de crear el ejecutable definitivo *.dol* a partir del archivo *.elf*.

2.3. Ejecución del programa

Una vez generado nuestro programa, existen dos maneras de ejecutarlo en la videoconsola. La primera, ya descrita, es lanzarlo a través del objetivo *make run*, para lo cual necesitamos tener correctamente conectada la Nintendo Wii a la red local, y *wiiload* bien configurado con la IP de la consola.

La otra manera de hacerlo, que es la necesaria para poder distribuir el programa, es guardar el ejecutable en un directorio de la tarjeta SD de la consola, cuya ruta debe ser */apps/XXX/boot.dol*, donde XXX es el nombre unix de nuestra aplicación (importante que no contenga espacios). Nótese que tanto el nombre *boot.dol* como el hecho de que el directorio que lo contiene esté dentro del directorio */apps* es obligatorio.

Debido a las características de *HomeBrew Channel*, podemos acompañar nuestro ejecutable con una imagen que servirá de icono para la aplicación (debe tener formato PNG, llamarse *icon.png*, y tener un tamaño de 128 píxeles de ancho y 48 de alto), y un fichero XML con la información que deseemos aportar sobre el programa (debe ser un XML con un formato concreto, llamado *meta.xml*). Ambos ficheros, la imagen y el archivo de datos, deben ir en el mismo directorio que el ejecutable.

Una referencia sobre los nodos del fichero de datos *meta.xml* que acompaña a una aplicación se puede encontrar a continuación:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app version="1">
  <name>El nombre de la aplicacion</name>
  <coder>Autor o autores del codigo de la aplicacion.</coder>
  <version>La version de la aplicacion.</version>
  <release_date>Dia de lanzamiento. Formato: AAAAMMDD</release_date>
  <short_description>Comentario que se muestra en el menu principal. No se
      recomienda que sea mayor de 30 caracteres</short_description>
  <long_description>Descripcion detallada del programa</long_description>
</app>
```

3. Consideraciones sobre programar con LibWiiEsp

A la hora de programar para una plataforma cerrada como es Nintendo Wii, hay que tener una serie de aspectos en cuenta, debido a la arquitectura concreta que tiene el hardware de la consola. Es muy importante tener en cuenta estos detalles, ya que en caso contrario se pueden producir errores o comportamientos inesperados, como texturas que se pisan entre sí en la memoria principal, o excepciones propias del sistema.

A continuación se describen todos esos pequeños detalles a tener en cuenta, que si bien sólo suponen un cambio en algunos hábitos a la hora de programar, nos aseguran que todo irá bien (al menos, en lo que al funcionamiento del hardware se refiere).

3.1. Big Endian

Lo primero que hay que tener en cuenta es que la representación de los datos en la consola Nintendo Wii se realiza con Big Endian, al contrario que las plataformas Intel (la arquitectura, no la marca del procesador), que utilizan Little Endian para almacenar la información. Cuando un dato ocupa más de un byte, se puede organizar de mayor a menor peso (esto sería Big Endian), o bien de menor a mayor peso (que es la organización en Little Endian).

Un ejemplo muy claro es la representación del *número mágico* que emplean los archivos de imágenes formadas por mapas de bits (también conocidos como *bitmaps*). Este número es, en su representación decimal, 19778. Si esta cifra la representamos en sistema hexadecimal con Little Endian el resultado sería 0x4D42; sin embargo, en un sistema que utilice Big Endian, esta cifra se representaría como el hexadecimal 0x424D en la memoria principal.

Así pues, como consecuencia de esto, siempre que queramos cargar un archivo binario (una imagen, una pista de música, etc.) en la consola Nintendo Wii, tenemos dos opciones: o bien modificamos la representación del archivo en la plataforma de origen (que normalmente será un ordenador con arquitectura Intel) para que el recurso esté ya representado como Big Endian, o podemos utilizar las funciones del espacio de nombres *endian* que se proporcionan en el archivo de cabecera *util.h* de *LibWiiEsp*. Este espacio de nombre aporta las siguientes dos funciones, que se encargan de transformar variables de 16 o 32 bits entre Little Endian y Big Endian (soportan ambas transformaciones):

```
1  u16 inline swap16(u16 a)
2  {
3    return ((a<<8) | (a>>8));
4  }
5
6  u32 inline swap32(u32 a)
7  {
8    return ((a)<<24|(((a)<<8) & 0x00FF0000)|(((a)>>8) & 0x0000FF00)|(a)>>24);
9  }
```

Así pues, únicamente utilizando estas dos funciones con cada dos o cuatro bytes cuando queramos cargar un recurso desde la tarjeta SD podremos evitar los problemas derivados de los distintos sistemas de representación. Para un ejemplo práctico sobre el uso de estas dos funciones, ver el código fuente de la clase Imagen de *LibWiiEsp*.

3.2. Los tipos de datos

A la hora de programar para Nintendo Wii se utilizan siempre estos tipos de datos:

Tipo de dato	Descripción	Rango
u8	Entero de 8 bits sin signo	0 a 255
s8	Entero de 8 bits con signo	-127 a 128
u16	Entero de 16 bits sin signo	rango 0 a 65535
s16	Entero de 16 bits con signo	-32768 a 32767
u32	Entero de 32 bits sin signo	0 a 0xffffffff
s32	Entero de 32 bits con signo	-0x80000000 a 0x7fffffff
u64	Entero de 64 bits sin signo	0 a 0xffffffffffffffff
s64	Entero de 64 bits con signo	-0x8000000000000000 a 0x7fffffffffffffff
f32	Flotante de 32 bits	-
f64	Flotante de 64 bits	-

Cuadro 1: Tipos de datos que se deben emplear al programar para Wii

Todos los tipos de datos disponibles se encuentran en */opt/devkitpro/libogc/include/gctypes.h*.

3.3. La alineación de los datos

Otro detalle importante que se debe contemplar a la hora de programar para Nintendo Wii es que el procesador de la consola necesita alinear los datos conforme a su tamaño. Es decir, si vamos a trabajar con un entero de 32 bits (4 bytes), sólo se pueden almacenar en posiciones de memoria múltiplos de cuatro: 0, 4, 8, 12... Lo mismo ocurre con variables de 16 o 64 bits.

Para ilustrar el comportamiento del procesador respecto a la alineación de los datos, podemos considerar el siguiente ejemplo:

```
1 struct ejemplo {
2   u8 entero;
3   u32 otro;
4 }
```

La estructura *ejemplo* no se representará igual compilando en una plataforma Intel que en la Wii, ya que en la primera, *entero* se situará en la posición 0 de la memoria, y *otro* ocupará de la posición 1 hasta la 5. Sin embargo, en nuestra consola *entero* ocupará el mismo lugar, la posición 0, pero la variable *otro* necesitará estar alineada a su tamaño, es decir, ocupará las posiciones 4 a 7 de la memoria, y las posiciones entre *entero* y *otro* se rellenarán con ceros para asegurar que el entero de 32 bits está alineado.

Esto revierte tanto en la cantidad de memoria ocupada, como en el hecho de que si el procesador encontrara un dato desalineado e intentara leerlo, se produciría un error en el sistema.

Para solventar este inconveniente, se debe jugar inteligentemente con las declaraciones de las variables, de tal manera que se organicen los datos de forma alineada.

3.4. Alineación de datos que usan DMA

Este es otro caso en el que influye la necesidad de alinear los datos con los que se trabaja. El procesador de la Nintendo Wii trabaja con datos cacheados, pero no así los periféricos (la tarjeta SD, los dispositivos USB o el lector DVD). Además, estos periféricos trabajan con una alineación fija de 32 bytes, y si no se contempla este detalle, se pueden producir errores de machacamiento de datos al realizar dos lecturas consecutivas desde periféricos.

Además, cuando leemos un dato desde un periférico, se corre el riesgo de machacar el contenido de la caché del procesador, por lo que es imprescindible volcar explícitamente los datos leídos en la memoria, es decir, asegurar de que la lectura se ha realizado completamente antes de realizar cualquier otra acción.

Es muy sencillo evitar esta situación, y es preparando la memoria en la que se almacenarán los datos leídos desde el periférico, de tal manera que esté alineada a 32 bytes y su tamaño sea múltiplo de 32. Para ilustrar cómo hacer esto, se muestra un ejemplo a continuación:

```
1   // Abrimos el archivo mediante un flujo
2   ifstream archivo;
3   archivo.open("SD:/apps/wiipang/media/sonido.pcm", ios::binary);
4
5   // Obtener el tamano del sonido
6   archivo.seekg(0, ios::end);
7   u16 size = archivo.tellg();
8   archivo.seekg(0, ios::beg);
9
10  // Calcular el relleno que hay que aplicar a la memoria reservada
11  // para que su tamano sea multiplo de 32
12  u8 relleno = (size * sizeof(s16)) % 32;
13
14  // Reservar memoria alineada: utilizamos memalign en lugar de malloc
15  s16* sonido = (s16*)memalign(32, size * sizeof(s16) + relleno);
16
17  // Realizar la lectura de datos desde el periferico
18  // Utilizar char* como tipo de lectura es por compatibilidad con libFat
19  archivo.read((char*)sonido, size);
20
21  // Fijar los datos leidos en la memoria, evita machacamiento en la cache
22  DCFlushRange(sonido, size * sizeof(s16) + relleno);
```

Como puede verse en el ejemplo, se calcula en primer lugar el relleno necesario para que la memoria que ocupa el archivo binario a cargar sea múltiplo de 32 bytes, a continuación se reserva la memoria alineada a 32 bytes utilizando la función *memalign* en lugar de *malloc*, y el último paso, tras leer la información desde el archivo, consiste en realizar el volcado explícito de información desde la caché de lectura a la memoria, utilizando la función *DCFlushRange* (esta función reibe la dirección de memoria a la que se quiere realizar el volcado, y el tamaño de ésta).

3.5. Depuración con *LibWiiEsp*

Con toda la información anterior descrita en este manual y la referencia completa de la biblioteca, se puede comenzar a desarrollar un videojuego sencillo. Como en todo proceso de desarrollo de software, ocurrirán errores, y aquí es donde se hace patente la falta de medios disponibles para depurar el código.

Existen dos tipos de errores que podremos encontrarnos a la hora de programar para Nintendo Wii, que corresponden con las fases de compilación y ejecución. A continuación, vamos a plantear cómo solucionar los posibles errores que pueden surgir en cada uno de estos momentos:

3.5.1. Errores de compilación

En la fase de compilación suelen ocurrir, sobretodo, errores de sintaxis, aunque también es posible que ocurran errores de enlazado si las rutas hasta los archivos de cabeceras o las bibliotecas de enlace estático no son correctas. El compilador nos avisará de cualquier tipo de error que ocurra, tanto en el preprocesado, como en la compilación propiamente dicha y en el enlazado. Así pues, prestando atención a los mensajes que pueda proporcionarnos el compilador sobre los errores o avisos que se den, podremos depurar el código fuente de nuestra aplicación.

Sobre los mensajes de enlazado, el enlazador nos proporcionará información suficiente para saber qué ha fallado durante esta operación, pero si se sigue al pie de la letra este manual (especialmente, la instalación del entorno y la creación del *makefile*) no debería haber ningún problema.

3.5.2. Errores de ejecución

En la fase de ejecución es cuando tenemos verdaderos problemas para depurar nuestra aplicación, y es que apenas hay herramientas que puedan facilitarnos el conocer el estado de los objetos del sistema, el contenido de las variables, etc.

LibWiiEsp proporciona un sistema de registro de eventos del sistema, la clase *Logger*, que permite que escribamos un log de errores, avisos e información variada. La forma de trabajar con esta clase es muy sencilla, y todos los detalles pueden consultarse en su documentación (ver referencia completa de la biblioteca). Pero hay ocasiones en que los errores en tiempo de ejecución requieren más precisión que una serie de mensajes que aportemos desde nuestro propio código, ya que el uso de la clase *Logger* está recomendado en casos de comportamiento inesperado del programa, pero no de errores como tal.

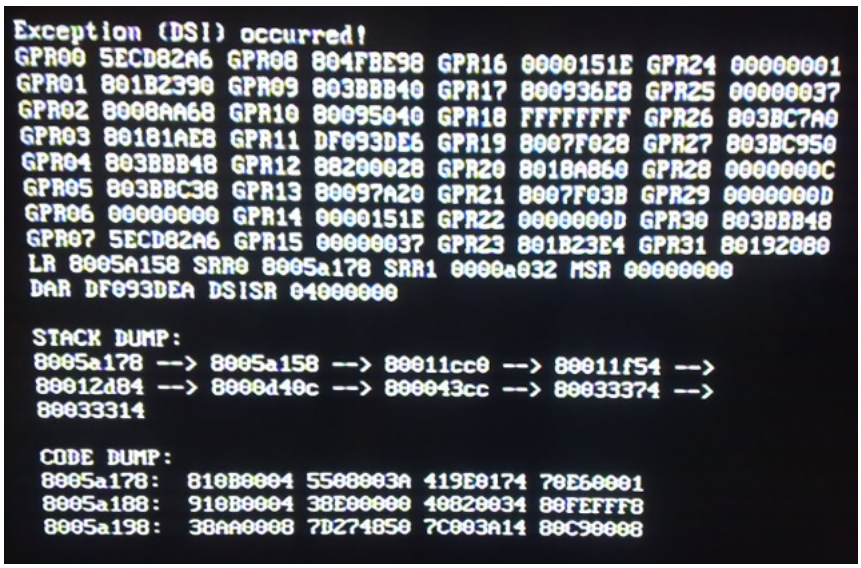

Figura 1: Ejemplo de pantalla de error en tiempo de ejecución

Cuando ocurre un error de ejecución en la Nintendo Wii, ésta nos presentará una pantalla de error parecida a la imagen de la figura 1. En esta pantalla de error pueden apreciarse tres partes principales. A la hora de localizar en qué parte de nuestro código se ha provocado este error, necesitamos fijarnos en la sección *STACK DUMP*, es decir, la segunda. Esta parte del mensaje de error nos detalla la traza de llamadas a función que se han realizado hasta llegar a la llamada que ha producido el error, estando cada elemento de la traza representado por una dirección de memoria en hexadecimal.

Esta informacion es muy útil, ya que podemos localizar a qué línea de nuestro código fuente corresponde cada llamada con una utilidad incluida en *DevKitPPC*, y esta es la utilidad *powerpc-eabi-addr2line*. Se puede localizar en la carpeta */opt/devkitpro/devkitPPC/bin* si se ha seguido este manual para instalar *LibWiiEsp*. Lo más cómodo es crear un enlace simbólico a esta utilidad en el directorio principal de nuestro proyecto, aunque también se puede añadir el directorio *devkitPPC/bin* a la ruta donde el sistema busca ejecutables.

Siguiendo el ejemplo de la figura 1, si queremos saber a qué archivo y qué línea de código pertenece la dirección *0x80011f54*, basta con ejecutar la siguiente línea de código en el directorio principal del proyecto:

```
./powerpc-eabi-addr2line -e boot.elf -f 0x80011f54
```

Para que esta utilidad funcione, debemos tener en el directorio donde la ejecutamos una copia del ejecutable *.elf* generado por nuestro proyecto (el mismo que hemos ejecutado en la consola y que nos ha dado el mensaje de error de la Figura 1). El parámetro *-e* recibe el nombre de este ejecutable *.elf*, y el parámetro *-f* es la dirección de memoria (en hexadecimal) que ha producido el error.

Un ejemplo del resultado de la ejecución de la utilidad *addr2line* puede ser el siguiente:

```
Actor
/home/rabbit/Escritorio/libwiiesp/src/actor.cpp:27
```

Lo cual nos indica que el error se ha producido en la línea 27 del fichero fuente actor.cpp de *LibWiiEsp*.

Un detalle a comentar es que, para salir de la pantalla de error que se muestra en la Figura 1, basta con pulsar el botón *Reset* de la consola, lo que nos devolverá al *HomeBrew Channel*.

En resumen, con la herramienta *addr2line* y la clase *Logger* podemos ir realizando una decente depuración de nuestra aplicación, que aunque hay que reconocer que no es muy cómodo, es mejor que ir dando palos de ciego.