Programación de la Consola Nintendo DS

VJ1214 Consolas y dispositivos de videojuegos

Grado en Diseño y Desarrollo de Videojuegos

Raúl Montoliu Colas
Juan Carlos Fernández Fernández
Maribel Castillo Catalán

Índice general

1. Introducción a la consola Nintendo DS

Este capítulo expone, en primer lugar, el conjunto de consolas de la familia Nintendo y posteriormente describe los principales elementos Hardware de la computadora Nintendo DS, que es la videoconsola a la que se hace referencia en el resto de los capítulos de este libro.

1.1 Las videoconsolas de Nintendo

Existe una amplia variedad de consolas de Nintendo:

- *GameBoy Advance (GBA)*: tiene un procesador *ARM7TDMI*, de 32 bits, junto con un procesador *Z80*, para dar soporte a los juegos de la *GameBoy* clásica (ver Figura 1.1a).
- *Nintendo DS (NDS)*: tiene un procesador ARM9 a 66Mhz y un procesador ARM7 a 33Mhz (ver Figura 1.1b).
- *Nintendo DS Lite*: se diferencia de la NDS normal en su aspecto más estilizado, en mejoras de consumo energético y los diferentes niveles de control de brillo de la pantalla (ver Figura 1.2a).
- *Nintendo DSi*: incorpora dos cámaras de baja resolución, pantallas ligeramente mayores, mejor sonido, más memoria y una nueva ranura para tarjetas *Secure Digital (SD)*. A cambio pierde la ranura de compatibilidad con los cartuchos de *GameBoy Advance (Slot2)* (ver Figura 1.2b).
- *Nintendo DSi XL*: conocida también como *DSi XL*, es prácticamente idéntica a la anterior, salvo que su forma es significativamente mayor.
- *Nintendo 3DS*: permite jugar con juegos y ver películas en 3D. Además, la nueva pantalla ofrece imágenes estereoscópicas sin necesidad de gafas especiales para disfrutar del efecto 3D. Incorpora una pantalla táctil, red inalámbrica ()*WiFi*), sensor de movimiento con giroscopio de tres ejes y acelerómetro de tres ejes (ver Figura 1.3a).
- *Nintendo 2DS*: conserva las mismas funciones y especificaciones que la *Nintendo 3DS*, salvo que no reproduce los videojuegos con efecto 3D, sino en 2D. Además, mantiene el tamaño de las pantallas de la *Nintendo 3DS* (ver Figura 1.3b).
- *New Nintendo 3DS*: esta consola cuenta con botones de colores. Las pantallas de la *New Nintendo 3DS* son 1,2 veces más grandes que las de la *Nintendo 3DS* original, mientras que

(a) *GameBoy Advance* (b) *Nintendo DS*

Figura 1.1: Videoconsolas de Nintendo: GBA y NDS

(a) *Nintendo DS Lite* (b) *Nintendo DSi*

Figura 1.2: Videoconsolas de Nintendo: NDSLite y NDSi

el tamaño de las pantallas de la *New Nintendo 3DS XL* son similares a las de su predecesora. (ver Figura 1.3c). Las ranuras de la tarjeta de juego, del lápiz y del botón de encendido se han trasladado a la base de la consola. Como nuevas características cabe destacar:

- El rastreo facial para que la cámara siga la línea de visión del jugador, de esta forma se amplía la gama de ángulos desde los que se puede ver el efecto 3D estereoscópico del sistema.
- La variación automática del brillo de las pantallas según la iluminación ambiental.
- La transferencia inalámbrica de archivos multimedia entre la consola y un ordenador.
- Una CPU más potente, un ARM11 Dual Core a 532MHz.

1.2 Hardware de la Nintendo DS (NDS)

A continuación se describe el hardware de la *Nintendo DS*:

- **Procesadores**: cuenta con dos procesadores, un ARM9 y un ARM7. El procesador ARM9 se encarga de la lógica principal del programa, mientras que el ARM7, como procesador secundario, se encarga básicamente de gestionar el audio, la red inalámbrica (*WiFi*) y algunas teclas. El hecho de que la consola NDS cuente con dos procesadores implica la generación de dos ejecutables distintos, uno para cada procesador. El ejecutable del ARM7 actúa como esclavo del ARM9, atendiendo peticiones de reproducción de sonido o comunicaciones vía

(a) *Nintendo 3DS* (b) *Nintendo 2DS* (c) *New Nintendo 3DS*

Figura 1.3: Videoconsolas de Nintendo: N3DS, N2DS y NNew3DS

WiFi.

- **Memorias**:
  - Memoria principal: tiene un tamaño de 4 MB. Dicha memoria almacena el ejecutable para el ARM9, así como la gran mayoría de datos del ejecutable. Ambos procesadores pueden acceder a esta memoria en cualquier momento. Si ambos intentan acceder a la vez, será el que tenga mayor prioridad el que accede, quedando el otro a la espera.
  - Memoria de vídeo *VRAM*: tiene 656 KB distribuidos en 9 bancos de memoria de vídeo, que se pueden usar con diferentes propósitos. A lo largo de las sesiones de prácticas se verán más detalles de esta memoria de vídeo.
  - Otras memorias: tiene las pseudo-cachés *WRAM* e *IWRAM* de 96Kb, una memoria *RAM* adicional para la *BIOS* y una memoria virtual para vídeo (*Virtual Video RAM*).
- **Gráficos**: el hardware de vídeo se compone de dos núcleos gráficos 2D, uno principal (*main*) y otro secundario (*sub*). Dichos núcleos se diferencian únicamente en que el motor principal puede *renderizar* tanto la memoria de vídeo virtual sin utilizar el motor 2D, como mapas de bits de 256 colores, así como utilizar el motor 3D para el renderizado de alguno de sus fondos.
- **Sonido**: dispone de altavoces estéreo y cuenta con 16 canales de audio independientes.
- **Comunicación inalámbrica**: soporta el estándar de protocolo de comunicaciones *IEEE 802.11*. El rango de comunicación inalámbrica varía de 10 a 30 metros, dependiendo de las circunstancias.
- **Entrada/Salida**: tiene un puerto para cartuchos de juegos de Nintendo DS y otro para juegos de Game Boy Advance2. La NDS cuenta con una entrada para auriculares estéreo y otra entrada para micrófono.
- **Doble pantalla**: las dos pantallas LCD son de 3 pulgadas. La pantalla inferior emplea tecnología táctil.
- **Temporizador**: cuenta con un reloj de tiempo de real, que puede ser utilizado por una aplicación o juego para definir diferentes respuestas dependiendo de la hora del día.

2. Herramientas para programar la NDS

Este capítulo está dedicado a la instalación de las herramientas necesarias para poder realizar videojuegos en la consola Nintendo DS. Así mismo se verá como poder realizar un primer programa *Hello World*.

2.1 Herramientas de desarrollo para programar con la NDS

2.1.1 Introducción

Se denomina *homebrew* al software *casero* no oficial realizado por programadores, ya sean aficionados o expertos, para cualquier plataforma. Generalmente, esta plataforma suele ser una videoconsola propietaria. El desarrollo de software *casero* está permitido en cualquiera de las consolas de Nintendo, siempre y cuando sea sin ánimo de lucro. En cualquier caso, se debe señalar que no todas las plataformas permiten el *homebrew*. El desarrollo de software para la Nintendo DS se puede realizar de dos maneras diferentes:

- Utilizando el kit comercial de desarrollo de software (*SDK*) de Nintendo.
- Utilizando *DevkitPro*, que es un conjunto de bibliotecas, compiladores y utilidades para desarrollar software para varias plataformas. Además, es libre y de descarga gratuita.

En los apartados siguientes de esta sección se presentarán las principales herramientas existentes que ayudan al desarrollo de aplicaciones para NDS.

2.1.2 DevkitPro

DevkitPro es un conjunto de bibliotecas, compiladores y utilidades que permiten desarrollar aplicaciones para las consolas *Game Boy Advance (GBA)*, *GP32*, *GP2X*, *Playstation Portable (PSP)*, *Nintendo DS* y *GameCube*. *DekvitPro* cuenta con cuatro *toolchains* que permiten escribir aplicaciones y juegos para las consolas citadas:

- *DevkitARM*: utilizado para el desarrollo de aplicaciones para *GBA*, *GP32* y *Nintendo DS*.
- *DevkitGP2X*: utilizado para el desarrollo de aplicaciones para la *GamePark GP2X*.
- *DevkitPPC*: utilizado para el desarrollo de aplicaciones para la *Nintendo GameCube*.
- *DevkitPSP*: utilizado para el desarrollo de aplicaciones para la *Sony PSP*.

2.1.3 DevkitARM

DevkitARM es un *toolchain* de los lenguajes *C* y *C++*, basado en la colección de compiladores *GNU (GCC)*, que permite crear binarios para la *arquitectura ARM*. Incluye todo lo necesario para crear software para la *Nintendo DS*, *GBA* y *GP32*. Las bibliotecas que incluye *DevkitARM* son las siguientes:

- *LibNDS*: anteriormente conocida como *NDSLIB*, es una biblioteca creada por Michael Noland y Jason Rogers. Esta biblioteca sirve como base para el desarrollo de programas para la Nintendo DS. LibNDS soporta casi todas las características de la NDS, incluyendo la pantalla táctil, el micrófono, el hardware 2D, el hardware 3D y las comunicaciones inalámbricas.
- *LibFAT*: contiene una serie de rutinas para leer y escribir en sistemas de ficheros *FAT (File Allocation Table)* como los de las tarjetas *Secure Digital (SD)*, *MultimediaCard (MMC)* o *CompactFlash (CF)*.
- *DSWifi*: permite a los desarrolladores usar la *WiFi* de la NDS de una manera similar a como los ordenadores usan la tarjeta de red inalámbrica.
- *LibGBA*: contiene las funciones necesarias para controlar el hardware de la *Game Boy Advance*.

Algunas de las herramientas más destacadas de *DevkitARM* son las siguientes:

- *Grit (GBA Image Transmogrifier)*: es un conversor de imágenes para la *Game Boy Advance* y la *Nintendo DS*. *Grit* acepta multitud de formatos de archivos (*bmp*, *pcx*, *png*, *gif*, *jpeg*, ...) con cualquier profundidad de bits y obtiene los datos para ser usados directamente en el código de un programa para *GBA* o *NDS*. Los datos que genera *Grit* pueden ser datos de una paleta, datos de teselas, datos de un mapa o datos de un gráfico. Los formatos de salida disponibles son, entre otros, archivo C, archivo binario o archivo *GNU Assembly*. Esta herramienta se empleará más adelante cuando se estudie la parte gráfica de la NDS.
- *arm-eabi-gcc*: es un compilador cruzado que genera código objeto para el ARM7 y el ARM9 a partir de código escrito en los lenguajes *C* o *C++*.
- *arm-eabi-ld*: es un enlazador que genera un archivo ejecutable en el formato estándar *ELF* para el entorno de ejecución ARM7 y ARM9 a partir del código objeto generado por *arm-eabi-gcc*.
- *arm-eabi-objcopy*: es una herramienta que genera los archivos ejecutables reducidos *.arm7* y *.arm9* a partir del archivo ejecutable con formato *ELF*. Esta herramienta reduce al mínimo las necesidades de memoria de la videoconsola. Para ello, extrae exclusivamente lo necesario para poder ejecutar el programa (instrucciones y datos).
- *ndstool*: combina los archivos ejectuables *.arm7* y *.arm9* en un único archivo con extensión *.nds* añadiendo una cabecera descriptiva al comienzo. Opcionalmente, puede combinar junto con los archivos ejecutables otros datos como, por ejemplo, datos de gráficos.
- *dsbuild*: genera un archivo con extensión *.ds.gba*, que permite arrancar el programa desde el *Slot2* (compatible con Game Boy Advance).

2.1.4 Entornos de desarrollo

Se puede definir un *IDE (Integrated Development Environment)* como un programa compuesto por un conjunto de herramientas útiles para un desarrollador de software. Como elementos básicos, un IDE cuenta con un editor de código, un compilador/intérprete y un depurador. También puede dar soporte a más de un lenguaje de programación.

Para desarrollar programas para la NDS se tienen las siguientes opciones:

- Cualquier entorno de desarrollo en C/C++ es válido para desarrollar programas para la NDS, pero suelen requerir dedicar tiempo a configurar tanto los compiladores como los ajustes necesarios de cada proyecto individual.
- Emplear un IDE pensado específicamente para el desarrollo en Nintendo DS. Por ejemplo,

Eclipse Ganymede dispone de un *plugin NDS*.

2.1.5 Emuladores

Un *emulador* es un programa que se ejecuta en un computador (sistema anfitrión del emulador) y se encarga de recrear el comportamiento de un computador diferente (sistema objetivo del emulador). La ventaja de utilizar un emulador de NDS es que no se necesita tener ni videoconsola ni cartuchos especiales. Sin embargo, las funcionalidades de la NDS que se soportan dependen del emulador utilizado.

WinDS Pro es un pack de emuladores para la NDS. En concreto dispone de los siguientes:

- Citra: emulador de Nintendo 3DS
- DeSmuME: emulador de Nintendo DS
- No$gba: emulador de Nintendo DS y Game Boy Advance
- VBA: emulador de Game Boy, Game Boy Color y Game Boy Advance

2.2 Instalación del entorno de desarrollo en Windows

Se puede encontrar información sobre el proceso de instalación en la siguiente página web:

`http://snipah.com/index.php?option=com_content&view=article&id=44&Itemid=53`

2.2.1 Instalación de *devkitpro*

Se accede a la página web:

`http://devkitpro.org/`

Se pulsa en *For instructions on installing the toolchains see our Getting Started pages* y posteriormente elegir *Windows Installer/Updater package* y descargar la última versión (p. ej. *devkitProUpdater-1.6.0.exe*).

2.2.2 Instalación de *WinDS Pro*

Se puede descargar la última versión de *WinDS Pro* de la siguiente página web:

`https://windsprocentral.blogspot.com.es/2016/10/winds-pro.html`

2.2.3 Instalación de *eclipse* con el *plugin de NDS*

Existen dos posibilidades:

1. Instalar el *eclipse* que ya contiene el *plugin de NDS*.
2. Instalar *eclipse* y después añadir el *plugin NDS ManagedBuilder*.

A continuación se describen los pasos a seguir para ambas opciones.

Instalar *eclipse* que ya contiene el *plugin de NDS*

En la página web indicada al comienzo de esta sección (*Instalación del entorno de desarrollo en Windows*), en concreto en *Full Eclipse packages* se pulsa en *Eclipse Full Package Win32 - Zip-Format* para descargar el fichero

`eclipse-cpp-ganymede-win32_nds.zip`

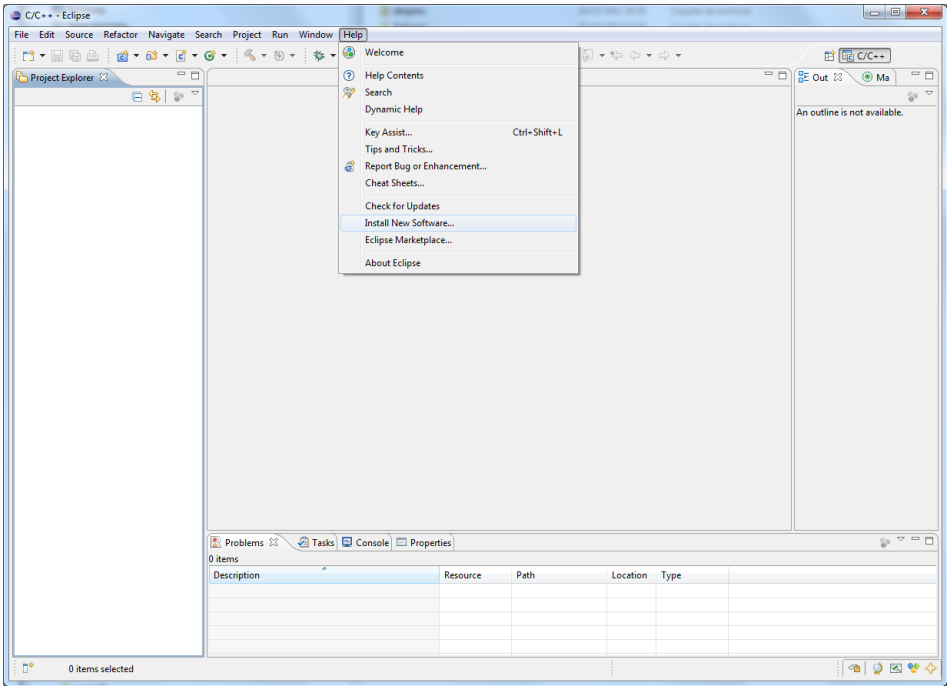

Figura 2.1: Instalación del plugin NDS en eclipse (parte 1).

Instalar *eclipse* y después añadir el *plugin NDS ManagedBuilder*

En este caso el primer paso consiste en la instalación de *eclipse*. Por cuestiones de compatibilidad se escoge la versión *Helios*. Se accede a la página web:

`http://www.eclipse.org/downloads/packages/release/Helios/R`

y se descarga la versión *Eclipse IDE for C/C++ Developers* para el sistema operativo apropiado a las necesidades del equipo con el que se va a trabajar. Una vez descomprimido el fichero ya se tiene instalado *eclipse*. El siguiente paso es instalar el plugin mediante el actualizador del propio *eclipse* que se encuentra en *Help->Install New Software*, tal y como muestra la Figura 2.1.

A continuación en la ventana *Available Software* se pulsa el botón *Add*, y se introduce la siguiente información:

- Name: *NDS Manager builder*
- Location: *http://dev.snipah.com/nds/updater*

Esta operación se refleja en la Figura 2.2.

Para que el proceso se realice de forma adecuada hay que tener la precaución de desactivar la opción *Group items by category*. De esta forma aparecerá el software buscado, debiendo activarse las casillas correspondientes a *devkitARM*, tal y como muestra la Figura 2.3.

Después de pulsar en sucesivos botones *Next* y aceptar la licencia, comienza la instalación del software. Durante dicho proceso puede aparecer la Figura 2.4.

Simplemente se pulsa en *OK* para continuar el proceso de instalación. Una vez finalizada la instalación se debe reiniciar *eclipse*.

Como comprobación de que todo ha ido correctamente, a la hora de crear el proyecto se debe observar que aparece algo parecido a lo mostrado en la Figura 2.5.

Se elige *Empty Project (libnds)*, y si después de pulsar en *Next* aparece lo mostrado en la Figura 2.6, entonces todo está correcto.

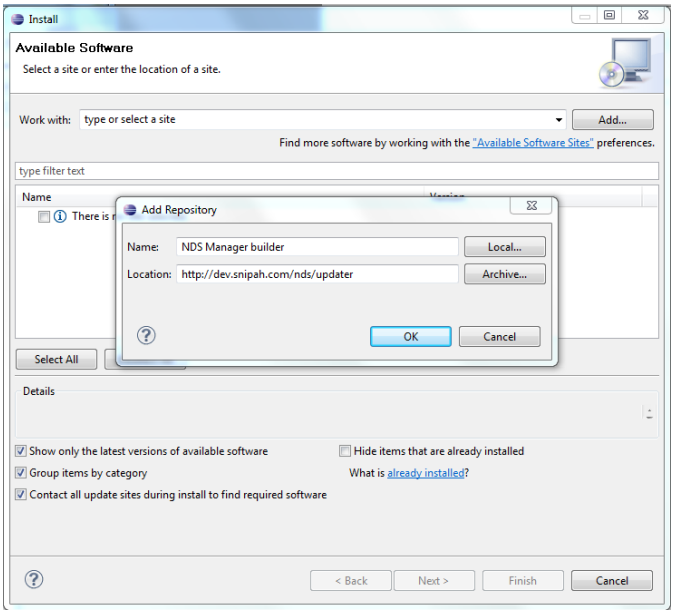

Figura 2.2: Instalación del plugin NDS en eclipse (parte 2).

2.3 Nuestro primer programa para NDS en Windows

En esta sección se van a ver los pasos para realizar nuestro primer programa para la NDS en el sistema operativo *Windows*.

2.3.1 Desarrollar código para la NDS sin emplear Eclipse

En este apartado se va a crear el primer programa en la NDS sin emplear *Eclipse* como herramienta de desarrollo.

Creación de la estructura de ficheros

Se puede emplear como punto de partida el ejemplo *hello_world* que aparece en el directorio *nds* del directorio *examples* de *DevkitPro*. En el laboratorio de prácticas, *DevkitPro* se encuentra en el directorio *C:*. En el directorio donde se vayan a almacenar los programas a desarrollar se crea un nuevo directorio que identifique el programa a desarrollar (p.ej. *ejemplo*). Dentro de ese directorio se crea el directorio *source*, que contendrá los ficheros necesarios para el código a desarrollar (p.ej. *main.c*). En el directorio *ejemplo* se copia el fichero *Makefile* del ejemplo *hello_world* de *DevkitPro*. De esta forma la estructura de ficheros que se tiene es la siguiente:

```
c:\mis_ejemplos
  - directorio ejemplo
    - fichero Makefile
    - directorio source
    - fichero main.c
```

Edición del fichero ejemplo

Para familiarizarse con el entorno de desarrollo de aplicaciones para Nintendo DS, se va a utilizar como ejemplo una aplicación en la que aparezca un saludo con el nombre del desarrollador del programa. Para escribir este código se puede emplear cualquier editor de texto. Según esto, el código del programa a desarrollar (*main.c*) es el siguiente:

```
1 #include<nds.h>
2 #include<stdio.h>
3 int main(void) {
```

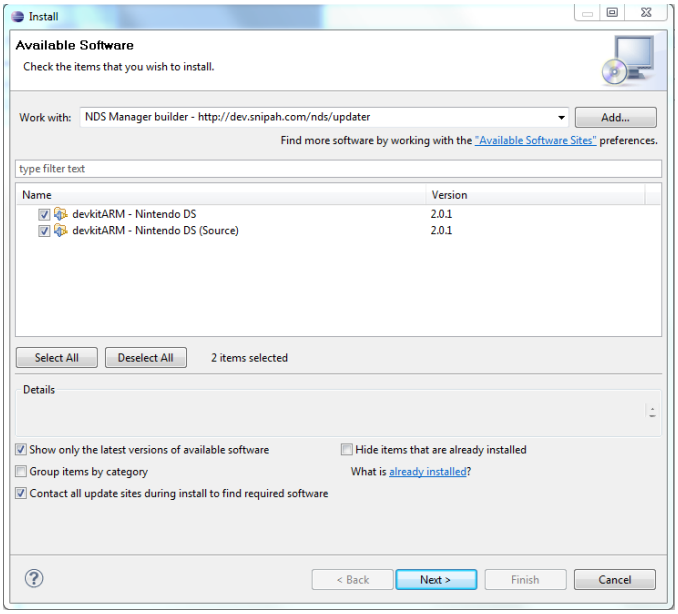

Figura 2.3: Instalación del plugin NDS en eclipse (parte 3).

```
4       consoleDemoInit();
5       iprintf("Hola mundo");  // Imprimir el mensaje
6       while(1) {} // Bucle que no hace nada.
7   }
```

En dicho código cabe destacar lo siguiente:

- *consoleDemoInit*: inicializa una consola de texto predeterminada, sin permitir elegir la pantalla donde se imprimie el texto. En este caso, será la pantalla inferior de la videoconsola. Se verán más detalles de cómo seleccionar la pantalla en la que se visualiza información en prácticas posteriores.
- *iprintf("Hola mundo")*: esta función imprime texto con formato, soportando solo números enteros.

Compilación del fichero ejemplo

El siguiente paso es compilar el programa, para ello se abre el *símbolo del sistema*. Una vez se está en el directorio *ejemplo* creado, se ejecuta el comando *make*:

```
C:\mis_ejemplos\ejemplo>dir
29/07/2013  15:10    <DIR>          .
29/07/2013  15:10    <DIR>          ..
02/04/2012  22:02             4.903 Makefile
29/07/2013  15:10    <DIR>          source
               1 archivos          4.903 bytes
               3 dirs  51.779.096.576 bytes libres


C:\mis_ejemplos\ejemplo>make
main.c
arm-none-eabi-gcc -MMD -MP -MF /d/mis_ejemplos/ejemplo/build/main.d -g -Wall
-O2 -march=armv5te -mtune=arm946e-s -fomit-frame-pointer -ffast-ma
th -mthumb -mthumb-interwork -I/d/mis_ejemplos/ejemplo/include
-I/d/mis_ejemplos/ejemplo/build -I/c/devkitPro/libnds/include
-I/d/mis_ejemplos/ejemplo/build -DARM9 -c /d/mis_ejemplos/ejemplo/source/main.c
-o main.o
linking ejemplo.elf
Nintendo DS rom tool 1.50.1 - Jun 19 2012
by Rafael Vuijk, Dave Murphy, Alexei Karpenko
```

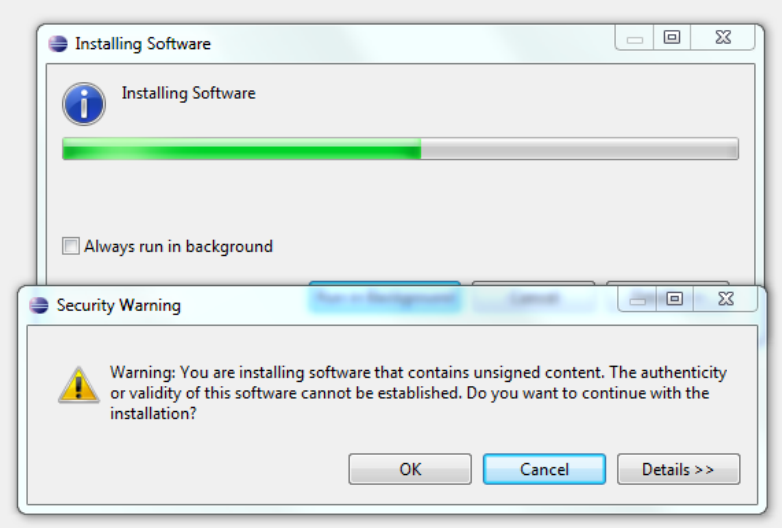

Figura 2.4: Instalación del plugin NDS en eclipse (parte 4).

```
built ... ejemplo.nds
```

Si no se han producido errores de compilación aparecerán los ficheros *ejemplo.elf* y *ejemplo.nds*.

```
C:\mis_ejemplos\ejemplo>dir
29/07/2013  15:15    <DIR>          .
29/07/2013  15:15    <DIR>          ..
29/07/2013  15:15    <DIR>          build
29/07/2013  15:15           234.949 ejemplo.elf
29/07/2013  15:15           134.208 ejemplo.nds
02/04/2012  22:02             4.903 Makefile
29/07/2013  15:10    <DIR>          source
               3 archivos        374.060 bytes
               4 dirs  51.778.568.192 bytes libres
```

El primero (*ejemplo.elf*) es el que contiene la información de depuración, por tanto, el depurador tendrá que trabajar necesariamente con él. Sin embargo, la consola (o el emulador) solo será capaz de ejecutar la imagen del cartucho *ejemplo.nds*. También se ha creado el directorio *build*, que por ahora no tiene interés para lo que se se está desarrollando. Para borrar todos los ficheros y directorios creados durante la compilación se puede ejecutar *make clean*.

Si se produjese un error relacionado con que no encuentra el compilador se puede realizar lo siguiente:

- Hacer una copia de los siguientes ficheros que se encuentran en *C:\ devkitPro \ devkitARM \ bin*:
  - arm-none-eabi-as
  - arm-none-eabi-g++
  - arm-none-eabi-gcc
  - arm-none-eabi-gdb
  - arm-none-eabi-objcopy
- Renombrar las copias con los siguientes nombres:
  - arm-eabi-as
  - arm-eabi-g++
  - arm-eabi-gcc
  - arm-eabi-gdb
  - arm-eabi-objcopy

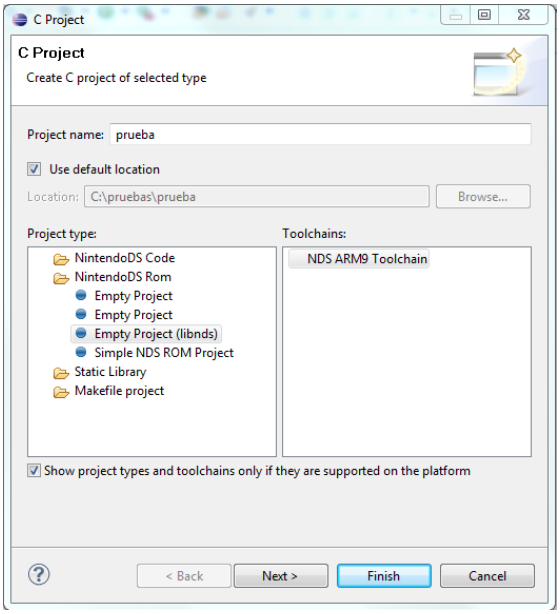

Figura 2.5: Instalación del plugin NDS en eclipse (parte 5).

Ejecución del fichero ejemplo en el emulador

Si no se ha producido ningún problema en la compilación, la salida del programa se puede ver en el emulador. Una vez abierto *WinDS Pro*, si se escoge el emulador *No$gba*, se pulsa en *File->Cartridge Menu (File Name)* y se busca el fichero *.nds* que nos interesa. En nuestro caso, y una vez elegido *ejemplo.nds* aparece la ventana en el emulador mostrada en la Figura 2.7.

Si se escoge el emulador *DeSmuME*, se pulsa en *File->Open ROM* y se busca el fichero *.nds* que nos interesa.

2.3.2 Desarrollar código para la NDS empleando Eclipse

En este apartado se va a crear el primer programa en la NDS empleando *Eclipse* como herramienta de desarrollo.

Creación de un proyecto para NDS

En el laboratorio de prácticas, *Eclipse* se encuentra en el directorio *C:*. Al iniciar *Eclipse* se pide el directorio donde se almacenará el proyecto a crear, tal como muestra la Figura 2.8.

Para crear un nuevo proyecto se debe pulsar en la ventana principal de *Eclipse* en *File->New->C project* (ver Figura 2.9).

Aparece la ventana (*C Project*) mostrada en la Figura 2.10.
en la que se debe configurar lo siguiente:

- El nombre del proyecto (p.ej. *ejemplo*).
- En *Project types* se selecciona *Nintendo DS Rom->Empty Project (libnds)*.
- Se pulsa en *Next*.

Aparece una ventana de *Select Configurations* en la que se pulsa en *Finish*.

Una vez creado el proyecto, si lo que aparece es la siguiente ventana se debe elegir el icono de *workbench*, tal y como se muestra en la Figura 2.11.

De esta forma, el proyecto creado aparecerá en la ventana de *workbench* de *Eclipse* (ver Figura 2.12).

Esta ventana podría aparecer directamente sin necesidad de elegir el icono de *workbench* de *Eclipse*.

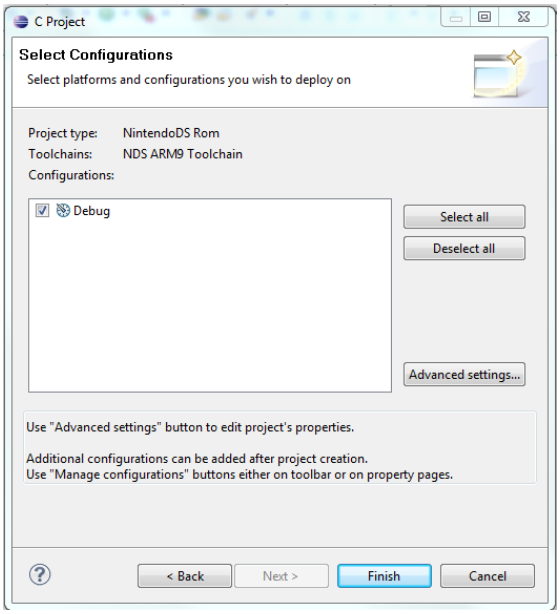

Figura 2.6: Instalación del plugin NDS en eclipse (parte 6).

Configuración del proyecto

Para iniciar la configuración del proyecto, se debe pulsar en la ventana principal de *Eclipse* en *Project Properties*, teniendo la precaución de tener activado el proyecto que se desea. Una vez realizada esta operación aparece la ventana (*Properties for ejemplo*) (ver Figura 2.13).

En dicha ventana se debe desplegar la pestaña *C/C++ Build*. Se resalta dicha opción. En el cuadro *Builder* de la pestaña *Builder Settings* se cambia la opción *Builder Type* a *Internal Builder* (ver Figura 2.14).

En la pestaña *C/C++ Build* se resalta *Settings*, apareciendo la ventana que se muestra en la Figura 2.15.

Se escoge *devkitARM C Linker->ARM* y se desactiva la opción *No FPU*. Finalmente se pulsa en *OK* (ver Figura 2.16).

Edición del fichero ejemplo

Para familiarizarse con el entorno de desarrollo de *Eclipse* para NDS, se utiliza el mismo ejemplo que el del apartado anterior. En primer lugar, se debe crear un fichero fuente en *lenguaje C* dentro del proyecto actual. Para ello se pulsa en *File->New->Source File* (ver Figura 2.17).

Aparece una nueva ventana (*New Source File*) (ver Figura 2.18).
en la que se debe realizar lo siguiente:

- En *Source File* se introduce *main.c*.
- En *Template* se selecciona *None*.
- Se pulsa en *Finish*.

A continuación en la ventana que hace referencia a *main.c* se introduce el mismo código que el del apartado 2.3.1.

Compilación del fichero ejemplo

El siguiente paso es compilar el programa, para ello se elige *Project->Build Project*. Si no se han producido errores de compilación aparecerán los ficheros *ejemplo.elf* y *ejemplo.nds* en el directorio *Debug*, tal y como se puede comprobar en la Figura 2.19.

Si se produjese un error relacionado con que no encuentra el compilador se puede realizar lo siguiente:

Figura 2.7: Ejecución del programa ejemplo en el emulador *No$gba*.

- Hacer una copia de los siguientes ficheros que se encuentran en *C:\ devkitPro \ devkitARM \ bin*:
  - arm-none-eabi-as
  - arm-none-eabi-g++
  - arm-none-eabi-gcc
  - arm-none-eabi-gdb
  - arm-none-eabi-objcopy
- Renombrar las copias con los siguientes nombres:
  - arm-eabi-as
  - arm-eabi-g++
  - arm-eabi-gcc
  - arm-eabi-gdb
  - arm-eabi-objcopy

Ejecución del fichero ejemplo en el emulador

Una vez abierto *WinDS Pro*, si se escoge el emulador *No$gba*, se pulsa en *File->Cartridge Menu (File Name)* y se busca el fichero *.nds* que nos interesa. En nuestro caso, y una vez elegido *ejemplo.nds* aparece la ventana en el emulador mostrada en la Figura 2.7.

Si se escoge el emulador *DeSmuME*, se pulsa en *File->Open ROM* y se busca el fichero *.nds* que nos interesa.

2.4 Instalación del entorno de desarrollo en Linux

Las operaciones a seguir para instalar el entorno de desarrollo en Linux son las siguientes:

1. Instalación de *devkitpro*. Se accede a la página web:

   `http://devkitpro.org/`

   Se pulsa en *For instructions on installing the toolchains see our Getting Started pages* y posteriormente elegir *Manual instructions for installing devkitARM* y seguir los pasos que

Figura 2.8: Creación de un proyecto para NDS usando Eclipse (parte 1).

aparecen en dicha página web.

2. Instalación de *WinDS Pro*. Se puede descargar la última versión de *WinDS Pro* de la siguiente página web:

   `https://windsprocentral.blogspot.com.es/2016/10/winds-pro.html`

3. Instalación de *desmume*. Se pueden seguir los pasos que aparecen en la siguiente página web:

   `http://wiki.desmume.org/index.php?title=Installing_DeSmuME_from_source_on_Linux`

   Recomendable emplear la opción *Install desmume from svn*.
4. Instalación de *eclipse Helios*. Se siguen los mismos pasos que los indicados para Windows.
5. Instalación del *plugin NDS ManagedBuilder*. En este caso será necesario instalar el plugin mediante el actualizador del propio Eclipse que se encuentra en *Help->Install New Software*. Para ello se emplea la *url* de actualizaciones del *NDS Managed builder (http://dev.snipah.com/nds/updater)*, que se deberá especificar en la ventana mostrada en la Figura 2.20.

Hay que tener la precaución de desactivar la opción *Group items by category*. Después de aceptar la licencia se instalará el plugin. Una vez finalizada la instalación se debe reiniciar Eclipse.

2.5 Nuestro primer programa para NDS en Linux

En esta sección se van a ver los pasos para realizar nuestro primer programa para la NDS en el sistema operativo *Linux*. **En el laboratorio se debe entrar con la cuenta *usuario***. Antes de nada se debe comprobar que las siguientes variables de entorno se encuentran en el fichero *.bash_profile* del *usuario*:

```
export DEVKITPRO=/opt/devkitpro/
export DEVKITARM=/opt/devkitpro/devkitARM/
```

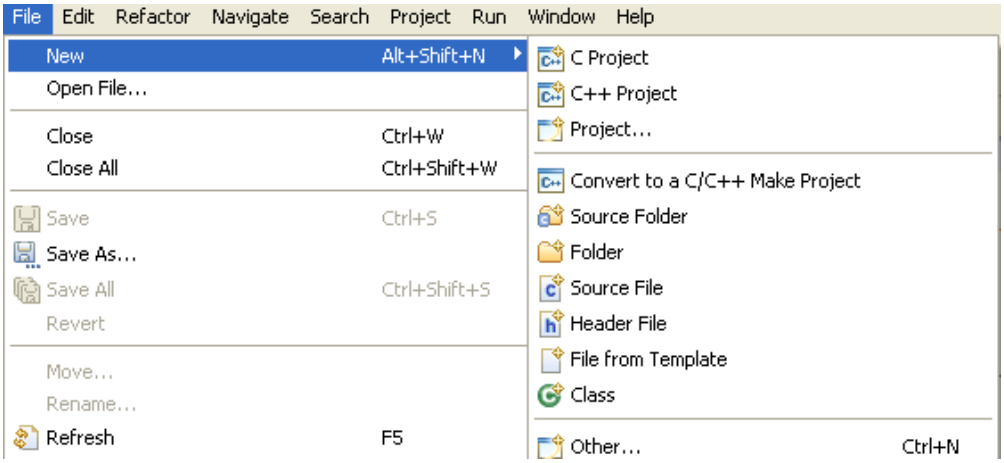

Figura 2.9: Creación de un proyecto para NDS usando Eclipse (parte 2).

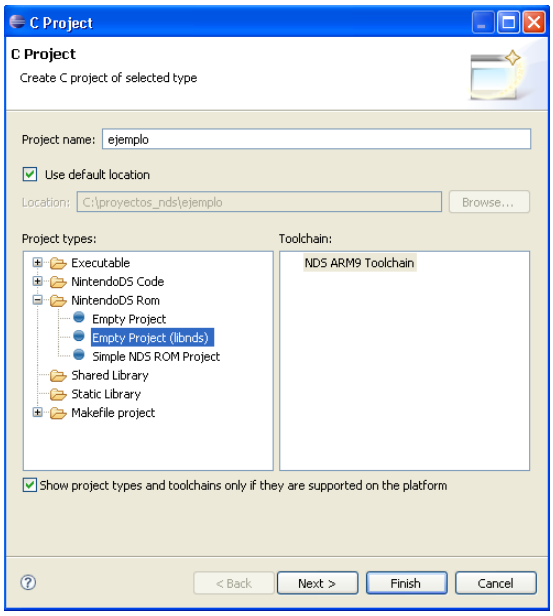

Figura 2.10: Creación de un proyecto para NDS usando Eclipse (parte 3).

2.5.1 Desarrollar código para la NDS sin emplear Eclipse

En este apartado se va a crear el primer programa en la NDS sin emplear *Eclipse* como herramienta de desarrollo.

Creación de la estructura de ficheros

Se puede emplear como punto de partida el ejemplo *hello_world* que aparece en el directorio *nds* del directorio *examples* de *DevkitPro*. En el laboratorio de prácticas, *DevkitPro* se encuentra en el directorio */opt/devkitpro*. En el directorio donde se vayan a almacenar los programas a desarrollar se crea un nuevo directorio que identifique el programa a desarrollar (p.ej. *ejemplo*). Dentro de ese directorio se crea el directorio *source*, que contendrá los ficheros necesarios para el código a desarrollar (p.ej. *main.c*). En el directorio *ejemplo* se copia el fichero *Makefile* del ejemplo *hello_world* de *DevkitPro*. De esta forma la estructura de ficheros que se tiene es la siguiente:

```
/home/usuario/mis_ejemplos
  - directorio ejemplo
```

Figura 2.11: Creación de un proyecto para NDS usando Eclipse (parte 4).

```
- fichero Makefile
- directorio source
- fichero main.c
```

Edición del fichero ejemplo

Para familiarizarse con el entorno de desarrollo de aplicaciones para Nintendo DS, se va a utilizar como ejemplo una aplicación en la que aparezca un saludo con el nombre del desarrollador del programa. Para escribir este código se puede emplear cualquier editor de texto.

Según esto, el código del programa a desarrollar (*main.c*) es el siguiente:

```
1  #include<nds.h>
2  #include<stdio.h>
3  int main(void)
4  {
5    consoleDemoInit();
6      iprintf("Hola Juan");  // Imprimir el mensaje
7      while(1){} // Bucle que no hace nada.
8      return 0; // Finalizar el programa
9  }
```

Compilación del fichero ejemplo

El siguiente paso es compilar el programa, para ello se abre el *Terminal (Sistema->Terminal)*. Una vez se está en el directorio *ejemplo* creado, se ejecuta el comando *make*:

```
[usuario@labsop02 ejemplo]# make
main.c
arm-none-eabi-gcc -MMD -MP -MF /root/mis_ejemplos/ejemplo/build/main.d -g -Wall
-O2 -march=armv5te -mtune=arm946e-s -fomit-frame-pointer -ffast-math
-mthumb -mthumb-interwork -I/root/mis_ejemplos/ejemplo/include
-I/root/mis_ejemplos/ejemplo/build -I/opt/devkitpro//libnds/include
-I/root/mis_ejemplos/ejemplo/build -DARM9 -c
```

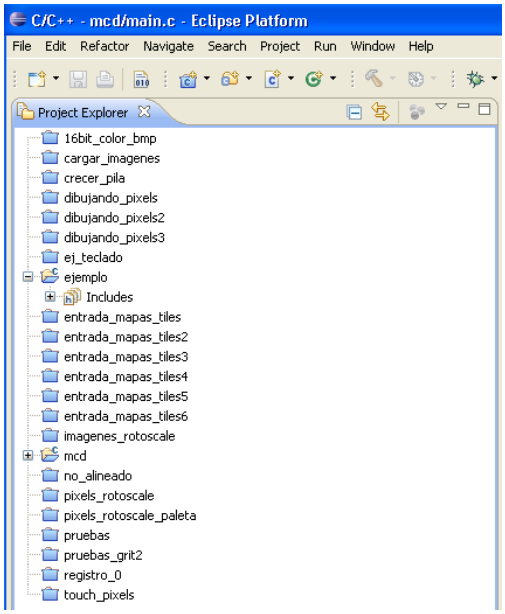

Figura 2.12: Creación de un proyecto para NDS usando Eclipse (parte 5).

```
/root/mis_ejemplos/ejemplo/source/main.c -o main.o
linking ejemplo.elf
Nintendo DS rom tool 1.50.1 - Jun 19 2012
by Rafael Vuijk, Dave Murphy, Alexei Karpenko
built ... ejemplo.nds
```

Si no se han producido errores de compilación aparecerán los ficheros *ejemplo.elf* y *ejemplo.nds*.

```
[usuario@labsop02 ejemplo]# ls -l
total 332
drwxr-xr-x 2 root root   4096 sep  5 18:01 build
-rwxr-xr-x 1 root root 234909 sep  5 18:01 ejemplo.elf
-rw-r--r-- 1 root root 134208 sep  5 18:01 ejemplo.nds
-rwxr-xr-x 1 root root   4903 abr  2  2012 Makefile
drwxr-xr-x 2 root root   4096 sep  5 18:00 source
```

Para borrar todos los ficheros y directorios creados durante la compilación se puede ejecutar *make clean*.

Si se produjese un error relacionado con que no encuentra el compilador se puede realizar lo siguiente:

- Hacer una copia de los siguientes ficheros que se encuentran en */opt/devkitpro/devkitARM/bin*:
  - arm-none-eabi-as
  - arm-none-eabi-g++
  - arm-none-eabi-gcc
  - arm-none-eabi-gdb
  - arm-none-eabi-objcopy
- Renombrar las copias con los siguientes nombres:
  - arm-eabi-as
  - arm-eabi-g++
  - arm-eabi-gcc

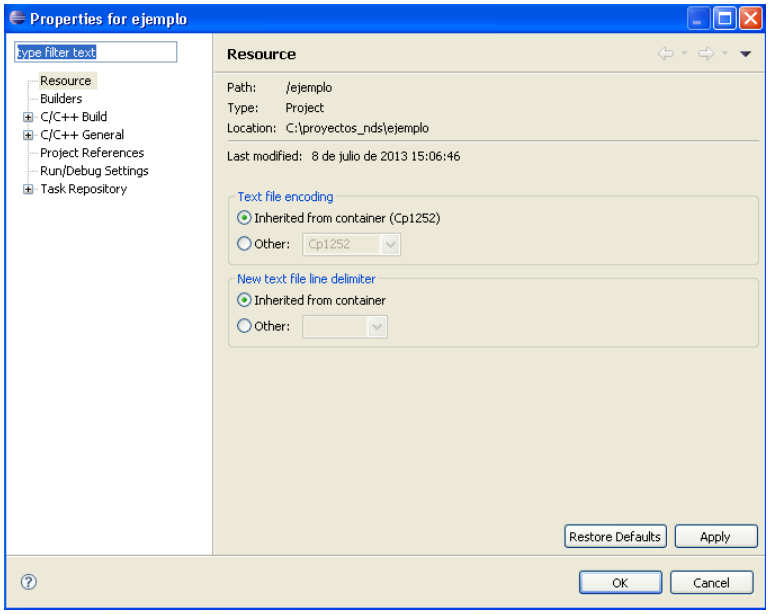

Figura 2.13: Creación de un proyecto para NDS usando Eclipse (parte 6).

- arm-eabi-gdb
- arm-eabi-objcopy

Ejecución del fichero ejemplo en el emulador

Si no se ha producido ningún problema en la compilación, la salida del programa se puede ver en el emulador. Una vez abierto *WinDS Pro*, si se escoge el emulador *No$gba*, se pulsa en *File->Cartridge Menu (File Name)* y se busca el fichero *.nds* que nos interesa. Si se escoge el emulador *DeSmuME*, se pulsa en *File->Open ROM* y se busca el fichero *.nds* que nos interesa.

2.5.2 Desarrollar código para la NDS empleando Eclipse

En este apartado se va a crear el primer programa en la NDS empleando *Eclipse* como herramienta de desarrollo.

Creación de un proyecto para NDS

En el laboratorio de prácticas, *Eclipse* se encuentra en el directorio */opt/eclipse-helios*. Al iniciar *Eclipse* se pide el directorio donde se almacenará el proyecto a crear (ver Figura 2.21).

Para crear un nuevo proyecto se debe pulsar en la ventana principal de *Eclipse* en *Archivo->Nuevo->Proyecto*. Aparece la ventana (*Proyecto nuevo*) mostrada en la Figura 2.22.

Se escoge *C/C++->Proyecto en C* y se pulsa en *Siguiente*, apareciendo la ventana *Proyecto C* mostrada en la Figura 2.23.

En dicha ventana se debe configurar lo siguiente:

- El nombre de proyecto (p.ej. *ejemplo*).
- En *Project type* se selecciona *NintendoDS Rom->Empty Project (libnds)*.
- Se pulsa en *Siguiente*.

Aparece una ventana de *Select Configurations* en la que se pulsa en *Finalizar*.

De esta forma, el proyecto creado aparecerá en la ventana de *workbench* de *Eclipse*, lo que se muestra en la Figura 2.24.

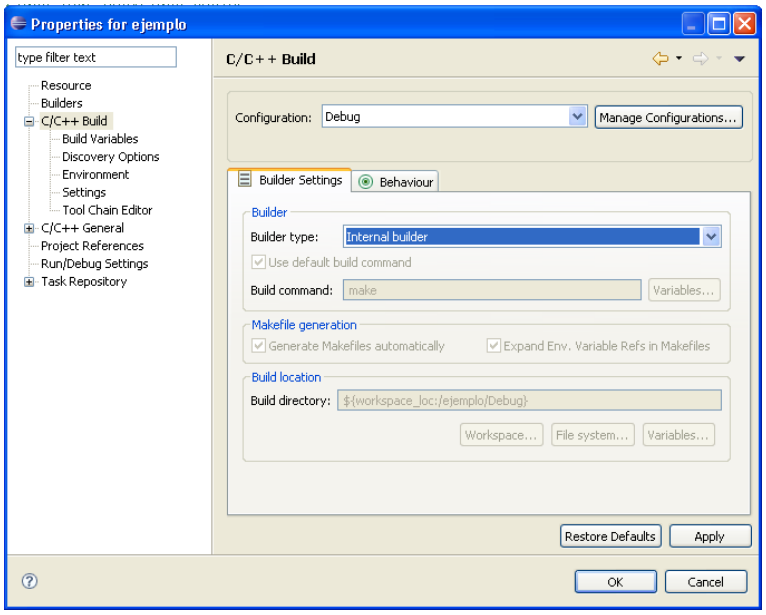

Figura 2.14: Creación de un proyecto para NDS usando Eclipse (parte 7).

Configuración del proyecto

Para iniciar la configuración del proyecto, se debe pulsar en la ventana principal de *Eclipse* en *Proyecto->Propiedades*, teniendo la precaución de tener activado el proyecto que se desea. Una vez realizada esta operación aparece la ventana (*Propiedades de ejemplo*) (ver Figura 2.25).

En dicha ventana se debe desplegar la pestaña *C/C++ Build*. Se resalta dicha opción. En el cuadro *Constructor* de la pestaña *Builder Settings* se cambia la opción *Builder Type* a *Internal Builder* (ver Figura 2.26).

En la pestaña *C/C++ Build* se resalta *Valores*, apareciendo la ventana mostrada en la Figura 2.27.

Se escoge *devkitARM C Linker->ARM* y se desactiva la opción *No FPU*. Finalmente se pulsa en *Aceptar* (ver Figura 2.28).

Edición del fichero ejemplo

Para familiarizarse con el entorno de desarrollo de *Eclipse* para NDS, se utiliza el mismo ejemplo que el del apartado anterior. En primer lugar, se debe crear un fichero fuente en *lenguaje C* dentro del proyecto actual. Para ello se pulsa en *Archivo->Nuevo->Source File*. Aparece una nueva ventana (*New Source File*) (ver Figura 2.29).
en la que se debe realizar lo siguiente:

- En *Source File* se introduce *main.c*.
- En *Template* se selecciona *Ninguno*.
- Se pulsa en *Finalizar*.

A continuación en la ventana que hace referencia a *main.c* se introduce el mismo código que el del apartado 2.3.1.

Compilación del fichero ejemplo

El siguiente paso es compilar el programa, para ello se elige *Proyecto->Construir proyecto*. Si no se han producido errores de compilación aparecerá el fichero *ejemplo.nds* en el directorio *Debug*. Si se produjese un error relacionado con que no encuentra el compilador se puede realizar lo siguiente:

- Hacer una copia de los siguientes ficheros que se encuentran en */opt/devkitpro/devkitARM/bin*:

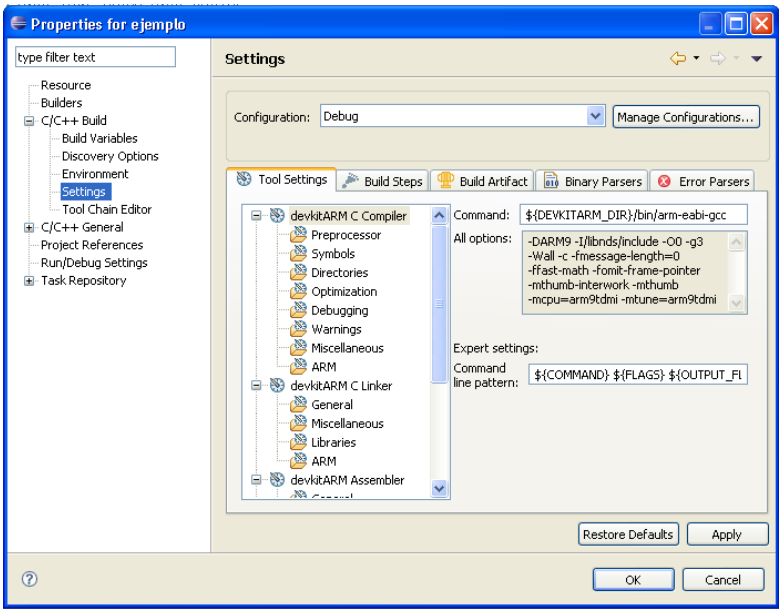

Figura 2.15: Creación de un proyecto para NDS usando Eclipse (parte 8).

- arm-none-eabi-as
- arm-none-eabi-g++
- arm-none-eabi-gcc
- arm-none-eabi-gdb
- arm-none-eabi-objcopy

- Renombrar las copias con los siguientes nombres:
  - arm-eabi-as
  - arm-eabi-g++
  - arm-eabi-gcc
  - arm-eabi-gdb
  - arm-eabi-objcopy

Ejecución del fichero ejemplo en el emulador

Una vez abierto *WinDS Pro*, si se escoge el emulador *No$gba*, se pulsa en *File->Cartridge Menu (File Name)* y se busca el fichero *.nds* que nos interesa. Si se escoge el emulador *DeSmuME*, se pulsa en *File->Open ROM* y se busca el fichero *.nds* que nos interesa.

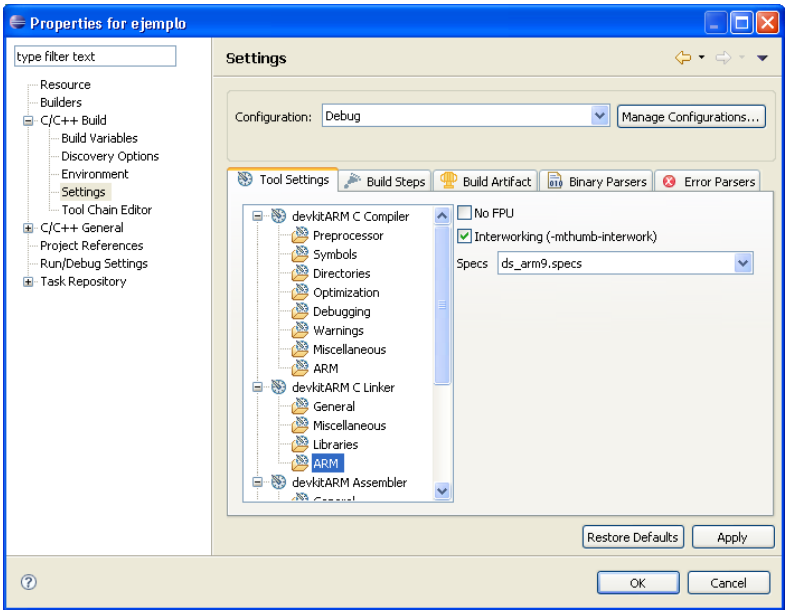

Figura 2.16: Creación de un proyecto para NDS usando Eclipse (parte 9).

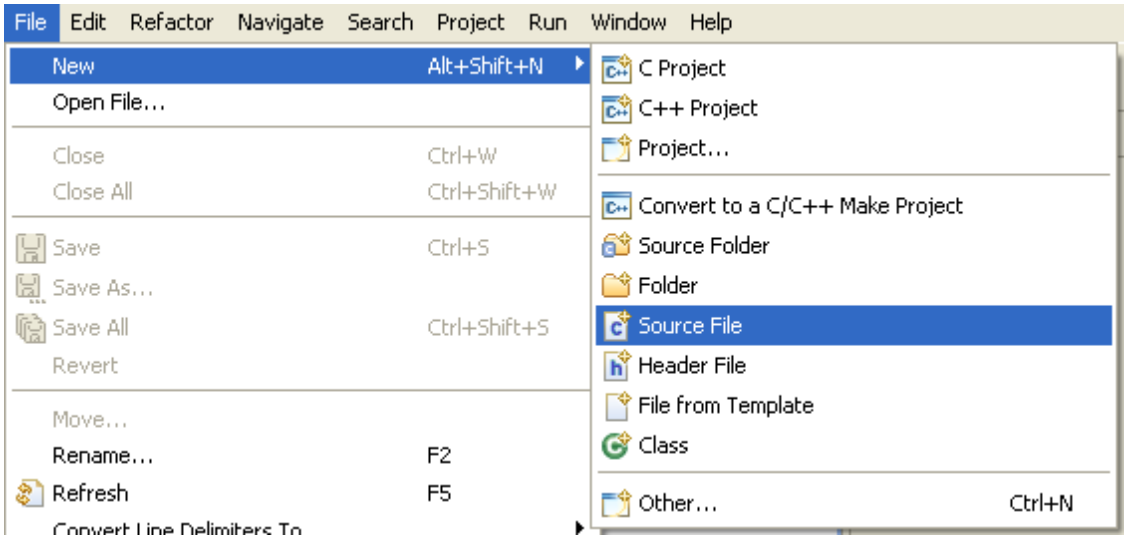

Figura 2.17: Creación de un proyecto para NDS usando Eclipse (parte 10).

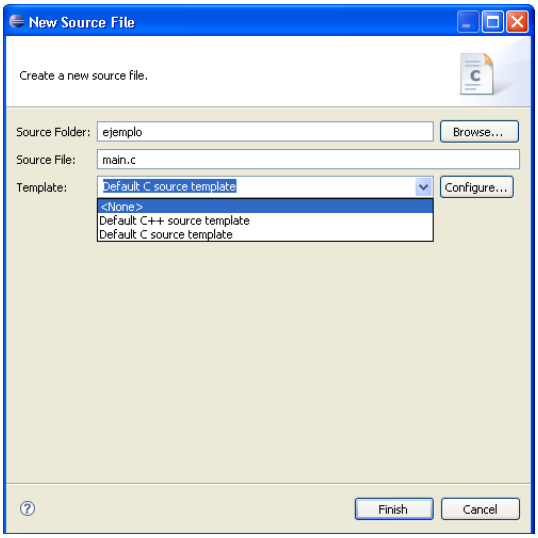

Figura 2.18: Creación de un proyecto para NDS usando Eclipse (parte 11).

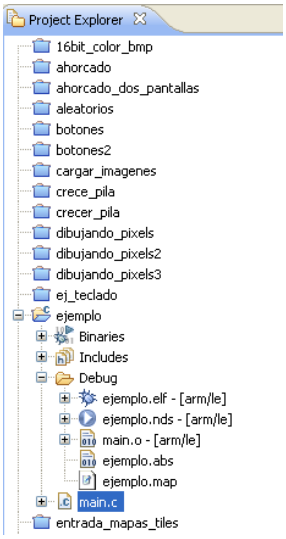

Figura 2.19: Creación de un proyecto para NDS usando Eclipse (parte 12).

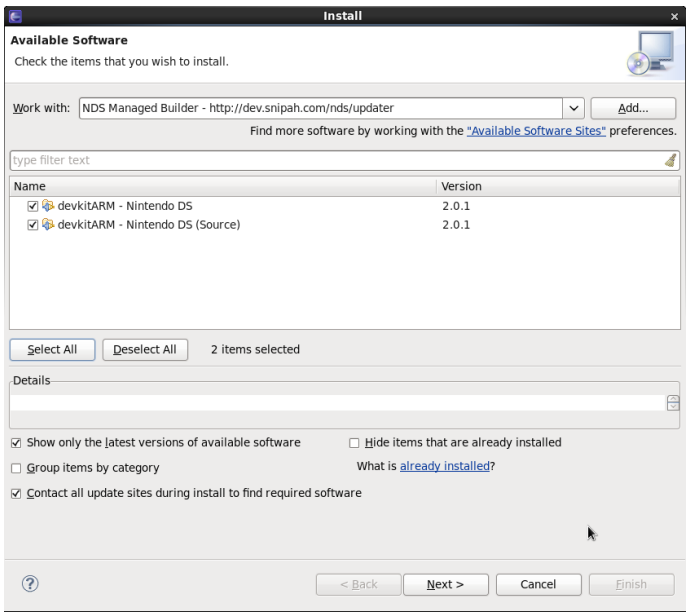

Figura 2.20: Instalación del *plugin NDS ManagedBuilder*

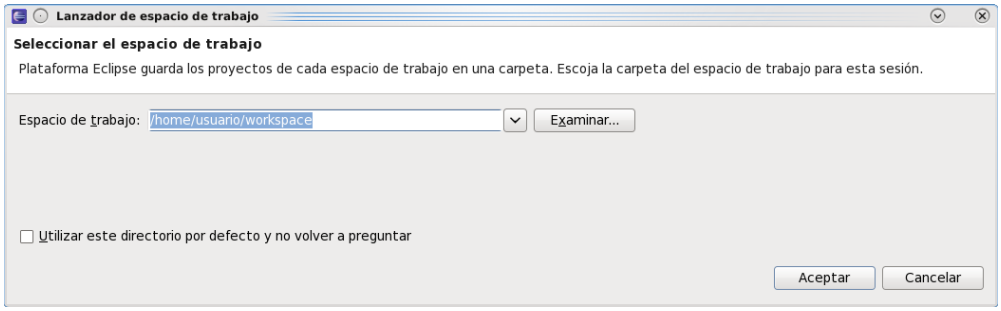

Figura 2.21: Creación de un proyecto para NDS usando Eclipse en Linux (parte 1).

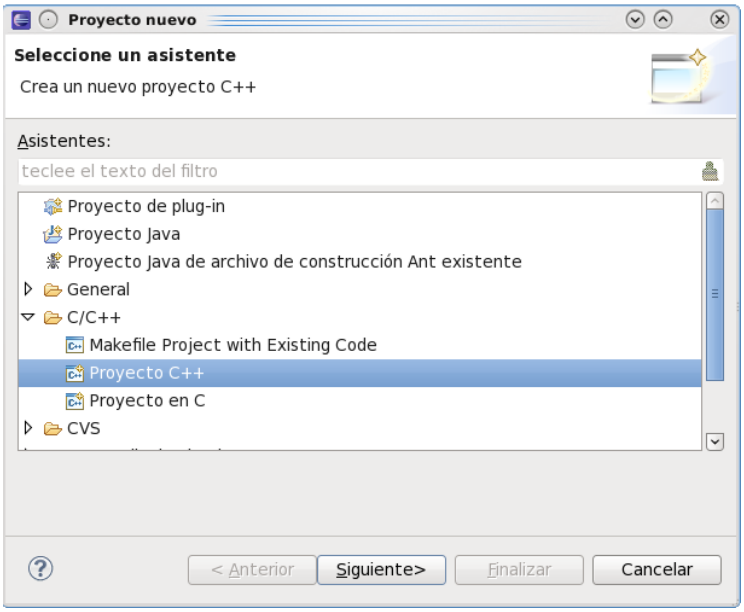

Figura 2.22: Creación de un proyecto para NDS usando Eclipse en Linux (parte 2).

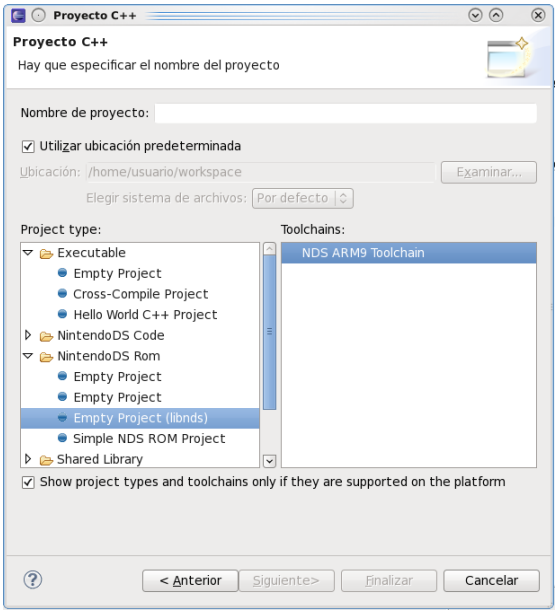

Figura 2.23: Creación de un proyecto para NDS usando Eclipse en Linux (parte 3).

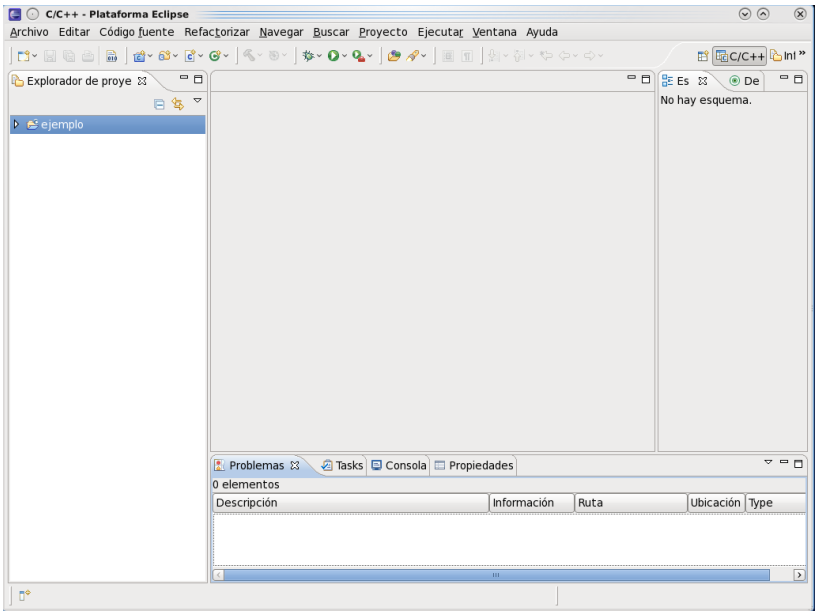

Figura 2.24: Creación de un proyecto para NDS usando Eclipse en Linux (parte 4).

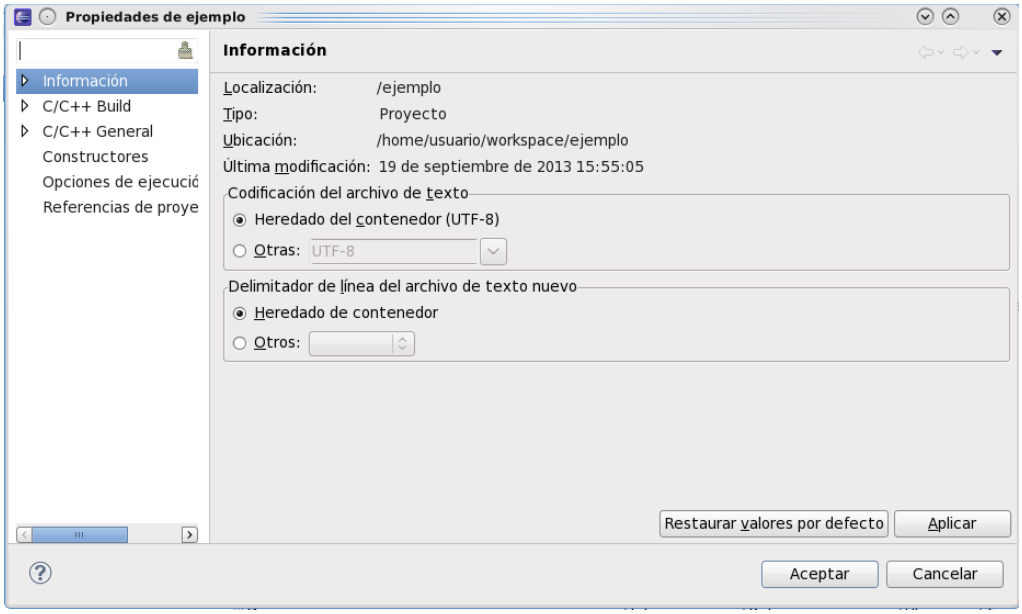

Figura 2.25: Creación de un proyecto para NDS usando Eclipse en Linux (parte 5).

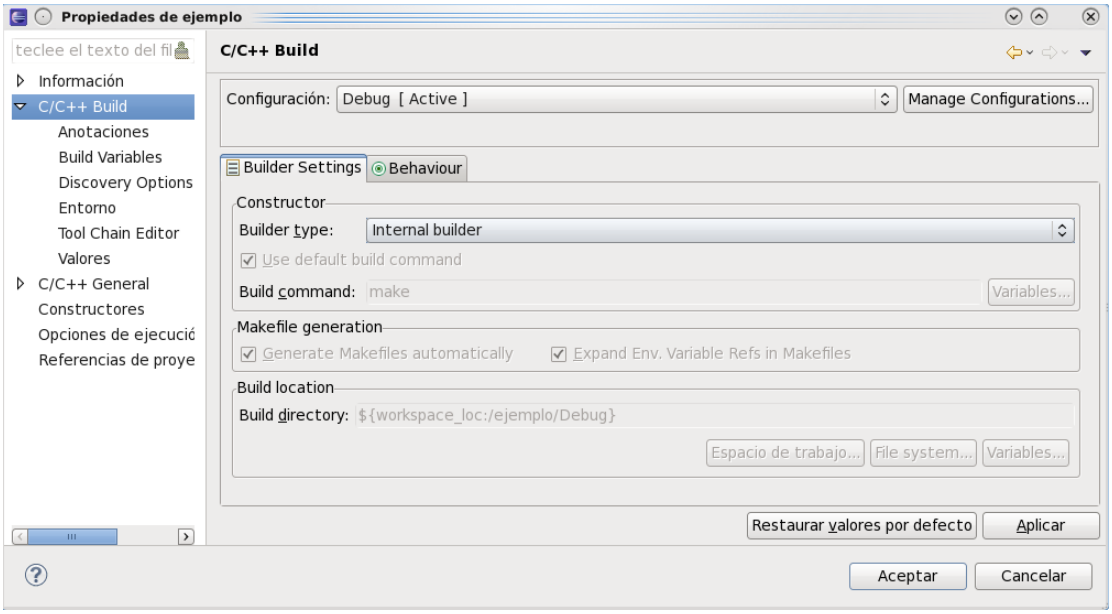

Figura 2.26: Creación de un proyecto para NDS usando Eclipse en Linux (parte 6).

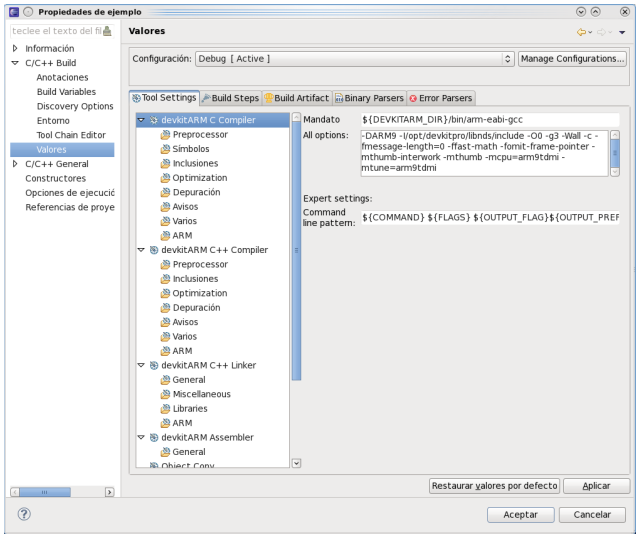

Figura 2.27: Creación de un proyecto para NDS usando Eclipse en Linux (parte 7).

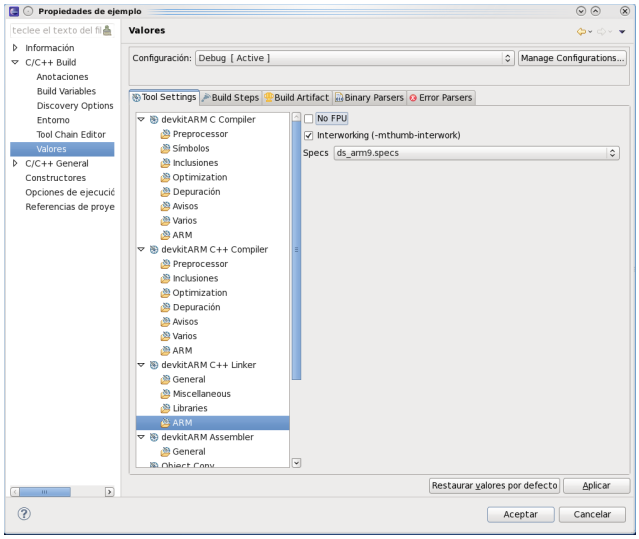

Figura 2.28: Creación de un proyecto para NDS usando Eclipse en Linux (parte 8).

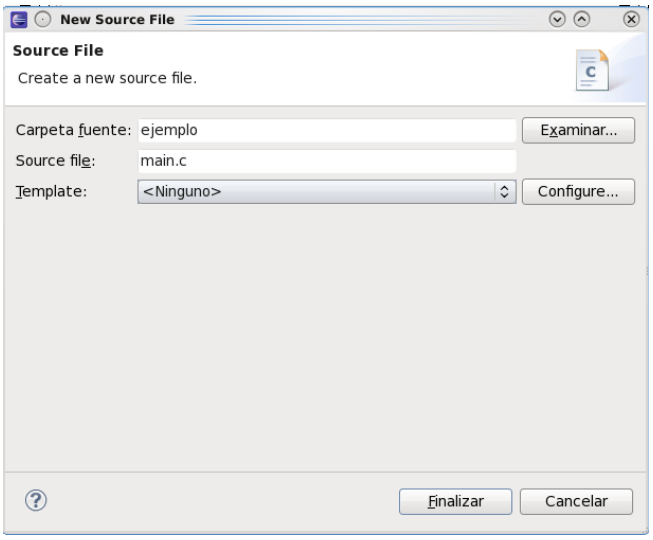

Figura 2.29: Creación de un proyecto para NDS usando Eclipse en Linux (parte 9).

3. Fundamentos para programar la NDS

En este capítulo, se estudiarán los elementos básicos para poder realizar aplicaciones en la consola NDS. En concreto, se estudiarán cuestiones relacionadas con la visualización de texto, la entrada de usuario (botones y pantalla táctil) y el temporizador.

Para obtener información sobre las funciones existentes, se recomienda ir a la página web: `http://libnds.devkitpro.org/`

La lista de ejercicios a realizar y el tiempo estimado (en minutos) para su realización se muestran en la Tabla 3.1.

3.1 Introducción a la programación en NDS

La estructura básica de las aplicaciones realizadas para NDS es la siguiente:

```
1 #include [...]
2 int main(void)
3 {
4   inicializar libNDS;
5   while(1){
6     // Bucle principal
7   }
8   return 0;
9 }
```

El bucle infinito sirve para simular el comportamiento de un videojuego al entrar en su bucle principal. El formato habitual de estos tipos de bucles es el siguiente:

```
Comienza el bucle:
    Comprobar la entrada de usuario
    Actualizar la lógica interna
    Comprobar el criterio de finalización del bucle
    Redibujar
Fin del bucle
```

Tabla 3.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo	Ejercicio	Tiempo
3.1	30’	3.10	10’
3.2	10’	3.11	20’
3.3	5’	3.12	20’
3.4	5’	3.13	10’
3.5	10’	3.14	30’
3.6	10’	3.15	10’
3.7	20’	3.16	10’
3.8	15’	3.17	10’
3.9	15’		

■ **Ejemplo 3.1** El siguiente código muestra un mensaje de texto por la pantalla:

```
#include<nds.h>
#include<stdio.h>
int main(void) {
  consoleDemoInit();
  iprintf("Me gusta programar videojuegos");  // Imprimir el mensaje
  while(1)
  {} // Bucle que no hace nada.
}
```

■

Ejercicio 3.1 Crea un nuevo proyecto usando el código anterior para comprobar que tienes bien instalado todo lo necesario para compilar y ejecutar juegos en la NDS. El capítulo 2 está dedicado a explicar todos los pasos que hay que seguir. ■

3.2 Salida de texto

La Nintendo DS tiene dos pantallas gráficas de tipo LCD (*Liquid Crystal Display*). Las dos tienen el mismo tamaño, 256x192 píxeles, y funcionan gracias a dos motores gráficos: principal o *main* y secundario o *sub*. Además, la pantalla inferior emplea tecnología táctil.

3.2.1 Visualización de texto en la pantalla

La pantalla tiene 32 columnas y 24 filas para visualizar texto, tal y como se puede observar en la Figura 3.1. La instrucción *iprintf* se emplea para visualizar texto por la pantalla. Para elegir la posición del texto en la pantalla se usa la secuencia de escape:

```
\x1b[<fila>;<columna>H
```

empleando como valor para la fila un entero entre 0 y 23, y el valor de la columna, un entero entre 0 y 31.

■ **Ejemplo 3.2** El siguiente código muestra un mensaje de texto en la fila 2, columna 5:

```
#include <nds.h>
#include <stdio.h>
int main(void)
```

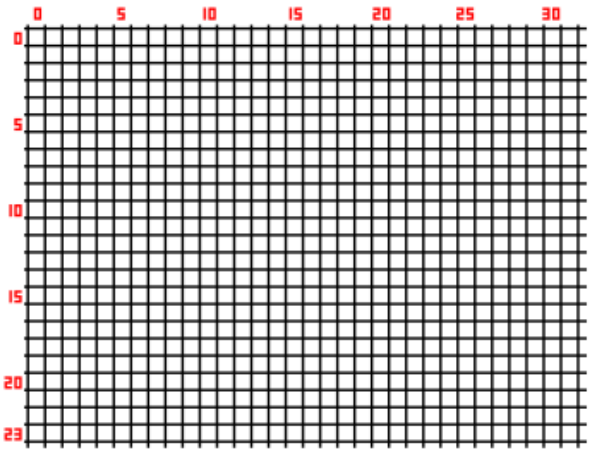

Figura 3.1: Pantalla de la NDS

```
4  {
5    consoleDemoInit();
6    int fila    = 2;
7    int columna = 5;
8    while(1) {
9      iprintf("\x1b[%d;%dHMensaje de texto", fila, columna);
10     swiWaitForVBlank();
11   }
12   return 0;
13 }
```

■

Ejercicio 3.2 Realiza un programa que muestre un mensaje de texto aproximadamente en el centro de la pantalla. ■

Ejercicio 3.3 Realiza un programa que muestre tres mensajes de texto en varias posiciones diferentes de la pantalla. ■

3.2.2 Control de las pantallas a utilizar

■ **Ejemplo 3.3** El siguiente programa (*superior.c*) crea una consola para escribir en la pantalla superior:

```
1  #include <nds.h>
2  #include <stdio.h>
3  int main(void)
4  {
5    PrintConsole pantalla;
6
7    videoSetMode(MODE_0_2D);
8
9    consoleInit(
10       &pantalla,          // Consola a inicializar
11       3,                  // Capa del fondo donde se imprimirá
12       BgType_Text4bpp,    // Tipo de fondo
13       BgSize_T_256x256,   // Tamaño del fondo
14       31,                 // Base del mapa
15       0,                  // Base del tile gráfico
```

```
        true,               // Sistema grafico a usar (main system)
        true);              // No cargar gráficos para la fuente

  while(1) {
  iprintf("\x1b[12;10HMensaje de texto");
  swiWaitForVBlank();     // Esperar al refresco de pantalla
  }
  return 0;
}
```

■

En este código cabe destacar lo siguiente:

- *PrintConsole* es el tipo de datos que define las consolas a utilizar a la hora de imprimir contenidos en pantalla. Se declara la variable *pantalla* de este tipo.
- La función *VideoSetMode* se encarga de inicializar el sistema gráfico principal (*main*), que es el que se usa en el ejemplo. Si se quiere imprimir en la pantalla inferior, se debería inicializar con *VideoSetModeSub*. Los modos de vídeo soportados (en este caso *MODE_0_2D*) dependen del sistema y el fondo que se estén utilizando, y se verán con más detalle en próximos capítulos.
- Para crear una consola con los parámetros deseados se usará la función *consoleInit*. Esta función tiene ocho parámetros de entrada, de los cuales el único que por ahora interesa es el penúltimo que indica el sistema gráfico que se va a usar: con el valor *true*, se utilizará el sistema principal (la pantalla superior), mientras que con el valor *false*, se imprimirá en la pantalla inferior.
- La función *swiWaitForVBlank* espera al refresco de la pantalla.

Ejercicio 3.4 Crea un nuevo proyecto, usando el programa *superior.c* y comprueba que el resultado obtenido es el esperado. ■

Para usar más de una consola, se deben seguir los pasos del ejemplo anterior, pero además se debe emplear la función *consoleSelect* para indicar qué consola se va a usar.

■ **Ejemplo 3.4** El siguiente programa (*dos_pantallas.c*) escribe un mensaje en cada una de las pantallas:

```
#include <nds.h>
#include <stdio.h>
int main(void)
{
  PrintConsole pantalla_sup, pantalla_inf;
  videoSetMode   (MODE_0_2D);
  videoSetModeSub(MODE_0_2D);

  consoleInit(&pantalla_sup,
              3,
              BgType_Text4bpp,
              BgSize_T_256x256,
              31,
              0,
              true,
              true);
  consoleInit(&pantalla_inf,
              3,
              BgType_Text4bpp,
              BgSize_T_256x256,
              31,
```

Figura 3.2: Resultado del programa *dos_pantallas.c*.

```
22                  0,
23                  false,
24                  true);
25
26    while(1) {
27      consoleSelect(&pantalla_sup);
28      iprintf("\x1b[12;3HEsta es la pantalla superior.");
29      consoleSelect(&pantalla_inf);
30      iprintf("\x1b[12;3HEsta es la pantalla inferior.");
31
32      swiWaitForVBlank();
33    }
34    return 0;
35  }
```

■

Ejercicio 3.5 Comprueba mediante la creación de un nuevo proyecto, usando el programa *dos_pantallas.c* que lo indicado ocurre tal y como se comenta. La Figura 3.2 muestra la salida esperada. ■

Ejercicio 3.6 Realiza un programa que muestre tres mensajes en la pantalla superior y otros tres en la pantalla inferior. ■

3.3 Teclado

La Nintendo DS tiene la posibilidad de simular el funcionamiento de un teclado empleando funciones de la biblioteca *libnds*.

■ **Ejemplo 3.5** El siguiente programa (*teclado.c*) muestra como funciona el acceso al teclado:

Figura 3.3: Salida del programa *teclado.c* en el emulador.

```c
#include <nds.h>
#include <stdio.h>
int main(void) {
  int key;  // Variable que almacena el código ascii de la tecla

  consoleDemoInit();
  keyboardDemoInit();  // Inicializa un teclado
  keyboardShow();      // Visualiza el teclado

  while(1) {
    key = keyboardUpdate();  // Procesa la tecla pulsada
                             // Retorna el código ascii
                             // -1 si no se ha pulsado tecla

    // Visualiza el carácter asociado al ascii de la tecla
    if (key > 0) iprintf("Tecla pulsada %c \n", key);

    swiWaitForVBlank();
  }
  return 0;
}
```

■

Este programa visualiza la tecla pulsada, para ello se ha introducido el formato *%c* en la instrucción *iprintf* para visualizar un carácter. La salida del programa *teclado.c* en el emulador se muestra en la Figura 3.3.

En la página web `http://libnds.devkitpro.org/keyboard_8h.html` se encuentra más información sobre el funcionamiento del teclado de la NDS.

Ejercicio 3.7 A partir del código del programa *teclado.c*, crea un programa que muestre tu nombre cuando se pulse y se mantenga pulsada una tecla cualquiera y que deje de mostrarlo cuando se deje de pulsar la tecla. ■

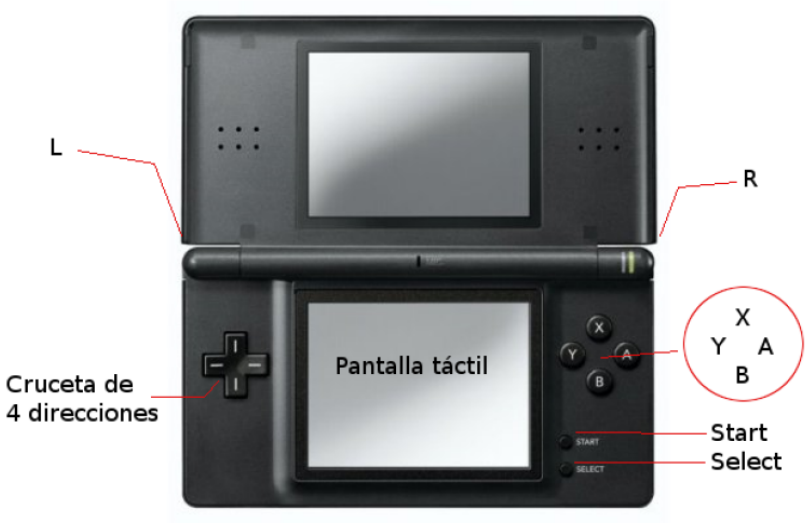

Figura 3.4: Botones de la NDS.

Ejercicio 3.8 A partir del código del programa *teclado.c*, crea un programa que de inicio muestre tu nombre, pero cuando se pulse una tecla cualquiera deje de mostrarlo. Si de nuevo se pulsa una tecla se volverá a mostrar, y así sucesivamente. ■

Ejercicio 3.9 A partir del código del programa *teclado.c*, crea un programa que muestre tu nombre cuando se pulse la tecla *s*. El nombre continuará visible hasta que se pulse la tecla *n*. Si se vuelve a pulsar la tecla *s*, el nombre volverá a ser visible y así sucesivamente. ■

3.4 Botones de la consola

La consola NDS presenta los siguientes botones como entrada de usuario (ver Figura 3.4):

- Una cruceta direccional a la izquierda de la pantalla inferior (cruceta de 4 direcciones).
- Cuatro botones (*A*, *B*, *X*, *Y*) a la derecha de la pantalla inferior.
- Dos botones laterales (*L* y *R*) situados detrás.
- Botón de *Select* y botón de *Start*, a la derecha de la pantalla inferior.
- La pantalla táctil inferior.

Además, el cierre de la consola tiene también un sensor para saber si está abierta o cerrada, que a efectos de programación, funciona como otro botón más.

La Figura 3.5 muestra la relación que existe entre los botones de la NDS y el teclado en un determinado emulador.

Los pasos a seguir para determinar las operaciones realizadas con los botones son los siguientes:

1. Primero, en cada paso por el bucle principal, se determinará si se ha pulsado algunos de los botones mediante la función *scanKeys*.
2. Después se puede obtener información sobre los botones con las siguientes funciones:
   - *keysCurrent*: obtiene el estado actual de los botones.
   - *keysDown*: obtiene los botones pulsados en ese instante.
   - *keysHeld*: obtiene los botones mantenidos.
   - *keysUp*: obtiene los botones liberados.

El resultado de estas funciones se compara mediante una *multiplicación bit a bit* con el valor del botón del que se quiere saber su estado. Como valores del botón cuyo estado se desea saber se puede emplear:

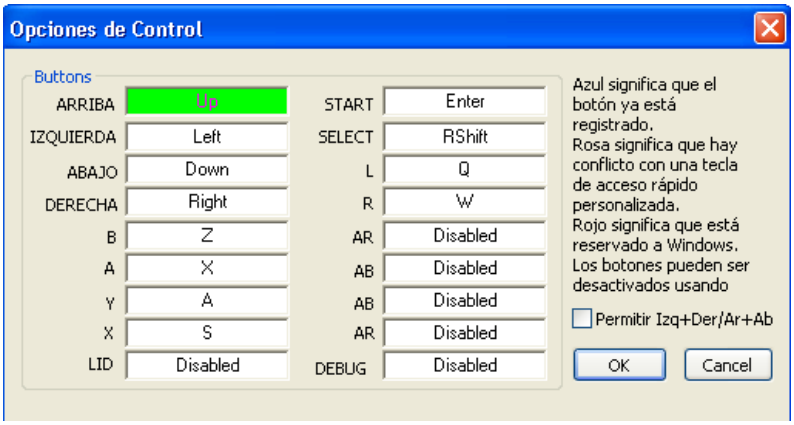

Figura 3.5: Relación que existe entre los botones de la NDS y el teclado.

- Los cuatro botones de la derecha: KEY_A, KEY_B, KEY_X, KEY_Y.
- Los botones laterales *L* y *R*: KEY_L, KEY_R.
- Botones *Select* y *Start*: KEY_SELECT, KEY_START.
- La cruceta de 4 direcciones: KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT.
- La pantalla táctil (solo como botón): KEY_TOUCH.
- La bisagra del cierre de la consola: KEY_LID.

La siguiente página web contiene información sobre el uso de la botonera `http://libnds.devkitpro.org/arm9_2input_8h.html`

■ **Ejemplo 3.6** El siguiente programa (*botones.c*) comprueba el estado instantáneo de cada botón, mostrando en pantalla si se ha pulsado:

```
#include <nds.h>
#include <stdio.h>
int main()
{
  u32 keys;
  consoleDemoInit();

  while(1){
    scanKeys();
    keys = keysCurrent(); // Se lee el estado actual de todos los botones

    // Se comprueba si se ha pulsado arriba en la cruceta
    if (keys & KEY_UP) iprintf("\x1b[10;5H Pulsado UP");
    else               iprintf("\x1b[10;5H           ");

   // Se comprueba si se ha pulsado en el boton A
   if (keys & KEY_A)   iprintf("\x1b[12;5H Pulsado A");
   else                iprintf("\x1b[12;5H          ");

   // Se comprueba si se ha pulsado el botón Start
   if (keys & KEY_START) iprintf("\x1b[15;5H Pulsado START");
   else                  iprintf("\x1b[15;5H               ");

   // Se comprueba si se ha pulsado la pantalla táctil
   if (keys & KEY_TOUCH) iprintf("\x1b[13;5H Pantalla tactil");
   else                  iprintf("\x1b[13;5H                 ");

   swiWaitForVBlank();
  }
```

```
30    return 0;
31  }
```

■

Se puede observar que se llama a la función *scanKeys*, y que posteriormente se llama a la función *keysCurrent* para conocer el estado actual de los botones. Dicha información se guarda en la variable *keys*, declarada de tipo *u32* (entero sin signo de 32 bits). Con el valor de esta variable se realiza la multiplicación bit a bit con el valor de un botón específico mediante:

```
1  if (keys & KEY_UP)
```

Si el botón está pulsado, se muestra en pantalla un mensaje indicando el botón en concreto, incluyendo la pantalla táctil. Si se quieren realizar acciones en momentos precisos (justo cuando se pulsa un botón, o cuando se libera), se podrán utilizar el resto de funciones.

Ejercicio 3.10 Comprueba mediante la creación de un nuevo proyecto, usando el código *botones.c*, que lo indicado ocurre tal y como se comenta. ■

Ejercicio 3.11 Realiza un programa que permita desplazar un mensaje de texto por la pantalla inferior empleando los botones de dirección (derecha, izquierda, arriba y abajo). Se debe controlar que el mensaje completo siempre se encuentre dentro de la pantalla, y que se desplace solo una posición en cada pulsación del correspondiente botón. ■

Ejercicio 3.12 Modifica el programa anterior para que al pulsar el botón *X* se cambie el mensaje a mostrar por otro mensaje. Al pulsar el botón *B* se deberá volver a mostrar el mensaje original. ■

3.5 Pantalla táctil

Como ya se ha comentado, la pantalla inferior emplea tecnología táctil. En la Nintendo DS se usa el sistema de coordenadas cartesiano mostrado en la Figura 3.6. En el emulador se emplea el ratón como puntero de la pantalla táctil.

La siguiente página web contiene información sobre el uso de la pantalla táctil `http://libnds.devkitpro.org/arm9_2input_8h.html`

■ **Ejemplo 3.7** El siguiente programa (*tactil.c*) cuenta el número de veces que se pulsa en la pantalla táctil.

```
1  #include <nds.h>
2  #include <stdio.h>
3
4  int main(void) {
5    touchPosition posicionXY;
6    int           contador;
7
8    consoleDemoInit();
9    contador = 0;
10
11   while(1)
12   {
13     scanKeys();
14     touchRead(&posicionXY); // Se lee la posición actual.
```

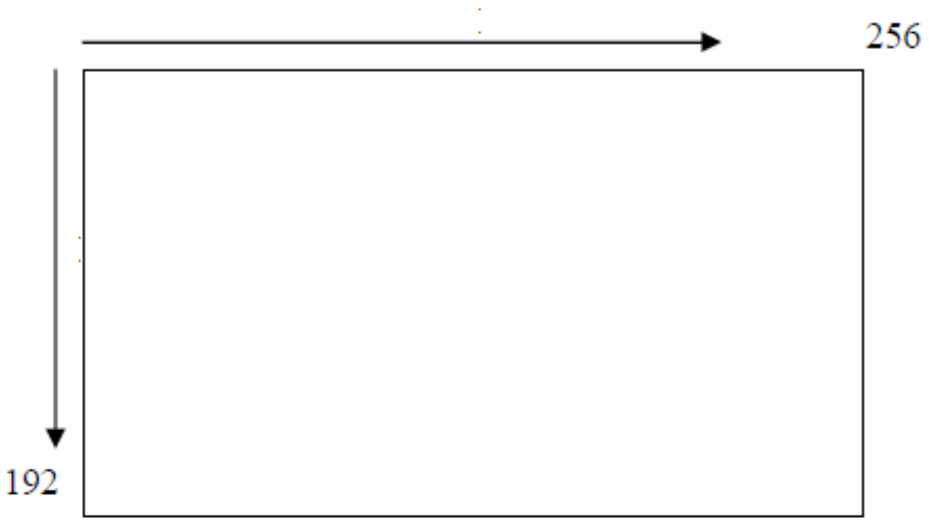

Figura 3.6: Sistema de coordenadas cartesiano usado para detectar donde ha pulsado el usuario en la pantalla táctil.

```
15      iprintf("\x1b[1;0HPosicion x=%04i y=%04i ",
16              posicionXY.px,
17              posicionXY.py);
18      iprintf("\x1b[2;0HContador=%04i",
19              contador);
20
21      if (keysDown() & KEY_TOUCH) contador++;
22
23      swiWaitForVBlank();
24    }
25    return 0;
26  }
```

■

De este código se puede destacar lo siguiente:

- Se crea una variable del tipo *touchPosition* (en este caso *posicionXY*) para reconocer la posición del puntero en la pantalla táctil. Mediante *posicionXY.px* se obtiene la posición respecto al eje *x*, mientras que con *posicionXY.py* se obtiene la posición respecto al eje *y*.
- La función *touchRead* lee el valor de la posición del puntero en la pantalla táctil.
- El formato *%04i* en la instrucción *iprintf* visualiza un número entero de 4 cifras (en decimal) poniendo ceros en la izquierda.

Ejercicio 3.13 Comprueba mediante la creación de un nuevo proyecto, usando el código *tactil.c*, que lo indicado ocurre tal y como se comenta. ■

Ejercicio 3.14 Realiza un programa que divida la pantalla inferior en 4 zonas. En cada una de ellas deberá aparecer un contador que contará el número de veces que se pulsa en cada zona. La Figura 3.7 muestra un ejemplo de la salida del programa requerido. ■

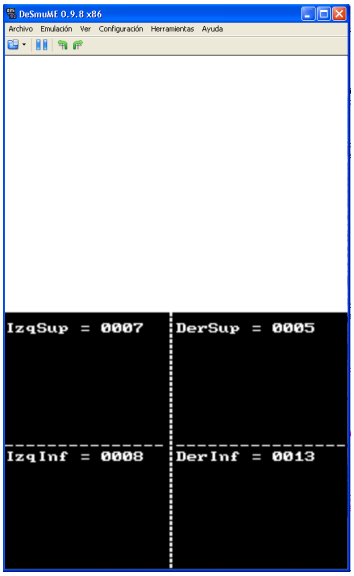

Figura 3.7: Salida ejemplo del programa requerido en la sección 3.5

3.6 Temporizador

La Nintendo DS cuenta con temporizadores (*timers*) de tiempo real, que pueden ser empleados por una aplicación o juego para definir diferentes respuestas dependiendo de la hora del día. Las características que tienen son las siguientes:

- Hay 8 temporizadores de 16 bits, 4 en el ARM9 y 4 en el ARM7.
- Funcionan como contadores de eventos.
- La frecuencia base con la que trabajan es de 33MHz, sobre la que se puede aplicar los siguientes divisores de frecuencia: 1, 64, 256 y 1024.
- Soportan la configuración en cascada, es decir, que cuando un temporizador se desborda el siguiente temporizador incrementa su cuenta.
- Pueden generar interrupciones.

La página web `http://libnds.devkitpro.org/timers_8h.html` muestra información sobre los temporizadores.

■ **Ejemplo 3.8** El siguiente programa (*tiempo.c*) muestra el tiempo que va transcurriendo en segundos:

```
1  #include <nds.h>
2  #include <stdio.h>
3  #include <time.h>
4  //se define la velocidad del reloj con ClockDivider_1024
5  #define TIMER_SPEED (BUS_CLOCK/1024)
6  int main()
7  {
8    consoleDemoInit();
9    uint ticks = 0;
10   timerStart(0, ClockDivider_1024, 0, NULL); // Iniciar el reloj
11   while (1)
12   {
13    ticks += timerElapsed(0);
14    iprintf("\x1b[1;0Hticks:  %u",ticks);
15    iprintf("\x1b[2;0Hseg.: %u:%u",
16            ticks/TIMER_SPEED,
17            ((ticks%TIMER_SPEED)*1000)/TIMER_SPEED);
```

```
18      swiWaitForVBlank();
19    }
20    return 0;
21  }
```

■

Se declara la variable *ticks* como *uint* (entero sin signo). En la llamada *timerStart* cabe destacar lo siguiente:

- El primer parámetro hace referencia al *temporizador 0*.
- El parámetro *ClockDivider_1024* hace referencia a aplicar una división de frecuencia de 1024.

La función *timerElapsed* proporciona el número de ticks que se han producido desde la última llamada a esa misma función, para el temporizador especificado (en este caso el 0). El formato *%u* en la instrucción *iprintf* visualiza un número entero sin signo.

Ejercicio 3.15 Comprueba mediante la creación de un nuevo proyecto, usando el código *tiempo.c*, que lo indicado ocurre tal y como se comenta. ■

Ejercicio 3.16 Modifica el programa *tiempo.c* para que finalice la cuenta si han transcurrido 15 segundos o si se ha pulsado el boton *Y*. ■

En ocasiones se desea que ocurra un evento cada cierto tiempo.

■ **Ejemplo 3.9** El siguiente programa (*eventos.c*) mueve la letra *X* hacia la derecha cada 2 segundos:

```
1   #include <nds.h>
2   #include <stdio.h>
3   #include <time.h>
4
5   #define TIMER_SPEED (BUS_CLOCK/1024)
6
7   int main()
8   {
9     consoleDemoInit();
10    uint ticks          = 0;
11    int  posicion       = 0;
12    int  proximo_cambio = 2;
13    int  segundos;
14
15    timerStart(0, ClockDivider_1024, 0, NULL);
16
17    while (1)
18    {
19      ticks    += timerElapsed(0);
20      segundos = (int) (ticks/TIMER_SPEED);
21
22      if (segundos >= proximo_cambio)
23      {
24        posicion       = posicion       + 1;
25        proximo_cambio = proximo_cambio + 2;
26      }
27      iprintf("\x1b[1;%dH ", posicion-1); // Borra anterior
28      iprintf("\x1b[1;%dHX", posicion);   // Dibuja X
29      swiWaitForVBlank();
30    }
31    return 0;
32  }
```

■

Ejercicio 3.17 Modifica el programa *eventos.c* para que la letra se mueva cada 3 segundos. ■

Es importante destacar que para programar este tipo de funcionalidad, es más conveniente el uso de interrupciones, tal como se verá posteriormente.

4. Programación de un juego sin gráficos

En este capítulo, se darán las instrucciones para la realización paso a paso un videojuego sin usar el sistema gráfico de la NDS.

La lista de ejercicios y el tiempo estimado (en minutos) para su realización se muestran en la Tabla 4.1.

4.1 Descripción del juego

El juego que se va a realizar es una versión simplificada del típico juego de carreras de caballos que se solían encontrar en las ferias itinerantes. La Figura 4.1[1] muestra un ejemplo de este tipo de juego. En este juego, el usuario debe introducir bolas en unos agujeros, y según el agujero donde ha caído la bola, el caballo irá más o menos deprisa. Gana el caballo que antes llegue a la meta.

En la versión que se va a desarrollar se usarán los conceptos aprendidos en el capítulo 3. En la pantalla superior se mostrará la carrera. La pantalla se dividirá en 4 carriles horizontales, uno por caballo. Para representar cada caballo se usará una letra. Al inicio los 4 caballos se situarán en la primera columna de la pantalla (columna 0). Ganará el primer caballo que alcance la meta situada en la última columna de la pantalla (columna 31).

En la pantalla inferior habrá un mensaje indicando el dinero que tiene el jugador. Al inicio el jugador tendrá 1000 euros.

También existirán cuatro botones, uno por caballo. El jugador deberá pulsar en uno de los cuatro botones para apostar por un caballo determinado. La apuesta es de 100 euros. Si el caballo por el que ha apostado el jugador gana, se obtendrán 200 euros que serán añadidos al dinero total. Si gana otro caballo, el jugador perderá ese dinero.

El juego continuará hasta que el jugador se quede sin dinero o alcance la cifra de 10 000 euros.

[1] La imagen se ha obtenido en la siguiente web: `http://www.parquedebolas.com/images/productos/gran/523000002.jpg`

Tabla 4.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo
4.1	10’
4.2	20’
4.3	30’
4.4	20’
4.5	10’
4.6	10’
4.7	10’
4.8	10’

Figura 4.1: Juego de las carreras de caballos.

4.2 Desarrollo del juego

■ **Ejemplo 4.1** El siguiente listado (*caballos_inicial.c*) muestra el esqueleto del programa que puedes usar para empezar:

```
#include <nds.h>
#include <stdio.h>

void MostrarCaballos(int posicion_caballos[]);
void MostrarBotones();
void MostrarDinero(int dinero);

int main(void)
{
  PrintConsole pantalla_sup, pantalla_inf;
  videoSetMode   (MODE_0_2D);
  videoSetModeSub(MODE_0_2D);
  consoleInit (&pantalla_sup, 3, BgType_Text4bpp,
               BgSize_T_256x256, 31, 0, true, true);
  consoleInit (&pantalla_inf, 3, BgType_Text4bpp,
               BgSize_T_256x256, 31, 0, false, true);

  int posicion_caballos[4];
  for (int i=0; i<4; i++)
```

```
20      posicion_caballos[i] = 0;
21
22    int dinero = 1000;
23
24    while(1)
25    {
26      consoleSelect  (&pantalla_sup);
27      MostrarCaballos(posicion_caballos);
28      consoleSelect  (&pantalla_inf);
29      MostrarDinero  (dinero);
30      MostrarBotones ();
31    }
32  }
33
34  void MostrarCaballos(int posicion_caballos[])
35  {
36    iprintf("\x1b[2;%dHA", posicion_caballos[0]);
37    iprintf("\x1b[8;%dHB", posicion_caballos[1]);
38    iprintf("\x1b[14;%dHC",posicion_caballos[2]);
39    iprintf("\x1b[20;%dHD",posicion_caballos[3]);
40  }
41
42  void MostrarBotones()
43  {
44    iprintf("\x1b[10;1HApuesta por un caballo: ");
45    iprintf("\x1b[12;4H----- ----- ----- -----");
46    iprintf("\x1b[13;4H- A - - B - - C - - D -");
47    iprintf("\x1b[14;4H----- ----- ----- -----");
48  }
49
50  void MostrarDinero(int dinero)
51  {
52    iprintf("\x1b[2;1HTienes %d euros", dinero);
53  }
```

■

Como se puede comprobar por el código anterior, los cuatro carriles para mostrar la evolución de los caballos estarán en las filas, 2, 8, 14 y 20 de la pantalla superior. En la fila 2 de la pantalla inferior se mostrará un mensaje indicando el dinero que tiene el jugador. En las filas 12, 13 y 14 se mostrarán los botones para apostar por los caballos.

Ejercicio 4.1 Crea un nuevo proyecto usando el código *caballos_inicial.c* y comprueba que funciona correctamente. ■

Para crear el programa completo debes resolver los siguientes ejercicios en el orden establecido. En primer lugar debes crear el código necesario para que los caballos se muevan hacia la meta. Al inicio todos los caballos tendrán una velocidad de una posición por unidad de tiempo. Antes de mostrar la posición de los caballos (función *MostrarCaballos*), debes llamar a una función de nombre *ActualizarPosicionCaballos* que dada la posición actual y la velocidad actual de cada caballo, actualice su posición. Debes tener en cuenta que la máxima posición que puede tener un caballo es la columna 31 y la mínima la 0. Así mismo, la mínima velocidad de un caballo es 0, y la máxima un valor que establezcas (por ejemplo 3).

Cada vez que actualices la posición de los caballos debes borrar la pantalla superior, para ello debes crear una funcion *BorrarPantalla* que será llamada antes de mostrar los caballos en la nueva posición.

Por último debes crear una función *ComprobarGanador* que devolverá el número de caballo

que ha llegado a la meta o -1 si no ha llegado todavía ninguno. En caso de empate, el ganador será siempre el que tenga el número menor. Por ejemplo, si llegan a la vez el 0 (letra A) y el 1 (letra B), el ganador será el 0.

■ **Ejemplo 4.2** El programa principal, con los cambios comentados, es el siguiente:

```
...
  int velocidad_caballos[4];
  for (int i=0; i<4; i++)
    velocidad_caballos[i] = 1;
...
  int hay_ganador = 0;
  while(1)
  {
    if (hay_ganador == 0)
    {
      int ganador = ComprobarGanador(posicion_caballos);
      if (ganador >= 0)
      {
        hay_ganador = 1;
        consoleSelect(&pantalla_inf);
        iprintf("\x1b[5;1HEl caballo ganador es el: %d",ganador);
      }
      else
      {
        ActualizarPosicionCaballos(posicion_caballos, velocidad_caballos);
        consoleSelect(&pantalla_sup);
        BorrarPantalla();
        MostrarCaballos(posicion_caballos);
      }
    }
    consoleSelect(&pantalla_inf);
    MostrarDinero(dinero);
    MostrarBotones();
}
...
```

■

Como puedes comprobar en el código anterior, la fila 5 de la pantalla inferior se usará para mostrar el caballo ganador.

> **Ejercicio 4.2** Implementa las funciones *BorrarPantalla*, *ActualizarPosicionCaballos* y *ComprobarGanador*. ■

Si el código funciona correctamente, habrás comprobado que los 4 caballos llegan a la vez a la meta, puesto que su velocidad es siempre la misma. Por lo tanto, el ganador es siempre el caballo 0. Además, puesto que en cada frame se actualiza la posición, el movimiento de los caballos no se aprecia.

Para solucionar el primer problema, debes modificar la función *ActualizarPosicionCaballos* para que dependiendo de un número aleatorio, la velocidad de los caballos pueda variar. Para modificar la velocidad de los caballos, se puede usar un generador de números aleatorios entre dos números reales usando la siguiente función [2]:

■ **Ejemplo 4.3**

```
double closed_interval_rand(double x0, double x1)
```

[2] Código obtenido de `https://bytes.com/topic/c/answers/223101-rand-between-0-1-a`

```
{
  return x0 + (x1 - x0) * rand() / ((double) RAND_MAX);
}
```

■

Por ejemplo, puedes obtener un número entre 0,0 y 1,0 llamando a la función anterior tal como se muestra a continuación:

■ Ejemplo 4.4

```
double numero = closed_interval_rand(0.0, 1.0)
```

■

Según el número obtenido puedes decidir aumentar o disminuir en la velocidad. Un posible algoritmo es el siguiente:

1. Si el número obtenido es menor a α, entonces aumentar la velocidad en una unidad.
2. Si es mayor a β, la velocidad disminuirá en una unidad.
3. En otro caso, la velocidad se mantiene sin cambios.

Por ejemplo, si $\alpha = 0{,}2$ y $\beta = 0{,}9$, habrá un 20 % de posibilidades de aumentar la velocidad, un 10 % de disminuir y un 70 % de que permanezca sin cambios.

Puedes cambiar los valores α y β para que no todos los caballos corran a la misma velocidad. También puedes establecer dichos valores al azar (usando el generador de números aleatorios). Por ejemplo, α puede variar entre 0,1 y 0,3, y *beta* entre 0,7 y 0,9.

Para evitar que el generador de números aleatorios genere siempre los mismos números, es conveniente inicializar la semilla del generador añadiendo el siguiente código al inicio del programa:

■ Ejemplo 4.5

```
srand (time(NULL));
```

■

Ejercicio 4.3 Modifica la función *ActualizarPosicionCaballos* para permitir cambios en la velocidad de los caballos. ■

Para evitar que se actualice la posición de los caballos en cada frame hemos de usar el temporizador. Para ello debes hacer que cada determinado número de segundos (por ejemplo 1), se llame a la parte del código que actualiza la posición. En el capítulo anterior hay un ejemplo que puedes usar como base.

Ejercicio 4.4 Modifica el programa para que el cambio de posición se realice cada determinado número de segundos. ■

Una vez llegado a este punto, tendremos un programa que mueve los caballos por la pantalla superior de forma que en cada ejecución el ganador puede ser cualquiera de los cuatro caballos.

Para finalizar el juego, tenemos que realizar las siguientes acciones:

- Comprobar si el usuario ha pulsado sobre un botón para apostar.
- Modificar el programa principal para que la carrera empiece cuando el jugador pulse sobre algún botón.
- Según el resultado de la carrera, actualizar el dinero del jugador.
- Modificar el programa para permitir varias carreras.

Ejercicio 4.5 Realiza una función *ObtenerApuesta* que devuelva el índice del caballo por el que se ha apostado o -1 si el usuario no ha pulsado ningún botón. ■

Ejercicio 4.6 Modifica el programa para que la carrera empiece cuando el jugador pulse en alguno de los botones. ■

Ejercicio 4.7 Modifica el programa para que actualice el dinero acumulado tras la realización de la carrera. ■

Ejercicio 4.8 Modifica el programa para que permita la realización de varias carreras. El juego continuará hasta que el jugador se quede sin dinero o alcance la cifra de 10 000 euros. En ambos casos, se deberá mostrar un mensaje informando de lo que ha acontecido. ■

5. Sistema de memoria gráfica de la NDS

Este capítulo trata sobre la realización de videojuegos con gráficos en la consola NDS. Este documento se ha redactado usando como principal fuente los capítulos 7, 8 y 9 del libro: Francisco Moya Fernández y María José Santofimia Romero. *Laboratorio de Estructura de Computadores empleando videoconsolas Nintendo DS*. UCLM, ISBN: 978-84-9981-039-4.

5.1 Introducción

El hardware de vídeo de la Nintendo DS se compone de dos núcleos gráficos 2D, uno principal o *main* y otro secundario o *sub*, diferenciados únicamente en que el motor principal puede renderizar tanto la memoria de vídeo virtual sin utilizar el motor 2D, como mapas de bits de 256 colores, así como utilizar el motor 3D para el renderizado de alguno de sus fondos.

El concepto de fondo es básico para comprender cómo funcionan los modos de vídeo de la Nintendo DS y se puede entender como un concepto equivalente al de capa o layer utilizado por algunas aplicaciones de diseño gráfico para facilitar la composición de una imagen a partir de la superposición del contenido de las capas. Los núcleos gráficos de la Nintendo DS disponen de cuatro fondos, etiquetados como *BG0*, *BG1*, *BG2* y *BG3*, cuya configuración dependerá del tipo de gráfico a representar.

Un modo gráfico básicamente agrupa un conjunto de configuraciones para cada uno de los fondos. La Figura 5.1 resume los modos disponibles para el núcleo principal y el secundario. Los seis primeros modos, *Mode 0* a *Mode 5*, son comunes para los dos núcleos. Además, el núcleo principal cuenta con el *Mode* 6 y con el modo *frame buffer*.

Existen tres tipos diferentes de configuraciones para los fondos 2D, que son *Text*, *Rotation* y *Extended Rotation* y el modo *framebuffer* que pinta la imagen directamente sin utilizar fondos. A los modos *Text* también se les llama modos teselados, y a los modos *Rotation* también se les conoce como modos *Rotoscale*.

Las imágenes que se desean mostrar en la pantalla deben ser traducidas a contenido binario, que se guardará junto con el programa y que debe ser copiado en un *banco de memoria VRAM (Video RAM)* que es donde está mapeada la pantalla. Mapear la pantalla a un banco de memoria

Graphics Modes

Main 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM display as a bitmap			
Sub 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

Figura 5.1: Modulos gráficos

Tabla 5.1: Tamaño y dirección de memoria de cada banco de memoria.

Banco	Tamaño	Dirección memoria
VRAM_A	128KB	0x6800000
VRAM_B	128KB	0x6820000
VRAM_C	128KB	0x6840000
VRAM_D	128KB	0x6860000
VRAM_E	64KB	0x6880000
VRAM_F	16KB	0x6890000
VRAM_G	16KB	0x6894000
VRAM_H	32KB	0x6898000
VRAM_I	16KB	0x68a0000

significa que a la pantalla se le asigna ese banco de memoria, de forma que lo que se haya escrito en ese banco de memoria será lo que se vea en la pantalla.

La Nintendo DS tiene 9 bancos de memoria de vídeo, que se pueden usar con diferentes propósitos. Cada de uno de estos bancos de memoria está etiquetado desde *VRAM_A* hasta *VRAM_I*. La Tabla 5.1 muestra cada uno de los bancos de memoria VRAM con su tamaño y su dirección de memoria. La cantidad total de memoria de vídeo es de 656KB.

La Figura 5.2 muestra los bancos de memoria que se pueden usar en cada motor gráfico y para que fin. Por ejemplo, el banco *VRAM C* se puede usar tanto por el motor *main* como por el *sub*. Sin embargo, el banco *VRAM A* solo puede ser usado por el motor *main*.

5.2 El registro *REG_POWERCNT*

El registro *REG_POWERCNT* es el encargado del encendido de todo el hardware gráfico. La Figura 5.3 muestra el significado de algunos de sus bits. El contenido de este registro permite

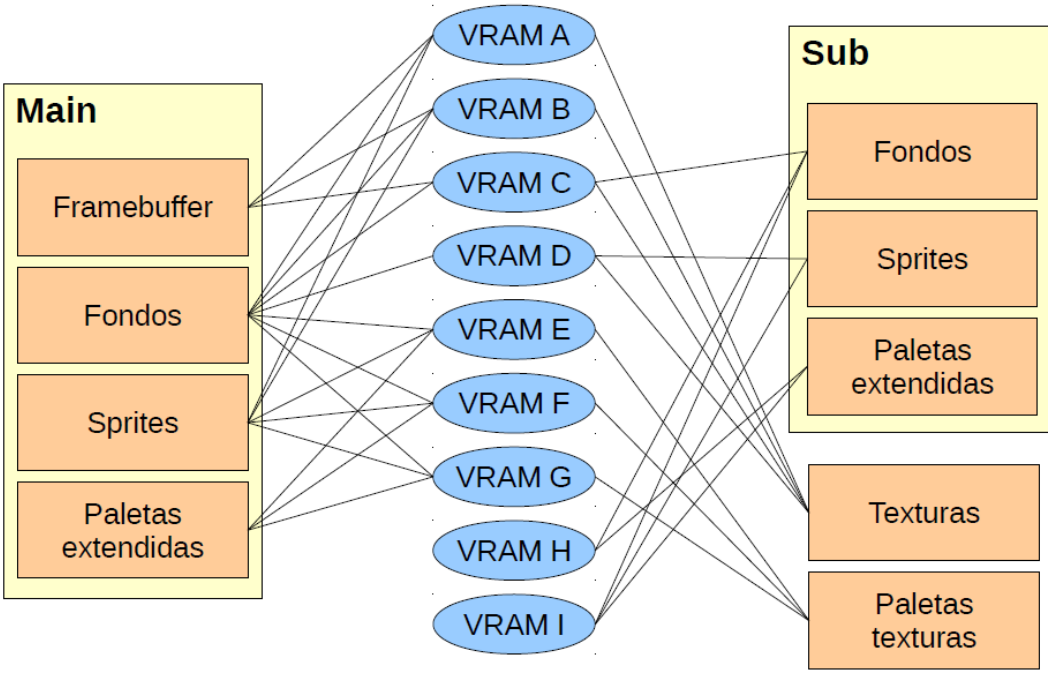

Figura 5.2: Bancos de memoria y que motores, y para que fin, los pueden usar.

activar ambas pantallas y los dos motores gráficos: el principal (*main*) y el secundario (*sub*) para trabajar en 2D. Además, el bit 15 permite intercambiar ambas pantallas y asignar el motor principal a la pantalla superior (top) en vez de a la inferior que es la que tiene por defecto.

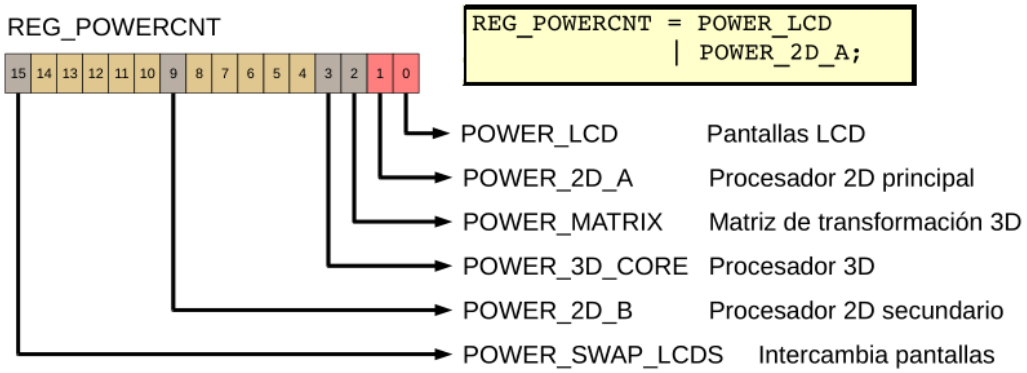

Figura 5.3: Significado de los bits del registro *REG_POWERCNT*

Por ejemplo, para activar ambas pantallas y únicamente el motor principal se puede usar la instrucción:

```
REG_POWERCNT = POWER_LCD | POWER_2D_A;
```

Para activar ambas pantallas y ambos motores se puede usar la instrucción:

```
REG_POWERCNT = POWER_LCD | POWER_2D_A | POWER_2D_B;
```

Se puede encontrar más información en la web: `http://libnds.devkitpro.org/system_8h.html`.

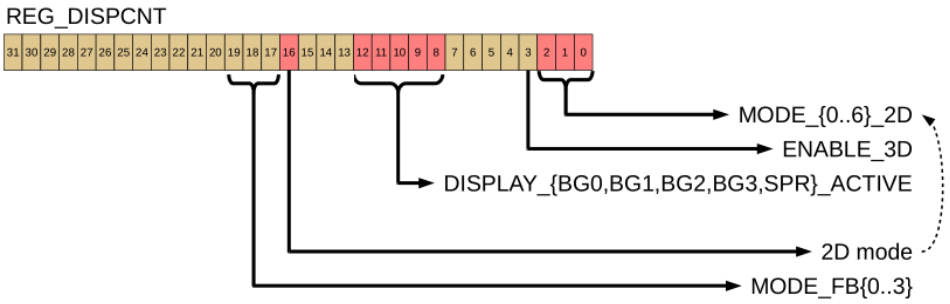

Figura 5.4: Significado de los bits del registro *REG_DISPCNT*

5.3 El registro *REG_DISPCNT*

El registro *REG_DISPCNT* es el que se encarga de controlar los modos y fondos activos. La Figura 5.4 muestra el significado de algunos de sus bits. Cabe destacar la finalidad de los siguientes bits:

- Bits 0-2: especifican el modo gráfico. Del 0 al 6 en binario con tres bits.
- Bits 8-12: especifican el fondo.
- Bits 17-19: especifican la configuración del modo gráfico *framebuffer*.

Por ejemplo, para indicar que se activa el *Modo 0* con el fondo *BG0* se debe usar la siguiente instrucción:

```
REG_DISPCNT = MODE_0_2D | DISPLAY_BG0_ACTIVE;
```

Esta operación también se puede llevar a cabo, en el motor *main*, mediante la instrucción:

```
videoSetMode(MODE_0_2D);
```

Para el motor *sub*, habrá que usar la instrucción:

```
videoSetModeSub(MODE_0_2D);
```

Se puede encontrar más información en la web: `http://libnds.devkitpro.org/video_8h.html`.

5.4 Los registros *VRAM_?_CR*

Cada *banco VRAM* tiene un registro *VRAM_?_CR* (donde ? puede tomar los valores de A a I) para activarlo y seleccionar su función. La Figura 5.5 muestra el significado de cada uno de sus bits. Cabe destacar que los bits etiquetados con *Modo* y *Desplazamiento* son los que permiten seleccionar la función del fondo configurado en el registro *REG_DISPCNT*. Por su parte el *bit 7* es el que permite activar el correspondiente *banco VRAM*.

Como ejemplo, para indicar que se activa *VRAM_A* y que se asigna a fondos del motor principal (*main*) se debe usar la instrucción:

```
VRAM_A_CR = VRAM_A_ENABLE | VRAM_A_MAIN_BG;
```

Esta operación también se podría llevar a cabo mediante:

```
vramSetBankA(VRAM_A_MAIN_BG);
```

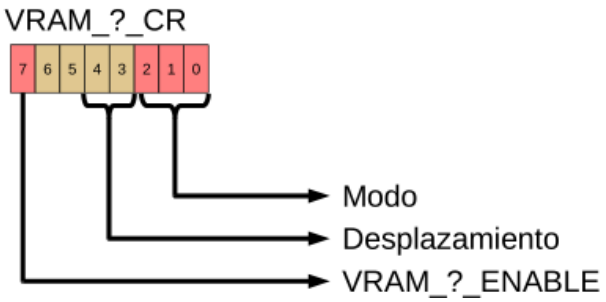

Figura 5.5: Significado de los bits del registro *VRAM_X_CR*

6. El modo framebuffer

Este capítulo explica el funcionamiento del modo framebuffer. La lista de ejercicios y el tiempo estimado (en minutos) para su realización se muestran en la Tabla 6.1.

6.1 El modo Framebuffer

La principal peculiaridad de este modo de representación es que la pantalla se mapea directamente a la memoria, sin utilizar el motor de renderizado, de forma que lo que se escribe en memoria es lo que se muestra. La pantalla se compone de 49152 píxeles, organizados en 192 líneas de 256 píxeles cada una, donde a su vez, el contenido de cada píxel representa un color en formato RGB de 16 bits.

El formato RGB de 16 bits para especificar el color se representa como una mezcla de una cantidad determinada del color rojo (R), verde (G) y azul (B), dedicando 5 bits para determinar esa cantidad: 0 indica ausencia de color y el 31 el máximo color. La librería *libnds* proporciona la macro *RBG15* para facilitar la definición de colores. Basta con indicar la cantidad de cada uno de ellos, en un rango de 0 a 31, para obtener el color en formato RGB de 16 bits. Por ejemplo, para obtener los colores: rojo, verde, azul y negro, habrá que usar las macros: *RGB15(31,0,0)*, *RGB15(0,31,0)*, *RGB15(0,0,31)*, *RGB15(0,0,0)*, respectivamente. El uso de esta macro simplifica bastante la tarea de dibujado en la pantalla, ya que únicamente hay que asignar píxel a píxel el valor correspondiente al color que se desea pintar.

Tal como muestra la Figura **??** únicamente el motor principal puede usar el modo framebuffer. Esto implica que solo se podrá mostrar una imagen en una de las dos pantallas a la vez.

6.1.1 Preparando el modo framebuffer

El modo framebuffer admite cuatro tipos de configuración, que reciben el nombre de FB0, FB1, FB2 o FB3 y que básicamente establecen la zona de memoria cuyo contenido se va a volcar en la pantalla. Las zonas de memoria donde se deberá volcar la información en cada una de las cuatro configuraciones son VRAM_A, VRAM_B, VRAM_C o VRAM_D, respectivamente.

En primer lugar hay que especificar el modo seleccionado, para lo cual se utiliza el registro REG_DISPCNT y la macro que se corresponda con el modo a utilizar. Por ejemplo, para usar

Tabla 6.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo
6.1	0’
6.2	0’
6.3	0’
6.4	0’
6.5	0’
6.6	0’
6.7	0’
6.8	0’

la configuración FB0 (y por lo tanto la zona de memoria VRAM_A) se deberá usar el siguiente código:

```
REG_DISPCNT = MODE_FB0;
```

Al especificar el modo de vídeo, indirectamente se está especificando la zona de memoria donde se deberán escribir los datos que se van a mostrar en la pantalla. Para poder utilizar los bancos de memoria es necesario que éstos estén habilitados y configurados según su finalidad. Para ello hay que usar el siguiente código:

```
VRAM_A_CR = VRAM_A_ENABLE | VRAM_A_LCD;
```

Una vez especificado el modo y habilitada la zona de memoria donde se escribirán los datos, el siguiente paso consiste en escribir en la memoria el valor que se asignará a cada uno de los píxeles de la pantalla, bien mediante una asignación manual, utilizando la macro RGB15 o escribiendo los datos correspondientes a una imagen específica.

6.1.2 Mostrar imágenes

El modo framebuffer no utiliza fondos ni motor de renderizado, por lo que la única tarea a realizar para mostrar una imagen es transferir al banco de memoria correspondiente los datos del gráfico a mostrar. Para ello, será necesario convertir esa imagen en un mapa de bits.

La conversión de imágenes se realiza con la herramienta *grit*, que ya viene integrada en *devkitPro*. En concreto, el comando *grit* se encuentra en el directorio *devkitPro/devkitARM/bin/grit.exe*. Por ejemplo, para convertir la imagen *imagen.png* al formato que se necesita, se deberá ejecutar la siguiente orden:

```
grit imagen.png -gb -gB16
```

La primera opción (-gb) indica que el formato de la conversión es un mapa de bits y la segunda opción (-gB16), que tiene una profundidad de 16 bits por píxel.

El comando creará un fichero de cabecera *imagen.h* y un fichero *imagen.s* que contiene el vector de datos correspondiente al mapa de bits de la imagen especificada (en lenguaje ensamblador ARM). El fichero de cabecera deberá ser incluido en el programa principal con la orden:

```
#include "imagen.h"
```

El fichero *imagen.s* deberá ser añadido al proyecto para que el linker puede añadir la imagen al ejecutable.

El siguiente listado es un programa que muestra la imagen *mario.png* en la pantalla inferior de la NDS:

```
#include <nds.h>
#include "mario.h"

int main (void)
{
  REG_POWERCNT = POWER_LCD | POWER_2D_A;
  REG_DISPCNT  = MODE_FB0;
  VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
  dmaCopy(marioBitmap, VRAM_A, 256*192*2);
  while (1)
  {
    swiWaitForVBlank();
  }
  return 0;
}
```

La función *dmaCopy* se encarga de copiar el vector de datos *marioBitmap* (declarado en mario.h) al banco de memoria (VRAM_A) que está mapeado a la pantalla.

En el siguiente código se cargan dos imágenes (una en VRAM_A y la otra en VRAM_B). Cuando se pulsa la tecla A (de la NDS) se mostrará una de ellas, cuando se pulsa la tecla X (de la NDS) se mostrará la otra:

```
#include <nds.h>
#include "mario.h"
#include "wallpaper.h"

int main (void)
{
  REG_POWERCNT = POWER_LCD | POWER_2D_A;

  VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
  dmaCopy(marioBitmap, VRAM_A, 256*192*2);

  VRAM_B_CR    = VRAM_ENABLE | VRAM_B_LCD;
  dmaCopy(wallpaperBitmap, VRAM_B, 256*192*2);

  // De inicio se muestra mario
  REG_DISPCNT  = MODE_FB0;
  while (1)
  {
    scanKeys();
    int held=keysHeld();

    if (held & KEY_A)
    {
      REG_DISPCNT  = MODE_FB0;
    }
    if (held & KEY_X)
    {
      REG_DISPCNT  = MODE_FB1;
    }

    swiWaitForVBlank();
  }
  return 0;
}
```

Como se puede comprobar, en primer lugar se carga la primera imagen en VRAM_A y la

segunda en VRAM_B. Para seleccionar que imagen se quiere mostrar, es necesario especificarlo cambiando el contenido del registro REG_DISPCNT.

6.1.3 Dibujar píxeles

Otra forma alternativa de representar gráficos en modo framebuffer consiste en componer una imagen a partir de la asignación de colores a cada uno de los píxeles que componen la pantalla. Al igual que en el caso anterior, el primer paso consistirá en especificar el modo de vídeo utilizado.

El siguiente paso difiere del apartado anterior, donde se hacía una transferencia directa de un vector de datos, correspondiente a la imagen a dibujar. En este caso, se accederá directamente a la región de memoria VRAM utilizada, para escribir uno por uno el valor del color que se desee asignar a cada uno de los píxeles que componen la pantalla. La macro *RGB15* permite obtener ese valor, indicando la cantidad de color rojo, verde y azul de la que se compone el color que se pintará en cada píxel.

Supongamos que se desea mostar un degradado vertical de color negro a azul, que ocupe toda la pantalla. El color negro se obtiene mediante la ausencia de color de las tres tonalidades, por lo tanto, mediante *RGB15(0,0,0)* obtenemos el valor correspondiente. Para hacer el degradado se deberá ir incrementando la cantidad de azul a medida que se vaya descendiendo en la pantalla. Dicho de otra forma, el nivel de color azul depende exclusivamente de la línea que estemos pintando. Por tanto basta con recorrer todos los puntos de la pantalla y pintarlos con el color correspondiente a la línea. El programa se podría implementar como sigue:

```
1  #include <nds.h>
2  #include <stdio.h>
3
4  int main (void)
5  {
6    REG_POWERCNT = POWER_LCD | POWER_2D_A;
7    REG_DISPCNT  = MODE_FB0;
8    VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
9
10   int lin, col;
11   unsigned short *fb = VRAM_A;
12   for(lin=0; lin<192; lin ++)
13     for(col=0; col<256; col ++)
14       fb[lin*256 + col] = RGB15 (0, 0, lin*32/192);
15
16   while(1)
17   {
18     swiWaitForVBlank();
19   }
20   return 0;
21 }
```

En la línea 11 se define un puntero de tipo entero sin signo de 16 bits y se hace que éste apunte a la dirección 0x6800000 (valor de la macro VRAM_A), que es el comienzo de la memoria del framebuffer FB0. El hecho de definir el puntero a la memoria como *unsigned short* es porque en este modo se necesita direccionar la memoria píxel a píxel de la pantalla que, como ya se ha comentado anteriormente, ocupan cada uno 16 bits (5 bits por componente y el bit más significativo sin usar).

En la línea 14, se utiliza el puntero fb para acceder al píxel correspondiente a la línea *lin* y a la columna *col*. Dado que los píxeles se almacenan por líneas y cada línea tiene 256 puntos tendremos que multiplicar el número de línea por 256 para ir al comienzo de la fila y sumar el número de columna para llegar al píxel correspondiente de la pantalla.

El siguiente programa explica como cambiar la imagen que se muestra en la pantalla. Al pulsar las teclas A, X y B (de la NDS) se mostrará la pantalla de color rojo, verde y azul, respectivamente. El código se muestra a continuación:

```
#include <nds.h>
#include <stdio.h>

int main (void)
{

  REG_POWERCNT = POWER_LCD | POWER_2D_A;
  VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
  VRAM_B_CR    = VRAM_ENABLE | VRAM_B_LCD;
  VRAM_C_CR    = VRAM_ENABLE | VRAM_A_LCD;

  u16 color_rojo  = RGB15(31,0,0);
  u16 color_verde = RGB15(0,31,0);
  u16 color_azul  = RGB15(0,0,31);

  unsigned short *fbA = VRAM_A;
  unsigned short *fbB = VRAM_B;
  unsigned short *fbC = VRAM_C;

  int lin,col;

  for(lin = 0; lin < 192; lin ++)
    for(col = 0; col < 256; col ++)
      fbA[lin*256 + col] = color_rojo;

  for(lin = 0; lin < 192; lin ++)
    for(col = 0; col < 256; col ++)
      fbB[lin*256 + col] = color_verde;

  for(lin = 0; lin < 192; lin ++)
    for(col = 0; col < 256; col ++)
      fbC[lin*256 + col] = color_azul;

  // de inicio FB0
  REG_DISPCNT  = MODE_FB0;
  while (1)
  {
    scanKeys();
    int held=keysHeld();
    if (held & KEY_A)
    {
      REG_DISPCNT  = MODE_FB0;
    }
    if (held & KEY_X)
    {
      REG_DISPCNT  = MODE_FB1;
    }
    if (held & KEY_B)
    {
      REG_DISPCNT  = MODE_FB2;
    }
    swiWaitForVBlank();
  }
  return 0;
}
```

6.2 El modo teselado

El modo teselado permite muchas más posibilidades a la hora de realizar videjuegos con gráficos que el modo framebuffer. En este apartado vamos a ver primero como funciona de forma básica este modo. Posteriormente se comentará de forma más extensa.

6.2.1 Funcionamiento básico del modo teselado

En el modo teselado, la pantalla se divide en celdas de 8x8 píxeles conocidas como teselas. Una tesela es un bitmap de 8x8 que representa un gráfico que se quiere mostrar en una de las celdas en las que se divide la pantalla.

Para definir una tesela, se deberá declarar un vector de 64 elementos ($8*8=64$) del tipo *u8*. Por ejemplo, el siguiente código define las figuras del comecocos, del fantasma y un fondo:

```
u8 comecocos[64] =
{
  2,2,1,1,1,1,2,2,
  2,2,1,1,1,1,1,2,
  2,2,2,1,1,1,1,1,
  2,2,2,2,1,1,1,1,
  2,2,2,2,1,1,1,1,
  2,2,2,1,1,1,1,1,
  2,2,1,1,1,1,1,2,
  2,2,1,1,1,1,2,2,
};

u8 fantasma[64] =
{
  2,2,2,3,3,2,2,2,
  2,3,3,3,3,3,3,2,
  3,3,3,3,3,3,3,3,
  3,4,2,3,3,4,2,3,
  3,4,2,3,3,4,2,3,
  3,3,3,3,3,3,3,3,
  3,3,3,3,3,3,3,3,
  3,2,3,2,2,3,2,3
};

u8 fondo[64] =
{
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2
};
```

Como se puede comprobar, se han definido los píxeles que componen la figura que representa la tesela. Se pueden definir hasta 1024 teselas diferentes. Existen dos formas de especificar los colores de los píxeles de las teselas:

- Modo de 256 colores: A cada píxel se le asigna un índice de la paleta de colores. El índice por lo tanto será un entero de 8 bits, con un valor comprendido entre 0 y 255. Este es el modo que se usará en esta sección. Por ejemplo, en las teselas anteriores, el número 1 indica el color almacenado en la posición 1 de la paleta de colores, el número 2 se refiere a la segunda posición, etc.

- Modo de 16 colores: En este caso un píxel es un índice a una paleta de 16 colores. Por lo tanto el índice será un entero de 4 bits, con un valor comprendido entre 0 y 15. Aunque este modo requiere menos memoria, dejando así más espacio para teselas y otros elementos en la VRAM, la calidad de la imagen renderizada se puede ver comprometida al disponer de solo 16 colores diferentes en cada tesela.

Para especificar los colores (en el modo de 256 colores), se debe modificar (en el programa principal) la paleta de colores mediante la variable *BG_PALETTE*. Por ejemplo, el código siguiente define los cuatro colores usados en las definiciones de las teselas anteriores:

```
  BG_PALETTE[1]=RGB15(28,0,0);
  BG_PALETTE[2]=RGB15(0,20,0);
  BG_PALETTE[3]=RGB15(0,0,31);
  BG_PALETTE[4]=RGB15(31,31,31);
```

En este modo de funcionamiento, el número de colores está limitado a 256 (de 0 a 255), número que se puede representar usando un byte. Por lo tanto, una tesela ocupa en memoria $8*8*1 = 64$ bytes.

Para definir cómo se muestran las teselas en la pantalla se puede declarar un vector que tendrá tantos elementos como teselas queramos mostrar. Por ejemplo, si queremos mostrar toda la pantalla, será necesario declarar el vector con 768 elementos del tipo u16 ($32*24 = 768$). En dicho vector, comúnmente llamado mapa de teselas, se especifica la tesela a mostrar usando un número entero (de 2 bytes). El identificador de la tesela es el orden en el que estará almacenada en la memoria.

El siguiente código declara el mapa de teselas para que cubra toda la pantalla (24 filas y 32 columnas):

```
u16 mapData[768] =
{
1,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,3,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,3,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2,   2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2
```

```
32 };
```

Como se puede comprobar, el elemento que se situará en la fila 0, columna 0 es la tesela con identificador 1 que será la que se almacenará en la memoria en primer lugar (será el comecocos), y la que se situará a su derecha es la tesela con identificador 2, que será la que se almacenará en segundo lugar (será el fondo). El mapa de teselas ocupará, en este caso, $768 * 2 = 1\,536$ bytes. Este vector se copiará en la memoria del fondo para que el motor gráfico sepa qué teselas tiene que mostrar en la pantalla.

Una de las principales ventajas de este modo es el ahorro en el uso de memoria que proporciona esta forma de trabajar. Esto es debido a que el mapa de teselas especifica donde está almacenada la tesela a mostrar, no el contenido de la misma. Es decir, en vez de guardar $24 * 32 * 64 = 49\,152$ bytes (24 filas, 32 columnas y cada tesela de 8×8 píxeles de 1 byte), se almacena $24 * 32 * 2 = 1\,536$ bytes (24 filas, 32 columnas y 2 bytes por cada identificador de la tesela) más 64 bytes por cada tesela usada. Como en este ejemplo se usan tres teselas, el tamaño total de memoria que se necesita es $1\,536 + 64 * 3 = 1\,728$ bytes, lo que supone un ahorro de más del 95 % de memoria.

Para configurar el modo teselado, deberemos usar las siguientes instrucciones:

```
1  REG_POWERCNT = POWER_ALL_2D;
2  REG_DISPCNT  = MODE_0_2D     | DISPLAY_BG0_ACTIVE ;
3  VRAM_A_CR    = VRAM_ENABLE   | VRAM_A_MAIN_BG ;
4  BGCTRL [0]   = BG_32x32      | BG_COLOR_256 | BG_MAP_BASE(0) |
       BG_TILE_BASE(1);
```

La línea 1 activa ambos motores gráficos. En la segunda línea indicamos que vamos a usar el modo 0 y el fondo 0 (ver Figura **??**) del motor principal. En la tercera línea activamos el banco de memoria *VRAM_A* y lo asignamos como memoria de fondos del motor principal. La cuarta línea se utiliza para configurar la memoria de fondos del fondo 0. Con *BG_32_32* elegimos el número de teselas máximo que podemos usar. Hay cuatro posibilidades 32×32, 32×64, 64×32 y 64×64. En nuestro caso se ha seleccionado 32×32. Con *BG_COLOR_256* indicamos que las teselas tendrán 256 colores que se especificarán en la paleta. Con *BG_MAP_BASE(0)* indicamos donde se va a guardar el mapa de teselas. En este caso el mapa de teselas se almacenará en el primer bloque (para mapas) de la *VRAM_A*. Por último, con *BG_TILE_BASE(1)* indicamos donde se almacenarán las teselas, que en este caso es a partir del segundo bloque (para teselas) de la *VRAM_A*.

La memoria *VRAM_A* tiene un tamaño de 128KB (ver Tabla **??**). Tanto la memoria de teselas como las teselas se deben copiar a este banco de memoria. Los mapas de teselas se almacenan en bloques de 2KB. Hay 32 posibles posiciones (de la 0 a la 31). En este ejemplo, el mapa de teselas se almacenará en el primer bloque mediante la macro *BG_MAP_BASE(0)*. Como el mapa de teselas ocupa $24 * 32 * 2 = 1\,536$ bytes, cabe perfectamente en un único bloque de 2KB. Si no cupiera, se deberían usar más bloques.

Las teselas se guardan en bloques de 16KB. Hay 16 posibles posiciones (de la 0 a la 15). En este ejemplo, las tres teselas se almacenarán en el segundo bloque mediante la macro *BG_TILE_BASE(1)*. Como las tres teselas ocupan $8 * 8 * 1 * 3 = 192$ bytes, cabe perfectamente en un único bloque de 16KB. Si no cupiera, se deberían usar más bloques.

Hay que tener en cuenta que la zona usada para almacenar el mapa de tesela no se puede solapar con la zona usada para almacenar las teselas. Por eso las teselas empiezan en el bloque 1, puesto que parte de la memoria del bloque 0 ya está ocupada por el mapa de teselas.

Una vez configurado el modo teselado, hemos de copiar a memoria tanto el mapa de teselas como las propias teselas. El código siguiente muestra este proceso:

```
1  static u8*  tileMemory = (u8*)  BG_TILE_RAM(1);
2  static u16* mapMemory  = (u16*) BG_MAP_RAM(0);
```

```
3
4     // Copia de teselas
5     dmaCopy(comecocos,tileMemory + 64,  sizeof(comecocos));
6     dmaCopy(fondo,    tileMemory + 128, sizeof(fondo));
7     dmaCopy(fantasma, tileMemory + 192, sizeof(fantasma));
8
9     // Copia del mapa de teselas a la memoria gráfica
10    pos_mapData = 0;
11    for(i=0;i<32;i++)
12      for(j=0;j<24;j++)
13      {
14        pos_mapMemory             = i*24+j;
15        mapMemory[pos_mapMemory] = mapData[pos_mapData];
16        pos_mapData ++;
17      }
```

En las dos primeras líneas creamos dos variables para acceder a la zona de la memoria en VRAM_A donde estará almacenado el mapa de teselas y a la zona de memoria (también en VRAM_A) donde estarán almacenadas las teselas. Es importante destacar que ambas variables se refieren a posiciones de memoria dentro de la *VRAM_A*, aunque cada una apunta a una zona diferente dentro de esta memoria.

En las líneas 5, 6 y 7 se copian las teselas a la zona de memoria que le corresponde, empezando por *tileMemory + 64* para la primera tesela. En este caso, la primera tesela es el comecocos, seguida del fondo y por último el fantasma. Por lo tanto, en el mapa de teselas pondremos 1 cuando queramos mostrar el comecocos, 2 para el fondo y 3 para el fantasma. Es importante destacar que las teselas se han empezado a almacenar a partir de la posición base + 64, puesto que la primera tesela que usamos tiene como índice 1 (y no 0). El desplazamiento es de 64 bytes puesto que cada tesela, en el modo de 256 colores, ocupa 8×8 píxeles de un byte, es decir 64 bytes. Si se especificara las teselas en el modo de 16 colores, el desplazamiento sería de 32 bytes, ya que en este caso la tesela ocupa 8×8 píxeles de un medio byte (4 bits).

En las líneas 11 a 17 se copia el mapa de teselas a la memoria gráfica.

A continuación se muestra el código completo del programa que muestra como resultado el gráfico que se muestra en la Figura 7.1

```
1  #include <nds.h>
2  #include <stdio.h>
3
4  u8 comecocos[64] =
5  {
6    2,2,1,1,1,1,2,2,
7    2,1,1,1,1,1,2,2,
8    1,1,1,1,1,2,2,2,
9    1,1,1,1,2,2,2,2,
10   1,1,1,1,2,2,2,2,
11   1,1,1,1,1,2,2,2,
12   2,1,1,1,1,1,2,2,
13   2,2,1,1,1,1,2,2,
14 };
15
16 u8 fondo[64] =
17 {
18   2,2,2,2,2,2,2,2,
19   2,2,2,2,2,2,2,2,
20   2,2,2,2,2,2,2,2,
21   2,2,2,2,2,2,2,2,
22   2,2,2,2,2,2,2,2,
```

```
23    2,2,2,2,2,2,2,2,
24    2,2,2,2,2,2,2,2,
25    2,2,2,2,2,2,2,2
26  };
27
28  u8 fantasma[64] =
29  {
30    2,2,2,3,3,2,2,2,
31    2,3,3,3,3,3,3,2,
32    3,3,3,3,3,3,3,3,
33    3,4,2,3,3,4,2,3,
34    3,4,2,3,3,4,2,3,
35    3,3,3,3,3,3,3,3,
36    3,3,3,3,3,3,3,3,
37    3,2,3,2,2,3,2,3
38  };
39
40  // Mapa de teselas 32x24 = 768
41  u16 mapData[768] =
42  {
43  1,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
44  2,2,2,3,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
45  2,3,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
46  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
47
48  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
49  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
50  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
51  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
52
53  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
54  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
55  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
56  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
57
58  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
59  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
60  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
61  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
62
63  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
64  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
65  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
66  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
67
68  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
69  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
70  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
71  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2
72  };
73
74  int main(void)
75  {
76    int i,j,pos_mapMemory,pos_mapData;
77
78    REG_POWERCNT = POWER_ALL_2D;
79    REG_DISPCNT  = MODE_0_2D    | DISPLAY_BG0_ACTIVE ;
80    VRAM_A_CR    = VRAM_ENABLE | VRAM_A_MAIN_BG ;
81    BGCTRL[0]    = BG_32x32     | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE
          (1);
82
```

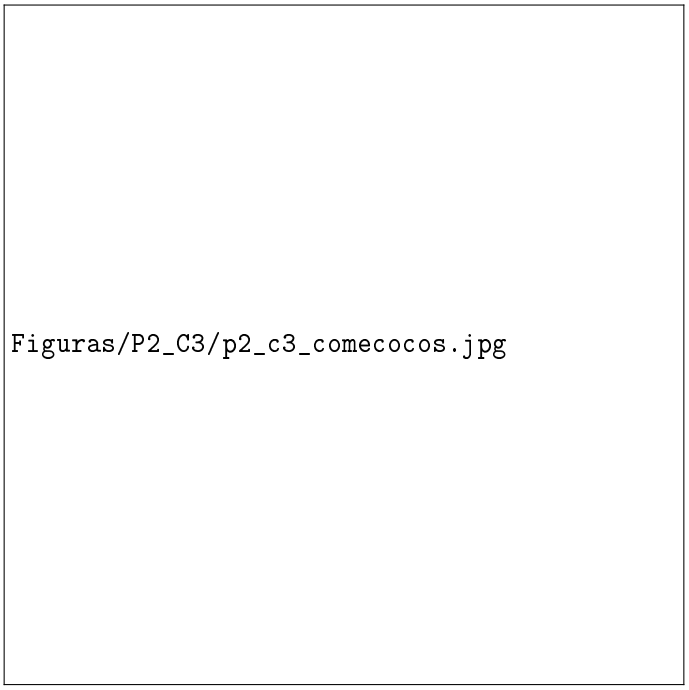

Figura 6.1: Salida del programa de esta sección en la NDS

```
83    BG_PALETTE[1]=RGB15(28,0,0);
84    BG_PALETTE[2]=RGB15(0,20,0);
85    BG_PALETTE[3]=RGB15(0,0,31);
86    BG_PALETTE[4]=RGB15(31,31,31);
87
88    static u8*  tileMemory = (u8*)  BG_TILE_RAM(1);
89    static u16*  mapMemory = (u16*) BG_MAP_RAM(0);
90
91    dmaCopy(comecocos, tileMemory + 64,  sizeof(comecocos));
92    dmaCopy(fondo,     tileMemory + 128, sizeof(fondo));
93    dmaCopy(fantasma,  tileMemory + 192, sizeof(fantasma));
94
95    pos_mapData = 0;
96    for(i=0;i<32;i++)
97      for(j=0;j<24;j++)
98      {
99        pos_mapMemory             = i*24+j;
100       mapMemory[pos_mapMemory] = mapData[pos_mapData];
101       pos_mapData ++;
102     }
103
104   while(1)
105   {
106     swiWaitForVBlank();
107   }
108 }
```

Si queremos que la pantalla se modifique como respuesta a un evento, únicamente tenemos que cambiar la memoria de teselas (mapMemory) correspondiente a la posición que queremos cambiar, modificando el índice a la tesela que queremos mostrar.

Por ejemplo, el siguiente fragmento de código mostrará al pulsar la tecla A de la NDS, un segundo comecocos en la celda situada en la fila 15 y columna 16:

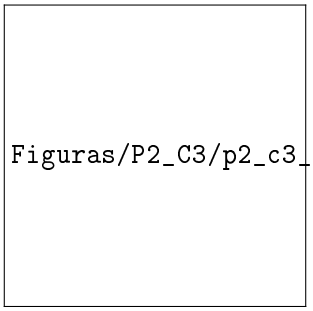

Figura 6.2: Significados de los bits del registro *REG_BGnCNT*.

```
while(1)
{
  scanKeys();
  int key = keysDown();

  if (key & KEY_A)
  {
    i                          = 15;
    j                          = 16;
    pos_mapMemory              = i*24+j;
    mapMemory[pos_mapMemory] = 1;
  }
  swiWaitForVBlank();
}
```

6.2.2 Funcionamiento avanzado del modo teselado

El motor gráfico renderiza los fondos teselados a partir de las entradas del *mapa* y la *colección de teselas* a las que hace referencia, estando estos datos contenidos en la memoria de fondos de la VRAM. Teniendo en cuenta que cada motor gráfico puede gestionar un máximo de cuatro fondos, es necesario que cada fondo activado configure la localización de sus datos de mapas y teselas. Para ello se utiliza el registro de configuración del fondo, *REG_BGnCNT* o *BGCTRL[n]*, donde *n* hace referencia al índice del fondo, tomando valores de 0 a 3. Este registro reserva 4 bits para especificar el desplazamiento de las teselas (en múltiplos de 16 KB) y 5 bits para el mapa (en múltiplos de 2 KB), de tal forma que las teselas admiten $2^4 = 16$ posibles valores del desplazamiento base y los mapas admiten $2^5 = 32$. La Figura 7.2 muestra el significado de cada uno de sus bits.

Guardar mapas de teselas en la VRAM

Un *mapa de teselas* puede empezar en cualquier dirección múltiplo de 2KB entre 0 x 2KB y 31 x 2KB. Estos valores expresados en hexadecimal serían desde 0 x 0x800 hasta 31 x 0x800. Para que se facilite la configuración de las direcciones base, *libnds* incluye la macro *BG_MAP_BASE(n)*, donde *n* se corresponde con el múltiplo de *2KB* seleccionado, entre 0 y 31. Por ejemplo, si se indica en el registro de configuración del fondo que los datos del mapa se encuentran ubicados a partir de *BG_MAP_BASE(1)*, significa que comenzarán en la posición de memoria *0x0800* a partir del principio de la memoria de fondos. Análogamente, decir que lo hacen en *BG_MAP_BASE(31)* significa que empiezan en la posición *0xf800*, tal y como se puede comprobar en la figura 7.3.

El motor de vídeo no tiene memoria física dedicada, por tanto, *parte de la memoria VRAM se debe configurar para ser utilizada como memoria de fondos*. El motor gráfico principal utiliza como memoria de fondos el rango de memoria de *0x06000000* a *0x0607ffff* (512KB máximo) mientras que el secundario emplea el rango de *0x06200000* a *0x0621ffff* (128KB máximo). Todos

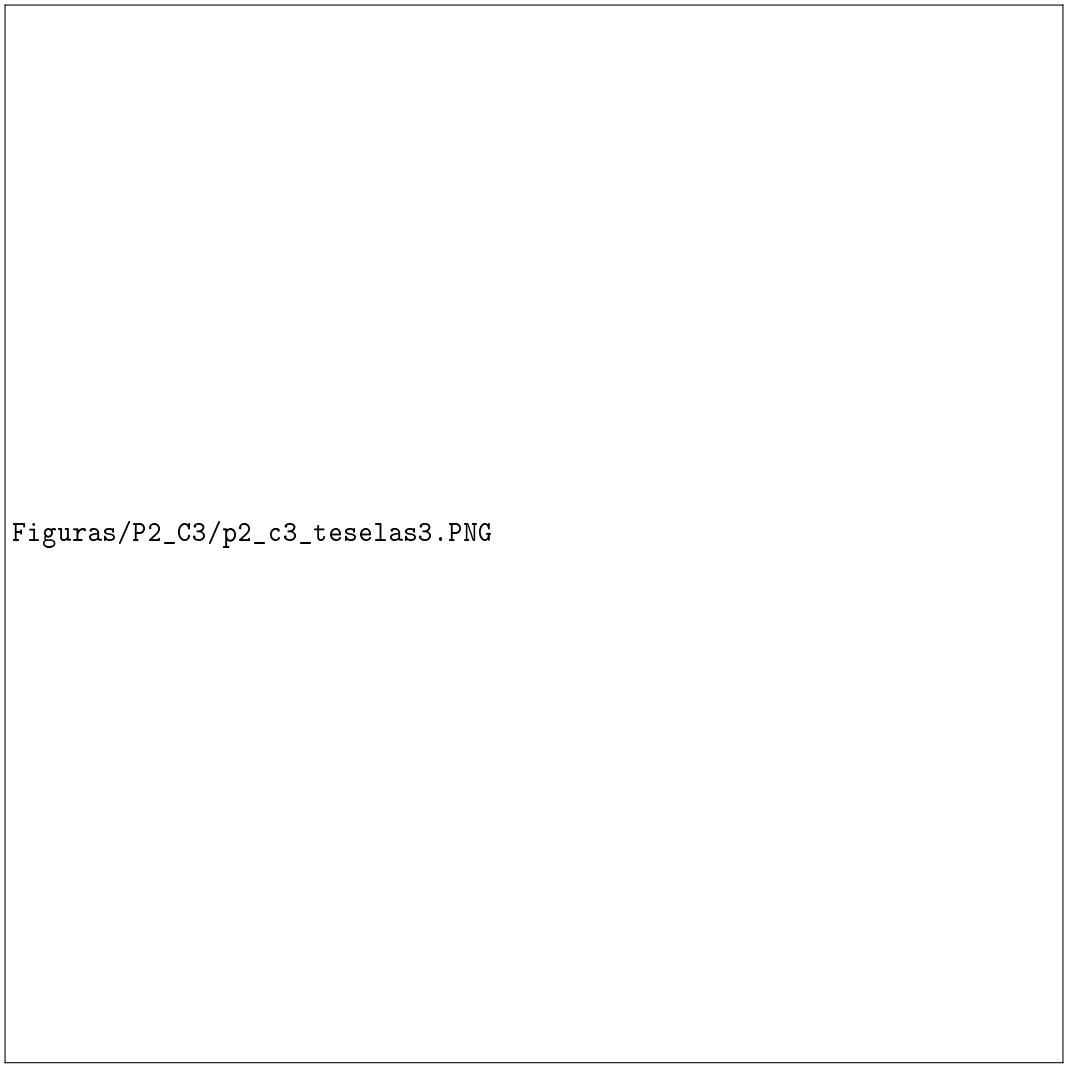

Figura 6.3: Desplazamientos en la VRAM.

los desplazamientos base de mapas y teselas se llevan a cabo tomando como dirección base el comienzo de estos rangos de memoria.

El tamaño de un mapa de teselas depende del tamaño del fondo:

- Un mapa de 32x32 teselas será una sucesión de 32x32 entradas de 16 bits (aspecto que se explicará con más detalle a continuación), por lo que el tamaño total será de 2x32x32 = 2KB. Es decir, en este caso el mapa entero cabe antes de la siguiente dirección base de otro mapa.
- Sin embargo, para un fondo de 64x64 teselas, los datos del mapa requieren 2x64x64 = 8KB, es decir, las siguientes 3 posibles direcciones base no estarían disponibles para otro mapa.

Estos cálculos se muestran de manera gráfica en la Figura 7.4.

Como se ha comentado previamente cada una de las entradas del mapa se representa mediante un conjunto de 16 bits, donde los 10 bits menos significativos especifican la tesela, razón por la que el conjunto máximo de teselas es de 1024, las máximas referenciables con 10 bits. Los dos bits siguientes se utilizan para especificar el efecto de espejo en la tesela, es decir, si tiene una reflexión horizontal, vertical o ambas. Finalmente, para teselas que utilizan paletas de 16 colores, se utilizan los 4 últimos bits para identificar la paleta de colores utilizada por esa tesela. La Figura 7.5 muestra dicha distribución.

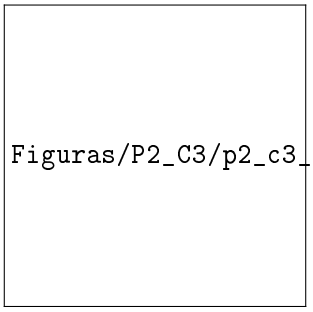

Figura 6.4: Cálculo del tamaño de un mapa de teselas.

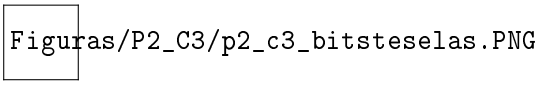

Figura 6.5: Entradas del mapa de teselas.

Guardar teselas en la VRAM

En cuanto a las teselas, ocurre lo mismo que para los mapas, ya que ambos comparten la misma memoria de fondos y al igual que los mapas, utilizarán los desplazamientos para gestionar esa memoria como si estuviera organizada en bloques, sólo que en este caso, de *tamaño 16KB*. La macro utilizada para especificar la dirección base es *BG_TILE_BASE(n)*, donde *n* toma valores entre 0 y 15. El número hace referencia al múltiplo de 16KB correspondiente, empezando en 0x0000, desplazándose en múltiplos de 0x4000 hasta el valor 0x3c000, tal y como se puede observar en la figura 7.3.

7. El modo teselado

Este capítulo explica el funcionamiento del modo teselado. La lista de ejercicios y el tiempo estimado (en minutos) para su realización se muestran en la Tabla 7.1.

7.1 El modo teselado

El modo teselado permite muchas más posibilidades a la hora de realizar videjuegos con gráficos que el modo framebuffer. En este apartado vamos a ver primero como funciona de forma básica este modo. Posteriormente se comentará de forma más extensa.

7.1.1 Funcionamiento básico del modo teselado

En el modo teselado, la pantalla se divide en celdas de 8x8 píxeles conocidas como teselas. Una tesela es un bitmap de 8x8 que representa un gráfico que se quiere mostrar en una de las celdas en las que se divide la pantalla.

Para definir una tesela, se deberá declarar un vector de 64 elementos ($8 * 8 = 64$) del tipo *u8*. Por ejemplo, el siguiente código define las figuras del comecocos, del fantasma y un fondo:

```
u8 comecocos[64] =
{
  2,2,1,1,1,1,2,2,
  2,2,1,1,1,1,1,2,
  2,2,2,1,1,1,1,1,
  2,2,2,2,1,1,1,1,
  2,2,2,2,1,1,1,1,
  2,2,2,1,1,1,1,1,
  2,2,1,1,1,1,1,2,
  2,2,1,1,1,1,2,2,
};

u8 fantasma[64] =
{
  2,2,2,3,3,2,2,2,
```

Tabla 7.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo
7.1	0’
7.2	0’
7.3	0’
7.4	0’
7.5	0’
7.6	0’
7.7	0’
7.8	0’

```
16    2,3,3,3,3,3,3,2,
17    3,3,3,3,3,3,3,3,
18    3,4,2,3,3,4,2,3,
19    3,4,2,3,3,4,2,3,
20    3,3,3,3,3,3,3,3,
21    3,3,3,3,3,3,3,3,
22    3,2,3,2,2,3,2,3
23  };
24
25  u8 fondo[64] =
26  {
27    2,2,2,2,2,2,2,2,
28    2,2,2,2,2,2,2,2,
29    2,2,2,2,2,2,2,2,
30    2,2,2,2,2,2,2,2,
31    2,2,2,2,2,2,2,2,
32    2,2,2,2,2,2,2,2,
33    2,2,2,2,2,2,2,2,
34    2,2,2,2,2,2,2,2
35  };
```

Como se puede comprobar, se han definido los píxeles que componen la figura que representa la tesela. Se pueden definir hasta 1024 teselas diferentes. Existen dos formas de especificar los colores de los píxeles de las teselas:

- Modo de 256 colores: A cada píxel se le asigna un índice de la paleta de colores. El índice por lo tanto será un entero de 8 bits, con un valor comprendido entre 0 y 255. Este es el modo que se usará en esta sección. Por ejemplo, en las teselas anteriores, el número 1 indica el color almacenado en la posición 1 de la paleta de colores, el número 2 se refiere a la segunda posición, etc.
- Modo de 16 colores: En este caso un píxel es un índice a una paleta de 16 colores. Por lo tanto el índice será un entero de 4 bits, con un valor comprendido entre 0 y 15. Aunque este modo requiere menos memoria, dejando así más espacio para teselas y otros elementos en la VRAM, la calidad de la imagen renderizada se puede ver comprometida al disponer de solo 16 colores diferentes en cada tesela.

Para especificar los colores (en el modo de 256 colores), se debe modificar (en el programa principal) la paleta de colores mediante la variable *BG_PALETTE*. Por ejemplo, el código siguiente define los cuatro colores usados en las definiciones de las teselas anteriores:

```
1   BG_PALETTE[1]=RGB15(28,0,0);
2   BG_PALETTE[2]=RGB15(0,20,0);
3   BG_PALETTE[3]=RGB15(0,0,31);
4   BG_PALETTE[4]=RGB15(31,31,31);
```

En este modo de funcionamiento, el número de colores está limitado a 256 (de 0 a 255), número que se puede representar usando un byte. Por lo tanto, una tesela ocupa en memoria $8*8*1 = 64$ bytes.

Para definir cómo se muestran las teselas en la pantalla se puede declarar un vector que tendrá tantos elementos como teselas queramos mostrar. Por ejemplo, si queremos mostrar toda la pantalla, será necesario declarar el vector con 768 elementos del tipo u16 ($32*24 = 768$). En dicho vector, comúnmente llamado mapa de teselas, se especifica la tesela a mostrar usando un número entero (de 2 bytes). El identificador de la tesela es el orden en el que estará almacenada en la memoria.

El siguiente código declara el mapa de teselas para que cubra toda la pantalla (24 filas y 32 columnas):

```
u16 mapData[768] =
{
1,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,3,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,3,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2
};
```

Como se puede comprobar, el elemento que se situará en la fila 0, columna 0 es la tesela con identificador 1 que será la que se almacenará en la memoria en primer lugar (será el comecocos), y la que se situará a su derecha es la tesela con identificador 2, que será la que se almacenará en segundo lugar (será el fondo). El mapa de teselas ocupará, en este caso, $768*2 = 1\ 536$ bytes. Este vector se copiará en la memoria del fondo para que el motor gráfico sepa qué teselas tiene que mostrar en la pantalla.

Una de las principales ventajas de este modo es el ahorro en el uso de memoria que proporciona esta forma de trabajar. Esto es debido a que el mapa de teselas especifica donde está almacenada la tesela a mostrar, no el contenido de la misma. Es decir, en vez de guardar $24*32*64 = 49\ 152$ bytes (24 filas, 32 columnas y cada tesela de 8×8 píxeles de 1 byte), se almacena $24*32*2 = 1\ 536$

bytes (24 filas, 32 columnas y 2 bytes por cada identificador de la tesela) más 64 bytes por cada tesela usada. Como en este ejemplo se usan tres teselas, el tamaño total de memoria que se necesita es $1\,536 + 64 * 3 = 1\,728$ bytes, lo que supone un ahorro de más del 95 % de memoria.

Para configurar el modo teselado, deberemos usar las siguientes instrucciones:

```
1   REG_POWERCNT = POWER_ALL_2D;
2   REG_DISPCNT  = MODE_0_2D     | DISPLAY_BG0_ACTIVE ;
3   VRAM_A_CR    = VRAM_ENABLE   | VRAM_A_MAIN_BG ;
4   BGCTRL [0]   = BG_32x32      | BG_COLOR_256 | BG_MAP_BASE(0) |
        BG_TILE_BASE(1);
```

La línea 1 activa ambos motores gráficos. En la segunda línea indicamos que vamos a usar el modo 0 y el fondo 0 (ver Figura **??**) del motor principal. En la tercera línea activamos el banco de memoria *VRAM_A* y lo asignamos como memoria de fondos del motor principal. La cuarta línea se utiliza para configurar la memoria de fondos del fondo 0. Con *BG_32_32* elegimos el número de teselas máximo que podemos usar. Hay cuatro posibilidades 32×32, 32×64, 64×32 y 64×64. En nuestro caso se ha seleccionado 32×32. Con *BG_COLOR_256* indicamos que las teselas tendrán 256 colores que se especificarán en la paleta. Con *BG_MAP_BASE(0)* indicamos donde se va a guardar el mapa de teselas. En este caso el mapa de teselas se almacenará en el primer bloque (para mapas) de la *VRAM_A*. Por último, con *BG_TILE_BASE(1)* indicamos donde se almacenarán las teselas, que en este caso es a partir del segundo bloque (para teselas) de la *VRAM_A*.

La memoria *VRAM_A* tiene un tamaño de 128KB (ver Tabla **??**). Tanto la memoria de teselas como las teselas se deben copiar a este banco de memoria. Los mapas de teselas se almacenan en bloques de 2KB. Hay 32 posibles posiciones (de la 0 a la 31). En este ejemplo, el mapa de teselas se almacenará en el primer bloque mediante la macro *BG_MAP_BASE(0)*. Como el mapa de teselas ocupa $24 * 32 * 2 = 1\,536$ bytes, cabe perfectamente en un único bloque de 2KB. Si no cupiera, se deberían usar más bloques.

Las teselas se guardan en bloques de 16KB. Hay 16 posibles posiciones (de la 0 a la 15). En este ejemplo, las tres teselas se almacenarán en el segundo bloque mediante la macro *BG_TILE_BASE(1)*. Como las tres teselas ocupan $8 * 8 * 1 * 3 = 192$ bytes, cabe perfectamente en un único bloque de 16KB. Si no cupiera, se deberían usar más bloques.

Hay que tener en cuenta que la zona usada para almacenar el mapa de tesela no se puede solapar con la zona usada para almacenar las teselas. Por eso las teselas empiezan en el bloque 1, puesto que parte de la memoria del bloque 0 ya está ocupada por el mapa de teselas.

Una vez configurado el modo teselado, hemos de copiar a memoria tanto el mapa de teselas como las propias teselas. El código siguiente muestra este proceso:

```
1   static u8*  tileMemory = (u8*)  BG_TILE_RAM(1);
2   static u16* mapMemory  = (u16*) BG_MAP_RAM(0);
3
4   // Copia de teselas
5   dmaCopy(comecocos,tileMemory + 64,  sizeof(comecocos));
6   dmaCopy(fondo,    tileMemory + 128, sizeof(fondo));
7   dmaCopy(fantasma, tileMemory + 192, sizeof(fantasma));
8
9   // Copia del mapa de teselas a la memoria gráfica
10  pos_mapData = 0;
11  for(i=0;i<32;i++)
12    for(j=0;j<24;j++)
13    {
14      pos_mapMemory             = i*24+j;
15      mapMemory[pos_mapMemory] = mapData[pos_mapData];
16      pos_mapData ++;
17    }
```

En las dos primeras líneas creamos dos variables para acceder a la zona de la memoria en VRAM_A donde estará almacenado el mapa de teselas y a la zona de memoria (también en VRAM_A) donde estarán almacenadas las teselas. Es importante destacar que ambas variables se refieren a posiciones de memoria dentro de la *VRAM_A*, aunque cada una apunta a una zona diferente dentro de esta memoria.

En las líneas 5, 6 y 7 se copian las teselas a la zona de memoria que le corresponde, empezando por *tileMemory + 64* para la primera tesela. En este caso, la primera tesela es el comecocos, seguida del fondo y por último el fantasma. Por lo tanto, en el mapa de teselas pondremos 1 cuando queramos mostrar el comecocos, 2 para el fondo y 3 para el fantasma. Es importante destacar que las teselas se han empezado a almacenar a partir de la posición base + 64, puesto que la primera tesela que usamos tiene como índice 1 (y no 0). El desplazamiento es de 64 bytes puesto que cada tesela, en el modo de 256 colores, ocupa 8×8 píxeles de un byte, es decir 64 bytes. Si se especificara las teselas en el modo de 16 colores, el desplazamiento sería de 32 bytes, ya que en este caso la tesela ocupa 8×8 píxeles de un medio byte (4 bits).

En las líneas 11 a 17 se copia el mapa de teselas a la memoria gráfica.

A continuación se muestra el código completo del programa que muestra como resultado el gráfico que se muestra en la Figura 7.1

```
1  #include <nds.h>
2  #include <stdio.h>
3
4  u8 comecocos[64] =
5  {
6    2,2,1,1,1,1,2,2,
7    2,1,1,1,1,1,2,2,
8    1,1,1,1,1,2,2,2,
9    1,1,1,1,2,2,2,2,
10   1,1,1,1,2,2,2,2,
11   1,1,1,1,1,2,2,2,
12   2,1,1,1,1,1,2,2,
13   2,2,1,1,1,1,2,2,
14 };
15
16 u8 fondo[64] =
17 {
18   2,2,2,2,2,2,2,2,
19   2,2,2,2,2,2,2,2,
20   2,2,2,2,2,2,2,2,
21   2,2,2,2,2,2,2,2,
22   2,2,2,2,2,2,2,2,
23   2,2,2,2,2,2,2,2,
24   2,2,2,2,2,2,2,2,
25   2,2,2,2,2,2,2,2
26 };
27
28 u8 fantasma[64] =
29 {
30   2,2,2,3,3,2,2,2,
31   2,3,3,3,3,3,3,2,
32   3,3,3,3,3,3,3,3,
33   3,4,2,3,3,4,2,3,
34   3,4,2,3,3,4,2,3,
35   3,3,3,3,3,3,3,3,
36   3,3,3,3,3,3,3,3,
37   3,2,3,2,2,3,2,3
```

```
};

// Mapa de teselas 32x24 = 768
u16 mapData[768] =
{
1,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,3,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,3,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,

2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,  2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2
};

int main(void)
{
  int i,j,pos_mapMemory,pos_mapData;

  REG_POWERCNT = POWER_ALL_2D;
  REG_DISPCNT  = MODE_0_2D    | DISPLAY_BG0_ACTIVE ;
  VRAM_A_CR    = VRAM_ENABLE | VRAM_A_MAIN_BG ;
  BGCTRL[0]    = BG_32x32    | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);

  BG_PALETTE[1]=RGB15(28,0,0);
  BG_PALETTE[2]=RGB15(0,20,0);
  BG_PALETTE[3]=RGB15(0,0,31);
  BG_PALETTE[4]=RGB15(31,31,31);

  static u8*  tileMemory = (u8*)  BG_TILE_RAM(1);
  static u16*  mapMemory = (u16*) BG_MAP_RAM(0);

  dmaCopy(comecocos, tileMemory + 64,  sizeof(comecocos));
  dmaCopy(fondo,     tileMemory + 128, sizeof(fondo));
  dmaCopy(fantasma,  tileMemory + 192, sizeof(fantasma));

  pos_mapData = 0;
  for(i=0;i<32;i++)
    for(j=0;j<24;j++)
```

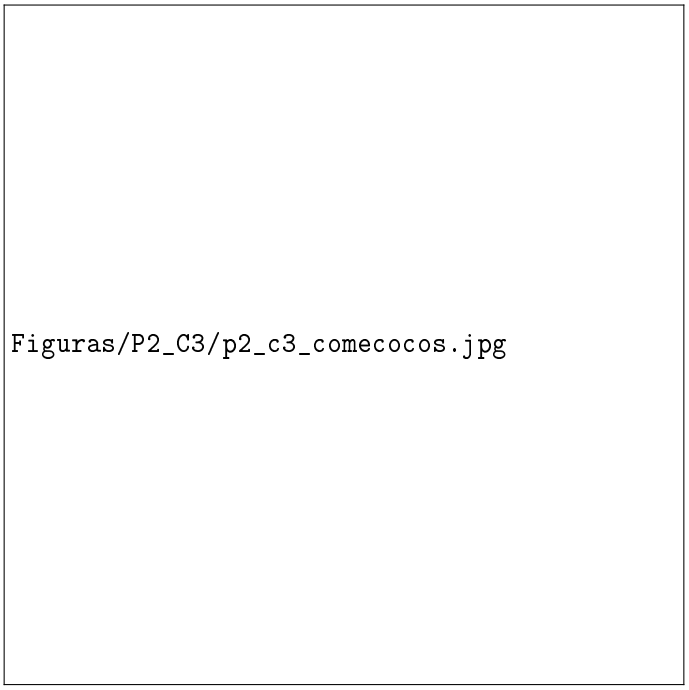

Figura 7.1: Salida del programa de esta sección en la NDS

```
98        {
99          pos_mapMemory             = i*24+j;
100         mapMemory[pos_mapMemory] = mapData[pos_mapData];
101         pos_mapData ++;
102       }
103
104     while(1)
105     {
106       swiWaitForVBlank();
107     }
108   }
```

Si queremos que la pantalla se modifique como respuesta a un evento, únicamente tenemos que cambiar la memoria de teselas (mapMemory) correspondiente a la posición que queremos cambiar, modificando el índice a la tesela que queremos mostrar.

Por ejemplo, el siguiente fragmento de código mostrará al pulsar la tecla A de la NDS, un segundo comecocos en la celda situada en la fila 15 y columna 16:

```
1   while(1)
2   {
3     scanKeys();
4     int key = keysDown();
5
6     if (key & KEY_A)
7     {
8       i                          = 15;
9       j                          = 16;
10      pos_mapMemory              = i*24+j;
11      mapMemory[pos_mapMemory] = 1;
12    }
13    swiWaitForVBlank();
14  }
```

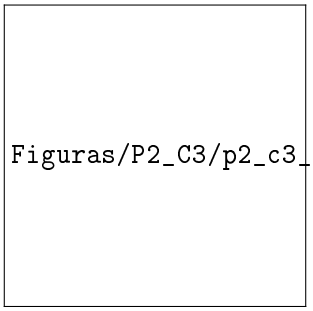

Figura 7.2: Significados de los bits del registro *REG_BGnCNT*.

7.1.2 Funcionamiento avanzado del modo teselado

El motor gráfico renderiza los fondos teselados a partir de las entradas del *mapa* y la *colección de teselas* a las que hace referencia, estando estos datos contenidos en la memoria de fondos de la VRAM. Teniendo en cuenta que cada motor gráfico puede gestionar un máximo de cuatro fondos, es necesario que cada fondo activado configure la localización de sus datos de mapas y teselas. Para ello se utiliza el registro de configuración del fondo, *REG_BGnCNT* o *BGCTRL[n]*, donde *n* hace referencia al índice del fondo, tomando valores de 0 a 3. Este registro reserva 4 bits para especificar el desplazamiento de las teselas (en múltiplos de 16 KB) y 5 bits para el mapa (en múltiplos de 2 KB), de tal forma que las teselas admiten $2^4 = 16$ posibles valores del desplazamiento base y los mapas admiten $2^5 = 32$. La Figura 7.2 muestra el significado de cada uno de sus bits.

Guardar mapas de teselas en la VRAM

Un *mapa de teselas* puede empezar en cualquier dirección múltiplo de 2KB entre 0 x 2KB y 31 x 2KB. Estos valores expresados en hexadecimal serían desde 0 x 0x800 hasta 31 x 0x800. Para que se facilite la configuración de las direcciones base, *libnds* incluye la macro *BG_MAP_BASE(n)*, donde *n* se corresponde con el múltiplo de *2KB* seleccionado, entre 0 y 31. Por ejemplo, si se indica en el registro de configuración del fondo que los datos del mapa se encuentran ubicados a partir de *BG_MAP_BASE(1)*, significa que comenzarán en la posición de memoria *0x0800* a partir del principio de la memoria de fondos. Análogamente, decir que lo hacen en *BG_MAP_BASE(31)* significa que empiezan en la posición *0xf800*, tal y como se puede comprobar en la figura 7.3.

El motor de vídeo no tiene memoria física dedicada, por tanto, *parte de la memoria VRAM se debe configurar para ser utilizada como memoria de fondos*. El motor gráfico principal utiliza como memoria de fondos el rango de memoria de *0x06000000* a *0x0607ffff* (512KB máximo) mientras que el secundario emplea el rango de *0x06200000* a *0x0621ffff* (128KB máximo). Todos los desplazamientos base de mapas y teselas se llevan a cabo tomando como dirección base el comienzo de estos rangos de memoria.

El tamaño de un mapa de teselas depende del tamaño del fondo:

- Un mapa de 32x32 teselas será una sucesión de 32x32 entradas de 16 bits (aspecto que se explicará con más detalle a continuación), por lo que el tamaño total será de 2x32x32 = 2KB. Es decir, en este caso el mapa entero cabe antes de la siguiente dirección base de otro mapa.
- Sin embargo, para un fondo de 64x64 teselas, los datos del mapa requieren 2x64x64 = 8KB, es decir, las siguientes 3 posibles direcciones base no estarían disponibles para otro mapa.

Estos cálculos se muestran de manera gráfica en la Figura 7.4.

Como se ha comentado previamente cada una de las entradas del mapa se representa mediante un conjunto de 16 bits, donde los 10 bits menos significativos especifican la tesela, razón por la que el conjunto máximo de teselas es de 1024, las máximas referenciables con 10 bits. Los dos bits siguientes se utilizan para especificar el efecto de espejo en la tesela, es decir, si tiene una reflexión horizontal, vertical o ambas. Finalmente, para teselas que utilizan paletas de 16 colores, se utilizan

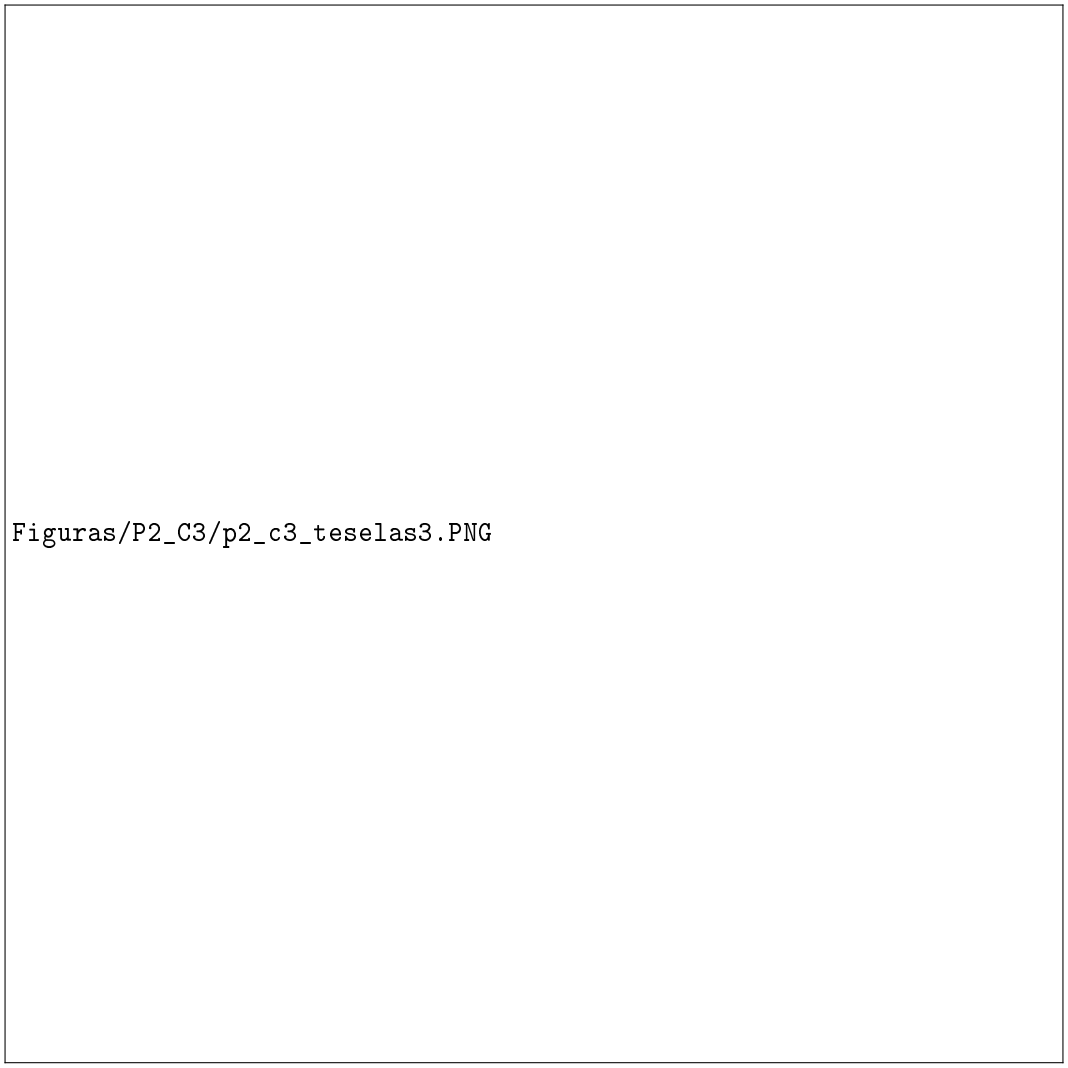

Figura 7.3: Desplazamientos en la VRAM.

los 4 últimos bits para identificar la paleta de colores utilizada por esa tesela. La Figura 7.5 muestra dicha distribución.

Guardar teselas en la VRAM

En cuanto a las teselas, ocurre lo mismo que para los mapas, ya que ambos comparten la misma memoria de fondos y al igual que los mapas, utilizarán los desplazamientos para gestionar esa memoria como si estuviera organizada en bloques, sólo que en este caso, de *tamaño 16KB*. La macro utilizada para especificar la dirección base es *BG_TILE_BASE(n)*, donde *n* toma valores entre 0 y 15. El número hace referencia al múltiplo de 16KB correspondiente, empezando en 0x0000, desplazándose en múltiplos de 0x4000 hasta el valor 0x3c000, tal y como se puede observar en la figura 7.3.

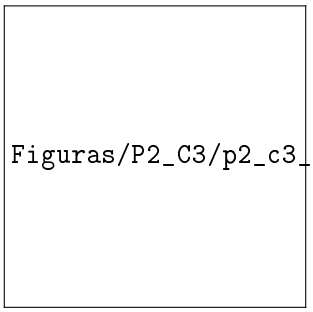

Figura 7.4: Cálculo del tamaño de un mapa de teselas.

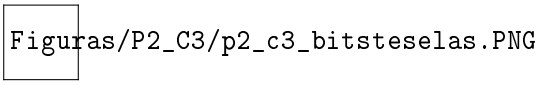

Figura 7.5: Entradas del mapa de teselas.

Bibliography