

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**

Copyright ©2018 Raúl Montoliu Colas, Juan Carlos Fernández Fernández, Maribel Castillo Catalán.

Esta obra se distribuye bajo la Licencia *Creative Commons Atribución-Compartir Igual 4.0 Internacional*. Puede consultar las condiciones de dicha licencia en: <http://creativecommons.org/licenses/by-sa/4.0/>



Índice general

1	Introducción a la consola Nintendo DS	5
1.1	Las videoconsolas de Nintendo	5
1.2	Hardware de la Nintendo DS (NDS)	6
2	Herramientas para programar la NDS	9
2.1	Herramientas de desarrollo para programar con la NDS	9
2.2	Instalación de <i>DekvitPro</i>	11
2.3	Instalación de los emuladores	11
2.4	Nuestro primer programa para NDS	12
3	Fundamentos para programar la NDS	17
3.1	Introducción a la programación en NDS	17
3.2	Salida de texto	18
3.3	Teclado	21
3.4	Botones de la consola	23
3.5	Pantalla táctil	25
3.6	Temporizador	27
4	Programación de un juego sin gráficos	31
4.1	Descripción del juego	31
4.2	Desarrollo del juego	32
5	Sistema de memoria gráfica de la NDS	37
5.1	Introducción	37

5.2	El registro <i>REG_POWERCNT</i>	38
5.3	El registro <i>REG_DISPCNT</i>	40
5.4	Los registros <i>VRAM_?_CR</i>	40
6	El modo framebuffer	43
6.1	El modo Framebuffer	43
6.2	Preparando el modo framebuffer	43
6.3	Mostrar imágenes	44
6.4	Dibujar píxeles	46
6.5	Ejercicios avanzados	48
7	El modo teselado	49
7.1	Introducción	49
7.2	Funcionamiento básico del modo teselado	49
7.3	Funcionamiento avanzado del modo teselado	58
7.4	Mostrar más de un fondo a la vez	60
7.5	Ejercicios avanzados	64
8	El sistema de Entrada/Salida	65
8.1	Introducción	65
8.2	Problemática de la E/S	65
8.3	E/S por consulta de estado	67
8.4	E/S por interrupciones	68
9	Entrada/Salida en la NDS	73
9.1	Introducción	73
9.2	Botones	73
9.3	Temporizadores	74
9.4	Gestión de interrupciones en el procesador	75
9.5	Entrada de datos utilizando los botones en la Nintendo DS	77
9.6	Ejercicios avanzados	81
10	Realización de un juego con gráficos	83
10.1	Descripción del juego	83
10.2	Desarrollo del juego	83
	Bibliography	89
	Books	89
	Articles	89

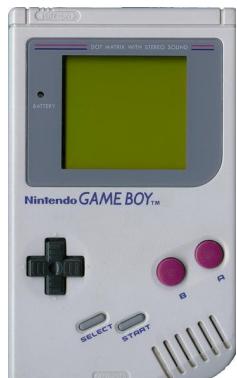
1. Introducción a la consola Nintendo DS

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*. *DevkitPro* 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 *NDSLlib*, es una biblioteca creada por Michael Noland y Jason Rogers. Esta biblioteca sirve como base para el desarrollo de programas para la Nintento 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 ejecutables *.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 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 de *DekvitPro*

Se accede a la página web:

<http://devkitpro.org/>

Se pulsa en *Click Here for instructions on installing the tools and getting started* y, según el sistema operativo, se siguen unas instrucciones u otras.

2.2.1 Instalación en Windows

Se debe elegir *Windows Installer/Updater package* y descargar la última versión (p. ej. *devkitProUpdater-3.0.3.exe*).

Una vez ha terminado la instalación se deben modificar unas variables de entorno. Por ejemplo, si la instalación se ha realizado en *c:\devkitPro*, habría que actualizar las siguientes variables:

DEVKITARM -> /c/devkitPro/devkitARM
DEVKITPRO -> /c/devkitPro/

y en la variable PATH añadir:

c:\devkitPro\devkitARM\bin

También sería conveniente 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

Finalmente, es recomendable renombrar las copias con los siguientes nombres:

- arm-eabi-as
- arm-eabi-g++
- arm-eabi-gcc
- arm-eabi-gdb
- arm-eabi-objcopy

2.2.2 Instalación en Linux

Pulsar en https://devkitpro.org/wiki/devkitPro_pacman y seguir los pasos explicados en esa página web.

Es recomendable 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

Posteriormente, hay que renombrar las copias con los siguientes nombres:

- arm-eabi-as
- arm-eabi-g++

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

2.3 Instalación de los emuladores

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

<https://windsprocentral.blogspot.com/>

Seleccionar *Descargar WinDS PRO* en Windows o *Descargar LinuxDS PRO* en Linux. Seguir las instrucciones.

2.4 Nuestro primer programa para NDS

En esta sección se van a ver los pasos para realizar nuestro primer programa para la NDS en los sistemas operativos *Windows* y *Linux*.

2.4.1 Desarrollar código en Windows

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) {
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 imprime 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-math
-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
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 exemplo.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 está desarrollando. Para borrar todos los ficheros y directorios creados durante la compilación se puede ejecutar *make clean*.

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 2.1.

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

2.4.2 Desarrollar código 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 2.1: Ejecución del programa ejemplo en el emulador *No\$gba*.

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
    - 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
/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 exemplo.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*.

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.

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:

```

1 #include<nds.h>
2 #include<stdio.h>
3 int main(void) {
4     consoleDemoInit();
5     iprintf("Me gusta programar videojuegos"); // Imprimir el mensaje
6     while(1)
7     {} // Bucle que no hace nada.
8 }
```

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:

```

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

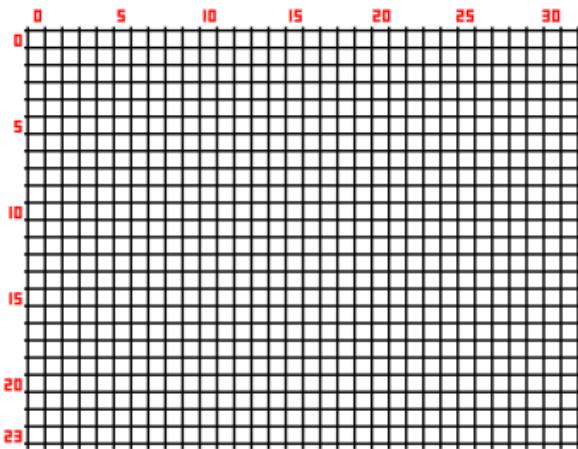


Figura 3.1: Pantalla de la NDS

```

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
```

```

16     true,           // Sistema grafico a usar (main system)
17     true);         // No cargar gráficos para la fuente
18
19 while(1) {
20     iprintf("\x1b[12;10HMensaje de texto");
21     swiWaitForVBlank(); // Esperar al refresco de pantalla
22 }
23 return 0;
24 }
```

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:

```

1 #include <nds.h>
2 #include <stdio.h>
3 int main(void)
4 {
5     PrintConsole pantalla_sup, pantalla_inf;
6     videoSetMode    (MODE_0_2D);
7     videoSetModeSub(MODE_0_2D);
8
9     consoleInit(&pantalla_sup,
10                 3,
11                 BgType_Text4bpp,
12                 BgSize_T_256x256,
13                 31,
14                 0,
15                 true,
16                 true);
17     consoleInit(&pantalla_inf,
18                 3,
19                 BgType_Text4bpp,
20                 BgSize_T_256x256,
21                 31,
```

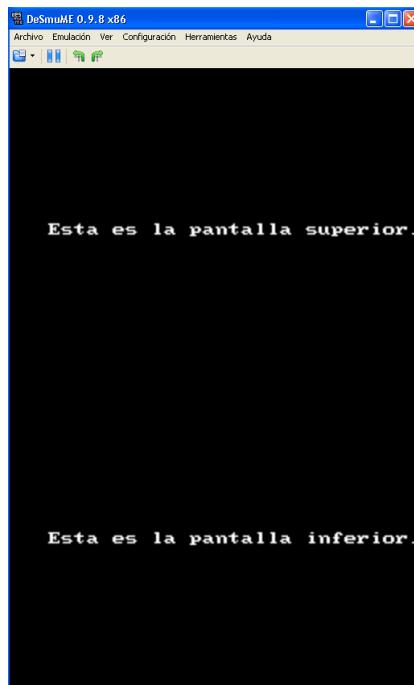


Figura 3.2: Resultado del programa *dos_pantallas.c*.

```

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:

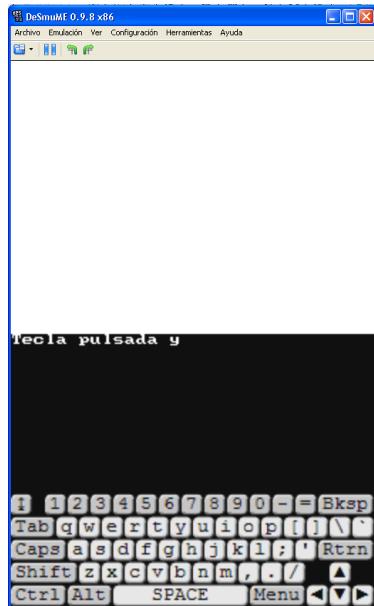


Figura 3.3: Salida del programa *teclado.c* en el emulador.

```

1 #include <nnds.h>
2 #include <stdio.h>
3 int main(void) {
4     int key; // Variable que almacena el código ascii de la tecla
5
6     consoleDemoInit();
7     keyboardDemoInit(); // Inicializa un teclado
8     keyboardShow(); // Visualiza el teclado
9
10    while(1) {
11        key = keyboardUpdate(); // Procesa la tecla pulsada
12                    // Retorna el código ascii
13                    // -1 si no se ha pulsado tecla
14
15        // Visualiza el carácter asociado al ascii de la tecla
16        if (key > 0) iprintf("Tecla pulsada %c \n", key);
17
18        swiWaitForVBlank();
19    }
20    return 0;
21 }
```

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. Despues 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:

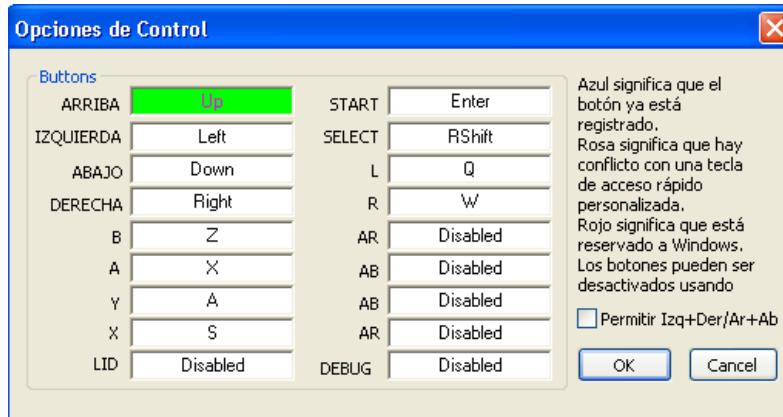


Figura 3.5: Relación que existe entre los botones de la NDS y el teclado.

- 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:

```

1 #include <n3ds.h>
2 #include <stdio.h>
3 int main()
4 {
5     u32 keys;
6     consoleDemoInit();
7
8     while(1){
9         scanKeys();
10        keys = keysCurrent(); // Se lee el estado actual de todos los botones
11
12        // Se comprueba si se ha pulsado arriba en la cruceta
13        if (keys & KEY_UP) iprintf("\x1b[10;5H Pulsado UP");
14        else                 iprintf("\x1b[10;5H           ");
15
16        // Se comprueba si se ha pulsado en el botón A
17        if (keys & KEY_A)    iprintf("\x1b[12;5H Pulsado A");
18        else                 iprintf("\x1b[12;5H           ");
19
20        // Se comprueba si se ha pulsado el botón Start
21        if (keys & KEY_START) iprintf("\x1b[15;5H Pulsado START");
22        else                  iprintf("\x1b[15;5H           ");
23
24        // Se comprueba si se ha pulsado la pantalla táctil
25        if (keys & KEY_TOUCH) iprintf("\x1b[13;5H Pantalla tactil");
26        else                  iprintf("\x1b[13;5H           ");
27
28        swiWaitForVBlank();
29 }
```

```

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
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.

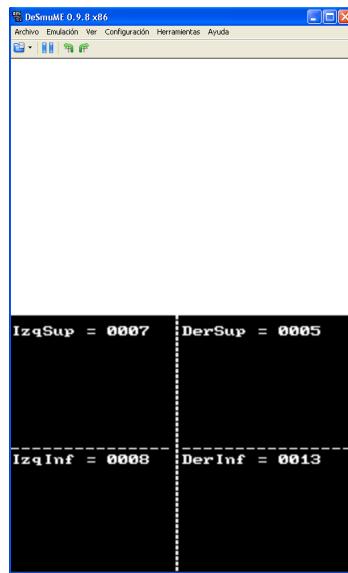


Figura 3.7: Salida ejemplo del programa requerido en la sección 3.5

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 botón *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 <nnds.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;%dH", 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 4.1¹ 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.

¹La 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:

```

1 #include <nds.h>
2 #include <stdio.h>
3
4 void MostrarCaballos(int posicion_caballos[]);
5 void MostrarBotones();
6 void MostrarDinero(int dinero);
7
8 int main(void)
9 {
10     PrintConsole pantalla_sup, pantalla_inf;
11     videoSetMode(MODE_0_2D);
12     videoSetModeSub(MODE_0_2D);
13     consoleInit(&pantalla_sup, 3, BgType_Text4bpp,
14                 BgSize_T_256x256, 31, 0, true, true);
15     consoleInit(&pantalla_inf, 3, BgType_Text4bpp,
16                 BgSize_T_256x256, 31, 0, false, true);
17
18     int posicion_caballos[4];
19     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 función *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:

```

1 ...
2     int velocidad_caballos[4];
3     for (int i=0; i<4; i++)
4         velocidad_caballos[i] = 1;
5 ...
6     int hay_ganador = 0;
7     while(1)
8     {
9         if (hay_ganador == 0)
10        {
11            int ganador = ComprobarGanador(posicion_caballos);
12            if (ganador >= 0)
13            {
14                hay_ganador = 1;
15                consoleSelect(&pantalla_inf);
16                iprintf("\x1b[5;1HEl caballo ganador es el: %d",ganador);
17            }
18            else
19            {
20                ActualizarPosicionCaballos(posicion_caballos, velocidad_caballos);
21                consoleSelect(&pantalla_sup);
22                BorrarPantalla();
23                MostrarCaballos(posicion_caballos);
24            }
25        }
26        consoleSelect(&pantalla_inf);
27        MostrarDinero(dinero);
28        MostrarBotones();
29    }
30 ...

```

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 función²:

■ **Ejemplo 4.3**

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

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

```
2 {  
3     return x0 + (x1 - x0) * rand() / ((double) RAND_MAX);  
4 }
```

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**

```
1 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 β 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**

```
1 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 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 qué 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

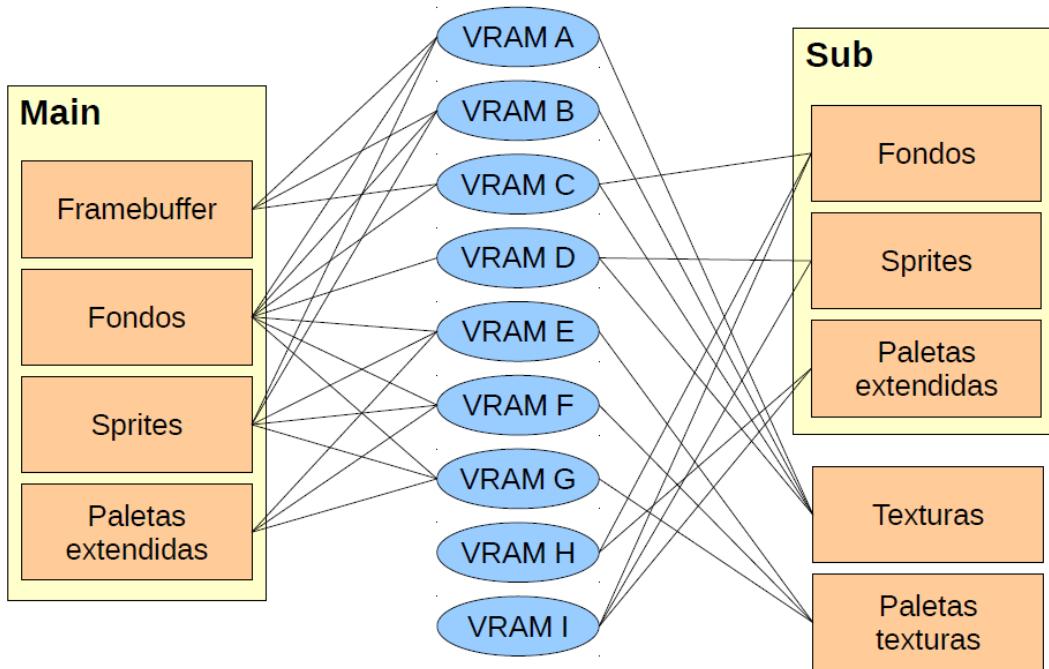


Figura 5.2: Bancos de memoria y que motores, y para que fin, los pueden usar.

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.

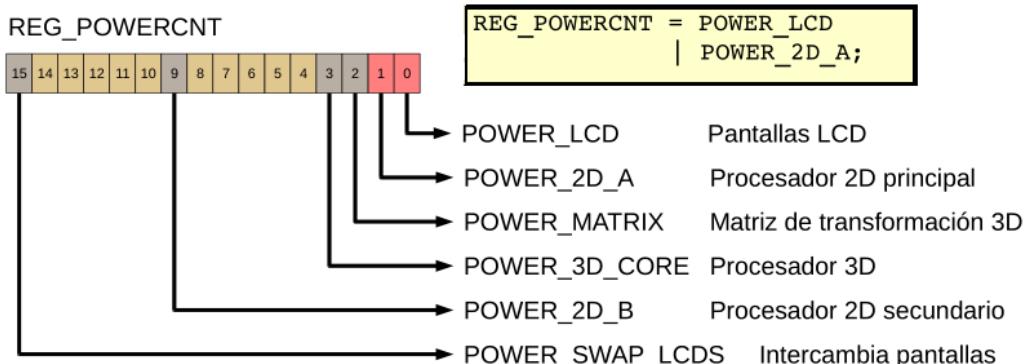


Figura 5.3: Significado de los bits del registro REG_POWERCNT

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.

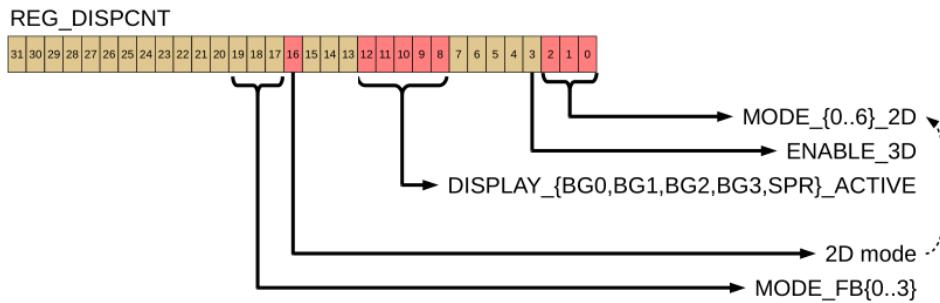


Figura 5.4: Significado de los bits del registro *REG_DISPCNT*

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* en el motor *main*, se debe usar la siguiente instrucción:

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

Para el motor secundario se usará la siguiente instrucción:

```
REG_DISPCNT_SUB = 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_ENABLE | VRAM_A_MAIN_BG;
```

Para el motor secundario se usará la siguiente instrucción:

```
VRAM_C_CR = VRAM_ENABLE | VRAM_C_SUB_BG;
```

En este caso se activa la *VRAM C*, puesto que el motor secundario no puede usar la *VRAM A* (ver figura 5.2).

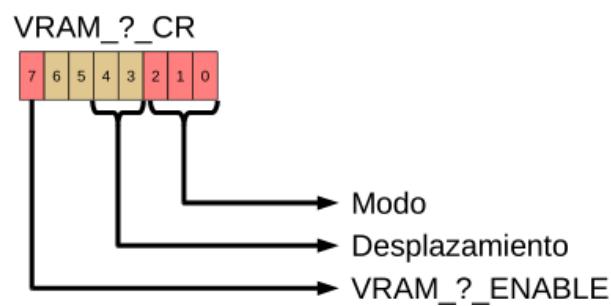


Figura 5.5: Significado de los bits del registro VRAM_?_CR

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. Además el modo Framebuffer solo está disponible para el motor principal.

La pantalla se compone de 49 152 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 *RGB15* 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 5.1 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.

6.2 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.

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

Ejercicio	Tiempo
6.1	20'
6.2	15'
6.3	25'
6.4	15'
6.5	15'
6.6	30'

En primer lugar hay que especificar el modo seleccionado, para lo cual se utiliza el registro REG_DISP_CNT y la macro que se corresponda con el modo a utilizar. Por ejemplo, para usar la configuración FB0 (y por lo tanto la zona de memoria VRAM_A) se deberá usar el siguiente código:

```
REG_DISP_CNT = 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_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.

6.3 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/tools/bin/grit.exe*¹. 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"
```

¹En versiones anteriores se encontraba en *devkitPro/devkitARM/bin/grit.exe*

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

- **Ejemplo 6.1** El siguiente programa (*unaimagen.c*) muestra la imagen *mario.png* en la pantalla inferior de la NDS:

```

1 #include <nds.h>
2 #include "mario.h"
3
4 int main (void)
{
5     REG_POWERCNT = POWER_LCD | POWER_2D_A;
6     REG_DISPCNT = MODE_FBO;
7     VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
8     dmaCopy(marioBitmap, VRAM_A, 256*192*2);
9
10    while (1)
11    {
12        swiWaitForVBlank();
13    }
14    return 0;
15 }
```

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.

Ejercicio 6.1 Crea un nuevo proyecto, usando como punto de partida el programa *unaimagen.c*, que muestre en la pantalla inferior una imagen de 256*192 píxeles. Recuerda que debes obtener los ficheros *.h* y *.c* de la imagen usando el comando *grit* y que dichos ficheros deben estar incluidos en el proyecto.

- **Ejemplo 6.2** En el siguiente programa (*dosimagenes.c*) 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:

```

1 #include <nds.h>
2 #include "mario.h"
3 #include "wallpaper.h"
4
5 int main (void)
{
6     REG_POWERCNT = POWER_LCD | POWER_2D_A;
7
8     VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
9     dmaCopy(marioBitmap, VRAM_A, 256*192*2);
10
11    VRAM_B_CR    = VRAM_ENABLE | VRAM_B_LCD;
12    dmaCopy(wallpaperBitmap, VRAM_B, 256*192*2);
13
14 // De inicio se muestra la que se encuentra en VRAM_A (FBO)
15    REG_DISPCNT = MODE_FBO;
16
17    while (1)
18    {
19        scanKeys();
20        int held=keysUp();
21
22        if (held & KEY_A)
23        {
```

```

24     REG_DISPCTR = MODE_FBO;
25 }
26 if (held & KEY_X)
27 {
28     REG_DISPCTR = MODE_FB1;
29 }
30
31 swiWaitForVBlank();
32 }
33 return 0;
34 }
```

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_DISPCTR. Los ficheros .s de ambas imágenes deben estar incluidos en el proyecto.

Ejercicio 6.2 Crea un nuevo proyecto, usando como punto de partida el programa *dosimages.c*, que muestre en la pantalla inferior una imagen de 256*192 píxeles. Cuando se pulse la tecla X se cambiará la imagen por otra. Si se pulsa de nuevo la tecla X, se volverá a cambiar la imagen mostrada y así sucesivamente. En total existirán 4 imágenes diferentes. ■

Ejercicio 6.3 Modifica el programa anterior para que el cambio de imagen a mostrar se realice cada un número determinado de segundos (por ejemplo cada 2 segundos). Al final del capítulo 3 hay un ejemplo que te puede servir de mucha ayuda. ■

6.4 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 deseé 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.

■ **Ejemplo 6.3** 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 (*degradado.c*) 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_FBO;
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.

Ejercicio 6.4 Modifica el programa anterior para que el degradado sea horizontal en vez de vertical. Para hacer el degradado se deberá ir incrementando la cantidad de azul a medida que se vaya desplazando hacia la derecha en la pantalla. Es decir, el nivel de color azul depende exclusivamente de la columna que estemos pintando. ■

■ **Ejemplo 6.4** El siguiente programa (*colores.c*) 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:

```

1 #include <nds.h>
2 #include <stdio.h>
3
4 int main (void)
5 {
6
7   REG_POWERCNT = POWER_LCD | POWER_2D_A;
8   VRAM_A_CR    = VRAM_ENABLE | VRAM_A_LCD;
9   VRAM_B_CR    = VRAM_ENABLE | VRAM_B_LCD;
10  VRAM_C_CR    = VRAM_ENABLE | VRAM_A_LCD;
11
12  u16 color_rojo  = RGB15(31,0,0);
13  u16 color_verde = RGB15(0,31,0);
14  u16 color_azul  = RGB15(0,0,31);
15
16  unsigned short *fbA = VRAM_A;
17  unsigned short *fbB = VRAM_B;
18  unsigned short *fbC = VRAM_C;
```

```

19 int lin,col;
20
21 for(lin = 0; lin < 192; lin++)
22     for(col = 0; col < 256; col++)
23         fbA[lin*256 + col] = color_rojo;
24
25 for(lin = 0; lin < 192; lin++)
26     for(col = 0; col < 256; col++)
27         fbB[lin*256 + col] = color_verde;
28
29 for(lin = 0; lin < 192; lin++)
30     for(col = 0; col < 256; col++)
31         fbC[lin*256 + col] = color_azul;
32
33 // de inicio FBO
34 REG_DISPCTR = MODE_FBO;
35 while (1)
36 {
37     scanKeys();
38     int held=keysHeld();
39     if (held & KEY_A)
40     {
41         REG_DISPCTR = MODE_FBO;
42     }
43     if (held & KEY_X)
44     {
45         REG_DISPCTR = MODE_FB1;
46     }
47     if (held & KEY_B)
48     {
49         REG_DISPCTR = MODE_FB2;
50     }
51     swiWaitForVBlank();
52 }
53 return 0;
54 }
```

Ejercicio 6.5 Realiza un programa que al pulsar las teclas A, X y B (de la NDS) muestre las banderas de tres países diferentes.

6.5 Ejercicios avanzados

Ejercicio 6.6 Realiza un programa que permita mover un cuadrado de 25x25 píxeles por la pantalla. Para ello se usarán los botones de las flechas de la NDS. Se deberá tener en cuenta los límites de la pantalla.

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 Introducción

El modo teselado permite muchas más posibilidades a la hora de realizar videjuegos con gráficos que el modo framebuffer.

7.2 Funcionamiento básico del modo teselado

En el modo teselado, la pantalla se divide en celdas 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:

```
1 u8 comecocos [64] =
2 {
3     2,2,1,1,1,1,2,2,
4     2,2,1,1,1,1,1,2,
5     2,2,2,1,1,1,1,1,
6     2,2,2,2,1,1,1,1,
7     2,2,2,2,1,1,1,1,
8     2,2,2,1,1,1,1,1,
9     2,2,1,1,1,1,1,2,
10    2,2,1,1,1,1,2,2,
11 };
12
13 u8 fantasma [64] =
14 {
15     2,2,2,3,3,2,2,2,
16     2,3,3,3,3,3,3,2,
```

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

Ejercicio	Tiempo
7.1	10'
7.2	10'
7.3	40'
7.4	15'
7.5	30'
7.6	15'

```

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);

```

Como se puede comprobar, la posición 1 de la paleta se refiere a un color rojo, la 2 a uno verde, la 3 a uno azul y la 4 al color blanco.

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):

```

1 u16 mapData[768] =
2 {
3     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,
4     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,
5     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,
6     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
7
8     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
9     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
10    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
11    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
12
13    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
14    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
15    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
16    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
17
18    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
19    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
20    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
21    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
22
23    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
24    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
25    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
26    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
27
28    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
29    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
30    2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,
31    2,2,2,2,2,2,2,2, 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). Además se mostrarán dos fantasmas, uno situado en la fila 1, columna 3, y el otro situado en la fila 2, columna 1. Las teselas que se muestran en cada posición se podrán modificar durante el transcurso del programa.

El mapa de teselas ocupará, en este caso, $768 * 2 = 1536$ 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 el ejemplo anterior se usaban 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.

Con respecto al modo framebuffer el ahorro de memoria es incluso mayor. En el modo framebuffer especificamos el color de cada uno de los 192x256 píxeles usando un número de 2 bytes. Por lo tanto, se necesitan $192 * 256 * 2 = 98\,302$ bytes.

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

```

1 REG_POWERCNT = POWER_ALL_2D;
2 REG_DISPCNT = MODE_0_2D | DISPLAY_BGO_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 del motor principal (ver Figura 5.1). 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 5.1). 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 teselas 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 // Copia del mapa de teselas a la memoria gráfica
9 pos_mapData = 0;
10 for(i=0; i<32; i++)
11     for(j=0; j<24; j++)
12     {
13         pos_mapMemory = i*24+j;
14         mapMemory[pos_mapMemory] = mData[pos_mapData];
15         pos_mapData++;
16     }
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 comeococos, seguida del fondo y por último el fantasma. Por lo tanto, en el mapa de teselas pondremos 1 cuando queramos mostrar el comeococos, 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.

■ **Ejemplo 7.1** A continuación se muestra el código completo del programa (*teselas1.c*) 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 comeccocos [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 u16 mapData[768] =
41 {
42     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, 2,2,2,2,2,2,2,2,2,
43     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,
44     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,
45     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
46
47     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
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,
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,
51
52     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
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,
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,
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,
56
57     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
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,
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,
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,
61
62     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
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,
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,
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,
66
67     2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,
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,
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,
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
71 };
72
73 int main(void)
74 {
75     int fila, columna, pos_mapMemory, pos_mapData;
76
77     REG_POWERCNT = POWER_ALL_2D;
78     REG_DISPCTL = MODE_0_2D | DISPLAY_BG0_ACTIVE ;
79     VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG ;
80     BGCTRL[0] = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE
81         (1);
82
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);

```

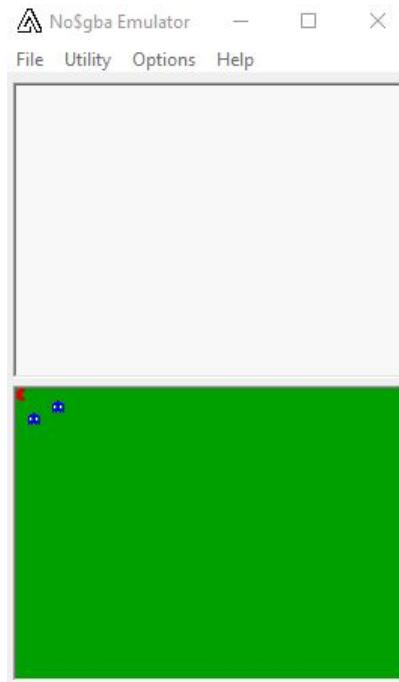


Figura 7.1: Salida del programa de esta sección en la NDS

```

88 static u16* mapMemory = (u16*) BG_MAP_RAM(0);
89
90 dmaCopy(comecocos, tileMemory + 64, sizeof(comecocos));
91 dmaCopy(fondo, tileMemory + 128, sizeof(fondo));
92 dmaCopy(fantasma, tileMemory + 192, sizeof(fantasma));
93
94 pos_mapData = 0;
95 for(fila=0;fila<24;fila++)
96     for(columna=0;columna<32;columna++)
97     {
98         pos_mapMemory = fila*32+columna;
99         mapMemory[pos_mapMemory] = mapa[pos_mapData];
100        pos_mapData++;
101    }
102
103 while(1)
104 {
105     swiWaitForVBlank();
106 }
107 }
```

Ejercicio 7.1 Crea un proyecto con el código del ejemplo anterior y comprueba que funciona correctamente.

Ejercicio 7.2 A partir del ejercicio anterior, modifica el mapa de teselas para mostrar 5 come-
cocos y 10 fantasmas repartidos por la pantalla.

El programa anterior también se podría haber realizado sin tener que haber creado el mapa de teselas inicial. Para ello escribimos directamente el número de tesela en la posición de memoria

que nos interese.

■ **Ejemplo 7.2** El siguiente programa (*teselas2.c*) muestra un ejemplo:

```

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 int main(void)
41 {
42     int i, fila, columna;
43
44     REG_POWERCNT = POWER_ALL_2D;
45     REG_DISPCTRL = MODE_0_2D | DISPLAY_BGO_ACTIVE ;
46     VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG ;
47     BGCTRL[0] = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE
        (1);
48
49     BG_PALETTE[1]=RGB15(28,0,0);
50     BG_PALETTE[2]=RGB15(0,20,0);
51     BG_PALETTE[3]=RGB15(0,0,31);
52     BG_PALETTE[4]=RGB15(31,31,31);
53
54     static u8* tileMemory = (u8*) BG_TILE_RAM(1);
55     static u16* mapMemory = (u16*) BG_MAP_RAM(0);
56

```

```

57 dmaCopy(comecocos, tileMemory + 64, sizeof(comecocos));
58 dmaCopy(fondo, tileMemory + 128, sizeof(fondo));
59 dmaCopy(fantasma, tileMemory + 192, sizeof(fantasma));
60
61 // fondo
62 for(i=0;i<32*24;i++)
63     mapMemory[i] = 2;
64
65 // comecocos en la fila 0, columna 0
66 fila = 0; columna = 0;
67 mapMemory[fila*32+columna] = 1;
68
69 // fantasma en la fila 1, columna 3
70 fila = 1; columna = 3;
71 mapMemory[fila*32+columna] = 3;
72
73 // fantasma en la fila 2, columna 1
74 fila = 2; columna = 1;
75 mapMemory[fila*32+columna] = 3;
76
77 while(1)
78 {
79     swiWaitForVBlank();
80 }
81 }
```

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.

■ **Ejemplo 7.3** Por ejemplo, el siguiente fragmento de código (el programa completo es *teselas3.c*) mostrará al pulsar la tecla A de la NDS, un segundo comecocos en la celda situada en la fila 15 y columna 16:

```

1 ...
2 while(1)
3 {
4     scanKeys();
5     int key = keysDown();
6
7     if (key & KEY_A)
8     {
9         fila = 15;
10        columna = 16;
11        pos_mapMemory = fila*32+columna;
12        mapMemory[pos_mapMemory] = 1;
13    }
14    swiWaitForVBlank();
15 }
```

Ejercicio 7.3 Realiza un programa que mueva el comecocos por la pantalla usando para ello los botones de las flechas de la NDS. Debes tener en cuenta los límites de la pantalla. ■

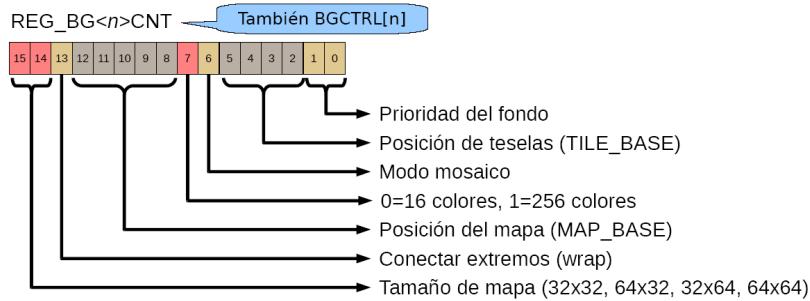


Figura 7.2: Significados de los bits del registro *REG_BGnCNT*.

7.3 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.

7.3.1 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 32×32 teselas será una sucesión de 32×32 entradas de 16 bits, por lo que el tamaño total será de $2 \times 32 \times 32 = 2KB$. Es decir, en este caso el mapa entero cabe antes de la siguiente dirección base de otro mapa.
- Sin embargo, para un fondo de 64×64 teselas, los datos del mapa requieren $2 \times 64 \times 64 = 8KB$, es decir, son necesarios cuatro bloques completos de 8KB.

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

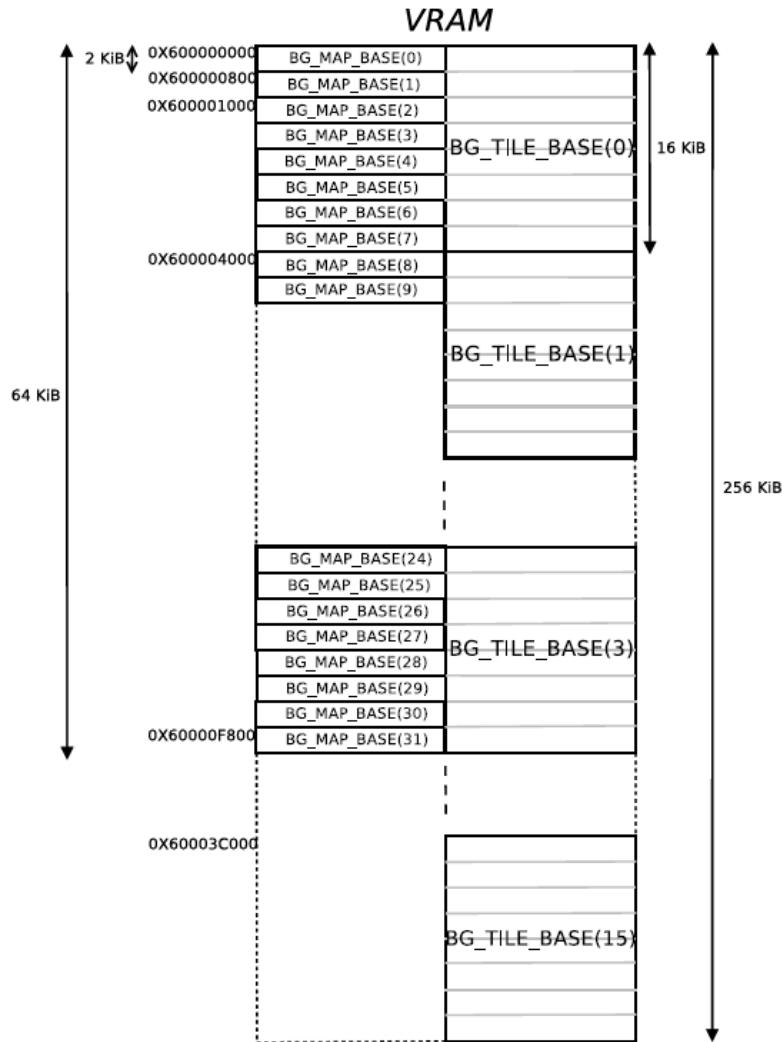


Figura 7.3: Desplazamientos en la VRAM.

horizontal, vertical o ambas. Finalmente, para teselas que utilizan paletas de 16 colores, se utilizan los 4 bits más significativos para identificar la paleta de colores utilizada por esa tesela. La Figura 7.5 muestra dicha distribución.

7.3.2 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.

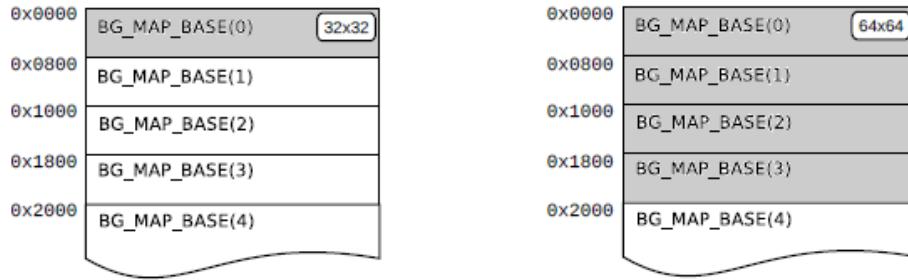


Figura 7.4: Cálculo del tamaño de un mapa de teselas.

Bits	15 14 13 12	11	10	9 8 7 6 5 4 3 2 1 0
Propósito	Paleta	Vertical	Horizontal	Índice

Figura 7.5: Entradas del mapa de teselas.

7.4 Mostrar más de un fondo a la vez

7.4.1 Ambos motores a la vez

Para configurar el motor secundario se usarán instrucciones muy similares a las usadas para el motor principal. La principal diferencia es el uso del sufijo *SUB* o la sustitución de *MAIN* por *SUB*.

■ **Ejemplo 7.4** El siguiente programa (*dosmotores.c*) muestra una composición de teselas en cada pantalla, la pantalla inferior usará el motor principal y la superior el secundario:

```

1 #include <n3ds.h>
2 #include <stdio.h>
3
4 u8 verde[64] =
5 {
6     1,1,1,1,1,1,1,1,
7     1,1,1,1,1,1,1,1,
8     1,1,1,1,1,1,1,1,
9     1,1,1,1,1,1,1,1,
10    1,1,1,1,1,1,1,1,
11    1,1,1,1,1,1,1,1,
12    1,1,1,1,1,1,1,1,
13    1,1,1,1,1,1,1,1,
14} ;
15
16 u8 rojo[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 int main( void )
29 {
30     REG_POWERCNT = POWER_ALL_2D;
31     REG_DISPCNT = MODE_0_2D | DISPLAY_BGO_ACTIVE;

```

```

32 REG_DISPCTRL = MODE_0_2D | DISPLAY_BG0_ACTIVE;
33 VRAM_A_CR     = VRAM_ENABLE | VRAM_A_MAIN_BG;
34 VRAM_C_CR     = VRAM_ENABLE | VRAM_C_SUB_BG;
35 BGCTRL[0]      = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);
36 BGCTRL_SUB[0] = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);
37
38 static u8* tileMemory    = (u8*) BG_TILE_RAM(1);
39 static u16* mapMemory    = (u16*) BG_MAP_RAM(0);
40 static u8* tileMemorySub = (u8*) BG_TILE_RAM_SUB(1);
41 static u16* mapMemorySub = (u16*) BG_MAP_RAM_SUB(0);
42
43 BG_PALETTE[1]   = RGB15(0,20,0);
44 BG_PALETTE[2]   = RGB15(20,0,0);
45 BG_PALETTE_SUB[1] = RGB15(0,20,0);
46 BG_PALETTE_SUB[2] = RGB15(20,0,0);
47
48 dmaCopy(verde, tileMemory + 64, sizeof(verde));
49 dmaCopy(rojo, tileMemorySub + 64, sizeof(rojo));
50
51 int fila, columna;
52
53 for(fila=0;fila<24;fila++)
54     for(columna=0;columna<32;columna++)
55     {
56         mapMemory [fila*32+columna] = 1;
57         mapMemorySub [fila*32+columna] = 1;
58     }
59
60 while(1)
61 {
62     swiWaitForVBlank();
63 }
64 }
```

Como se puede comprobar, se configura el fondo 0 de cada uno de los motores para tener un mapa de teselas de 32×32 teselas. El motor principal almacenará el mapa de teselas y las teselas en la VRAM_A. El motor secundario lo hará en la VRAM_C. Hay que fijarse que aunque las macros usadas para indicar en qué parte de la memoria se van a almacenar el mapa de teselas y las teselas son iguales, en realidad no se refieren a las mismas posiciones de memoria, pues cada fondo usa una memoria VRAM diferente.

Cada fondo almacena únicamente la tesela que va usar (líneas 48 y 49). Así para el fondo 0, la tesela en la posición 1 es la tesela *verde*. Sin embargo, para el fondo 1, en la posición 1 se encontrará la tesela *rojo*.

Para ambos fondos, se rellena la pantalla completa con la tesela situada en primer lugar. El resultado final se muestra en la Figura 7.6.

Ejercicio 7.4 Partiendo del programa anterior realiza las modificaciones que sean necesarias para que al pulsar la tecla A de la NDS, se intercambien los colores mostrados en las pantallas. Al pulsar la tecla X se volverá a la situación inicial.

7.4.2 Superponer fondos en la misma pantalla

■ **Ejemplo 7.5** El siguiente programa (*dosfondos.c*) muestra como superponer dos fondos en la misma pantalla:

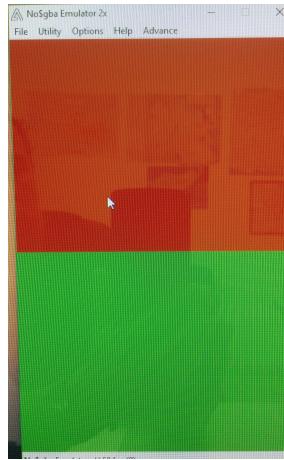
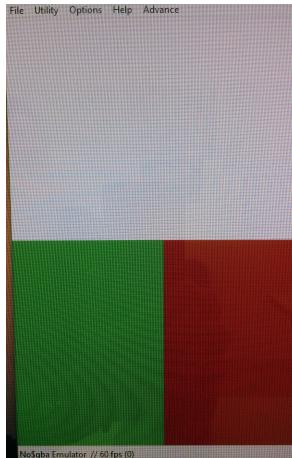


Figura 7.6: Resultado del programa *dosmotores.c*

```

1 #include <nds.h>
2 #include <stdio.h>
3
4 u8 verde[64] =
5 {
6     1,1,1,1,1,1,1,1,
7     1,1,1,1,1,1,1,1,
8     1,1,1,1,1,1,1,1,
9     1,1,1,1,1,1,1,1,
10    1,1,1,1,1,1,1,1,
11    1,1,1,1,1,1,1,1,
12    1,1,1,1,1,1,1,1,
13    1,1,1,1,1,1,1,1,
14 };
15
16 u8 rojo[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 int main( void )
29 {
30     REG_POWERCNT = POWER_ALL_2D;
31     REG_DISPCNT = MODE_0_2D | DISPLAY_BGO_ACTIVE | DISPLAY_BG1_ACTIVE;
32     VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;
33     BGCTRL[0] = BG_32x32 | BG_COLOR_256 |
34             BG_MAP_BASE(0) | BG_TILE_BASE(1) | BG_PRIORITY(1);
35     BGCTRL[1] = BG_32x32 | BG_COLOR_256 |
36             BG_MAP_BASE(1) | BG_TILE_BASE(2) | BG_PRIORITY(0);
37
38     static u8* tileMemoryB0 = (u8*) BG_TILE_RAM(1);
39     static u8* tileMemoryB1 = (u8*) BG_TILE_RAM(2);
40
41     static u16* mapMemoryB0 = (u16*) BG_MAP_RAM(0);

```

Figura 7.7: Resultado del programa *dosfondos.c*

```

42 static u16* mapMemoryB1 = (u16*) BG_MAP_RAM(1);
43
44 BG_PALETTE[1] = RGB15(0,20,0);
45 BG_PALETTE[2] = RGB15(20,0,0);
46
47 dmaCopy(verde, tileMemoryB0 + 64, sizeof(verde));
48 dmaCopy(rojo, tileMemoryB1 + 64, sizeof(rojo));
49
50 int fila, columna;
51
52 for(fila=0;fila<24;fila++)
53     for(columna=0;columna<32;columna++)
54     {
55         mapMemoryB0[fila*32+columna] = 1;
56         if (columna>16)
57             mapMemoryB1[fila*32+columna] = 1;
58     }
59
60 while(1)
61 {
62     swiWaitForVBlank();
63 }
64 }
```

Tal como muestra el código, se han configurado dos fondos para el motor principal. El primero de ellos (fondo 0) estará abajo por tener menos prioridad. El segundo (fondo 1) se mostrará por encima del fondo anterior. La prioridad se establece con la macro *BG_PRIORITY(N)*, donde *N* es la prioridad y admite valores de 0 a 3. Número más bajos de *N* indican más prioridad (más arriba en el orden de visualización). Ambos fondos se almacenarán en la VRAM_A, situando los mapas de teselas y las teselas en zonas no coincidentes. El primer fondo se pinta completamente con la tesela de color verde. El segundo fondo pinta únicamente las columnas de la derecha con las tesela de color rojo, permitiendo que en la parte izquierda se vea el primer fondo. El resultado final se muestra en la Figura 7.7.

Ejercicio 7.5 Realiza un programa que muestre en la pantalla inferior cuatro fondos diferentes. De más profundo a mas exterior los colores de cada uno de los fondos serán verde, rojo, azul y

amarillo. Deberás pintar las teselas de forma que se vea claro el orden de los fondos.



7.5 Ejercicios avanzados

Ejercicio 7.6 Realiza un programa que muestre en cada pantalla cuatro fondos diferentes.



8. El sistema de Entrada/Salida

Este capítulo trata sobre el sistema de entrada y salida que todos los computadores necesitan tener para poder interactuar con los usuarios.

Para profundizar más en el tema, se recomienda la consulta del capítulo 8 del libro *Introducción a la arquitectura de computadores con QtARMSim y Arduino* de Sergio Barrachina Mir et al.

8.1 Introducción

Las videoconsolas actuales se pueden considerar computadores cuya principal función es ejecutar juegos. Todos ellos disponen de una amplia variedad de dispositivos, llamados periféricos, que permiten la comunicación con el usuario. Esta comunicación entre el computador y el periférico se lleva a cabo mediante operaciones de Entrada/Salida (E/S). El objetivo de este capítulo es el análisis de cómo se llevan a cabo este tipo de operaciones.

8.2 Problemática de la E/S

Las videoconsolas, y los computadores en general, no funcionan de forma aislada. Deben relacionarse e interactuar con el entorno que les rodea, ya sea con sus usuarios, con otros computadores o con otros aparatos electrónicos. Los dispositivos que permiten esta comunicación reciben el nombre de **periféricos**. Hay una amplia variedad y se pueden clasificar según la función que realizan:

- Periféricos de salida: sirven para que el computador presente información o lleve a cabo determinadas acciones. Por ejemplo, los monitores o pantallas y las impresoras.
- Periféricos de entrada: sirven para introducir información en el computador, como los teclados y los escáneres.
- Periféricos de E/S: permiten que la información fluya en los dos sentidos. Destacan los discos duros y las pantallas táctiles, estas muy utilizadas en las videoconsolas.

Cada uno de estos periféricos presenta determinados requerimientos de comunicación. Por ejemplo, un teclado permite introducir caracteres en un computador y debe enviar la información de qué teclas se están pulsando. Sin embargo, la velocidad con la que debe transmitir dicha información

no tiene que ser demasiado alta. Al fin y al cabo, el teclado estará siendo utilizado por una persona, que en el mejor de los casos utilizará más de dos dedos y tecleará sin mirar el teclado. Por otro lado, un disco duro permite almacenar y recuperar información digital: documentos, programas, juegos, información de las jugadas, etc. Pero en el caso de un disco duro, la velocidad con la que se transmite la información es crucial. De hecho, será tanto mejor cuanto más rápido permita la lectura o escritura de la información.

Los periféricos, además de por las funciones que realizan y de los requerimientos de comunicación, también se diferencian por la tecnología que emplean, es decir, por la forma en la que llevan a cabo su cometido. Es distinta la tecnología utilizada en un disco duro que en una impresora. Incluso, en el caso de las impresoras por ejemplo, se pueden clasificar en: térmicas, de impacto, de inyección de tinta, láseres, por sublimación, etc. Cada una de ellas utiliza una tecnología distinta para llevar a cabo el mismo fin: presentar sobre papel, u otro tipo de material, una determinada información.

8.2.1 Controlador E/S

Toda esta variedad de funciones, tecnologías y necesidades de comunicación hace necesaria la existencia de hardware específico que se encargue de la comunicación entre el procesador y los periféricos. Este hardware recibe el nombre de **controlador de E/S**. Este dispositivo se conecta, por un lado al bus del sistema y por el otro al dispositivo periférico. El procesador para realizar una operación de E/S con el periférico se comunica con su controlador. Los controladores ocultan al procesador las dificultades propias de la gestión de los diferentes dispositivos y proporcionan una interfaz uniforme para realizar este tipo de operaciones. Por ejemplo, en el caso de los discos duros, cada fabricante utiliza una determinada tecnología para la fabricación de sus productos. Cada disco duro incorpora un controlador de E/S que conoce las peculiaridades de funcionamiento de dicho disco duro en concreto. Pero además, dicho controlador de E/S cumple con una serie de normas que son comunes a todos los controladores de E/S para discos duros y que permiten la comunicación del disco duro con el procesador. Un controlador de E/S puede gestionar a uno o varios periféricos simultáneamente.

Los controladores disponen de tres registros básicos internos para llevar a cabo su función:

- **Registro de control:** Almacena órdenes que indican cómo se va a realizar la operación de E/S.
- **Registro de estado:** Contiene el estado del periférico y de la operación de E/S que se acaba de realizar.
- **Registro de datos:** Este registro también recibe el nombre de **puerto de entrada o salida**. Almacena el dato que se va a intercambiar con el periférico:
 - En una operación de entrada, el periférico envía el dato al controlador de E/S, se almacena en este registro y, finalmente, el procesador lo lee.
 - En una operación de salida, el procesador escribe el dato en este registro y, desde él, se enviará al periférico.

8.2.2 Mapa de direcciones de la E/S

Cada registro se identifica en el sistema por una dirección. El conjunto de todas estas direcciones forma lo que se conoce como **mapa de direcciones de E/S o espacio de direcciones de E/S**. El procesador, cuando necesite leer o escribir información sobre alguno de estos registros utilizará su dirección para seleccionarlo como ocurre con la memoria.

Este mapa de direcciones pueden formar parte de dos espacios de direccionamiento distintos:

- Espacio de direcciones de memoria: **E/S mapeado en memoria**. En el sistema existe un mapa de direcciones único, una parte (superior o inferior) se asigna a la memoria y el resto a la E/S. El procesador lee o escribe en estos registros de la misma forma como lo hace en

cualquier otra parte de la memoria, utilizando las instrucciones de acceso a memoria (de lectura o escritura en memoria) proporcionando la dirección correspondiente del registro accedido. Esta es la modalidad utilizada por ARM (procesador empleado en la Nintendo DS).

- Espacio de direcciones de E/S: **E/S aislado.** Existen dos espacios de direccionamiento distintos, uno para memoria y otro para E/S. El procesador lee o escribe en los registros utilizando instrucciones especiales para acceder a los puertos de E/S. Esta es la modalidad utilizada por los procesadores basados en la arquitectura x86.

8.2.3 Operaciones de E/S

Una operación de E/S requiere una secuencia de operaciones que se detallan a continuación:

1. **Comprobación del estado del periférico:** es necesario conocer cuando está disponible para realizar la operación de E/S. Esto se realiza leyendo el registro de estado.
2. **Envío de las órdenes al registro de control:** hay que escribir la secuencia de palabras de control necesarias para llevar a cabo la operación de E/S en el periférico. Esto requerirá una o varias operaciones de escritura en el registro de control.
3. **Intercambio de información con el controlador.** Esto se realiza leyendo (entrada) o escribiendo (salida) en el registro de datos del controlador de E/S, para recoger o enviar el dato, respectivamente.
4. **Comprobación del estado del periférico:** para conocer si la operación se ha desarrollado correctamente. De nuevo se realiza una operación de lectura sobre el registro de estado.

Estos pasos se recogen en lo que se llamará programa de E/S. Este consistirá en una secuencia de instrucciones que leerán o escribirán sobre los registros del controlador.

En esta secuencia de operaciones destacan dos aspectos básicos:

- Sincronización entre el procesador y el periférico para que se lleve a cabo el intercambio de información. Los dos dispositivos trabajan a velocidades muy distintas y tiene que utilizarse alguna técnica para conocer en qué instante se puede llevar a cabo este intercambio. Se utilizan básicamente dos técnicas:
 - Consulta de estado. Comprobación periódica del estado del periférico para determinar la disponibilidad.
 - Interrupciones. El propio periférico avisa cuando está disponible.
- Transferencia de datos entre el procesador y el dispositivo. Este intercambio se lleva a cabo cuando los datos estén disponibles y se puede realizar entre el dispositivo y la memoria o el procesador. Da lugar a dos técnicas:
 - Utilizando instrucciones de transferencia de datos. Se utilizan las instrucciones que tiene el procesador para transferir información entre registros y memoria o puertos de E/S.
 - Por acceso directo a memoria. Permite realizar transferencias de bloques de información entre la memoria y el dispositivo sin que intervenga el procesador. Se dispone de un controlador que se programa para que realice la transferencia.

Todo esto da lugar a diferentes formas de llevar a cabo las operaciones de E/S que se detallan en los apartados siguientes.

8.3 E/S por consulta de estado

Con esta técnica, la sincronización entre el procesador y el periférico se realiza haciendo esperar al procesador hasta que se pueda realizar el intercambio de información entre ellos.

Esta espera se implementa en el propio programa de E/S mediante un bucle que se repetirá hasta que cambie el estado del periférico indicando que el intercambio se puede hacer. Cada iteración

del bucle lee el registro de estado, de forma que si no está preparado para hacer la transferencia continuará con la siguiente iteración y, si está preparado, finaliza el bucle y ejecuta las instrucciones que realizan la transferencia. Un ejemplo de implementación se muestra a continuación:

```

Leer estado periférico
while (! preparado)
{
    Leer estado periférico
}
Enviar datos a/desde el periférico

```

En el registro de estado hay un bit (llamado bit de *ready o preparado*) que cambia de valor (de 0 a 1 o al revés) cuando se almacena/libera la información contenida en el registro de datos, indicando que ha llegado un dato desde el periférico o se ha enviado el dato hacia el periférico. La condición del bucle analiza este bit y determina si continua la espera o no (se ejecutan las instrucciones del bucle o no). Como puede apreciarse en la implementación no requiere ningún hardware especial. La sincronización es la espera que proporciona el propio bucle.

8.4 E/S por interrupciones

Una gestión más eficiente de la E/S conlleva que sean los propios dispositivos los que avisen al procesador de cuándo se puede hacer el intercambio de información entre ellos. Este aviso consiste en que el controlador solicita lo que se conoce como **una interrupción** al procesador. El procesador, si lo tiene permitido, detiene lo que está procesando, atiende la interrupción solicitada, lleva a cabo la transferencia de información y continúa con el procesamiento interrumpido. De esta forma, el procesador no malgasta tiempo en la espera y permite compaginar eficientemente las diferentes velocidades de funcionamiento de los distintos periféricos y del procesador. Mediante esta técnica el procesador y los periféricos pueden trabajar de forma paralela, es decir, mientras un determinado periférico está realizando una operación de E/S, el procesador puede estar ejecutando instrucciones que resuelven otra tarea distinta o atendiendo a otro periférico.

Esta técnica implica realizar los siguientes pasos para realizar una operación de E/S:

1. Inicio de la operación: El programa que contiene la operación de E/S envía todos los comandos necesarios al controlador para que programe la operación en el periférico.
2. El controlador ejecuta la operación sobre el periférico y cuando está preparado para realizar el intercambio de información con el procesador le avisa, solicitando una interrupción.
3. El procesador detecta la interrupción y la procesa. El procesamiento consiste en ejecutar otro programa (rutina de tratamiento de la interrupción) que llevará a cabo el intercambio de información.

Como ya se ha dicho anteriormente, lo más interesante de esta técnica es que entre el paso 1) y 2), el procesador no tiene que esperar, puede estar ejecutando un programa distinto.

Esta técnica implica que el procesador debe disponer de hardware y software para que se puedan realizar y atender las peticiones:

- Hardware
 - Señal de control que forma parte del bus de control, llamada en ocasiones **IRQ (Interruption ReQuest)**, que será la que activarán los periféricos para solicitar la interrupción. Se activa desde el periférico y la recibe el procesador.
 - Señal de control que forma parte del bus de control, llamada normalmente **IACK (Interruption ACKnowlegde)**, que será la que el procesador activará cuando detecta que hay una interrupción pendiente y la va a atender. Se activa desde el procesador y la recibe el periférico.

- Circuitos de control para llevar a cabo el procesamiento de una interrupción.
 - Software
 - Rutina que contiene las instrucciones necesarias para atender la interrupción solicitada.
Por ejemplo:
 - Si un periférico de entrada solicita una interrupción: se tiene que leer el dato introducido que está en el registro de datos.
 - Si se trata de un periférico de salida: habrá que escribir un nuevo dato en el registro de datos para enviarlo al exterior.
- Este software recibe el nombre de **Rutina de tratamiento de la interrupción (RTI)**. La RTI tiene que estar almacenada en memoria como cualquier programa y el procesador tiene que conocer en qué dirección está para que pueda ejecutarla.

8.4.1 Procesamiento de una interrupción

A continuación se resumen las fases en los que se divide el procesamiento de una interrupción desde que el periférico la solicita hasta que el procesador la atiende y continúa con el programa que ha interrumpido para atenderla.

1. El periférico inicia todo el proceso cuando está preparado para realizar un intercambio de información con el procesador **activando la señal IRQ**:
 - Operación de entrada: en el registro de datos se ha cargado un nuevo dato desde el periférico.
 - Operación de salida: se ha liberado el registro de datos con el envío al periférico del dato que contenía.
2. Esta petición habrá llegado al procesador mientras éste se encuentra ejecutando instrucciones y su modo de funcionamiento hace que se realicen las siguientes operaciones:
 - 2.1 Finaliza el ciclo de una instrucción.** El que se estaba ejecutando mientras se ha recibido la petición de interrupción.
 - 2.2 Comprueba el estado de las interrupciones.** Comprueba si hay alguna interrupción pendiente por parte de alguno de los dispositivos conectados a las líneas IRQ. En el registro de estado del procesador se almacena información sobre qué interrupciones puede atender y cuáles no. Detectada una petición comprueba si tiene permiso para atenderla. Así, según tenga o no permiso, lleva a cabo operaciones distintas:
 - No tiene permiso: continúa con la ejecución de la siguiente instrucción. Hace caso omiso de la petición.
 - Sí que tiene permiso: Se inicia lo que se conoce como **procesamiento de una interrupción** y cuyas acciones se detallan a continuación.
 - 2.3 Almacenamiento del estado actual del procesador.** Salva el PC (contador de programa) y el registro de estado del procesador en registros internos o en la pila del sistema.
 - 2.4 Obtención de la dirección de la RTI.** La RTI, como cualquier otro programa, está almacenada en memoria y el procesador tiene que conseguir su dirección de inicio para ejecutarla. Hay diferentes técnicas:
 - Dirección fija. El procesador genera dicha dirección y la carga en el PC.
 - La dirección se la proporciona el periférico. El procesador activa la señal **IACK**, que la recibe el periférico y éste responde enviando la dirección a través del bus de datos. El procesador la recoge y la carga en el PC.
 - A partir de este momento la ejecución de instrucciones por parte del procesador continuará desde esta dirección.
 - 2.5 Ejecución de la RTI.** La primera tarea de esta rutina es salvar el resto de registros del procesador. A continuación, lleva a cabo las acciones oportunas para tratar el evento que provocó la interrupción. Por ejemplo, si la interrupción es por una nueva entrada

de datos, entonces la gestión consistirá en leer el registro de datos del periférico y almacenar la información en otro registro del procesador o en la memoria. La rutina finaliza su ejecución recuperando todo el estado del procesador, entre ellos el PC que permite continuar con el programa interrumpido.

2.6 Continúa la ejecución del programa detenido.

Ejecutando la siguiente instrucción.

Esta descripción indica que la ejecución de la RTI durante el procesamiento de una interrupción es como un salto a una subrutina. La principal diferencia es, que en la gestión de las interrupciones, el salto no se conoce cuando se va a producir. Dicho salto se producirá cuando el procesador detecte la activación de la línea IRQ y, entonces, tiene que averiguar la dirección de la RTI para ejecutarla.

8.4.2 Enmascaramiento de las interrupciones

La gestión de las operaciones de E/S en los dispositivos periféricos, en la mayoría de ellos, es algo que se puede programar a través de su registro de control. El registro de control dispone de uno o varios bits para las interrupciones y fijan si las puede solicitar o no. Si el dispositivo puede solicitar interrupción se dice que está **habilitada**. Si no puede, se dice que está **enmascarada**. Esto determina cómo se llevarán a cabo las operaciones de E/S. Si las tiene enmascaradas su gestión se realizará por consulta de estado.

Independientemente de como tenga el periférico configurada la interrupción, el procesador también puede fijar su estado. Es decir, las puede tener habilitadas o enmascaradas. Si el **procesador** tiene la interrupción **enmascarada** significa que aunque el periférico active dicha línea, el procesador no atenderá ninguna de las peticiones que llegan a través de ella. Durante la etapa de procesamiento de una interrupción, el procesador comprueba el estado de la interrupción, si está enmascarada no continúa con el procesamiento. A este nivel, el enmascaramiento/habilitación de una interrupción se realiza poniendo determinados bits del registro de estado del procesador a un valor prefijado.

8.4.3 Identificación de la interrupción

Un aspecto importante relacionado con la gestión de las interrupciones es averiguar qué dispositivo ha solicitado la interrupción. La problemática surge cuando hay varios que están conectados a la misma línea **IRQ** y cualquiera de ellos pueda activarla (solicitar interrupción). La identificación se puede realizar utilizando dos técnicas distintas:

Por consulta, también llamadas no vectorizadas: Durante el procesamiento de una interrupción, el procesador genera una dirección fija que se carga en el PC y genera el salto a la RTI. En esta rutina (rutina general) se comprueba qué periférico ha hecho la solicitud. Esta información se puede extraer del registro de estado del controlador de E/S, puesto que tiene un conjunto de bits que indican el estado de la petición (activa/no activa). Estos bits cambian automáticamente cuando se activa la línea y vuelven al estado inicial una vez atendida. Cuando se identifica el dispositivo, se salta a su RTI mediante una instrucción de salto a la dirección correspondiente.

Si hay peticiones simultáneas, las interrupciones se atienden en el orden en el que se hace la lectura de los registros de estado para determinar quién ha interrumpido.

Vectorizadas: Durante el procesamiento de una interrupción, el procesador activa la línea (IACK). Cuando un dispositivo recibe esta señal envía por el bus de datos la dirección de la RTI. Esta información la recibe el procesador y la carga en el PC.

La línea IACK está conectada a todos los periféricos que realizan la petición a través de la misma IRQ. La conexión es en cadena. De forma que cuando el procesador activa esta línea, llega al primer dispositivo de la cadena, si él ha realizado la petición envía la dirección. Si no es así, envía IACK al siguiente dispositivo de la cadena. Así sucesivamente hasta que se alcance uno que sí que ha hecho la petición, entonces no envía dicha señal a los siguientes

dispositivos. La conexión de los dispositivos establece el orden en el que se servirán las interrupciones en caso que haya peticiones simultáneas.

9. Entrada/Salida en la NDS

Este capítulo explica como funciona el sistema de entrada y salida de la consola Nintendo DS. La lista de ejercicios y el tiempo estimado (en minutos) para su realización se muestran en la Tabla 9.1.

9.1 Introducción

La videoconsola Nintendo DS dispone de una amplia variedad de periféricos muy sencillos que permiten la interacción con el usuario. Los más destacados son:

- Altavoces estéreos que cuentan con 16 canales de audio independientes.
- Dos pantallas LCD de 3 pulgadas. La pantalla inferior utiliza tecnología táctil.
- Varios temporizadores que pueden ser utilizados por una aplicación o un juego para definir diferentes respuestas dependiendo del tiempo.
- Un conjunto de botones que realizan diversas funciones de teclado.

En este capítulo se describirán fundamentalmente las características de los botones y los temporizadores, así como la gestión de interrupciones que permite realizar operaciones de E/S en la Nintendo DS empleando los periféricos citados.

9.2 Botones

El conjunto de botones que tiene la Nintendo DS constituyen un sencillo teclado que permite hacer operaciones de entrada muy simples, tal y como se muestran en la Figura 9.1. Esta figura muestra, además de todos los botones que tiene la Nintendo DS, qué procesador gestiona a cada uno de los periféricos.

La gestión de las operaciones se realiza a través de dos registros: *REG_KEYCNT* y *REG_KEYINPUT*. La figura 9.2 muestra el significado de cada uno de los bits de ambos registros, donde cada uno de ellos representa un botón.

- *REG_KEYCNT*: es el *Registro de control* y permite habilitar o enmascarar la interrupción asociada con la pulsación de un botón. Si el bit está a 1, cuando se pulsa dicho botón se activará la petición de interrupción. En caso contrario, no se activará. Además de los bits

Tabla 9.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo	Ejercicio	Tiempo
9.1	5'	9.6	5'
9.2	10'	9.7	10'
9.3	15'	9.8	15'
9.4	5'	9.9	15'
9.5	15'	9.10	25'

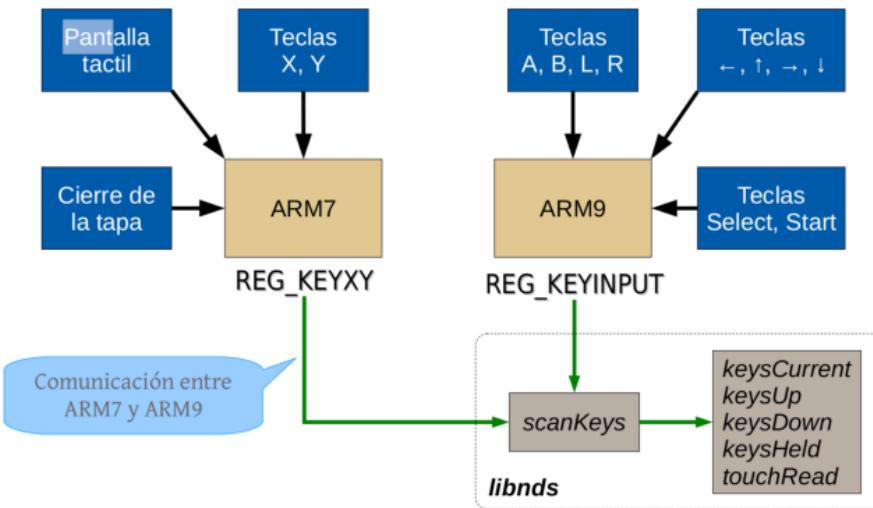


Figura 9.1: Botones en la Nintendo NDS

para cada botón, este registro de control para las interrupciones generadas por los botones tiene dos bits específicos:

- El bit 14 permite enmascarar de forma general todas las interrupciones de los botones. Si este bit está a 0, no se activará la petición de interrupción cuando se pulse cualquier botón, aunque hubiera bits a 1 en alguno de los bits correspondientes a los botones.
- El bit 15 sirve para que las interrupciones se produzcan por la pulsación de un solo botón (cuando el bit está a 0), o de varios botones a la vez (cuando está a 1). En este último caso, la interrupción se genera cuando estén pulsados a la vez todos los botones que tengan un 1.

Este registro está mapeado en la dirección de memoria `0x4000132`.

- **REG_KEYINPUT:** Es el *Registro de datos*. Almacena los botones pulsados. Pone a 0 el bit del botón que ha sido pulsado. Cuando se lee dicho registro el valor del bit vuelve al valor inicial a 1. La disposición de los bits del 0 al 13 es la misma que la descrita en el registro *REG_KEYCNT*. Este registro está mapeado en la dirección de memoria `0x4000130`.

9.3 Temporizadores

Un temporizador contiene un contador programable que cuenta de forma ascendente o descendente a la velocidad que le marca la frecuencia de trabajo. La cuenta, la dirección del conteo y el divisor de frecuencia, en la mayoría de los dispositivos, es programable. La característica más importante de estos dispositivos es que cuando alcanzan la cuenta final pueden solicitar una interrupción al procesador, siempre y cuando la tengan habilitada. Por ejemplo, supongamos un temporizador que cuenta de forma ascendente desde el valor que se carga en el contador hasta el

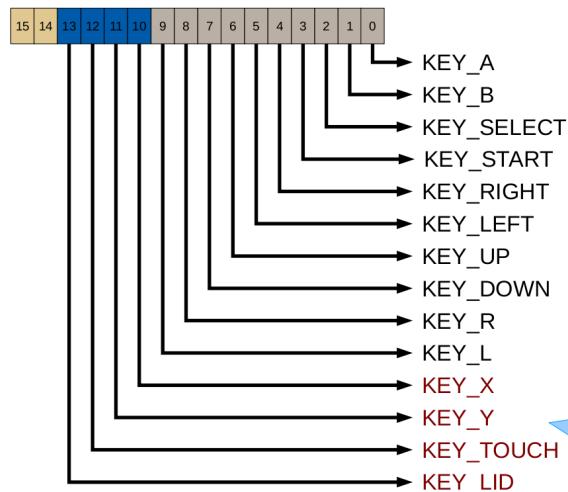


Figura 9.2: Significado de los bits de los registros de los botones

máximo de cuenta que puede alcanzar. Sea 255 (tiene 8 bits) el valor máximo y 20 el valor inicial. El contador inicia la cuenta ascendente en el momento que se le da la orden de inicio desde 20 hasta 255 y cuando alcanza ese valor solicitará una interrupción al procesador. El tiempo transcurrido entre ambas cuentas depende de la frecuencia del contador.

La Nintendo DS dispone de 8 temporizadores de 16 bits: cuatro en el ARM7 y cuatro en el ARM9. En los temporizadores que tiene la Nintendo DS, el contador siempre contará de forma ascendente desde la cuenta cargada (programable) hasta la máxima que puede alcanzar el contador (fijada por el número de bits, 16). En este caso, la programación del temporizador consistirá en establecer el valor inicial de la cuenta, el divisor de frecuencia, habilitar la interrupción y enviar la orden de inicio del contador. El temporizador solicitará una interrupción al procesador cuando alcanza la cuenta máxima.

La gestión de las operaciones con los temporizadores se lleva a cabo mediante dos registros:

- ***TIMER_CR(n)*** (*Registro de control*): Se utiliza para programar el modo de trabajo del temporizador, donde *n* indica cuál se está utilizando de los cuatro posibles (0,1,2,3). En este registro se escribe la siguiente información de control:

- Divisor de frecuencia utilizado: 1, 64, 256, 1024.
- Modo de trabajo: cascada o no. Es decir, si conectamos varios temporizadores en cadena para ampliar el rango de la cuenta.
- Habilitación/enmascaramiento de la interrupción: 1 habilitada, 0 enmascarada.
- Habilitación del temporizador: si se pone a 1 el bit correspondiente, el temporizador empieza la cuenta a partir de este momento.

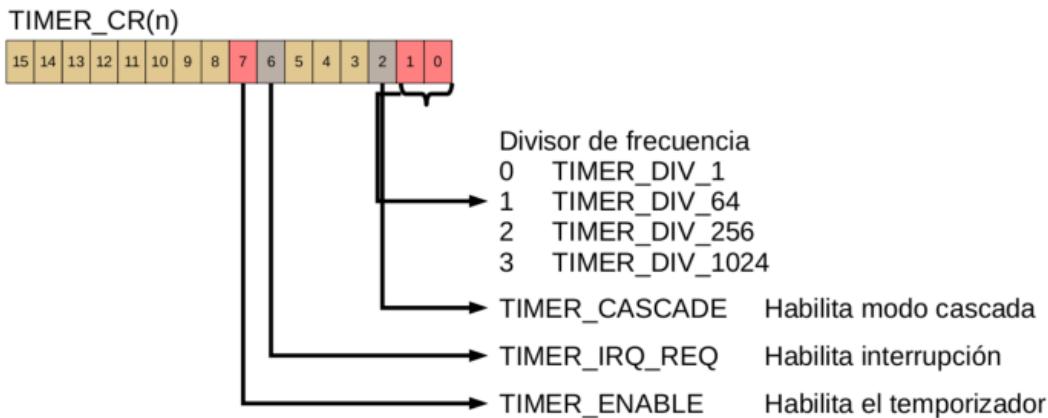
En la figura 9.3 se muestra la distribución de los bits del registro *TIMER_CR(n)* para su programación.

- ***TIMER_DATA(n)*** (*Registro de datos*): Almacena la cuenta de inicio.

Los registros *TIMER_DATA(0)* y *TIMER_CR(0)* están mapeados en las direcciones de memoria 0x4000100 y 0x4000102, respectivamente; los registros *TIMER_DATA(1)* y *TIMER_CR(1)* en las direcciones de memoria 0x4000104 y 0x4000106, respectivamente; y así sucesivamente.

9.4 Gestión de interrupciones en el procesador

El controlador de interrupciones del procesador ARM que se encarga de la gestión de interrupciones dispone de varios registros:

Figura 9.3: Campos de bits de `TIMER_CR(n)`

- *Registro maestro de interrupciones (IME):* Es un registro de 16 bits, cuya función es la de permitir o no que se produzcan interrupciones. En la biblioteca `libnds` es el registro `REG_IME` y está mapeado en memoria en la dirección `0x4000208`.
- *Registro de activación de interrupciones (IE):* Es un registro de 32 bits que indica qué interrupciones se quieren gestionar. Cada bit activa o desactiva un tipo de interrupción (no todos se usan). En la biblioteca `libnds` es el registro `REG_IE` y está mapeado en memoria en la dirección `0x4000210`.
- *Registro de petición de interrupciones (IF):* Es un registro de 32 bits en el que se pone a 1 el bit del tipo de interrupción que ha solicitado para interrumpir la CPU. En la biblioteca `libnds` es el registro `REG_IF` (de 32 bits) y está mapeado en memoria en la dirección `0x4000214`.

Aunque la Nintendo DS puede gestionar una amplia variedad de interrupciones hardware, las de E/S son un caso particular. Se utilizan para avisar al procesador de que el periférico requiere su atención.

En este capítulo sólo vamos a trabajar con dos tipos principales:

- Interrupciones para los temporizadores: `IRQ_TIMER0`, `IRQ_TIMER1`, `IRQ_TIMER2`, e `IRQ_TIMER3`. Cuando el contador del temporizador 0, 1, 2 ó 3 alcanza la cuenta máxima se activará la interrupción correspondiente.
- Interrupciones de los botones de la Nintendo DS: `IRQ_KEYS`. Cuando se pulsa alguno de los botones se activará esta interrupción.

El procesador de la consola Nintendo DS utiliza el direccionamiento **E/S mapeado en memoria** para asignar direcciones a los registros de los controladores. Esta técnica consiste en asignar a estos registros direcciones del mapa de direcciones de memoria y se realizan operaciones de lectura o escritura con las instrucciones que permiten el acceso a memoria.

Las interrupciones son **vectorizadas**. Esto quiere decir que durante la fase de procesamiento de una interrupción, la dirección de inicio de la RTI será enviada al procesador. Estas direcciones se almacenarán en una tabla en memoria y se enviará al procesador cuando la necesite. Esta tabla contiene todos los vectores de interrupción (direcciones de inicio de las RTIs) y que se identifica por el índice, es decir, posición que ocupan en dicha tabla.

El entorno de programación que estamos utilizando dispone de una biblioteca `libnds` que contiene un conjunto de funciones que facilita la programación de las operaciones de E/S en la Nintendo DS y, en particular, la gestión mediante interrupciones. Las principales funciones son:

- `irqSet`: se utiliza para definir la rutina de tratamiento de interrupción que gestiona una interrupción. Con esta llamada se inicializa la tabla de interrupciones con la dirección de inicio. Esta función tendrá dos parámetros: la interrupción y el nombre de la rutina de tratamiento.

- *irqEnable*: se utiliza para habilitar las interrupciones en el procesador. Esta función tiene como parámetros la lista de interrupciones a habilitar separadas por el símbolo !.
- *irqDisable*: sirve para enmascarar una interrupción en el procesador. A esta función se le pasa como parámetros la lista de interrupciones que se quieren enmascarar separadas por el símbolo !.
- *irqClear*: permite enmascarar interrupciones y además, también, las elimina de la tabla de interrupciones. Se le pasa como parámetro tantas interrupciones como se quieran eliminar.

9.5 Entrada de datos utilizando los botones en la Nintendo DS

En este apartado se van a describir algunos ejemplos muy sencillos en los que se van a realizar operaciones de entrada utilizando los botones y temporizadores. Su gestión se realizará por consulta de estado e interrupciones.

9.5.1 Consulta de estado

Como se ha descrito previamente, es una técnica muy sencilla para la gestión de la E/S. Consiste en implementar una espera hasta que se conozca que se ha realizado una entrada. Cuando se detecta se recoge el dato introducido.

Este programa muestra un posible ejemplo de implementación:

```
Leer estado periférico
while (! preparado) { //bucle de espera
    Leer estado periférico // hasta preparado
}
Recoger dato
```

Como se muestra en el código, el bucle de espera se basa en la comprobación de la condición *preparado*. Su implementación dependerá de las características de funcionamiento del periférico. En el caso de los botones de la Nintendo DS, el registro *REG_KEYINPUT* tiene un bit para cada botón y que cuando se pulsa, este bit se pone a 0. Cuando se lee dicho registro se vuelve a poner a 1. Leyendo este registro se puede conocer en qué instante se ha producido la pulsación de un botón.

■ **Ejemplo 9.1** Veamos un ejemplo concreto, *consulta_estado.c*, en el que se espera la pulsación del botón A. El siguiente código muestra la implementación:

```
1 #include <nnds.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     consoleDemoInit();
7     while(1)
8     {
9         while (REG_KEYINPUT != 0x03FE)
10             iprintf("\x1b[6;0H Esperando Boton A      ");
11             iprintf("\x1b[6;0HBoton A pulsado      ");
12             swiWaitForVBlank();
13     }
14 }
```

En este programa se ha implementado un bucle, etiquetado como *bucle de espera*, que el procesador estará ejecutando mientras se cumpla la condición, es decir, mientras el bit 0 del registro *REG_KEYINPUT* sea 1. Esta condición será falsa cuando este bit sea 0 y esto se produce cuando

se pulsa el botón A de la NDS. El procesador espera en este bucle hasta que se pulse dicho botón. Cuando esto ocurre continúa con la ejecución. Esto se repite de manera indefinida.

Ejercicio 9.1 Crea un proyecto con el código del ejemplo anterior y comprueba que funciona correctamente. ■

Ejercicio 9.2 Modifica el programa anterior para que se espere a la pulsación del botón B en vez del A. ■

Ejercicio 9.3 Crea un programa que cuente el número de veces que se pulsan cada uno de los botones A, B, UP y DOWN. ■

9.5.2 Interrupciones

En esta técnica, el periférico avisa al procesador que está preparado para llevar a cabo la transferencia de información. Para llevar a cabo su gestión es necesario:

- inicialización del controlador,
- inicialización del procesador,
- diseño de la rutina de tratamiento de la interrupción (RTI),
- actualización de la tabla de direcciones para que durante la fase de procesamiento de una interrupción esta información se envíe al procesador para que realice el salto a la RTI, y
- diseño de la RTI para gestionar esta interrupción.

■ **Ejemplo 9.2** Para ver cómo se implementa esta gestión en la Nintendo DS se muestra un ejemplo sencillo (*interrupciones.c*), donde se describe cómo se llevan a cabo todas estas acciones. El programa captura la pulsación del botón A y muestra el mensaje del botón pulsado:

```

1 #include <nds.h>
2 #include <stdio.h>
3 int contador = 0;
4
5 void int_boton()
6 {
7     if (REG_KEYINPUT == 0x03FE)
8     {
9         iprintf("\x1b[1;0H Boton A Pulsado");
10        contador = 0;
11    }
12 }
13
14 int main(void)
15 {
16     irqSet(IRQ_KEYS, int_boton);
17     irqEnable(IRQ_KEYS);
18     REG_KEYCNT = 0x4001;
19
20     consoleDemoInit();
21     while (1)
22     {
23         contador++;
24         iprintf("\x1b[1;0HPulsa A ");
25         iprintf("\x1b[14;0HContador = %04i", contador);
26
27         swiWaitForVBlank();
28     }
29 }
```

Este código contiene el programa principal (*main*) y la rutina que trata la interrupción (*int_boton*). Las operaciones que se realizan en el programa principal son las siguientes:

1. Asignar la RTI *int_boton* que gestiona la interrupción de los botones (*IRQ_KEYS*) mediante la función *IrqSet*. Esta rutina se almacena en memoria y actualiza la tabla de interrupciones con la dirección de inicio de esta rutina. Esta dirección se proporcionará al procesador durante el procesamiento de una interrupción.
2. Inicialización en el procesador: Habilitar la interrupción de los botones en el procesador (*IRQ_KEYS*) mediante *irqEnable*.
3. Inicialización del controlador: Habilitar la interrupción del botón A en el controlador. Esto se realiza poniendo a 1 el bit 0 del registro *REG_KEYCNT*. De esta forma cuando se pulse el botón A se activará la interrupción *IRQ_KEYS*.
4. A continuación el programa principal implementa un bucle infinito. El procesador estará ejecutando dicho bucle, y cuando se pulse el botón A, interrumpe su ejecución, atiende la interrupción y retorna.

Por otro lado, la RTI codifica las acciones que se tienen que ejecutar cuando se pulsa algún botón. La tarea principal de la RTI es identificar qué botón ha solicitado la interrupción y muestra el mensaje del botón pulsado.

Ejercicio 9.4 Comprueba que el programa anterior funciona correctamente.

Ejercicio 9.5 Modifica el programa realizado en el Ejercicio 9.3 para usar interrupciones en vez de consulta de estado. Puedes partir del ejemplo anterior.

9.5.3 E/S con el temporizador por interrupciones

En este apartado se muestra cómo se trabaja con el temporizador.

■ **Ejemplo 9.3** A continuación se muestra un ejemplo de su funcionamiento (*temporizador.c*). Se va a utilizar para contabilizar los segundos transcurridos desde que se inicializa.

```
1 #include <n3ds.h>
2 #include <stdio.h>
3 int contador = 0;
4
5 void int_timer()
6 {
7     contador++;
8 }
9
10 int main(void)
11 {
12     consoleDemoInit();
13     irqEnable(IRQ_TIMERO);
14     irqSet(IRQ_TIMERO, int_timer);
15
16     TIMER_DATA(0)=32764;
17     TIMER_CR(0) = TIMER_DIV_1024 | TIMER_ENABLE | TIMER_IRQ_REQ ;
18     while(1)
19     {
20         iprintf("\x1b[12;2H%2i", contador);
21         swiWaitForVBlank();
22     }
}
```

Este código contiene el programa principal (*main*) y la rutina que trata la interrupción (*int_timer*). Las operaciones que se realizan en el programa principal son las siguientes:

1. Inicialización en el procesador: Habilitar las interrupciones del temporizador 0 *IRQ_TIMER0* en el procesador.
2. Actualización de la tabla de vectores de interrupción para que la interrupción del temporizador sea gestionada por la rutina *int_timer*.
3. Inicialización del temporizador:
 - Inicialización del temporizador: Cargar el valor inicial de la cuenta en el registro *TIMER_DATA(0)* del temporizador para que sea posible contabilizar T segundos. El temporizador trabaja a una frecuencia base de 33.554 Mhz (*frec_base*). Para fijar este valor inicial, *cuenta_inicial*, y la frecuencia de trabajo, *frec_trabajo*, se utilizará la siguiente fórmula:

```

cuenta_final = 65.536
(cuenta maxima del contador de 16 bits)
frec_base =33.554 MHz
frec_trabajo = frec_base/divisor
T_ciclo = 1/frec_trabajo
T = (cuenta_final-cuenta_inicial)*T_ciclo
cuenta_inicial = cuenta_final - T * frec_trabajo
  
```

En este caso, si se desea generar una interrupción por segundo, T será 1, y el valor que se obtiene para *cuenta_inicial* será 32.768. Por tanto se asigna este valor a *TIMER_DATA(0)*.

- Habilitación de la interrupción del temporizador: Consiste en definir el divisor de frecuencia, habilitar la interrupción y dar la orden de inicio. Esta información se especifica en el registro *TIMER_DATA(0)*. En la configuración del registro de control *TIMER_CR(0)* se indica que se emplea un división de frecuencia de 1024, que se habilita el temporizador (*TIMER_ENABLE*) y que se genera una interrupción cuando se alcanza la cuenta máxima (*TIMER IRQ_REQ*). Con esta configuración se consigue generar una interrupción por segundo (aproximadamente).
4. En el programa principal se estará ejecutando un bucle infinito que muestra la variable segundos, que contabiliza los segundos transcurridos. A medida que se produce la interrupción del *timer*, la gestiona y continúa con la ejecución.

La RTI *int_timer* incrementa la variable segundos. El valor de esta variable se corresponde con los segundos transcurridos desde el comienzo de la ejecución, puesto que el temporizador está programado para generar una interrupción cada segundo.

Ejercicio 9.6 Comprueba el funcionamiento del programa anterior.

Ejercicio 9.7 Cambia el programa anterior para que el contador se incremente cada medio segundo.

Ejercicio 9.8 Cambia el programa anterior para que existan dos contadores, uno que se incremente cada segundo y otro cada 2.

9.6 Ejercicios avanzados

Ejercicio 9.9 Modifica el programa carreras de caballos para que el paso del tiempo se realice con interrupciones. Puedes partir de la solución proporcionada por el profesor. ■

Ejercicio 9.10 Realiza un programa que cada segundo incremente una variable por 1, empezando por el valor 0. El valor de esta variable se deberá mostrar por pantalla usando una cadena de caracteres. Para ello se deberá usar un temporizador controlado mediante interrupciones. Además, cuando se pulse el botón A, se cambiará la frecuencia de actualización para que sea cada 2 segundos (1 interrupción cada 2 segundos). Si se pulsa el botón B, la frecuencia será cada 0.5 segundos (2 interrupciones por segundo). Si se pulsa el botón UP la frecuencia volverá a ser cada segundo. Por último, si se pulsa el botón DOWN, la cuenta se reiniciará a cero. ■

10. Realización de un juego con gráficos

En este capítulo, se darán las instrucciones para la realización paso a paso de un videojuego usando el sistema gráfico de la NDS. La lista de ejercicios y el tiempo estimado (en minutos) para su realización se muestra en la Tabla 10.1.

10.1 Descripción del juego

El juego que se va a realizar es una versión simplificada del típico juego en el que una rana tiene que cruzar una carretera llena de vehículos evitando ser atropellada. La Figura 10.1¹ muestra un ejemplo de este tipo de juego. En el siguiente vídeo <https://www.youtube.com/watch?v=OpuduHSWBcw> podemos ver el funcionamiento de una versión de este juego.

En la versión que se va a desarrollar se usarán los conceptos aprendidos en los capítulos 5, 6, 7, 8 y 9.

10.2 Desarrollo del juego

La parte gráfica del juego se visualizará en la pantalla inferior. En la pantalla superior se mostrarán mensajes sobre el juego. Se pondrán realizar más de una partida, puesto que el objetivo será llegar a la meta intentando superar el récord actual. Para todas las cuestiones relacionadas con la entrada y salida (botones y temporizadores) se deberán emplear interrupciones.

10.2.1 Primeros pasos

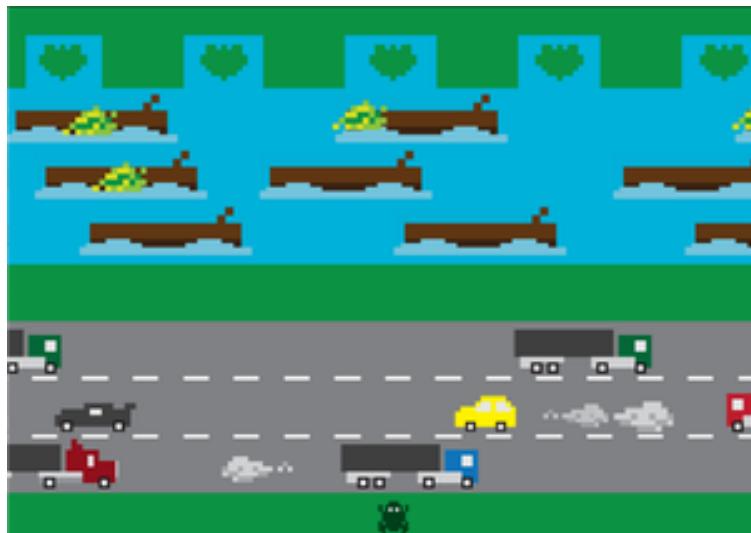
En primer lugar debemos crear las teselas para los gráficos del juego. Crearemos 6 teselas diferentes:

- Tesela 0: La rana.
- Tesela 1: Filas inferiores que serán la de posición inicial de la rana.
- Tesela 2: Fila superior que es donde tiene que llegar la rana.

¹La imagen se ha obtenido en la siguiente web: <https://frogmatters.files.wordpress.com/2008/04/frogster.png>

Tabla 10.1: Ejercicios del capítulo y tiempo estimado para su realización.

Ejercicio	Tiempo	Ejercicio	Tiempo
10.1	10'	10.7	20'
10.2	20'	10.8	40'
10.3	30'	10.9	20'
10.4	20'	10.10	40'
10.5	20'		
10.6	20'		

Figura 10.1: Juego *Crossing frogs*.

- Tesela 3: Fondo de la carretera.
- Tesela 4: Línea que divide la carretera.
- Tesela 5: Vehículo que hay que evitar.

■ **Ejemplo 10.1** El programa *ranas_inicial.c* muestra un ejemplo de las 6 teselas creadas y colocadas en la pantalla. El programa principal del mismo es el siguiente (en el programa completo están definidas las teselas y el mapa de teselas):

```

1 ...
2 int main( void )
3 {
4     int fila;
5     int columna;
6     int pos_mapMemory;
7     int pos_mapData;
8     int record;
9     int tiempo;
10
11    REG_POWERCNT = POWER_ALL_2D;
12    REG_DISPCNT = MODE_0_2D | DISPLAY_BG0_ACTIVE ;
13    VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG ;
14    BGCTRL [0] = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE
15        (1);
16
17    static u8* tileMemory = (u8*) BG_TILE_RAM(1);
18    static u16* mapMemory = (u16*) BG_MAP_RAM(0);

```

```

18 BG_PALETTE [0]=RGB15( 0, 0, 0); // Negro
19 BG_PALETTE [1]=RGB15(31,31,31); // Blanco
20 BG_PALETTE [2]=RGB15(15,15,15); // Gris
21 BG_PALETTE [3]=RGB15( 0,31, 0); // verde
22 BG_PALETTE [4]=RGB15(20,20,0); // amarillo apagado
23 BG_PALETTE [5]=RGB15( 0, 0,31); // azul
24
25
26 dmaCopy(t_rana,      tileMemory ,      sizeof(t_rana));
27 dmaCopy(t_salida,    tileMemory + 64, sizeof(t_salida));
28 dmaCopy(t_meta,      tileMemory + 128, sizeof(t_meta));
29 dmaCopy(t_carretera, tileMemory + 192, sizeof(t_carretera));
30 dmaCopy(t_mitad,     tileMemory + 256, sizeof(t_mitad));
31 dmaCopy(t_coches,    tileMemory + 320, sizeof(t_coches));
32
33 pos_mapData = 0;
34 for(fila=0;fila<24;fila++)
35   for(columna=0;columna<32;columna++)
36   {
37     pos_mapMemory          = fila*32+columna;
38     mapMemory[pos_mapMemory] = mapaData[pos_mapData];
39     pos_mapData++;
34   }
41
42 consoleDemoInit();
43
44 tiempo = 0;
45 record = 1000;
46 while(1)
47 {
48   iprintf("\x1b[4;2HTiempo = %d      ", tiempo);
49   iprintf("\x1b[6;2HRecord = %d      ", record);
50   swiWaitForVBlank();
51 }
52 }
```

Ejercicio 10.1 Crea un nuevo proyecto con el código anterior y comprueba que funciona correctamente.

Ejercicio 10.2 Modifica las teselas para poner tus propios gráficos.

10.2.2 Movimiento de la rana

La rana debe partir de una posición inicial en la fila inferior y se debe mover hasta alcanzar la fila superior. Para ello se usarán los botones de las flechas de la NDS (*UP*, *DOWN*, *LEFT* y *RIGHT*). A la hora de mover la rana es importante tener en cuenta los límites de la pantalla.

Ejercicio 10.3 Modifica el programa para que la rana se pueda mover. El control de qué botón se ha pulsado se tiene que hacer con interrupciones. Para ello se recomienda la creación de la función *ConfigurarInterrupciones()* donde se deberá incluir todo el código necesario para configurar las interrupciones y la función *MoverRana()* donde se deberá incluir todo el código necesario para realizar el movimiento de la rana. Será asimismo necesario tener las variables *rana_fila* y *rana_columna* para conocer en todo momento la posición actual de la rana.

Habréas comprobado que al mover la rana es necesario *borrar* el rastro que va dejando.

Ejercicio 10.4 Modifica el programa para que la rana no deje el rastro al moverse. Para ello tendrás que poner la tesela correspondiente (*salida*, *meta*, *mitad* o *carretera*) en la posición actual de la rana. Posteriormente deberás actualizar la posición de la rana teniendo en cuenta el botón pulsado y poner la tesela de la rana en la nueva posición. ■

También comprobarás que al mover la rana sobre las diferentes zonas del escenario, el color de fondo de la rana se mantiene. Sería interesante que el fondo de la rana se adaptase a cada zona del escenario.

Ejercicio 10.5 Modifica el programa para que el color de fondo de la rana se modifique según la posición de la pantalla donde se encuentre. Puedes crear varias teselas para la rana (según la zona del escenario donde se encuentre) o cambiar la paleta de colores. ■

10.2.3 Control del tiempo

El objetivo es alcanzar la meta en el menor tiempo que sea posible. Para ello será necesario tener un contador del número de segundos que tarda la rana en alcanzar la meta.

Ejercicio 10.6 Modifica el programa para controlar el tiempo que tarda la rana en llegar a la meta. Para ello se debe usar interrupciones y por lo tanto será necesario modificar la función *ConfigurarInterrupciones()* para configurar un temporizador. también será necesario crear una función que se ejecutará cada vez que ocurra una interrupción relacionada con el temporizador que hemos configurado. El tiempo actual se mostrará en la fila 4 de la pantalla superior. ■

10.2.4 Repetición de partidas

Al iniciar el programa nos mostrará una pantalla que nos pedirá pulsar el botón A (de la NDS) para iniciar la partida. Para ello, se mostrará un mensaje en la fila 2 de la pantalla superior para informar al usuario. Una vez pulsado el botón A, se inicializará el tiempo a 0, se colocará la rana en la posición inicial y el usuario ya podrá mover la rana.

Cuando el usuario alcance la fila superior se dará por finalizada la partida. Entonces, el programa debe comprobar si se ha superado el récord (mostrado en la fila 6). Si es así, se reemplazará el récord mostrado en la pantalla superior por el nuevo tiempo.

Una vez finalizada la partida, aparecerá de nuevo un mensaje en la fila 2 de la pantalla superior indicando que es necesario pulsar el botón A para iniciar una nueva partida.

Se recomienda implementar una máquina de estados con dos estados:

- Estado 0: se está esperando a que el usuario pulse la tecla A.
- Estado 1: el usuario está jugando.

Del estado 0 al 1 se pasa pulsando la tecla A. Del estado 1 al 0 se pasa cuando la rana alcanza la meta o cuando un vehículo atropella a la rana (tal como se verá más adelante).

Hay que tener en cuenta que, estando en el estado 0, los botones de dirección no deben estar habilitados, y que, estando en el estado 1, el botón A no debe estar habilitado.

Ejercicio 10.7 Modifica el programa para que se contemplen las mejoras especificadas en este apartado. Se recomienda tener una función *IniciarPartida()* donde se realicen todas las instrucciones necesarias para inicializar cada partida. ■

10.2.5 Movimiento de los coches

La coches de la parte superior de la carretera se deberán mover de derecha a izquierda, y los de la inferior al revés. Hay 10 filas en la parte superior (de la 1 a la 10) y 9 en la inferior (de la 13 a la 21) por donde se pueden mover los coches.

Al inicializarse cada partida se deberá llenar la carretera con coches de forma aleatoria, pero teniendo en cuenta que deberán haber menos teselas de coches que teselas de carretera para que exista un camino para llegar a la meta.

Para conseguir el movimiento de los coches en la pantalla superior, podemos hacer que cada cierto tiempo (por ejemplo cada 1 segundo) se copie en una determinada posición de la pantalla el contenido de la tesela situada a su derecha, creando un efecto de movimiento hacia la izquierda. Para la parte inferior, se hará al revés para conseguir un efecto de movimiento hacia la derecha.

Hay que tener en cuenta que la rana, en el caso de estar en la carretera, debe permanecer en el mismo sitio una vez finalizado el movimiento de los coches.

Ejercicio 10.8 Modifica el programa para que se contemplen las mejoras especificadas en este apartado. ■

10.2.6 Detectando colisiones

Para finalizar el juego se deberá añadir el código que permita detectar si un vehículo ha atropellado a la rana. Si ese es el caso, el juego deberá finalizar mostrando un mensaje *GAME OVER* y volviendo a mostrar la pantalla de inicio. Deberás comprobar si hay colisión tanto cuando se mueve la rana, como cuando se mueven los coches.

Ejercicio 10.9 Modifica el programa para que se detecten las colisiones y si es el caso se dé por finalizada la partida. ■

10.2.7 Añadir mejoras

A continuación se detallan algunas mejoras que se pueden implementar:

- Varios niveles de dificultad variando el número de coches en la pantalla. Cuantos más coches a la vez, más difícil.
- Varios niveles de dificultad variando la velocidad de desplazamiento de los coches. En unas filas los coches se moverán más rápido que en otras.
- Añadir otros vehículos como autobuses que pueden ocupar más de una tesela.
- Cambiar el gráfico de la rana según la dirección del movimiento.
- Cambiar el juego para que los gráficos sean de 2x2 teselas, permitiendo mayor calidad gráfica.

Ejercicio 10.10 Implementa alguna mejora de las anteriores o alguna otra idea que se te ocurra para mejorar el juego. ■

Bibliography

Books

Articles