

TUTORIAL:

PROGRAMACIÓN DE CONSOLAS DE 8 y 16 BITS, PARA NOVATOS

2ª PARTE : ENTRANDO EN HARINA... O en CPU, según se quiera :-)

Por David Senabre, Marzo 2006

## DEFINITIVAMENTE QUIERO APRENDER

Muy bien, si ya sabes un poco de qué va esto, pero no tienes experiencia, necesitarás un poco de ayuda, o de lo contrario muchas ganas para no frustrarte.

Lo primero es decidir qué quieres programar. En este documento contrastaré 4 opciones, y daré algunos conceptos básicos un poco más específicos. Mi objetivo no es hacer un tutorial completo de cómo programar, desde cero, porque no acabaría :)

Lo que quiero es preparar el terreno, para que luego puedas entender otros documentos, y ser capaz de entender algún programa sencillo, y aprender a programar.

## 8 ó 16 BITS

Si es lo primero que haces te recomiendo 8 bits, no porque sean 8 bits, sino porque la NES y las Master System tienen una arquitectura más simple, y su procesador es también más sencillo de programar.

Pero eres libre de elegir. Te recomiendo que consigas las herramientas necesarias, los documentos, y una demo sencilla, como las que he hecho yo, que están pensadas para aprender.

Antes de poder decidir de manera más o menos conciente, deberías conocer algunos términos más, a parte de los que introduje en la primera parte. Y ese es el objetivo de este tutorial. Me centraré sobre las 8 bits, de momento.

## FUNDAMENTOS DE UNA CPU

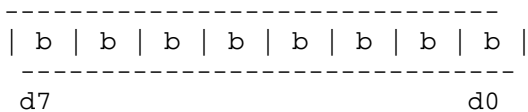
Como vas a programar en ensamblador, necesitas conocer la estructura interna de la CPU. Voy a resumir los conceptos imprescindibles:

\* Has de tener claro que **una CPU es la máquina más estúpida del mundo**. Sólo hace lo que le dices, y si lo que le dices es una tontería, hará una tontería. Es especialmente importante entender esto, porque puedes hacer un programa que es entendible por la CPU, pero luego lo pruebas y todo se cuelga.

\* La CPU tiene **registros internos**, que usa para mover datos, almacenar resultados intermedios, y operar con ellos. Pueden ser de 8 bits, de 16 o de 32, lo que indica el número de bits que puede almacenar. Para que los imagines mejor, supón que son diminutas memorias terriblemente rápidas, que están dentro de la CPU, y que ésta usa continuamente en su funcionamiento. Son mucho más rápidos que la memoria RAM, y por lo tanto permiten que el procesador pueda trabajar a mayor velocidad.

No debes confundir estos registros con los puertos o registros de entrada/salida (visto en la 1ª parte). Los puertos de entrada/salida son direcciones de la CPU que van dirigidas a algún dispositivo del sistema, y que sirven para comunicarse con ellos, mientras que los registros de la CPU son pequeñas memorias dentro de la CPU, muy

rápidas, que usa como almacenamiento temporal para llevar a cabo sus tareas. Comprenderás mejor la diferencia a nada que empieces a programar. De todas formas, de cara al programador, puedes imaginar un registro de 8 bits como lo hacías para un puerto de entrada/salida.



Donde "b" representa cada bit, d0 el bit menos significativo, y d7 el más significativo.

Hay registros de propósito general, que se pueden usar para lo que el programador quiera, y algunos registros de propósito específico, que tiene una función definida. Entre éstos últimos destaca el PC (contador de programa) que almacena la dirección de la próxima instrucción a ejecutar.

\* Básicamente una CPU hace lo siguiente: **buscar instrucción en la memoria, ejecutar instrucción, buscar siguiente instrucción, ejecutar instrucción,...**

Un programa se compone de una lista de instrucciones que la CPU va ejecutando una tras otra. Todas ellas están en memoria (ROM o RAM), y el registro PC se encarga de apuntar a la dirección de la siguiente instrucción a ejecutar.

Una instrucción no es más que una secuencia concreta de "unos" y "ceros", con significado para la CPU, y que le ordena hacer algo. Se llama **ciclo de instrucción** a los pasos que da una CPU para ejecutar una instrucción. El esquema general y simplificado es el siguiente. La CPU carga la instrucción almacenada en la dirección de memoria que indica su registro PC, la trae a su interior, la decodifica, para ver qué significa, y la ejecuta. También aumenta su registro PC, para apuntar a la siguiente instrucción. Y así continuamente. Cada CPU tiene una lista de instrucciones donde figuran todas las cosas que sabe hacer.

Vale, si no tenías ni idea de esto puede ser complejo imaginarlo sin un ejemplo. Así que ejemplo.

<b>CPU (2). INTRODUCCIÓN A ENSAMBLADOR</b>
--------------------------------------------

El 6502 tiene un registro acumulador (A), y 2 registros de propósito general (X e Y). Todos de 8 bits.

Cada línea a continuación es una instrucción máquina, escrita en ensamblador para simplificar la cosa, que es como se hace en la vida real, para no tener que recordar las secuencias de "unos" y "ceros".

```
lda #$3f  
sta $2006
```

"lda" es una instrucción que carga en el registro A, el valor hexadecimal 3F.  
\$ significa valor en hexadecimal  
# significa que le vas a dar un valor constante. Si no lo pusieras, la CPU entendería que quieres cargar no el número \$3F, sino el contenido de la posición \$3F de la memoria. Por lo tanto, si la posición de memoria \$3F almacenara el valor \$67, se cargaría \$67 en el registro A.

"sta" es una instrucción que almacena el valor del registro A en la posición de memoria \$2006. Como ves no lleva #, porque nos referimos a una posición de memoria, no a un valor constante. No tendría sentido almacenar un valor en #\$2006, porque #\$2006 no es nada más que un número.

¿Que hacen estas 2 instrucciones? Almacenar el valor #\$3F en la posición de memoria \$2006 (que en este caso no es una posición de la RAM de la NES, sino un puerto que se usa para acceder a la VRAM).

Esta misma operación, en otra CPU se hace de otra manera. Pondré varios ejemplos para contrastar.

Un Z80 es un procesador de 8 bits, que tiene también un registro acumulador A, pero además cuenta con 6 registros de propósito general de 8 bits (B,C,D,E,H y L) que se pueden emparejar para hacer grupos de 16 bits, para formar AB, CD y HL. Aunque internamente se siguen moviendo de 8 bits en 8 bits, pero a la hora de programar en ensamblador se pueden usar en pares, simplificando ciertas tareas. Tiene además otros registros que sirven de índice, y puntero a la pila.

Volviendo a nuestro ejemplo, en el Z80 se haría así

```
ld    a, $3F
ld    ($2006), a
```

"ld" es una instrucción que carga el segundo operando en el primero. En la primera línea, el \$3F en el registro A. En la segunda línea, el registro A en la posición \$2006.

Fíjate que aquí no se usa el # como antes. En este ensamblador, si quieres acceder a una posición de memoria debes poner el valor entre paréntesis, que indica algo así como "acceder al contenido de..."

Imagina que quieres guardar el valor \$30 pero no en el registro A, sino en el posición de memoria que contiene el registro A.

```
ld a, $30
ld (a), $3F
```

Primero se almacena el valor \$30 en A, luego se almacena el valor \$3F en la posición \$30 (que es el valor que almacena A).

En el caso del 6502 esto sería un poco más complicado.

```
lda #$30
sta $0010
lda #$3F
sta ($0010)
```

- 1) carga el valor constante \$30 en A.
- 2) "sta" guarda el valor de A en la posición dada, en este caso \$10, por ejemplo, suponiendo que es una posición de la RAM (en la NES, lo es).
- 3) cargar en A el valor constante \$3F
- 4) y almacenarlo en la posición que apunta \$10. Como en \$10 hemos escrito \$30, el valor de A se almacenará en la posición \$30. Si hubieramos escrito lo mismo sin paréntesis, se habría hecho lo que en el punto 2, almacenar el \$3F en la posición \$10.

Espero que este ejemplo, y un poco de esfuerzo por tu parte, te sirva para que esto de las CPU y las instrucciones tengan una existencias más real en tu cabeza :)

Debes saber que hay deferentes maneras de mover datos, como has podido ver. De un valor constante a un registro. De un registro a una posición de memoria que tú le das. O de un registro a una posición que apunta la dirección que tú le das.

Esto se llama **modos de direccionamiento**, y hace referencia a la manera de acceder a los datos. Pon mucha atención a lo que sigue. Pondré sólo unos pares de ejemplo sencillos e imprescindibles.

### CPU (3). INTRODUCCIÓN A MODOS DE DIRECCIONAMIENTO

*La primera línea es para el caso del 6502, la segunda para el Z80*

#### **Direccionamiento inmediato:**

Mueve un valor constante a un destino

```
lda #$30
ld a, $30
```

#### **Direccionamiento directo:**

Accede a una posición de memoria que tú le das de forma directa.

```
sta $0010
ld ($2006), a
```

#### **Direccionamiento indirecto:**

Accede a una posición de memoria apuntada por la dirección o registro que tú le das.

```
sta ($0010)
ld (a), $3F
```

Como puedes ver, el Z80 no puede hacer un direccionamiento indirecto de memoria, del tipo

```
ld ($0010), $3F
```

Porque esto significa para él direccionamiento directo, y guardaría \$3F en la posición \$10. No vale hacer,

```
ld (($0010)), $3F
```

Si quieres hacer un acceso indirecto, y almacenar \$3F en la posición que apunta la posición \$10, tienes que usar un registro, cargar el contenido de la posición \$10, y acceder con el registro. Pero como las direcciones en el Z80 son de 16 bits, tendrás que usar un par de registros. Usemos el BC ahora.

```
ld bc, ($10)
ld (bc), $3F
```

La segunda línea es un direccionamiento indirecto, sí, pero de registro. El Z80 no soporta un direccionamiento indirecto a una posición de memoria, mientras que para el 6502 era más sencillo, porque sí lo soporta, y esto se haría,

```
lda #$3F
sta ($0010)
```

En resumen... si X es un valor numérico:

- Para el 6502.  
#X, valor inmediato X  
X, posición de memoria X  
(X), posición que apunta el contenido de X

- Para el Z80.  
X, valor inmediato X  
(X), posición de memoria X  
(R), posición que apunta el contenido del registro R.  
Pero no hay manera de decirle que apunte a la posición que apunta otra posición haciendo ((X)) ni nada parecido. Para paliar esto deberás apoyarte en el indirecto usando registros. Para compensar esto también existe el direccionamiento indexado, pero es algo que no tratare de momento, y que por cierto también dispone el 6502.

Como ves, y espero que tengas más claro, tendrás que familiarizarte con la arquitectura de cada CPU. Espero que este ejemplo te haya expuesto la importancia de este hecho, que a lo mejor has oído muchas veces sin entender realmente el porqué.

Existen otros conceptos básicos como la pila, los saltos y las subrutinas, que resumiré más adelante.

#### ACCEDIENDO AL CHIP GRÁFICO

Con controlar la CPU no es suficiente si quieres mostrar algo por pantalla. Deberás programar el chip gráfico. Pero tranquilo, este no se programa como la CPU, es decir, no hay que hacerlo en ensamblador. Realmente es más sencillo, sólo tienes que darle comandos.

Como ya dije en el anterior tutorial, en la consola, a parte de la RAM y la ROM, existen puertos, que son zonas del mapa de memoria que no se refieren a memoria, sino a ciertos dispositivos, y sirven para controlarlos y comunicarse con ellos. Es decir, permiten a la CPU interactuar con el resto de elementos del sistema. Si quieres escribir algo en la VRAM, que no es accesible desde la CPU, sino sólo a través del chip de vídeo, deberás usar puertos del chip de vídeo. Hay puertos para dar la dirección de la VRAM donde quieres escribir, y puertos para escribir los datos en ella.

De nuevo recorro a un ejemplo. Imagina que quieres escribir en la posición \$23C8 de la VRAM. En la NES, el puerto para dar la dirección es el \$2006, y el que se usa para escribir o leer esa dirección es el \$2007. Vamos a escribir un \$1 en la posición \$23C8

```
lda #$23
sta $2006
lda #$C8
sta $2006
```

```
lda #$1
sta $2007
```

Y si quisieramos leer, cambiaríamos las últimas 2 líneas por

```
lda $2007
```

Y en el registro A tendríamos el byte leído.

En la Master System en cambio, los puertos no están el mapa de memoria de la CPU, sino que se accede a ellos a través de las instrucciones "in" y "out". El puerto de dirección de VRAM es el \$BF, y el de acceso a esa dirección, el \$BE. Escribamos pues un \$1 en la posición \$23C8

```
out ($bf), $C8
out ($bf), $23
out ($be), $1
```

*Lo de arriba está mal, y luego diré porqué. Habría que haber escrito "out (\$bf), (\$23 | \$40), pero no lo he hecho por no liar ahora.*

Si quisiéramos leer, basta con cambiar la última línea por

```
in a, ($be)
```

Para leer el byte y guardarlo en el registro A, aunque podríamos haber usado cualquier otro.

Igualmente, hay puertos para controlar el chip de vídeo en sí (para activar y desactivar la imagen, los sprites, el lugar donde almacena los datos, la resolución, modo gráfico,...).

Los mandos también se leen de un puerto. Cuando quieres saber qué botones están pulsados, lees en el puerto correspondiente. En la NES es el \$4016, y en la Master System el \$DC. Más adelante diré como leer el estado del pad.

## USANDO EL CHIP GRÁFICO

Esto es de las cosas más difíciles de entender, junto con el ensamblador. Cuando lo tengas más o menos claro, el camino se allanará. Ánimo.

Para poder mostrar gráficos por pantalla necesitas hacer varias cosas.

- 0) Inicializar el chip gráfico.
- 1) Cargar los gráficos en la VRAM.
- 2) Cargar una paleta.
- 3) Cargar el plano (o planos) de scroll.
- 4) Cargar sprites, si deseas que haya (no es necesario).
- 5) Activar la imagen.

Bueno, voy a pasar por encima de cada punto tocando los puntos fundamentales, pero recuerda que no voy a dar una exhaustiva descripción de cómo hacer todo. Cuando elijas programar algo en concreto, tendrás que profundizar en ello, y referirte a los documentos específicos. Lo que yo busco es dar la base para poder entenderlos.

### NES

**0) El chip tiene varios registros que lo controlan.** En los documentos técnicos se tratan todos ellos y para qué sirven. Normalmente se requiere escribir ciertos valores en ellos para inicializarlo y ponerlo a funcionar como a nosotros nos interesa.

Ejemplo.

LA NES sólo tiene 2 registros de control de la PPU. El \$2000 y el \$2001. Si abres un documento técnico, como por ejemplo de Yoshi, y buscas registros, verás esta información respecto al \$2000.

Address	Description
\$2000	PPU Control Register #1 (W) D7: Execute NMI on VBlank 0 = Disabled 1 = Enabled D6: PPU Master/Slave Selection --+ 0 = Master +-- UNUSED 1 = Slave --+ D5: Sprite Size 0 = 8x8 1 = 8x16 D4: Background Pattern Table Address 0 = \$0000 (VRAM) 1 = \$1000 (VRAM) D3: Sprite Pattern Table Address 0 = \$0000 (VRAM) 1 = \$1000 (VRAM) D2: PPU Address Increment 0 = Increment by 1 1 = Increment by 32 D1-D0: Name Table Address 00 = \$2000 (VRAM) 01 = \$2400 (VRAM) 10 = \$2800 (VRAM) 11 = \$2C00 (VRAM)

Nos dice para que vale cada bit que escribimos en él. NMI es una interrupción, hablaré de ello más adelante. Los bits D4, D3 y D1-D0, sirven para indicar en qué lugar de VRAM almacenaremos diversas cosas, y "PPU Address Increment" indica cuantas posiciones aumentará la dirección de la VRAM, cuando escribimos algo en ella. Nosotros de momento queremos que sea 1, porque así cuando escribamos cosas en la VRAM, cada byte que escribimos hará que la posición avance en uno, y así podremos escribir el siguiente byte sin tener que volver a darle la dirección.

Vamos a inicializar la PPU, desactivando el "NMI on VBlank", usando "PPU Increment by 1", tamaño de sprites de 8x8, "Background Pattern Table" en posición \$0000 de VRAM, la de sprites en \$1000, y el "Name Table" en \$2000.

En binario sería: 00001000.

D7 es el bit más a la izquierda. D0 el más a la derecha.

En hexadecimal esto sería \$08

```
lda  #$08
sta  $2000
```

Listo. También tendremos que inicializar el registro \$2001, que básicamente sirve para activar y desactivar los sprites y el plano de scroll.

### 1) Cargar los gráficos en la VRAM.

Pattern table es la tabla que almacena los tiles que vamos a usar. Los tiles que vamos a usar para el plano de scroll los almacenaremos en \$0000 de la VRAM, y los de los sprites, si los hubiera, en \$1000,

tal y como acabamos de decirle a la PPU. Ojo porque puedes poner ambos en la misma dirección, y decirle a la PPU que quieres usar \$0000 para sprites también. En ese caso, los tiles de la zona \$0000-\$0FFF se usarán para ambos, plano y sprites. Si tienes pocos tiles, podría ser más sencillo esto.

Para poner los tiles en esa zona, tendremos que hacer, con un editor, el archivo con los tiles, y salvarlo en formato de la NES (por ejemplo usando Tile Molester, y codec 2bpp planar composite). Luego le diremos al ensamblador que nos coloque ese archivo en el sitio adecuado.

Para entender bien esto procura entender que las ROMS de NES tienen 2 memorias, PGR-ROM y CHR-ROM. En PGR-ROM irá nuestro programa, y en CHR-ROM los tiles. Las pattern tables van en la CHR-ROM, y en la VRAM interna de la consola sólo están las 2 Name Tables que uses, la paleta y la información de los sprites. Si miras el mapa de memoria de la PPU, verás que todo está dentro de la misma tabla, pero realmente sólo las pattern tables están en la CHR-ROM, y se acceden desde la PPU como si fuera parte de su VRAM.

Es así porque la VRAM interna es de solo 2Kb, y eso es lo que ocupan 2 Name Tables con sus respectivos Attribute Table. Cada Pattern Table ocupa \$1000 posiciones, lo que es lo mismo, 4Kb. Cada tile ocupa 16 bytes, porque son de 8x8 pixels, y cada píxel se representa con 2 bits, o sea 8x8x2 bits, que dividido por 8 para pasarlo a bytes, da 16, y como caben 256 tiles en cada tabla, 256\*16 = 4Kb. Por tanto las 2 Pattern Tables ocupan 8Kb.

Por tanto no hay que cargar los tiles en la VRAM interna, accederemos a ellos en la CHR-ROM, donde están grabados (sólo recuerda, aunque no lo voy a tratar, que hay juegos que usan una CHR-RAM, y que si graban tiles en ella).

## 2) Cargar una paleta.

La paleta es de 32 colores, y cada color es representados por un byte. Los primeros 16 son la plata para el plano de scroll, y los siguientes para sprites. Cada paleta de 16 bytes está a su vez dividida en 4 grupos de 4 colores. Un sprite, o un tile en el plano de scroll, deberán usar uno de estos grupos de 4 colores. Sí, sólo pueden mostrar 4 colores. Y realmente 3, porque el primero de cada grupo de 4 colores representa el color transparente. Si un sprite tiene un píxel con color 0, eso significa que no hay que dibujar nada ahí, y que se verá lo que hay detrás de él. En un plano de scroll, representa su color de fondo, el que se muestra donde hay un píxel transparente.

Para cargar la paleta tienes que escribir en su zona de memoria de la PPU, que empieza en \$3F00.

```
lda #$3f    ; A <- $3F
sta $2006   ;
lda #0      ; A <- 0
sta $2006   ; Apuntar a dirección $3F00
```

E ir escribiendo los colores en \$2007. Como hemos puesto "PPU Increment 1", cuando escribamos en \$3F00, automáticamente se pondrá la dirección \$3F01, y sólo tendremos que escribir el siguiente byte hasta que lo hagamos para los 32 colores.

Para saber cual es cada color deberás mirar una tabla en el que vengan, porque al contrario que en el resto de consolas que conozco, los códigos de color en la NES no tienen mucho sentido.

### 3) información sobre qué gráficos mostrar y en qué sitio

Esto va en la Name Table. La Name Table es una tabla, o matriz, de 32x30, que almacena información sobre cada bloque en el plano de scroll. Este plano se divide en 32 celdas de ancho, y 30 de alto.

Si quieres que la primera celda, la de la esquina superior izquierda, muestre el tile número 34, deberás escribir 34 en la primera posición de la Name Table correspondiente. Le sigue el de su derecha, hasta llegar al número 32. El siguiente será el de la izquierda del todo, pero el de la fila de abajo. Así hasta cubrir las 30 filas.

Esto lo entenderás mejor con una demo o practicando.

Para escribir al principio de la Name Table 0 un 34.

```
lda #$20
sta $2006
lda #$00
sta $2006 ;direccion VRAM = $2000, Name Table 0, offset 0
```

```
lda #34
sta $2007
```

Para que se muestren los tiles 34, 35, 32 y 30, consecutivamente, en la parte superior izquierda, deberemos añadir,

```
lda #35
sta $2007
lda #32
sta $2007
lda #30
sta $2007
```

Aunque cuando hay muchas cosas que copiar no se usa este procedimiento porque resultaría muy largo, sino que se usan bucles. También hablaré de ello más adelante. De momento sólo entender lo básico.

### 4) Cargar sprites, si deseas que haya (no es necesario).

La información de los sprites se almacena en la llamada SPR-Table. En ella se dedican 4 bytes a cada sprite.

Cuando consultes un documento técnico, podrás ver que el;

- Primer byte representa su coordenada Y en la pantalla
- Segundo byte el índice del tile que muestra
- Tercer byte, información extra que ahora comento.
- Cuarto byte, coordenada X en la pantalla.

El tercer byte permite girar el sprite, de manera que mire para el lado contrario, por ejemplo, lo que evita tener que dibujarlo girado en la Pattern Table. También permite darle prioridad, o no, para que se dibuje delante o detrás del plano de scroll, y permite decir que paleta usará. La paleta se le indica con un número de 0 a 3, que seleccionará uno de los 4 grupos de 4 colores de la paleta de sprites.

Si quieres usar 2 sprites, tendrás que escribir los 4 bytes del primero, y luego los 4 bytes del otro, todo seguido, y así hasta un límite de 64 posibles.

Para escribir tienes 2 opciones. De momento sólo comento la forma manual, que está bien para pocos sprites. Se usan los puertos \$2003 y \$2004. Escribe en \$2003 la dirección de la tabla de sprites donde quieres escribir, y \$2004 para escribir el dato. Recuerda que si quieres acceder al segundo sprite, tendrás que acceder a la posición \$4, a la \$5 para el tile a mostrar, etc... y si quieres acceder al tile que muestra el primer sprite, a la posición \$1. El tercer sprite, si lo hubiera, comenzará en la posición \$8 y así sucesivamente, hasta llegar a \$FF, que es la coordenada X del sprite número 64, y final de la tabla de sprites.

De nuevo, en el documento de Yoghi podrás encontrar información sobre la tabla de sprites.

Byte	Bits	Description
0	YYYYYYY	Y Coordinate - 1. Consider the coordinate the upper-left corner of the sprite itself.
1	IIIIIIII	Tile Index #
2	vhp000cc	Attributes v = Vertical Flip (1=Flip) h = Horizontal Flip (1=Flip) p = Background Priority 0 = In front 1 = Behind c = Upper two (2) bits of colour
3	XXXXXXXX	X Coordinate (upper-left corner)

### 5) Activar la imagen.

```
lda #%00011110
sta $2001
```

Dice a la PPU que muestre los sprites y el plano de scroll, a parte de otras cosas. Dejo como ejercicio al lector que lo compruebe. Como puedes ver, en lugar de hexadecimal, a veces está bien dar el número en binario, porque al principio ves mejor la correspondencia de cada bit con el registro. Cuando ganes experiencia, lo verás tanto o mejor en hexadecimal.

Ten en cuenta que todo esto para otra consola sería muy distinto, porque cada una tiene su propio chip de video. En la Master System el proceso es más o menos igual de fácil. Y digo fácil porque en la Megadrive las cosas se complican, ya que tiene más puertos, más opciones, más planos, etc... y en el caso de la SNES se complica aún más.

Ahora haré lo mismo pero para el caso de una Master System.

### MASTER SYSTEM

#### 0) Inicializar el chip gráfico.

Aquí usaremos a saco los puertos \$BF y \$BE. Procura prestar atención a su uso. Al final haré un breve resumen, pero mi objetivo es sólo dar un vistazo superficial a todo el proceso.

Habría que inicializar al menos los registros \$0 y \$1. Por ejemplo, para decirle a la VDP que use la resolución y modo gráfico estándar, scroll normal, sprites de 8x8 pixels (que también es lo normal), y que desactive las interrupciones, se tendría que hacer;

```

out ($bf), $06
out ($bf), $80      ;escribir 06 en el puerto $0

out ($bf), $C6
out ($bf), $81      ;escribir C6 en el puerto $1

```

En resumidas cuentas, para escribir a un registro del VDP hay que escribir 2 bytes en el puerto \$BF. El primer byte es el dato a escribir en el registro del VDP, y el segundo es el número de registro, con el bit más significativo a 1. Por ejemplo, el registro 1 en binario es 00000001, con el bit más significativo a 1, sería 10000001, que en hexadecimal es \$81.

Habría que hacer otras cosillas, sobretodo decirle donde vamos a colocar la Name Table y la tabla de sprites, que es un poco más difícil que en la NES. Para ello se usan los registros \$2 y \$5. Pero de momento no hagamos nada, y por defecto la dirección donde tendremos que poner la Name Table es \$3800, y la tabla de sprites en \$3F00.

Las posiciones de VRAM desde \$0000 a \$37FF las podemos usar para almacenar tiles.

### 1) Cargar los gráficos en la VRAM.

Al contrario que la NES, en la Master System gráficos y programa van en la misma memoria (y así ocurre con todas las consolas que conozco, excepto la NES). Por ello, los tiles debemos cargarlos desde la posición de la ROM donde los hayamos puesto, a la VRAM. Y normalmente lo haremos al comienzo de la VRAM, y suponiendo que no hayamos cambiada la dirección de las tablas, podremos llenar hasta la posición \$37FF como dije al final del anterior apartado.

Para escribir en la VRAM tenemos que escribir al puerto \$BF. Lo haremos CASI como cuando escribimos a los registros de la VDP, PERO ahora, en lugar de poner a 1 el bit más significativo (bit 7), tenemos que poner a 1 el bit 6.

Para no liar con comparaciones, diré como hay que hacerlo. Primero, tenemos que dividir la dirección de VRAM en 2 parte, el byte bajo y el alto. Si queremos acceder a \$2342, pues el byte alto será \$23 y el otro \$42. Segundo, tenemos que escribir primero el byte bajo, y luego el alto. Tercero, tenemos que poner a 1 el bit 6 del byte alto.

De manera que quedaría la primera escritura \$42, y la segunda \$63.

Y si queremos copiar algo a la dirección \$0000, tendremos que escribir \$4000 al puerto \$BF, en dos veces, o sea \$00, y luego \$40. Fíjate que el 4 en binario es 0100 (y por tanto, el bit 6 lo hemos puesto a 1).

```

out ($bf), $00
out ($bf), $40

```

También podemos escribir la dirección que queramos, haciendo una operación OR con el byte alto y \$40 (fíjate que eso pondrá a 1 el bit 6 del byte alto)

```

out ($bf), $00
out ($bf), ($00 | $40)      ;la "|" significa OR

```

y luego para escribir en la dirección indicada, lo hacemos simplemente en el puerto \$BE. Si queremos copiar el contenido del registro A;

```
out ($be),a
```

## **2) Cargar una paleta.**

La paleta está compuesta por 32 colores como en la NES. Los primeros 16 para el plano de scroll, y los segundos para los sprites. Tienes que escribir los colores en la memoria de paleta, igual que en la NES, pero aquí tendrás que usar el puerto \$BF.

Se hace igual que si fueras a escribir en la VRAM, pero ahora tiene que poner los bits 6 y 7 del byte alto a 1.

Por ejemplo, para escribir el primer color de la paleta, la dirección será \$0000, partiendo en 2 bytes, y poniendo los bits 6 y 7 del byte alto a 1

```
out ($bf),$00  
out ($bf),$C0
```

Si quieres escribir el color número 5, la dirección sería;

```
out ($bf),$05  
out ($bf),$C0
```

y una vez dada, escribe a \$BE el color. Ahora es más sencillo que en la NES, porque el color se compone de un valor de rojo, otro de verde y otro de azul (RGB). 2 bits para cada componente, o sea 4 valores posibles de R,G y B.

El byte que escribas debe tener este formato:

```
xxBBGGRR      (xx; da igual lo que pongas)
```

Un color con 3 de rojo, 1 de azul y 0 de verde sería xx010011

## **3) información sobre qué gráficos mostrar y en qué sitio**

Esto significa cargar la Name Table con información de qué tile se muestra en cada bloque del plano de scroll.

Si la tabla está en \$3800, deberás empezar a escribir ahí, y los 2 primeros bytes se referirán al bloque de 8x8 de arriba a la izquierda. Cada bloque ocupa 2 bytes en la tabla, y tiene este formato

```
xxxPCVHN.NNNNNNNN
```

O sea que tendrás que hacer 2 escrituras para llenar un tile. Primero el byte bajo, como de costumbre en la Master System. Lo más importante es el valor de N, ya que indica el tile que se va a mostrar. El resto sirven para girar el sprite, darle prioridad o cambiar la paleta.

Para mostrar el tile nº 23 en la esquina superior izquierda, suponiendo \$3800 como principio de la tabla;

```
out ($bf),$00  
out ($bf),($38 | $40)      ;bit 6 a 1 porque vamos a hacer
```

```

                                ; escritura a VRAM
out ($be), $0                    ;no cambiamos el byte alto
out ($be), 23

```

Si quieres usar el tile 256, necesitarás usar el byte alto, ya que 256 es 00000001.00000000

```

out ($be), $01                    ;usamos 8° bit de N
out ($be), 23

```

#### **4) Cargar sprites, si deseas que haya (no es necesario).**

Se hace igual, escribiendo en la tabla de sprites, que por defecto está en \$3F00. Es parecido a la NES, se usan 4 bytes para definir un sprite. Uno para posición X, otro para posición Y, otro para tile a mostrar, y otro que no se usa realmente.

Lo que pasa es que la organización de la tabla es poco más complicada. Los primeros 64 bytes son la posición Y de los 64 sprites posibles. Luego vienen 64 bytes que no se usan. Y luego vienen la posición X del sprite 0, y luego su índice de tile, después la posición X del sprite 1, y luego su índice de tile... y así sucesivamente.

#### **5) Activar la imagen.**

Activar y desactivar la imagen es tan sencillo como poner a 1 ó 0 un bit del registro \$1 del VDP, el bit 6.

```

out ($bf), $C6
out ($bf), $81                    ;activar imagen

out ($bf), $86
out ($bf), $81                    ;desactivar imagen

```

#### **Resumen del uso del puertos \$BF**

Siempre escribiremos 2 veces a este puerto, primero el byte bajo y luego el alto. Y lo usaremos para decir donde queremos escribir o leer. Distingiendo 3 posibles casos.

*Escribir a un puerto del VDP*

Byte alto: 10xPPPPPP  
 Byte bajo: DDDDDDDD

*Escribir a VRAM*

Byte alto: 01AAAAAAA  
 Byte bajo: AAAAAAAA

*Escribir a CRAM (paleta)*

Byte alto: 11000000  
 Byte bajo: 000AAAAA

*Leer de VRAM*

Byte alto: 00AAAAAAA  
 Byte bajo: AAAAAAAA

A: Dirección

D: Datos a escribir (en el caso de un puerto de VDP)

x: nada

## DE MOMENTO...

Me queda por introducir las subrutinas, interrupciones y bucles. Eso será para la segunda parte. Intentaré reusarlo cuanto pueda, para poder empezar a programar lo antes posible. Será entonces cuando todo empiece a coger forma, así que tranquilo si aún te parece complicado.

## ENTONCES, QUE CONSOLA ESCOJO?

Ya deberías tener una noción para poder al menos entender que ventajas e inconvenientes tiene cada consola.

Como pequeño resumen;

La **NES** es inferior en cuanto a colores. Sólo permite usar una paleta de 12 colores para todos los sprites, y otra de 12 para el fondo. Pero además, cada sprite sólo puede usar a la vez 3 de los 12 colores de la paleta de sprites. Lo mismo sucede con cada grupo de 2x2 tiles del fondo. La **Master System** tiene una paleta de 15 colores para sprites, y 15 para fondos, sin limitaciones.

La **NES** tiene menos memoria VRAM que la **Master System**, pero dispone de una memoria ROM específica (CHR-ROM) para los gráficos, que permite no tener que cargar en la VRAM más que las Name Table, atributos y sprites, mientras que la Master system tiene que cargar todos los gráficos que quiera usar (como sucede en casi todas las consolas, por cierto).

La PPU de la **NES** es poco más fácil de programar que el VDP de la **Master System**, pero realmente la diferencia no es muy importante. También es cierto que el VDP es más versátil y potente que la PPU; permite varios modos gráficos, y especificar donde almacenar los datos de la VRAM (name table, tiles, sprites), por ejemplo.

La **NES** tiene menos memoria RAM que la **Master System**. Aún así debería ser suficiente.

Yo empezaría por la NES si tienes intención de aprender para más consolas en el futuro. Sino, ve directamente a la que más te interese.

Pero te recomiendo que si no lo tienes claro sigas prestando atención a los ejemplos de ambas consolas del tercer tutorial. Seguiré mostrando como hacer las cosas en ambos sistemas.