Ensamblador para ZX Spectrum
Batalla espacial

Ensamblador para ZX Spectrum Batalla espacial por Juan Antonio Rubio García

Batalla espacial

0x00 Introducción

Batalla espacial es el primer programa que desarrollé en ensamblador para ZX Spectrum, quitándome de esta manera la espina que tenía clavada desde pequeño. Batalla espacial no es más que una prueba de los conocimientos adquiridos siguiendo el curso Curso de Ensamblador Z80 de Compiler Software.

En Batalla espacial los gráficos son de un carácter, se usan UDG (User-Defined Graphic) y el movimiento es carácter a carácter, por lo que las diferencias con respecto a PorompomPong son patentes.

Batalla espacial hace uso de las interrupciones, cosa que en PorompomPong no hace, también hace un mayor uso de las rutinas de la ROM y al usar UDG y RST $10 para pintar, es necesario cambiar de canal para pintar en la parte superior de la pantalla, o en la línea de comandos.

Herramientas necesarias

- Editor de texto, ya sea Notepad, Notepad++, Visual Studio Code o cualquier otro con el que os sintáis cómodos.
- Compilador de ensamblador PASMO: está disponible para Windows, Linux y Mac, y es compatible con Raspberry Pi OS, que es el sistema desde el que estoy redactando el presente tutorial.
- Emulador de ZX Spectrum: aquí tenéis varios para elegir como ZEsarUX, Fuse, Retro Virtual Machine, etc., dependiendo del sistema operativo que uséis. En mi caso vuelvo a optar por ZEsarUX.

Con respecto a PASMO, recomiendo a los usuarios de Windows que incluyan la ruta del ejecutable en la variable de entorno Path. Aquí tenéis un vídeo que muestra como hacerlo en Windows 10.

https://www.youtube.com/watch?v=fyVR0gbqECg

Estructura del tutorial

A diferencia de PorompomPong, Batalla espacial no se desarrolló con idea de realizar un tutorial, por lo que la estructura es distinta, no voy a ir desarrollando, haciendo y deshaciendo como hice en PorompomPong, en esta ocasión ya lo tengo terminado y ahora toca escribir “Cómo se hizo”, aunque si hay cambios en el código con respecto de lo que hay publicado.

En lo primeros pasos vamos a definir la nave, la vamos a mover, luego vamos a abordar los enemigos y el disparo. Una vez realizado lo anterior, vamos a implementar la mecánica del juego, las colisiones, las puntuaciones, las vidas disponibles, etc.

En los últimos pasos vamos a implementar el menú de inicio, la selección de controles, inicio y fin de partida, marcadores, los efectos de sonido y, en general, a decorar un poco el resultado final.

Lo último será añadir una pantalla de carga, a ver que tal nos sale esta vez.

Entorno de trabajo

Esta vez no voy a trabajar bajo Windows, voy a usar el reciente regalo que me han hecho, lo que algunos llaman el Spectrum del siglo XXI, una Raspberry Pi 400.

El emulador que voy a usar es ZEsarUX, el compilador PASMO y el editor Visual Studio Code, por lo que en este aspecto no cambio nada con respecto a PorompomPong.

En esta ocasión voy a mostrar algo que no mostré en el tutorial anterior, la forma de depurar con ZEsarUX, cosa que espero que os sea de utilidad, aunque lo dejaremos para el final.

El resultado final

Batalla espacial es un sencillo juego mata marcianos, compatible con los modelos de ZX Spectrum 16K, 48K, +2 y +3, manejable con teclado, joystick Sinclair y Kempstom, y que consta de treinta niveles.

Decir que consta de treinta niveles quizá sea algo pretencioso, ya que la mecánica del juego no cambia, aunque lo que sí cambia son los enemigos, habiendo un total de treinta distintos, uno por cada nivel.

El movimiento de nuestra nave es horizontal, por lo que solo es necesario un gráfico, mientras que el movimiento de los enemigos es diagonal, siendo necesarios cuatro gráficos (arriba-derecha, arriba-izquierda, abajo-derecha, abajo-izquierda). El disparo de nuestra nave consta también de un solo gráfico, mientras que la explosión de la nave, cuando nos matan, consta de cuatro gráficos, con lo que haremos una pequeña animación cuando nos maten.

Vamos a definir otros ocho gráficos para el marco de la pantalla, y un gráfico más en blanco para borrar gráficos impresos en pantalla.

Conclusión

Poco más se puede añadir, espero que este tutorial os sirva para aprender y sobre todo que os divierta.

Gran parte de lo que vamos a ver en Batalla espacial lo expliqué en PorompomPong, motivo por el cual en ocasiones no pararé a explicar muchas instrucciones.

En la próxima entrega empezamos definiendo los gráficos y practicando la conversión hexadecimal/binario.

0x01 Definición de gráficos

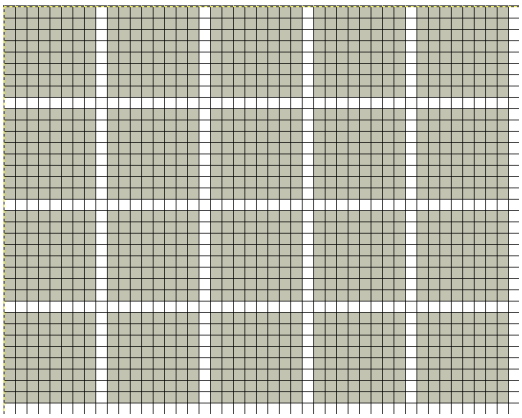
En este capítulo vamos a definir todos los gráficos que vamos a usar en Batalla espacial, aprovechando para practicar la conversión hexadecimal/binario, por lo que vamos a hacer nuestra primera práctica.

Ya comenté en el capítulo anterior que estoy usando una Raspberry Pi 400, y eso me impone alguna limitación, como que no tengo disponible el programa que suelo usar para diseñar los gráficos para ZX Spectrum, así que he hecho dos plantillas para usar con GIMP, y que simula el área de dibujo de ZX Paintbrush, que es el programa que uso en Windows.

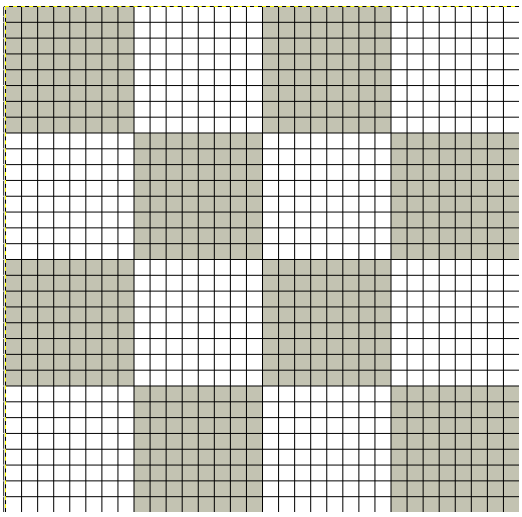
Las plantillas que he preparado son las siguientes:

- 8x8Template: para diseñar gráficos de 8x8 píxeles.
- 256x192Template: tapiz con el tamaño de la pantalla del ZX Spectrum.

Para batalla espacial vamos a usar 8x8Template, os pondré la imagen de los gráficos, los códigos hexadecimales de los mismos, y vuestra labor consistirá en convertir, de cabeza, esos códigos hexadecimales para dibujar los gráficos en las plantillas.



Sección de plantilla 8x8



Sección de plantilla 256x192

Conversión hexadecimal/binario

Aunque en un primer momento pueda resultar complicado hacer la conversión de un número hexadecimal a binario, y viceversa, la realidad es que es muy sencillo y prácticamente directa, necesitamos saber el valor de cada bit (pixel) en bloques de cuatro, lo que nos da un valor comprendido entre 0 y F, que es el valor que se puede representar con cada dígito hexadecimal.

En un byte, cada bit a uno tiene un valor específico, siendo los siguientes:

Bit	7	6	5	4	3	2	1	0
Valor	128	64	32	16	8	4	2	1

Cuando hacemos la conversión hexadecimal/binario, dividimos el byte en dos bloques de cuatro bits (nibble), lo que resulta en un rango de valores entre 0 y F (8 + 4 + 2 + 1 = 15 = F). De esta manera, para convertir de binario a hexadecimal, tan solo hay que sumar el valor de los bits a 1 de cada nibble, lo cual nos da el valor en hexadecimal.

Suponed que tenemos el siguiente valor en binario:

01011001

Si sumamos los valores de los nibbles, el resultado sería:

0 + 4 + 0 + 1 = 5 8 + 0 + 0 + 1 = 9

Resultando que 01011001 en hexadecimal es 59.

En hexadecimal, un byte se representa con dos dígitos. Pero, ¿qué pasa si el valor de algunos de los nibbles es mayor de 9? Veamos un ejemplo:

11011011 = 8 + 4 + 0 + 1 = 13 y 8 + 0 + 2 + 1 = 11

¿Cómo representamos 13 y 11 con solo dos dígitos? En hexadecimal los valores de 10 a 15 se representan con letras, usando la siguiente nomenclatura:

Decimal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Por lo que, en el ejemplo anterior, el valor hexadecimal de 11011011 es DB.

Practicado la conversión hexadecimal/binario

Una buena forma de aprender es con la práctica, y eso es lo que propongo a continuación; vamos a ver la definición de cada uno de los UDG que vamos a usar (los valores hexadecimales) y vamos a dibujarlos haciendo la conversión de hexadecimal a binario.

Al trabajar con nibble (4 bits), la tabla de conversión para un byte sería la siguiente:

Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1

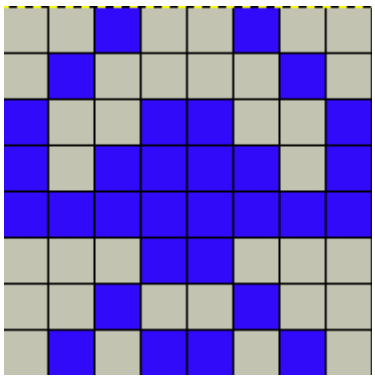
Vamos a crear la carpeta Paso01, y dentro de ella el archivo Var.asm. Teniendo esta tabla a mano, vamos a dibujar la nave, cuya definición en hexadecimal (que vamos a copiar en el archivo creado) es la siguiente:

```
udgsCommon:

db $24, $42, $99, $bd, $ff, $18, $24, $5a     ; $90 Nave
```

Vamos a hacer la conversión a binario.

Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$24			X			X		
$42		X					X	
$99	X			X	X			X
$bd	X		X	X	X	X		X
$ff	X	X	X	X	X	X	X	X
$18				X	X			
$24			X			X		
$5a		X		X	X		X	

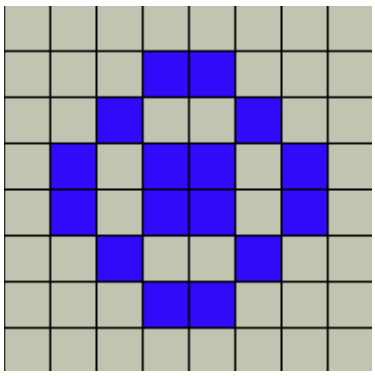
Si trasladáis esta conversión a ZX Paintbrush, o a las plantillas que os he dejado, el resultado debe ser el siguiente, aquí está nuestra nave:



Vamos a seguir practicando, haciendo las conversiones para el disparo y la animación de la explosión de la nave. Es importante que intentéis trasladar vosotros de hexadecimal a binario, y de ahí al gráfico.

```
db $00, $18, $24, $5a, $5a, $24, $18, $00; $91 Disparo
```

Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$00								
$18				X	X			
$24			X			X		
$5a		X		X	X		X	
$5a		X		X	X		X	
$24			X			X		
$18				X	X			
$00								



```
db $00, $00, $00, $00, $24, $5a, $24, $18     ; $92 Explosión 1
```

Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$00								
$00								
$00								
$00								
$24			X			X		
$5a		X		X	X		X	
$24			X			X		
$18				X	X			

```
db $00, $00, $00, $14, $2a, $34, $24, $18     ; $93 Explosión 2
```

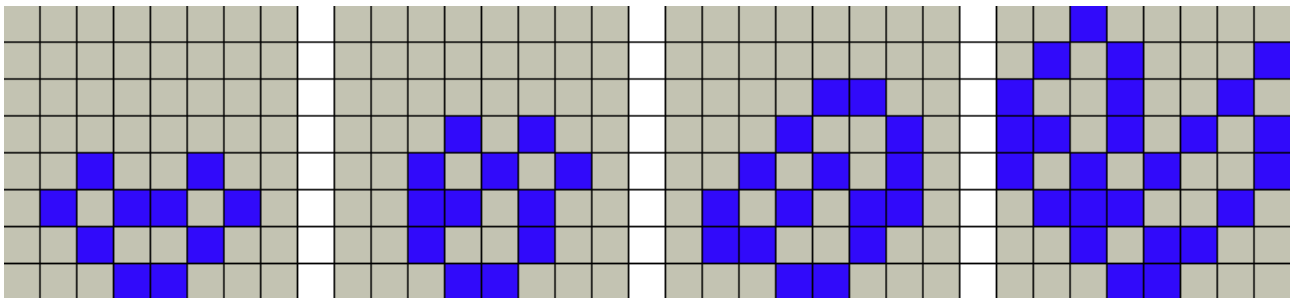
Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$00								
$00								
$00								
$14				X		X		
$2a			X		X		X	
$34			X	X		X		
$24			X			X		
$18				X	X			

```
db $00, $00, $0c, $12, $2a, $56, $64, $18     ; $94 Explosión 3
```

Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$00								
$00								
$0c					X	X		
$12				X			X	
$2a			X		X		X	
$56		X		X		X	X	
$64		X	X			X		
$18				X	X			

```
db $20, $51, $92, $d5, $a9, $72, $2c, $18     ; $95 Explosión 4
```

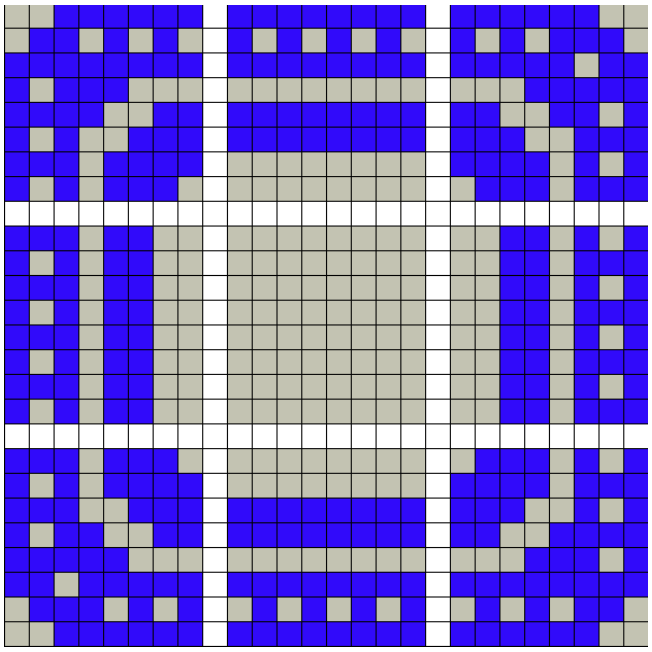
Byte	7	6	5	4	3	2	1	0
Valor	8	4	2	1	8	4	2	1
$20			X					
$51		X		X				X
$92	X			X			X	
$d5	X	X		X		X		X
$a9	X		X		X			X
$72		X	X	X			X	
$2c			X		X	X		
$18				X	X			



A partir de aquí solo voy a poner la definición hexadecimal y la imagen del aspecto final de cada UDG.

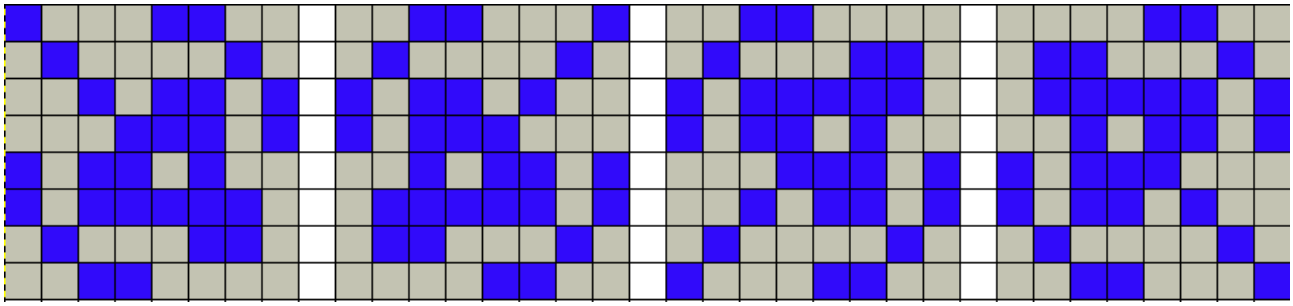
Quizá os preguntéis que significa el número que hay en cada comentario de cada definición; es el código del carácter que estamos redefiniendo, ese el código de carácter que mandaremos a imprimir para pintar el gráfico. No os preocupéis si ahora no lo entendéis, más adelante lo veréis mucho más claro.

```
db $3f, $6a, $ff, $b8, $f3, $a7, $ef, $ae     ; $96 Esquina superior izquierda
db $ff, $aa, $ff, $00, $ff, $ff, $00, $00     ; $97 Horizontal superior
db $fc, $ae, $fb, $1f, $cd, $e7, $f5, $77     ; $98 Esquina superior derecha
db $ec, $ac, $ec, $ac, $ec, $ac, $ec, $ac     ; $99 Lateral izquierda
db $35, $37, $35, $37, $35, $37, $35, $37     ; $9a Lateral derecha
db $ee, $af, $e7, $b3, $f8, $df, $75, $3f     ; $9b Esquina inferior izquierda
db $00, $00, $ff, $ff, $00, $ff, $55, $ff     ; $9c Horizontal inferior
db $75, $f7, $e5, $cf, $1d, $ff, $56, $fc     ; $9d Esquina inferior derecha
db $00, $00, $00, $00, $00, $00, $00, $00     ; $9e Blanco
```

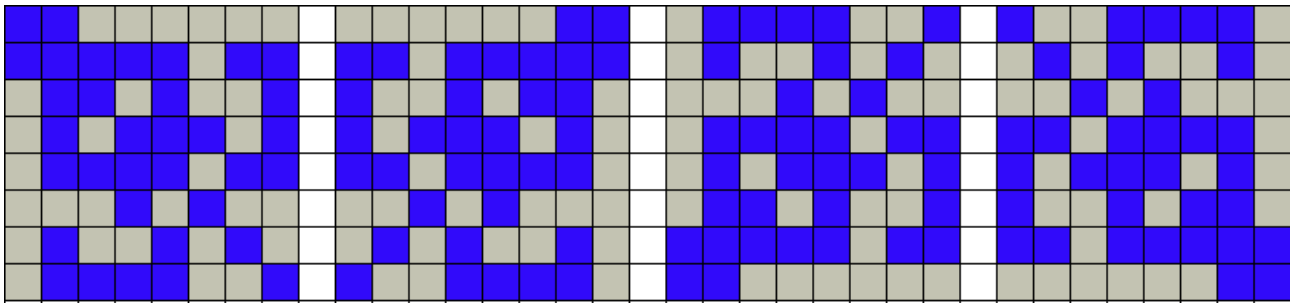


```
udgsEnemiesLevel1:
```

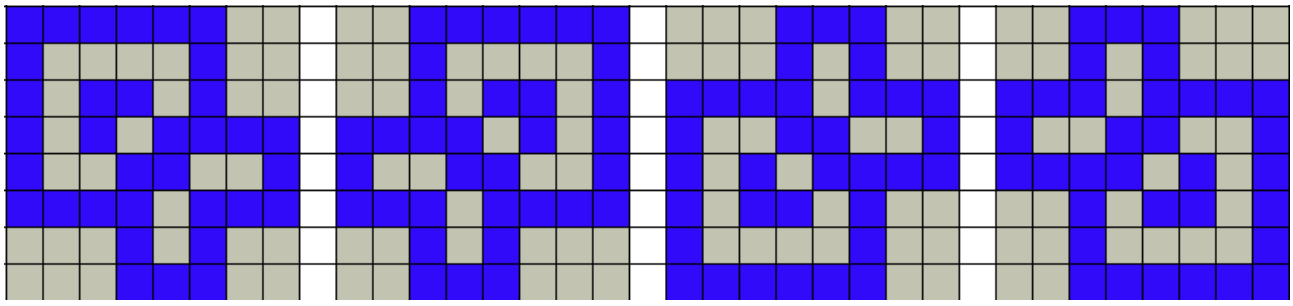
```
db $8c, $42, $2d, $1d, $b4, $be, $46, $30     ; $9f Left/Up
db $31, $42, $b4, $b8, $2d, $7d, $62, $0c     ; $a0 Rigth/Up
db $30, $46, $be, $b4, $1d, $2d, $42, $8c     ; $a1 Left/Down
db $0c, $62, $7d, $2d, $b8, $b4, $42, $31     ; $a2 Rigth/Down
```



```
udgsEnemiesLevel2:
db $c0, $fb, $69, $5d, $7b, $14, $4a, $79     ; $9f Left/Up
db $03, $df, $96, $ba, $de, $28, $52, $9e     ; $a0 Rigth/Up
db $79, $4a, $14, $7b, $5d, $69, $fb, $c0     ; $a1 Left/Down
db $9e, $52, $28, $de, $ba, $96, $df, $03     ; $a2 Rigth/Down
```

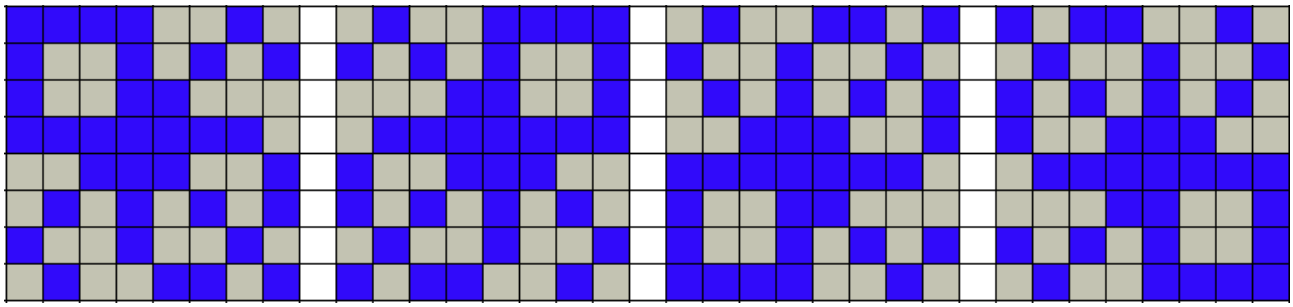


```
udgsEnemiesLevel3:
db $fc, $84, $b4, $af, $99, $f7, $14, $1c     ; $9f Left/Up
db $3f, $21, $2d, $f5, $99, $ef, $28, $38     ; $a0 Rigth/Up
db $1c, $14, $f7, $99, $af, $b4, $84, $fc     ; $a1 Left/Down
db $38, $28, $ef, $99, $f5, $2d, $21, $3f     ; $a2 Rigth/Down
```

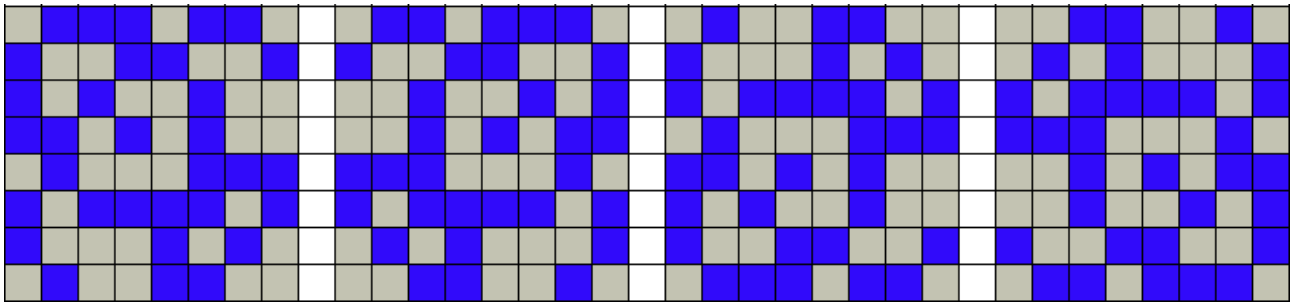


```
udgsEnemiesLevel4:
```

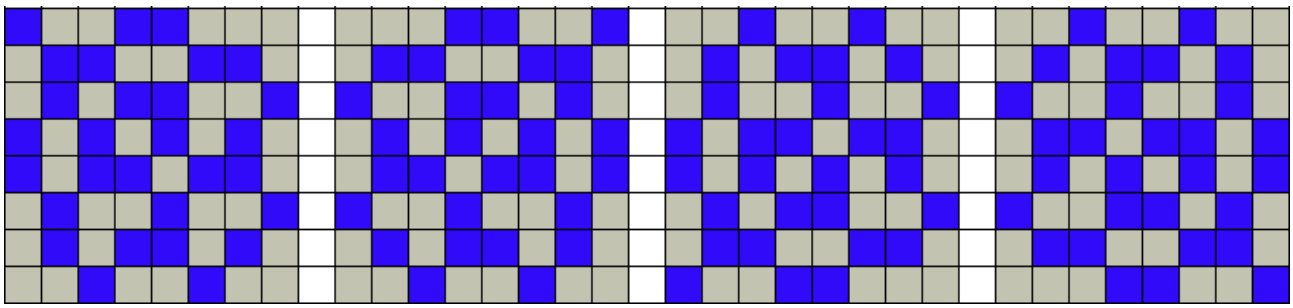
```
db $f2, $95, $98, $fe, $39, $55, $92, $4d     ; $9f Left/Up
db $4f, $a9, $19, $7f, $9c, $aa, $49, $b2     ; $a0 Rigth/Up
db $4d, $92, $55, $39, $fe, $98, $95, $f2     ; $a1 Left/Down
db $b2, $49, $aa, $9c, $7f, $19, $a9, $4f     ; $a2 Rigth/Down
```



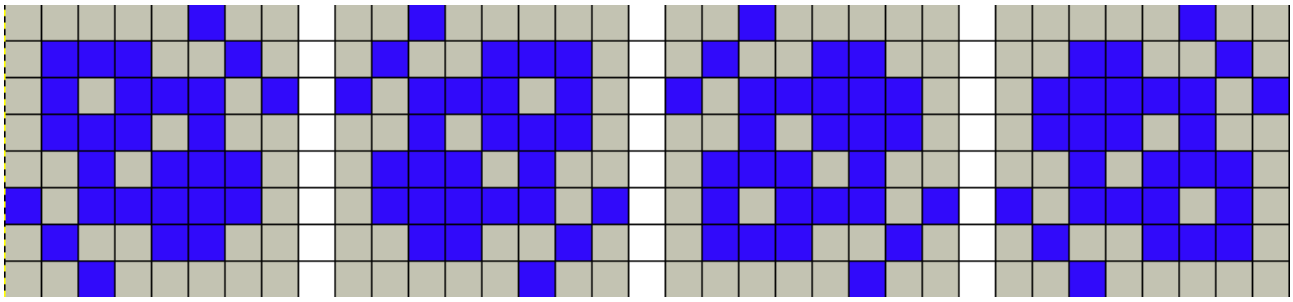
```
udgsEnemiesLevel5:
db $76, $99, $a4, $d4, $47, $bd, $8a, $4c     ; $9f Left/Up
db $6e, $99, $25, $2b, $e2, $bd, $51, $32     ; $a0 Rigth/Up
db $4c, $8a, $bd, $47, $d4, $a4, $99, $76     ; $a1 Left/Down
db $32, $51, $bd, $e2, $2b, $25, $99, $6e     ; $a2 Rigth/Down
```



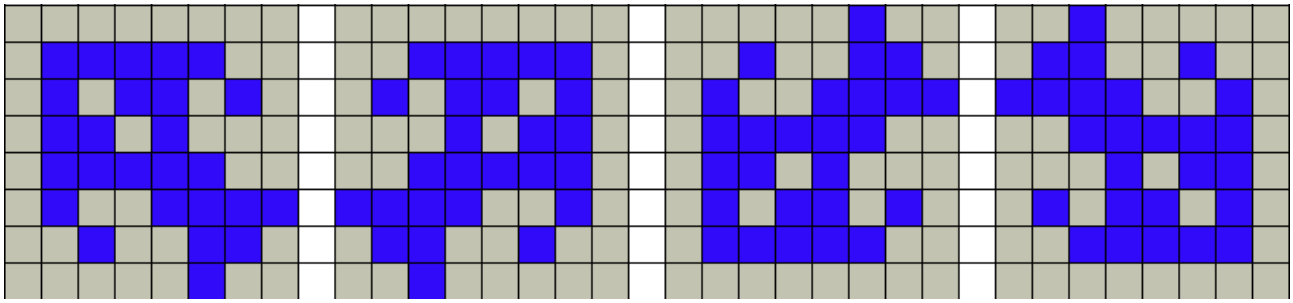
```
udgsEnemiesLevel6:
db $98, $66, $59, $aa, $b6, $49, $5a, $24     ; $9f Left/Up
db $19, $66, $9a, $55, $6d, $92, $5a, $24     ; $a0 Rigth/Up
db $24, $5a, $49, $b6, $aa, $59, $66, $98     ; $a1 Left/Down
db $24, $5a, $92, $6d, $55, $9a, $66, $19     ; $a2 Rigth/Down
```



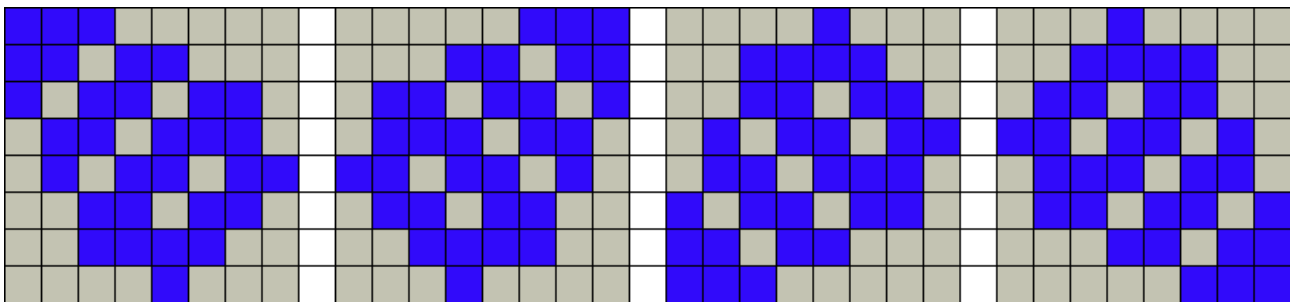
```
udgsEnemiesLevel7:
db $04, $72, $5d, $74, $2e, $be, $4c, $20     ; $9f Left/Up
db $20, $4e, $ba, $2e, $74, $7d, $32, $04     ; $a0 Rigth/Up
db $20, $4c, $be, $2e, $74, $5d, $72, $04     ; $a1 Left/Down
db $04, $32, $7d, $74, $2e, $ba, $4e, $20     ; $a2 Rigth/Down
```



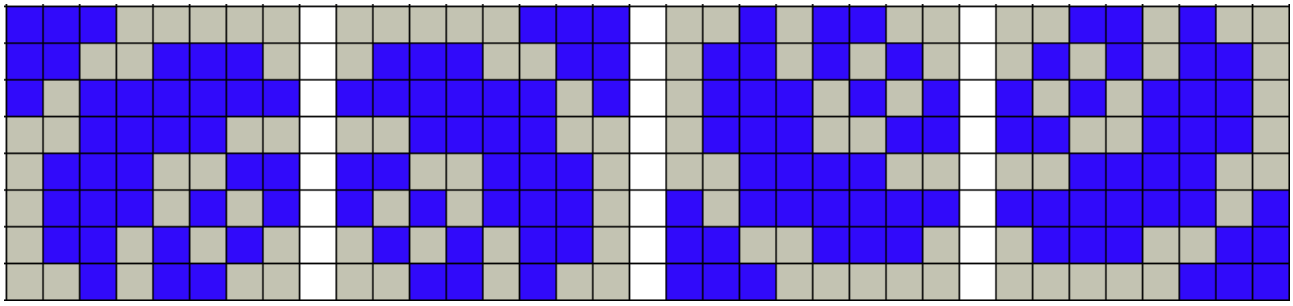
```
udgsEnemiesLevel8:
db $00, $7c, $5a, $68, $7c, $4f, $26, $04     ; $9f Left/Up
db $00, $3e, $5a, $16, $3e, $f2, $64, $20     ; $a0 Rigth/Up
db $04, $26, $4f, $7c, $68, $5a, $7c, $00     ; $a1 Left/Down
db $20, $64, $f2, $3e, $16, $5a, $3e, $00     ; $a2 Rigth/Down
```



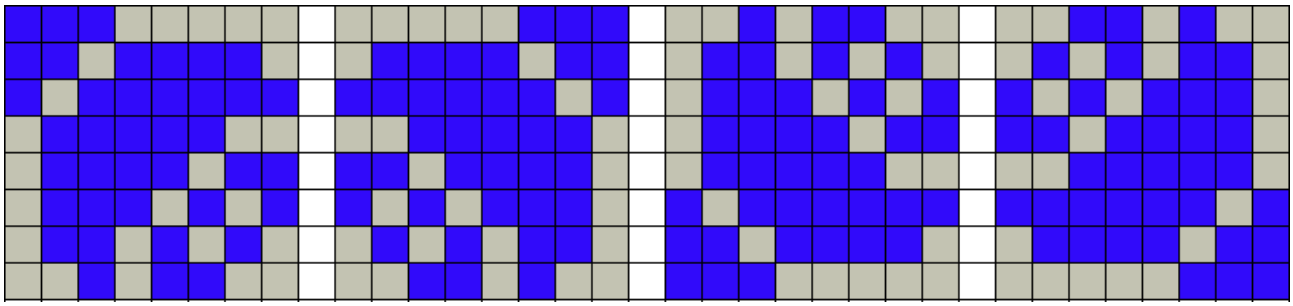
```
udgsEnemiesLevel9:
db $e0, $d8, $b6, $6e, $5b, $36, $3c, $08     ; $9f Left/Up
db $07, $1b, $6d, $76, $da, $6c, $3c, $10     ; $a0 Rigth/Up
db $08, $3c, $36, $5b, $6e, $b6, $d8, $e0     ; $a1 Left/Down
db $10, $3c, $6c, $da, $76, $6d, $1b, $07     ; $a2 Rigth/Down
```



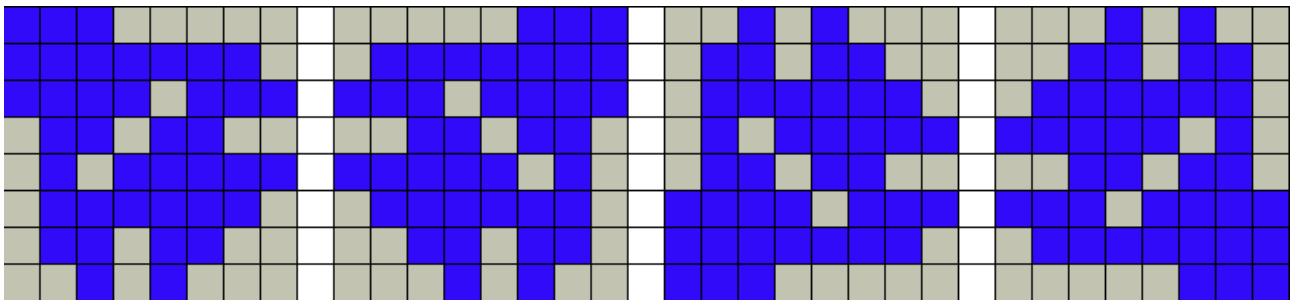
```
udgsEnemiesLevel10:
db $e0, $ce, $bf, $3c, $73, $75, $6a, $2c     ; $9f Left/Up
db $07, $73, $fd, $3c, $ce, $ae, $56, $34     ; $a0 Rigth/Up
db $2c, $6a, $75, $73, $3c, $bf, $ce, $e0     ; $a1 Left/Down
db $34, $56, $ae, $ce, $3c, $fd, $73, $07     ; $a2 Rigth/Down
```



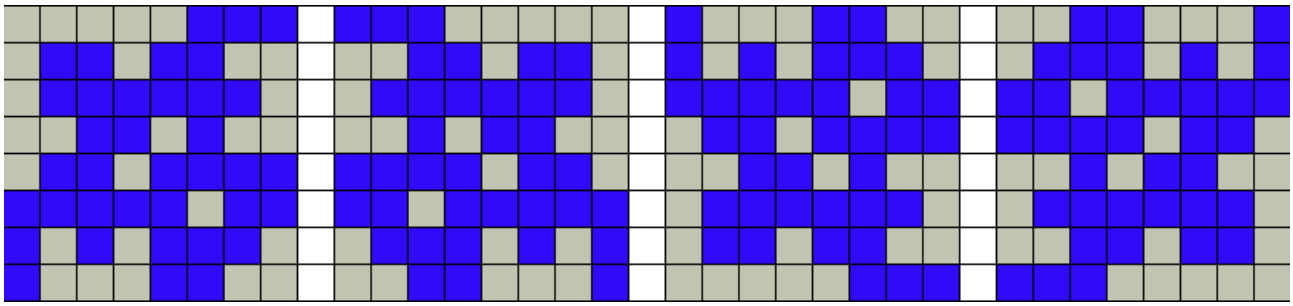
```
udgsEnemiesLevel11:
db $e0, $de, $bf, $7c, $7b, $75, $6a, $2c     ; $9f Left/Up
db $07, $7b, $fd, $3e, $de, $ae, $56, $34     ; $a0 Rigth/Up
db $2c, $6a, $75, $7b, $7c, $bf, $de, $e0     ; $a1 Left/Down
db $34, $56, $ae, $de, $3e, $fd, $7b, $07     ; $a2 Rigth/Down
```



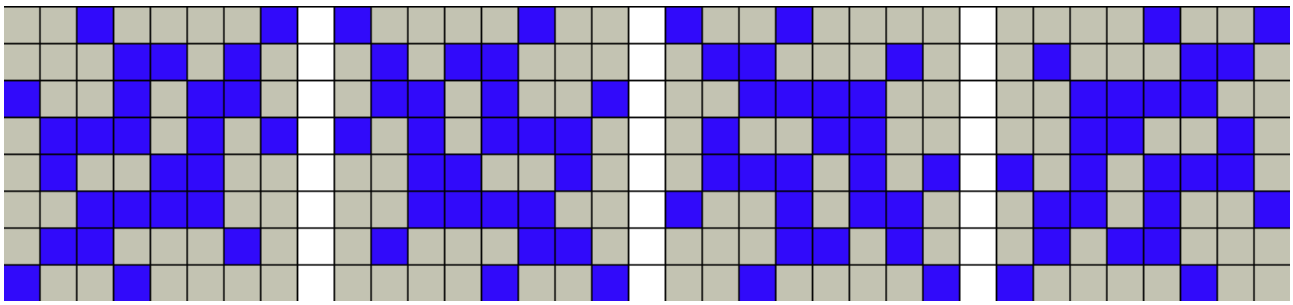
```
udgsEnemiesLevel12:
db $e0, $fe, $f7, $6c, $5f, $7e, $6c, $28     ; $9f Left/Up
db $07, $7f, $ef, $36, $fa, $7e, $36, $14     ; $a0 Rigth/Up
db $28, $6c, $7e, $5f, $6c, $f7, $fe, $e0     ; $a1 Left/Down
db $14, $36, $7e, $fa, $36, $ef, $7f, $07     ; $a2 Rigth/Down
```



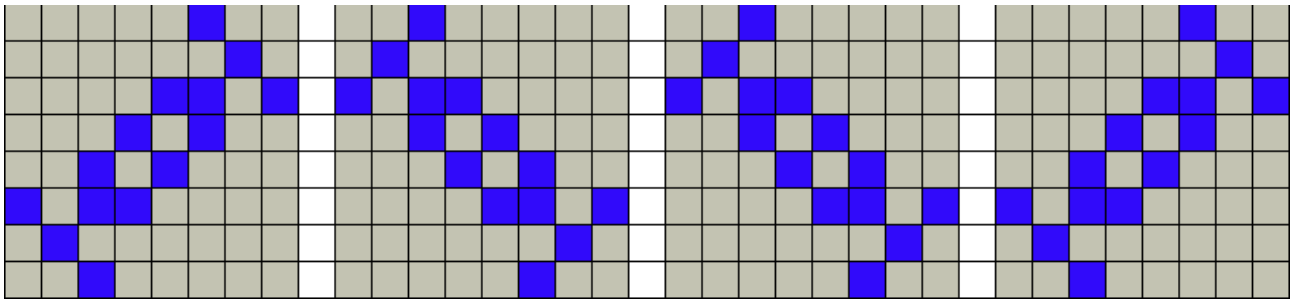
```
udgsEnemiesLevel13:
db $07, $6c, $7e, $34, $6f, $fb, $ae, $8c     ; $9f Left/Up
db $e0, $36, $7e, $2c, $f6, $df, $75, $31     ; $a0 Rigth/Up
db $8c, $ae, $fb, $6f, $34, $7e, $6c, $07     ; $a1 Left/Down
db $31, $75, $df, $f6, $2c, $7e, $36, $e0     ; $a2 Rigth/Down
```



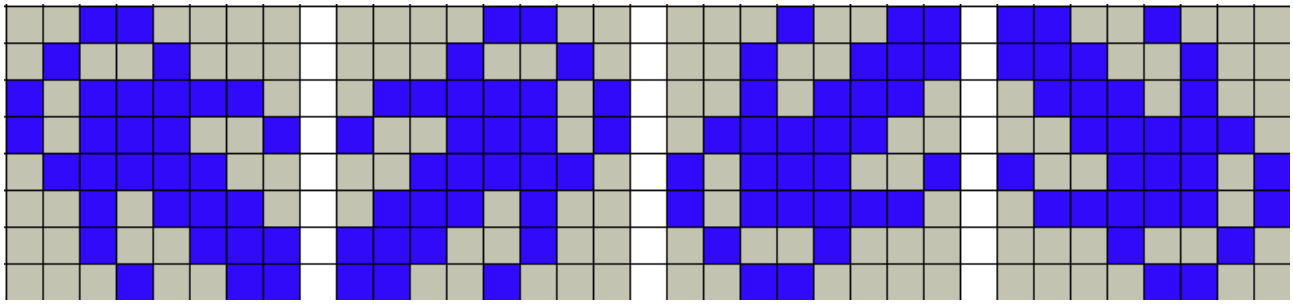
```
udgsEnemiesLevel14:
db $21, $1a, $96, $75, $4c, $3c, $62, $90     ; $9f Left/Up
db $84, $58, $69, $ae, $32, $3c, $46, $09     ; $a0 Rigth/Up
db $90, $62, $3c, $4c, $75, $96, $1a, $21     ; $a1 Left/Down
db $09, $46, $3c, $32, $ae, $69, $58, $84     ; $a2 Rigth/Down
```



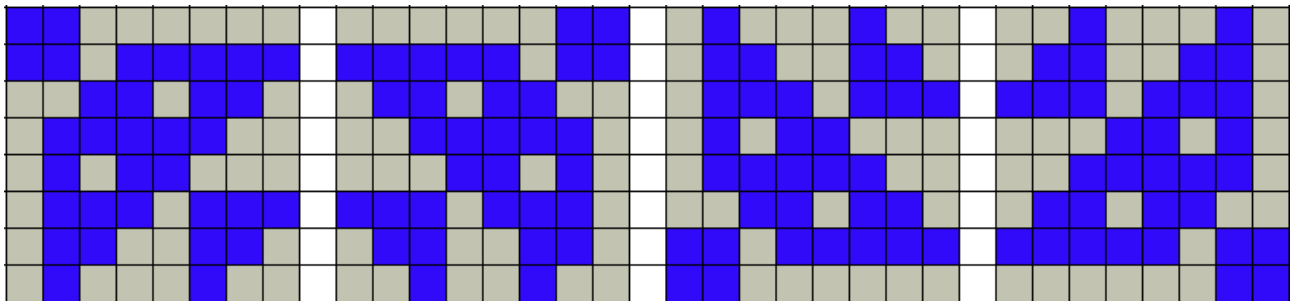
```
udgsEnemiesLevel15:
db $04, $02, $0d, $14, $28, $b0, $40, $20     ; $9f Left/Up
db $20, $40, $b0, $28, $14, $0d, $02, $04     ; $a0 Rigth/Up
db $20, $40, $b0, $28, $14, $0d, $02, $04     ; $a1 Left/Down
db $04, $02, $0d, $14, $28, $b0, $40, $20     ; $a2 Rigth/Down
```



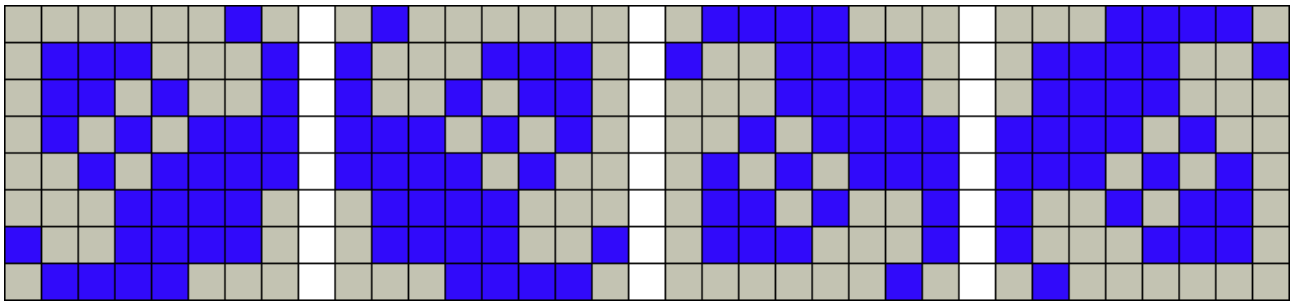
```
udgsEnemiesLevel16:
db $30, $48, $be, $b9, $7c, $2e, $27, $13     ; $9f Left/Up
db $0c, $12, $7d, $9d, $3e, $74, $e4, $c8     ; $a0 Rigth/Up
db $13, $27, $2e, $7c, $b9, $be, $48, $30     ; $a1 Left/Down
db $c8, $e4, $74, $3e, $9d, $7d, $12, $0c     ; $a2 Rigth/Down
```



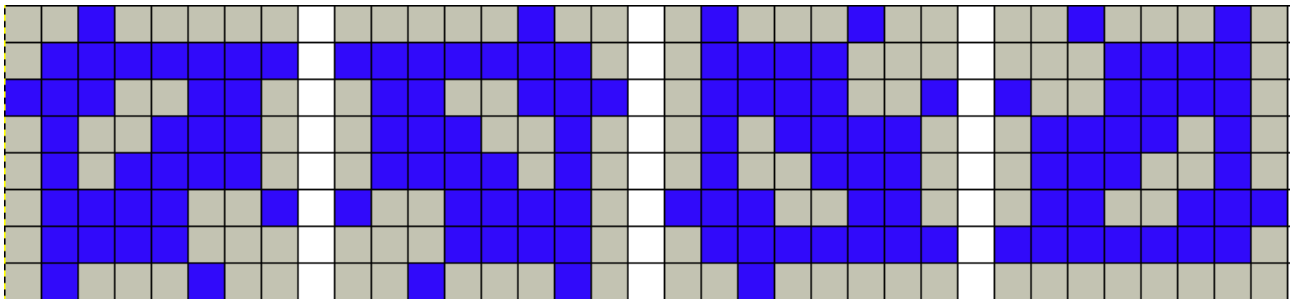
```
udgsEnemiesLevel17:
db $c0, $df, $36, $7c, $58, $77, $66, $44     ; $9f Left/Up
db $03, $fb, $6c, $3e, $1a, $ee, $66, $22     ; $a0 Rigth/Up
db $44, $66, $77, $58, $7c, $36, $df, $c0     ; $a1 Left/Down
db $22, $66, $ee, $1a, $3e, $6c, $fb, $03     ; $a2 Rigth/Down
```



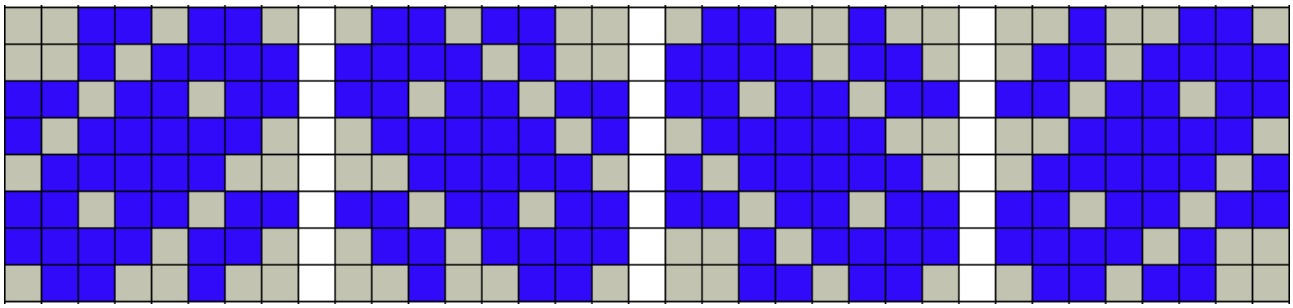
```
udgsEnemiesLevel18:
db $02, $71, $69, $57, $2f, $1e, $9e, $78     ; $9f Left/Up
db $40, $8e, $96, $ea, $f4, $78, $79, $1e     ; $a0 Rigth/Up
db $78, $9e, $1e, $2f, $57, $69, $71, $02     ; $a1 Left/Down
db $1e, $79, $78, $f4, $ea, $96, $8e, $40     ; $a2 Rigth/Down
```



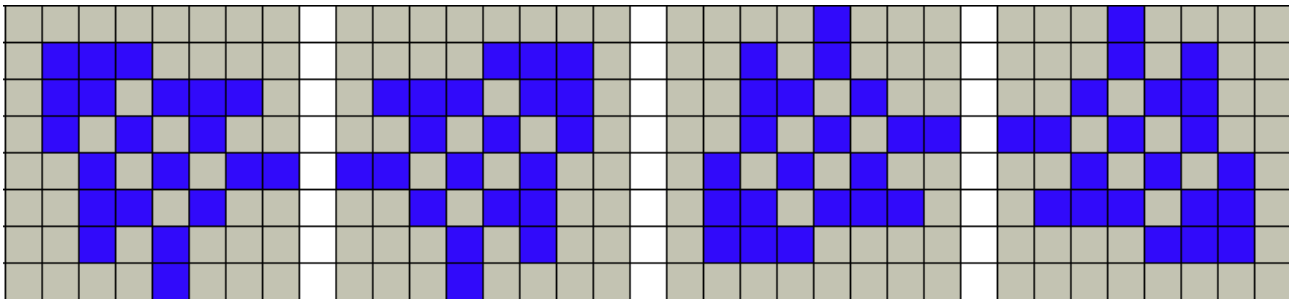
```
udgsEnemiesLevel19:
db $20, $7f, $e6, $4e, $5e, $79, $78, $44     ; $9f Left/Up
db $04, $fe, $67, $72, $7a, $9e, $1e, $22     ; $a0 Rigth/Up
db $44, $78, $79, $5e, $4e, $e6, $7f, $20     ; $a1 Left/Down
db $22, $1e, $9e, $7a, $72, $67, $fe, $04     ; $a2 Rigth/Down
```



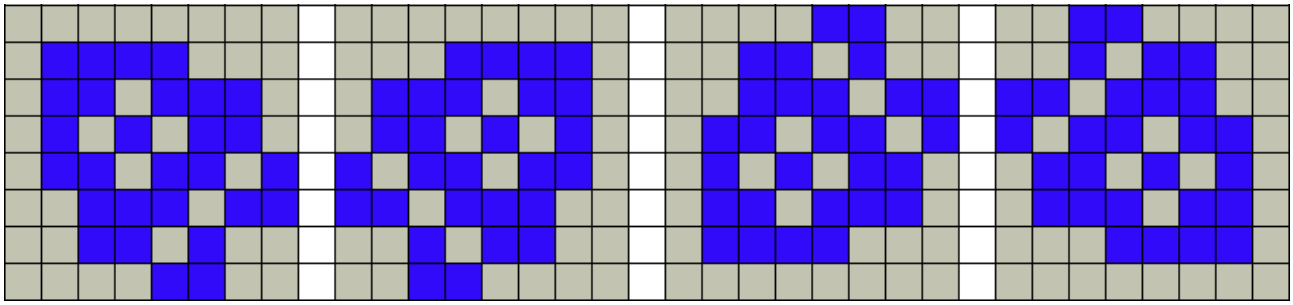
```
udgsEnemiesLevel20:
db $36, $2f, $db, $be, $7c, $db, $f6, $64     ; $9f Left/Up
db $6c, $f4, $db, $7d, $3e, $db, $6f, $26     ; $a0 Rigth/Up
db $64, $f6, $db, $7c, $be, $db, $2f, $36     ; $a1 Left/Down
db $26, $6f, $db, $3e, $7d, $db, $f4, $6c     ; $a2 Rigth/Down
```



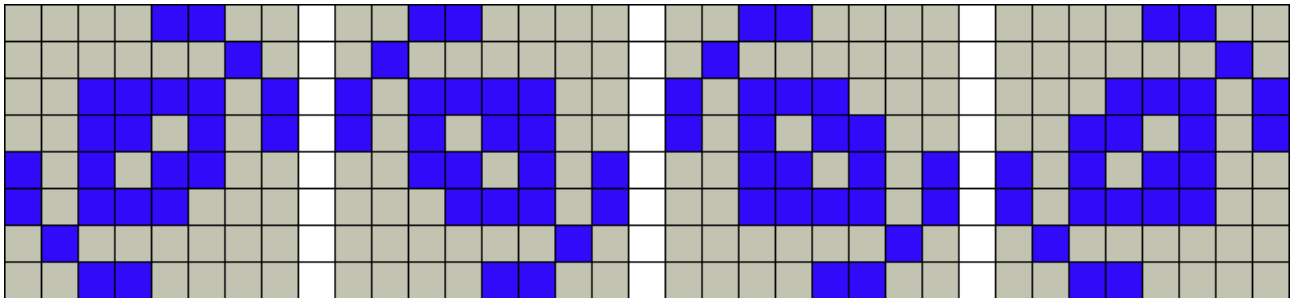
```
udgsEnemiesLevel21:
db $00, $70, $6e, $54, $2b, $34, $28, $08     ; $9f Left/Up
db $00, $0e, $76, $2a, $d4, $2c, $14, $10     ; $a0 Rigth/Up
db $08, $28, $34, $2b, $54, $6e, $70, $00     ; $a1 Left/Down
db $10, $14, $2c, $d4, $2a, $76, $0e, $00     ; $a2 Rigth/Down
```



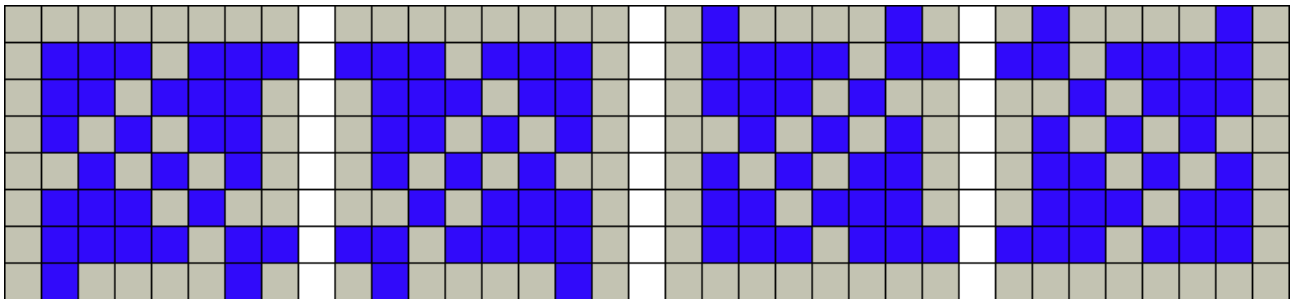
```
udgsEnemiesLevel22:
db $00, $78, $6e, $56, $6d, $3b, $34, $0c     ; $9f Left/Up
db $00, $1e, $76, $6a, $b6, $dc, $2c, $30     ; $a0 Rigth/Up
db $0c, $34, $3b, $6d, $56, $6e, $78, $00     ; $a1 Left/Down
db $30, $2c, $dc, $b6, $6a, $76, $1e, $00     ; $a2 Rigth/Down
```



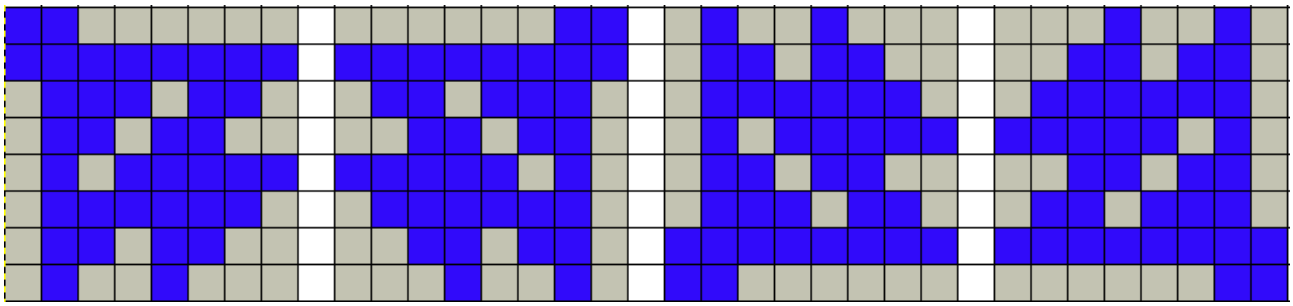
```
udgsEnemiesLevel23:
db $0c, $02, $3d, $35, $ac, $b8, $40, $30     ; $9f Left/Up
db $30, $40, $bc, $ac, $35, $1d, $02, $0c     ; $a0 Rigth/Up
db $30, $40, $b8, $ac, $35, $3d, $02, $0c     ; $a1 Left/Down
db $0c, $02, $1d, $35, $ac, $bc, $40, $30     ; $a2 Rigth/Down
```



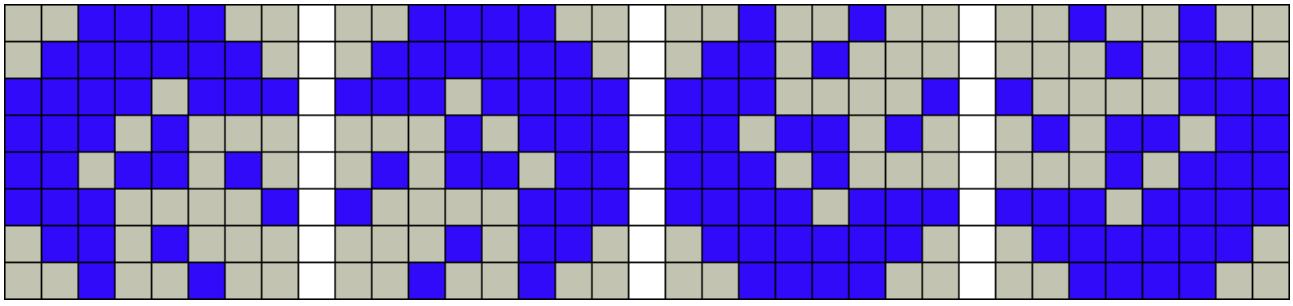
```
udgsEnemiesLevel24:
db $00, $77, $6e, $56, $2a, $74, $7b, $42     ; $9f Left/Up
db $00, $ee, $76, $6a, $54, $2e, $de, $42     ; $a0 Rigth/Up
db $42, $7b, $74, $2a, $56, $6e, $77, $00     ; $a1 Left/Down
db $42, $de, $2e, $54, $6a, $76, $ee, $00     ; $a2 Rigth/Down
```



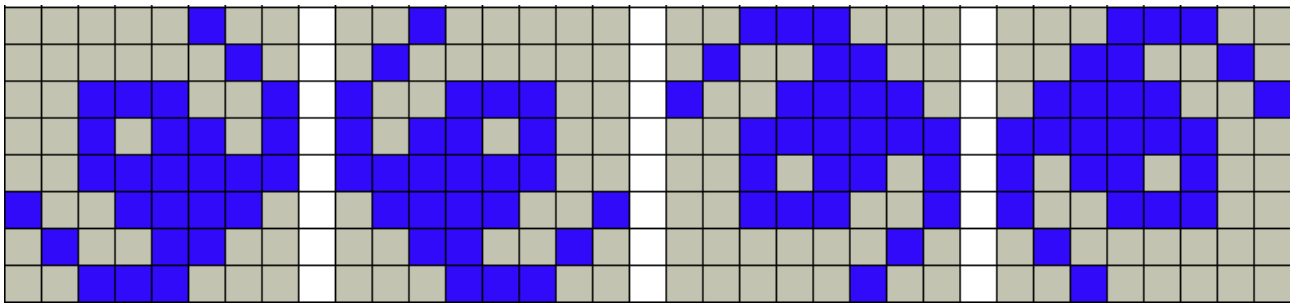
```
udgsEnemiesLevel25:
db $c0, $ff, $76, $6c, $5f, $7e, $6c, $48     ; $9f Left/Up
db $03, $ff, $6e, $36, $fa, $7e, $36, $12     ; $a0 Rigth/Up
db $48, $6c, $7e, $5f, $6c, $76, $ff, $c0     ; $a1 Left/Down
db $12, $36, $7e, $fa, $36, $6e, $ff, $03     ; $a2 Rigth/Down
```



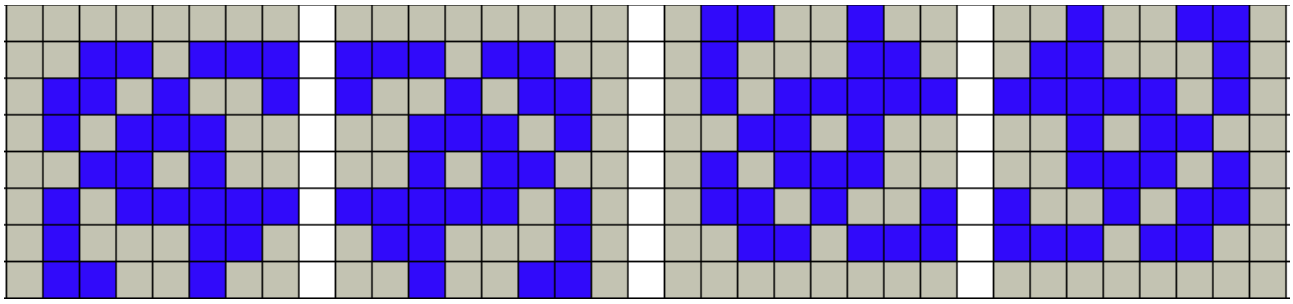
```
udgsEnemiesLevel26:
db $3c, $7e, $f7, $e8, $da, $e1, $68, $24     ; $9f Left/Up
db $3c, $7e, $ef, $17, $5b, $87, $16, $24     ; $a0 Rigth/Up
db $24, $68, $e1, $da, $e8, $f7, $7e, $3c     ; $a1 Left/Down
db $24, $16, $87, $5b, $17, $ef, $7e, $3c     ; $a2 Rigth/Down
```



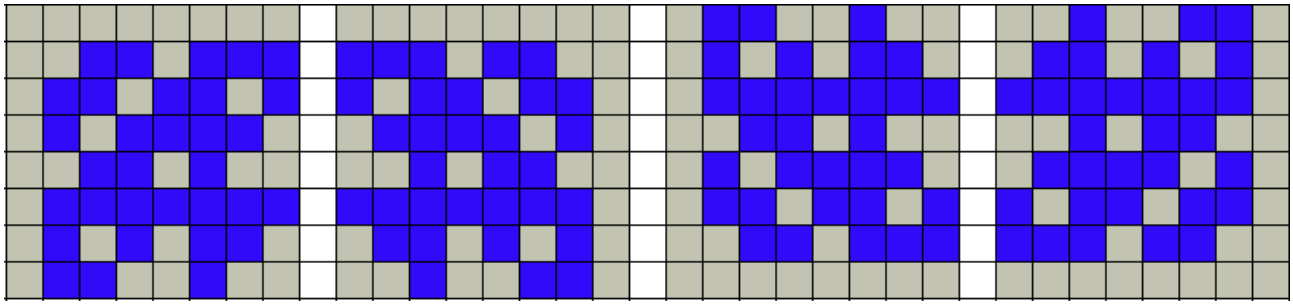
```
udgsEnemiesLevel27:
db $04, $02, $39, $2d, $3f, $9e, $4c, $38     ; $9f Left/Up
db $20, $40, $9c, $b4, $fc, $79, $32, $1c     ; $a0 Rigth/Up
db $38, $4c, $9e, $3f, $2d, $39, $02, $04     ; $a1 Left/Down
db $1c, $32, $79, $fc, $b4, $9c, $40, $20     ; $a2 Rigth/Down
```



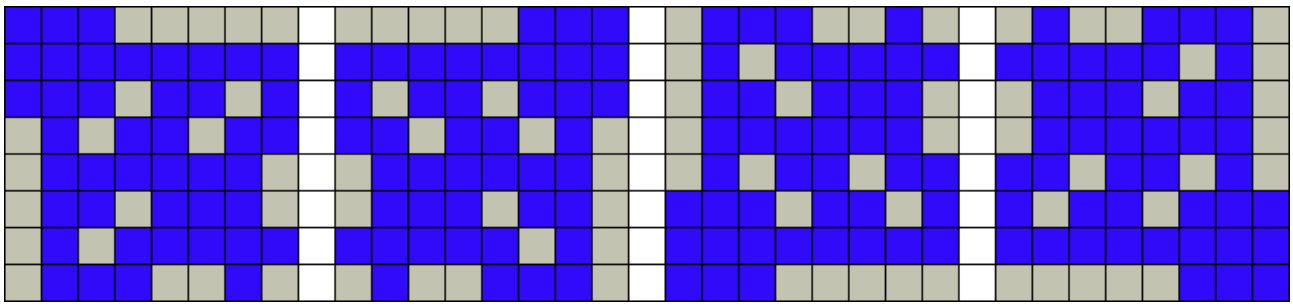
```
udgsEnemiesLevel28:
db $00, $37, $69, $5c, $34, $5f, $46, $64     ; $9f Left/Up
db $00, $ec, $96, $3a, $2c, $fa, $62, $26     ; $a0 Rigth/Up
db $64, $46, $5f, $34, $5c, $69, $37, $00     ; $a1 Left/Down
db $26, $62, $fa, $2c, $3a, $96, $ec, $00     ; $a2 Rigth/Down
```



```
udgsEnemiesLevel29:
db $00, $37, $6d, $5e, $34, $7f, $56, $64     ; $9f Left/Up
db $00, $ec, $b6, $7a, $2c, $fe, $6a, $26     ; $a0 Rigth/Up
db $64, $56, $7f, $34, $5e, $6d, $37, $00     ; $a1 Left/Down
db $26, $6a, $fe, $2c, $7a, $b6, $ec, $00     ; $a2 Rigth/Down
```



```
udgsEnemiesLevel30:
db $e0, $ff, $ed, $5b, $7e, $6e, $5f, $72     ; $9f Left/Up
db $07, $ff, $b7, $da, $7e, $76, $fa, $4e     ; $a0 Rigth/Up
db $72, $5f, $6e, $7e, $5b, $ed, $ff, $e0     ; $a1 Left/Down
db $4e, $fa, $76, $7e, $da, $b7, $ff, $07     ; $a2 Rigth/Down
```



Conclusión

Con esto ya tenemos definidos los gráficos que vamos a usar: tenemos la nave, el disparo, la explosión de la nave, el marco de la pantalla y los enemigos.

Podéis observar que todos los enemigos tienen el mismo código de carácter, lo cual es debido a que hay un número limitado de caracteres para usar como UDG. No os preocupéis, más adelante veremos una forma de salvar esta limitación.

Vuelvo a insistir en la importancia de practicar la conversión hexadecimal / binario, así que no lo dejéis para otro día, es un ejercicio sencillo.

En el próximo capítulo veremos como usar los UDG y pintaremos todos nuestros gráficos en pantalla.

0x02 Pintando UDG

En este capítulo vamos empezar a dibujar usando UDG. El mapa de caracteres del ZX Spectrum está compuesto por doscientos cincuenta y seis valores, de los cuales podemos redefinir veintiuno, en concreto los que se encuentran entre el $90 (144) y el $A4 (164), ambos inclusive.

Antes de empezar, creamos una carpeta que se llame Paso02 y copiamos el arhivo Var.asm (dónde definimos los gráficos que vamos a usar) desde la carpeta Paso01.

¿Dónde están los UDG?

El valor de la dirección de memoria $5C7B contiene la dirección de memoria donde están los gráficos definidos por el usuario, por lo que lo único que tenemos que hacer es cargar en esa dirección de memoria, la dirección dónde están definidos nuestros gráficos. Una vez que lo tengamos, al pintar con RST $10 cualquier carácter comprendido entre $90 y $A4, pintara los gráficos que hemos definido.

Vamos a crear un nuevo fichero llamado Const.asm y vamos a añadir la línea siguiente:

```
; Dirección de memoria donde se cargan los gráficos definidos por el usuario.
UDG:                EQU $5c7b
```

En esta constante guardamos la dirección de memoria dónde cargaremos la dirección dónde están nuestros gráficos.

Pintamos nuestros UDG

Es el momento de hacer nuestra primera prueba, vamos a pintar los UDG. Creamos un fichero llamado Main.asm y vamos a añadir las líneas siguientes:

```
org                 $5dad


Main:
ld                  a, $90
ld                  b, $15


Loop:
push                af
rst                 $10
pop                 af
inc                 a
djnz                Loop


ret
```

```
end               Main
```

Lo primero que hacemos es indicar dónde se va a cargar el programa, ***ORG $5DAD***. El programa lo cargamos en la posición $5DAD (23981), ya que Batalla espacial va a ser un programa compatible con modelos 16K.

La siguiente línea es una etiqueta, ***Main,*** el punto de entrada del programa.

Lo siguiente que hacemos es cargar 144 en A, ***LD A, $90***, y 21 en B, ***LD B, $15***, para pintar desde el carácter 144 al 164, haciendo un total de veintiún caracteres.

El siguiente paso es hacer el bucle de veintiuna iteraciones, empezando con la etiqueta del mismo, ***Loop***, y a continuación preservamos en la pila el valor del registro A, ***PUSH AF***, lo cual es muy importante ya que la siguiente instrucción, ***RST $10,*** imprime en pantalla el carácter cuyo código esté cargado en el registro A, y luego modifica el valor de dicho registro. Acto seguido recuperamos de la pila el valor del registro A, ***POP AF***.

A continuación, incrementamos A, ***INC A***, para que apunte al siguiente carácter y decrementamos B y saltamos a Loop si no ha llegado a cero, ***DJNZ Loop***. Por último volvemos al Basic, ***RET***.

Con la última línea le indicamos a PASMO que debe incluir en el cargador Basic una llamada a la dirección de memoria donde se encuentra la etiqueta Main.

Es el momento de compilar y ver los resultados en el emulador.

```
pasmo --name Marciano --tapbas Main.asm Maciano.tap --public
```

Pero, ¿hemos pintado nuestros gráficos?

Como podemos ver, hemos pintado las letras mayúsculas de la A a la U, esto es debido a que en ningún momento hemos indicado dónde están nuestros gráficos.

Seguimos en el archivo Main.asm, justo debajo de la etiqueta Main añadimos las líneas siguientes:

```
ld                  hl, udgsCommon
ld                  (UDG), hl
```

Cargamos en HL la dirección de memoria dónde están nuestros gráficos, ***LD HL, udgsCommon***, y luego cargamos ese valor en la dirección de memoria en la que hay que indicar dónde están nuestros gráficos, ***LD (UDG), HL***.

Dado que tanto udgsCommon, como UDG no están definidas en el fichero Main.asm, justo detrás de la instrucción ***RET*** hay que añadir los includes para los ficheros Const.asm y Var.asm.

```
include     "Const.asm"
include     "Var.asm"
```

Ahora sí, podemos volver a compilar el programa, cargarlo en el emulador y ver nuestros gráficos en pantalla.

Mucho mejor, ¿verdad? Pero, hemos pintado veintiún gráficos: la nave, el disparo, la explosión, el marco, el carácter vacío, los cuatro gráficos del enemigo uno, y dos gráficos del enemigo dos. ¿Cómo vamos a pintar los otros dos gráficos del enemigo dos y los gráficos de los veintiocho enemigos restantes?

Cargamos los UDG de los enemigos

Si observáis la definición de los gráficos, la primera etiqueta se llama ***udgsCommon***, y esto debería darnos una pista de como lo vamos a hacer. Como UDG comunes tenemos definidos quince gráficos (nave, disparo, explosión, marco y blanco), por lo que vamos a definir treinta y dos bytes para poder ir volcando en ellos los gráficos de los enemigos; lo vamos a hacer así porque los enemigos son uno por nivel, y la operación de volcado solo la tenemos que hacer una vez, justo con el cambio de nivel.

En el archivo Var.asm, justo por encima de la etiqueta ***udgsEnemiesLeve1*** añadimos las siguientes líneas:

```
udgsExtension:
db $00, $00, $00, $00, $00, $00, $00, $00     ; $9f Left/Up
db $00, $00, $00, $00, $00, $00, $00, $00     ; $a0 Rigth/Up
db $00, $00, $00, $00, $00, $00, $00, $00     ; $a1 Left/Down
db $00, $00, $00, $00, $00, $00, $00, $00     ; $a2 Rigth/Down
```

En este bloque de memoria es dónde vamos a ir volcando los gráficos de los enemigos, dependiendo del nivel en el que nos encontremos.

Probad a compilar ahora y observad que pinta. ¿Faltan los gráficos del enemigo uno verdad? Está pintando udgsExtension.

Creamos un nuevo archivo, Graph.asm, y vamos a implementar en él la rutina que carga en ***udgsExtension*** los gráficos de los enemigos de cada nivel, dato que recibe en A.

Para calcular la dirección de memoria donde se encuentran los gráficos, vamos a multiplicar el nivel por treinta y dos (bytes que ocupan los gráficos) y el resultado se lo vamos a sumar a la dirección de memoria donde se encuentran los gráficos del primer enemigo.

```
LoadUdgsEnemies:
dec                a
ld                 h, $00
ld                 l, a
```

Dado que los niveles van de uno a treinta, decrementamos A, ***DEC A***, para que no sume un nivel de más (si el nivel es cero, tiene que sumar cero veces a udgsEnemies, si es dos una vez, etc.).

Lo siguiente es cargar el nivel en HL, para lo cual cargamos cero en H, ***LD H, $00***, y el nivel en L, ***LD L, A***.

```
add                hl, hl
add                hl, hl
add                hl, hl
add                hl, hl
add                hl, hl
```

Multiplicamos el nivel por treinta y dos, sumando HL a si mismo cinco veces, ***ADD HL, HL***. La primera suma es igual a multiplicar por dos, la segunda por cuatro y las siguientes por ocho, por dieciséis, y por treinta y dos.

```
ld                 de, udgsEnemiesLevel1
add                hl, de
ld                 de, udgsExtension
ld                 bc, $20
```

```
ldir
ret
```

Por último, cargamos la dirección de los gráficos del enemigo uno en DE, ***LD DE, udgsEnemiesLevel1***, y se lo sumamos a HL, ***ADD HL, DE***, cargamos la dirección de la extensión de udgs en DE, ***LD DE, udgsExtension***, cargamos en BC en número de bytes que vamos a cargar en udgsExtension, ***LD BC, $20***, y cargamos los treinta y dos bytes de los gráficos del enemigo del nivel a udgsExtension, ***LDIR***. Finalmente salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Carga los gráficos definidos por el usuario relativos a los enemigos
;
; Entrada: A -> Nivel de 1 a 30
;
; Altera el valor de los registros A, BC, DE y HL
; ------------------------------------------------------------------------------
LoadUdgsEnemies:
dec                a                   ; Decrementa A para que no sume un nivel de más

ld                 h, $00
ld                 l, a                ; Carga en HL el nivel
add                hl, hl              ; Multiplica por 2
add                hl, hl              ; por 4
add                hl, hl              ; por 8
add                hl, hl              ; por 16
add                hl, hl              ; por 32
ld                 de, udgsEnemiesLevel1      ; Carga la dirección de los gráficos del
                                       ; enemigo 1 en DE
add                hl, de              ; Lo suma a HL
ld                 de, udgsExtension   ; Carga en DE la dirección de la extensión
ld                 bc, $20             ; Carga en BC el número de bytes a copiar, 32
ldir                           ; Copia los bytes del enemigo en los de extensión

ret
```

Y ahora vamos a probar la nueva rutina, para lo cual vamos a editar el archivo Main.asm, empezando por cambiar la instrucción ***LD B, $15***, justo encima de la etiqueta Loop, y la dejamos como sigue, para imprimir los quince primeros UDG, los comunes:

```
ld                  b, $0f
```

El resto lo vamos a implementar entre la instrucción ***DJNZ Loop*** y la instrucción ***RET***.

```
ld                  a, $01
ld                  b, $1e
```

Cargamos en A el nivel uno, ***LD A, $01***, y en B el número de niveles totales (treinta), ***LD B, $1E***. Implementamos un bucle para pintar los enemigos de los treinta niveles.

```
Loop2:
push                af
push                bc
call                LoadUdgsEnemies
```

Preservamos los valores de AF, ***PUSH AF***, y de BC, ***PUSH BC***, ya que usamos A y B para controlar que enemigos pintamos y las iteraciones del bucle. A continuación, llamamos a la rutina que carga los gráficos del enemigo del nivel en udgsExtension, ***CALL LoadUdgsEnemies***.

```
ld                  a, $9f
rst                 $10
ld                  a, $a0
rst                 $10
ld                  a, $a1
rst                 $10
ld                  a, $a2
rst                 $10
```

Los caracteres correspondientes a los gráficos de los enemigos son $9F, $A0, $A1 y $A2; los vamos cargando en A, ***LD A, $9F***, y pintando, ***RST $10***. Repetimos la operación con $A0, $A1 y $A2.

```
pop                 bc
pop                 af
inc                 a
djnz                Loop2
```

Recuperamos el valor de BC, ***POP BC***, de AF, ***POP AF***, incrementamos A para pasar al siguiente nivel, ***INC A***, y repetimos hasta que B sea 0, ***DJNZ Loop2***.

Por último, a final del fichero y antes de ***END Main***, incluimos el fichero Graph.asm.

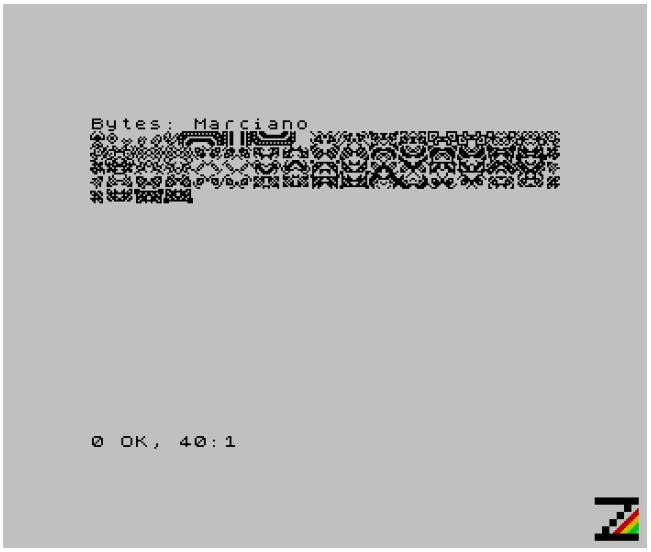
```
include     "Graph.asm"
```

El código final de Main.asm es el siguiente:

```
org          $5dad

Main:
ld           hl, udgsCommon
ld           (UDG), hl

ld           a, $90
ld           b, $0f
Loop:
push         af
rst          $10
pop          af
inc          a
djnz         Loop

ld           a, $01
ld           b, $1e
Loop2:
push         af
push         bc
call         LoadUdgsEnemies
ld           a, $9f
rst          $10
ld           a, $a0
rst          $10
ld           a, $a1
rst          $10
ld           a, $a2
rst          $10
pop          bc
pop          af
inc          a
djnz         Loop2

ret
```

```
include     "Const.asm"
include     "Var.asm"
include     "Graph.asm"


end         Main
```

Compilamos y cargamos en el emulador; ya pintamos todos nuestros gráficos.



Conclusión

Llegados a este punto, ya tenemos definidos todos los gráficos y hemos aprendido como pintarlos.

En el próximo capítulo pintaremos el área de juego.

0x03 Área de juego

Creamos la carpeta Paso03 y copiamos los archivos Const.asm, Graph.asm, Main.asm y Var.asm desde la carpeta Paso02.

Antes de comenzar a pintar el área de juego es necesario saber que la pantalla del ZX Spectrum se divide en dos zonas, la parte superior con veintidós líneas (de la cero a la veintiuna), y la parte inferior (la línea de comandos).

Si cargáis el programa resultante del capítulo anterior, una vez que se ejecuta, si pulsamos la tecla ENTER se muestra el listado del cargador. Si ejecutáis la línea 40 (RUN 40), se deberían volver a pintar nuestros gráficos, pero no es así. En realidad si se pintan, pero se pintan en la línea de comandos; si estáis atentos veréis como se pintan y luego desaparecen.

Tras ejecutar nuestro programa, la parte de la pantalla activa es la línea de comandos, así que necesitamos un mecanismo para activar la parte de la pantalla en dónde queramos pintar.

Cambiando la pantalla activa

La parte superior de la pantalla es la dos, y la inferior es la uno. En la ROM hay una rutina que activa uno u otro canal, dependiendo del valor que haya en el registro A.

Abrimos el archivo Const.asm y añadimos la líneas siguientes:

```
; ------------------------------------------------------------------------
; Rutina de la ROM que abre el canal de la pantalla.
;
; Entrada: A -> Canal     1 = línea de comandos
;                         2 = pantalla superior
; ------------------------------------------------------------------------
OPENCHAN:          EQU $1601
```

A continuación, abrimos el archivo Main.asm y justo debajo de la etiqueta Main añadimos las líneas siguientes:

```
ld                 a, $02
call               OPENCHAN
```

Cargamos en A el canal que queremos activar, ***LD A, $02***, y luego llamamos a la rutina de la ROM para activarlo, ***CALL OPENCHAN***.

Compilamos, cargamos en el emulador, pulsamos cualquier tecla, ejecutamos la línea 40 (RUN 40) y ahora sí se vuelven a pintar nuestros gráficos en el lugar correcto.

Pintamos cadenas de texto

Al trabajar con UDG, podríamos decir que pintamos caracteres, y como tal vamos a pintar la pantalla de juego.

Lo primero que vamos a implementar es una rutina que pinta cadenas de caracteres, indicando la dirección de memoria de la cadena y la longitud de la misma.

Creamos un nuevo archivo, Print.asm, e implementamos la rutina PrintString, que recibe en HL la dirección de la cadena y en B la longitud de la misma. Esta rutina altera el valor de los registros AF, B y HL.

```
PrintString:
ld                  a, (hl)
rst                 $10
inc                 hl
djnz                PrintString
ret
```

Carga en A el carácter a pintar, ***LD A, (HL)***, pinta el carácter, ***RST $10***, apunta HL al siguiente carácter, ***INC HL***, y repite la operación hasta que B valga 0, ***DJNZ PrintString***. Finalmente, sale, ***RET***.

El aspecto final de la rutina, una vez comentada, es el siguiente:

```
; ------------------------------------------------------------------------------
; Pinta cadenas.
;
; Entrada:  HL = primera posición de memoria de la cadena
;           B  = longitud de la cadena.
; Altera el valor de los registros AF, B y HL
; ------------------------------------------------------------------------------
PrintString:
ld          a, (hl)         ; Carga en A el carácter a pintar
rst         $10             ; Pinta el carácter
inc         hl              ; Apunta HL al siguiente carácter
djnz        PrintString     ; Hasta que B valga 0
ret
```

El siguiente paso es probar, así que vamos a editar el archivo Main.asm. Lo primero, para que no se nos olvide, es incluir el archivo Print.asm antes ***END Main***:

```
include     "Print.asm"
```

Justo antes de los includes, vamos a definir una cadena con dos etiquetas, la de la propia cadena y una segunda etiqueta para marcar el final de la misma.

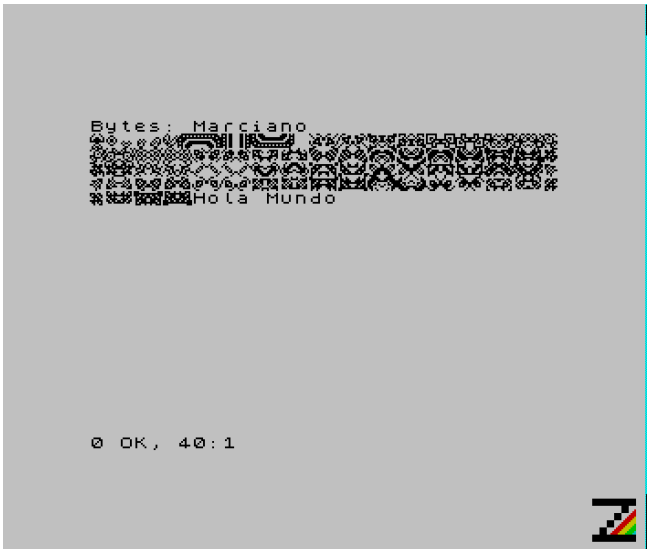
```
Cadena:
db 'Hola Mundo'
Cadena_Fin:
```

```
db $
```

Justo antes del ***RET*** que nos saca al Basic, vamos a añadir la llamada a la nueva rutina.

```
ld          hl, Cadena
ld          b, Cadena_Fin - Cadena
call        PrintString
```

Compilamos, cargamos en el emulador y vemos los resultados.

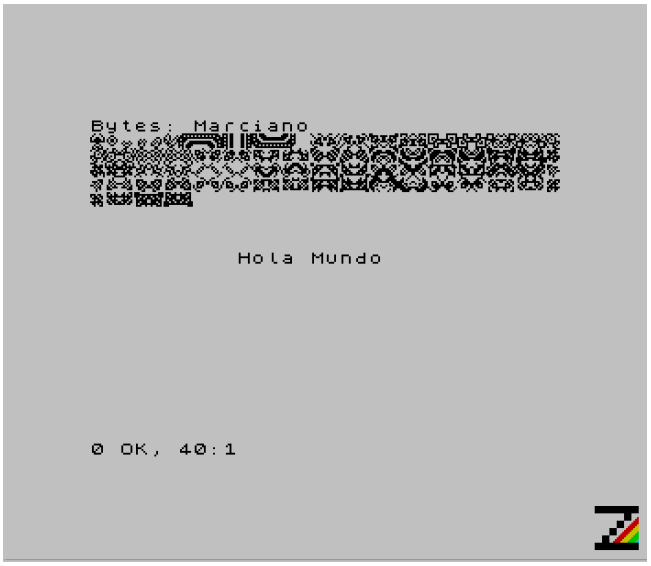


Como se puede apreciar, se ha pintado la cadena Hola Mundo a continuación de nuestros gráficos.

Vamos a añadir unos caracteres antes de la cadena, y la vamos a dejar como sigue:

```
db $16, $0a, $0a, 'Hola Mundo'
```

Volvemos compilar, cargamos en el emulador y vemos el resultado.



Como se puede observar, la cadena Hola Mundo aparece más centrada, lo que hemos conseguido con los caracteres que hemos añadido por delante de la cadena, en concreto $16 que es el carácter de control de AT (instrucción Basic para posicionar el cursor), y las coordenadas Y y X.

A continuación se muestra una lista de los caracteres de control que podemos usar al pintar las cadenas de esta manera, y los parámetros que hay que enviar.

Carácter	**Código**	**Parámetros**	**Valores**
DELETE	$0c		
ENTER	$0d		
INK	$10	Color	De $00 a $07
PAPER	$11	Color	De $00 a $07
FLASH	$12	No/Sí	De $00 a $01
BRIGHT	$13	No/Sí	De $00 a $01
INVERSE	$14	No/Sí	De $00 a $01
OVER	$15	No/Sí	De $00 a $01
AT	$16	Coordenadas Y y X	Y = de $00 a $15 X = de $00 a $1f
TAB	$17	Número de tabulaciones	

Es muy importante que todos los códigos de control vayan seguidos de sus parámetros, para evitar resultados no deseados. Así mismo, si se imprime una cadena después de TAB, hay que añadir un espacio en blanco como primer carácter de la cadena.

Como ejercicio, probad distintas combinaciones con los caracteres de control, probad a darle color, parpadeo, etc.

Pintamos la pantalla de juego

La pantalla de juego esta bordeada por un marco, pero antes de pintar nada vamos a limpiar el archivo Main.asm, quitando todo lo que nos sobra; borramos desde las dos líneas anteriores a la etiqueta ***Loop***, hasta la línea anterior de la instrucción ***RET***, también borramos la definición de ***Cadena*** y ***Cadena_Fin***.

El aspecto de Main.asm debe ser el siguiente:

```
org   $5dad


Main:
ld      a, $02
call    OPENCHAN
ld      hl, udgsCommon
ld      (UDG), hl
ret


include "Const.asm"
include "Var.asm"
```

```
include "Graph.asm"

include "Print.asm"


end     Main
```

Ahora vamos a definir en Var.asm las cadenas necesarias para pintar el marco. La vamos a incluir antes de ***udgsCommon***.

```
; ------------------------------------------------------------------------------

; Marco de la pantalla

; ------------------------------------------------------------------------------

frameTopGraph:

db $16, $00, $00, $10, $01

db $96, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97,
$97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $98

frameBottomGraph:

db $16, $15, $00

db $9b, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c,
$9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9d

frameEnd:
```

En la primera línea ***DB*** definimos la posición de la parte de arriba, ***$16, $00, $00 y*** la tinta, ***$10, $01***.

En la siguiente línea definimos la parte superior del marco, primero la esquina superior izquierda, ***$96***, luego treinta posiciones de horizontal superior, ***$97***, y por último la esquina superior derecha, ***$98***; todos estos números los pusimos en los comentarios de la definición de los gráficos.

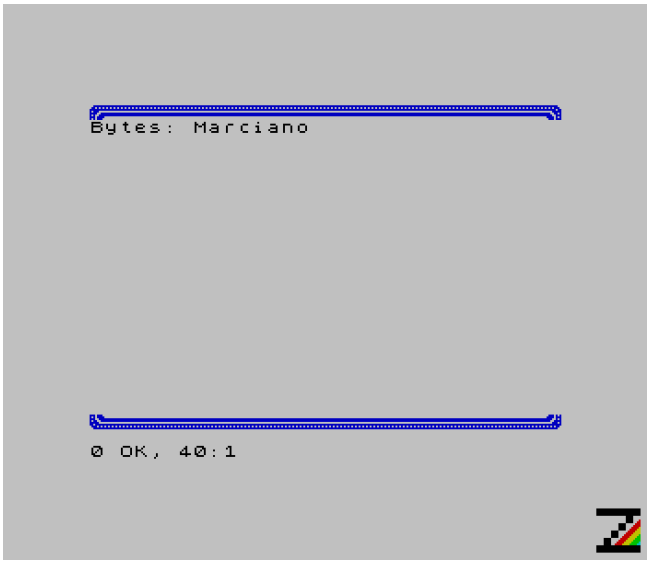
En la siguiente línea definimos la posición de la parte de abajo, ***$16, $15, $00***.

En la siguiente línea definimos la parte inferior del marco, primero la esquina inferior izquierda, ***$9b***, luego treinta posiciones de la horizontal inferior, ***$9c***, y por último la esquina inferior derecha, ***$9d***.

Vamos a ver como se pinta todo esto. Volvemos a Main.asm y justo encima del ***RET***, incluimos las líneas siguientes:

```
ld                hl, frameTopGraph

ld                b, frameEnd - frameTopGraph

call              PrintString
```

Compilamos, cargamos en el emulador y vemos los resultados.



Hemos pintado la parte superior del marco y la inferior, aunque el marco no está completo ya que faltan los laterales.

Vamos a implementar una rutina que imprima el marco, y lo vamos a hacer en Print.asm.

```
PrintFrame:
ld          hl, frameTopGraph
ld          b, frameBottomGraph - frameTopGraph
call        PrintString
```

Cargamos en HL la dirección de memoria de la parte superior del marco, ***LD HL, frameTopGraph***, cargamos en B la longitud, restando a la dirección de inicio de la parte inferior la dirección de inicio de la parte superior, ***LD B, frameBottomGraph – frameTopGraph***, y llamamos a la rutina que pinta las cadenas, ***CALL PrintString***.

```
ld          hl, frameBottomGraph
ld          b, frameEnd - frameBottomGraph
call        PrintString
```

Cargamos en HL la dirección de memoria de la parte inferior del marco, ***LD HL, frameBottomGraph***, cargamos en B la longitud, restando a la dirección de memoria donde acaba la cadena la dirección de inicio de la parte inferior, ***LD B, frameEnd – frameBottomGraph***, y llamamos a la rutina que pinta las cadenas, ***CALL PrintString***.

Ya solo queda implementar un bucle para pintar los laterales.

```
ld          b, $01
printFrame_loop:
ld          a, $16
rst         $10
ld          a, b
rst         $10
```

```
ld          a, $00
rst         $10
ld          a, $99
rst         $10
```

Cargamos en B la línea en la que empezamos a pintar los laterales, ***LD B, $01***, cargamos en A el carácter de control de AT, ***LD A, $16***, y lo "pintamos", ***RST $10***, cargamos en A la coordenada Y, ***LD A, B***, y la "pintamos", ***RST $10***, cargamos en A la columna cero, ***LD A, $00***, *y la "pintamos"*, ***RST $10***, y por último, cargamos en A el carácter del lateral izquierdo, ***LD A, $99***, y lo pintamos, ***RST $10***.

Hacemos lo mismo con el lateral derecho, debido a que el código es prácticamente el mismo, solo marcamos las dos líneas que cambian.

```
ld          a, $16
rst         $10
ld          a, b
rst         $10
ld          a, $1f ; ¡CAMBIO!
rst         $10
ld          a, $9a ; ¡CAMBIO!
rst         $10
```

Y llegamos a la parte final de la rutina.

```
inc         b
ld          a, b
cp          $15
jr          nz, printFrame_loop


ret
```

Apuntamos B a la siguiente línea, ***INC B***, cargamos el valor en A, ***LD A, B***, y comprobamos si B apunta a la línea veintiuno (línea donde se encuentra la parte inferior del marco), ***CP $15***, y de no ser así repite el bucle hasta que llegue a la línea veintiuno, ***JR NZ, printFrame_loop***. Una vez que B apunta a la línea veintiuno, salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta el marco de la pantalla.
;
; Altera el valor de los registros HL, B y AF.
; ------------------------------------------------------------------------
PrintFrame:
```

```
ld          hl, frameTopGraph          ; Carga en HL la dirección de la parte superior
ld          b, frameBottomGraph - frameTopGraph ; Carga en B la longutid
call        PrintString                ; Pinta la cadena

ld          hl, frameBottomGraph       ; Carga en HL la dirección de la parte inferior
ld          b, frameEnd - frameBottomGraph ; Carga en B la longitudd
call        PrintString                ; Pinta la cadena

ld          b, $01                     ; Apunta B a la línea 1
printFrame_loop:
ld          a, $16                     ; Carga en A el carácter de control de AT
rst         $10                        ; Lo "pinta"
ld          a, b                       ; Carga en A la línea
rst         $10                        ; La "pinta"
ld          a, $00                     ; Carga en A la columna
rst         $10                        ; La “pinta”
ld          a, $99                     ; Carga en A el carácter lateral izquierdo
rst         $10                        ; Lo pinta

ld          a, $16                     ; Carga en A el carácter de control de AT
rst         $10                        ; Lo "pinta"
ld          a, b                       ; Carga en A la línea
rst         $10                        ; La "pinta"
ld          a, $1f                     ; Carga en A la columna
rst         $10                        ; La “pinta”
ld          a, $9a                     ; Carga en A el carácter lateral derecho
rst         $10                        ; Lo pinta

inc         b                          ; Apunta B a la línea siguiente
ld          a, b                       ; Carga el valor de B en A
cp          $15                        ; Comprueba si está en la línea veintiuno
jr          nz, printFrame_loop        ; Si no es así, sigue con el bucle

ret
```

Ha llegado el momento de probar si se pinta todo el marco. Volvemos al archivo Main.asm y sustituimos estas lineas:

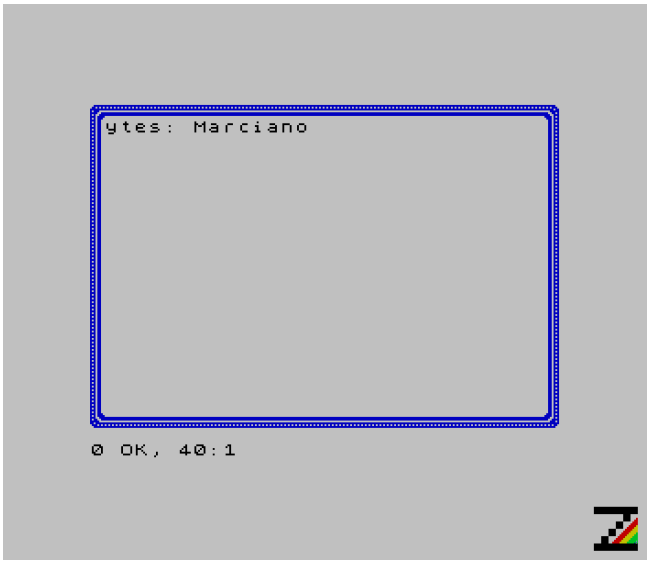
```
ld          hl, frameTopGraph
ld          b, frameEnd - frameTopGraph
```

```
call        PrintString
```

Por esta otra:

```
call        PrintFrame
```

Compilamos, cargamos en el emulador y vemos el resultado.



Como vemos, ya hemos pintado el marco de la pantalla, pero quedan cosas por hacer; no hemos borrado la pantalla y se ven cosas que no deberían estar.

Limpiamos y coloreamos la pantalla

Muchos de los listados Basic que se pueden ver, en algún momento tienen una línea parecida a esta:

```
BORDER 0: INK 7: PAPER 0: CLS
```

Con esta línea se pone el borde de la pantalla en negro, la tinta en blanco y el fondo negro, y por último se limpia la pantalla aplicando los atributos de tinta y fondo.

Vamos a usar la ROM del ZX Spectrum para asignar tinta, fondo, limpiar la pantalla y vamos a implementar el cambio de color del borde.

Empezamos con la parte en la que limpiamos la pantalla, para lo cual abrimos el archivo Const.asm y añadimos las líneas siguientes:

```
; Variable de sistema donde están los atributos de color permanentes
ATTR_P:     EQU $5c8d
; ------------------------------------------------------------------------------
; Rutina de la ROM que limpia la pantalla usando el valor que hay en ATTR_P.
; ------------------------------------------------------------------------------
CLS:        EQU $0daf
```

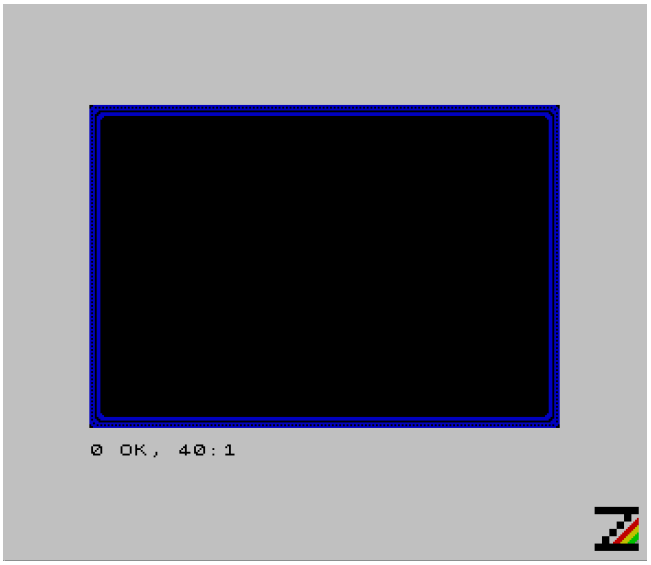
En ATTR_P se guardan los atributos permanentes de color en formato FBPPPIII, dónde F = FLASH (0/1), B = BRIGHT (0/1), PPP = PAPER (de 0 a 7) e III = INK (de 0 a 7). Por otro lado, CLS limpia la pantalla aplicando los atributos que hay en ATTR_P.

Volvemos a Main.asm y justo antes de la llamada a PrintFrame, añadimos las líneas siguientes:

```
ld          hl, ATTR_P
ld          (hl), $07
call        CLS
```

Cargamos la dirección de memoria de los atributos permanentes en HL, ***LD HL, ATTR_P***, y ponemos FLASH a 0, BRIGHT a 0, PAPER a 0 e INK a 7, ***LD (HL), $07***. Por último, llamamos a la rutina que limpia la pantalla, ***CALL CLS***.

Compilamos, cargamos en el emulador y vemos los resultados.



Ahora vamos a cambiar el color al borde. Ya vimos en PorompomPong que la rutina BEEPER de la ROM cambia el color del borde, por lo que es necesario guardar los atributos en una variable de sistema; los atributos tienen el mismo formato que el visto para ATTR_P.

Volvemos al archivo Const.asm y añadimos la constante para la variable de sistema donde se guardan los atributos del borde.

```
; Variable de sistema donde se guarda el borde. También usada por BEEPER.
; También se guardan aquí los atributos de la línea de comandos.
BORDCR:     EQU $5c48
```

Y ahora volvemos a Main.asm y ponemos el borde en negro, justo debajo de ***CALL CLS***.

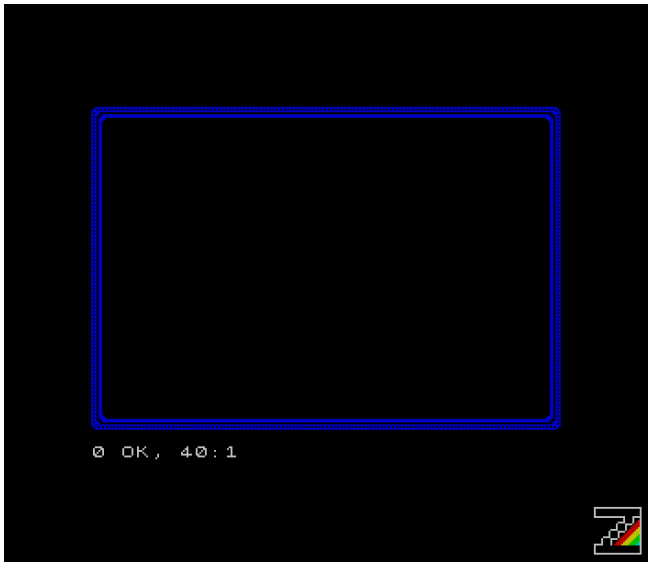
```
xor         a
out         ($fe), a
ld          a, (BORDCR)
and         $c7
or          $07
```

```
ld          (BORDCR), a
```

Ponemos A a cero, ***XOR A***, y el borde en negro, ***OUT ($FE), A***. A continuación, cargamos el valor de BORDCR en A, ***LD A, (BORDCR)***, desechamos el color del borde y así lo ponemos en negro = 0, ***AND $C7***, ponemos la tinta en blanco, ***OR $07***, y cargamos el valor en BORDCR, ***LD (BORDCR), A***.

La instrucción ***OR $07*** solo es necesaria mientras volvamos al Basic, recordemos que en BORDCR están los atributos de color de la línea de comandos y si no cambiamos la tinta a blanca, se queda como originalmente está, en negro, igual que el color que le hemos dado al fondo.

Compilamos, cargamos en el emulador y vemos los resultados.



Ya solo nos queda pintar el área de información de la partida, cosa que haremos en la línea de comandos.

Pintamos la información de la partida

Lo primero es definir la línea de título del área de información de la partida, lo que vamos a hacer al inicio de Var.asm.

```
; ------------------------------------------------------------------------------
; Título de información de la partida
; ------------------------------------------------------------------------------
infoGame:
db  $10, $03, $16, $00, $00
db 'Vidas   Puntos   Nivel  Enemigos'
infoGame_end:
```

En la primera línea ponemos la tinta en magenta y posicionamos el cursor en las coordenadas 0,0. En la línea siguiente definimos los títulos de la información.

Vamos implementar en Print.asm la rutina que pinta los títulos de la información.

```
PrintInfoGame:
ld          a, $01
call        OPENCHAN
```

Como los títulos de la información se pintan en la línea de comandos, lo primero que hay que hacer es activar el canal uno. Cargamos uno en A, ***LD A, $01***, y llamamos al cambio de canal, ***CALL OPENCHAN***.

```
ld          hl, infoGame
ld          b, infoGame_end - infoGame
call        PrintString
```

Cargamos en HL la dirección del mensaje de información, ***LD HL, infoGame***, cargamos en B la longitud del mensaje, ***LD B, infoGame_end – infoGame***, y llamamos a pintar la cadena, ***CALL PrintString***.

```
ld          a, $02
call        OPENCHAN


ret
```

Por último, volvemos a activar el canal dos (la pantalla superior), y salimos.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta los títulos de información de la partida.
; Altera el valor de los registros A, C y HL.
; ------------------------------------------------------------------------
PrintInfoGame:
ld          a, $01              ; Carga 1 en A
call        OPENCHAN            ; Activa el canal 1, línea de comando
ld          hl, infoGame        ; Carga la dirección de la cadena de títulos en HL
ld          b, infoGame_end - infoGame       ; Carga la longitud en B
call        PrintString         ; Pinta la cadena de títulos
ld          a, $02              ; Carga 2 en A
call        OPENCHAN            ; Activa el canal 2, pantalla superior


ret
```

Ahora volvemos a Main.asm y justo después de ***CALL PrintFrame***, añadimos la llamada para pintar los títulos de información de partida.

```
call        PrintInfoGame
```

Si ahora mismo compiláramos, probad si queréis, da la sensación de que lo último que hemos implementado no funciona, no pinta los títulos de información. En realidad si lo hace, pero al volver al Basic, el mensaje ***0 OK, 40:1*** lo borra.

Para evitar esto, vamos a quedarnos en un bucle infinito; localizamos la instrucción ***RET*** que nos devuelve a Basic y la sustituimos por las líneas siguientes:

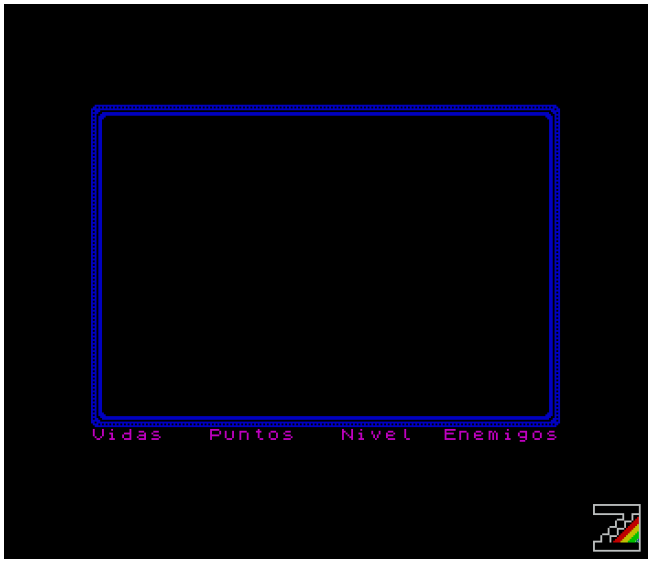
```
Main_loop:
jr          Main_loop
```

El código final de Main.asm es el siguiente:

```
org         $5dad


Main:
ld          a, $02
call        OPENCHAN

ld          hl, udgsCommon
ld          (UDG), hl

ld          hl, ATTR_P
ld          (hl), $07
call        CLS

xor         a
out         ($fe), a
ld          a, (BORDCR)
and         $c7
or          $07
ld          (BORDCR), a

call        PrintFrame
call        PrintInfoGame

Main_loop:
jr          Main_loop

include     "Const.asm"
include     "Var.asm"
include     "Graph.asm"
```

```
include     "Print.asm"


end         Main
```

Ahora sí, compilamos, cargamos en el emulador y vemos el resultado.



No sé que os parece a vosotros, pero a mí, que los títulos estén tan pegados al marco, no me gusta. Dado que en la línea de comandos solo disponemos de dos líneas sin que haga scroll, y debajo de la línea de títulos hay que pintar los datos, solo nos queda una opción, quitarle una línea al área de juego.

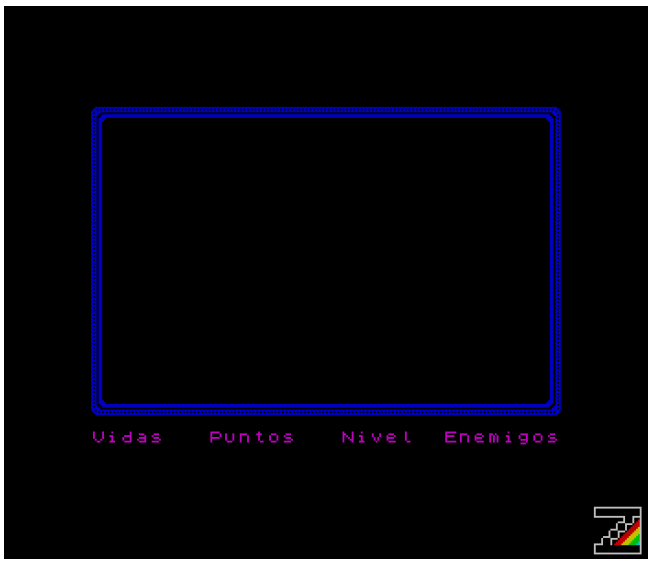
Vamos al archivo Var.asm, localizamos la etiqueta ***frameBottomGraph***, y justo en la línea de abajo vemos el ***DB*** que posiciona el cursor; vamos a modificar esta línea para que la coordenada Y sea veinte en lugar de veintiuno.

```
db $16, $14, $00
```

Ahora tenemos que ir a Print.asm y localizar la etiqueta ***printFrame_loop***. Este bucle se ejecuta hasta que B vale veintiuno; tenemos que modificar esta condición para que se ejecute hasta que valga veinte. Dos líneas por encima del ***RET*** de este método tenemos ***CP $15***, esta es la línea que tenemos que modificar dejándola de la siguiente manera:

```
cp          $14                 ; Comprueba si está en la línea veinte
```

Volvemos a compilar, cargamos en el emulador y vemos los resultados.



Conclusión

Llegados a este punto, ya tenemos nuestro área de juego.

En el próximo capítulo incluiremos la nave y la moveremos.

0x04 Nave

Antes de nada, creamos la carpeta Paso04 y copiamos, desde la carpeta Paso03, los archivos Const.asm, Graph.asm, Main.asm, Print.asm y Var.asm.

En este capítulo vamos a implementar la nave, su movimiento, y por tanto, los controles.

Posicionamiento en pantalla

Hasta ahora, hemos usado el carácter de control de la instrucción ***AT*** y las coordenadas para posicionarnos por la pantalla, pero esto resulta lento.

Abrimos Graph.asm y al inicio del archivo vamos a implementar una rutina que hace lo mismo, pero es más rápida.

```
; ------------------------------------------------------------------------------
; Posiciona el cursor en las coordenadas especificadas.
;
; Entrada:  B = Coordenada Y (24 a 3).
;           C = Coordenada X (32 a 1).
; Altera el valor de los registros AF y BC.
; ------------------------------------------------------------------------------
At:
push  de    ; Preserva el valor de DE
push  hl    ; Preserva el valor de HL
call  $0a23 ; Llama a la rutina de la ROM
pop   hl    ; Recupera el valor de HL
pop   de    ; Recupera el valor de DE


ret
```

En esta rutina hacemos uso de la rutina de la ROM que posiciona el cursor. Preservamos el valor de DE, ***PUSH DE***, y el de HL, ***PUSH HL***. Una vez preservados estos valores, llamamos a la rutina de la ROM, ***CALL $0a23***, y recuperamos los valores de HL, **POP HL**, *y también el de DE*, **POP DE**. *Finalmente, salimos,* ***RET***.

En los comentarios podéis observar que esta rutina, en realidad la de la ROM, también altera el valor de AF y BC, pero no los preservamos; no nos va a afectar que lo altere y por eso nos ahorramos dos ***PUSH*** y dos ***POP***.

Otra cosa a tener muy en cuenta, y una pista os la dan los comentarios, es que para la rutina de la ROM la esquina superior izquierda está en las coordenadas Y=24 y X=32, por lo que trabajaremos con las coordenadas invertidas con respecto a la instrucción ***AT***.

Vamos a abrir el archivo Const.asm y añadimos las constantes de las coordenadas.

```
; Coordenadas de la pantalla para la rutina de la ROM que posiciona el cursor,
```

```
; relativas al área de juego (el marco).
COR_X: EQU $20  ; Coordenada X de la esquina superior izquierda
COR_Y: EQU $18  ; Coordenada Y de la esquina superior izquierda
MIN_X: EQU $00  ; A restar de COR_X para X esquina superior izquierda
MIN_Y: EQU $00  ; A restar de COR_Y para Y esquina superior izquierda
MAX_X: EQU $1f  ; A restar de COR_X para X esquina inferior derecha
MAX_Y: EQU $15  ; A restar de COR_Y para Y esquina inferior derecha
```

Recordemos que la directiva ***EQU*** no se compila, por lo que no aumenta el tamaño del binario, lo que hace es sustituir la etiqueta por el valor en aquellos lugares en dónde se encuentre.

Pintamos la nave

Lo primero que vamos a hacer es poner la nave en nuestra zona de juego; la nave se va a mover de izquierda a derecha en la parte inferior de la zona de juego.

Al ser la nave una parte móvil, necesitamos saber en todo momento en que posición está, y la posición inicial, tal y como vimos con las palas en PorompomPong.

Abrimos el archivo Var.asm (el que hay en la carpeta Paso04), y añadimos, tras las declaraciones de los títulos de información de la partida, las líneas siguientes:

```
; ------------------------------------------------------------------------
; Declaraciones de los gráficos de los distintos personajes
; y la configuración de coordenadas (Y, X)
; ------------------------------------------------------------------------

; ------------------------------------------------------------------------
; Nave
; ------------------------------------------------------------------------
shipPos:
dw $0511
```

En el caso de la nave, solo vamos a definir la posición, ***DW $0511***, un valor de dos bytes, primero la coordenada Y y luego la X, que durante la partida cargaremos en BC para posicionar la nave. La posición $0511 es el resultado de restar 19 ($13) y 15 ($0f) de las coordenadas de la esquina superior derecha que usa la rutina de la ROM ($1820).

Abrimos el archivo Const.asm e incluimos constantes para el carácter de la nave, la posición inicial y los topes a izquierda y derecha.

```
; Código de carácter de la nave, posición inicial y topes
SHIP_GRAPH: EQU $90
SHIP_INI:   EQU $0511
SHIP_TOP_L: EQU $1e
```

```
SHIP_TOP_R: EQU $01
```

Para pintar los colores correctos, y para no repetir el código una y otra vez, vamos a implementar una rutina para cambiar el color de tinta, la vamos a implementar en el archivo Graph.asm. Esta rutina recibe en A el valor de la tinta.

Antes de implementar la rutina, abrimos el archivo Const.asm y añadimos la constante de la posición de memoria donde se guardan los atributos de color actuales. Estos atributos son los usados por ***RST $10*** para asignar el color al carácter que pinta.

```
; Variable de sistema donde están los atributos de color actuales
ATTR_T:     EQU $5c8f
```

Y ahora sí, abrimos el archivo Graph.asm e implementamos la rutina Ink.

```
; ------------------------------------------------------------------------
; Cambia la tinta
;
; Entrada:  A -> Color de la tinta
; Altera el valor del registro A.
; ------------------------------------------------------------------------
Ink:
exx                     ; Preserva el valor de BC, DE y HL
ld      b, a            ; Carga la tinta en B
ld      a, (ATTR_T)     ; Carga los atributos actuales en A
and     $f8             ; Desecha los bits de la tinta
or      b               ; Añade la tinta
ld      (ATTR_T), a     ; Carga los atributos actuales
exx                     ; Recupera el valor de BC, DE y HL


ret
```

Dado que necesitamos apoyarnos en el registro B, lo primero que hacemos es preservar su valor con la instrucción ***EXX***, que lo que hace es intercambiar el valor de los registros BC, DE y HL con los registros alternativos ‘BC, ‘DE y ‘HL con tan solo un byte y cuatro ciclos de reloj, siendo más rápido y ocupando menos que si hubiéramos usado la pila.

Cargamos en B el valor de la tinta, ***LD B, A***, cargamos en A los atributos actuales, ***LD A, (ATTR_T)***, desechamos la tinta, ***AND $F8***, añadimos la tinta, ***OR B***, y cargamos el resultado en los atributos actuales, ***LD (ATTR_T), A***. Por último, recuperamos el valor de los registros BC, DE y HL, ***EXX***, y salimos, ***RET***.

Ahora necesitamos una rutina que pinte la nave; la vamos a implementar en el archivo Print.asm.

```
PrintShip:
ld          a, $07
```

```
call        Ink
```

Cargamos en A la tinta blanca, ***LD A, $07***, y llamamos a cambiar la tinta, ***CALL Ink***.

```
ld          bc, (shipPos)
call        At
```

Cargamos en BC la posición actual de la nave, ***LD BC, (shipPos)***, y llamamos a posicionar el cursor, ***CALL At***.

```
ld          a, SHIP_GRAPH
rst         $10


ret
```

Cargamos en A el código de carácter de la nave, ***LD A, SHIP_GRAPH***, la pintamos, ***RST $10***, *y salimos*, **RET**.

Ya tenemos lista la rutina que pinta la nave, cuyo aspecto final es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta la nave en la posición actual.
; Altera el valor de los registros A y BC.
; ------------------------------------------------------------------------
PrintShip:
ld                  a, $07              ; Carga en A la tinta blanca
call                Ink                 ; Llama al cambio de tinta

ld                  bc, (shipPos)       ; Carga en BC la posición actual de la nave
call                At                  ; Llama a posicionar el cursor

ld                  a, SHIP_GRAPH       ; Carga en A el carácter de la nave
rst                 $10                 ; La pinta


ret
```

Antes de dejar el archivo Print.asm, vamos a volver sobre la rutina que pinta el marco, en concreto a la parte en la que se hacemos un bucle para pintar los laterales.

```
printFrame_loop:
ld                  a, $16                ; Carga en A el carácter de control de AT
rst                 $10                    ; Lo "pinta"
ld                  a, b                   ; Carga en A la línea
rst                 $10                    ; La "pinta"
```

```
ld                a, $00                    ; Carga en A la columna
rst               $10                       ; La "pinta"
ld                a, $99                    ; Carga en A el carácter lateral izquierdo
rst               $10                       ; Lo pinta

ld                a, $16                    ; Carga en A el carácter de control de AT
rst               $10                       ; Lo "pinta"
ld                a, b                      ; Carga en A la línea
rst               $10                       ; La "pinta"
ld                a, $1f                    ; Carga en A la columna
rst               $10                       ; La "pinta"
ld                a, $9a                    ; Carga en A el carácter lateral derecho
rst               $10                       ; Lo pinta

inc               b                         ; Apunta B a la línea siguiente
ld                a, b                      ; Carga el valor de B en A
cp                $14                       ; Comprueba si está en la línea veintiuno
jr                nz, printFrame_loop ; Si no es así, sigue con el bucle
```

Como podemos observar, las seis lineas siguientes a ***printFrame_loop*** posicionan el cursor para pintar en el lateral izquierdo, más adelante vemos otras seis líneas que hacen lo mismo para el lateral derecho.

Yo estoy usando Visual Studio Code con la extensión Z80 Assembly meter, y por eso sé que esta rutina, desde ***printFrame_loop*** hasta ***JR NZ, printFrame_loop***, consume ciento sesenta y cinco ciclos de reloj y veintiocho bytes.

Dado que ya hemos implementado una rutina que posiciona el cursor, acabamos de implementar At, vamos a sustituir estas líneas por llamadas a esa rutina.

Lo primero es, justo encima de printFrame_loop, modificar la línea ***LD B, $01***, y dejarla como sigue:

```
ld          b, COR_Y - $01
```

Recordad que con la rutina de la ROM las coordenadas están invertidas. Apuntamos B a la línea uno, ***LD B, COR_Y - $01***.

Borramos la primeras seis líneas justo debajo de ***printFrame_loop*** y las sustituimos por las siguientes:

```
ld      c, COR_X - $01
call    At
```

Unas líneas más abajo, borramos desde ***LD A, $16*** hasta el ***RST $10*** que hay justo encima de ***LD A, $9a***, y sustituimos estas líneas por las siguientes:

```
ld      c, COR_X - MAX_X
call    At
```

Ahora vamos a sustituir desde ***INC B*** hasta ***JR NZ, printFrame_loop***.

```
dec     b
ld      a, COR_Y - MAX_Y + $01
sub     b
jr      nz, printFrame_loop
```

De esta manera, el aspecto final de la parte modificada es el siguiente:

```
ld      b, COR_Y - $01          ; Apunta B a la línea 1
printFrame_loop:
ld      c, COR_X - MIN_X        ; Apunta C a la columna 0
call    At                      ; Posiciona el cursor
ld      a, $99                  ; Carga en A el carácter lateral izquierdo
rst     $10                     ; Lo pinta

ld      c, COR_X - MAX_X        ; Apunta C a la columna 31
call    At                      ; Posiciona el cursor
ld      a, $9a                  ; Carga en A el carácter lateral derecho
rst     $10                     ; Lo pinta

dec     b                       ; Decrementa B
ld      a, COR_Y - MAX_Y + $01  ; Apunta A a la línea 20
sub     b                       ; Resta la siguiente línea
jr      nz, printFrame_loop     ; Si el resultado no es cero, sigue con el bucle
```

A simple vista, la rutina es más corta, consumiendo veintidós bytes y ciento once ciclos de reloj, pero cuidado, no es oro todo lo que reluce, a esos ciclos de reloj hay que sumarle los ciclos de reloj de la rutina At, que son sesenta y nueve (los bytes no los añadimos pues la rutina la vamos a usar desde más sitios).

Una vez sumados todos los valores, la nueva rutina ocupa veintidós bytes y cada iteración del bucle consume ciento ochenta ciclos de reloj, quince más que la implementación anterior, aunque hemos ahorrado seis bytes. Además, la rutina de la ROM tarda menos que posicionarnos usando el código de control de ***AT***.

¿Qué hacemos? ¿Cómo lo dejamos?

Dado que la rutina que pinta el borde no es crítica ya que solo lo pintamos al inicio de cada nivel, y cuatro ciclos de reloj no se van a notar, optamos por el ahorro de seis bytes, nos quedamos con la nueva implementación.

Solo nos queda pintar la nave, vamos a Main.asm y justo debajo de ***CALL PrintInfoGame***, añadimos la llamada a pintar la nave.

```
call    PrintShip
```

Compilamos, cargamos en el emulador y vemos los resultados.

Movemos la nave

La nave se tiene que mover como respuesta a alguna acción del jugador, en nuestro caso a la pulsación de tres teclas: Z para mover la nave hacia la izquierda, X para mover la nave hacia la derecha y V para disparar.

Creamos el archivos Ctrl.asm e implementamos la rutina que lee el teclado y devuelve las teclas de control pulsadas.

La rutina que vamos a implementar lee el teclado y devuelve en el registro D las teclas de control pulsadas, parecido a como se hizo en PorompomPong, poniendo a uno el bit cero si se ha pulsado la tecla Z, el bit uno se ha pulsado la tecla X y el bit dos si se ha pulsado la tecla V.

```
CheckCtrl:
ld          d, $00
ld          a, $fe
in          a, ($fe)
```

Primero ponemos D a cero, ***LD D, $00***, cargamos en A la semifila Cs-V, ***LD A, $FE***, y leemos el teclado, ***IN A, ($FE)***.

```
checkCtrl_fire:
bit         $04, a
jr          nz, checkCtrl_left
set         $02, d
```

Comprobamos si se ha pulsado la tecla V, ***BIT $04, A.*** En el caso de que no se haya pulsado, saltamos a comprobar si se ha pulsado la tecla para mover hacia la izquierda, ***JR NZ, checkCtrl_left***. Si se ha pulsado, pone a uno el bit dos del registro D, marcando que se ha pulsado el disparo, ***SET $02, D***.

Cuando leemos del teclado, el estado de las teclas de la semifila leída está en el registro A, a uno las teclas que no se han pulsado y cero las que sí (el bit cero hace referencia a la tecla más alejada del centro del teclado y el cuatro a la más cercana).

La instrucción ***BIT*** evalúa el estado del bit especificado ($04), del registro especificado (A), y según esté a cero o uno, activa o desactiva el flag Z. La instrucción ***SET*** pone a uno el bit especificado ($02), del registro especificado (D). La instrucción ***RES*** es la contraria a ***SET***, pone el bit a cero. ***RES*** y ***SET*** no afectan al registro F.

```
checkCtrl_left:
bit         $01, a
jr          nz, checkCtrl_right
set         $00, d
```

Comprobamos si se ha pulsado la tecla Z, ***BIT $01, A.*** En el caso de que no se haya pulsado, saltamos a comprobar si se ha pulsado la tecla para mover hacia la derecha, ***JR NZ, checkCtrl_right***. Si se ha pulsado, pone a uno el bit cero del registro D, marcando que se ha pulsado izquierda, ***SET $00, D***.

```
checkCtrl_right:
bit         $02, a
ret         nz
set         $01, d
```

Comprobamos si se ha pulsado la tecla X, ***BIT $02, A.*** En el caso de que no se haya pulsado, salimos. Si se ha pulsado, pone a uno el bit cero del registro D, marcando que se ha pulsado derecha, ***SET $00, D***.

```
checkCtrl_testLR:
ld          a, d
and         $03
sub         $03
ret         nz
ld          a, d
and         $04
ld          d, a


checkCtrl_end:
ret
```

Para finalizar, comprobamos si se ha pulsado al mismo tiempo izquierda y derecha, en cuyo caso desactivamos ambas.

Cargamos el A el valor de D, ***LD A, D***, nos quedamos el valor de los bits cero y uno, ***AND $03***, y le restamos tres, ***SUB $03***. Si el resultado no es cero salimos, ***RET NZ***, ya que no estaban los dos bits a uno y no tenemos que hacer nada.

Si no hemos salido, cargamos de nuevo el valor de D en A, ***LD A, D***, *nos quedamos solo con el valor del bit dos (disparo),* ***AND $04***, y cargamos el valor en D, ***LD D, A***, desactivando de este modo la pulsación simultánea de izquierda y derecha. Finalmente, salimos, ***RET***.

La última etiqueta, ***checkCtrl_end***, no es necesaria, pero la ponemos para clarificar cual es el final de la rutina.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Evalúa si se ha pulsado alguna de la teclas de dirección.
; Las teclas de dirección son:
;     Z      ->     Izquierda
;     X      ->     Derecha
;     V      ->     Disparo
;
; Retorna:  D      ->     Teclas pulsadas.
;                  Bit 0  ->     Izquierda
;                  Bit 1  ->     Derecha
;                  Bit 2  ->     Disparo
;
; Altera el valor de los registros A y D
; ------------------------------------------------------------------------------
CheckCtrl:
ld          d, $00              ; Pone D a 0
ld          a, $fe              ; Carga la semifila Cs-V en A
in          a, ($fe)            ; Leee el teclado


checkCtrl_fire:
bit         $04, a              ; Evalúa si se ha pulsado la V
jr          nz, checkCtrl_left  ; Si no se ha pulsado, salta
set         $02, d              ; Activa el bit 2 de D


checkCtrl_left:
bit         $01, a              ; Evalúa si se ha pulsado la Z
jr          nz, checkCtrl_right ; Si no se ha pulsado, salta
```

```
set         $00, d              ; Activa el bit 0 de D


checkCtrl_right:
bit         $02, a              ; Evalúa si se ha pulsado la X
ret         nz                  ; Si no se ha pulsado, sale
set         $01, d              ; Activa el bit 1 de D


checkCtrl_testLR:
ld          a, d                ; Carga el valor de D en A
and         $03                 ; Se queda con el valor de los bits 0 y 1
sub         $03                 ; Comprueba si están activos los dos bits
ret         nz                  ; Si el resultad no es cero, no están
                                ; activos los dos bits y sale
ld          a, d                ; Carga el valor de D en A
and         $04                 ; Desactiva los bits 0 y 1
ld          d, a                ; Carga el valor de A en D


checkCtrl_end:
ret
```

Abrimos el archivo Main.asm y en cada iteración del bucle ***Main_loop*** vamos a llamar a la rutina que acabamos de implementar. Justo debajo de la etiqueta ***Main_loop*** añadimos la línea siguiente:

```
call    CheckCtrl
```

Un poco más abajo, en la parte donde tenemos los includes, añadimos el include para el archivo Ctrl.asm.

```
include     "Ctrl.asm"
```

Compilamos y comprobamos que compila bien; no hay errores.

Cuando se mueva la nave, primero hay que borrarla de la posición actual y volver a pintarla en la nueva posición.

En Const.asm, justo encima de las constantes que definimos para la nave, añadimos la siguiente constante:

```
; Código de carácter del carácter en blanco
WHITE_GRAPH:EQU $9e
```

Abrimos el archivo Print.asm y, al inicio del mismo, implementamos la rutina que borra la nave. Como esta rutina la vamos a usar también para borrar los enemigos y el disparo, recibe en BC las coordenadas del carácter que vamos a borrar.

```
DeleteChar:
call    At
```

Como DeleteChar recibe en BC las coordenadas del carácter que vamos a borrar, el primer paso es posicionar el cursor, ***CALL At***.

```
ld      a, WHITE_GRAPH
rst     $10


ret
```

Lo siguiente es cargar en A el carácter en banco, ***LD A, WHITE_GRAPH***, y lo pintamos, ***RST $10***, borrando así lo que hubiera pintado en esas coordenadas.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Borra un carácter de la pantalla
;
; Entrada:  BC -> Coordenadas Y/X del carácter
; Altera el valor de los resgistros AF
; ------------------------------------------------------------------------
DeleteChar:
call    At              ; Llama a posicionar el cursor


ld      a, WHITE_GRAPH  ; Carga en A el carácter de blanco
rst     $10             ; Lo pinta y borra la nave


ret
```

Para probar que funciona, abrimos Main.asm y, justo debajo de ***CALL CheckCtrl***, añadimos las líneas siguientes:

```
ld          bc, (shipPos)
call        DeleteChar
call        PrintShip
```

Con estas líneas borramos y pintamos la nave en cada iteración de ***Main_loop***. Si compilamos y cargamos en el emulador, veremos que la nave parpadea constantemente, señal de que el borrado de la nave funciona.

Ahora vamos a mover la nave. Creamos un nuevo archivo, Game.asm, dónde empezamos por implementar la rutina que cambia la posición de la nave y la pinta, esta rutina recibe en el registro D el estado de los controles (antes de continuar añadimos en la sección de includes de Main.asm el archivo Game.asm).

```
MoveShip:
ld      bc, (shipPos)
bit     $01, d
jr      nz, moveShip_right
```

Cargamos la posición actual de la nave en BC, **LD BC, (shipPos)**, comprobamos si se ha pulsado el control derecha, ***BIT $01, D***, en cuyo caso saltamos a la parte que controla el movimiento hacia la derecha, ***JR NZ, moveShip_right***.

```
bit     $00, d
ret     z
```

Comprobamos si se ha pulsado el control izquierda, ***BIT $00, D***, y si no se ha pulsado salimos, ***RET Z***.

```
moveShip_left:
ld      a, SHIP_TOP_L + $01
sub     c
ret     z
call    DeleteChar
inc     c
ld      (shipPos), bc
jr      moveShip_print
```

Si se pulsado el control izquierda, comprobamos si podemos mover la nave. Cargamos en A el tope al que se puede mover la nave hacia la izquierda, ***LD A, SHIP_TOP_L + $01***, y le restamos la columna actual de la posición de la nave, ***SUB C***. Si el resultado es cero, la nave ya está en el tope y no se puede mover más hacia la derecha, así que salimos, ***RET Z***.

Si no hemos salido, borramos la nave de la posición actual, ***CALL DeleteChar***, incrementamos C para apuntar a la columna justo a la izquierda de la posición actual, ***INC C***, actualizamos en memoria la nueva posición de la nave, ***LD (shipPos), BC***, y saltamos al final de la rutina, ***jr moveShip_print***.

La rutina que controla el movimiento de la nave hacia la derecha es prácticamente igual a la que controla el movimiento hacia la izquierda, por lo que solo vamos a marcar y explicar los cambios.

```
moveShip_right:
ld      a, SHIP_TOP_R + $01     ; ¡CAMBIO!
sub     c
ret     z
call    DeleteChar
dec     c                       ; ¡CAMBIO!
ld      (shipPos), bc
```

```
ret
```

Cargamos en A el tope al que se puede mover la nave hacia la derecha, ***LD A, SHIP_TOP_R***. *D*ecrementamos C para apuntar a la columna justo a la derecha de la posición actual, ***DEC C***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Mueve la nave
;
; Entrada:  D -> Estado de los controles
; Altera el valor de los registros AF y BC
; ------------------------------------------------------------------------
MoveShip:
ld      bc, (shipPos)       ; Carga la posición actual de la nave en BC
bit     $01, d              ; Comprueba si el control derecha viene activo
jr      nz, moveShip_right  ; En cuyo caso, sale

bit     $00, d              ; Comprueba si el control izquierda viene activo
ret     z                   ; Si no es así, sale

moveShip_left:
ld      a, SHIP_TOP_L + $01 ; Carga en A el tope para la nave por la izquierda
sub     c                   ; Le resta la columna actual de la nave
ret     z                   ; Si es la misma columna, sale
call    DeleteChar          ; Borra la nave de la posición actual
inc     c                   ; Apunta C a la culumna a la izquierda a la actual
ld      (shipPos), bc       ; Actualiza la posición de la nave
jr      moveShip_print      ; Salta al final de la rutina

moveShip_right:
ld      a, SHIP_TOP_R + $01 ; Carga en A el tope para la nave por la derecha
sub     c                   ; Le resta la columna actual de la nave
ret     z                   ; Si es la misma columna, sale
call    DeleteChar          ; Borra la nave de la posición actual
dec     c                   ; Apunta C a la culumna a la derecha a la actual
ld      (shipPos), bc       ; Actualiza la posición de la nave

moveShip_print:
call    PrintShip           ; Pintamos la nave
```

```
ret
```

Antes de continuar, recordemos que anteriormente comentamos que At alteraba el valor de los resgistros BC y AF pero que no nos afectaba. Ahora que At se llama desde varios puntos, el cambio del registro BC si que nos afecta. La solución es tan sencilla como añadir a At ***PUSH BC*** y ***POP BC*** para preservar y recuperar el valor de BC, aunque vamos a hacer otra implementación y de paso vamos a ahorrar bytes y ciclos de reloj.

La nueva implementación de At queda de la siguiente manera:

```
; ------------------------------------------------------------------------------
; Posiciona el cursor en las coordenadas especificadas.
;
; Entrada:  B = Coordenada Y (24 a 3).
;           C = Coordenada X (32 a 1).
; Altera el valor de los registros AF
; ------------------------------------------------------------------------------
At:
push    bc      ; Preservamos el valor de BC
exx             ; Preservamos el valor de BC, DE y HL
pop     bc      ; Recuperamos el valor de BC
call    $0a23   ; Llama a la rutina de la ROM
exx             ; Recuperamos el valor de BC, DE y HL


ret
```

Preservamos el valor de BC que es donde están las coordenadas, ***PUSH BC***, preservamos el valor de los registros BC, DE y HL intercambiando su valor con los de los registros alternativos, ***EXX***, recuperamos el valor de BC (coordenadas) de la pila, ***POP BC***, y llamamos a la rutina de la ROM que posiciona el cursor, ***CALL $0A23***.

Llegados a este punto, el valor de BC, DE y HL ha cambiado, lo recuperamo desde los registros alternativos, ***EXX***, y salimos, ***RET***.

La rutina ahora ocupa ocho bytes y tarda cincuenta y seis ciclos de reloj en ejecutarse, frente a los diez bytes y noventa ciclos de reloj que ocuparía usando la pila para los tres registros.

Es hora de comprobar si se mueve la nave, volvemos a Main.asm y sustituimos las líneas que hemos añadido antes:

```
ld      bc, (shipPos)
call    DeleteShip
call    PrintShip
```

por:

```
call    MoveShip
```

Compilamos, cargamos en el emulador y vemos los resultados.

La nave se mueve tanto a izquierda como a derecha, pero volvemos a tener el mismo problema que tuvimos en PorompomPong, se mueve extremadamente rápido, más rápido que las palas de Porompompong, ya que las palas se movían píxel a píxel y la nave se mueve carácter a carácter.

Podríamos solucionarlo igual que hicimos entonces, no moviendo la nave en cada iteración del bucle, pero dado que vamos a usar las interrupciones para más cosas, lo vamos a hacer a través de ellas y así vemos algo que no vimos en PorompomPong.

Conclusión

Ya tenemos la nave en el área de juego y la movemos, pero hemos observado un problema que ya tuvimos en PorompomPong, se mueve extremadamente rápido.

En el próximo capítulo solucionaremos esto con las interrupciones e implementaremos la parte del disparo.

0x05 Interrupciones y disparo

En este capítulo vamos a implementar el manejo de las interrupciones; si queréis saber más sobre ellas os recomiendo que leáis el capítulo dedicado a la mismas en el curso de Compiler Software, y la forma de implementarlas para el modelo 16K.

El ZX Spectrum genera un total de cincuenta interrupciones por segundo en sistemas PAL, y sesenta en sistemas NTSC.

Creamos la carpeta Paso05 y copiamos desde la carpeta Paso04 los archivos Const.asm, Ctrl.asm, Game.asm, Graph.asm, Main.asm, Print.asm y Var.asm.

Antes de continuar comprobamos cuánto ocupa nuestro programa, veremos que ronda los mil seiscientos bytes.

Interrupciones

La rutina que se ejecuta cuando se genera una interrupción la vamos a implementar en el archivo Int.asm, así que lo creamos.

Siguiendo lo explicado en el curso de Compiler Software, vamos a cargar $28 (40) en el registro I y nuestra rutina en la dirección $7e5c (32348), con lo que dejamos cuatrocientos diecinueve bytes para la rutina. Teniendo en cuenta que el programa lo cargamos en $5dad (23981), nos quedan ocho mil trescientos sesenta y siete bytes para nuestro juego.

Abrimos el archivo Main.asm y, justo antes de ***Main_loop***, añadimos las líneas para preparar las interrupciones.

```
di
ld      a, $28
ld      i, a
im      2
ei
```

Deshabilitamos las interrupciones, ***DI***, cargamos $28 (40) en el registro A, ***LD A, $28***, y cargamos A en el registro I, ***LD I, A***. Cambiamos al modo de interrupción a dos, ***IM 2***, y activamos las interrupciones, ***EI***.

La rutina la vamos a implementar en el archivo Int.asm, de manera que lo abrimos y añadimos las líneas siguientes:

```
org     $7e5c


Isr:
push    hl
push    de
push    bc
push    af
```

Cargamos la rutina ***Isr*** en la dirección $7E5C (32348), ***ORG $7E5C*** y preservamos el valor de HL, DE, BC y AF, ***PUSH HL, PUSH DE, PUSH BC, PUSH AF***.

De momento no hacemos nada más y salimos.

```
Isr_end:
pop         af
pop         bc
pop         de
pop         hl
ei
reti
```

Recuperamos el valor de AF, BC, DE y HL, ***POP AF, POP BC, POP DE, POP HL***, activamos las interrupciones, ***EI***, y salimos, ***RETI***.

Ahora vamos a Main.asm y, al final, justo antes de ***END Main***, incluímos el archivo Int.asm.

Compilamos, cargamos en el emulador y probamos. Aparentemente no pasa nada, pero ¿qué pasa si comprobamos lo que ocupa ahora nuestro programa? Pues que ocupa la friolera de más de nueve mil bytes. ¿Cómo es esto posible? Al cargar la rutina en la dirección $7E5C, PASMO rellena con ceros todo el espacio desde dónde acababa el programa anteriormente hasta dónde acaba ahora, por eso ocupa tanto. Si cargamos en un emulador no suele importar, la carga la podemos hacer inmediata, pero en un ZX Spectrum real estamos añadiendo tiempo de carga innecesariamente.

Compilamos en múltiples ficheros

Para evitar que nuestro programa crezca innecesariamente, vamos a compilar por separado el archivo Int.asm y el resto, y también vamos a prescindir del cargador que nos genera PASMO y vamos a hacer el nuestro propio.

Creación del cargador

Desde el emulador vamos a generar un cargador BASIC personalizado, que vamos a grabar como el archivo Cargador.tap. El código del cargador es el siguiente:

```
10 CLEAR 23980
20 LOAD “”CODE
30 LOAD “”CODE 32348
40 RANDOMIZE USR 23981
```

Grabamos nuestro cargador con la siguiente instrucción:

```
SAVE “BATALLAESP” LINE 10
```

De esta manera, al cargar el programa, se auto ejecuta en la línea diez.

Compilación en varios ficheros

Vamos a compilar por separado el archivo Int.asm, generando el archivo Int.tap, y el resto del programa generando el archivo Marciano.tap. Por último, vamos a concatenar los ficheros Cargador.tap, Marciano.tap e Int.tap en el fichero BatallaEspacial.tap.

Dado que realizar estas operaciones cada vez que compilemos y queramos ver los resultados es tedioso, vamos a crear un script; para los que uséis Windows, cread en la carpeta Paso05 el archivo make.bat, para los que usamos Linux, ejecutamos desde la línea de comandos:

```
touch make
```

Luego damos permisos de ejecución.

```
chmod +x make
```

Antes de seguir, abrimos el archivo Main.asm y borramos, casi al final, el include del archivo Int.asm.

Y ahora podemos editar el archivo make o make.bat. Las dos primeras líneas son comunes tanto en Windows como en Linux.

```
pasmo --name Marciano --tap Main.asm Marciano.tap --public
pasmo --name Int --tap Int.asm Int.tap
```

Primero compilamos el programa y luego compilamos el archivo Int.asm y generamos el arhivo Int.tap. Observad que en lugar de --tapbas hemos puesto –tap, pues el cargador BASIC lo hemos hecho a mano.

Por último, combinamos Cargador.tap, Marciano.tap e Int.tap en BatallaEspacial.tap.

Para los que usamos Linux, añadimos la línea siguiente al final del archivo make:

```
cat Cargador.tap Marciano.tap Int.tap > BatallaEspacial.tap
```

Para los que usáis Windows, la línea que tenéis que añadir es la siguiente:

```
copy Cargador.tap+Marciano.tap+Int.tap BatallaEspacial.tap
```

A partir de este momento, la manera de compilar será ejecutando make o make.bat, y en el emulador cargaremos el archivo BatallaEspacial.tap.

Compilamos, cargamos en el emulador y vemos que todo sigue igual, pero el tamaño de BatallaEspacial.tap esta muy por debajo de nueve mil bytes.

Ralentizamos la nave

Vimos en la entrega anterior que la nave se movía muy rápido, para solucionar esta cuestión vamos a usar las interrupciones para mover la nave un máximo de cincuenta veces por segundo (en sistemas PAL, sesenta en NTSC), es decir, vamos a mover la nave cuándo salte la interrupción.

Abrimos el archivo Var.asm y al inicio del mismo añadimos lo siguiente:

```
; ------------------------------------------------------------------------
; Indicadores
;
; Bit 0 -> se debe mover la nave            0 = No, 1 = Si
; ------------------------------------------------------------------------
flags:
db $00
```

Volvemos a Int.asm y tras ***PUSH AF*** añadimos las líneas siguientes:

```
ld          hl, flags
set         $00, (hl)
```

Cargamos en HL la dirección de memoria de flags, ***LD HL, flags***, y ponemos el bit cero a uno, ***SET $00, (HL)***.

Ahora vamos al archivo Game.asm y justo debajo de la etiqueta ***MoveShip***, añadimos las líneas siguientes:

```
ld          hl, flags           ; Cargamos la dirección de memoria de flags en HL
bit         $00, (hl)           ; Comprueba si el bit 0 está activo
ret         z                   ; Si no es así, sale
res         $00, (hl)           ; Desactiva el bit 0 de flags
```

Cargamos en HL la dirección de memoria de flags, ***LD HL, flags***, comprobamos si hay que mover la nave, ***BIT $00, (HL)***, y si no es así salimos, ***RET Z***. Si hay que mover la nave ponemos el bit cero a cero, ***SET $00, (HL)***, de esta manera no volveremos a mover la nave hasta que salte una interrupción y vuelva a poner a uno el bit cero de flags.

Esto que hemos implementado hará que nuestra nave se mueva cincuenta veces por segundo (o sesenta, según el sistema), así que compilamos y …

Efectivamente, tenemos errores de compilación.

ERROR on line 10 of file Int.asm

ERROR: Symbol 'flags' is undefined

Hasta ahora, en el archivo Main.asm incluíamos todos los archivos .asm que tenemos, pero hemos quitado el ***include*** del archivo Int.asm para compilarlo por separado, por lo que en Int.asm no se conoce la etiqueta flags. La solución es sencilla, en Int.asm hay que sustituir ***LD HL, flags*** por ***LD HL, direccionMemoria***.

Echemos un vistazo a la línea que usamos para compilar.

```
pasmo --name Marciano --tap Main.asm Marciano.tap --public
```

El último parámetro, ***--public***, genera un fichero con ese nombre donde podemos ver cada una de las etiquetas de nuestro programa, en que dirección de memoria se encuentran. En mi caso, ***flags***

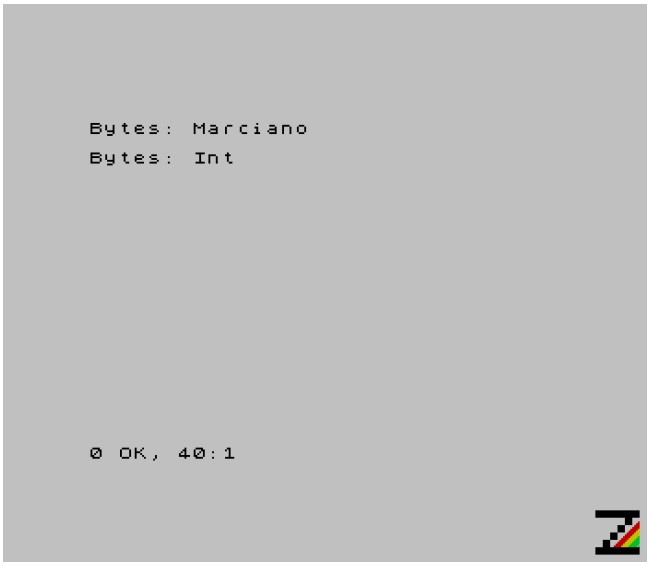
está en la dirección de memoria $5dee, por lo que solo hay que ir a Int.asm y sustituir ***LD HL, flags*** por ***LD HL, $5DEE***.

Ahora sí, compilamos, cargamos en el emulador y vemos que la nave se mueve más lenta, pero seguimos teniendo un problema; poned una instrucción ***NOP*** al inicio de Main.asm, justo antes de la etiqueta ***Main***.

Compilad y veréis que compila bien, cargad en el emulador y veréis que ha dejado de funcionar. Si ahora vais a --public, veréis que la etiqueta ***flags*** está en la dirección $5DEF, sin embargo en el archivo Int.asm el valor que se carga en HL es $5DEE. Cada vez que modifiquemos algo de código, es muy probable que la dirección de memoria de ***flags*** cambie, así que tenemos que asegurarnos de cambiarla también en Int.asm.

Para evitar que cambie la dirección de memoria de ***flags***, vamos a abrir el archivo Var.asm, vamos a cortar la declaración de ***flags*** y la vamos a pegar al inicio del archivo Main.asm, justo debajo de ***ORG $5DAD***, de esta manera nos aseguramos que ***flags*** siempre esté en la dirección de memoria $5DAD; no olvidéis sustituir $5DEE por $5DAD en el archivo Int.asm.

Compilamos, cargamos en el emulador y todo vuelve a funcionar, pero cuidado, funciona porque hemos inicializado ***flags*** a cero, que es el código de la instrucción ***NOP***, que lo que hace es tardar cuatro ciclos de reloj en ejecutarse, nada más. Si la iniciáramos con otro valor, por ejemplo $C9, el programa nada más ejecutarse volvería al BASIC, ya que $C9 es ***RET***, probad y veréis.



Esto tiene fácil solución, antes de la etiqueta ***flags***, añadimos ***JR Main***, aunque dado que solo vamos a necesitar la etiqueta ***flags*** y la iniciamos a cero, no tenemos problema, pero tened cuidado con esto.

No es necesario añadir ***JR Main,*** En caso de añadirlo, la dirección de ***flags*** cambia.

Implementamos el disparo

Igual que hicimos con la nave, vamos a incluir las constantes necesarias para el manejo del disparo en el archivo Const.asm.

```
; Código de carácter del disparo y tope
```

```
FIRE_GRAPH: EQU $91
FIRE_TOP_T: EQU COR_Y
```

De igual manera, en el archivo Var.asm vamos a añadir una etiqueta para guardar la posición actual del disparo.

```
; ------------------------------------------------------------------------
; Disparo
; ------------------------------------------------------------------------
firePos:
dw $0000
```

En Print.asm vamos a implementar la rutina que pinta el disparo.

```
PrintFire:
ld          a, $02
call        Ink
```

Cargamos en A el color de tinta rojo, ***LD A, $02***, y llamamos a cambiar la tinta, ***CALL Ink***.

```
ld          bc, (firePos)
call        At
```

Cargamos en BC la posición actual del disparo, ***LD BC, (firePos)***, y llamamos a posicionar el cursor, ***CALL At***.

```
ld          a, FIRE_GRAPH
rst         $10


ret
```

Cargamos en A el código de carácter del disparo, ***LD A, FIRE_GRAPH***, lo pintamos, ***RST $10***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta el disparo en la posición actual.
; Altera el valor de los registros AF y BC.
; ------------------------------------------------------------------------
PrintFire:
ld          a, $02              ; Carga en A la tinta roja
call        Ink                 ; Llama al cambio de tinta


ld          bc, (firePos)       ; Carga en BC la posición actual del disparo
```

```
call        At                  ; Llama a posicionar el cursor


ld          a, FIRE_GRAPH       ; Carga en A el carácter del fuego
rst         $10                 ; Lo pinta


ret
```

Implementamos en Game.asm la rutina que mueve el disparo.

```
MoveFire:
ld      hl, flags
bit     $01, (hl)
jr      nz, moveFire_try
bit     $02, d
ret     z
set     $01, (hl)
ld      bc, (shipPos)
inc     h
jr      moveFire_print
```

Cargamos la dirección de memoria de ***flags*** en HL, ***LD HL, flags***, comprobamos si el bit uno (disparo) está activo, ***BIT $01, (HL)***, y de ser así saltamos, ***JR NZ, moveFire_try***. Si el bit uno no esta activo, comprobamos si se ha pulsado el control disparo, ***BIT $02, D***, y de no ser así salimos, ***RET Z***. En caso de haberse pulsado, activamos el bit de disparo en ***flags***, ***SET $01, (HL)***, cargamos la posición actual de la nave en BC, ***LD BC, (shipPos)***, apuntamos a la línea superior, ***INC B***, y saltamos a pintarlo, ***JR moveFire_print***.

```
moveFire_try:
ld      bc, (firePos)
call    DeleteChar
inc     b
ld      a, FIRE_TOP_T
sub     b
jr      nz, moveFire_print
res     $01, (hl)


ret
```

Si el disparo estaba activo, cargamos la posición del disparo en BC, ***LD BC, (firePos)***, borramos el disparo, ***CALL DeleteChar***, incrementamos B para apuntar a la línea superior, ***INC B***, cargamos en A el tope superior del disparo, ***LD A, FIRE_TOP_T***, y le restamos B***, SUB B***, si el resultado no es cero, todavía no hemos llegado al tope y saltamos, ***JR NZ, moveFire_print***. Si hemos llegamos al tope desactivamos el disparo, ***RES $01, (HL)***, y salimos, ***RET***.

```
moveFire_print:
ld      (firePos), bc
call    PrintFire


ret
```

Si no hemos llegado al tope o acabamos de activar el disparo, actualizamos la posición actual del disparo, ***LD (firePos), BC***, llamamos a pintar el disparo, ***CALL PrintFire***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Mueve el disparo
;
; Entrada:  D -> Estado de los controles
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------
MoveFire:
ld      hl, flags           ; Carga en HL la dirección de memoria de flags
bit     $01, (hl)           ; Evalúa si el bit del disparo está activo
jr      nz, moveFire_try    ; Si está activo, salta
bit     $02, d              ; Evalúa si el control de disparo está activo
ret     z                   ; Si no está activo, sale
set     $01, (hl)           ; Activa el bit del disparo en flags
ld      bc, (shipPos)       ; Carga la posición actual de la nave en HL
inc     b                   ; Apunta a la línea superior
jr      moveFire_print      ; Salta a pintar el diparo


moveFire_try:
ld      bc, (firePos)       ; Carga en BC la posición actual del disparo
call    DeleteChar          ; Borra el disparo
inc     b                   ; Apunta B a la línea superior
ld      a, FIRE_TOP_T       ; Carga en A el tope superior del disparo
sub     b                   ; Le restamos coordenada Y del disparo
jr      nz, moveFire_print  ; Si son distintos, no ha llegado al tope, salta
res     $01, (hl)           ; Desactiva el disparo


ret


moveFire_print:
```

```
ld      (firePos), bc       ; Actualiza la posición del disparo
call    PrintFire           ; Pinta el disparo


ret
```

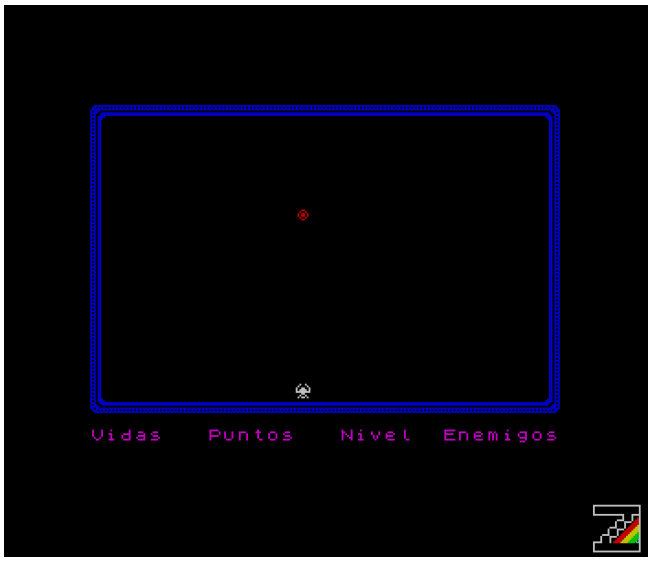
Es hora de probar el disparo, abrimos el achivo Main.asm y al inicio, en la declaración de **flags**, añadimos el comentario para el bit uno.

```
; Bit 1 -> el disparo está activo           0 = No, 1 = Sí
```

Localizamos la etiqueta ***Main_loop***, y entre las líneas ***CALL CheckCtrl*** y ***CALL MoveShip*** añadimos la llamada al movimiento del disparo.

```
call    MoveFire
```

Compilamos, cargamos en el emulador y vemos los resultados.



En la imagen no se aprecia, pero en el emulador podemos ver que parece que el disparo realiza ráfagas, y si dejamos el disparo pulsado es como si no parase de disparar, es un efecto óptico debido a que movemos el disparo más deprisa de lo que la ULA refresca la pantalla, lo vamos a dejar así para que parezca que disparamos varias veces a un mismo tiempo.

Conclusión

Hemos empezado a trabajar con interrupciones, temporizado el movimiento de la nave e implementado el disparo, además de compilar el programa en varios ficheros y personalizar el cargador.

En el próximo capítulo introduciremos los enemigos.

0x06 Enemigos

En este capítulo vamos a incluir los enemigos. Lo primero es crear la carpeta Paso06, copiamos desde la carpeta Paso05 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, Print.asm, Var.asm y make, o make.bat si estáis trabajando en Windows.

Definimos los enemigos

Los enemigos son elementos móviles, y como tales necesitamos saber la posición actual y la inicial. En total vamos a tener un máximo de veinte enemigos en pantalla, y vamos a usar dos bytes para especificar la posición actual del enemigo y alguna configuración más que nos va a hacer falta.

Abrimos el archivo Var.asm y tras la definición del marco de la pantalla, añadimos la configuración de los enemigos.

```
; ------------------------------------------------------------------------------
; Configuración de los enemigos
;
; 2 bytes por enemigo.
; ------------------------------------------------------------------------------
; Byte 1                        | Byte 2
; ------------------------------------------------------------------------------
; Bit 0-4:  Posición Y          | Bit 0-4:  Posición X
; Bit 5:    Libre               | Bit 5:    Libre
; Bit 6:    Libre               | Bit 6:    Dirección X 0 = Left 1 = Right
; Bit 7:    Activo 1/0          | Bit 7:    Dirección Y 0 = Up   1 = Down
; ------------------------------------------------------------------------------
enemiesConfig:
db $96, $dd, $96, $d7, $96, $d1, $96, $cb, $96, $c5
db $93, $9d, $93, $97, $93, $91, $93, $8b, $93, $85
db $90, $dd, $90, $d7, $90, $d1, $90, $cb, $90, $c5
db $8d, $9d, $8d, $97, $8d, $91, $8d, $8b, $8d, $85
enemiesConfigIni:
db $96, $dd, $96, $d7, $96, $d1, $96, $cb, $96, $c5
db $93, $9d, $93, $97, $93, $91, $93, $8b, $93, $85
db $90, $dd, $90, $d7, $90, $d1, $90, $cb, $90, $c5
db $8d, $9d, $8d, $97, $8d, $91, $8d, $8b, $8d, $85
enemiesConfigEnd:
```

En el primer byte vamos a tener la coordenada Y, bits del cero al cuatro, y si el enemigo está o no activo, bit siete. En el segundo byte vamos a tener la coordenada X, bits del cero al cuatro, la dirección horizontal, bit seis, y la dirección vertical, bit siete.

Según los bits seis y siete del segundo byte, la dirección del enemigo es:

- 00b $00 Izquierda / Arriba
- 01b $01 Izquierda / Abajo
- 10b $02 Derecha / Arriba
- 11b $03 Derecha / Abajo

Basándonos en esto, vamos a añadir la definición de los gráficos de los enemigos. Justo antes de ***enemiesConfig*** añadimos dicha definición.

```
; ------------------------------------------------------------------------------
; Gráficos de los enemigos
;
; 00 Up-Left
; 01 Up-Right
; 10 Down-Left
; 11 Down-Right
; ------------------------------------------------------------------------------
enemiesGraph:
db $9f, $a0, $a1, $a2
```

Si buscamos los UDG de los enemigos, veremos que todo concuerda.

Volviendo a la definición de los enemigos, vamos a tener un total de veinte enemigos, repartidos en cuatro filas con cinco enemigos por fila.

Vamos a ver la definición de los enemigos de izquierda a derecha y de arriba a abajo; recordad que trabajamos con las coordenadas invertidas.

Hexadecimal	Binario	Definición
$96, $dd	10010110, 11011101	Activo Línea 22 Abajo/Derecha Columna 29
$96, $d7	10010110, 11010111	Activo Línea 22 Abajo/Derecha Columna 23
$96, $d1	10010110, 11010001	Activo Línea 22 Línea/Derecha Columna 17
$96, $cb	10010110, 11001011	Activo Línea 22 Abajo/Derecha Columna 11

Hexadecimal	Binario	Definición
$96, $c5	10010110, 11000101	Activo Línea 22 Abajo/Derecha Columna 5
$93, $9d	10010011, 10011101	Activo Línea 19 Abajo/Izquierda Columna 29
$93, $97	10010011, 10010111	Activo Línea 19 Abajo/Izquierda Columna 23
$93, $91	10010011, 10010001	Activo Línea 19 Abajo/Izquierda Columna 17
$93, $8b	10010011, 10001011	Activo Línea 19 Abajo/Izquierda Columna 11
$93, $85	10010011, 10000101	Activo Línea 19 Abajo/Izquierda Columna 5
$90, $dd	10010000, 11011101	Activo Línea 16 Abajo/Derecha Columna 29
$90, $d7	10010000, 11010111	Activo Línea 16 Abajo/Derecha Columna 23
$90, $d1	10010000, 11010001	Activo Línea 16 Abajo/Derecha Columna 17
$90, $cb	10010000, 11001011	Activo Línea 16 Abajo/Derecha Columna 11
$90, $c5	10010000, 11000101	Activo Línea 16 Abajo/Derecha Columna 5

Hexadecimal	Binario	Definición
$8d, $9d	10001101, 10011101	Activo Línea 13 Abajo/Izquierda Columna 29
$8d, $97	10001101, 10010111	Activo Línea 13 Abajo/Izquierda Columna 23
$8d, $91	10001101, 10010001	Activo Línea 13 Abajo/Izquierda Columna 17
$8d, $8b	10001101, 10001011	Activo Línea 13 Abajo/Izquierda Columna 11
$8d, $85	10001101, 10000101	Activo Línea 13 Abajo/Izquierda Columna 5

Una vez definidos los gráficos de los enemigos y su configuración, podemos proceder a pintarlos.

Pintamos los enemigos

La rutina que pinta los enemigos la vamos a implementar en Print.asm.

```
PrintEnemies:
ld      a, $06
call    Ink


ld      hl, enemiesConfig
ld      d, $14
```

Cargamos en A la tinta amarilla, ***LD A, $06***, cambiamos la tinta, ***CALL Ink***, cargamos la dirección de memoria de los enemigos en HL, ***LD HL, enemiesConfig***, y el número total de enemigos en D, ***LD D, $14***, veinte enemigos.

```
printEnemies_loop:
bit     $07, (hl)
jr      z, printEnemies_endLoop
```

Evaluamos si el enemigo está activo, ***BIT $07, (HL)***, y de no estarlo saltamos, ***JR Z, printEnemies_endLoop***.

```
push    hl


ld      a, (hl)
and     $1f
ld      b, a
```

Preservamos el valor de HL, ***PUSH HL***, cargamos el primer byte de la configuración del enemigo en A, ***LD A, (HL)***, nos quedamos con la coordenada Y, ***AND $1F***, y la cargamos en B, ***LD B, A***.

```
inc     hl
ld      a, (hl)
and     $1f
ld      c, a
call    At
```

Apuntamos HL al segundo byte de la configuración del enemigo, ***INC HL***, cargamos el valor en A, ***LD A, (HL)***, nos quedamos con la coordenada X, ***AND $1F***, cargamos el valor en C, ***LD C, A***, y posicionamos el cursor, ***CALL At***.

```
ld      a, (hl)
and     $c0
rlca
rlca
ld      c, a
ld      b, $00
```

Cargamos el segundo byte de la configuración del enemigo en A, ***LD A, (HL)***, nos quedamos con los bits de dirección, ***AND $c0***, pasamos el valor a los bits cero y uno, ***RLCA RLCA***, cargamos el valor en C, ***LD C, A***, y ponemos B a cero, ***LD B, $00***.

```
ld      hl, enemiesGraph
add     hl, bc
ld      a, (hl)
rst     $10
```

Cargamos en HL la dirección de memoria en la que definimos los caracteres para los enemigos, ***LD HL, enemiesGraph***, le sumamos la dirección (izquierda, arriba ...) del enemigo, ***ADD HL, BC***, cargamos en A el carácter del enemigo que hay que pintar, ***LD A, (HL)***, y lo pintamos, ***RST $10***.

```
pop      hl


printEnemies_endLoop:
inc     hl
inc     hl
```

```
dec     d
jr      nz, printEnemies_loop

ret
```

Recuperamos el valor de HL, ***POP HL***, apuntamos HL al primer byte de la configuración del siguiente enemigo, ***INC HL INC HL***, decrementamos D, ***DEC D***, y seguimos hasta que D sea cero y hayamos recorrido todos los enemigos. Finalmente, salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta los enemigos
;
; Altera el valor de los registros AF, BC, D y HL.
; ------------------------------------------------------------------------
PrintEnemies:
ld      a, $06                      ; Carga en A la tinta amarilla
call    Ink                         ; Cambia la tinta

ld      hl, enemiesConfig           ; Carga la dirección de la configuración
                                    ; del enemigo en HL
ld      d, $14                      ; Carga en D 20 enemigos

printEnemies_loop:
bit     $07, (hl)                   ; Evalúa si el enemigo está activo
jr      z, printEnemies_endLoop     ; Si no lo está, salta

push    hl                          ; Preserva el valor de HL

ld      a, (hl)                     ; Carga el primer byte de configuración en A
and     $1f                         ; Se queda con la coordenda Y
ld      b, a                        ; La carga en B

inc     hl                          ; Apunta HL al segundo byte
ld      a, (hl)                     ; Carga el valor en A
and     $1f                         ; Se queda con la coordenada X
ld      c, a                        ; La carga en C
call    At                          ; Posiciona el cursor
```

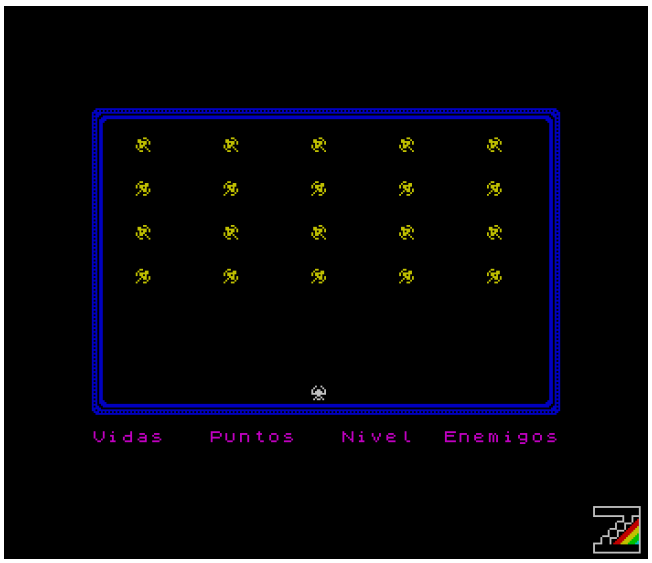
```
ld      a, (hl)                     ; Vuelve a cargar el segudo byte en A
and     $c0                         ; Se queda con la dirección (izquierda ...)
rlca                                ; Pone el valor en los bits 0 y 1
rlca
ld      c, a                        ; Carga el valor en C
ld      b, $00                      ; Pone B a cero

ld      hl, enemiesGraph            ; Carga en HL el carácter del gráfico del enemigo
add     hl, bc                      ; Le suma la dirección de enemigo (izquierda ...)
ld      a, (hl)                     ; Carga en A el gráfico del enemigo
rst     $10                         ; Lo pinta

pop      hl                         ; Recupera el valor de HL

printEnemies_endLoop:
inc     hl                          ; Apunta HL al primer byte de la configuración
inc     hl                          ; del enemigo siguientes

dec     d                           ; Decrementa D
jr      nz, printEnemies_loop       ; hasta que D sea 0

ret
```

Para probar si funciona, vamos a ir a Main.asm y justo antes de Main_loop vamos a añadir las líneas siguientes:

```
ld      a, $01
call    LoadUdgsEnemies
call    PrintEnemies
```

Cargamos el nivel uno en A, ***LD A, $01***, cargamos los gráficos de los enemigos del nivel uno en udgsExtension, ***CALL LoadUdgsEnemies***, y pintamos los enemigos, ***CALL PrintEnemies***.

Compilamos, cargamos en el emulador y vemos los resultados.



Movemos los enemigos

Los enemigos no los vamos a mover en cada iteración del bucle, como hacemos con el disparo, los vamos a mover cada N interrupciones, por lo qué lo primero que vamos a hacer es añadir otro comentario en la etiqueta ***flag***, al inicio de Main.asm.

```
; Bit 2 -> se deben mover los enemigos      0 = No, 1 = Sí
```

Lo siguiente es establecer los límites de la pantalla hasta dónde los enemigos pueden llegar; estos límites los vamos a establecer en Const.asm.

```
; Topes de los enemigos
ENEMY_TOP_T:   EQU COR_Y - MIN_Y
ENEMY_TOP_B:   EQU COR_Y - MAX_Y + $01
ENEMY_TOP_L:   EQU COR_X - MIN_X
ENEMY_TOP_R:   EQU COR_X - MAX_X
```

Los límites que hemos establecido son arriba, abajo, izquierda y derecha.

Para que los enemigos se muevan cada N interrupciones, y como en ***flags*** vamos a usar un bit para indicar si se deben mover o no, vamos a ir al archivo Int.asm y vamos a activar dicho bit, para lo que añadimos la siguiente línea justo debajo de ***SET $00, (HL)***:

```
set     $02, (hl)
```

Ya tenemos todos listo para poder implementar en Game.asm la rutina que mueve los enemigos.

```
MoveEnemies:
ld      hl, flags
bit     $02, (hl)
ret     z
res     $02, (hl)
```

Cargamos en HL la dirección de memoria de ***flags***, ***LD HL, flags***, comprobamos si el bit de movimiento de los enemigos está activo, ***BIT $02, (HL)***, y si no está activo, salimos, ***RET Z***. En el caso de estar activo, lo desactivamos para que no pase por aquí en la próxima iteración de ***Main_loop***, ***RES $02, (HL)***.

```
ld      d, $14
ld      hl, enemiesConfig


moveEnemies_loop:
bit     $07, (hl)
jr      z, moveEnemies_loopEnd
```

Cargamos en D el número total de enemigos (veinte), ***LD D, $14***, cargamos en HL la dirección de memoria de la configuración de los enemigos, ***LD HL, enemiesConfig***, comprobamos si el enemigo está activo, ***BIT $07, (HL)***, y si no está activo saltamos al final del bucle, ***JR Z, moveEnemies_loopEnd***.

```
push    hl


ld      a, (hl)
and     $1f
ld      b, a


inc     hl
ld      a, (hl)
and     $1f
ld      c, a


call    DeleteChar


pop     hl
```

Preservamos el valor de HL, ***PUSH HL***, cargamos en A el primer byte de la configuración del enemigo, ***LD A, (HL)***, nos quedamos con la coordenada Y, ***AND $1F***, y la cargamos en B, ***LD B, A***.

Apuntamos HL al segundo byte de la configuración del enemigo, ***INC HL***, cargamos el valor en A**,** ***LD A, (HL)***, nos quedamos con la coordenada X, ***AND $1F***, y la cargamos en C, ***LD C, A***.

Borramos el enemigo, ***CALL DeleteChar***, y recuperamos el valor de HL, ***POP HL***.

```
ld      b, (hl)
inc     hl
ld      c, (hl)
```

Cargamos el primer byte de la configuración del enemigo en B, ***LD B, (HL)***, apuntamos HL al segundo byte de la configuración, ***INC HL***, y cargamos el valor en C, ***LD C, (HL)***.

```
moveEnemies_X:
ld      a, c
and     $1f

bit     $06, c
jr      nz, moveEnemies_X_right
```

Cargamos el valor del segundo byte de la configuración del enemigo en A, ***LD A, C***, y nos quedamos con la coordenada X, ***AND $1F***.

Comprobamos el bit de dirección horizontal del enemigo, ***BIT $06, C***, si está a uno, el enemigo se desplaza hacia la derecha y salta, ***JR Z, moveEnemies_X_right***. Si no ha saltado, el enemigo se mueve hacia la izquierda.

```
moveEnemies_X_left:
inc     a
sub     ENEMY_TOP_L
jr      z, moveEnemies_X_leftChg

inc     c
jr      moveEnemies_Y

moveEnemies_X_leftChg:
set     $06, c
jr      moveEnemies_Y
```

Incrementamos A para que apunte a la columna a la izquierda de la actual, ***INC A***, restamos el tope por la izquierda, ***SUB ENEMY_TOP_L***, y si el resultado es cero, ha llegado al tope y salta para cambiar la dirección, ***JR Z, moveEnemies_X_leftChg***.

Si no hay que cambiar la dirección, apunta C a la columna a la izquierda de la actual, ***INC C***, y salta al movimiento vertical, ***JR moveEnemies_Y***.

Si hay que cambiar la dirección, activa el bit seis de C para poner dirección horizontal hacia la derecha, ***SET $06, C***, y salta al movimiento vertical, ***JR moveEnemies_Y***.

Si el enemigo no se mueve hacia la izquierda, se mueve hacia la derecha.

```
moveEnemies_X_right:
dec     a
sub     ENEMY_TOP_R
jr      z, moveEnemies_X_rightChg
```

```
dec     c
jr      moveEnemies_Y


moveEnemies_X_rightChg:
res     $06, c
```

Decrementamos A para que apunte a la columna a la derecha de la actual, ***DEC A***, *restamos el tope por la derecha,* ***SUB ENEMY_TOP_R***, y si el resultado es cero, ha llegado al tope y salta para cambiar la dirección, ***JR Z, moveEnemies_X_rightChg***.

Si no hay que cambiar la dirección, apunta C a la columna a la derecha de la actual, ***DEC C***, y salta al movimiento vertical, ***JR moveEnemies_Y***.

Si hay que cambiar la dirección, desactiva el bit seis de C para poner dirección horizontal hacia la izquierda, ***RES $06, C***, y empezamos con el movimiento vertical.

```
moveEnemies_Y:
ld      a, b
and     $1f
bit     $07, c
jr      nz, moveEnemies_Y_down
```

Cargamos en A el valor del primer byte de la configuración del enemigo, ***LD A, B***, y nos quedamos con la coordenada Y***, AND $1F***.

Comprobamos el bit siete de C para saber la dirección vertical del enemigo, ***BIT $07, C***, y si está a uno el enemigo se mueve hacia abajo y salta, ***JR NZ, moveEnemies_Y_down***.

Si el bit está a cero, el enemigo se mueve hacia arriba.

```
moveEnemies_Y_up:
inc     a
sub     ENEMY_TOP_T
jr      z, moveEnemies_Y_upChg


inc     b
jr      moveEnemies_endMove


moveEnemies_Y_upChg:
set     $07, c
jr      moveEnemies_endMove
```

Incrementamos A para que apunte a la línea superior a la actual, ***INC A***, restamos el tope por arriba, ***SUB ENEMY_TOP_T***, y si el resultado es cero, hemos llegado al tope y saltamos para cambiar la dirección, ***JR Z, moveEnemies_Y_upChg***.

Si no hemos llegado al tope, incrementamos B para que apunte a la línea superior a la actual, ***INC B***, y saltamos al final del bucle, ***JR moveEnemies_endMove***.

Si hay que cambiar la dirección, activamos el bit siete de C para cambiar la dirección hacia abajo, ***SET $07, C***, y saltamos al final del bucle, ***JR moveEnemies_endMove***.

Si el enemigo no se mueve hacia arriba, se mueve hacia abajo.

```
moveEnemies_Y_down:
dec     a
sub     ENEMY_TOP_B
jr      z, moveEnemies_Y_downChg

dec     b
jr      moveEnemies_endMove

moveEnemies_Y_downChg:
res     $07, c
```

Decrementamos A para que apunte a la línea inferior a la actual, ***DEC A***, restamos el tope por abajo, ***SUB ENEMY_TOP_B***, y si el resultado es cero, hemos llegado al tope y saltamos para cambiar la dirección, ***JR Z, moveEnemies_Y_downChg***.

Si no hemos llegado al tope, decrementamos B para que apunte a la línea inferior a la actual, ***DEC B***, *y saltamos al final del bucle*, ***JR moveEnemies_endMove***.

Si hay que cambiar la dirección, desactivamos el bit siete de C para cambiar la dirección hacia arriba, ***RES $07, C***.

```
moveEnemies_endMove:
ld      (hl), c
dec     hl
ld      (hl), b

moveEnemies_endLoop:
inc     hl
inc     hl
dec     d
jr      nz, moveEnemies_loop
```

Actualizamos en memoria el segundo byte de la configuración del enemigo, ***LD (HL), C***, apuntamos HL al primer byte, ***DEC HL***, y actualizamos en memoria, ***LD (HL), B***.

Apuntamos HL al primer byte de la configuración del siguiente enemigo, ***INC HL***, ***INC HL***, decrementamos D, ***DEC D***, y seguimos en el bucle hasta que D sea cero y hayamos recorrido los veinte enemigos, ***JR Z, moveEnemies_loop***.

```
moveEnemies_end:
call    PrintEnemies


ret
```

Pintamos los enemigos en las nuevas posiciones, ***CALL PrintEnemies***, y salimos, ***RET***.

Con esto, hemos implementado la rutina que mueve los enemigos, cuyo aspecto final es el siguiente:

```
; ------------------------------------------------------------------------
; Mueve los enemigos.
;
; Altera el valor de lo registros AF, BC, D y HL.
; ------------------------------------------------------------------------
MoveEnemies:
ld      hl, flags           ; Cargamos la dirección de memoria de flags en HL
bit     $02, (hl)           ; Comprueba si el bit 2 está activo
ret     z                   ; Si no es así, sale
res     $02, (hl)           ; Desactiva el bit 2 de flags

ld      d, $14              ; Carga en D el número total de enemigos (20)
ld      hl, enemiesConfig   ; Cara en HL la dirección de la configuración
                            ; de los enemigos
moveEnemies_loop:
bit     $07, (hl)           ; Comprueba si el enemigo está activo
jr      z, moveEnemies_endLoop  ; Si no lo está, salta a final del bucle

push    hl                  ; Preserva el valor de HL

ld      a, (hl)             ; Carga en A el primer byte de la cofiguración
and     $1f                 ; Se queda con la coordenada Y
ld      b, a                ; Carga el valor en B

inc     hl                  ; Apunta HL al segundo byte de la configuración
ld      a, (hl)             ; Carga el valor en A
and     $1f                 ; Se queda con la coordeanda X
ld      c, a                ; Carga el valor en C

call    DeleteChar          ; Borra el enemigo
```

```
pop     hl                  ; Recupera el valor de HL

ld      b, (hl)             ; Carga en B el primer byte de la configuración
inc     hl                  ; Apunta HL al segundo byte de la configuración
ld      c, (hl)             ; Carga en C el segundo byte de la configuración

moveEnemies_X:
ld      a, c                ; Carga en A el segundo byte de la configuración
and     $1f                 ; Se queda con la coordeada X

bit     $06, c              ; Evalúa la dirección horizontal del enemigo
jr      nz, moveEnemies_X_right ; Si está a uno, hacia la derecha, salta

moveEnemies_X_left:
inc     a                   ; Apunta A a la columna anterior
sub     ENEMY_TOP_L         ; Resta el tope por la izquierda
jr      z, moveEnemies_X_leftChg    ; Si es cero, ha llegado al tope, salta

inc     c                   ; Apunta C a la columna anterior
jr      moveEnemies_Y       ; Salta al movimiento vertical

moveEnemies_X_leftChg:
set     $06, c              ; Pone la dirección horizontal hacia la derecha
jr      moveEnemies_Y       ; Salta al movimiento vertical

moveEnemies_X_right:
dec     a                   ; Apunta A a la columna posterior
sub     ENEMY_TOP_R         ; Resta el tope por la derecha
jr      z, moveEnemies_X_rightChg   ; Si es cero, ha llegado al tope, salta

dec     c                   ; Apunta C a la columna posterior
jr      moveEnemies_Y       ; Salta al movimiento vertical

moveEnemies_X_rightChg:
res     $06, c              ; Pone la dirección horizontal hacia la izquierda

moveEnemies_Y:
```

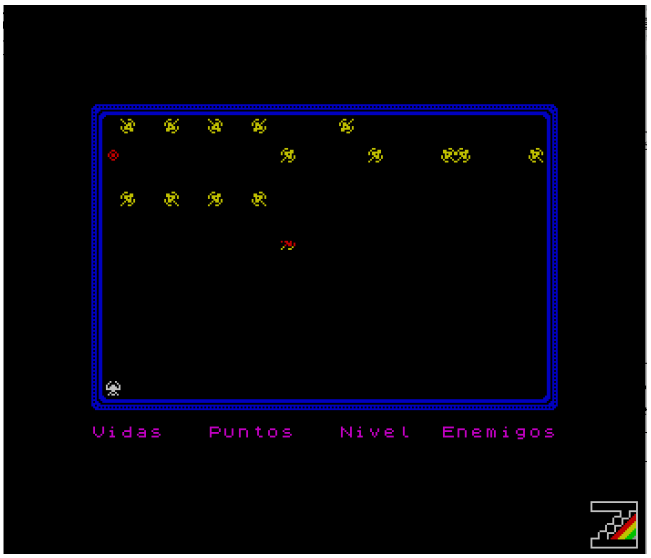
```
ld      a, b                ; Carga en A el primer byte de la configuración
and     $1f                 ; Se qued con la coordenda Y
bit     $07, c              ; Evalúa la dirección vertical del enemigo
jr      nz, moveEnemies_Y_down  ; Si está a uno, hacia abajo, salta

moveEnemies_Y_up:
inc     a                   ; Apunta A a la línea anterior
sub     ENEMY_TOP_T         ; Resta el tope por arriba
jr      z, moveEnemies_Y_upChg  ; Si es cero, ha llegado al tope, salta

inc     b                   ; Apunta B a la línea posterior
jr      moveEnemies_endMove ; Salta al final del bucle

moveEnemies_Y_upChg:
set     $07, c              ; Pone la dirección vertical hacia abajo
jr      moveEnemies_endMove ; Salta al final del bucle

moveEnemies_Y_down:
dec     a                   ; Apunta A a la línea posterior
sub     ENEMY_TOP_B         ; Resta el tope por abajo
jr      z, moveEnemies_Y_downChg    ; Si es cero, ha llegado al tope, salta

dec     b                   ; Apunta B a la línea posterior
jr      moveEnemies_endMove ; Salta al final del bucle

moveEnemies_Y_downChg:
res     $07, c              ; Pone la dirección vertical hacia arriba

moveEnemies_endMove:
ld      (hl), c             ; Actualiza el segundo byte de la configuración
dec     hl                  ; Apunta HL al primer byte de la configuración
ld      (hl), b             ; Actualiza el primer byte de la configuración

moveEnemies_endLoop:
inc     hl
inc     hl                  ; Aputa HL al primer byte de la configuración
                            ; del siguiente enemigo
dec     d                   ; Decrementa D
```

```
jr      nz, moveEnemies_loop    ; Hasta que D sea cero (20 enemigos)


moveEnemies_end:
call    PrintEnemies        ; Pinta los enemigos


ret
```

Ha llegado el momento de ver como se mueven los enemigos. Abrimos el archivo Main.asm y en la etiqueta ***Main_loop***, justo debajo de ***CALL MoveShip***, añadimos la línea siguiente:

```
call    MoveEnemies
```

Compilamos, cargamos en el emulador y vemos los resultados.



¿Qué tal? ¿Se mueven los enemigos? Sí, se mueven, pero van demasiado deprisa y el disparo se ha vuelto más lento. Vamos a ralentizar el movimiento de los enemigos, y lo vamos hacer desde la rutina de la interrupciones, así que vamos al archivo Int.asm.

Vamos a hacer algo parecido a lo que hicimos en PorompomPong, vamos a añadir un contador al final del archivo para controlar cuando activamos el movimiento de los enemigos.

```
countEnemy: db $00
```

Ahora entre las líneas ***SET $00, (HL)*** y ***SET $02, (HL)*** vamos a implementar el uso de este contador.

```
ld      a, (countEnemy)
inc     a
ld      (countEnemy), a
sub     $03
jr      nz, Isr_end
ld      (countEnemy), a
set     $02, (hl)
```

Cargamos en A el valor del contador, ***LD A, (countEnemy)***, incrementamos A, ***INC A***, y actualizamos el contador en memoria, ***LD (countEnemy), A***. Restamos a A el valor que tiene que alcanzar el contador para activar el movimiento, ***SUB $03***, y si no ha llegado salta al final de la rutina, ***JR NZ, Isr_end***.

Si ya ha alcanzado el valor, pone a cero el contador, ***LD (countEnemy), A***, y activa el bit para mover los enemigos, ***SET $02, (HL)***.

Compilamos, cargamos en el emulador y vemos que hemos recuperado la velocidad del disparo y que los enemigos se siguen moviendo rápido.

Conclusión

Ya tenemos en movimiento todos los elementos de nuestro juego.

En el siguiente capítulo implementaremos la detección de colisiones, para poder matar a los enemigos y que ellos nos maten a nosotros, y poder ir cambiando de nivel, hasta un total de treinta.

0x07 Colisiones y cambio de nivel

En este capítulo vamos a incluir las colisiones del disparo con los enemigos, de los enemigos con la nave, y los cambios de nivel. Lo primero es crear la carpeta Paso07, copiamos desde la carpeta Paso06 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, Print.asm, Var.asm y make, o make.bat si estáis trabajando en Windows.

Colisiones de los enemigos con el disparo

Lo primero que vamos a implementar son las colisiones entre los enemigos y el disparo. Como recordaremos, en el primer byte de la configuración de cada enemigo, el bit siete nos dice si está activo o no, pudiendo con esto decidir si se pinta o no.

La rutina que vamos a implementar va a comprobar si algún enemigo está en las mismas coordenadas que el disparo, y de ser así lo deshabilita.

Implementamos la rutina al inicio de archivo Game.asm.

```
CheckCrashFire:
ld      a, (flags)
and     $02
ret     z
```

Cargamos en A el valor de los flags, ***LD A, (flags)***, nos quedamos con el bit uno para comprobar si el disparo está activo, ***AND $02***, y salimos si no lo está, ***RET Z***.

```
ld      de, (firePos)
ld      hl, enemiesConfig
ld      b, enemiesConfigEnd - enemiesConfigIni
sra     b
```

Cargamos en DE las coordenadas del disparo, ***LD DE, (firePos)***, apuntamos HL a la configuración de los enemigos, ***LD HL, enemiesConfig***, cargamos en B el número de bytes totales de la configuración de los enemigos, ***LD B, enemiesConfigEnd – enemiesConfigIni***, y lo dividimos entre dos para calcular el número de enemigos, ***SRA B***, ya que la configuración de cada enemigo ocupa dos bytes.

SRA desplaza todos los bits hacia la derecha, el valor del bit cero lo pone en el acarreo y mantiene el valor del bit siete, para conservar el signo. ***SRA*** hace una división entera entre dos y dado que el número de enemigos que tenemos es par, nos vale.

```
checkCrashFire_loop:
ld      a, (hl)
inc     hl
bit     $07, a
jr      z, checkCrashFire_endLoop
```

Cargamos en A el primer byte de la configuración del enemigo, ***LD A, (HL)***, apuntamos HL al segundo byte de la configuración, ***INC HL***, evaluamos si el enemigo está activo, ***BIT $07, A***, y saltamos si no lo está, ***JR Z, checkCrashFire_endLoop***.

```
and     $1f
cp      d
jr      nz, checkCrashFire_endLoop
```

Si el enemigo está activo, nos quedamos con la coordenada Y, ***AND $1F***, comparamos con la coordenada Y del disparo, ***CP D***, y saltamos si no son la misma, ***JR NZ, checkCrashFire_endLoop***.

```
ld      a, (hl)
and     $1f
cp      e
jr      nz, checkCrashFire_endLoop
```

Cargamos en A el segundo byte de la configuración del enemigo, ***LD A, (HL)***, nos quedamos con la coordenada X, ***AND $1F***, comparamos con la coordenada X del disparo, ***CP E***, y saltamos si no son la misma.

```
dec     hl
res     $07, (hl)
ld      b, d
ld      c, e
call    DeleteChar


ret
```

Si el disparo y el enemigo colisionan, apuntamos HL al primer byte de la configuración del enemigo, ***DEC HL***, desactivamos el enemigo, ***RES $07, (HL)***, cargamos la coordenada Y del disparo en B, ***LD B, D***, cargamos la coordenada X del disparo en C, ***LD C, E***, borramos el disparo y/o enemigo, ***CALL DeleteChar***, y salimos de la rutina, ***RET***.

```
checkCrashFire_endLoop:
inc     hl
djnz    checkCrashFire_loop


ret
```

Si el disparo y el enemigo no colisionan, apuntamos HL al byte primer byte de la configuración del siguiente enemigo, ***INC HL***, y repetimos el bucle mientras B sea mayor que cero, ***DJNZ checkCrashFire_loop***. Una vez finalizado el bucle, salimos de la rutina, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones del disparo con los enemigos.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashFire:
ld      a, (flags)              ; Carga los flags en A
and     $02                     ; Evalúa si el disparo está activo
ret     z                       ; Si no está activo, sale

ld      de, (firePos)           ; Carga en DE la posición del disparo
ld      hl, enemiesConfig       ; Apunta HL a la definición del primer enemigo
ld      b, enemiesConfigEnd - enemiesConfigIni  ; Carga en B el número de bytes
                                                ; de la configuración de los enemigos
sra     b                       ; Lo divide entre dos, B = número de enemigos

checkCrashFire_loop:
ld      a, (hl)                 ; Carga en A la coordenada Y del enemigo
inc     hl                      ; Apunta HL a la coordenada X del enemigo
bit     $07, a                  ; Evalúa si el enemigo está activo
jr      z, checkCrashFire_endLoop   ; Si no está activo, salta
and     $1f                     ; Se queda con la coordenada Y de enemigo
cp      d                       ; Lo compara con la coordenada Y del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales salta
ld      a, (hl)                 ; Carga en A la coordenada X del enemigo
and     $1f                     ; Se queda con la coordenada X
cp      e                       ; Lo compara con la coordenada X del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales, salta

dec     hl                      ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)               ; Desactiva el enemigo
ld      b, d                    ; Carga la coordenada Y del disparo en B
ld      c, e                    ; Carga la coordenada X del disparo en C
call    DeleteChar              ; Borra el disparo y/o el enemigo

ret                             ; Sale de la rutina

checkCrashFire_endLoop:
```

```
inc     hl                      ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashFire_loop     ; Bucle mientras B > 0


ret
```

Ha llegado el momento de probar si las colisiones funcionan, abrimos el archivo Main.asm, localizamos la etiqueta ***Main_loop***, y justo debajo de ***CALL MoveFire***, preservamos el valor de DE (es donde tenemos las pulsaciones de los controles), ***PUSH DE***, añadimos la llamada a la rutina que acabamos de implementar, ***CALL CheckCrashFire***, y recuperamos el valor de DE, ***POP DE***, quedando de la siguiente manera:

```
Main_loop:
call    CheckCtrl
call    MoveFire
push    de
call    CheckCrashFire
pop     de
call    MoveShip
call    MoveEnemies
jr      Main_loop
```

Compilamos, cargamos en el emulador y probamos.

Tenemos dos problemas, uno de ellos heredado:

- Si no movemos la nave, se borra y no se vuelve a pintar.
- Una vez que ya no hay naves, no podemos hacer otra cosa que volver a cargar el juego.

El primer problema no lo vamos a abordar, si la nave se borra es porque ha colisionado con un enemigo, más adelante pintaremos una explosión.

Cambio de nivel

Para el cambio de nivel lo primero que tenemos que controlar es el número de enemigos que hay activos, al llegar a cero hay que cambiar de nivel. Lo segundo, es el número de niveles que tenemos, un total de treinta; por ahora al pasar al nivel treinta y uno, volveremos al uno, más adelante llegaremos al final del juego.

Abrimos el archivo Var.asm y vamos a añadir una variable para el número de enemigos activos y otra para el nivel actual, al inicio del archivo, después de los títulos de información de la partida.

```
; ------------------------------------------------------------------------
; Información de la partida
; ------------------------------------------------------------------------
enemiesCounter:
db  $14
levelCounter:
db  $01
```

Antes de implementar la rutina que hace el cambio de nivel, vamos a hacer unos cambios para usar levelCounter. Abrimos el archivo Graph.asm y localizamos la rutina ***LoadUdgsEnemies***. Esta rutina recibe en A el nivel, pero ya no es necesario pues ese valor lo va a tomar de ***levelCounter***. Añadimos la siguiente línea al inicio de la rutina:

```
ld      a, (levelCounter)
```

Cargamos en A el nivel actual, ***LD A, (levelCounter)***.

En los comentarios de la rutina, borramos la línea referente a la entrada en A del nivel, quedando tal y como sigue:

```
; ------------------------------------------------------------------------
; Carga los gráficos definidos por el usuario relativos a los enemigos
;
; Altera el valor de los registros AF, BC, DE y HL
; ------------------------------------------------------------------------
LoadUdgsEnemies:
ld      a, (levelCounter)       ; Carga en A el nivel
dec     a                       ; Decrementa A para que no sume un nivel de más
ld      h, $00
ld      l, a                    ; Carga el resultado en HL
add     hl, hl                  ; Multiplica por 2
add     hl, hl                  ; por 4
add     hl, hl                  ; por 8
add     hl, hl                  ; por 16
add     hl, hl                  ; por 32
```

```
ld      de, udgsEnemiesLevel1   ; Carga la dirección del enemigo 1 en DE
add     hl, de                  ; Lo suma a HL
ld      de, udgsExtension       ; Carga en DE la dirección de la extensión
ld      bc, $20                 ; Carga en BC el número de bytes a copiar, 32
ldir                            ; Copia los bytes del enemigo en los de extensión


ret
```

En el archivo Game.asm, buscamos la etiqueta ***checkCrahsFire_endLoop***, justo por encima de ella hay un ***RET*** y justo por encima de este ***RET*** añadimos las siguientes líneas:

```
ld      hl, enemiesCounter      ; Apunta HL al contador de enemigos
dec     (hl)                    ; Resta un enemigo
```

En el archivo Main.asm, tres líneas por encima de ***Main_loop***, justo antes de ***CALL LoadUdgsEnemies***, borramos la línea ***LD A, $01***, pues el nivel se toma ya de ***levelCounter***.

Compilamos, cargamos en el emulador y comprobamos que todo sigue funcionando.

Ahora, en el archivo Game.asm, vamos a implementar el cambio de nivel, que consiste en cargar los gráficos de los enemigos del siguiente nivel, reiniciar la configuración de los enemigos, y actualizar los contadores que acabamos de añadir.

```
ChangeLevel:
ld      a, (levelCounter)
inc     a
cp      $1f
jr      c, changeLevel_end
ld      a, $01
```

Cargamos en A el nivel actual, ***LD A, (levelCounter)***, incrementamos A para pasar al siguiente nivel, ***INC A***, y comprobamos si el siguiente nivel es el treinta y uno, ***CP $1F***. Si no hemos llegado al nivel treinta y uno saltamos a la parte final de la rutina, ***JR C, changeLevel_end***. Si hemos llegado al nivel treinta y uno, recordad que solo tenemos treinta niveles, no saltamos y ponemos A a $01.

```
changeLevel_end:
ld      (levelCounter), a
call    LoadUdgsEnemies


ld      a, $14
ld      (enemiesCounter), a


ld      hl, enemiesConfigIni
ld      de, enemiesConfig
```

```
ld      bc, enemiesConfigEnd - enemiesConfigIni
ldir


ret
```

Cargamos el siguiente nivel en memoria, ***LD (levelCounter), A***, cargamos los gráficos del enemigo del siguiente nivel, ***CALL LoadUdgsEnemies***, cargamos el número de enemigos totales en A, ***LD A, $14***, y actualizamos el valor en memoria, ***LD (enemiesCounter), A***.

Por último, reiniciamos la configuración de los enemigos. Apuntamos HL a la configuración inicial de los enemigos, ***LD HL, enemiesConfigIni***, apuntamos DE a la configuración de los enemigos, ***LD DE, enemiesConfig***, cargamos en BC el número de bytes de los que se compone la configuración de los enemigos, ***LD BC, enemiesConfigEnd – enemiesConfigIni***, cargamos la configuración inicial de los enemigos en la configuración de los enemigos, ***LDIR***, y salimos, ***RET***.

El funcionamiento de LDIR ya se explicó en PorompomPong.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Cambia de nivel.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
ChangeLevel:
ld      a, (levelCounter)       ; Carga el nivel actual en A
inc     a                       ; Carga en A el siguiente nivel
cp      $1f                     ; Compara si el nivel es el 31
jr      c, changeLevel_end      ; Si no es el 31, salta
ld      a, $01                  ; Si es el 31, lo pone a 1


changeLevel_end:
ld      (levelCounter), a       ; Actualiza el nivel en memoria
call    LoadUdgsEnemies         ; Carga los gráficos de los enemigos


ld      a, $14                  ; Carga en A el número total de enemigos
ld      (enemiesCounter), a     ; Lo carga en memoria


ld      hl, enemiesConfigIni    ; Apunta HL a la configuración inicial
ld      de, enemiesConfig       ; Apunta DE a la configuración
ld      bc, enemiesConfigEnd - enemiesConfigIni ; Carga en BC la longitud
                                                ; de la configuración
ldir                            ; Carga la configuración inicial en la configuración
```

```
ret
```

Para finalizar, tenemos que usar lo implementado; lo vamos a hacer en Main.asm. Localizamos la rutina ***MainLoop***, localizamos la quinta línea, ***POP DE***, y justo debajo de ella añadimos lo siguiente:

```
ld      a, (enemiesCounter)
or      a
jr      z, Main_restart
```

Cargamos en A el número de enemigos activos, ***LD A, (enemiesCounter)***, comprobamos si hemos llegado a cero***, CP $00***, y saltamos si es así, ***JR Z, Main_restart***.

Ahora vamos al final del archivo, y justo encima del primer ***include*** añadimos lo siguiente:

```
Main_restart:
call    ChangeLevel
jr      Main_loop
```

Cambiamos al siguiente nivel, ***CALL ChangeLevel***, y volvemos al inicio del bucle, ***JR Main_loop***.

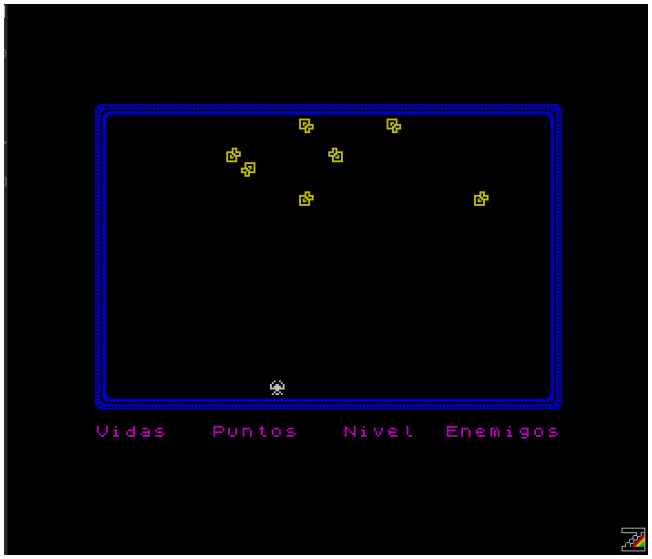
Dado que Main.asm va creciendo, veamos cual es el aspecto que debe tener ahora:

```
org     $5dad

; ------------------------------------------------------------------------------
; Indicadores
;
; Bit 0 -> se debe mover la nave            0 = No, 1 = Sí
; Bit 1 -> el disparo está activo           0 = No, 1 = Sí
; Bit 2 -> se deben mover los enemigos      0 = No, 1 = Sí
; ------------------------------------------------------------------------------
flags:
db $00

Main:
ld      a, $02
call    OPENCHAN

ld      hl, udgsCommon
ld      (UDG), hl

ld      hl, ATTR_P
```

```
ld      (hl), $07
call    CLS

xor     a
out     ($fe), a
ld      a, (BORDCR)
and     $c7
or      $07
ld      (BORDCR), a

call    PrintFrame
call    PrintInfoGame
call    PrintShip

di
ld      a, $28
ld      i, a
im      2
ei

call    LoadUdgsEnemies
call    PrintEnemies

Main_loop:
call    CheckCtrl
call    MoveFire

push    de
call    CheckCrashFire
pop     de

ld      a, (enemiesCounter)
or      a
jr      z, Main_restart

call    MoveShip
call    MoveEnemies
jr      Main_loop
```

```
Main_restart:
call    ChangeLevel
jr      Main_loop

include "Const.asm"
include "Var.asm"
include "Graph.asm"
include "Print.asm"
include "Ctrl.asm"
include "Game.asm"

end     Main
```

Compilamos, cargamos en el emulador y, si todo ha ido bien, vemos como cambian los enemigos cuando los hemos destruido todos.



Colisiones de los enemigos con la nave

En esta primera aproximación, lo único que vamos a hacer es pintar una explosión cuando algún enemigo choque contra la nave, más adelante nos restará una vida.

Primero vamos a implementar la rutina que pinta la explosión, abrimos el archivo Print.asm.

```
PrintExplosion:
ld      a, $02
call    Ink

ld      bc, (shipPos)
ld      d, $04
```

```
ld      e, $92
```

Cargamos dos en A (dos = color rojo), ***LD A, $02***, y cambiamos el color de la tinta, ***CALL INK***. Cargamos en BC la posición de la nave, ***LD BC, (shipPos)***, cargamos en D el número de UDG totales que tiene la explosión, ***LD D, $04***, y cargamos en E el primer UDG de la explosión, ***LD E, $92***.

```
printExplosion_loop:
call    At
ld      a, e
rst     $10
halt
halt
halt
halt
inc     e
dec     d
jr      nz, printExplosion_loop


jp      PrintShip
```

Posicionamos el cursor en las coordenadas de la nave, ***CALL AT***, cargamos en A el UDG, ***LD A, E***, y lo pintamos, ***RST $10***. Esperamos cuatro interrupciones, ***HALT, HALT, HALT, HALT***, apuntamos E al siguiente UDG, ***INC E***, decrementamos D, ***DEC D***, y nos quedamos en el bucle hasta que D valga cero, ***JR NZ, printExplosion_loop***.

Por último, volvemos a pintar la nave y salimos, ***JP PrintShip***. Aprovechamos el ***RET*** de ***PrintShip*** para salir. Podríamos haber llamado a ***PrintShip*** *y luego salido:*

```
call    PrintShip
ret
```

Pero con ***JP*** nos ahorramos un byte y diecisiete ciclos de reloj.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta la explosión de la nave
;
; Altera los valores de los registros AF, BC y DE.
; ------------------------------------------------------------------------
PrintExplosion:
ld      a, $02
call    Ink             ; Pone la tinta en rojo
```

```
ld      bc, (shipPos)   ; Carga en BC la posición de la nave
ld      d, $04          ; Carga en D el número de UDG totales de la explosión
ld      e, $92          ; Carga en E el primer UDG de la explosión
printExplosion_loop:
call    At              ; Posiciona el cursor
ld      a, e            ; Carga en A el UDG
rst     $10             ; Lo pinta
halt
halt
halt
halt                    ; Espera 4 interrupciones
inc     e               ; Apunta E al siguiente UDG
dec     d               ; Decrementa D
jr      nz, printExplosion_loop ; Bucle hasta que D = 0

jp      PrintShip       ; Pinta la nave y sale por allí
```

Ahora, en Game.asm, vamos a implementar las colisiones entre los enemigos y la nave que, como veréis, es bastante parecida a la rutina que implementa las colisiones de los enemigos con el disparo.

```
CheckCrashShip:
ld      de, (shipPos)
ld      hl, enemiesConfig
ld      b, enemiesConfigEnd - enemiesConfigIni
sra     b
```

Cargamos en HL la posición de la nave, ***LD DE, (shipPos)***, apuntamos HL a la configuración de los enemigos ***LD HL, enemiesConfig***, cargamos en B el número de bytes de la configuración, ***LD B, enemiesConfigEnd – enemiesConfigIni***, y lo dividimos entre dos para obtener el número de enemigos, ***SRA B***.

```
checkCrashShip_loop:
ld      a, (hl)
inc     hl
bit     $07, a
jr      z, checkCrashShip_endLoop
```

Cargamos en A el primer byte de la configuración del enemigo, ***LD A, (HL)***, apuntamos HL al segundo byte de la configuración del enemigo, ***INC HL***, comprobamos si el enemigo está activo, ***BIT $07, A***, y saltamos si no es así, ***JR Z, checkCrashShip_endLoop***.

```
and     $1f
cp      d
jr      nz, checkCrashShip_endLoop
```

Nos quedamos con la coordenada Y del enemigo, ***AND $1F***, comparamos con la coordenada Y de la nave, ***CP D***, y saltamos si no son la misma, ***JR NZ, checkCrashShip_endLoop***.

```
ld      a, (hl)
and     $1f
cp      e
jr      nz, checkCrashShip_endLoop
```

Cargamos el segundo byte de la configuración del enemigo en A, ***LD A, (HL)***, nos quedamos con la coordenada X, ***AND $1F***, comparamos con la coordenada X de la nave, ***CP E***, y saltamos si no son la misma, ***JR NZ, checkCrashShip_endLoop***.

```
dec     hl
res     $07, (hl)

ld      hl, enemiesCounter
dec     (hl)

jp      PrintExplosion
```

Si pasamos por aquí, ha habido colisión. Apuntamos HL al primer byte de la configuración del enemigo, ***DEC HL***, desactivamos el enemigo, ***RES $07,(HL)***, apuntamos HL al contador de enemigos, ***LD HL, enemiesCounter***, y le restamos uno, ***DEC (HL)***. Finalmente, saltamos a pintar la explosión y salimos, ***JP PrintExplosion***, usando la misma técnica que hemos visto en ***PrintExplosion***.

```
checkCrashShip_endLoop:
inc     hl
djnz    checkCrashShip_loop

ret
```

Si no ha habido colisión, apuntamos HL al primer byte de la configuración del siguiente enemigo, ***INC HL***, y permanecemos en el bucle hasta que B valga cero y hayamos recorrido todos los enemigos, ***DJNZ checkCrashShip_loop***. Para finalizar, salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones de los enemigos con la nave.
;
```

```
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashShip:
ld      de, (shipPos)       ; Carga en DE la posición de nave
ld      hl, enemiesConfig   ; Apunta HL a la configuración de los enemigos
ld      b, enemiesConfigEnd - enemiesConfigIni  ; B = bytes totales configuración
sra     b                   ; B = B / 2 = número de enemigos

checkCrashShip_loop:
ld      a, (hl)             ; Carga en A la coordenada Y del enemigo
inc     hl                  ; Apunta HL a la coordenada X del enemigo
bit     $07, a              ; Evalúa si el enemigo está activo
jr      z, checkCrashShip_endLoop   ; Si no lo está, salta

and     $1f                 ; Se queda con la coordenada Y del enemigo
cp      d                   ; Compara con la coordenada Y de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

ld      a, (hl)             ; Carga en A la coordenada X del enemigo
and     $1f                 ; Se queda con la coordenada X de enemigo
cp      e                   ; Compara con la coordenada X de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

dec     hl                  ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)           ; Desactiva el enemigo

ld      hl, enemiesCounter  ; Apunta HL al contador de enemigos
dec     (hl)                ; Resta un enemigo

jp      PrintExplosion      ; Pinta la explosión y sale

checkCrashShip_endLoop:
inc     hl                  ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashShip_loop ; En bucle hasta que B = 0

ret
```

Ha llegado el momento de probar las colisiones entre la nave y los enemigos. Abrimos Main.asm, localizamos la rutina ***Main_loop*** y vemos que la última línea es ***JR Main_loop***. Justo encima de esta línea vamos a añadir la llamada a la comprobación de las colisiones entre nave y enemigos:

```
call    CheckCrashShip
```

Compilamos, cargamos en el emulador y vemos los resultados.

Conclusión

Hemos implementado las colisiones entre el disparo y los enemigos, y entre los enemigos y la nave. También hemos implementado el cambio de nivel cuando hemos destruido todos los enemigos.

En el próximo capítulo vamos a implementar una transición entre niveles y el marcador.

0x08 Transición entre niveles y marcador

En este capítulo vamos a implementar una transición entre niveles y el marcador.

Como siempre, creamos la carpeta Paso08 y copiamos desde la carpeta Paso07 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, Print.asm, Var.asm y make, o make.bat si estáis trabajando en Windows.

Transición de cambio de nivel

Lo primero que vamos a implementar es una rutina que cambie los atributos de color de la pantalla, asignando los que vengan en el registro A. Abrimos el archivo Graph.asm.

```
Cla:
ld      hl, $5800
ld      (hl), a
ld      de, $5801
ld      bc, $02ff
ldir


ret
```

Apuntamos HL a la primera dirección del área de atributos, ***LD HL, $5800***, cargamos los nuevos atributos en esa dirección, ***LD (HL), A***, apuntamos DE a la siguiente dirección, ***LD DE, $5801***, cargamos en BC el número de posiciones totales del area de atributos menos una (la primera ya esta cambiada), ***LD BC, $02FF***, cambiamos todo el área de atributos, ***LDIR***, y salimos, ***RET***.

El aspecto de la rutina, una vez comentada, es el siguiente:

```
; ------------------------------------------------------------------------------
; Cambia los atributos de color de la pantalla.
;
; Entrada:  A = Atributos de color (FBPPPIII).
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
Cla:
ld      hl, $5800   ; Apunta HL a la dirección de inicio de los atributos
ld      (hl), a     ; Carga los atributos
ld      de, $5801   ; Apunta DE a la segunda dirección de los atributos
ld      bc, $02ff   ; Carga en BC el número de posiciones a cambiar
ldir                ; Cambia los atributos de la pantalla


ret
```

Ahora vamos a implementar, también en Graph.asm, la rutina que vamos a usar como transición de un nivel a otro. Esta rutina es una variación de la rutina FadeScreen que podéis encontrar en el Curso de Ensamblador Z80 de Compiler Software.

Vamos a recorrer todo el área de vídeo para realizar un máximo de ocho desplazamientos en cada byte para limpiarlo.

```
FadeScreen:
ld      b, $08


fadeScreen_loop1:
ld      hl, $4000
ld      de, $1800
```

Cargamos en B el número de iteraciones del bucle exterior, ***LD B, $08***, apuntamos HL al inicio del área de vídeo, ***LD HL, $4000***, y cargamos en DE la longitud total del área de vídeo (la parte de los píxeles), ***LD DE, $1800***.

```
fadeScreen_loop2:
ld      a, (hl)
or      a
jr      z, fadeScreen_cont


bit     $00, l
jr      z, fadeScreen_right


rla
jr      fadeScreen_cont


fadeScreen_right:
rra
```

Cargamos en A el byte al que apunta HL, ***LD A, (HL)***, comprobamos si está limpio (está a cero), ***OR A***, y saltamos si lo está, ***JR Z, fadeScreen_cont***.

Si no saltamos, comprobamos si la dirección a la que apunta HL es par o impar, ***BIT $00, L***, y si lo es (el bit 0 vale cero) saltamos, ***JR Z, fadeScreen_right***. Si la dirección de memoria es impar, no saltamos, rotamos A hacia la izquierda, ***RLA***, y saltamos, ***JR fadeScreen_cont***. Si la dirección de memoria es par, rotamos A hacia la derecha, ***RRA***.

Antes de continuar vamos a detenernos en tres líneas, la primera de ellas es ***OR A***. Llegados a este punto, queremos saber si el byte del área de vídeo al que punta HL está limpio (tiene todos los bits a cero) y lo podríamos haber hecho con ***CP $00***, consumiendo dos bytes y siete ciclos de reloj. ***OR A*** solo da como resultado cero, si A vale cero, consumiendo un byte y cuatro ciclos de reloj; la afectación de los flags es muy parecida, y en el caso del flag de acarreo el resultado es el mismo. En lugar de ***OR A***, podríamos poner ***AND A***, el resultado es el mismo.

Las siguientes líneas a tener en cuenta son ***RLA*** y ***RRA***. En ambos casos se rota el registro A, en el caso de ***RLA*** hacia de izquierda, y hacia la derecha en el caso de ***RRA***.

- ***RLA***: rota el byte hacia la izquierda, el valor del bit 7 lo pone en el acarreo, y el valor que tiene el acarreo lo pasa al bit 0.
- ***RRA***: rota el byte hacia la derecha, el valor del bit 0 lo pone en el acarreo, y el valor que tiene el acarreo lo pasa al bit 7.

C = 1 Byte = 10000001

RLA	RRA
C = 1 Byte = 00000011	C = 1 Byte = 11000000
C = 0 Byte = 00000111	C = 0 Byte = 11100000
C = 0 Byte = 00001110	C = 0 Byte = 01110000

Según podemos ver en esta tabla, si en algún momento de la rutina ***FadeScreen***, antes de hacer la rotación, el acarreo está a uno, se nos puede quedar algún píxel sin limpiar; pero esto no nos va a pasar pues otras de las cosas que hace ***OR A*** es poner el acarreo a cero, de igual manera pasa si usamos ***CP $00***.

Llegamos a la parte final de la rutina.

```
fadeScreen_cont:
ld      (hl), a
inc     hl


dec     de
ld      a, d
or      e
jr      nz, fadeScreen_loop2


ld      a, b
dec     a
push    bc
call    Cla
pop     bc


djnz    fadeScreen_loop1


ret
```

Actualizamos la posición de vídeo a la que apunta HL con el valor rotado, ***LD (HL), A***, y apuntamos HL a la siguiente posición del área de vídeo, ***INC HL***.

Decrementamos DE, ***DEC DE***, cargamos el valor de D en A, ***LD A, D***, lo mezclamos con E, ***OR E***, y seguimos en bucle hasta que DE sea cero, ***JR NZ, fadeScreen_loop2***.

Cargamos el valor de B en A, ***LD A, B***, decrementamos A para que el valor quede comprendido entre siete y cero, ***DEC A***, preservamos el valor de BC, ***PUSH BC***, cambiamos los colores de la pantalla, ***CALL Cla***, recuperamos el valor de BC, ***POP BC***, y seguimos en bucle hasta que B valga cero, ***DJNZ fadeScreen_loop1***. Finalmente, salimos.

Vamos a parar nuevamente para explicar una parte del código más en profundidad.

```
dec     de
ld      a, d
or      e
jr      nz, fadeScreen_loop2
```

Hasta ahora hemos hecho bucles usando registros de 8 bits, como es en este caso el bucle externo; cargamos ocho en B, ***LD B, $08***, y más adelante, con ***DJNZ***, decrementamos B y si el resultado no es cero saltamos y seguimos en el bucle, gracias a que ***INC*** o ***DEC*** cuando se hace sobre un registro de 8 bits afecta al flag Z.

En el caso de los registro de 16 bits, los incrementos y los decrementos no afectan al flag Z, de esta manera si solo decrementamos el registro y luego comprobamos si se ha activado el flag Z, nos encontramos ante un bucle infinito. Para hacer un bucle utilizando un registro de 16 bits, tras decrementar el registro, cargamos una de sus partes en A y luego hacemos ***OR*** con la otra parte, y en el caso de que ambos valores valgan cero, tal y como hemos visto anteriormente en este mismo capítulo, el resultado es cero, se activa el flag Z y saldremos del bucle.

El resultado final de la rutina es el siguiente.

```
; ------------------------------------------------------------------------------
; Efecto de desvanecimento de la pantalla.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
FadeScreen:
ld      b, $08      ; El bucle exterior se repite 8 veces, una por bit

fadeScreen_loop1:
ld      hl, $4000   ; Apunta HL al inicio del área de vídeo
ld      de, $1800   ; Carga en DE la longitud del área de vídeo

fadeScreen_loop2:
ld      a, (hl)     ; Carga en A el byte apuntado por HL
or      a           ; Comprueba si tiene algún píxel activo
jr      z, fadeScreen_cont  ; Si no hay ninguno activo, salta

bit     $00, l      ; Comprueba si la dirección apuntada por HL es par/impar
```

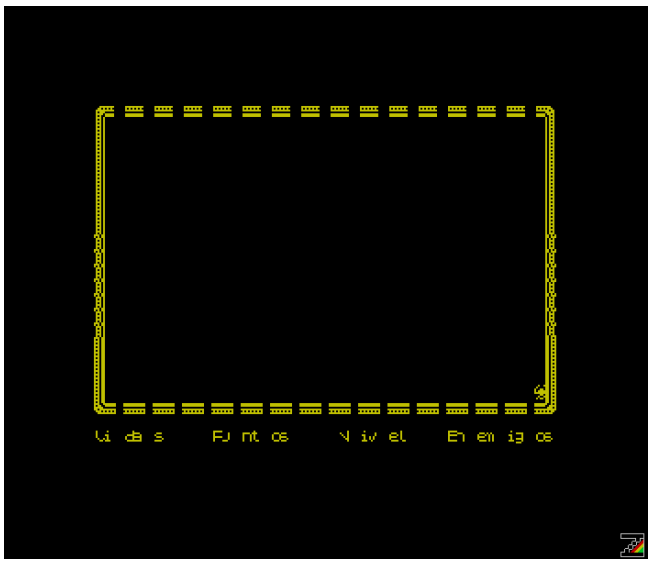
```
jr      z, fadeScreen_right     ; Si es par, salta

rla                 ; Rota A un bit a la izquierda
jr      fadeScreen_cont

fadeScreen_right:
rra                 ; Rota A un bite a la derecha

fadeScreen_cont:
ld      (hl), a     ; Actualiza la posición de vídeo apuntada por HL
inc     hl          ; Apunta HL a la siguiente posición

dec     de
ld      a, d
or      e
jr      nz, fadeScreen_loop2    ; Bucle hasta que BC = 0

ld      a, b        ; Carga B en A
dec     a           ; Decrementa A para que quede entre 0 y 7
push    bc          ; Preserva el valor de BC
call    Cla         ; Cambia los colores de la pantalla
pop     bc          ; Recupera el valor de BC

djnz    fadeScreen_loop1    ; Bucle hasta que B = 0

ret
```

Y llega el momento de probar lo implementado; abrimos Main.asm, localizamos la rutina ***Main_restart***, y justo debajo, antes de ***CALL ChangeLevel***, agregamos las siguientes líneas:

```
call    FadeScreen
call    PrintFrame
call    PrintInfoGame
call    PrintShip
```

Llamamos al efecto de fusión de la pantalla, ***CALL FadeScreen***, pintamos el marco de la pantalla, ***CALL PrintFrame***, pintamos la información de la partida***, CALL PrintInfoGame***, y pintamos la nave, ***CALL PrintShip***.

Compilamos, cargamos en el emulador, matamos a todas la naves enemigas y vemos el efecto de fundido de la pantalla.



Marcador

En el marcador vamos a mostrar el número de vidas que tenemos, los puntos conseguidos, el nivel por el que vamos y los enemigos que quedan, por lo que vamos a necesitar alguna declaración más y vamos a introducir un nuevo concepto: los números BCD.

Números BCD

Un byte es capaz de contener números de 0 a 255, si trabajamos con números BCD este rango se reduce de 0 a 99. Los números BCD dividen el byte en dos nibbles (4 bits) y en cada uno de ellos almacena valores del 0 al 9, por lo que el valor hexadecimal 0x10, que en decimal es 16, trabajando con BCD sería 10, es decir, veríamos el número en notación hexadecimal como si fiera decimal, siendo esto muy útil, por ejemplo a la hora de pintarlos o para operar con números de más de 16 bits.

Para poder operar de esta manera con los números, tenemos la instrucción ***DAA*** (Decimal Adjust Accumulator), que funciona de la siguiente forma:

- Comprueba los bits 0, 1, 2 y 3, si contienen un dígito no BCD, mayor que nueve, o el flag H está activo, suma o resta $06 (0000 0110b) al byte, dependiendo de la operación que se ha realizado.
- Comprueba los bits 4, 5, 6, y 7, si contienen un dígito no BCD, mayor que nueve, o el flag C está activo, suma o resta $60 (0110 0000b) al byte, dependiendo de la operación que se ha realizado.

Después de cada instrucción aritmética, incremento o decremento hay que ejecutar ***DAA***. Veamos un ejemplo.

```
ld    a, $09    ; A = $09
inc   a         ; A = $0a
daa             ; A = $10
dec   a         ; A = $0f
daa             ; A = $09
```

```
add   a, $03    ; A = $0c
daa             ; A = $12


sub   a, $03    ; A = $0f
daa             ; A = $09
```

Número de enemigos y nivel

Ahora, vamos a abrir el archivo Var.asm y a localizar la etiqueta ***enemiesCounter***, que como vemos define veinte en hexadecimal ($14), y lo vamos a cambiar por el valor en BCD, $20. Localizamos la etiqueta ***levelCounter***, y vemos que define $01; en este caso no lo vamos a cambiar, pero vamos añadir un byte con el mismo valor, usado el primer byte para cargar los enemigos de cada nivel y hacer el cambio de nivel, y el segundo byte para pintar el número de nivel en el que estamos.

```
; ------------------------------------------------------------------------------
; Información de la partida
; ------------------------------------------------------------------------------
enemiesCounter:
db  $20
levelCounter:
db  $01, $01
```

Estas dos etiquetas las usamos en partes de nuestro programa, pero no estamos teniendo en cuenta que ahora vamos a trabajar con números BCD, por lo que tenemos que localizar los lugares donde se usan y modificar su comportamiento.

La primera modificación la vamos a hacer en la rutina ***ChangeLevel***, que está en el archivo Game.asm, añadiendo siete líneas. Las cuatro primeras líneas las vamos a añadir al inicio de la rutina.

```
ld      a, (levelCounter + 1)
inc     a
daa
ld      b, a
```

Cargamos el nivel actual en formado BCD en A, ***LD A, (levelCounter + 1)***, incrementamos el nivel, ***INC A***, realizamos el ajuste decimal, ***DAA***, y cargamos el valor en B, ***LD B, A***.

Ahora, justo por encima de la etiqueta ***changeLevel_end*** añadimos la siguiente línea:

```
ld      b, a
```

Si pasamos por aquí, el siguiente nivel sería el treinta y uno y solo tenemos treinta, por lo que cargamos $01 en A, y ahora cargamos ese valor en el registro donde tenemos el nivel en formato BCD, ***LD B, A***.

Seguimos tomando como referencia la etiqueta ***changeLevel_end***, tras la cual actualizamos el nivel en memoria. Justo debajo de esta línea, ***LD (levelCounter), a***, vamos a añadir las líneas que actualizan en memoria el nivel en formato BCD.

```
ld      a, b
ld      (levelCounter + 1), a
```

Cargamos en A el nivel actual en BCD, ***LD A, B***, y lo actualizamos en memoria, ***LD (levelCounter + 1), A***.

Dos líneas más abajo, sustituimos ***LD A, $14*** por ***LD A, $20***, para tener el número total de enemigos en BCD.

El aspecto final de la rutina, una vez realizadas las modificaciones, es el siguiente.

```
; ------------------------------------------------------------------------
; Cambia de nivel.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
ChangeLevel:
ld      a, (levelCounter + 1)   ; Carga en A el nivel actual en BCD
inc     a                       ; Incrementa el nivel
daa                             ; Hace el ajuste decimal
ld      b, a                    ; Carga el valor en B
ld      a, (levelCounter)       ; Carga el nivel actual en A
inc     a                       ; Carga en A el siguiente nivel
cp      $1f                     ; Compara si el nivel es el 31
jr      c, changeLevel_end      ; Si no es el 31, salta
ld      a, $01                  ; Si es el 31, lo pone a 1
ld      b, a                    ; Cargamos el valor en B

changeLevel_end:
ld      (levelCounter), a       ; Actualiza el nivel en memoria
ld      a, b                    ; Carga en A el nivel en BCD
ld      (levelCounter + 1), a   ; Lo actualiza en memoria
call    LoadUdgsEnemies         ; Carga los gráficos de los enemigos

ld      a, $20                  ; Carga en A el número total de enemigos
ld      (enemiesCounter), a     ; Lo carga en memoria

ld      hl, enemiesConfigIni    ; Apunta HL a la configuración inicial
```

```
ld      de, enemiesConfig       ; Apunta DE a la configuración
ld      bc, enemiesConfigEnd - enemiesConfigIni ; Carga en BC la longitud
                                                ; de la configuración
ldir                            ; Carga la configuración inicial en la configuración


ret
```

Si ahora compiláramos, veríamos que al matar a todos los enemigos no se produciría el cambio de nivel, debido a que ahora el número de enemigos es $20 (32) y no hemos adaptado todas las rutinas para trabajar con BCD.

Seguimos en el archivo Game.asm, localizamos las etiqueta ***checkCrashFire_endLoop***, justo por encima de esta etiqueta hay un ***RET***, y justo por encima están las instrucciones que restan un enemigo al contador, ***LD HL, enemiesCounter*** y ***DEC (HL)***. Vamos a sustituir esas dos líneas por las siguientes:

```
ld      a, (enemiesCounter)
dec     a
daa
ld      (enemiesCounter), a
```

Cargamos en A el número de enemigos, ***LD A, (enemiesCounter)***, le restamos uno, ***DEC A***, realizamos el ajuste decimal, ***DAA***, y actualizamos el valor en memoria, ***LD (enemiesCounter), A***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Evalúa las colisiones del disparo con los enemigos.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
CheckCrashFire:
ld      a, (flags)              ; Carga los flags en A
and     $02                     ; Evalúa si el disparo está activo
ret     z                       ; Si no está activo, sale


ld      de, (firePos)           ; Carga en DE la posición del disparo
ld      hl, enemiesConfig       ; Apunta HL a la definición del primer enemigo
ld      b, enemiesConfigEnd - enemiesConfigIni  ; Carga en B el número de bytes
                                                ; de la configuración de los enemigos
sra     b                       ; Lo divide entre dos, B = número de enemigos


checkCrashFire_loop:
```

```
ld      a, (hl)                 ; Carga en A la coordenada Y del enemigo
inc     hl                      ; Apunta HL a la coordenada X del enemigo
bit     $07, a                  ; Evalúa si el enemigo está activo
jr      z, checkCrashFire_endLoop   ; Si no está activo, salta
and     $1f                     ; Se queda con la coordenada Y del enemigo
cp      d                       ; Lo compara con la coordenada Y del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales salta
ld      a, (hl)                 ; Carga en A la coordenada X del enemigo
and     $1f                     ; Se que con la coordenada X
cp      e                       ; Lo compara con la coordenada X del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales, salta

dec     hl                      ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)               ; Desactiva el enemigo
ld      b, d                    ; Carga la coordenada Y del disparo en B
ld      c, e                    ; Carga la coordenada X del disparo en C
call    DeleteChar              ; Borra el disparo y/o el enemigo
ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
ld      (enemiesCounter), a     ; Actualiza el valor en memoria

ret                             ; Sale de la rutina

checkCrashFire_endLoop:
inc     hl                      ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashFire_loop     ; Bucle mientras B > 0

ret
```

Hay otra rutina en dónde restamos un enemigo, la rutina que evalúa las colisiones de la nave con el enemigo. Localizamos la etiqueta ***checkCrashShip_endLoop***, justo encima encontramos ***JP PrintExplosion***, y justo encima encontramos dos líneas iguales a las que hemos sustituido, y que tenemos que sustituir igual que hemos hecho antes.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones de los enemigos con la nave.
;
; Altera el valor de lo registros AF, BC, DE y HL.
```

```
; ------------------------------------------------------------------------
CheckCrashShip:
ld      de, (shipPos)       ; Carga en DE la posición de nave
ld      hl, enemiesConfig   ; Apunta HL a la configuración de los enemigos
ld      b, enemiesConfigEnd - enemiesConfigIni  ; B = bytes totales configuración
sra     b                   ; B = B / 2 = número de enemigos

checkCrashShip_loop:
ld      a, (hl)             ; Carga en A la coordenada Y del enemigo
inc     hl                  ; Apunta HL a la coordenada X del enemigo
bit     $07, a              ; Evalúa si el enemigo está activo
jr      z, checkCrashShip_endLoop   ; Si no lo está, salta

and     $1f                 ; Se queda con la coordenada Y del enemigo
cp      d                   ; Compara con la coordenada Y de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

ld      a, (hl)             ; Carga en A la coordenada X del enemigo
and     $1f                 ; Se queda con la coordenada X de enemigo
cp      e                   ; Compara con la coordenada X de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

dec     hl                  ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)           ; Desactiva el enemigo

ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
ld      (enemiesCounter), a     ; Actualiza el valor en memoria

jp      PrintExplosion      ; Pinta la explosión y sale

checkCrashShip_endLoop:
inc     hl                  ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashShip_loop ; En bucle hasta que B = 0

ret
```

Si ahora compilamos y cargamos en el emulador, todo vuelve a funcionar.

Pintando números BCD

Vamos a implementar una rutina que pinte los números BCD en pantalla, y como veréis es algo relativamente sencillo. Para calcular el código de carácter de cada uno de los dígitos, solo hay que sumarle el carácter cero.

Abrimos Print.asm y vamos a implementar la rutina que pinte los números BCD, recibiendo en HL la dirección de memoria donde está el número a pintar.

```
PrintBCD:
ld      a, (hl)
and     $f0
rra
rra
rra
rra
add     a, '0'
rst     $10
```

Cargamos el número a pintar en A, ***LD A, (HL)***, nos quedamos con las decenas, ***AND $F0***, ponemos el valor en los bits del cero al tres, ***RRA RRA RRA RRA***, le sumamos el código del carácter 0, ***ADD A, '0'***, y pintamos las decenas, ***RST $10***.

```
ld      a, (hl)
and     $0f
add     a, '0'
rst     $10


ret
```

Cargamos el número a pintar en A, ***LD A, (HL)***, nos quedamos con las unidades, ***AND $0F***, le sumamos el código del carácter 0, ***ADD A, '0'***, y pintamos las unidades, ***RST $10***. Finalmente, salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pinta números en formato BCD
;
; Entrada:  HL -> Puntero al número a pintar
;
; Altera el valor de los registros AF.
; ------------------------------------------------------------------------
PrintBCD:
ld      a, (hl)     ; Carga en A el número a pintar
```

```
and     $f0         ; Se queda con las decenas
rra
rra
rra
rra                 ; Lo pone en los bits 0 a 3
add     a, '0'      ; Le suma el carácter 0
rst     $10         ; Pinta el dígito

ld      a, (hl)     ; Carga el A el número a pintar
and     $0f         ; Se queda con las unidades
add     a, '0'      ; Le suma el carácter 0
rst     $10         ; Pinta el dígito

ret
```

Pintando el marcador

Vamos a implementar la rutina que pinta el marcador: vidas, puntos, nivel y enemigos.

Lo primero que vamos a hacer es definir las constantes de localización de cada uno de los elementos del marcador, para lo que abrimos el archivo Const.asm y añadimos las siguientes líneas:

```
COR_ENEMY:  EQU $1705   ; Coordenadas de la información de los enemigos
COR_LEVEL:  EQU $170d   ; Coordenadas de la información del nivel
COR_LIVE:   EQU $171e   ; Coordenadas de la información de las vidas
COR_POINT:  EQU $1717   ; Coordenadas de la información de los puntos
```

Los valores de la información de la partida los vamos a pintar en la línea de comandos, disponemos de dos líneas en este lugar. Recordad que para la rutina de la ROM que posiciona el cursor, la esquina superior izquierda es $1820, o lo que es lo mismo Y = 24, X = 32, por lo que los valores se van a pintar en la línea 23 y en las columnas 5, 13, 30 y 23. Si restamos a 24 y 32 lo valores de fila y columnas, el resultado son las coordenadas si la esquina superior derecha fuera $0000.

En Var.asm vamos a añadir las definiciones para llevar el control de las vidas y los puntos.

```
livesCounter:
db  $05
pointsCounter:
dw  $0000
```

Y ahora abrimos Print.asm para implementar la rutina que va a pintar los valores del marcador.

```
PrintInfoValue:
ld      a, $05
call    Ink
```

```
ld      a, $01
call    OPENCHAN
```

Cargamos en A la tinta cinco, ***LD A, $05***, y llamamos al cambio de tinta, ***CALL Ink***. Dado que los valores los vamos a pintar en la línea de comandos, cargamos en A el canal uno, ***LD A, $01***, y abrimos el canal, ***CALL OPENCHAN***.

```
ld      bc, COR_LIVE
call    At
ld      hl, livesCounter
call    PrintBCD
```

Cargamos en BC la posición donde pintamos las vidas, ***LD BC, COR_LIVE***, posicionamos el cursor, ***CALL At***, apuntamos HL al contador de vidas, ***LD HL, livesCounter***, y pintamos las vidas, ***CALL PrintBCD***.

```
ld      bc, COR_POINT
call    At
ld      hl, pointsCounter + 1
call    PrintBCD
ld      hl, pointsCounter
call    PrintBCD
```

Cargamos en BC la posición donde pintamos los puntos, ***LD BC, COR_POINT***, posicionamos el cursor, ***CALL At***, apuntamos HL a las unidades de millar y las centenas de los puntos, ***LD HL, pointsCounter + 1***, y lo pintamos, ***CALL PrintBCD***. Apuntamos HL a las decenas y las unidades de los puntos, ***LD HL, pointsCounter***, *y lo pintamos,* **CALL PrintBCD**.

```
ld      bc, COR_LEVEL
call    At
ld      hl, levelCounter + 1
call    PrintBCD
```

Cargamos en BC la posición donde pintamos el nivel, ***LD BC, COR_LEVEL***, posicionamos el cursor, ***CALL At***, apuntamos HL al contador de niveles en formato BCD, ***LD HL, levelCounter + 1***, y lo pintamos, ***CALL PrintBCD***.

```
ld      bc, COR_ENEMY
call    At
ld      hl, enemiesCounter
call    PrintBCD
```

Cargamos en BC la posición donde pintamos el contador de enemigos, ***LD BC, COR_ENEMY***, posicionamos el cursor, ***CALL At***, apuntamos HL al contador de enemigos, ***LD HL, enemiesCounter***, y lo pintamos, ***CALL PrintBCD***.

```
ld      a, $02
call    OPENCHAN


ret
```

Antes de salir, activamos la pantalla superior. Cargamos el canal dos en A, ***LD A, $02***, cambiamos el canal, ***CALL OPENCHAN***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

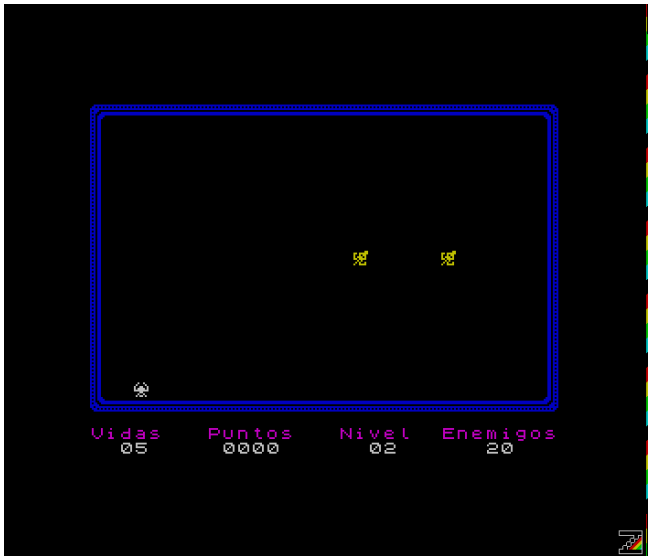
```
; ------------------------------------------------------------------------------
; Pinta los valores de la información de la partida.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
PrintInfoValue:
ld      a, $05                  ; Carga la tinta 5 en A
call    Ink                     ; Cambia la tinta

ld      a, $01                  ; Carga 1 en A
call    OPENCHAN                ; Activa el canal 1, línea de comando

ld      bc, COR_LIVE            ; Carga la posición de las vidas en BC
call    At                      ; Posiciona el cursor
ld      hl, livesCounter        ; Apunta HL al contador de vidas
call    PrintBCD                ; Lo pinta

ld      bc, COR_POINT           ; Carga en BC la posición de los puntos
call    At                      ; Posiciona el cusor
ld      hl, pointsCounter + 1   ; Apunta HL a unidades de millar y centenas
call    PrintBCD                ; Lo pinta
ld      hl, pointsCounter       ; Apunta HL a decenas y unidades
call    PrintBCD                ; Lo pinta

ld      bc, COR_LEVEL           ; Carga en BC la posición de los niveles
call    At                      ; Posiciona el cursor
ld      hl, levelCounter + 1    ; Apunta HL al contador de niveles en BCD
```

```
call    PrintBCD                ; Lo pinta


ld      bc, COR_ENEMY           ; Carga en BC la posición de los enemigos
call    At                      ; Posiciona el cursor
ld      hl, enemiesCounter      ; Apunta HL al contador de enemigos
call    PrintBCD                ; Lo pinta


ld      a, $02                  ; Carga 2 en A
call    OPENCHAN                ; Activa el canal 2, pantalla superior


ret
```

Es el momento de ver si lo que hemos implementado funciona, abrimos Main.asm, localizamos la etiqueta ***Main*** y la instrucción ***DI***, justo encima añadimos la siguiente línea para llamar a pintar la información de la partida:

```
call    PrintInfoValue
```

Localizamos ahora la etiqueta Main_restart, y justo antes de la última línea, ***JR, Main_loop***, añadimos la misma línea de antes. Compilamos, cargamos en el emulador y vemos los resultados.



Como podemos observar, solo se está actualizando el nivel, pero el resto de información de la partida no se actualiza. Además, no se está pintando con el color que hemos definido.

La parte del color es lo primero que vamos a solucionar. La variable de sistema donde cargamos los atributos de la pantalla en la rutina Ink, afecta a la pantalla superior, por lo que no se ve afectada la línea de comandos. Los atributos de la linea de comandos están en la misma variable de sistema donde están los atributos del borde (BORDCR), de manera que vamos a realizar dos modificaciones.

Abrimos el archivo Print.asm, localizamos la rutina ***PrintInfoValue***, y borramos las dos primeras líneas, ***LD A, $05*** y ***CALL Ink***, ya que como hemos visto, no cambia los atributos de la línea de comandos.

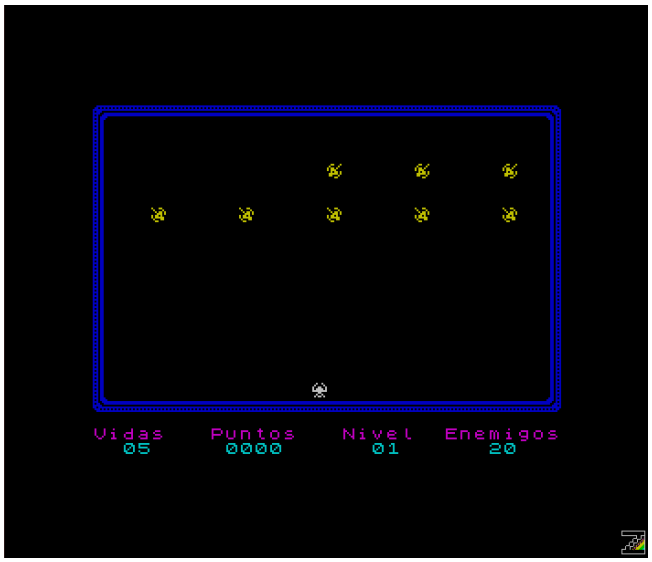
Volvemos al archivo Main.asm, localizamos la etiqueta ***Main***, y vamos a modificar la parte en la que se asigna el color al borde, cuyo aspecto actual es el siguiente:

```
xor     a
out     ($fe), a
ld      a, (BORDCR)
and     $c7
or      $07
ld      (BORDCR), a
```

Vamos a modificar las líneas cuatro y cinco, tras lo cual el aspecto queda de la siguiente manera:

```
xor     a
out     ($fe), a
ld      a, (BORDCR)
and     $c0
or      $05
ld      (BORDCR), a
```

Compilamos, cargamos en el emulador y comprobamos que ahora sí, los valores se pintan en el color elegido.



Y ahora vamos a actualizar el resto de valores del marcador. Cada vez que el disparo alcanza un enemigo, tenemos que restar un enemigo al contador y sumar cinco puntos. Por otro lado, cada vez que un enemigo alcanza nuestra nave, tenemos que restar una vida y restar un enemigo.

Abrimos el archivo Game.asm, localizamos la etiqueta ***checkCrahsFire_endLoop*** y observamos que en las líneas que hay por encima ya se resta un enemigo al contador.

```
ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
```

```
ld      (enemiesCounter), a     ; Actualiza el valor en memoria


ret
```

Nos falta añadir cinco puntos por haber acabado con un enemigo y pintar la información de la partida. Vamos a añadir las siguientes líneas entre ***LD (enemiesCounter), A*** y ***RET***.

```
ld      a, (pointsCounter)
add     a, $05
daa
ld      (pointsCounter), a
ld      a, (pointsCounter + 1)
adc     a, $00
daa
ld      (pointsCounter + 1), a
call    PrintInfoValue
```

Cargamos la unidades y las decenas de los puntos en A, ***LD A, (pointsCounter)***, le añadimos cinco, ***ADD A, $05***, hacemos el ajuste decimal, ***DAA***, y cargamos el valor en memoria, ***LD (pointsCounter), A***.

Sumar cinco a las unidades y el ajuste decimal puede provocar un acarreo, por ejemplo si el valor era noventa y cinco, por lo que tenemos que sumar uno a las centenas, en concreto el acarreo.

Cargamos en A las centenas y las unidades de millar, ***LD A, (pointsCounter + 1)***, sumamos cero con acarreo a A, ***ADC A, $00***, hacemos el ajuste decimal, ***DAA***, cargamos el valor en memoria, ***LD (PointsCounter + 1), A***, y pintamos la información de la partida, ***CALL PrintInfoValue***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones del disparo con los enemigos.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashFire:
ld      a, (flags)              ; Carga los flags en A
and     $02                     ; Evalúa si el disparo está activo
ret     z                       ; Si no está activo, sale


ld      de, (firePos)           ; Carga en DE la posición del disparo
ld      hl, enemiesConfig       ; Apunta HL a la definición del primer enemigo
ld      b, enemiesConfigEnd - enemiesConfigIni  ; Carga en B el número de bytes
                                                ; de la configuración de los enemigos
```

```
sra     b                       ; Lo divide entre dos, B = número de enemigos

checkCrashFire_loop:
ld      a, (hl)                 ; Carga en A la coordenada Y del enemigo
inc     hl                      ; Apunta HL a la coordenada X del enemigo
bit     $07, a                  ; Evalúa si el enemigo está activo
jr      z, checkCrashFire_endLoop   ; Si no está activo, salta
and     $1f                     ; Se queda con la coordenada Y de enemigo
cp      d                       ; Lo compara con la coordenada Y del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales salta
ld      a, (hl)                 ; Carga en A la coordenada X del enemigo
and     $1f                     ; Se que con la coordenada X
cp      e                       ; Lo compara con la coordenada X del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales, salta

dec     hl                      ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)               ; Desactiva el enemigo
ld      b, d                    ; Carga la coordenada Y del disparo en B
ld      c, e                    ; Carga la coordenada X del disparo en C
call    DeleteChar              ; Borra el disparo y/o el enemigo
ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
ld      (enemiesCounter), a     ; Actualiza el valor en memoria
ld      a, (pointsCounter)      ; Carga en A las unidadaes y decenas
add     a, $05                  ; Suma 5
daa                             ; Hace el ajuste decimal
ld      (pointsCounter), a      ; Actualiza el valor en memoria
ld      a, (pointsCounter + 1)  ; Carga en A las centenas y unidades de millar
adc     a, $00                  ; Suma 0 con acarreo
daa                             ; Hace el ajuste decimal
ld      (pointsCounter + 1), a  ; Actualiza el valor en memoria
call    PrintInfoValue          ; Pinta la información de la partida

ret                             ; Sale de la rutina

checkCrashFire_endLoop:
inc     hl                      ; Apunta HL a la coordenada Y del siguiente enemigo
```

```
djnz    checkCrashFire_loop     ; Bucle mientras B > 0


ret
```

En este código hemos utilizado una instrucción de no habíamos visto hasta ahora, ***ADC*** (Add With Carry). Esta instrucción suma el valor indicado a A, más el valor del acarreo, de tal manera que al sumar cero a A, si el acarreo está a uno sumaríamos uno a A, lo que comúnmente conocemos como "me llevo una".

La posibilidades de ADC son:

Mnemócico	Ciclos	Bytes	S	Z	H	P	N	C
ADC A, r	4	1	*	*	*	V	0	*
ADC A, N	7	2	*	*	*	V	0	*
ADC A, (HL)	7	1	*	*	*	V	0	*
ADC A, (IX+N)	19	3	*	*	*	V	0	*
ADC A, (IY+N)	19	3	*	*	*	V	0	*
ADC HL, BC	15	2	*	*	?	V	0	*
ADC HL, DE	15	2	*	*	?	V	0	*
ADC HL, HL	15	2	*	*	?	V	0	*
ADC HL, SP	15	2	*	*	?	V	0	*

* Afecta al flag, V = overflow, 0 = pone el flag a 0, ? = valor desconocido

Ahora ya solo son queda restar una vida cuando la nave choca contra un enemigo. Seguimos en Game.asm, localizamos la etiqueta ***checkCrashShip_endLoop***, *justo por encima encontramos la línea* ***JP PrintExplosion***, y justo por encima vamos a añadir las líneas siguientes:

```
ld      a, (livesCounter)
dec     a
daa
ld      (livesCounter), a
call    PrintInfoValue
```

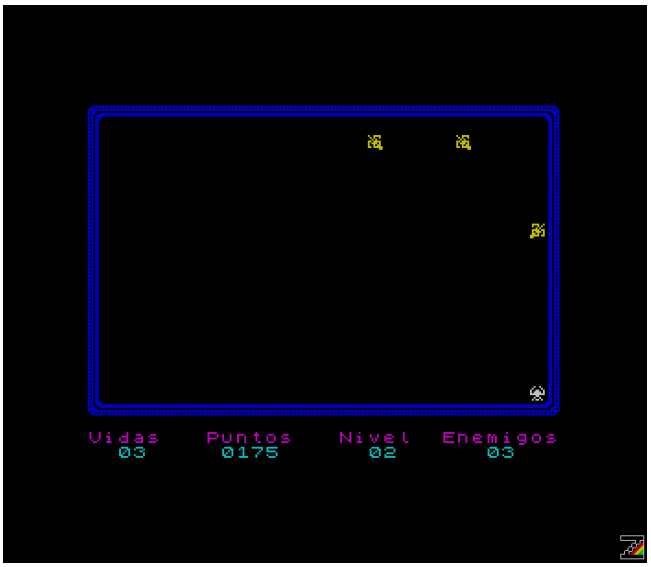
Cargamos en A las vidas, ***LD A, (livesCounter)***, quitamos una, ***DEC A***, hacemos el ajuste decimal, ***DAA***, cargamos el valor en memoria, ***LD (livesCounter), A***, y pintamos la información de la partida, ***CALL PrintInfoValue***. Como podemos ver, al ser el contador de vidas de un solo byte, la modificación ha sido menos que la que hemos realizado para el contador de puntos.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones de los enemigos con la nave.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashShip:
ld      de, (shipPos)       ; Carga en DE la posición de nave
ld      hl, enemiesConfig   ; Apunta HL a la configuración de los enemigos
```

```
ld      b, enemiesConfigEnd - enemiesConfigIni  ; B = bytes totales configuración
sra     b                   ; B = B / 2 = número de enemigos

checkCrashShip_loop:
ld      a, (hl)             ; Carga en A la coordenada Y del enemigo
inc     hl                  ; Apunta HL a la coordenada X del enemigo
bit     $07, a              ; Evalúa si el enemigo está activo
jr      z, checkCrashShip_endLoop   ; Si no lo está, salta

and     $1f                 ; Se queda con la coordenada Y del enemigo
cp      d                   ; Compara con la coordenada Y de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

ld      a, (hl)             ; Carga en A la coordenada X del enemigo
and     $1f                 ; Se queda con la coordenada X de enemigo
cp      e                   ; Compara con la coordenada X de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

dec     hl                  ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)           ; Desactiva el enemigo

ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
ld      (enemiesCounter), a     ; Actualiza el valor en memoria
ld      a, (livesCounter)   ; Carga las vidas en A
dec     a                   ; Quita una
daa                         ; Hace el ajuste decimal
ld      (livesCounter), a   ; Actualiza el valor en memoria
call    PrintInfoValue      ; Pinta la información de la partida

jp      PrintExplosion      ; Pinta la explosión y sale

checkCrashShip_endLoop:
inc     hl                  ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashShip_loop ; En bucle hasta que B = 0

ret
```

Es hora de ver si lo que hemos implementado funciona. Compilamos, cargamos en el emulador y vemos los resultados.



Conclusión

Ya tenemos todo listo para poder empezar a jugar nuestras primeras partidas. Hemos implementado una transición entre niveles y el marcador.

En el siguiente capítulo implementaremos el menú de inicio y el fin de la partida.

0x09 Comienza la partida

En este capítulo vamos a implementar el inicio y el fin de la partida.

Al igual que en capítulos anteriores, creamos la carpeta Paso09 y copiamos desde la carpeta Paso08 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm y Var.asm.

Antes de empezar con el objetivo de este capítulo, vamos a revisar la rutina PrintString para ver dos variaciones.

Rutina PrintString

Las variaciones que vamos a ver de la rutina PrintString las vamos a implementar en un nuevo archivo que vamos a llamar TestPrint.asm, luego decidiremos que rutina va a ser la definitiva.

Abrimos el archivo TestPrint.asm y añadimos el siguiente código.

```
org     $5dad

TestPrint:
ld      hl, string
ld      b, stringEOF - string
call    PrintString

ld      hl, stringNull
call    PrintStringNull

ld      hl, stringFF
call    PrintStringFF

ret

; ------------------------------------------------------------------------
; Pinta cadenas.
;
; Entrada:  HL = primera posición de memoria de la cadena
;           B  = longitud de la cadena.
;
; Altera el valor de los registros AF y HL
; ------------------------------------------------------------------------
PrintString:
ld      a, (hl)         ; Carga en A el carácter a pintar
```

```
rst     $10             ; Pinta el carácter
inc     hl              ; Apunta HL al siguiente carácter
djnz    PrintString     ; Hasta que B valga 0

ret

; ------------------------------------------------------------------------
; Pinta cadenas.
;
; Entrada:  HL = primera posición de memoria de la cadena
;
; Altera el valor de los registros AF y HL
; ------------------------------------------------------------------------
PrintStringNull:
ld      a, (hl)         ; Carga en A el carácter a pintar
or      a               ; Comprueba si es 0
ret     z               ; De ser así, sale
rst     $10             ; Pinta el carácter
inc     hl              ; Apunta HL al siguiente carácter
jr      PrintStringNull ; Bucle hasta que termine de pintar la cadena

; ------------------------------------------------------------------------
; Pinta cadenas.
;
; Entrada:  HL = primera posición de memoria de la cadena
;
; Altera el valor de los registros AF y HL
; ------------------------------------------------------------------------
PrintStringFF:
ld      a, (hl)         ; Carga en A el carácter a pintar
cp      $ff             ; Comprueba si es $FF
ret     z               ; De ser así sale
rst     $10             ; Pinta el carácter
inc     hl              ; Apunta HL al siguiente carácter
jr      PrintStringFF   ; Bucle hasta que termine de pintar la cadena

string:
db      $10, $05, $11, $03, $16, $05, $0a, "Hola Ensamblador"
```

```
stringEOF:
db      $00


stringNull:
db      $10, $07, $11, $01, $16, $07, $0a, "Hola Ensamblador", $00


stringFF:
db      $10, $02, $11, $07, $16, $09, $0a, "Hola Ensamblador", $ff


end     TestPrint
```

En este código podemos ver tres rutinas PrintString:

- PrintString: la rutina tal cual la tenemos ahora.
- PrintStringNull: rutina que pinta cadenas y usa como fin de cadena el carácter nulo ($00).
- PrintStringFF: rutina que pinta cadenas y usa como fin de cadena el carácter $FF.

La primera de estás rutinas ya la conocemos, por lo que vamos a explicar la rutina PrintStringNull, ya que PrintStringFF solo se diferencia en una línea a ésta.

```
PrintStringNull:
ld      a, (hl)
or      a
ret     z
rst     $10
inc     hl
jr      PrintStringNull
```

PrintStringNull y ***PrintStringFF***, reciben en HL la primera posición de la cadena (al igual que ***PrintString***), pero no necesitan conocer la longitud de la misma.

Cargamos en A el carácter al que apunta HL, ***LD A, (HL)***, comprobamos si es cero, ***OR A***, y salimos si es así, ***RET Z***. La línea que cambia en ***PrintStringFF*** con respecto a ***PrintStrinNull*** es ***OR A***, *que la cambiamos por* ***CP $FF***, ya que $FF es el carácter que se usa en este caso como fin de cadena. Debemos recordar que el resultado de ***OR A*** solo es cero si A vale cero.

Si el carácter cargado en A no es el de fin de cadena, pintamos el carácter, ***RST $10***, apuntamos HL al siguiente carácter, ***INC HL***, y seguimos en bucle hasta que pinte toda la cadena, ***JR PrintStringNull***.

El uso de una u otra rutina tiene sus pros y sus contras. La primera comparativa la vamos a realizar sobre bytes y ciclos de reloj.

	Bytes	Ciclos
PrintString	6	47/42
PrintStringNull	7	51/45

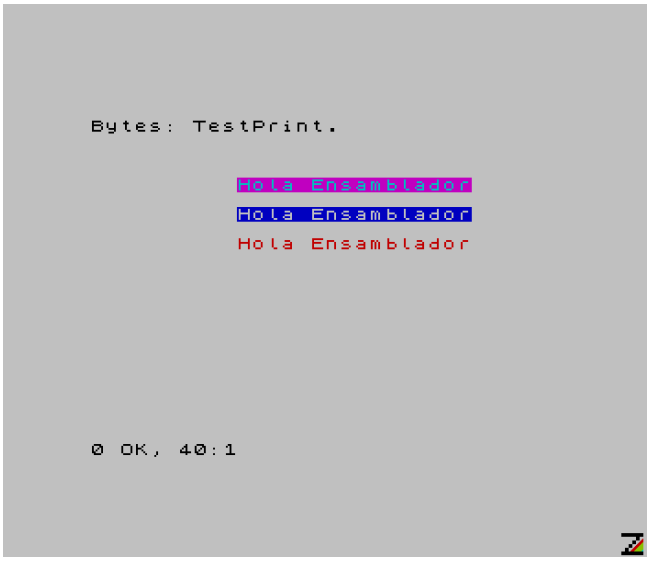
Si vemos esta tabla, la rutina más optima es la primera ya que ocupa menos bytes y es más rápida. La realidad es que sí es más rápida, pero no ocupa menos bytes, ya que cada vez que la llamemos hay que añadir dos bytes de cargar en B la longitud de la cadena. Si usamos mucho esta rutina, rápidamente vemos que el ahorro de bytes no es tal.

La opción lógica es la segunda rutina, que es más rápida y ocupa menos bytes que la tercera, pero como vamos a ver más adelante, tiene sus desventajas.

TestPrint es un programa individual, para compilarlo tenemos que invocar PASMO desde la línea de comandos:

```
pasmo --name TestPrint --tapbas TestPrint.asm TestPrint.bas
```

Antes de continuar, vamos a compilar el programa TestPrint, lo cargamos en el emulador y vemos los resultados.



Como podemos ver, todo ha ido bien. Tenemos tres cadenas, y hemos pintado cada una con una rutina distinta.

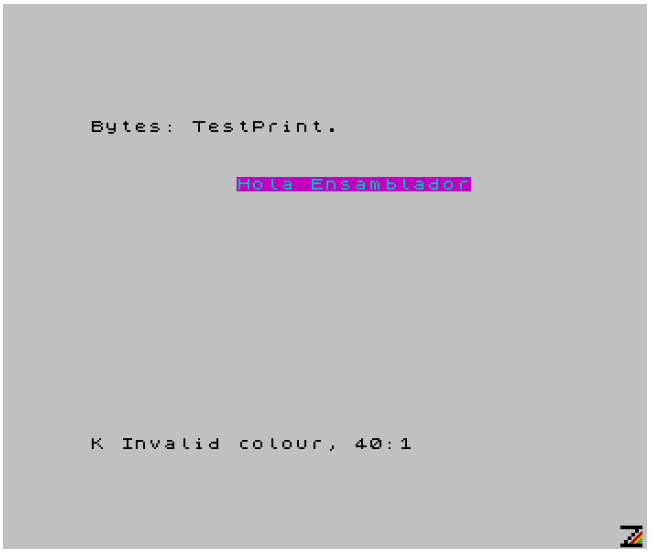
Vamos a ver ahora los inconvenientes de la segunda rutina, para lo cual vamos a modificar la segunda cadena, que ahora es:

```
stringNull:
db      $10, $07, $11, $01, $16, $07, $0a, "Hola Ensamblador", $00
```

Y vamos a modificar el segundo byte, $07, por $00.

```
stringNull:
db      $10, $00, $11, $01, $16, $07, $0a, "Hola Ensamblador", $00
```

Compilamos, cargamos en el emulador y vemos los resultados.



Aquí hay algo que no funciona, pero ¿qué? Revisemos la definición de las cadenas.

```
string:
db      $10, $05, $11, $03, $16, $05, $0a, "Hola Ensamblador"
stringEOF:
db      $00


stringNull:
db      $10, $00, $11, $01, $16, $07, $0a, "Hola Ensamblador", $00


stringFF:
db      $10, $02, $11, $07, $16, $09, $0a, "Hola Ensamblador", $ff
```

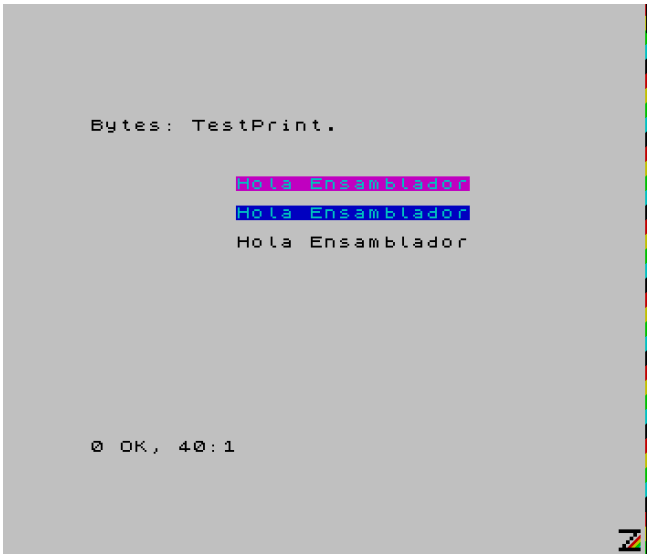
La segunda cadena, ***stringNull***, la terminamos con $00 porque la rutina pinta hasta que se encuentra este valor. Todas las cadenas empiezan con $10, que es el código de ***INK***, por lo que el siguiente byte debe ser un código de color, del $00 al $07.

Cuando se pasa la cadena ***stringNull*** a la rutina ***PrintStringNull***, lee el primer carácter, $10 ***(INK)***, lee el siguiente carácter, $00, y sale.

Lo siguiente que hace el programa es cargar en HL la cadena ***stringFF*** y llamar a la rutina ***PrintStringFF***. Esta rutina lee el primer carácter, $10 ***(INK)***, y lo pinta, pero como lo anterior también ha sido un ***INK***, lo que espera ahora es un código de color, y lo que le pasamos es ***$10 (16)***, un color que no es válido, de ahí el mensaje **K Invalid Colour, 40:1**.

Vamos a volver a modificar el segundo byte de la cadena ***stringNull***, poniéndolo a $05, y vamos a modificar el segundo byte de la cadena ***stringFF***, que ahora vale $02 y lo ponemos a $00.

Compilamos, cargamos en el emulador y vemos los resultados.



Como podemos ver, vuelve a funcionar y pinta la tercera cadena en color negro, $00, por lo que en esta ocasión nos decantamos por la rutina ***PrintStringFF***.

Copiamos el código de la rutina, abrimos el archivo Print.asm, localizamos la rutina ***PrintString*** y sustituimos el código de dicha rutina por el código que acabamos de copiar. El aspecto final de la rutina deber ser el siguiente:

```
; ------------------------------------------------------------------------
; Pinta cadenas terminadas en $FF.
;
; Entrada:  HL = primera posición de memoria de la cadena
;
; Altera el valor de los registros AF y HL
; ------------------------------------------------------------------------
PrintString:
ld      a, (hl)         ; Carga en A el carácter a pintar
cp      $ff             ; Comprueba si es $FF
ret     z               ; De ser así sale
rst     $10             ; Pinta el carácter
inc     hl              ; Apunta HL al siguiente carácter
jr      PrintString     ; Bucle hasta que termine de pintar la cadena
```

Ahora tenemos que modificar la definición de las cadenas y las llamadas a ***PrintString***.

Seguimos en el archivo Print.asm, localizamos la etiqueta **PrintFrame** y borramos la segunda y la quinta línea.

```
ld      hl, frameTopGraph       ; Carga en HL la dirección de la parte superior
ld      b, frameBottomGraph - frameTopGraph ; Carga en B la longitud
call    PrintString             ; Pinta la cadena
```

```
ld      hl, frameBottomGraph    ; Carga en HL la dirección de la parte inferior
ld      b, frameEnd - frameBottomGraph  ; Carga en B la longitud
call    PrintString             ; Pinta la cadena
```

Localizamos la etiqueta ***PrintInfoGame*** y borramos la cuarta línea.

```
ld      a, $01          ; Carga 1 en A
call    OPENCHAN        ; Activa el canal 1, línea de comando
ld      hl, infoGame    ; Carga la dirección de la cadena de títulos en HL
ld      b, infoGame_end - infoGame  ; Carga la longitud en B
```

Abrimos el archivo Var.asm, localizamos la etiqueta ***infoGame*** y después de enemigos añadimos ***$FF***. Borramos la etiqueta ***infoGame_end*** pues ya no tiene ninguna utilidad.

```
infoGame:
db  $10, $03, $16, $00, $00
db 'Vidas   Puntos   Nivel  Enemigos', $ff
```

Localizamos la etiqueta ***frameTopGraph*** y añadimos al final de la definición de los bytes ***$FF***. En la siguiente etiqueta, ***frameBottomGraph***, hacemos lo mismo y borramos la etiqueta ***frameEnd***, que ya no tiene ninguna utilidad.

```
frameTopGraph:
db $16, $00, $00, $10, $01
db $96, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97,
$97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $97, $98, $ff
frameBottomGraph:
db $16, $14, $00
db $9b, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c,
$9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9c, $9d, $ff
```

Compilamos (ahora ya lo volvemos a hacer ejecutando make o make.bat), cargamos en el emulador y comprobamos que todo sigue funcionando. También vemos que el programa ocupa ahora tres bytes menos; con cada línea que hemos quitado en la que cargábamos la longitud de la cadena en B, hemos reducido dos bytes (seis en total), pero al poner $ff al final de las cadenas hemos vuelto a añadir tres bytes.

Inicio y fin de la partida

Vamos a implementar una pantalla de inicio a modo de menú y dos finales, uno para cuando nos matan sin haber conseguido finalizar el juego, y otro para cuando logramos finalizar el juego.

Inicio de la partida

En la pantalla de inicio, vamos a mostrar un texto a modo de presentación, las teclas de control y la selección de los distintos tipos de control.

Abrimos el archivo Var.asm y al inicio del mismo incluimos la definición de la pantalla de inicio.

```
title:
db  $10, $02, $16, $00, $08, "BATALLA ESPACIAL", $0d, $0d, $0d, $ff


firstScreen:
db  $10, $06, "Las naves alienigenas atacan la", $0d
db  "Tierra, el futuro depende de ti.", $0d, $0d
db  "Destruye todos los enemigos que", $0d
db  "puedas, y protege el planeta.", $0d, $0d, $0d
db  $10, $03, "Z - Izquierda", $16, $0a, $15,"X - Derecha"
db  $16, $0c, $0b, "V - Disparo", $0d, $0d
db  $10, $04, "1 - Teclado       3 - Sinclair 1", $0d, $0d
db  "2 - Kempston      4 - Sinclair 2", $0d, $0d, $0d
db $10, $05, "Apunta, dispara, esquiva a las", $0d
db "naves enemigas, vence y libera", $0d
db "al planeta de la amenza."
db $ff
```

Lo primero que hacemos es poner la tinta en rojo, ***$10, $02***, luego posicionamos el cursor en la línea 0, columna 8, ***$16, $00, $08***, pintamos el nombre del juego, ***BATALLA ESPACIAL***, y añadimos tres retornos de carro, ***$0d, $0d, $0d***. Seguimos definiendo el resto de líneas hasta que acabamos con el delimitador de cadena, ***$FF***, que es valor que espera la rutina ***PrintString*** para saber hasta donde tiene que pintar.

Abrimos ahora el archivo Print.asm y al final del mismo vamos a implementar la rutina que pinta la pantalla de inicio y que, más adelante, guardará la elección de controles que hayamos hecho.

```
PrintFirstScreen:
call    CLS
ld      hl, title
call    PrintString
ld      hl, firstScreen
call    PrintString
```

Limpiamos la pantalla, ***CALL CLS***, cargamos en HL la dirección de memoria en la que empieza la definición del título, ***LD HL, title***, lo pintamos, ***CALL PrintString***,cargamos en HL la dirección de memoria en la que empieza la definición de la pantalla, ***LD HL, firtScreen***, y llamamos a la rutina que pinta las cadenas, ***CALL PrintString***.

Dado que más adelante vamos a permitir elegir entre cuatro tipo de controles, lo vamos a ir preparando.

```
printFirstScreen_op:
ld      a, $f7
in      a, ($fe)
```

```
bit     $00, a
jr      nz, printFirstScreen_op
call    FadeScreen


ret
```

Cargamos en A la semifila 1-5, ***LD A, $F7***, leemos el teclado, ***IN A, ($FE)***, comprobamos si se ha pulsado el uno (Teclado), ***BIT $00, A***, y de no ser así sigue en bucle hasta que se pulse el uno, ***JR NZ, printFirstScreen_op***. Realizamos el efecto de fundido de la pantalla, ***CALL FadeScreen***, y salimos, ***RET***.

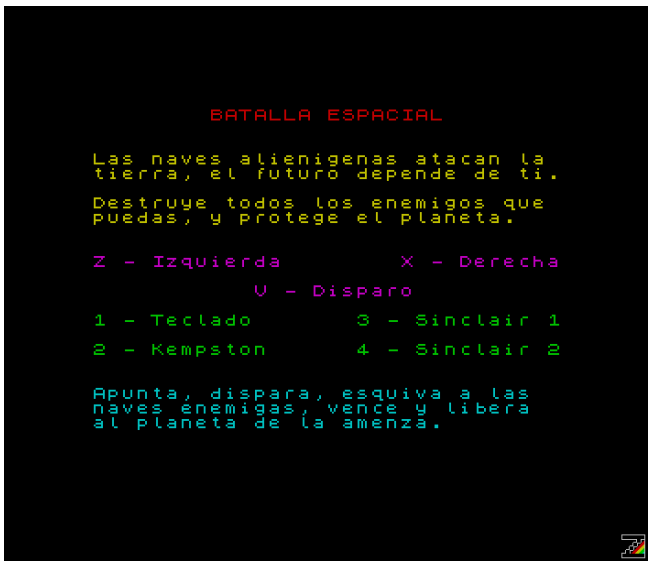
El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pantalla de presentación y selección de controles.
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------------
PrintFirstScreen:
call    CLS                 ; Limpia la pantalla
ld      hl, title           ; Carga en HL la definición del título
call    PrintString         ; Pinta el título
ld      hl, firstScreen     ; Carga en HL la definición de la pantalla
call    PrintString         ; Pinta la pantalla


printFirstScreen_op:
ld      a, $f7              ; Carga en A la semifila 1-5
in      a, ($fe)            ; Lee el teclado
bit     $00, a              ; Comprueba si se ha pulsado el 1
jr      nz, printFirstScreen_op  ; Si no se ha pulsado, sigue hasta que se pulse
call    FadeScreen          ; Fundido de pantalla


ret
```

Es hora de comprobar si lo que acabamos de implementar funciona. Abrimos el archivo Main.asm, localizamos la etiqueta ***Main*** y dentro de la misma la llamada a pintar el marco, ***CALL PrintFirstScreen***. Justo por encima de esta llamada vamos a incluir la llamada a la rutina que pinta la pantalla de inicio y que más adelante servirá para seleccionar el tipo de controles.

```
call    PrintFirstScreen
```

Compilamos, cargamos en el emulador y vemos los resultados.



Como podemos comprobar, ahora sale la pantalla de inicio, y no salimos de ella hasta que pulsamos el uno. Todavía quedan cosas por hacer, pero de momento lo dejamos así y seguimos con el fin de partida.

Fin de la partida

El fin de partida se puede dar de dos modos distintos: se nos acaban las cinco vidas de las que vamos a disponer y perdemos, o superamos el nivel treinta y ganamos.

En base a lo expuesto en el párrafo anterior, vamos a definir dos pantallas de fin distintas. Volvemos al archivo Var.asm y tras la definición de ***firstScreen***, añadimos las definición de las dos pantallas de fin.

```
gameOverScreen:
db  $10, $06, "Has perdido todas tus naves, no", $0d
db  "has podido salvar la Tierra.", $0d, $0d
db  "El planeta ha sido invadido por", $0d
db  "los aliengenas.", $0d, $0d
db  "Puedes volver a intentarlo, de", $0d
db  "ti depende salvar la Tierra.", $ff

winScreen:
db  $10, $06, "Enhorabuena, has destruido a los"
db  "alienigenas, salvaste la Tierra.", $0d, $0d
db  "Los habitantes del planeta te", $0d
db  "estaran eternamente agradecidos.", $ff

pressEnter:
db  $10, $04, $16, $10, $03, "Pulsa Enter para continuar", $ff
```

Igual que hicimos con la pantalla de inicio, vamos a implementar las rutinas que impriman las pantallas de fin, y que esperen la pulsación de la tecla Enter para continuar. Vamos al archivo Print.asm, y nos situamos al final del mismo.

La rutina que vamos a implementar, recibe en registro A el valor cero si es final de partida porque hemos perdido, y distinto de cero si es fin de partida porque hemos ganado.

```
PrintEndScreen:
push    af
call    FadeScreen
ld      hl, title
call    PrintString
pop     af
or      a
jr      nz, printEndScreen_Win
```

Preservamos el valor de A, ***PUSH AF***, hacemos el fundido de pantalla, ***CALL FadeScreen***, apuntamos HL al inicio de la cadena del título, ***LD HL, title***, y la pintamos, ***CALL PrintString***. Recuperamos el valor de AF, ***POP AF***, evaluamos si A vale cero, ***OR A***, y saltamos si no es así, ***JR NZ, printEndScreen_Win***.

```
printEndScreen_GameOver:
ld      hl, gameOverScreen
call    PrintString
jr      printEndScreen_WaitKey
```

Si el valor de A es cero, apuntamos HL al inicio de la definición de la pantalla de fin de partida si hemos perdido, ***LD HL, gameOverScreen***, la pintamos, ***CALL PrintString***, y saltamos para esperar la pulsación de la tecla Enter, ***JR printEndScreen_WaitKey***.

```
printEndScreen_Win:
ld      hl, winScreen
call    PrintString
```

Si el valor de A es distinto de cero, apuntamos HL al inicio de la definición de la pantalla de fin de partida si hemos ganado, ***LD HL, winScreen***, y la pintamos, ***CALL PrintString***.

Preparamos el resto para esperar a que el jugador presione la tecla Enter.

```
printEndScreen_WaitKey:
ld      hl, pressEnter
call    PrintString
call    PrintInfoGame
call    PrintInfoValue
```

Apuntamos HL al inicio de la cadena que pide que se pulse la tecla Enter, ***LD HL, pressEnter***, la pintamos, ***CALL PrintString***, pintamos los títulos de la información de la partida, ***CALL PrintInfoGame***, y pintamos la información de la partida para mostrar al jugador el nivel al que ha llegado y los puntos que ha obtenido, ***CALL PrintInfoValue***.

```
printEndScreen_WaitKeyLoop:
ld      a, $bf
in      a, ($fe)
rra
jr      c, printEndScreen_WaitKeyLoop
call    FadeScreen


ret
```

Cargamos en A la semifila Enter-H, ***LD A, $BF***, leemos el teclado, ***IN A, ($FE)***, rotamos el registro A hacia la derecha, ***RRA***, y seguimos en bucle hasta que el flag de acarreo no esté activo, ***JR C, printEndScreen_WaitKeyLoop***. Una vez que el Enter se ha pulsado, hacemos el fundido de pantalla, ***CALL FadeScreen***, y salimos, ***RET***.

La forma en la que evaluamos si se ha pulsado el uno es la siguiente: cuando leemos del teclado la semifila Enter-H, el bit cero indica si el Enter se ha pulsado o no, a valor uno si no se ha pulsado y a cero si sí se ha pulsado. Al rotar el registro A hacia la derecha, el valor del bit cero se pone en el acarreo, de tal forma que si se activa, es que no se ha pulsado el Enter y si se desactiva, sí se ha pulsado.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Pantalla de fin de partida.
;
; Entrada:  A -> Tipo de fin, 0 = Game Over, !0 = Win.
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------------
PrintEndScreen:
push    af                      ; Preserva el valor de AF
call    FadeScreen              ; Fundido de pantalla
ld      hl, title               ; Apunta HL al título
call    PrintString             ; Pinta el título
pop     af                      ; Recupera el valor de AF
or      a                       ; Evalúa si A vale 0
jr      nz, printEndScreen_Win  ; Si no vale 0, salta


```

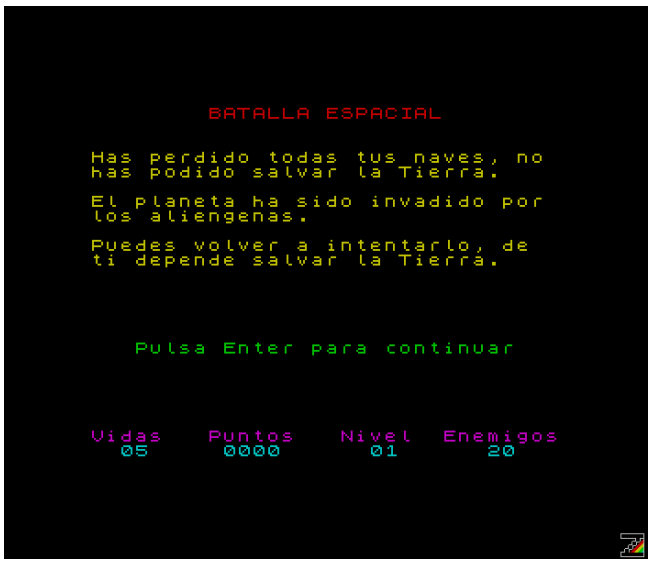
```
printEndScreen_GameOver:
ld      hl, gameOverScreen      ; Apunta HL a la pantalla de Game Over
call    PrintString             ; La pinta
jr      printEndScreen_WaitKey  ; Salta a esperar pulsación de Enter

printEndScreen_Win:
ld      hl, winScreen           ; Apunta HL a la pantalla de Win
call    PrintString             ; La pinta

printEndScreen_WaitKey:
ld      hl, pressEnter          ; Apunta HL a la cadena 'Pulse Enter'
call    PrintString             ; La pinta
call    PrintInfoGame           ; Pinta los títulos de información de la partida
call    PrintInfoValue          ; Pinta los datos de la partida

printEndScreen_WaitKeyLoop:
ld      a, $bf                  ; Carga a semifila Enter-H en A
in      a, ($fe)                ; Lee el teclado
rra                             ; Rota A a la derecha para ver estado del Enter
jr      c, printEndScreen_WaitKeyLoop   ; Si hay acarreo no se ha pulsado, bucle
call    FadeScreen              ; Fundido de pantalla

ret
```

Ahora tenemos que probar si nuestras pantallas de fin de partida se muestran bien, vamos al archivo Main.asm, localizamos la línea ***CALL PrintFirstScreen*** que hemos añadido antes, y justo por encima de ella vamos a añadir las siguientes líneas:

```
xor     a
call    PrintEndScreen
ld      a, $01
call    PrintEndScreen
```

Ponemos A a cero, ***XOR A***, pintamos la pantalla de fin de partida, ***CALL PrintEndScreen***, ponemos A a uno, ***LD A, $01***, y pintamos la pantalla de fin de partida.

Compilamos, cargamos en el emulador y vemos el resultado.



La primera llamada que hacemos a la rutina que pinta la pantalla de fin, la hacemos con A valiendo cero, de ahí que pinte la pantalla correspondiente a cuando hemos perdido todas nuestras vidas.

Si presionamos la tecla Enter, ponemos A a uno y volvemos a llamar a la rutina, esta vez se pinta la pantalla correspondiente a cuando hemos superado los treinta niveles.

Ahora si pulsamos Enter deberíamos ver la pantalla de inicio.

Ahora tenemos que encajar todo esto para que cada cosa esté en su lugar. Lo primero que vamos ha hacer es eliminar las últimas cuatro líneas que hemos utilizado para probar la rutina ***PrintEndScreen*** y las vamos a sustituir por las siguientes para inicializar los datos de la partida:

```
Main_start:
xor     a
ld      hl, enemiesCounter
ld      (hl), $20
inc     hl
ld      (hl), a ; $1d
inc     hl
```

```
ld      (hl), $01 ; $29
inc     hl
ld      (hl), $05
inc     hl
ld      (hl), a
inc     hl
ld      (hl), a


call    ChangeLevel
```

Ponemos A a cero, ***XOR A***. Apuntamos HL al contador de enemigos, ***LD HL, enemiesCounter***, y lo ponemos a veinte en BCD, ***LD (HL), $20***. Apuntamos HL al contador de niveles, ***INC HL***, *y lo ponemos a cero*, ***LD (HL), A***. Apuntamos HL al contador de niveles BCD, ***INC HL***, y lo ponemos a cero, ***LD (HL), A***. Apuntamos HL al contador de vidas, ***INC HL***, y lo ponemos a cinco, ***LD (HL), $05***. Apuntamos HL al primer byte del marcador de puntos en BCD, ***INC HL***, y lo ponemos a cero, ***LD (HL), A***. Apuntamos HL al segundo byte, ***INC HL***, y lo ponemos a cero, ***LD (HL), A***. Por último, llamamos al cambio de nivel para que reinicie los enemigos y cargue el nivel uno.

En las líneas que cargamos el nivel hemos comentado los valores ***$1D*** y ***$29***. Más adelante pondremos estos valores para probar el fin del juego superando tan solo el último nivel.

Buscamos las líneas en las que cargamos el vector de interrupciones, desde ***DI*** hasta ***EI***, las cortamos y las pegamos justo encima de la etiqueta ***MainStart***.

Localizamos la etiqueta ***Main_loop*** y justo al final, encima de ***JR Main_loop*** añadimos la comprobación de si seguimos teniendo vidas.

```
ld      a, (livesCounter)
or      a
jr      z, GameOver
```

Cargamos en A en número de vidas, ***LD A, (livesCounter)***, comprobamos si son cero, ***OR A***, y saltamos si es así, ***JR Z, GameOver***.

Localizamos la etiqueta ***Main_restart***, y justo debajo de ella añadimos la comprobación de si hemos superado el último nivel.

```
ld      a, (levelCounter)
cp      $1e
jr      z, Win
```

Cargamos en A el contador de nivel, ***LD A, (levelCounter)***, evaluamos si estamos en el último, ***CP $1E***, y saltamos si es así, ***JR Z, Win***.

Localizamos la línea ***CALL ChangeLevel***, que está casi al final de ***Main_restart***, la cortamos y la pegamos debajo de ***CALL FadeScreen***, siendo de esta manera la quinta línea de ***Main_restart***.

Vamos al final de ***Main_restart***, y justo debajo vamos a implementar el fin de partida.

```
GameOver:
xor     a
call    PrintEndScreen
jp      Main_start
```

Ponemos A a cero, ***XOR A***, pintamos la pantalla de fin, ***CALL PrintEndScreen***, y volvemos al inicio, ***JP MainStart***.

```
Win:
ld      a, $01
call    PrintEndScreen
jp      Main_start
```

Ponemos A a uno, ***LD A, $01***, pintamos la pantalla de fin, ***CALL PrintEndScreen***, y volvemos al inicio, ***JP MainStart***.

Como podemos ver, en esta ocasión hemos usado ***JP*** en lugar de ***JR***, debido a que si en ***Win*** ponemos ***JR Main_start*** nos da un error de salto fuera de rango.

Vamos a realizar un nuevo cambio, en esta ocasión vamos a hacer que al cambiar de nivel la nave se pinte en la posición inicial. Vamos al archivo Game.asm, localizamos la etiqueta ***changeLevel_end*** *y antes del* **RET** *añadimos lo siguiente:*

```
ld      hl, shipPos             ; Apunta HL a la posición de la nave
ld      (hl), SHIP_INI          ; Carga la posición inicial
```

Apuntamos HL a la posición de la nave, ***LD HL, shipPos***, y cargamos la posición inicial, ***LD (HL), SHIP_INI***. Dado que el Z80 es Little Endian, HL apunta a la coordenada X de la nave y al cargar ***SHIP_INI*** en (HL), carga el segundo byte definido en ***SHIP_INI*** en la coordenada X de la nave, ***LD (HL), $11***.

Y llegamos a la hora de la verdad, compilamos cargamos en el emulador y si toda va bien, al perder las cinco vidas acaba la partida, Game Over.

Volvemos a Main.asm y las líneas:

```
ld      (hl), a ; $1d
inc     hl
ld      (hl), a ; $29
```

Las dejamos como:

```
ld      (hl), $1d
inc     hl
ld      (hl), $29
```

Compilamos, cargamos y al iniciar la partida lo hacemos en el nivel treinta. Lo superamos y fin de partida, Win.

Dado que hemos modificado varias cosas en Main.asm, el aspecto que debe tener ahora es el siguiente:

```
org     $5dad

; ------------------------------------------------------------------------
; Indicadores
;
; Bit 0 -> se debe mover la nave            0 = No, 1 = Sí
; Bit 1 -> el disparo está activo           0 = No, 1 = Sí
; Bit 2 -> se deben mover los enemigos      0 = No, 1 = Sí
; ------------------------------------------------------------------------
flags:
db $00

Main:
ld      a, $02
call    OPENCHAN

ld      hl, udgsCommon
ld      (UDG), hl

ld      hl, ATTR_P
ld      (hl), $07
call    CLS

xor     a
out     ($fe), a
ld      a, (BORDCR)
and     $c0
or      $05
ld      (BORDCR), a

di
ld      a, $28
ld      i, a
im      2
ei
```

```
Main_start:
xor     a
ld      hl, enemiesCounter
ld      (hl), $20
inc     hl
ld      (hl), a ; $1d
inc     hl
ld      (hl), a ; $29
inc     hl
ld      (hl), $05
inc     hl
ld      (hl), a
inc     hl
ld      (hl), a

call    ChangeLevel
call    PrintFirstScreen
call    PrintFrame
call    PrintInfoGame
call    PrintShip
call    PrintInfoValue

call    LoadUdgsEnemies
call    PrintEnemies

Main_loop:
call    CheckCtrl
call    MoveFire

push    de
call    CheckCrashFire
pop     de

ld      a, (enemiesCounter)
cp      $00
jr      z, Main_restart

call    MoveShip
```

```
call    MoveEnemies
call    CheckCrashShip

ld      a, (livesCounter)
or      a
jr      z, GameOver

jr      Main_loop

Main_restart:
ld      a, (levelCounter)
cp      $1e
jr      z, Win

call    FadeScreen
call    ChangeLevel
call    PrintFrame
call    PrintInfoGame
call    PrintShip
call    PrintInfoValue
jr      Main_loop

GameOver:
xor     a
call    PrintEndScreen
jp      Main_start

Win:
ld      a, $01
call    PrintEndScreen
jp      Main_start

include "Const.asm"
include "Var.asm"
include "Graph.asm"
include "Print.asm"
include "Ctrl.asm"
include "Game.asm"
```

```
end     Main
```

Conclusión

Llegados a este punto, ya podemos echar nuestras primeras partidas, aunque todavía nos quedan cosas por hacer.

En el próximo capítulo implementaremos el control con joystick y daremos las posibilidad de obtener vidas extras.

0x0A Joystick y vida extra

En este capítulo vamos a implementar los controles con joystick y a conseguir una vida extra cada quinientos puntos. Creamos la carpeta Paso10 y copiamos desde la carpeta Paso09 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm y Var.asm.

Antes de nada vamos a implementar un retardo entre nivel y nivel, para que nos dé tiempo a prepararnos.

Retardo

Abrimos el archivo Game.asm y al final del mismo vamos a implementar la rutina que va a producir aproximadamente medio segundo de retardo. La ULA produce cincuenta interrupciones por segundo en sistemas PAL, sesenta en NTSC, y vamos a implementar un bucle que espera veinticinco interrupciones.

```
; ------------------------------------------------------------------------
; Espera veinticinco interrupciones.
; ------------------------------------------------------------------------
Sleep:
ld      b, $19      ; Carga veinticinco en B
sleep_Loop:
halt                ; Espera a una interrupción
djnz    sleep_Loop  ; Hasta que B valga 0


ret
```

No explicamos el código pues con los comentarios, y los conocimientos que tenemos hasta ahora, es suficiente para entenderlo.

Para ver el funcionamiento de esta rutina, abrimos el archivo Main.asm, localizamos la etiqueta ***Main_start***, y al final, justo después de ***CALL PrintEnemies***, añadimos la llamada a la rutina de retardo.

```
call    Sleep
```

Localizamos la etiqueta ***Main_restart***, y al final, justo antes de ***JR Main_loop***, añadimos las siguientes líneas:

```
call    PrintEnemies
call    Sleep
```

Ahora compilamos, cargamos en el emulador y vemos que desde que se pitan los enemigos, hasta que se mueven, hay un retardo.

Joystick

Dado que vamos a implementar los controles usando el joystick, además de las teclas, tenemos otras tres posibilidades distintas de control, y en algún sitio tenemos que guardar el tipo de controles que ha seleccionado el jugador. Abrimos el archivo Var.asm, localizamos la etiqueta ***enemiesCounter*** y justo por encima de ella agregamos una nueva etiqueta:

```
controls:
db  $00
```

Aquí vamos a guardar la selección de controles que ha hecho el jugador.

Ahora abrimos el archivo Print.asm y localizamos la etiqueta ***printFirstScreen_op***. Vamos a borrar las líneas ***BIT $00, A y JR NZ, printFirstScreen_op***, pues las vamos a sustituir por la nueva implementación. El resto de líneas las dejamos y justo encima de ***CALL FadeScreen***, vamos a añadir la líneas siguientes:

```
printFirstScreen_end:
ld      a, b
ld      (controls), a
```

Hemos añadido una nueva etiqueta, ***printFirstScreen_end***, y según podemos ver, en B tenemos los controles que se han seleccionado, lo cargamos en A, ***LD A, B***, y de ahí lo cargamos en memoria, ***LD (controls), A***.

Ahora vamos a implementar el resto de la rutina en el lugar donde estaban las líneas que hemos borrado, justo debajo de la lectura del teclado, ***IN A, ($FE)***.

```
ld      b, $01
rra
jr      nc, printFirstScreen_end
inc     b
rra
jr      nc, printFirstScreen_end
inc     b
rra
jr      nc, printFirstScreen_end
inc     b
rra
jr      c, printFirstScreen_op
```

Conviene que recordemos que al leer el teclado, el estado de las teclas vienen en los bits del cero a cuatro, correspondiendo el bit cero con la tecla más alejada del centro del teclado y el cuatro con la más cercana. También conviene recordar que el bit viene a cero si la tecla se ha pulsado, y a uno si no se ha pulsado.

Ponemos B a uno, la opción de teclas, ***LD B, $01***, rotamos A a la derecha, poniendo el valor del bit cero (tecla 1) en el flag de acarreo, ***RRA***, y si el flag de acarreo se ha desactivado, el bit estaba a cero, la tecla se ha pulsado y saltamos porque han seleccionado teclado, ***JR NC, printFirstScreen_end***.

Si el flag de acarreo está activo, incrementamos B para que contenga el valor dos (Kempston), volvemos a rotar poniendo el valor del bit cero (tecla 2 tras la rotación anterior) en el flag de acarreo, ***RRA***, e igual que antes, salta si se ha desactivado el acarreo, ***JR NC, printFirstScreen_end***.

Si no se ha pulsado la tecla 2, rotamos y comprobamos las teclas 3 y 4, con especial atención al último **JR**, *en este caso* ***JR C, printFirstScreen_op***. Si no se ha pulsado tampoco la tecla 4, el acarreo está activo y salta para volver a leer en teclado y estar en bucle hasta que se pulse alguna tecla del 1 al 4.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Pantalla de presentación y selección de controles.
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------------------
PrintFirstScreen:
call    CLS                 ; Limpia la pantalla
ld      hl, title           ; Carga en HL la definición del título
call    PrintString         ; Pinta el título
ld      hl, firstScreen     ; Carga en HL la definición de la pantalla
call    PrintString         ; Pinta la pantalla


printFirstScreen_op:
ld      a, $f7              ; Carga en A la semifila 1-5
in      a, ($fe)            ; Lee el teclado
ld      b, $01              ; Carga 1 en B, opción teclas
rra                         ; Rota A a la derecha para saber si ha pulsado 1
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Kempston
rra                         ; Rota A a la derecha para saber si ha pulsado 2
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Sinclar 1
rra                         ; Rota A a la derecha para saber si ha pulsado 3
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Sinclar 2
rra                         ; Rota A a la derecha para saber si ha pulsado 4
```

```
jr      c, printFirstScreen_op      ; Si hay acarreo, no se ha pulsado, bucle


printFirstScreen_end:
ld      a, b                ; Carga en A la opción seleccionada
ld      (controls), a       ; Lo carga en memoria
call    FadeScreen          ; Fundido de pantalla


ret
```

Y ahora hay que utilizar los controles que se hayan seleccionado, y lo primero que debemos saber es la manera de leer el joystick.

En el caso de los joystick Sinclair, cada uno de ellos está mapeado con una semifila del teclado, en el caso del Kempston, no. Otra diferencia es que en el caso de los joystick Sinclair, las direcciones pulsadas vienen a cero, mientras que en los Kempston vienen a uno.

A continuación, vemos una tabla en la que se detalla la manera de leer las pulsaciones del joystick, y en que bit tenemos cada dirección.

Joystick	Semifila	Puerto	Arriba	Abajo	Izquierda	Derecha	Disparo
Sinclair 1	$EF (0-6)	$FE	1	2	4	3	0
Sinclair 2	$F7 (1-5)	$FE	3	2	0	1	4
Kempston		$1F	3	2	1	0	4

Con estos datos, ya podemos abrir el archivo Ctrl.asm y modificar rutina ***CheckCtrl*** para que tenga en cuenta los cuatro tipos de controles disponibles.

La primera línea de esta rutina es ***LD D, $00*** y justo debajo de ella vamos a implementar la gestión de los controles. Borramos desde justo debajo de ***LD D, $00*** hasta justo encima de ***RET***, quedando de la siguiente manera:

```
CheckCtrl:
ld      d, $00              ; Pone D a 0


ret
```

Al final de la rutina comprobábamos si se habían pulsado a la vez izquierda y derecha, ***checkCtrl_testLR***, y de ser así omitíamos ambas pulsaciones. Vamos a prescindir de esta comprobación ya que si se pulsan las dos, la nave se moverá hacia la derecha (ver ***MoveShip*** en Game.asm) y quitando esta parte de la rutina ahorramos diez bytes y treinta y ocho o cuarenta y cuatro ciclos de reloj.

Y ahora empezamos a implementar justo después de ***LD D, $00***.

```
ld      a, (controls)
dec     a
jr      z, checkCtrl_Keys
```

```
dec     a
jr      z, checkCtrl_Kempston
dec     a
jr      z, checkCtrl_Sinclair1
```

Cargamos en A los controles seleccionados por el jugador, que puede ser un valor que va del uno al cuatro, ***LD A, (controls)***, y decrementamos A, ***DEC A***. Si A valía uno, tras el decremento vale cero y saltamos, ***JR Z, checkCtrl_Keys***. Si no se ha seleccionado el teclado, decrementamos de nuevo A y comprobamos si se ha seleccionado Kempston, y si no es así hacemos los mismo para comprobar si se ha seleccionado Sinclair 1. Si no se ha seleccionado ninguna de las opciones anteriores, se ha seleccionado Sinclair 2.

Anteriormente, para comprobar si se ha pulsado una tecla usábamos la instrucción ***BIT n, r***, que ocupa dos bytes y tarda ocho ciclos de reloj. Dado que con una sola lectura al puerto obtenemos el estado de todas las direcciones, en esta ocasión vamos a utilizar rotaciones del registro A, que ocupan un byte y tardan cuatro ciclos de reloj. En concreto, vamos a utilizar un máximo de cinco rotaciones, ocupando cinco bytes y tardando veinte ciclos de reloj. La opción sería usar tres instrucciones ***BIT*** que ocupan seis bytes y veintiocho ciclos de reloj, por lo que con las rotaciones ahorramos bytes y ciclos de reloj.

```
checkCtrl_Sinclair2:
ld      a, $f7
in      a, ($fe)
checkCtrl_Sinclair2_left:
rra
jr      c, checkCtrl_Sinclair2_right
set     $00, d
checkCtrl_Sinclair2_right:
rra
jr      c, checkCtrl_Sinclair2_fire
set     $01, d
checkCtrl_Sinclair2_fire:
and     $04
ret     nz
set     $02, d
ret
```

Cargamos en A la semifila 1-5, ***LD A, $F7***, y leemos el teclado, ***IN A, ($FE)***. Rotamos A hacia la derecha para comprobar si se ha pulsado la dirección izquierda, ***RRA***, y en caso de no haberse pulsado se activa el flag de acarreo y salta, ***JR C, chechCtrl_Sinclair2_right***. Si sí se ha pulsado, activamos el bit cero de D, ***SET $00, D***.

Rotamos A hacia la derecha para comprobar si se ha pulsado la dirección derecha, ***RRA***, y en caso de no haberse pulsado se activa el flag de acarreo y salta, ***JR C, chechCtrl_Sinclair2_fire***. Si sí se ha pulsado, activamos el bit uno de D, ***SET $01, D***.

Ahora, el disparo lo tenemos en el bit dos, comprobamos si está pulsado, ***AND $04***, y salta si no lo está, ***RET NZ***. Si sí se ha pulsado, activamos el bit dos de D, ***SET $02, D*** y salimos, ***RET***.

Ahora vamos a gestionar la selección Kempston.

```
checkCtrl_Kempston:
in      a, ($1f)
checkCtrl_Kempston_right:
rra
jr      nc, checkCtrl_Kempston_left
set     $01, d
checkCtrl_Kempston_left:
rra
jr      nc, checkCtrl_Kempston_fire
set     $00, d
checkCtrl_Kempston_fire:
and     $04
ret     z
set     $02, d
ret
```

Leemos el puerto treinta y uno, ***IN A, ($1F)***. Rotamos A hacia la derecha para comprobar si se ha pulsado la dirección derecha, ***RRA***, y en caso de no haberse pulsado se desactiva el flag de acarreo y salta, ***JR NC, chechCtrl_Kempston_left***. Si sí se ha pulsado, activamos el bit uno de D, ***SET $01, D***.

Rotamos A hacia la derecha para comprobar si se ha pulsado la dirección izquierda, ***RRA***, y en caso de no haberse pulsado se desactiva el flag de acarreo y salta, ***JR NC, chechCtrl_Kempston_fire***. Si sí se ha pulsado, activamos el bit uno de D, ***SET $00, D***.

Ahora, el disparo lo tenemos en el bit dos, comprobamos si está pulsado, ***AND $04***, y salta si no lo está, ***RET Z***. Si sí se ha pulsado, activamos el bit dos de D, ***SET $02, D*** y salimos, ***RET***.

La gestión de la selección Sinclair 1 y teclado es igual a la de Sinclair 2, cambiando el orden de comprobación de las direcciones, por lo que vamos a ver el aspecto final de la rutina.

```
; ------------------------------------------------------------------------
; Evalúa si se ha pulsado alguna de la teclas de dirección.
; Las teclas de dirección son:
;     Z      ->     Izquierda
;     X      ->     Derecha
;     V      ->     Disparo
```

```
;
; Kempston, Sinclair 1 y Sinclair 2
;
; Retorna:  D      ->     Teclas pulsadas.
;                  Bit 0  ->     Izquierda
;                  Bit 1  ->     Derecha
;                  Bit 2  ->     Disparo
;
; Altera el valor de los registros A y D
; ------------------------------------------------------------------------------
CheckCtrl:
ld      d, $00                  ; Pone D a 0
ld      a, (controls)           ; Carga en A la selección de controles
dec     a                       ; Decrementa A
jr      z, checkCtrl_Keys       ; Si es 0 salta a control teclado
dec     a                       ; Decrementa A
jr      z, checkCtrl_Kempston   ; Si es 0 salta a control Kempston
dec     a                       ; Decrementa A
jr      z, checkCtrl_Sinclair1  ; Si es 0 salta a control Sinclair 1

; Control Sinclair 2
checkCtrl_Sinclair2:
ld      a, $f7                  ; Carga la semifila 1-5 en A
in      a, ($fe)                ; Lee el teclado
checkCtrl_Sinclair2_left:
rra                             ; Rota A para comprobar izquierda
jr      c, checkCtrl_Sinclair2_right ; Si hay acarreo, no pulsado, salta
set     $00, d                  ; Si no hay acarreo, activa bit izquierda
checkCtrl_Sinclair2_right:
rra                             ; Rota A para comprobar derecha
jr      c, checkCtrl_Sinclair2_fire ; Si hay acarreo, no pulsado, salta
set     $01, d                  ; Si no hay acarreo, activa bit derecha
checkCtrl_Sinclair2_fire:
and     $04                     ; Comprueba si el disparo está activo
ret     nz                      ; Si no es cero, no pulsado, sale
set     $02, d                  ; Si es cero, activa bit disparo
ret                             ; Sale
```

```
; Control Kempston
checkCtrl_Kempston:
in      a, ($1f)                ; Lee el puerto 31
checkCtrl_Kempston_right:
rra                             ; Rota A para comprobar derecha
jr      nc, checkCtrl_Kempston_left ; Si no hay acarreo, no pulsado, salta
set     $01, d                  ; Si hay acarreo, activa bit derecha
checkCtrl_Kempston_left:
rra                             ; Rota A para comprobar izquierda
jr      nc, checkCtrl_Kempston_fire ; Si no hay acarreo, no pulsado, salta
set     $00, d                  ; Si hay acarreo, activa bit izquierda
checkCtrl_Kempston_fire:
and     $04                     ; Comprueba si el disparo está activo
ret     z                       ; Si es cero, no pulsado, sale
set     $02, d                  ; Si no es cero, activa bit disparo
ret                             ; Sale

; Control Sinclair 1
checkCtrl_Sinclair1:
ld      a, $ef                  ; Carga la semifila 0-6 en A
in      a, ($fe)                ; Lee el teclado
checkCtrl_Sinclair1_fire:
rra                             ; Rota A para comprobar disparo
jr      c, checkCtrl_Sinclair1_right    ; Si hay acarreo, no pulsado, salta
set     $02, d                  ; Si no hay acarreo, activa bit disparo
checkCtrl_Sinclair1_right:
rra
rra
rra                             ; Rota A para comprobar derecha
jr      c, checkCtrl_Sinclair1_left ; Si hay acarreo, no pulsado, salta
set     $01, d                  ; Si no hay acarreo, activa bit derecha
checkCtrl_Sinclair1_left:
rra                             ; Rota A para comprobar izquierda
ret     c                       ; Si hay acarreo, no pulsado, sale
set     $00, d                  ; Si no hay acarreo, activa bit izquierda
ret                             ; Sale

checkCtrl_Keys:
```

```
ld      a, $fe                  ; Carga la semifila Cs-V en A
in      a, ($fe)                ; Lee el teclado
checkCtrl_Key_left:
rra
rra                             ; Rota A para comprobar izquierda
jr      c, checkCtrl_right      ; Si hay acarreo, no pulsado, salta
set     $00, d                  ; Si no hay acarreo, activa bit izquierda
checkCtrl_right:
rra                             ; Rota A para comprobar derecha
jr      c, checkCtrl_fire       ; Si hay acarreo, no pulsado, salta
set     $01, d                  ; Si no hay acarreo, activa bit derecha
checkCtrl_fire:
and     $02                     ; Comprueba si el disparo está activo
ret     nz                      ; Si no es cero, no pulsado, sale
set     $02, d                  ; Si es cero, activa bit disparo
ret                             ; Sale
```

Compilamos, cargamos en el emulador y probamos los distintos controles.

Vida extra

Vamos a implementar que cada quinientos putos conseguidos el jugador obtenga una vida extra.

Vamos al archivo Var.asm, localizamos la etiqueta ***pointsCounter***, y justo debajo de ella vamos a añadir una nueva etiqueta:

```
extraCounter:
dw  $0000
```

En ***extraCounter*** vamos a controlar el acumulado de puntos, hasta que llegue a quinientos, para dar una vida extra.

El siguiente paso es inicializar el valor de ***extraCounter*** con cada inicio de partida. Vamos al archivo Main.asm, localizamos la etiqueta ***Main_start***, y nos fijamos en las primeras líneas:

```
xor a
ld      hl, enemiesCounter
ld      (hl), $20
inc     hl
ld      (hl), a ; $1d
inc     hl
ld      (hl), a ; $29
inc     hl
ld      (hl), $05
```

```
inc     hl
ld      (hl), a
inc     hl
ld      (hl), a
```

En esta parte inicializamos los valores de la partida, y como vemos ocupa diecisiete bytes y tarda noventa y dos ciclos de reloj. Cada pareja de instrucciones ***INC HL*** y ***LD (HL), A***, ocupa dos bytes y tarda trece ciclos de reloj. Dado que tendríamos que añadir dos parejas más para inicializar los dos nuevos bytes que hemos añadido con la etiqueta ***extraCounter***, añadiríamos cuatro bytes y veintiséis ciclos de reloj, resultando en un total de veintiún bytes y ciento dieciocho ciclos de reloj, además de que el código crece de manera repetitiva.

En lugar de inicializar los valores como hacemos ahora, vamos a usar la instrucción ***LDIR***, que copia el valor de la posición memoria a la que apunta el registro HL en la posición de memoria a la que apunta el registro DE. Tras la copia, incrementa HL, incrementa DE y decrementa BC. Repite estas operaciones hasta que BC sea cero.

Borramos el código que usamos para inicializar los valores y lo sustituimos por el siguiente:

```
ld      hl, enemiesCounter
ld      de, enemiesCounter + 1
ld      (hl), $00
ld      bc, $08
ldir
ld      a, $05
ld      (livesCounter), a
```

Apuntamos HL a la posición de memoria dónde se encuentra el contador de enemigos, ***LD HL, enemiesCounter***, y apuntamos DE a la posición siguiente.

Ponemos la posición de memoria a la que apunta HL a cero, ***LD (HL), $00***, cargamos en BC el número de posiciones que vamos a poner a cero, además de la primera, ***LD BC, $08***, y ponemos a cero el resto de posiciones de memoria, ***LDIR***.

No todos los valores se inician a cero, las vidas se inician a cinco, por lo que cargamos cinco en A, ***LD A, $05***, y lo subimos a memoria, ***LD (livesCounter), A***.

De la manera en la que acabamos de implementar la inicialización, el código ocupa dieciocho bytes y tarda ochenta y un ciclos de reloj, por lo que hemos ganado bytes y tiempo de proceso. De igual manera, el código queda más legible, y en el caso de que necesitemos añadir algún byte más que haya que inicializar, solo tendremos que cambiar el valor que cargamos en BC.

El aspecto final del inicio de ***Main_start*** es el siguiente:

```
Main_start:
ld      hl, enemiesCounter
ld      de, enemiesCounter + 1
ld      (hl), $00
```

```
ld      bc, $08
ldir
ld      a, $05
ld      (livesCounter), a


call    ChangeLevel
```

Ya solo queda la parte final, acumular puntos en ***extraCounter***, dar una vida extra y poner el contador a cero al llegar a quinientos puntos.

Vamos al archivo Game.asm, localizamos la etiqueta ***ChecCrashFire*** y vamos al final de la misma y vemos que el aspecto es el siguiente:

```
ld      (pointsCounter + 1), a  ; Actualiza el valor en memoria
call    PrintInfoValue          ; Pinta la información de la partida


ret                             ; Sale de la rutina


checkCrashFire_endLoop:
inc     hl                      ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashFire_loop     ; Bucle mientras B > 0


ret
```

La parte en la que vamos a acumular los puntos para lograr la vida extra va entre las líneas ***LD (pointsCounter + 1), A*** y ***CALL PrintInfoValue***, así que procedemos:

```
ld      hl, (extraCounter)
ld      bc, $0005
add     hl, bc
ld      (extraCounter), hl
ld      bc, $01f4
sbc     hl, bc
jr      nz, checkCrashFire_cont
ld      (extraCounter), hl
ld      a, (livesCounter)
inc     a
daa
ld      (livesCounter), a
checkCrashFire_cont:
```

Si se llega a esta parte de la rutina es porque se ha alcanzado a un enemigo y hemos sumado cinco puntos.

Cargamos el contador de puntos para vida extra en HL, ***LD HL, (extraCounter)***, cargamos los cinco puntos que vamos a sumar en BC, ***LD BC, $0005***, se lo sumamos a HL, ***ADD HL, BC***, y lo actualizamos en memoria, ***LD (extraCounter), HL***.

Cargamos en BC quinientos, ***LD BC, $01F4***, se lo restamos a HL, ***SBC HL, BC***, y si el resultado no es cero saltamos pues no se ha llegado a quinientos puntos, ***JR NZ, checkCrashFire_cont***. Si el valor de HL tras la resta fuera cero, sí habríamos llegado a quinientos puntos.

SBC es la resta con acarreo, la única resta que nos permite realizar el Z80 al operar con registros de 16 bits. En este caso concreto, es muy importante que el flag de acarreo este desactivado, cosa que sabemos que es así pues antes de la resta hemos sumado cinco a HL y, en nuestro caso, el valor de HL nunca va a superar quinientos.

Si no hemos saltado es porque el valor de HL había llegado a quinientos, ahora es cero. Actualizamos el contador en memoria poniéndolo a cero, ***LD (extraCounter), HL***, cargamos el contador de vidas en A, ***LD A, (livesCounter)***, incrementamos A para añadir una vida, ***INC A***, hacemos el ajuste decimal, ***DAA***, y actualizamos el valor en memoria, ***LD (livesCounter), A***. Por último, antes de la línea ***CALL PrintInfoValue***, añadimos la etiquetaaa la que salta si no hemos llegado a quinientos puntos, ***checkCrashFire_cont***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones del disparo con los enemigos.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashFire:
ld      a, (flags)              ; Carga los flags en A
and     $02                     ; Evalúa si el disparo está activo
ret     z                       ; Si no está activo, sale

ld      de, (firePos)           ; Carga en DE la posición del disparo
ld      hl, enemiesConfig       ; Apunta HL a la definición del primer enemigo
ld      b, enemiesConfigEnd - enemiesConfigIni  ; Carga en B el número de bytes
                                                ; de la configuración de los enemigos
sra     b                       ; Lo divide entre dos, B = número de enemigos

checkCrashFire_loop:
ld      a, (hl)                 ; Carga en A la coordenada Y del enemigo
inc     hl                      ; Apunta HL a la coordenada X del enemigo
bit     $07, a                  ; Evalúa si el enemigo está activo
jr      z, checkCrashFire_endLoop   ; Si no está activo, salta
and     $1f                     ; Se queda con la coordenada Y de enemigo
```

```
cp      d                       ; Lo compara con la coordenada Y del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales salta
ld      a, (hl)                 ; Carga en A la coordenada X del enemigo
and     $1f                     ; Se queda con la coordenada X
cp      e                       ; Lo compara con la coordenada X del disparo
jr      nz, checkCrashFire_endLoop  ; Si no son iguales, salta

dec     hl                      ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)               ; Desactiva el enemigo
ld      b, d                    ; Carga la coordenada Y del disparo en B
ld      c, e                    ; Carga la coordenada X del disparo en C
call    DeleteChar              ; Borra el disparo y/o el enemigo
ld      a, (enemiesCounter)     ; Carga en A el número de enemigos
dec     a                       ; Resta uno
daa                             ; Hace el ajuste decimal
ld      (enemiesCounter), a     ; Actualiza el valor en memoria
ld      a, (pointsCounter)      ; Carga en A las unidades y decenas
add     a, $05                  ; Suma 5
daa                             ; Hace el ajuste decimal
ld      (pointsCounter), a      ; Actualiza el valor en memoria
ld      a, (pointsCounter + 1)  ; Carga en A las centenas y unidades de millar
adc     a, $00                  ; Suma 1 con acarreo
daa                             ; Hace el ajuste decimal
ld      (pointsCounter + 1), a  ; Actualiza el valor en memoria
ld      hl, (extraCounter)      ; Carga en HL el contador de vida extra
ld      bc, $0005               ; Carga 5 en BC
add     hl, bc                  ; Se lo suma a HL
ld      (extraCounter), hl      ; Lo carga en memoria
ld      bc, $01f4               ; Carga 500 en BC
sbc     hl, bc                  ; Se lo resta a HL
jr      nz, checkCrashFire_cont ; Si el resultado no es 0, salta
ld      (extraCounter), hl      ; Si es 0, pone a cero el contador de vida extra
ld      a, (livesCounter)       ; Carga en A el contador de vidas
inc     a                       ; Suma una vida
daa                             ; Hace el ajuste decimal
ld      (livesCounter), a       ; Actualiza en memoria
checkCrashFire_cont:
call    PrintInfoValue          ; Pinta la información de la partida
```

```
ret                             ; Sale de la rutina


checkCrashFire_endLoop:
inc     hl                      ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashFire_loop     ; Bucle mientras B > 0


ret
```

Compilamos, cargamos en el emulador y vemos los resultados; cada quinientos puntos conseguimos una vida extra.

Para poder verificar que funciona, podemos iniciar la partida con el ***extraCounter*** a $1EF (495), de tal manera que al alcanzar un enemigo conseguiremos una vida.

También podéis hacer algo de lo que quizá ya os hayáis percatado, al iniciar el nivel, con el disparo pulsado, desplazaos hacia la derecha y quedaos allí, iréis pasando casi todos los niveles sin que os maten. Este es un aspecto que tenemos que cambiar, de lo contrario se pueden pasar los treinta niveles usando esta técnica.

Cambio del disparo

Lo primero que vamos a hacer es cambiar el disparo, a ver si así solucionamos algo. Seguimos en el archivo Game.asm, localizamos la etiqueta ***MoveFire***, y tras la primera linea, ***LD HL, flags***, añadimos las siguientes:

```
bit     $00, (hl)
ret     z
```

Comprobamos si el bit cero está activo, **BIT $00, (HL)**, y salimos si no lo está.

El bit cero de flags es el que indica si debemos mover la nave, de manera que ahora el disparo se mueve con la misma cadencia que la nave.

Compilamos, cargamos en el emulador y vemos los resultados.

Como podréis comprobar vosotros mismos, la técnica de desplazarse haca la derecha ya no resulta, pero a mi me gusta más que el disparo de la sensación de que es continuo, y además habría que pulir algo más el movimiento ya que cuando quedan pocos enemigos, se queda parado.

Vamos a comentar las dos líneas que hemos añadido pues la solución va a estar en modificar el comportamiento de los enemigos.

Conclusión

En este capítulo hemos implementado un retardo para el cambio entre niveles, el control por joystick y la consecución de vidas extra. También hemos visto un truco para pasar todos los niveles sin el más mínimo esfuerzo, y hemos probado a cambiar el comportamiento del disparo para evitar esto, pero no nos ha convencido.

En el próximo capítulo vamos a centrarnos en modificar el comportamiento de los enemigos.

0x0B Comportamiento de los enemigos

En este capítulo nos vamos a centrar en el comportamiento de los enemigos.

Aunque al inicio del tutorial comenté que el desarrollo estaba hecho y que lo único que iba a hacer es tutorizarlo, la realidad es que según he ido revisando el código, he ido cambiando cosas con respecto del original, y una de ellas es en lo relativo al comportamiento de los enemigos.

Creamos la carpeta Paso11 y copiamos desde la carpeta Paso10 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm y Var.asm.

Cambios de dirección

Para dar la sensación de que el movimiento de los enemigos es algo menos previsible, vamos a hacer que cada cuatro segundos se cambie la dirección de los mismos. En el programa original usaba una rutina que generaba números pseudoaleatorios, cosa que voy a simplificar en esa ocasión.

Lo primero que vamos a hacer es abrir el archivo Main.asm, localizamos la etiqueta ***flags*** al inicio del mismo, y añadimos un comentario para el bit tres.

```
; ------------------------------------------------------------------------------
; Indicadores
;
; Bit 0 -> se debe mover la nave            0 = No, 1 = Sí
; Bit 1 -> el disparo está activo           0 = No, 1 = Sí
; Bit 2 -> se deben mover los enemigos      0 = No, 1 = Sí
; Bit 3 -> cambia dirección enemigos        0 = No, 1 = Sí
; ------------------------------------------------------------------------------
flags:
db $00
```

Cada cuatro segundos, activaremos el bit tres y se cambiará la dirección de los enemigos.

Para que el cambio de dirección de los enemigos no sea siempre el mismo, vamos a utilizar una etiqueta auxiliar; abrimos el archivo Var.asm, localizamos la etiqueta ***extraCounter*** y añadimos las líneas siguientes:

```
; ------------------------------------------------------------------------------
; Valores auxiliares
; ------------------------------------------------------------------------------
swEnemies:
db  $00
enemiesColor:
db  $06
```

La etiqueta que vamos a usar es ***swEnemies***. Como podéis ver, he añadido otra etiqueta más, que vamos a usar para añadir un pequeño efecto de color a los enemigos.

Ahora vamos a implementar la rutina que va a realizar el cambo de dirección de los enemigos. Abrimos el archivo Game.asm, e implementamos al principio la rutina que cambia la dirección de los enemigos.

```
ChangeEnemies:
ld      hl, flags
bit     $03, (hl)
ret     z
res     $03, (hl)

ld      b, $14
ld      hl, enemiesConfig
ld      a, (swEnemies)
ld      c, a
```

Cargamos en HL la dirección de los flags, ***LD HL, flags***, comprobamos si el bit de cambio de dirección está activo, ***BIT $03, (HL)***, y salimos si no lo está, ***RET Z***.

Desactivamos el bit si está activo, ***RES $03, (HL)***, cargamos en B el número total de enemigos, ***LD B, $14***, cargamos en HL la dirección de la configuración de los enemigos, ***LD HL, enemiesConfig***, cargamos en A el valor de la etiqueta auxiliar que usamos para el cambio de dirección de los enemigos, ***LD A, (swEnemies)***, *y preservamos el valor cargándolo en C,* **LD C, A**.

```
changeEnemies_loop:
bit     $07, (hl)
jr      z, changeEnemies_endLoop

inc     hl
ld      a, (hl)
and     $3f
or      c
ld      (hl), a

dec     hl
ld      a, c
add     a, $40
ld      c, a
```

Comprobamos si el enemigo está activo, ***BIT $07, (HL)***, y saltamos si no lo está.

La dirección del enemigo está en los bits seis y siete del segundo byte de la configuración, por lo que apuntamos HL a este segundo byte, ***INC HL***, cagamos el valor en A, ***LD A, (HL)***, desechamos

la dirección actual del enemigo, ***AND $3F***, agregamos la nueva dirección, ***OR C***, y la actualizamos en memoria, ***LD (HL), A***.

Apuntamos HL de nuevo al primer byte de la configuración, ***DEC HL***, cargamos la nueva dirección en A, ***LD A, C***, sumamos uno a la nueva dirección ($40 = **01**00 0000), ***ADD A, $40***, y cargamos el valor en C, ***LD C, A***.

```
changeEnemies_endLoop:
inc     hl
inc     hl
djnz    changeEnemies_loop
```

Apuntamos HL al primer byte del siguiente enemigo, ***INC HL***, ***INC HL***, y repetimos hasta que B valga cero y hayamos recorrido los veinte enemigos, ***DJNZ chengeEnemies_loop***.

```
changeEnemies_end:
ld      a, c
ld      (swEnemies), a


ret
```

Cargamos la nueva dirección en A, ***LD A, C***, actualizamos el valor en memoria para la próxima vez que haya que cambiar la dirección, ***LD (swEnemies), A***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Cambia la dirección de los enemigos.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------
ChangeEnemies:
ld      hl, flags           ; Carga la dirección de memoria de flags en HL
bit     $03, (hl)           ; Comprueba si el bit 3 (cambio dirección) está activo
ret     z                   ; Si no es así, sale
res     $03, (hl)           ; Desactiva el bit 3 de flags

ld      b, $14              ; Carga en B el número total de enemigos (20)
ld      hl, enemiesConfig   ; Cara en HL la dirección de la configuración
                            ; de los enemigos
ld      a, (swEnemies)      ; Carga en A el auxiliar para cambiar la dirección
ld      c, a                ; Preserva el valor en C
changeEnemies_loop:
bit     $07, (hl)           ; Comprueba si el enemigo está activo
```

```
jr      z, changeEnemies_endLoop  ; Si no lo está, salta a final del bucle

inc     hl                  ; Apunta HL al segundo byte de la configuración
ld      a, (hl)             ; Carga el valor en HL
and     $3f                 ; Desecha la dirección
or      c                   ; Agrega la nueva dirección
ld      (hl), a             ; Actualiza la dirección en memoria

dec     hl                  ; Apunta HL al primer byte de la configuración
ld      a, c                ; Recupera la nueva dirección
add     a, $40              ; Le suma uno a la dirección ($40 = 0100 0000)
ld      c, a                ; Preserva la nueva dirección en C

changeEnemies_endLoop:
inc     hl                  ; Apunta HL al primer byte de la configuración
inc     hl                  ; del siguiente enemigo
djnz    changeEnemies_loop  ; Hasta que B sea cero (20 enemigos)

changeEnemies_end:
ld      a, c                ; Recupera la nueva dirección
ld      (swEnemies), a      ; La actualiza en memoria

ret
```

A esta nueva rutina hay que llamarla desde el bucle principal del programa. Volvemos al archivo Main.asm, localizamos la etiqueta ***Main_loop***, localizamos la línea ***CALL MoveShip***, y justo debajo de ella y antes de ***CALL MoveEnemies***, añadimos la siguiente línea:

```
call    ChangeEnemies
```

Si queréis, podéis compilar, cargar en el emulador y ver que todo sigue funcionando igual, no hemos roto nada, pero tampoco se produce el cambio de dirección, debido a que en ningún momento estamos activando el bit tres de flags.

Abrimos el archivo Int.asm para implementar la activación de este bit cada cuatro segundos (en sistemas PAL), dando uso así a las interrupciones.

Lo primero que vamos a hacer es añadir una constante al inicio del archivo, justo por debajo de ***ORG $7E5C***:

```
T1:     EQU $c8
```

A ***T1*** le asignamos un valor de $C8, doscientos en decimal, que resulta de multiplicar cincuenta, que son las interrupciones que tenemos por segundo en sistemas PAL, por cuatro segundos.

Al final del archivo vamos a añadir una etiqueta que vamos a usar para llevar el conteo de las interrupciones hasta que lleguen a doscientas, cuatro segundos.

```
countT1:    db $00
```

Y ahora vamos a modificar la rutina de la interrupción. Localizamos la etiqueta ***Isr_end***, y justo por encima de ella implementamos la parte con la que se controlan los cuatros segundos de los que venimos hablando.

```
Isr_T1:
ld      a, (countT1)
inc     a
ld      (countT1), a
sub     T1
jr      nz, Isr_end
ld      (countT1), a
set     $03, (hl)
```

Cargamos el valor del contador en A, ***LD A, (countT1)***, incrementamos A, ***INC A***, y actualizamos el valor del contador, ***LD (countT1), A***.

Restamos el número de interrupciones que hay que alcanzar para activar el flag de cambio de dirección, ***SUB T1***, y saltamos si no se ha alcanzado, ***JR NZ, Isr_end***.

Si se han alcanzado los cuatros segundos (doscientas interrupciones), el resultado de la resta anterior es cero, y con ese valor actualizamos el contador, ***LD (countT1), A***, y por último activamos el bit de cambio de dirección, ***SET $03, (HL)***.

Hay que cambiar otra línea. Tres líneas por encima de ***Isr_T1***, encontramos la línea ***JR NZ, Isr_end***. Esta línea hay que cambiarla dejándola de la siguiente manera:

```
jr      nz, Isr_T1
```

Ahora sí, compilamos, cargamos en el emulador y comprobamos que, cada cuatro segundos, los enemigos cambian de dirección. De igual manera vemos que lo de irse a la derecha y disparar ya no da tan buen resultado, en el primer nivel quizá sí, pero en los siguientes, no.

Ahora obligamos al jugador a moverse por la pantalla, pero la velocidad a la que se mueven los enemigos no permite ver bien hacia dónde van, por lo que deberíamos bajar dicha velocidad. Volvemos a Int.asm, localizamos la parte en la que se activa el bit para mover los enemigos:

```
ld      a, (countEnemy)
inc     a
ld      (countEnemy), a
sub     $02
jr      nz, Isr_T1
ld      (countEnemy), a
set     $02, (hl)
```

Y cambiamos ***SUB $02*** por ***SUB $03***.

Compilamos, cargamos en el emulador y comprobamos que ahora es más llevadero. Ajustad la velocidad como más os guste.

Todavía tenemos que trabajar más en el comportamiento de lo enemigos, pero ahora vamos a añadir un efecto de color.

Cambio de color

Como comenté anteriormente en este capítulo, vamos a añadir un efecto de color al movimiento de los enemigos, para ello hemos añadido la etiqueta ***enemiesColor*** en Var.asm.

El efecto va a consistir en cambiar el color de los enemigos desde uno (azul) a siete (blanco) cada vez que se muevan.

Vamos al archivo Print.asm y localizamos la etiqueta ***PrintEnemies***, y justo debajo de ella vamos a añadir la implementación del efecto de color.

Lo primero es cambiar la primera línea de la rutia, ***LD A, $06***.

```
ld      a, (enemiesColor)           ; Carga en A la tinta
```

De esta manera, el color en el que se pintan los enemigos lo tomamos de la nueva etiqueta.

La primera vez que pintamos los enemigos en un nivel, lo hacemos en amarillo. Vamos al archivo Game.asm, localizamos la etiqueta ***ChangeLevel***, y al inicio de la misma añadimos estas dos líneas:

```
ld      a, $06                  ; Carga el color amarillo en A
ld      (enemiesColor), a       ; Actualiza el color en memoria
```

Si ahora compilamos y cargamos en el emulador, debemos seguir viendo como los enemigos se pintan en amarillo.

Seguimos en el archivo Game.asm y localizamos la etiqueta ***MoveEnemies***, cuyo aspecto inicial es el siguiente:

```
MoveEnemies:
ld      hl, flags           ; Cargamos la dirección de memoria de flags en HL
bit     $02, (hl)           ; Comprueba si el bit 2 está activo
ret     z                   ; Si no es así, sale
res     $02, (hl)           ; Desactiva el bit 2 de flags

ld      d, $14              ; Carga en D el número total de enemigos (20)
ld      hl, enemiesConfig   ; Carga en HL la dirección de la configuración
                            ; de los enemigos
moveEnemies_loop:
```

La implementación del cambio de color la vamos a realizar justo después de la línea ***RES $02, (HL)***.

```
ld      a, (enemiesColor)
inc     a
cp      $08
jr      c, moveEnemies_cont
ld      a, $01

moveEnemies_cont:
ld      (enemiesColor), a
```

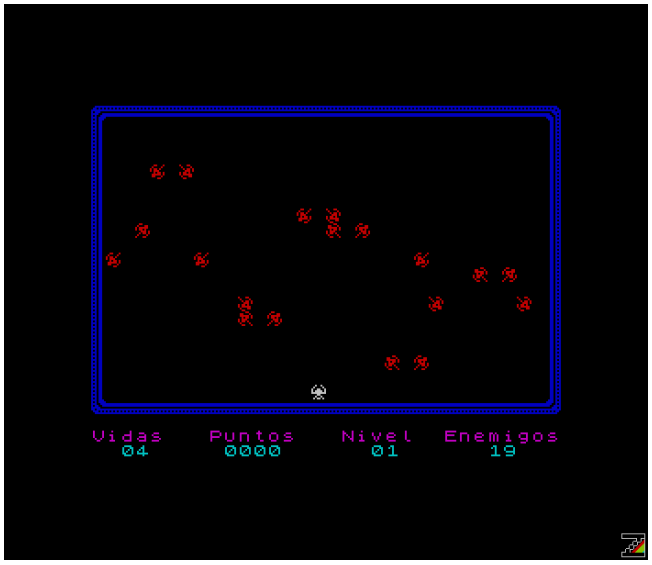
Cargamos en A el color de los enemigos, ***LD A, (enemiesColor)***, lo incrementamos, ***INC A***, comprobamos si hemos llegado a ocho, ***CP $08***, y saltamos si no lo hemos hecho, ***JR C, moveEnemies_cont***. Si no hemos saltado, hemos llegado a ocho y ponemos el color en azul, ***LD A, $01***. Por último, actualizamos el color en memoria, ***LD (enemiesColor), A***.

El aspecto del inicio de la rutina queda de la siguiente manera:

```
MoveEnemies:
ld      hl, flags           ; Carga la dirección de memoria de flags en HL
```

```
bit     $02, (hl)           ; Comprueba si el bit 2 está activo
ret     z                   ; Si no es así, sale
res     $02, (hl)           ; Desactiva el bit 2 de flags

ld      a, (enemiesColor)   ; Carga el color de los enemigos en A
inc     a                   ; Lo incrementa
cp      $08                 ; Comprueba si ha llegado a 8
jr      c, moveEnemies_cont ; Si no ha llegado, salta
ld      a, $01              ; Pone el color en azul

moveEnemies_cont:
ld      (enemiesColor), a   ; Actualiza el color en memoria
ld      d, $14              ; Carga en D el número total de enemigos (20)
ld      hl, enemiesConfig   ; Carga en HL la dirección de la configuración
                            ; de los enemigos
moveEnemies_loop:
```

Compilamos y cargamos en el emulador. Ahora sí podemos ver como los enemigos cambian de color.



Hasta ahora hemos ido cambiando el comportamiento de los enemigos, primero para que situarnos en una parte de la pantalla no nos permita superar los treinta niveles sin más, y segundo para dar un poco más de vistosidad.

Ha llegado el momento de abarcar el cambio más importante, vamos a dotar a nuestros enemigos de disparo.

Disparos enemigos

El disparo de los enemigos se va a activar cuando estén encima de nosotros y va a haber un máximo de cinco disparos activos a un mismo tiempo.

El primer paso es declarar las constantes que vamos a necesitar, abrimos Const.asm y localizamos la etiqueta ***WHITE_GRAPH*** que actualmente es una directiva ***EQU*** con el valor $9e, código de carácter que se corresponde con el gráfico que hemos definido para el carácter en blanco, lo cual es innecesario ya que el carácter en blanco ya está definido, es el carácter $20 (32). Dejamos la línea de la siguiente manera:

```
WHITE_GRAPH:EQU $20
```

Ahora localizamos la etiqueta ENEMY_TOP_R y justo debajo vamos a añadir las constantes para el número total de enemigos, el código de carácter para el disparo del enemigo y el número de disparos que puede haber activos a un mismo tiempo.

```
ENEMIES:        EQU $14
ENEMY_GRA_F:    EQU $9e
FIRES:          EQU $05
```

Como podemos observar, el código de carácter $9e va a ser ahora el disparo de los enemigos.

Abrimos el archivo Var.asm, localizamos la etiqueta ***udgsCommon*** y nos vamos a la última línea:

```
db $00, $00, $00, $00, $00, $00, $00, $00     ; $9e Blanco
```

Modificamos esta línea y la dejamos como sigue:

```
db $00, $3c, $2c, $2c, $2c, $2c, $18, $00     ; $9e Disparo enemigo
```

Si hicisteis las prácticas del capítulo 0x01 Definición de gráficos, deberíais ser capaces que dibujar en papel o en las plantillas que se proporcionaron las representación del disparo enemigo, solo tenéis que hacer la conversión de hexadecimal a binario.

Justo encima de la etiqueta ***udgsCommon*** vamos a añadir etiquetas para la configuración de los disparos enemigos, y para llevar la cuenta de cuántos de ellos están activos.

```
; ------------------------------------------------------------------------------
; Configuración de los disparos de los enemigos
;
; 2 bytes por disparo.
; ------------------------------------------------------------------------------
; Byte 1                        | Byte 2
; ------------------------------------------------------------------------------
; Bit 0-4:  Posición Y          | Bit 0-4:  Posición X
; Bit 5:    Libre               | Bit 5:    Libre
; Bit 6:    Libre               | Bit 6:    Libre
; Bit 7:    Activo 1/0          | Bit 7:    Libre
; ------------------------------------------------------------------------------
enemiesFire:
defs FIRES * $02
```

```
enemiesFireCount:
db $00
```

Con ***DEFS*** reservamos tantos bytes como sea el resultado de multiplicar ***FIRES*** (número máximo de disparos enemigos a un mismo tiempo) por dos (dos bytes por disparo). Como podéis ver, la configuración de los disparos enemigos guardan cierta similitud con la configuración de los enemigos.

Abrimos el archivo Game.asm y vamos a empezar con la implementación necesaria para que nuestros enemigos disparen y nos pongan las cosas más difíciles.

Vamos a implementar una rutina que desactive todos los disparos enemigos, que llamaremos cada vez que iniciemos un nuevo nivel. Esta rutina la vamos a poner justo antes de la rutina ***Sleep***.

```
ResetEnemiesFire:
ld      hl, enemiesFire
ld      de, enemiesFire + $01
ld      bc, FIRES * 02
ld      (hl), $00
ldir


ret
```

Apuntamos HL al primer byte de la configuración de los disparos enemigos, ***LD HL, enemiesFire***, apuntamos DE al byte siguiente, ***LD DE, enemiesFire + $01***, cargamos en BC el número de bytes que vamos a limpiar, ***LD BC, FIRES * 02***, limpiamos el primer byte, ***LD (HL), $02***, limpiamos el resto de bytes, ***LDIR***, y salimos, ***RET***.

En realidad también estamos limpiando (poniendo a cero) el contador de disparos activos, lo cual no nos va a suponer ningún problema. No obstante, si lo queréis evitar podéis añadir ***DEC BC*** antes de ***LDIR***.

El aspecto de la rutina, una vez comentada, es el siguiente:

```
; ------------------------------------------------------------------------
; Inicializa la configuración de los disparos enemigos
;
; Altera el valor de los registros BC, DE y HL.
; ------------------------------------------------------------------------
ResetEnemiesFire:
ld      hl, enemiesFire         ; Apunta HL a la configuración de los disparos
ld      de, enemiesFire + $01   ; Apunta DE al byte siguiente
ld      bc, FIRES * 02          ; Carga en BC en número de bytes a limpiar
ld      (hl), $00               ; Limpia el primer byte
ldir                            ; Limpia el resto
```

```
ret
```

La configuración de los disparos enemigos es una suerte de lista. Necesitamos una rutina para que actualice dicha lista, vea cuantos disparos están activos, los ponga al inicio de la lista y actualice en memoria el número de disparos activos. Esta rutina la vamos a implementar justo delante de ***ResetEnemiesFire***.

Es poco probable que queramos tener más de cinco disparos enemigos a la vez, es posible que incluso tengamos menos. Basados en esto, vamos a hacer una rutina que no es la más optima, pero que funciona.

La rutina que vamos a implementar, por cada elemento de la lista va a recorrer la lista en su totalidad, por lo que vamos a usar dos bucles anidados. Para finalizar, vamos a implementar un tercer bucle para actualizar el número de disparos activos.

```
RefreshEnemiesFire:
ld      b, FIRES
xor     a
refreshEnemiesFire_loopExt:
push    bc
ld      ix, enemiesFire
ld      b, FIRES
```

Cargamos en B el número máximo de disparos para que sea el contador del bucle exterior, ***LD B, FIRES***, y ponemos A a cero, ***XOR A***. Preservamos BC, ***PUSH BC***, apuntamos IX a la configuración de los disparos de los enemigos, ***LD IX, enemiesFire***, y volvemos a cargar en B el número máximo de disparos, en este caso como contador del bucle interior, ***LD B, FIRES***.

En este caso, a la memoria vamos a acceder de manera indexada con el registro IX; en este capítulo de Porompompong se habla sobre los registros del Z80.

```
refreshEnemiesFire_loopInt:
bit     $07, (ix+$00)
jr      nz, refreshEnemiesFire_loopIntCont
ld      c, (ix+$02)
ld      (ix+$00), c
ld      c, (ix+$03)
ld      (ix+$01), c
ld      (ix+$02), a
```

Evaluamos si el disparo está activo, ***BIT $07, (IX+$00)***, y saltamos si lo está, ***JR NZ, refreshEnemiesFire_loopIntCont***.

Si no está activo cargamos el primer byte del siguiente disparo en C, ***LD C, (IX+$02)***, y luego lo cargamos en el primer byte del disparo apuntado por IX, ***LD (IX+$00), C***. Cargamos el segundo

byte del siguiente disparo en C, ***LD C, (IX+03)***, y luego lo cargamos en el segundo byte del disparo apuntado por IX, ***LD (IX+$01), C***.

Por último, ponemos el primer byte del segundo disparo a cero, ***LD (IX+$02), A***.

```
refreshEnemiesFire_loopIntCont:
inc     ix
inc     ix
djnz    refreshEnemiesFire_loopInt


pop     bc
djnz    refreshEnemiesFire_loopExt
```

Apuntamos IX al primer byte del siguiente disparo, ***INC IX, INC IX***, y seguimos en bucle hasta que B sea igual a cero, ***DJNZ refreshEnemiesFire_loopInt***.

Recuperamos BC para obtener el contador del bucle exterior, ***PUSH BC***, y seguimos en bucle hasta que B sea igual a cero, ***DJNZ refreshEnemiesFire_loopExt***.

Llegados a este punto ya tenemos todos los disparos activos al inicio de la lista, solo queda contar cuantos disparos están activos. El conteo de los disparos activos lo vamos a llevar en A, recordad que ya lo pusimos a cero al inicio de la rutina, ***XOR A***.

```
ld      b, FIRES
ld      hl, enemiesFire
refreshEnemiesFire_loopCount:
bit     $07, (hl)
jr      z, refreshEnemiesFire_end
inc     a
refreshEnemiesFire_loopCountCont:
inc     hl
inc     hl
djnz    refreshEnemiesFire_loopCount


refreshEnemiesFire_end:
ld      (enemiesFireCount), a


ret
```

Cargamos en B el número máximo de disparos, ***LD B, FIRES***, y apuntamos HL a la configuración de los disparos, ***LD HL, enemiesFire***.

Evaluamos si el disparo está activo, ***BIT $07, (HL)***, y si no lo está salta, ***JR Z, refreshEnemiesFire_end***.

Si está activo, incrementamos A para sumar un disparo activo, ***INC A***, apuntamos HL al primer byte del siguiente disparo, ***INC HL***, ***INC HL***, y seguimos en bucle hasta que B sea igual a cero, ***DJNZ refreshEnemiesFire_loopCount***.

Por último, actualizamos el número de disparos activos en memoria, ***LD (enemiesFireCount), A***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Actualiza la configuración de los disparos enemigos
;
; Altera el valor de los registros AD, BC, HL e IX.
; ------------------------------------------------------------------------------
RefreshEnemiesFire:
ld      b, FIRES                    ; Carga en B el número máximo de disparos
xor     a                           ; Pone A a 0
refreshEnemiesFire_loopExt:
push    bc                          ; Preserva BC
ld      ix, enemiesFire             ; Apunta IX a la configuración de los disparos
ld      b, FIRES                    ; Carga en B el número máximo de disparos
refreshEnemiesFire_loopInt:
bit     $07, (ix+$00)               ; Evalúa si el disparo está activo
jr      nz, refreshEnemiesFire_loopIntCont  ; Si lo está, salta
ld      c, (ix+$02)                 ; Carga el byte 1 del siguiente disparo en C
ld      (ix+$00), c                 ; Lo carga en el byte 1 del disparo actual
ld      c, (ix+$03)                 ; Carga el byte 2 del siguiente disparo en C
ld      (ix+$01), c                 ; Lo carga en el byte 2 del disparo actual
ld      (ix+$02), a                 ; Pone a cero el byte 1 del siguiente disparo
refreshEnemiesFire_loopIntCont:
inc     ix
inc     ix                          ; Apunta IX al byte 1 del siguiente disparo
djnz    refreshEnemiesFire_loopInt  ; Bucle hasta que B = 0

pop     bc                          ; Recupera BC para bucle exterior
djnz    refreshEnemiesFire_loopExt  ; Bucle hasta que B = 0

; Actualiza el número de disparos activos
ld      b, FIRES                    ; Carga en B el número máximo de disparos
ld      hl, enemiesFire             ; Apunta HL a la configuración de los disparos
refreshEnemiesFire_loopCount:
```

```
bit     $07, (hl)                   ; Evalúa si el disparo está activo
jr      z, refreshEnemiesFire_end   ; Si no lo está, salta
inc     a                           ; Incrementa A = contador de disparos
refreshEnemiesFire_loopCountCont:
inc     hl
inc     hl                          ; Apunta HL al byte 1 del siguiente disparo
djnz    refreshEnemiesFire_loopCount    ; Bucle hasta que B = 0

refreshEnemiesFire_end:
ld      (enemiesFireCount), a       ; Actualiza el contador de disparos en memoria

ret
```

Ahora vamos a implementar la rutina que va a ir activando los disparos. Los disparos se van a activar cuando el enemigo esté en la misma coordenada horizontal que la nave, siempre que no estén ya activos todos los disparos.

Seguimos en Game.asm, localizamos la etiqueta ***MoveEnemies***, y justo encima de ella implementamos la rutina que activa los disparos enemigos.

```
EnableEnemiesFire:
ld      de, (shipPos)
ld      hl, enemiesConfig
ld      b, ENEMIES
```

Cargamos en DE la posición de la nave, ***LD DE, (shipPos)***, apuntamos HL a la configuración de los enemigos, ***LD HL, enemiesConfig***, y cargamos en B el número máximo de enemigos, ***LD B, ENEMIES***.

```
enableEnemiesFire_loop:
ld      a, (enemiesFireCount)
cp      FIRES
ret     nc

push    bc
ld      a, (hl)
ld      b, a
inc     hl
and     $80
jr      z, enableEnemiesFire_loopCont

ld      a, (hl)
```

```
and     $1f
cp      e
jr      nz, enableEnemiesFire_loopCont
```

Cargamos en A el número de disparos activos, ***LD A, (enemiesFireCount)***, lo comparamos con el número máximo de disparos, ***CP FIRES***, y salimos si lo hemos alcanzado, ***RET NC***. Recordad que ***CP*** resta al registro A el valor especificado, sin cambiar el valor de A pero si el registro F. El flag de acarreo se activa mientras A sea menor que el valor indicado en ***CP***, por lo tanto el flag de acarreo se desactiva cuando A sea mayor o igual al valor indicado en ***CP***.

Preservamos el valor de BC, ***PUSH BC***, cargamos en A el primer byte de la configuración del enemigo, ***LD A, (HL)***, lo cargamos en B, **LD B, *A***, apuntamos HL al segundo byte de la configuración***, INC HL***, comprobamos si el enemigo está activo, ***AND $80***, y saltamos si no lo está, ***JR Z, enableEnemiesFire_loopCont***.

En el caso de que el enemigo esté activo, cargamos en A el segundo byte de la configuración, ***LD A, (HL)***, nos quedamos con la coordenada X, ***AND $1F***, los comparamos con la coordenada X de la nave, ***CP E***, y saltamos si no son la misa, ***JR NZ, enableEnemiesFire_loopCont***.

Si no hemos saltado, tenemos que activar el disparo.

```
ld      c, a
push    hl
push    bc
ld      hl, enemiesFire
ld      a, (enemiesFireCount)
add     a, a
ld      b, $00
ld      c, a
add     hl, bc
pop     bc
ld      (hl), b
inc     hl
ld      (hl), c
ld      hl, enemiesFireCount
inc     (hl)
pop     hl
```

Cargamos en C la coordenada X del enemigo, ***LD C, A***, con esto ya tenemos lista la configuración del disparo. Preservamos HL, ***PUSH HL***, preservamos BC, ***PUSH BC***, apuntamos HL a la configuración de los disparos enemigos, ***LD HL, enemiesFire***, cargamos en A el número de disparos activos, ***LD A, (enemiesFireCount)***, lo multiplicamos por dos, ***ADD A, A***, ponemos B a cero, ***LD B, $00***, cargamos en C el número de bytes que hay que desplazarse, ***LD C, A***, y se lo sumamos a HL para que apunte a la posición de la lista dónde vamos a poner la configuración del disparo, ***ADD HL, BC***.

Recuperamos la configuración del disparo, ***POP BC***, cargamos el primer byte de la configuración del disparo en memoria, ***LD (HL), B***, apuntamos HL al segundo byte de la lista, ***INC HL***, cargamos el segundo byte de la configuración del disparo en memoria, ***LD (HL), C***, apuntamos HL al contador de disparos enemigos, ***LD HL, enemiesFireCount***, lo incrementamos, ***INC (HL)***, y recuperamos HL para que apunte al segundo byte de la configuración del enemigo, ***POP HL***.

```
enableEnemiesFire_loopCont:
pop     bc
inc     hl
djnz    enableEnemiesFire_loop


ret
```

Recuperamos el valor de BC para recuperar el contador del bucle, ***POP BC***, apuntamos HL al primer byte de la configuración del siguiente enemigo, ***INC HL***, y seguimos en bucle hasta que recorramos todos los enemigos, B sea igual a cero, ***DJNZ enebleEnemiesFire_loop***. Por último, salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Habilita el disparo del enemigo.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
EnableEnemiesFire:
ld      de, (shipPos)           ; Carga en DE la posición de la nave
ld      hl, enemiesConfig       ; Apunta HL a la configuración de los enemigos
ld      b, ENEMIES              ; Carga en B el número total de enemigos

enableEnemiesFire_loop:
ld      a, (enemiesFireCount)   ; Carga en A el número de disparos activos
cp      FIRES                   ; Lo compara con el máximo de disparos
ret     nc                      ; Sale si se ha alcanzado

push    bc                      ; Preserva el valor de BC
ld      a, (hl)                 ; Carga en A el primer byte de la configuración
ld      b, a                    ; Lo carga en B
inc     hl                      ; Apunta HL al segundo byte de la configuración
and     $80                     ; Evalúa si el enemigo está activo
jr      z, enableEnemiesFire_loopCont   ; Si no lo está, salta
```

```
ld      a, (hl)                 ; Carga en A el segundo byte de la configuración
and     $1f                     ; Se queda con la coordenada X
cp      e                       ; La compara con la de la nave
jr      nz, enableEnemiesFire_loopCont  ; Si no son la misma, salta

; Activa el disparo
; La configuración del disparo es la del enemigo
ld      c, a                    ; Carga en C la coordenada X del enemigo
push    hl                      ; Preserva HL
push    bc                      ; Preserva BC, configuración del disparo
ld      hl, enemiesFire         ; Apunta HL a los disparos de los enemigos
ld      a, (enemiesFireCount)   ; Carga en A el contador de disparos
add     a, a                    ; Lo multiplica por dos, dos bytes por disparo
ld      b, $00
ld      c, a                    ; Carga el desplazamiento en BC
add     hl, bc                  ; Apunta HL al disparo que hay que activar
pop     bc                      ; Recupera BC, configuración del disparo
ld      (hl), b                 ; Carga en memoria el primer byte de la configuración
inc     hl                      ; Apunta HL al segundo byte de la configuración
ld      (hl), c                 ; Lo carga en memoria
ld      hl, enemiesFireCount    ; Apunta HL al contador de disparos enemigos
inc     (hl)                    ; Lo incrementa en memoria
pop     hl                      ; Recupera HL, segundo byte configuración enemigo

enableEnemiesFire_loopCont:
pop     bc                      ; Recupera el valor de BC
inc     hl                      ; Apunta HL al primer byte de configuración
                                ; del enemigo siguiente
djnz    enableEnemiesFire_loop  ; Hasta que se recorra todos los enemigos, B = 0

ret
```

Como hemos dicho anteriormente, los disparos se habilitan cuando los enemigos están en la misma coordenada horizontal que la nave, por ese motivo la llamada a ***EnableEnemies*** la vamos a hacer desde la rutina ***MoveEnemies***.

En ***MoveEnemies*** vamos a cambiar dos cosas: primero localizamos la etiqueta ***moveEnemies_cont***, y la segunda línea, que ahora presenta este aspecto:

```
ld      d, $14              ; Carga en D el número total de enemigos (20)
```

La dejamos así:

```
ld      d, ENEMIES          ; Carga en D el número total de enemigos
```

Antes hemos declarado la constante ***ENEMIES***, y ahora hemos de sustituir la constante en todos aquellos lugares donde se carga el número de enemigos.

Ahora localizamos la etiqueta ***moveEnemies_end*** y tras la primera línea:

```
call    PrintEnemies        ; Pinta los enemigos
```

Añadimos la llamada a ***EnableEnemiesFire***:

```
call    EnableEnemiesFire   ; Habilita los disparos de los enemigos
```

Ahora vamos a implementar una rutina para que mueva los disparos enemigos, igual que tenemos una rutina para mover los enemigos.

Seguimos en Game.asm, localizamos la etiqueta ***MoveFire*** y justo delante de ella implementamos la rutina que va a mover los disparos enemigos.

```
MoveEnemiesFire:
ld      a, $03
call    Ink
ld      hl, flags
bit     $04, (hl)
ret     z
res     $04, (hl)
```

Cargamos en A la tinta magenta, ***LD A, $03***, cambiamos la tinta, ***CALL Ink***, cargamos en HL los flags, ***LD HL, flags***, y comprobamos si el bit cuatro está activo, ***BIT $04, (HL)***. Si el bit no está activo salimos, ***RET Z***, si sí lo está lo desactivamos, ***RES $04, (HL)***.

Por lo que vemos, vamos a usar otro bit de nuestro byte de flags.

```
ld      d, FIRES
ld      hl, enemiesFire
moveEnemiesFire_loop:
ld      b, (hl)
inc     hl
ld      c, (hl)
dec     hl
```

Cargamos en D el número máximo de disparos, ***LD D, FIRES***, apuntamos HL a la configuración de los disparos enemigos, ***LD HL, enemiesFire***, cargamos el primer byte de la configuración en B, ***LD B, (HL)***, apuntamos HL al segundo byte, ***INC HL***, lo cargamos en C, ***LD C, (HL)***, y volvemos a apuntar HL al primer byte, ***DEC HL***.

```
bit     $07, b
jr      z, moveEnemiesFire_loopCont
```

```
res     $07, b
call    DeleteChar
ld      a, ENEMY_TOP_B + $01
cp      b
jr      z, moveEnemiesFire_loopCont
dec     b
call    At
ld      a, ENEMY_GRA_F
rst     $10
set     $07, b
```

Evaluamos si el disparo está activo, ***BIT $07, B***, y saltamos si no lo está, ***JR Z, moveEnemiesFire_loopCont***. Si el disparo está inactivo podríamos salir de la rutina, pero no lo hacemos para que la rutina siempre tarde lo mismo en ejecutarse, o al menos lo más parecido posible entre cada ejecución.

Si el disparo está activo, nos quedamos con la coordenada Y, ***RES $07, B***, borramos el disparo de su posición actual, ***CALL DeleteChar***, cargamos en A el tope vertical al que puede llegar el disparo por abajo, ***LD A, ENEMY_TOP_B + $01***, lo comparamos con la coordenada Y, ***CP B***, y saltamos si la hemos alcanzado, ***JR Z, moveEnemiesFire_loopCont***.

Si no hemos alcanzado el tope, apuntamos la coordenada Y a la línea siguiente, ***DEC B***, posicionamos el cursor, ***CALL At***, cargamos en A el gráfico del disparo enemigo, ***LD A, ENEMY_GRA_F***, lo pintamos, ***RST $10***, y dejamos el disparo activado, ***SET $07, B***.

```
moveEnemiesFire_loopCont:
ld      (hl), b
inc     hl
inc     hl


dec     d
jr      nz, moveEnemiesFire_loop


jp      RefreshEnemiesFire
```

Al llegar a este punto, hemos activado o desactivado el disparo y actualizado la coordenada Y según corresponda. Actualizamos el primer byte de la configuración del disparo en memoria, ***LD (HL), D***, apuntamos HL al primer byte del disparo siguiente, ***INC HL***, ***INC HL***, decrementamos D que es donde tenemos el número de iteraciones del bucle, ***DEC D***, y seguimos en bucle hasta que D valga cero, ***JR NZ, moveEnemiesFire_loop***.

Finalmente, saltamos a refrescar la lista de disparos y salimos por allí, ***JP RefreshEnemiesFire***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
```

```
; Mueve el disparo del enemigo.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
MoveEnemiesFire:
ld      a, $03          ; Cara la tinta 3 en A
call    Ink             ; Cambia la tinta
ld      hl, flags       ; Apunta HL a los flags
bit     $04, (hl)       ; Comprueba si está activo el flag mover disparo enemigo
ret     z               ; Si no lo está, sale
res     $04, (hl)       ; Desactiva el flag mover disparo enemigo

ld      d, FIRES        ; Carga en D en número máximo de disparos
ld      hl, enemiesFire ; Apunta HL a los disparos enemigos
moveEnemiesFire_loop:
ld      b, (hl)         ; Carga en B la coordenada Y del disparo
inc     hl              ; Apunta HL a la coordenada X
ld      c, (hl)         ; La carga en C
dec     hl              ; Apunta HL a la coordenada Y

bit     $07, b          ; Evalúa si el disparo está activo
jr      z, moveEnemiesFire_loopCont ; Salta si no lo está
res     $07, b          ; Se queda con la coordenada Y
call    DeleteChar      ; Borra el disparo de su posición actual
ld      a, ENEMY_TOP_B + $01    ; Carga en A el tope
cp      b               ; Lo compara con la coordenada Y
jr      z, moveEnemiesFire_loopCont ; Si es la misma, salta
dec     b               ; Apunta B a la línea siguiente
call    At              ; Posiciona el cursor
ld      a, ENEMY_GRA_F  ; Carga en A el gráfico del disparo
rst     $10             ; Lo pinta
set     $07, b          ; Deja el disparo activo

moveEnemiesFire_loopCont:
ld      (hl), b         ; Actualiza la coordenada Y del disparo
inc     hl
inc     hl              ; Apunta HL al primer byte de la configuración
                        ; del siguiente enemigo
```

```
dec     d
jr      nz, moveEnemiesFire_loop

jp      RefreshEnemiesFire  ; Actualiza los disparos enemigos y sale
```

Ya tenemos todo casi listo para poder ver los disparos enemigos en pantalla.

Al inicio de la rutina ***MoveEnemies***, recuperamos el valor de la etiqueta ***flags*** y evaluamos si el bit cuatro está activo.

Vamos al archivo Main.asm, y en los comentarios de ***flags***, vamos a añadir el siguiente:

```
; Bit 4 -> mover disparo enemigo            0 = No, 1 = Sí
```

Seguimos en Main.asm y vamos a aprovechar para incluir las llamadas a algunas de las rutinas que hemos implementado.

Localizamos la etiqueta ***Main_start***, y justo delante de ***CALL ChangeLevel*** vamos a incluir una llamada a la inicialización de los disparos enemigos:

```
call    ResetEnemiesFire
```

Localizamos la rutina ***Main_loop***, y entre las líneas ***CALL MoveEnemies*** y ***CALL CheckCrashShip***, añadimos la llamada a la rutina que mueve los disparos enemigos:

```
call    MoveEnemiesFire
```

Aprovechamos también para comentar la línea ***CALL CheckCrashShip***, para que no nos maten los enemigos y poder ver mejor como quedan los disparos.

Por último, localizamos la rutina ***Main_restart***, y casi al final, justo antes de ***CALL Sleep***, añadimos otra llamada a la inicialización de lo disparos enemigos:

```
call    ResetEnemiesFire
```

Hemos acabado en Main.asm, pero todavía nos queda activar el bit cuatro de ***flags*** para que todo esto funcione.

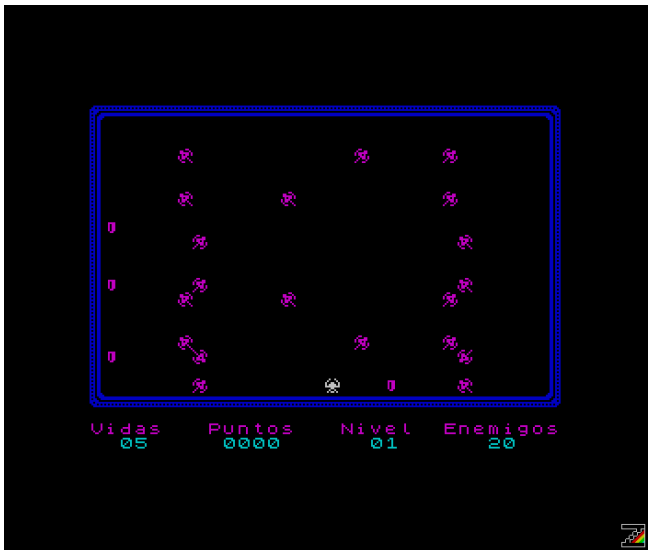
Vamos al archivo Int.asm, y lo primero que tenemos que hacer es decidir a que velocidad se va a mover el disparo enemigo. Sin implementar nada nuevo tenemos dos velocidades a elegir: a la velocidad a la que se mueve la nave (en cada interrupción), o la velocidad a la que se mueven los enemigos (cada N interrupciones).

En mi caso he optado por la segunda opción. Localizamos la línea ***SET $02, (HL)***, y justo debajo añadimos:

```
set     $04, (hl)
```

Si queréis que se mueva a la velocidad a la que se mueve la nave, esta línea la tenéis que poner justo debajo de ***SET $00, (HL)***.

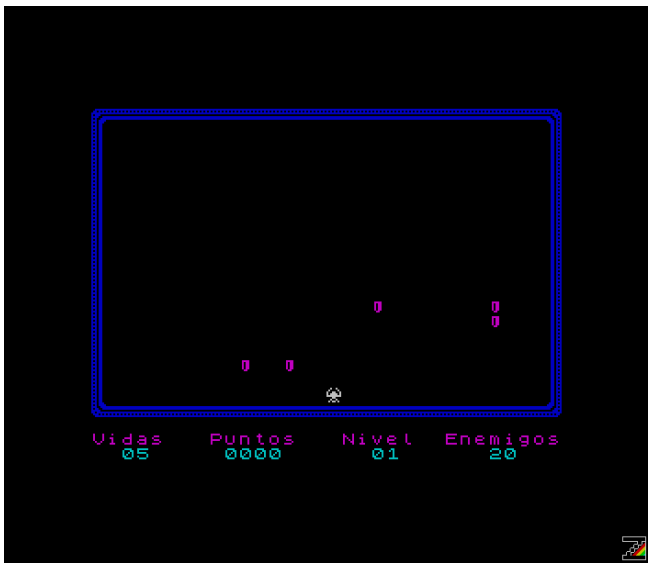
Hemos implementado una buena cantidad de líneas, y ya es hora de que probemos y veamos los resultados; compilamos y cargamos en el emulador.



Si todo va bien, ya pueden verse los disparos enemigos.

Dije anteriormente que cinco disparos enemigos simultáneos quizá fuera excesivo. Para apreciar mejor esto, en Game.asm localizamos la rutina ***MoveEnemies*** y, casi al final, comentamos la línea ***CALL PrintEnemies*** para poder ver mejor los disparos.

Compilamos, cargamos en el emulador y vemos los resultados.



Si a esto le sumamos los enemigos, quizá sea demasiado. Descomentamos la línea ***CALL PrintEnemies***, y en Main.asm, en la rutina ***MainLoop***, localizamos la línea ***CALL CheckCrashShip*** y quitamos el comentario.

Compilamos, cargamos en el emulador y verificamos que los enemigos nos vuelven a matar. Ya solo queda hacer que los disparos enemigos también nos maten.

Antes de implementar las colisiones entre la nave y los disparos enemigos, recordad que declaramos una constante con el número total de enemigos, ***ENEMIES***, pero todavía tenemos partes del código en las que no la estamos usando.

Vamos a Print.asm, localizamos la rutina ***PrintEnemies***, y modificamos la línea ***LD D, $14*** dejándola así:

```
ld      d, ENEMIES
```

El resto de modificaciones las vamos a realizar en Game.asm.

Localizamos la rutina ***ChangeEnemies***, localizamos la línea ***LD B, $14*** y la modificamos dejándola así:

```
ld      b, ENEMIES
```

Localizamos las rutina ***CheckCrashFire***, borramos las líneas:

```
ld      b, enemiesConfigEnd - enemiesConfigIni


sra     b
```

Y las sustituimos por:

```
ld      b, ENEMIES
```

Hacemos la misma modificación en la rutina ***CheckCrashShip***.

Y ahora ya sí, implementamos las colisiones entre la nave y los disparos enemigos. Seguimos en la rutina ***CheckCrashShip***, vamos al final y justo antes de ***RET***, añadimos las nuevas colisiones.

```
checkCrashShipFire:
ld      de, (shipPos)
ld      a, (enemiesFireCount)
ld      b, a
ld      hl, enemiesFire
```

Cargamos en DE la posición de la nave, ***LD DE, (shipPos)***, cargamos en A el número de disparos enemigos activados, ***LD A, (enemiesFireCount)***, lo cargamos en B, ***LD B, A***, y apuntamos HL a la configuración de los disparos, ***LD HL, enemiesFire***.

```
checkCrashShipFire_loop:
ld      a, (hl)
inc     hl
res     $07, a
cp      d
jr      nz, checkCrashShipFire_loopCont
ld      a, (hl)
cp      e
jr      nz, checkCrashShipFire_loopCont
```

Cargamos el primer byte de la configuración del disparo en A, ***LD A, (HL)***, apuntamos HL al segundo byte, ***INC HL***, nos quedamos con la coordenada Y, ***RES $07, A,*** la comparamos con la coordenada Y de la nave, ***CP D***, y saltamos si no son la misma, ***JR NZ, checkCrashShipFire_loopCont***.

Si la coordenada Y del disparo y la de la nave son la misma, cargamos en A la coordenada X del disparo, ***LD A, (HL)***, la comparamos con la coordenada X de la nave, ***CP E***, y saltamos si no son la misma, ***JR NZ, checkCrashShipFire_loopCont***.

```
dec     hl
res     $07, (hl)
ld      a, (livesCounter)
dec     a
daa
ld      (livesCounter), a
call    PrintInfoValue
call    PrintExplosion
jp      RefreshEnemiesFire
```

Si hay colisión entre el disparo y la nave, apuntamos HL al primer byte de la configuración del disparo, ***DEC HL***, desactivamos el disparo, ***RES $07, (HL)***, cargamos en A el número de vidas, ***LD A, (livesCounter)***, quitamos una vida, ***DEC A***, hacemos el ajuste decimal, ***DAA***, y actualizamos en memoria, ***LD (livesCounter), A***.

Por último, pintamos la información de la partida, ***CALL PrintInfoValue***, pintamos la explosión, ***CALL PrintExploxion***, y actualizamos la lista de disparos enemigos y salimos por allí, ***JP RefreshEnemiesFire***.

```
checkCrashShipFire_loopCont:
inc     hl
djnz    checkCrashShipFire_loop
```

Si no ha habido colisión, apuntamos HL al primer byte de la configuración del siguiente disparo, ***INC HL***, y seguimos en bucle hasta que hayamos recorrido todos los disparos y el valor de B sea cero, ***DJNZ checkCrashShipFire_loop***.

El aspecto final de la detección de colisiones entre la nave y los enemigos (naves y disparos enemigos) es el siguiente:

```
; ------------------------------------------------------------------------
; Evalúa las colisiones de los enemigos y los disparos con la nave.
;
; Altera el valor de lo registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
CheckCrashShip:
ld      de, (shipPos)       ; Carga en DE la posición de nave
```

```
ld      hl, enemiesConfig   ; Apunta HL a la configuración de los enemigos
ld      b, ENEMIES          ; Carga en B el número de enemigos
checkCrashShip_loop:
ld      a, (hl)             ; Carga en A la coordenada Y del enemigo
inc     hl                  ; Apunta HL a la coordenada X del enemigo
bit     $07, a              ; Evalúa si el enemigo está activo
jr      z, checkCrashShip_endLoop   ; Si no lo está, salta

and     $1f                 ; Se queda con la coordenada Y del enemigo
cp      d                   ; Compara con la coordenada Y de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

ld      a, (hl)             ; Carga en A la coordenada X del enemigo
and     $1f                 ; Se queda con la coordenada X de enemigo
cp      e                   ; Compara con la coordenada X de la nave
jr      nz, checkCrashShip_endLoop  ; Si no son iguales, salta

dec     hl                  ; Apunta HL a la coordenada Y del enemigo
res     $07, (hl)           ; Desactiva el enemigo

ld      a, (enemiesCounter) ; Carga en A el número de enemigos
dec     a                   ; Resta uno
daa                         ; Hace el ajuste decimal
ld      (enemiesCounter), a ; Actualiza el valor en memoria
ld      a, (livesCounter)   ; Carga las vidas en A
dec     a                   ; Quita una
daa                         ; Hace el ajuste decimal
ld      (livesCounter), a   ; Actualiza el valor en memoria
call    PrintInfoValue      ; Pinta la información de la partida

jp      PrintExplosion      ; Pinta la explosión y sale

checkCrashShip_endLoop:
inc     hl                  ; Apunta HL a la coordenada Y del siguiente enemigo
djnz    checkCrashShip_loop ; En bucle hasta que B = 0

checkCrashShipFire:
; Comprueba colisiones entre disparos enemigos y nave
```

```
ld      de, (shipPos)           ; Carga en DE la posición de la nave
ld      a, (enemiesFireCount)
ld      b, a                    ; Carga el B el número de disparos activos
ld      hl, enemiesFire         ; Apunta HL a la configuración de los disparos
checkCrashShipFire_loop:
ld      a, (hl)                 ; Carga la coordenada Y del disparo en A
inc     hl                      ; Apunta HL a la coordenada X
res     $07, a                  ; Se queda con la coordenada Y
cp      d                       ; Compara si es la misma que la de la nave
jr      nz, checkCrashShipFire_loopCont ; Si no es la misma, salta
ld      a, (hl)                 ; Carga la coordenada X del disparo en A
cp      e                       ; Compara si es la misma que la de la nave
jr      nz, checkCrashShipFire_loopCont ; Si no es la misma, salta
; Si llega aquí, la nave ha colisionado con el disparo
dec     hl                      ; Apunta HL al primer byte del disparo
res     $07, (hl)               ; Desactiva el disparo
ld      a, (livesCounter)       ; Carga las vidas en A
dec     a                       ; Quita una
daa                             ; Hace el ajuste decimal
ld      (livesCounter), a       ; Actualiza el valor en memoria
call    PrintInfoValue          ; Pinta la información de la partida
call    PrintExplosion          ; Pinta la explosión
jp      RefreshEnemiesFire      ; Actualiza los disparos enemigos y sale

checkCrashShipFire_loopCont:
inc     hl                      ; Apunta HL al siguiente disparo
djnz    checkCrashShipFire_loop ; Bucle hasta que B = 0

ret
```

Llegados a este punto, ya tenemos el disparo de las naves enemigas implementado. Compilamos, cargamos en el emulador y vemos los resultados.

Ajuste de la dificultad

Quizá ahora la dificultad haya subido demasiado, o quizá no. Sea como fuere, vamos a ver algunos pequeños cambios que podemos realizar para ajustar la dificultad.

Los cambios que os propongo es para que hagáis pruebas, pero no los dejéis de manera permanente pues más adelante vamos a añadir una opción en el menú para que el jugador pueda seleccionar entre distintos niveles de dificultad.

La primera manera de reducir la dificultad va a ser reduciendo la velocidad a la que se mueven los enemigos y sus disparos. Vamos al archivo Int.asm, localizamos la línea ***SUB $03*** y sustituimos $03 por $04, $05, $06, etc. Recordad que cuanto mayor sea este número, menor es la velocidad a la que se mueven los enemigos. Haced pruebas y veréis que la velocidad se reduce.

Otra forma de reducir la dificultad es reducir el número de disparos simultáneos. Vamos al archivo Const.asm, localizamos la etiqueta ***FIRES*** y cambiamos su valor por $01, $02, etc. Compilamos y vemos que al haber menos disparos a un mismo tiempo, la dificultad también se reduce.

También podemos reducir la dificultad evitando que los enemigos colisionen con la nave, para lo cual localizamos la etiqueta ***moveEnemies_Y_down*** y dos lineas más abajo tenemos ***SUB ENEMY_TOP_B***, modificamos está línea y la dejamos como sigue:

```
sub     ENEMY_TOP_B  + $01
```

Como podéis observar, los enemigos ya no colisionan con la nave, lo que reduce la dificultad. Si la reduce demasiado para vuestro gusto, probad a aumentar el número de disparos enemigos simultáneos.

Si los enemigos no colisionan con la nave, nos podríamos ahorra gran parte de la rutina ***CheckCrashShip***, pero como vamos a cambiar estos comportamientos en base a la selección del jugador, la dejamos como está.

Otra forma que se me ocurre de disminuir la dificultad es al estilo Plaga Galáctica, el primer juego que cargué en mi Amstrad CPC 464. En Plaga Galáctica se dispone de tres vidas para superar cada nivel.

Para comenzar cada nivel con cinco vidas, vamos a Main.asm, localizamos la etiqueta ***Main_restart***, y antes de ***CALL FadeScreen*** añadimos las líneas siguientes:

```
ld      hl, livesCounter
ld      (hl), $05
```

Con estas líneas, contaremos con cinco vidas al inicio de cada nivel.

Conclusión

En este capítulo hemos cambiado el comportamiento de los enemigos y los hemos dotado de disparo. También hemos visto distintas formas de ajustar la dificultad.

En el próximo capítulo vamos a implementar el sonido.

0x0C Sonido

En este capítulo vamos a implementar la música. A diferencia de la versión original de Batalla Espacial, en esta ocasión vamos a incluir música durante la partida, además de los efectos de sonido que ya teníamos.

Creamos la carpeta Paso12 y copiamos desde la carpeta Paso11 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm y Var.asm.

Probando, probando

Antes de implementar el sonido en nuestro juego, vamos a realizar una serie de pruebas para ver como lo hacemos.

Dentro de la carpeta Paso12 vamos a crear la carpeta Sound, dentro de la misma vamos a ir añadiendo los archivos de código de las pruebas que vamos a realizar.

Creamos, dentro de Sound, el archivo Const.asm para añadir las constantes que nos van a hacer falta, empezando por la dirección de memoria de la rutina ***BEEPER*** de la ROM, rutina que hace sonar las notas que enviemos.

```
; ------------------------------------------------------------------------------
; Rutina beeper de la ROM.
;
; Entrada:  HL -> Nota.
;           DE -> Duración.
;
; Altera el valor de los registros AF, BC, DE, HL e IX.
; ------------------------------------------------------------------------------
BEEP:   EQU $03b5
```

Podemos observar en los comentarios que esta rutina recibe en el registro HL la nota, y en DE la duración (frecuencia).

Lo siguiente que vamos a añadir al archivo son las notas que, al igual que las frecuencias, obtuve en Programandala, y podéis descargar desde aquí.

```
; ------------------------------------------------------------------------------
; Notas a cargar en HL
; ------------------------------------------------------------------------------
C_0:    EQU $6868
Cs_0:   EQU $628d
D_0:    EQU $5d03
Ds_0:   EQU $57bf
E_0:    EQU $52d7
```

```
F_0:    EQU $4e2b
Fs_0:   EQU $49cc
G_0:    EQU $45a3
Gs_0:   EQU $41b6
A_0:    EQU $3e06
As_0:   EQU $3a87
B_0:    EQU $373e
C_1:    EQU $3425
Cs_1:   EQU $3134
D_1:    EQU $2e6f
Ds_1:   EQU $2bd3
E_1:    EQU $295c
F_1:    EQU $2708
Fs_1:   EQU $24d5
G_1:    EQU $22c2
Gs_1:   EQU $20cd
A_1:    EQU $1ef4
As_1:   EQU $1d36
B_1:    EQU $1b90
C_2:    EQU $1a02
Cs_2:   EQU $188b
D_2:    EQU $1728
Ds_2:   EQU $15da
E_2:    EQU $149e
F_2:    EQU $1374
Fs_2:   EQU $125b
G_2:    EQU $1152
Gs_2:   EQU $1058
A_2:    EQU $0f6b
As_2:   EQU $0e9d
B_2:    EQU $0db8
C_3:    EQU $0cf2
Cs_3:   EQU $0c36
D_3:    EQU $0b86
Ds_3:   EQU $0add
E_3:    EQU $0a40
F_3:    EQU $09ab
Fs_3:   EQU $091e
```

```
G_3:    EQU $089a
Gs_3:   EQU $081c
A_3:    EQU $07a6
As_3:   EQU $0736
B_3:    EQU $06cd
C_4:    EQU $066a
Cs_4:   EQU $060c
D_4:    EQU $05b3
Ds_4:   EQU $0560
E_4:    EQU $0511
F_4:    EQU $04c6
Fs_4:   EQU $0480
G_4:    EQU $043d
Gs_4:   EQU $03ff
A_4:    EQU $03c4
As_4:   EQU $038c
B_4:    EQU $0357
C_5:    EQU $0325
Cs_5:   EQU $02f7
D_5:    EQU $02ca
Ds_5:   EQU $02a0
E_5:    EQU $0279
F_5:    EQU $0254
Fs_5:   EQU $0231
G_5:    EQU $020f
Gs_5:   EQU $01f0
A_5:    EQU $01d3
As_5:   EQU $01b7
B_5:    EQU $019c
C_6:    EQU $0183
Cs_6:   EQU $016c
D_6:    EQU $0156
Ds_6:   EQU $0141
E_6:    EQU $012d
F_6:    EQU $011b
Fs_6:   EQU $0109
G_6:    EQU $00f8
Gs_6:   EQU $00e9
```

```
A_6:    EQU $00da
As_6:   EQU $00cc
B_6:    EQU $00bf
C_7:    EQU $00b2
Cs_7:   EQU $00a7
D_7:    EQU $009c
Ds_7:   EQU $0091
E_7:    EQU $0087
F_7:    EQU $007e
Fs_7:   EQU $0075
G_7:    EQU $006d
Gs_7:   EQU $0065
A_7:    EQU $005e
As_7:   EQU $0057
B_7:    EQU $0050
C_8:    EQU $004a
Cs_8:   EQU $0044
D_8:    EQU $003e
Ds_8:   EQU $0039
E_8:    EQU $0034
F_8:    EQU $0030
Fs_8:   EQU $002b
G_8:    EQU $0027
Gs_8:   EQU $0023
A_8:    EQU $0020
As_8:   EQU $001c
B_8:    EQU $0019
```

La nomenclatura usada se basa en la notación anglosajona, siendo:

Do	C
Re	D
Mi	E
Fa	F
Sol	G
La	A
Si	B

La ese minúscula que sigue a algunas notas indica que es sostenida, el número es la escala. Si el número es menor, el sonido es más grave, por el contrario, si es mayor, el sonido es más agudo.

Lo siguiente a añadir son las frecuencias, para las que hemos usado la misma notación, pero añadiendo ***_f*** para distinguirlas de las notas. Las frecuencias aquí expuestas hacen que las notas suenen durante un segundo, por lo que si las dividimos entre dos sonarían durante medio segundo. Tened en cuenta que mientras emitimos un sonido nuestro programa se va a parar, es por eso que vamos a dividir las frecuencias entre treinta y dos.

```
; ------------------------------------------------------------------------------
; Frecuencias a cargar en DE, 1 segundo ( / 2 = 0.5 ....)
; ------------------------------------------------------------------------------
C_0_f:  EQU $0010 / $20
Cs_0_f: EQU $0011 / $20
D_0_f:  EQU $0012 / $20
Ds_0_f: EQU $0013 / $20
E_0_f:  EQU $0014 / $20
F_0_f:  EQU $0015 / $20
Fs_0_f: EQU $0017 / $20
G_0_f:  EQU $0018 / $20
Gs_0_f: EQU $0019 / $20
A_0_f:  EQU $001b / $20
As_0_f: EQU $001d / $20
B_0_f:  EQU $001e / $20
C_1_f:  EQU $0020 / $20
Cs_1_f: EQU $0022 / $20
D_1_f:  EQU $0024 / $20
Ds_1_f: EQU $0026 / $20
E_1_f:  EQU $0029 / $20
F_1_f:  EQU $002b / $20
Fs_1_f: EQU $002e / $20
G_1_f:  EQU $0031 / $20
Gs_1_f: EQU $0033 / $20
A_1_f:  EQU $0037 / $20
As_1_f: EQU $003a / $20
B_1_f:  EQU $003d / $20
C_2_f:  EQU $0041 / $20
Cs_2_f: EQU $0045 / $20
D_2_f:  EQU $0049 / $20
Ds_2_f: EQU $004d / $20
E_2_f:  EQU $0052 / $20
```

```
F_2_f:  EQU $0057 / $20
Fs_2_f: EQU $005c / $20
G_2_f:  EQU $0062 / $20
Gs_2_f: EQU $0067 / $20
A_2_f:  EQU $006e / $20
As_2_f: EQU $0074 / $20
B_2_f:  EQU $007b / $20
C_3_f:  EQU $0082 / $20
Cs_3_f: EQU $008a / $20
D_3_f:  EQU $0092 / $20
Ds_3_f: EQU $009b / $20
E_3_f:  EQU $00a4 / $20
F_3_f:  EQU $00ae / $20
Fs_3_f: EQU $00b9 / $20
G_3_f:  EQU $00c4 / $20
Gs_3_f: EQU $00cf / $20
A_3_f:  EQU $00dc / $20
As_3_f: EQU $00e9 / $20
B_3_f:  EQU $00f6 / $20
C_4_f:  EQU $0105 / $20
Cs_4_f: EQU $0115 / $20
D_4_f:  EQU $0125 / $20
Ds_4_f: EQU $0137 / $20
E_4_f:  EQU $0149 / $20
F_4_f:  EQU $015d / $20
Fs_4_f: EQU $0172 / $20
G_4_f:  EQU $0188 / $20
Gs_4_f: EQU $019f / $20
A_4_f:  EQU $01b8 / $20
As_4_f: EQU $01d2 / $20
B_4_f:  EQU $01ed / $20
C_5_f:  EQU $020b / $20
Cs_5_f: EQU $022a / $20
D_5_f:  EQU $024b / $20
Ds_5_f: EQU $026e / $20
E_5_f:  EQU $0293 / $20
F_5_f:  EQU $02ba / $20
Fs_5_f: EQU $02e4 / $20
```

```
G_5_f:  EQU $0310 / $20
Gs_5_f: EQU $033e / $20
A_5_f:  EQU $0370 / $20
As_5_f: EQU $03a4 / $20
B_5_f:  EQU $03db / $20
C_6_f:  EQU $0417 / $20
Cs_6_f: EQU $0455 / $20
D_6_f:  EQU $0497 / $20
Ds_6_f: EQU $04dd / $20
E_6_f:  EQU $0527 / $20
F_6_f:  EQU $0575 / $20
Fs_6_f: EQU $05c8 / $20
G_6_f:  EQU $0620 / $20
Gs_6_f: EQU $067d / $20
A_6_f:  EQU $06e0 / $20
As_6_f: EQU $0749 / $20
B_6_f:  EQU $07b8 / $20
C_7_f:  EQU $082d / $20
Cs_7_f: EQU $08a9 / $20
D_7_f:  EQU $092d / $20
Ds_7_f: EQU $09b9 / $20
E_7_f:  EQU $0a4d / $20
F_7_f:  EQU $0aea / $20
Fs_7_f: EQU $0b90 / $20
G_7_f:  EQU $0c40 / $20
Gs_7_f: EQU $0cfa / $20
A_7_f:  EQU $0dc0 / $20
As_7_f: EQU $0e91 / $20
B_7_f:  EQU $0f6f / $20
C_8_f:  EQU $105a / $20
Cs_8_f: EQU $1153 / $20
D_8_f:  EQU $125b / $20
Ds_8_f: EQU $1372 / $20
E_8_f:  EQU $149a / $20
F_8_f:  EQU $15d4 / $20
Fs_8_f: EQU $1720 / $20
G_8_f:  EQU $1880 / $20
Gs_8_f: EQU $19f5 / $20
```

```
A_8_f:  EQU $1b80 / $20
As_8_f: EQU $1d23 / $20
B_8_f:  EQU $1ede / $20
```

Hemos añadido muchas constantes, pero recordad que ***EQU*** no se compila, por lo que no ocupa nada. Lo que hace el compilador es sustituir la etiqueta ***EQU*** por el valor que representa allí dónde aparezca.

Ahora que tenemos definidas las notas y sus frecuencias, vamos a definir las canciones, que en nuestro caso van a ser dos. En esta primera prueba no vais a reconocerlas, pero más adelante seguro que sí.

Para definir las canciones vamos a usar las etiquetas que acabamos de añadir en el archivo Const.asm. Creamos el archivo Var.asm (recordad que estamos dentro de la carpeta Sound) y vamos a agregar al mismo la primera canción.

```
Song_1:
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2
dw  D_3_f, D_3, D_3_f, D_3, D_3_f, D_3, Ds_3_f, Ds_3, As_2_f, As_2, Fs_2_f, Fs_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2


dw  G_3_f, G_3, G_2_f, G_2, G_2_f, G_2, G_3_f, G_3, Fs_3_f, Fs_3, F_3_f, F_3, E_3_f, E_3, Ds_3_f, Ds_3, E_3_f, E_3
dw  Gs_2_f, Gs_2, Cs_3_f, Cs_3, C_3_f, C_3, B_2_f, B_2, As_2_f, As_2, A_2_f, A_2, As_2_f, As_2
dw  Ds_2_f, Ds_2, Fs_2_f, Fs_2, Ds_2_f, Ds_2, Fs_2_f, Fs_2, As_2_f, As_2, G_2_f, G_2, As_2_f, As_2, D_3_f, D_3


dw  G_3_f, G_3, G_2_f, G_2, G_2_f, G_2, G_3_f, G_3, Fs_3_f, Fs_3, F_3_f, F_3, E_3_f, E_3, Ds_3_f, Ds_3, E_3_f, E_3
dw  Gs_2_f, Gs_2, Cs_3_f, Cs_3, C_3_f, C_3, B_2_f, B_2, As_2_f, As_2, A_2_f, A_2, As_2_f, As_2
dw  Ds_2_f, Ds_2, Fs_2_f, Fs_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, A_2_f, A_2, G_2_f, G_2


dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2
```

Como podéis ver, la definición consta de parejas de valores de 16 bits (usando las etiquetas que hemos definido en Const.asm), la frecuencia seguida de la nota.

Vamos a añadir ahora la segunda de las canciones.

```
Song_2:
dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4


dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4


dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, B_4_f, B_4, A_4_f, A_4,
G_4_f, G_4, G_4_f, G_4, A_4_f, A_4, B_4_f, B_4, A_4_f, A_4, E_4_f, E_4, Fs_4_f, Fs_4
dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, C_5_f, C_5, B_4_f, B_4,
A_4_f, A_4, G_4_f, G_4, D_5_f, D_5, D_5_f, D_5, A_4_f, A_4
dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, B_4_f, B_4, A_4_f, A_4,
G_4_f, G_4, G_4_f, G_4, A_4_f, A_4, B_4_f, B_4, A_4_f, A_4, E_4_f, E_4, Fs_4_f, Fs_4


dw  D_5_f, D_5, D_5_f, D_5, G_5_f, G_5, F_5_f, F_5, Ds_5_f, Ds_5, D_5_f, D_5, C_5_f,
C_5, B_4_f, B_4, A_4_f, A_4, G_4_f, G_4, D_5_f, D_5


dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4


dw  $0000
```

Observamos una última línea, ***DW $0000***. Vamos a usar este valor para indicar que las canciones se han terminado y que, a partir de aquí, vuelva al principio de las mismas.

Por último, vamos a añadir la variable para guardar la siguiente nota.

```
ptrSound:
dw      Song_2
```

Es el momento de implementar la rutina que vamos a usar para hacer sonar las canciones. Creamos el archivo Sound.asm e implementamos la rutina Play.

```
Play:
ld      hl, (ptrSound)
ld      e, (hl)
inc     hl
```

```
ld      d, (hl)
ld      a, d
or      e
jr      nz, play_cont
```

Cargamos en HL la dirección de la siguiente nota, ***LD HL, (ptrSound)***, cargamos en E el byte inferior de la frecuencia, ***LD E, (HL)***, apuntamos HL a byte alto, ***INC HL***, y lo cargamos en D, ***LD D, (HL)***. Cargamos el valor de D en A, ***LD A, D***, comprobamos si el valor de la frecuencia es $0000 para saber si estamos en el final de la canción, ***OR E***, y si no es así saltamos, ***JR NZ, play_cont***.

```
play_reset:
ld      hl, Song_1
ld      (ptrSound), hl
ret
```

Si hemos llegado al final de las canciones, cargamos en HL la dirección de la canción uno, ***LD HL, Song_1***, cargamos la primera nota en memoria, ***LD (ptrSound), HL***, y salimos, ***RET***.

```
play_cont:
inc     hl
ld      c, (hl)
inc     hl
ld      b, (hl)
inc     hl
ld      (ptrSound), hl
ld      h, b
ld      l, c

call    BEEP

ret
```

Si no hemos llegado al final de la canción apuntamos HL al byte inferior de la nota, ***INC HL***, lo cargamos en C, ***LD C, (HL)***, apuntamos HL al byte superior de la nota, ***INC HL***, lo cargamos en B, ***LD B, (HL)***, apuntamos HL al byte inferior de la frecuencia de la siguiente nota, ***INC HL***, y lo actualizamos en memoria, ***LD (ptrSound), HL***, cargamos el byte superior de la nota en H, ***LD H, B***, cargamos el byte inferior en L, ***LD L, C***, y hacemos sonar la nota, ***CALL BEEP***. Por último, salimos, ***RET***.

Ya solo queda ver si todo esto funciona. Creamos el archivo Test1.asm y añadimos las líneas siguientes:

```
org     $5dad

```

```
Loop:
call    Play


jr      Loop


include "Const.asm"
include "Sound.asm"
include "Var.asm"


end     Loop
```

Ponemos la dirección de memoria dónde se cargará nuestro programa para que sea compatible con modelos de 16K, ***ORG $5DAD***, llamamos a la rutina que se encarga de hacer sonar las canciones, ***CALL Play***, y nos quedamos en un bucle infinito, ***JR Loop***.

En la parte final del archivo, incluimos los ficheros necesarios e indicamos a PASMO (***END Loop***) que ejecute el programa.

Para ver si funciona, compilamos, cargamos en el emulador y escuchamos. Recordad que para compilar usamos la línea de comandos.

```
pasmo --tapbas Test1.asm Test1.tap
```

Si todo ha ido bien, podréis distinguir dos canciones distintas, y los que tengan mejor oído incluso podrán reconocer de que canciones se trata.

Ritmo y compás

Efectivamente, las música no suena como tal, es necesario controlar el ritmo al que deben sonar las notas, y vamos a hacer uso de las interrupciones para ello; creamos el archivo Test2.asm y empezamos a implementar.

```
org     $5dad


; ------------------------------------------------------------------------
; Indicadores para la música.
;
; Bit 7 -> Reproducir sonido                1 = Sí / 0 = No
; ------------------------------------------------------------------------
music:
db  $00
```

Como siempre, empezamos con la posición de memoria donde se carga el programa, ***ORG $5DAD***, y declaramos una etiqueta que va a servir para interactuar con las interrupciones, ***music***.

```
Main:
```

```
ld      hl, Song_2
ld      (ptrSound), hl


di
ld      a, $28
ld      i, a
im      2
ei
```

Apuntamos HL a la segunda canción, ***LD HL, Song_2***, y cargamos el valor en memoria, ***LD (ptrSound), HL***. Desactivamos las interrupciones, ***DI***, cargamos cuarenta en A, ***LD A, $28***, lo cargamos en I, ***LD I, A***, pasamos al modo dos de interrupciones, ***IM 2***, y activamos la interrupciones, ***EI***. En realidad, las interrupciones se activarían tras las siguiente instrucción.

```
Loop:
ld      a, (music)
bit     $07, a
jr      z, Loop
and     $7f
ld      (music), a


call    Play


jr      Loop
```

Cargamos el valor de ***music*** en A, ***LD A, (music)***, evaluamos si el bit siete está a uno, ***BIT $07, A***, y saltamos si no lo está, ***JR Z, Loop***.

Si el bit está a uno lo ponemos a cero, ***AND $7F***, lo cargamos en memoria, ***LD (music), A***, emitimos el sonido, ***CALL Play***, y seguimos en el bucle, ***JR Loop***.

```
include "Const.asm"
include "Sound.asm"
include "Var.asm"


end     Main
```

Por último, incluimos los ficheros y le indicamos a PASMO que incluya la llamada al programa en el cargador.

Ahora ya solo queda usar las interrupciones para controlar el ritmo. Creamos el archivo Int2.asm y empezamos.

```
org     $7e5c
```

```
MUSIC:  EQU $5dad


counter:db  $00
```

Empezamos indicando donde se carga la rutina de interrupciones, ***ORG $7e5c***, declarando una constante con la dirección de memoria donde está la etiqueta de los indicadores para la música y una etiqueta para controlar el número de interrupciones que tienen que pasar para que emitamos algún sonido.

El resto de la implementación la vamos a realizar entre ***MUSIC*** y ***counter***.

```
Isr:
push    af
push    bc
push    de
push    hl
```

Preservamos el valor de los registros, ***PUSH AF***, ***PUSH BC***, ***PUSH DE*** y ***PUSH HL***.

```
ld      a, (counter)
inc     a
ld      (counter), a
cp      $06
jr      nz, isr_end


xor     a
ld      (counter), a


ld      hl, MUSIC
set     $07, (hl)
```

Cargamos en A el valor del contador, ***LD A, (counter)***, lo incrementamos, ***INC A***, lo cargamos en memoria, ***LD (counter), A***, evaluamos si ha llegado a seis, ***CP $06***, y saltamos de no ser así, ***JR NZ, isr_end***.

Si el contador ha llegado a seis lo ponemos a cero, ***XOR A***, lo cargamos en memoria, ***LD (counter), A***, apuntamos HL a los indicadores para la música, ***LD HL, MUSIC***, y activamos el bit siete, ***SET $07, (HL)***.

```
isr_end:
pop     hl
pop     de
pop     bc
```

```
pop     af


ei
reti
```

Recuperamos el valor de los registros, ***POP HL***, ***POP DE***, ***POP BC*** y ***POP AF***, activamos las interrupciones, ***EI***, y salimos, ***RETI***.

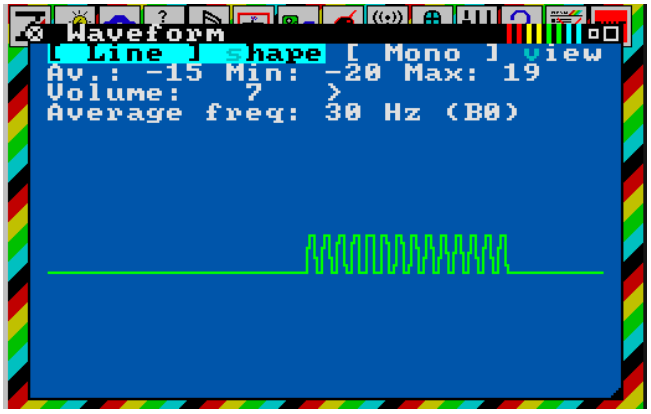
Volvemos a Test2.asm y justo encima de ***END Main*** incluimos el archivo Int2.asm.

```
include "Int2.asm"
```

El orden en el que se incluyen los archivos, en este caso concreto, es muy importante; primero vamos a compilar y ver como suena.

```
pasmo --tapbas Test2.asm Test2.tap
```

Esta es la forma de onda que podemos ver en el emulador.



Ahora ya sí se pueden distinguir las canciones, aunque las dos van al mismo ritmo y no debería ser así.

Respecto al orden de los ***include***, probad a poner el de Int2.asm por encima del de Var.asm. Compilad, cargad en el emulador (con el modelo de 16K) y a ver que pasa… ¡No funciona!

Comprobad el tamaño del programa Test2.tap, a mi me salen 9324 bytes. Volved a poner el ***include*** de Int.asm debajo del de Var.asm. Compilad y comprobad lo que ocupa ahora, a mi me salen 8520 bytes. ¿A qué se debe esta diferencia?

Al contrario que en el juego, no estamos compilando los ficheros por separado, estamos compilando como uno solo gracias a los ***include***.

La memoria de los modelos de 16K va desde la posición $0000 hasta la $7FFF. Nosotros cargamos la rutina de interrupciones en la posición $7E5C, quedando cuatrocientos diecinueve bytes hasta la posición $7FFF pero ojo, hay que contar con la pila.

Cuando ponemos el ***include*** de Int2.asm el último, desde la posición $7E5C cargamos treinta dos bytes, que es lo que ocupa la rutina de las interrupciones que hemos implementado; quedamos muy lejos de ocupar los cuatrocientos diecinueve bytes que tenemos disponibles.

Si ponemos el ***include*** de Var.asm después del de Int.asm, tras los treinta y dos bytes de la rutina de interrupciones cargamos los ochocientos cuatro bytes de la definición de las canciones, sumado un total de ochocientos treinta y seis bytes ($0344). Si estos bytes se los sumamos a la dirección donde cargamos la rutina de interrupciones, $7E5C + $0344, nos da como resultado $81A0, muy por encima de la capacidad de los modelos de 16K.

Distintos ritmos

Para conseguir que las canciones, o incluso parte de ellas, vayan a distintos ritmos, vamos a añadir un nuevo valor de 16 bits: en el byte superior vamos a poner $FF, indicando a nuestro programa que se trata de un cambio de ritmo, mientras que en el byte inferior vamos a poner el ritmo, un valor de $00 a $0F.

Creamos el archivo Var3.asm y copiamos dentro todo el código del archivo Var.asm. Vamos a añadir dos cambios de ritmo. Localizamos la etiqueta ***Song_1*** y justo debajo de ella añadimos:

```
dw  $ff0c
```

Cuando el programa se encuentre con esto, lo va a interpretar como un cambio de ritmo ($FF) y que tienen que pasar doce interrupciones ($0C) entre cada nota.

Localizamos la etiqueta ***Song_2*** y justo debajo de ella añadimos:

```
dw  $ff06
```

En este caso tienen que pasar seis interrupciones ($06) entre cada nota, por lo que podemos deducir que la segunda canción va a sonar el doble de rápido que la primera.

Creamos el archivo Int3.asm y copiamos dentro todo el código del archivo Int.asm. Creamos el archivo Test3.asm y copiamos dentro todo el código de Test2.asm.

Empezamos modificando el archivo Test3.asm. En los ***include*** sustituimos Var.asm e Int.asm por Var3.asm e Int3.asm y añadimos una línea de comentario a la etiqueta ***music***.

```
; Bit 0 a 3 -> Ritmo
```

El resto de las modificaciones las vamos a realizar justo antes de ***DI***, añadiendo las siguientes líneas.

```
ld      a, (hl)
and     $0f
ld      (music), a
```

En las líneas anteriores apuntábamos HL a Song_2, y ahora cargamos el valor al que apunta HL en A, ***LD A, (HL)***, nos quedamos con los bits donde ponemos el ritmo, ***AND $0F***, y cargamos el valor en los indicadores para la música, ***LD (music), A***.

Vamos al archivo Int3.asm y al final del mismo, justo por debajo de ***counter***, vamos a añadir una nueva etiqueta para guardar el ritmo que lleva la canción.

```
times:      db  $00
```

Ahora, localizamos la etiqueta ***Isr*** y cinco líneas más abajo ***LD A, (counter)***; justo por encima de esta línea agregamos la siguiente:

```
isr_cont:
```

Cuatro líneas más abajo encontramos ***CP $06***; modificamos esta línea dejándola como sigue:

```
cp      (hl)
```

Otras cuatro líneas más abajo encontramos ***LD hl, MUSIC***; justo por encima de esta línea agregamos la siguiente:

```
isr_set:
```

Ahora volvemos a la etiqueta Isr y después de los cuatro ***PUSH*** implementamos la parte en la que vamos a controlar los cambios de ritmo.

```
ld      a, (MUSIC)
and     $0f
ld      hl, times
cp      (hl)
jr      z, isr_cont
ld      (hl), a
xor     a
ld      (counter), a
jr      isr_set
```

Cargamos en A el valor de los indicadores de la música, ***LD A, (MUSIC)***, nos quedamos con el ritmo, ***AND $0F***, apuntamos HL a la variable donde guardamos el ritmo que lleva la canción, ***LD HL, times***, lo comparamos con el ritmo que marcan los indicadores, ***CP (HL)***, y si son iguales saltamos pues no ha cambiado, ***JR Z, isr_cont***.

Si ha cambiado el ritmo, ponemos el nuevo ritmo en memoria, ***LD (HL), A***, ponemos A a cero, ***XOR A***, y ponemos el contador a cero, ***LD (counter), A***. Por último, saltamos, ***JR isr_set***.

El aspecto que debe tener Int3.asm es el siguiente:

```
org     $7e5c


MUSIC:  EQU $5dad


Isr:
push    af
push    bc
push    de
push    hl
```

```
ld      a, (MUSIC)
and     $0f
ld      hl, times
cp      (hl)
jr      z, isr_cont
ld      (hl), a
xor     a
ld      (counter), a
jr      isr_set

isr_cont:
ld      a, (counter)
inc     a
ld      (counter), a
cp      (hl)
jr      nz, isr_end

xor     a
ld      (counter), a

isr_set:
ld      hl, MUSIC
set     $07, (hl)

isr_end:
pop     hl
pop     de
pop     bc
pop     af

ei
reti

counter:    db  $00
times:      db  $00
```

Ya solo falta modificar la rutina ***Play*** para que tenga en cuenta los cambios de ritmo; vamos al archivo Sound.asm y tras la quinta línea, ***OR E***, vamos a añadir las siguientes:

```
jr      z, play_reset
cp      $ff
```

Con ***OR E*** comprobamos si se ha llegado al fin de las canciones, en cuyo caso saltamos, ***JR Z, play_reset***. Si no hemos llegado al final de las canciones, comprobamos si estamos ante un cambio de ritmo, ***CP $FF***.

La siguiente línea ya la teníamos antes, ***JR NZ, play_cont***, y ahora lo que hace es saltar si no hay un cambio de ritmo.

Seguimos añadiendo líneas justo debajo de ***JR NZ, play_cont***.

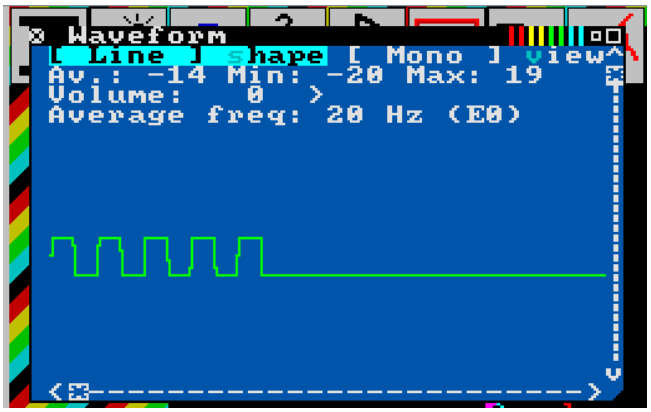
```
ld      a, e
ld      (music), a
inc     hl
ld      (ptrSound), hl
ret
```

Si no hemos saltado es porque hay un cambio de ritmo. Cargamos el nuevo ritmo en A, ***LD A, E***, lo cargamos en los indicadores para la música, ***LD (music), A***, apuntamos HL a la próxima nota (a la frecuencia), ***INC HL***, actualizamos el valor del puntero a la próxima nota, ***LD (ptrSound), HL***, y salimos, ***RET***. El resto de la rutina se queda como está.

Es el momento de ver como suena, compilamos, cargamos en el emulador y escuchamos los resultados.

```
pasmo --tapbas Test3.asm Test3.tap
```

Si todo va bien, las dos canciones se reproducen a distinto ritmo, lo cual podéis apreciar escuchando los resultados o viendo la forma de onda, donde veréis claramente que van a distinta velocidad.



Conclusión

En este capítulo hemos hecho pruebas de como implementar la música.

En el próximo capítulo vamos a utilizar lo aprendido para integrar la música en nuestro juego.

0x0D Música

Ha llegado el momento de integrar en nuestro juego todo lo que vimos en el capítulo anterior, habrá alguna pequeña variación, pero es prácticamente lo mismo. Además, ha llegado la hora de dejar todo el código comentado, hay partes que todavía no las tenemos comentadas y si lo dejamos así, con el tiempo, es posible que nos cueste más saber que es lo que hacemos ahí y, más importante aún, ¿por qué?

Creamos la carpeta Paso13 y copiamos desde la carpeta Paso12 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm, Var.asm y la carpeta Sound.

Constantes

Lo primero que vamos a hacer es declarar las constantes necesarias en el archivo Const.asm, empezando por la rutina de la ROM que vamos a usar. Localizamos la etiqueta ***UDG*** y justo debajo de ella añadimos lo siguiente:

```
; ------------------------------------------------------------------------------
; Rutina beeper de la ROM.
;
; Entrada:  HL -> Nota.
;           DE -> Duración.
;
; Altera el valor de los registros AF, BC, DE, HL e IX.
; ------------------------------------------------------------------------------
BEEP:     EQU $03b5
```

Es muy importante que leamos los comentarios, pues según vemos esta rutina de la ROM altera el valor de casi todos los registros, cosa que no hemos tenido en cuenta en el capítulo anterior, pero si que lo vamos a tener en cuenta en este capítulo.

Al final del archivo Const.asm vamos a añadir las constantes de las notas y las frecuencias.

```
; ------------------------------------------------------------------------------
; Notas a cargar en HL
; ------------------------------------------------------------------------------
C_0:    EQU $6868
Cs_0:   EQU $628d
D_0:    EQU $5d03
Ds_0:   EQU $57bf
E_0:    EQU $52d7
F_0:    EQU $4e2b
Fs_0:   EQU $49cc
```

```
G_0:    EQU $45a3
Gs_0:   EQU $41b6
A_0:    EQU $3e06
As_0:   EQU $3a87
B_0:    EQU $373e
C_1:    EQU $3425
Cs_1:   EQU $3134
D_1:    EQU $2e6f
Ds_1:   EQU $2bd3
E_1:    EQU $295c
F_1:    EQU $2708
Fs_1:   EQU $24d5
G_1:    EQU $22c2
Gs_1:   EQU $20cd
A_1:    EQU $1ef4
As_1:   EQU $1d36
B_1:    EQU $1b90
C_2:    EQU $1a02
Cs_2:   EQU $188b
D_2:    EQU $1728
Ds_2:   EQU $15da
E_2:    EQU $149e
F_2:    EQU $1374
Fs_2:   EQU $125b
G_2:    EQU $1152
Gs_2:   EQU $1058
A_2:    EQU $0f6b
As_2:   EQU $0e9d
B_2:    EQU $0db8
C_3:    EQU $0cf2
Cs_3:   EQU $0c36
D_3:    EQU $0b86
Ds_3:   EQU $0add
E_3:    EQU $0a40
F_3:    EQU $09ab
Fs_3:   EQU $091e
G_3:    EQU $089a
Gs_3:   EQU $081c
```

```
A_3:    EQU $07a6
As_3:   EQU $0736
B_3:    EQU $06cd
C_4:    EQU $066a
Cs_4:   EQU $060c
D_4:    EQU $05b3
Ds_4:   EQU $0560
E_4:    EQU $0511
F_4:    EQU $04c6
Fs_4:   EQU $0480
G_4:    EQU $043d
Gs_4:   EQU $03ff
A_4:    EQU $03c4
As_4:   EQU $038c
B_4:    EQU $0357
C_5:    EQU $0325
Cs_5:   EQU $02f7
D_5:    EQU $02ca
Ds_5:   EQU $02a0
E_5:    EQU $0279
F_5:    EQU $0254
Fs_5:   EQU $0231
G_5:    EQU $020f
Gs_5:   EQU $01f0
A_5:    EQU $01d3
As_5:   EQU $01b7
B_5:    EQU $019c
C_6:    EQU $0183
Cs_6:   EQU $016c
D_6:    EQU $0156
Ds_6:   EQU $0141
E_6:    EQU $012d
F_6:    EQU $011b
Fs_6:   EQU $0109
G_6:    EQU $00f8
Gs_6:   EQU $00e9
A_6:    EQU $00da
As_6:   EQU $00cc
```

```
B_6:    EQU $00bf
C_7:    EQU $00b2
Cs_7:   EQU $00a7
D_7:    EQU $009c
Ds_7:   EQU $0091
E_7:    EQU $0087
F_7:    EQU $007e
Fs_7:   EQU $0075
G_7:    EQU $006d
Gs_7:   EQU $0065
A_7:    EQU $005e
As_7:   EQU $0057
B_7:    EQU $0050
C_8:    EQU $004a
Cs_8:   EQU $0044
D_8:    EQU $003e
Ds_8:   EQU $0039
E_8:    EQU $0034
F_8:    EQU $0030
Fs_8:   EQU $002b
G_8:    EQU $0027
Gs_8:   EQU $0023
A_8:    EQU $0020
As_8:   EQU $001c
B_8:    EQU $0019


; ---------------------------------------------------------------------------
; Frecuencias a cargar en DE, 1 segundo ( / 2 = 0.5 ....)
; ---------------------------------------------------------------------------
C_0_f:  EQU $0010 / $20
Cs_0_f: EQU $0011 / $20
D_0_f:  EQU $0012 / $20
Ds_0_f: EQU $0013 / $20
E_0_f:  EQU $0014 / $20
F_0_f:  EQU $0015 / $20
Fs_0_f: EQU $0017 / $20
G_0_f:  EQU $0018 / $20
Gs_0_f: EQU $0019 / $20
```

```
A_0_f:  EQU $001b / $20
As_0_f: EQU $001d / $20
B_0_f:  EQU $001e / $20
C_1_f:  EQU $0020 / $20
Cs_1_f: EQU $0022 / $20
D_1_f:  EQU $0024 / $20
Ds_1_f: EQU $0026 / $20
E_1_f:  EQU $0029 / $20
F_1_f:  EQU $002b / $20
Fs_1_f: EQU $002e / $20
G_1_f:  EQU $0031 / $20
Gs_1_f: EQU $0033 / $20
A_1_f:  EQU $0037 / $20
As_1_f: EQU $003a / $20
B_1_f:  EQU $003d / $20
C_2_f:  EQU $0041 / $20
Cs_2_f: EQU $0045 / $20
D_2_f:  EQU $0049 / $20
Ds_2_f: EQU $004d / $20
E_2_f:  EQU $0052 / $20
F_2_f:  EQU $0057 / $20
Fs_2_f: EQU $005c / $20
G_2_f:  EQU $0062 / $20
Gs_2_f: EQU $0067 / $20
A_2_f:  EQU $006e / $20
As_2_f: EQU $0074 / $20
B_2_f:  EQU $007b / $20
C_3_f:  EQU $0082 / $20
Cs_3_f: EQU $008a / $20
D_3_f:  EQU $0092 / $20
Ds_3_f: EQU $009b / $20
E_3_f:  EQU $00a4 / $20
F_3_f:  EQU $00ae / $20
Fs_3_f: EQU $00b9 / $20
G_3_f:  EQU $00c4 / $20
Gs_3_f: EQU $00cf / $20
A_3_f:  EQU $00dc / $20
As_3_f: EQU $00e9 / $20
```

```
B_3_f:  EQU $00f6 / $20
C_4_f:  EQU $0105 / $20
Cs_4_f: EQU $0115 / $20
D_4_f:  EQU $0125 / $20
Ds_4_f: EQU $0137 / $20
E_4_f:  EQU $0149 / $20
F_4_f:  EQU $015d / $20
Fs_4_f: EQU $0172 / $20
G_4_f:  EQU $0188 / $20
Gs_4_f: EQU $019f / $20
A_4_f:  EQU $01b8 / $20
As_4_f: EQU $01d2 / $20
B_4_f:  EQU $01ed / $20
C_5_f:  EQU $020b / $20
Cs_5_f: EQU $022a / $20
D_5_f:  EQU $024b / $20
Ds_5_f: EQU $026e / $20
E_5_f:  EQU $0293 / $20
F_5_f:  EQU $02ba / $20
Fs_5_f: EQU $02e4 / $20
G_5_f:  EQU $0310 / $20
Gs_5_f: EQU $033e / $20
A_5_f:  EQU $0370 / $20
As_5_f: EQU $03a4 / $20
B_5_f:  EQU $03db / $20
C_6_f:  EQU $0417 / $20
Cs_6_f: EQU $0455 / $20
D_6_f:  EQU $0497 / $20
Ds_6_f: EQU $04dd / $20
E_6_f:  EQU $0527 / $20
F_6_f:  EQU $0575 / $20
Fs_6_f: EQU $05c8 / $20
G_6_f:  EQU $0620 / $20
Gs_6_f: EQU $067d / $20
A_6_f:  EQU $06e0 / $20
As_6_f: EQU $0749 / $20
B_6_f:  EQU $07b8 / $20
C_7_f:  EQU $082d / $20
```

```
Cs_7_f: EQU $08a9 / $20
D_7_f:  EQU $092d / $20
Ds_7_f: EQU $09b9 / $20
E_7_f:  EQU $0a4d / $20
F_7_f:  EQU $0aea / $20
Fs_7_f: EQU $0b90 / $20
G_7_f:  EQU $0c40 / $20
Gs_7_f: EQU $0cfa / $20
A_7_f:  EQU $0dc0 / $20
As_7_f: EQU $0e91 / $20
B_7_f:  EQU $0f6f / $20
C_8_f:  EQU $105a / $20
Cs_8_f: EQU $1153 / $20
D_8_f:  EQU $125b / $20
Ds_8_f: EQU $1372 / $20
E_8_f:  EQU $149a / $20
F_8_f:  EQU $15d4 / $20
Fs_8_f: EQU $1720 / $20
G_8_f:  EQU $1880 / $20
Gs_8_f: EQU $19f5 / $20
A_8_f:  EQU $1b80 / $20
As_8_f: EQU $1d23 / $20
B_8_f:  EQU $1ede / $20
```

Variables

El siguiente paso es añadir las variables necesarias: el puntero a la siguiente nota y las canciones. Abrimos el archivo Var.asm y añadimos al final las siguientes líneas:

```
; ------------------------------------------------------------------------------
; Datos necesarios para la música.
; ------------------------------------------------------------------------------


; ------------------------------------------------------------------------------
; Siguiente nota
; ------------------------------------------------------------------------------
ptrSound:
dw  $0000


; ------------------------------------------------------------------------------
```

```
; Canciones
; ---------------------------------------------------------------------------
Song_1:
dw  $ff0c
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f,
Ds_2, As_2_f, As_2, G_2_f, G_2
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f,
Ds_2, As_2_f, As_2, G_2_f, G_2
dw  D_3_f, D_3, D_3_f, D_3, D_3_f, D_3, Ds_3_f, Ds_3, As_2_f, As_2, Fs_2_f, Fs_2,
Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2

dw  G_3_f, G_3, G_2_f, G_2, G_2_f, G_2, G_3_f, G_3, Fs_3_f, Fs_3, F_3_f, F_3, E_3_f,
E_3, Ds_3_f, Ds_3, E_3_f, E_3
dw  Gs_2_f, Gs_2, Cs_3_f, Cs_3, C_3_f, C_3, B_2_f, B_2, As_2_f, As_2, A_2_f, A_2,
As_2_f, As_2
dw  Ds_2_f, Ds_2, Fs_2_f, Fs_2, Ds_2_f, Ds_2, Fs_2_f, Fs_2, As_2_f, As_2, G_2_f, G_2,
As_2_f, As_2, D_3_f, D_3

dw  G_3_f, G_3, G_2_f, G_2, G_2_f, G_2, G_3_f, G_3, Fs_3_f, Fs_3, F_3_f, F_3, E_3_f,
E_3, Ds_3_f, Ds_3, E_3_f, E_3
dw  Gs_2_f, Gs_2, Cs_3_f, Cs_3, C_3_f, C_3, B_2_f, B_2, As_2_f, As_2, A_2_f, A_2,
As_2_f, As_2
dw  Ds_2_f, Ds_2, Fs_2_f, Fs_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, A_2_f, A_2,
G_2_f, G_2

dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f,
Ds_2, As_2_f, As_2, G_2_f, G_2
dw  G_2_f, G_2, G_2_f, G_2, G_2_f, G_2, Ds_2_f, Ds_2, As_2_f, As_2, G_2_f, G_2, Ds_2_f,
Ds_2, As_2_f, As_2, G_2_f, G_2

Song_2:
dw  $ff05
dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4

dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5
dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4
```

```
dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, B_4_f, B_4, A_4_f, A_4,
G_4_f, G_4, G_4_f, G_4, A_4_f, A_4, B_4_f, B_4, A_4_f, A_4, E_4_f, E_4, Fs_4_f, Fs_4

dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, C_5_f, C_5, B_4_f, B_4,
A_4_f, A_4, G_4_f, G_4, D_5_f, D_5, D_5_f, D_5, A_4_f, A_4

dw  D_4_f, D_4, D_4_f, D_4, E_4_f, E_4, E_4_f, E_4, C_5_f, C_5, B_4_f, B_4, A_4_f, A_4,
G_4_f, G_4, G_4_f, G_4, A_4_f, A_4, B_4_f, B_4, A_4_f, A_4, E_4_f, E_4, Fs_4_f, Fs_4


dw  D_5_f, D_5, D_5_f, D_5, G_5_f, G_5, F_5_f, F_5, Ds_5_f, Ds_5, D_5_f, D_5, C_5_f,
C_5, B_4_f, B_4, A_4_f, A_4, G_4_f, G_4, D_5_f, D_5


dw  D_4_f, D_4, D_4_f, D_4, D_4_f, D_4, G_4_f, G_4, D_5_f, D_5

dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5

dw  C_5_f, C_5, B_4_f, B_4, A_4_f, A_4, G_5_f, G_5, D_5_f, D_5

dw  C_5_f, C_5, B_4_f, B_4, C_5_f, C_5, A_4_f, A_4


dw  $0000
```

Como vimos en el capítulo anterior, necesitamos tener una variable de indicadores para la música. Vamos a Main.asm y vemos los indicadores del juego, ***flags***. Justo debajo vamos añadir los indicadores para la música.

```
; ------------------------------------------------------------------------
; Indicadores de la música
;
; Bit 0 a 3 -> Ritmo
; Bit 7 -> suena                            0 = No, 1 = Sí
; ------------------------------------------------------------------------
music:
db $00
```

Recordad que estas etiquetas tienen que estar de inicio a cero, de lo contrario todo podría dejar de funcionar tal y como vimos en el capítulo 5.

Reproducción

Continuamos ahora con la rutina que se va a encargar de reproducir el sonido; vamos al archivo Game.asm.

Antes de nada, es necesario indicar que canción va a sonar y el ritmo. Dado que tenemos dos canciones, en los niveles pares vamos a empezar con la primera canción, mientras que en los impares vamos a empezar con la segunda canción.

Localizamos la etiqueta ChangeLevel, y vemos que la octava y novena línea son:

```
inc     a
cp      $1f
```

Justo entre estas dos líneas vamos a implementar el cambio de canción dependiendo del nivel:

```
ld      hl, Song_1
bit     $00, a
jr      z, changeLevel_cont
ld      hl, Song_2
changeLevel_cont:
ld      (ptrSound), hl


ex      af, af'
ld      a, (hl)
ld      (music), a
ex      af, af'
```

Apuntamos HL a la primera canción, ***LD HL, Song_1***, comprobamos el bit cero de A para saber si el siguiente nivel es par o impar, ***BIT $00, A***, y saltamos si es par. Si es impar apuntamos HL a la segunda canción, ***LD HL, Song_2***.

Actualizamos el puntero a la nota siguiente (en realidad el ritmo), ***LD (ptrSound), HL***, *preservamos el valor de AF ya que A contiene el siguiente nivel*, ***EX AF, AF'***, cargamos en A el ritmo, ***LD A, (HL)***, actualizamos los indicadores de la música, ***LD (music), A***, y recuperamos el valor de AF, ***EX AF, AF'***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Cambia de nivel.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
ChangeLevel:
ld      a, $06                  ; Carga el color amarillo en A
ld      (enemiesColor), a       ; Actualiza el color en memoria


ld      a, (levelCounter + 1)   ; Carga en A el nivel actual en BCD
inc     a                       ; Incrementa el nivel
daa                             ; Hace el ajuste decimal
ld      b, a                    ; Carga el valor en B
ld      a, (levelCounter)       ; Carga el nivel actual en A
inc     a                       ; Carga en A el siguiente nivel


ld      hl, Song_1              ; Apunta HL a la canción 1
bit     $00, a                  ; Evalúa el bit cero del nivel para saber si es par
```

```
jr      z, changeLevel_cont     ; Si es par salta
ld      hl, Song_2              ; Si es impar apunta HL a la canción 2

changeLevel_cont:
ld      (ptrSound), hl          ; Actualiza la siguiente nota
ex      af, af'                 ; Preserva el registro AF (A = siguiente nivel)
ld      a, (hl)                 ; Carga en A el byte inferior de la nota (ritmo)
ld      (music), a              ; Lo carga en los indicadores de la música
ex      af, af'                 ; Recupera el valor de AF
cp      $1f                     ; Compara si el nivel es el 31
jr      c, changeLevel_end      ; Si no es el 31, salta

ld      a, $01                  ; Si es el 31, lo pone a 1
ld      b, a                    ; Cargamos el valor en B

changeLevel_end:
ld      (levelCounter), a       ; Actualiza el nivel en memoria
ld      a, b                    ; Carga en A el nivel en BCD
ld      (levelCounter + 1), a   ; Lo actualiza en memoria
call    LoadUdgsEnemies         ; Carga los gráficos de los enemigos

ld      a, $20                  ; Carga en A el número total de enemigos
ld      (enemiesCounter), a     ; Lo carga en memoria

ld      hl, enemiesConfigIni    ; Apunta HL a la configuración inicial
ld      de, enemiesConfig       ; Apunta HL a la configuración
ld      bc, enemiesConfigEnd - enemiesConfigIni ; Carga en BC la longitud
                                                ; de la configuración
ldir                            ; Carga la configuración inicial en la configuración

ld      hl, shipPos             ; Apunta HL a la posición de la nave
ld      (hl), SHIP_INI          ; Carga la posición inicial

ret
```

Y ahora vamos a implementar la rutina que va a hacer sonar las canciones, lo vamos a hacer antes de la rutina ***RefreshEnemiesFire***.

```
Play:
ld      hl, (ptrSound)
```

```
ld      e, (hl)
inc     hl
ld      d, (hl)
ld      a, d
or      e
jr      z, play_reset
```

Cargamos en HL la dirección de la nota actual, ***LD HL, (ptrSound)***, cargamos en E el byte inferior de la frecuencia, ***LD E, (HL)***, apuntamos HL al byte superior, ***INC HL***, lo cargamos en D***, LD D, (HL)***, lo cargamos en A, ***LD A, D***, y evaluamos si es el fin de las canciones, ***OR E***, en cuyo caso saltamos, ***JR Z, play_reset***.

```
cp      $ff
jr      nz, play_cont
ld      a, e
ld      (music), a
inc     hl
ld      (ptrSound), hl
ret
```

Si no hemos saltado, comprobamos si la nota en realidad es un cambio de ritmo, ***CP $FF***, y saltamos si no es así, ***JR NZ, play_cont***.

Si no hemos saltado, cargamos el ritmo en A, ***LD A, E***, actualizamos los indicadores para la música, ***LD (music), A***, apuntamos HL a la siguiente nota (frecuencia), ***INC HL***, actualizamos el puntero, ***LD (ptrSound), HL***, y salimos, ***RET***.

```
play_reset:
ld      hl, Song_1
ld      (ptrSound), hl
ret
```

Si hemos llegado al final de la canciones apuntamos HL a la primera canción, ***LD HL, Song_1***, actualizamos el puntero, ***LD (ptrSound), HL***, y salimos, ***RET***.

```
play_cont:
inc     hl
ld      c, (hl)
inc     hl
ld      b, (hl)
inc     hl
ld      (ptrSound), hl
ld      h, b
ld      l, c
```

Si no hemos llegado al final de las canciones, ni ha habido un cambio de ritmo, apuntamos HL al byte inferior de la nota, ***INC HL***, lo cargamos en C, ***LD C, (HL)***, apuntamos HL al byte superior, ***INC HL***, lo cargamos en B, ***LD B, (HL)***, apuntamos HL a la siguiente nota, ***INC HL***, actualizamos el puntero, ***LD (ptrSound), HL***, cargamos el byte superior de la nota en H, ***LD H, B***, y el inferior en L, ***LD L, C***.

Al inicio del capítulo declaramos la etiqueta ***BEEP*** con el valor de la dirección de memoria donde está alojada la rutina BEEPER de la ROM. Si vemos los comentarios, esta rutina recibe en HL la nota y en DE la frecuencia, por lo que ya tenemos todo listo para llamarla.

He aquí la diferencia fundamental con respecto a la implementación que hicimos en el capítulo anterior dado que vamos a añadir algún tipo de efecto especial, motivo éste para que implementemos en una rutina propia la llamada a la ROM.

```
Play_beep:
push    af
push    bc
push    de
push    hl
call    BEEP
pop     hl
pop     de
pop     bc
pop     af


ret
```

Preservamos los valores de los registros, ***PUSH AF***, ***PUSH BC***, ***PUSH DE***, ***PUSH HL***, llamamos a la rutina de la ROM para hacer sonar la nota, ***CALL BEEP***, recuperamos el valor de los registros, ***POP HL***, ***POP DE***, ***POP BC***, ***POP AF***, y salimos, ***RET***.

Ahora podemos llamar a ***Play*** para que vaya sonando la música durante la partida, y a ***Play_beep*** para hacer sonar notas sueltas, como los efectos de sonido.

Es muy importante no cambiar el orden en el que están implementadas las rutinas, tened en cuenta que ***Play*** sale por ***Play_beep***, si cambiamos el orden el resultado puede no ser el deseado.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Hace sonar las canciones.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------
Play:
ld      hl, (ptrSound)      ; Carga en HL la dirección de la nota actual
```

```
ld      e, (hl)             ; Carga en E el byte inferior de la frecuencia
inc     hl                  ; Apunta HL al byte superior
ld      d, (hl)             ; Lo carga en D
ld      a, d                ; Lo carga en A
or      e                   ; Comprueba si es el final de la canción
jr      z, play_reset       ; Salta si es el final
cp      $ff                 ; Comprueba si escambio de ritmo
jr      nz, play_cont       ; Si no cambia el ritmo salta
ld      a, e                ; Carga el nuevo ritmo en A
ld      (music), a          ; Carga el nuevo ritmo en los indicadores de la música
inc     hl                  ; Apunta HL a la siguiente nota
ld      (ptrSound), hl      ; Actualiza el puntero
ret

play_reset:
ld      hl, Song_1          ; Apunta HL a la primera canción
ld      (ptrSound), hl      ; Actualiza el puntero
ret

play_cont:
inc     hl                  ; Apunta HL al byte inferior de la nota
ld      c, (hl)             ; Lo carga en C
inc     hl                  ; Apunta HL al byte superior de la nota
ld      b, (hl)             ; Lo carga en B
inc     hl                  ; Apunta HL a la frecuencia de la siguiente nota
ld      (ptrSound), hl      ; Actualiza el puntero
ld      h, b
ld      l, c                ; Carga la nota en HL

; ---------------------------------------------------------------------------
; Hace sonar una nota.
;
; Entrada:  HL -> Nota
;           DE -> Frecuencia
; ---------------------------------------------------------------------------
Play_beep:
push    af
push    bc
```

```
push    de
push    hl      ; Preserva el valor de los registros
call    BEEP    ; Llama a la rutina de la ROM
pop     hl
pop     de
pop     bc
pop     af      ; Recupera el valor de los registros


ret
```

Tenemos que incluir la llamada a Play desde el bucle principal del juego. Volvemos al archivo Main.asm, localizamos la etiqueta ***Main_loop*** y dentro de la misma la línea ***CALL CheckCrashShip***. Justo debajo de esta línea añadimos las siguientes:

```
ld      hl, music
bit     $07, (hl)
jr      z, main_loopCont
res     $07, (hl)
call    Play


main_loopCont:
```

Apuntamos HL a los indicadores para la música, ***LD HL, music***, comprobamos si el bit siete está activo, ***BIT $07, (HL)***, y saltamos si no lo está, ***JR Z, main_loopCont***.

Si el bit siete está activo lo desactivamos, ***RES $07, (HL)***, y hacemos sonar la siguiente nota de las canciones, ***CALL Play***.

Por último, añadimos la etiqueta a la que saltamos cuando el bit siete no está activo, ***main_loopCont***.

El aspecto de Main.asm una vez comentado es el siguiente:

```
org     $5dad


; ------------------------------------------------------------------------------
; Indicadores
;
; Bit 0 -> se debe mover la nave            0 = No, 1 = Sí
; Bit 1 -> el disparo está activo           0 = No, 1 = Sí
; Bit 2 -> se deben mover los enemigos      0 = No, 1 = Sí
; Bit 3 -> cambia dirección enemigos        0 = No, 1 = Sí
; Bit 4 -> mover disparo enemigo            0 = No, 1 = Sí
; ------------------------------------------------------------------------------
```

```
flags:
db $00
; ------------------------------------------------------------------------------
; Indicadores de la música
;
; Bit 0 a 3 -> Ritmo
; Bit 7 -> suena                          0 = No, 1 = Sí
; ------------------------------------------------------------------------------
music:
db $00

Main:
ld      a, $02
call    OPENCHAN                    ; Abre el canal 2, pantalla superior

ld      hl, udgsCommon              ; Apunta HL a la dirección de los UDG
ld      (UDG), hl                   ; Cambia la dirección de los UDG

ld      hl, ATTR_P                  ; Apunta HL a la dirección de los atributos
permanentes
ld      (hl), $07                   ; Pone la tinta en blanco y fondo en negro
call    CLS                         ; Limpia la pantalla

xor     a                           ; A = 0
out     ($fe), a                    ; Borde = negro
ld      a, (BORDCR)                 ; Carga el valor de BORDCR en A
and     $c0                         ; Se queda con brillo y flash
or      $05                         ; Pone la tinta a 5 y el fondo a 0
ld      (BORDCR), a                 ; Actualiza BORDCR

di                                  ; Desactiva la interrupciones
ld      a, $28                      ; Carga 40 en A
ld      i, a                        ; Lo carga en el registro I
im      2                           ; Pasa a modo 2 de interrupciones
ei                                  ; Activa las interrupciones

ld      a, (flags)                  ; Carga los indicadores en A
Main_start:
ld      hl, enemiesCounter          ; Apunta HL al contador de enemigos
```

```
ld      de, enemiesCounter + $01    ; Apunta DE al contador de niveles
ld      (hl), $00                   ; Pone a cero el contador de enemigos
ld      bc, $08                     ; Carga en BC el número de bytes a limpiar
ldir                                ; Limpia los bytes
ld      a, $05
ld      (livesCounter), a           ; Pone el contador de vidas a 5

call    ResetEnemiesFire            ; Inicializa los disparos enemigos
call    ChangeLevel                 ; Cambia de nivel
call    PrintFirstScreen            ; Pinta la pantalla de menú y espera
call    PrintFrame                  ; Pinta el marco
call    PrintInfoGame               ; Pinta los títulos de información de la partida
call    PrintShip                   ; Pinta la nave
call    PrintInfoValue              ; Pinta la información de la partida

call    LoadUdgsEnemies             ; Carga los enemigos
call    PrintEnemies                ; Los pinta
; Retardo
call    Sleep                       ; Produce un retardo antes de empezar el nivel

; Bucle principal
Main_loop:
call    CheckCtrl                   ; Comprueba la pulsación de los controles
call    MoveFire                    ; Muevo el disparo

push    de                          ; Preserva DE
call    CheckCrashFire              ; Evalúa las colisiones entre enemigos y disparo
pop     de                          ; Recupera DE

ld      a, (enemiesCounter)         ; Carga el número de enemigos activos en A
or      a                           ; Comprueba si es 0
jr      z, Main_restart             ; Si es 0 salta

call    MoveShip                    ; Mueve la nave
call    ChangeEnemies               ; Cambia la dirección de los enemigos si procede
call    MoveEnemies                 ; Mueve los enemigos
call    MoveEnemiesFire             ; Mueve los disparos de los enemigos
call    CheckCrashShip              ; Evalúa las colisiones entre la nave
```

```
                                    ; y los enemigos y sus disparos

ld      hl, music                   ; Apunta HL a los indicadores para la música
bit     $07, (hl)                   ; Evalúa si debe sonar una nota
jr      z, main_loopCont            ; Si no debe sonar, salta
res     $07, (hl)                   ; Desactiva el bit siete de music
call    Play                        ; Hace sonar la nota

main_loopCont:
ld      a, (livesCounter)           ; Carga las vidas en A
or      a                           ; Comprueba si están a cero
jr      z, GameOver                 ; Si están a cero salta, ¡GAME OVER!

jr      Main_loop                   ; Bucle principal

Main_restart:
ld      a, (levelCounter)           ; Carga el número de nivel en A
cp      $1e                         ; Comprueba si es el 31 (tenemos 30)
jr      z, Win                      ; Si es el 31 salta, ¡VICTORIA!

call    FadeScreen                  ; Hace el fundido de la pantalla
call    ChangeLevel                 ; Cambia de nivel
call    PrintFrame                  ; Pinta el marco
call    PrintInfoGame               ; Pinta los títulos de información
call    PrintShip                   ; Pinta la nave
call    PrintInfoValue              ; Pinta la información
call    PrintEnemies                ; Pinta los enemigos
call    ResetEnemiesFire            ; Reinicia los disparos de los enemigos
; Retardo
call    Sleep                       ; Produce un retardo
jr      Main_loop                   ; Bucle principal

; ¡GAME OVER!
GameOver:
xor     a                           ; Pone A = 0
call    PrintEndScreen              ; Pinta la pantalla de fin y espera
jp      Main_start                  ; Menú principal
```

```
; ¡VICTORIA!
Win:
ld      a, $01                      ; Pone A = 1
call    PrintEndScreen              ; Pinta la pantalla de fin y espera
jp      Main_start                  ; Menú principal

include "Const.asm"
include "Var.asm"
include "Graph.asm"
include "Print.asm"
include "Ctrl.asm"
include "Game.asm"

end     Main
```

Control de música por interrupciones

Es la rutina de interrupciones el lugar donde indicamos el momento en el que tiene que sonar una nueva nota, vamos al archivo Int.asm y lo primero que vamos a hacer añadir dos constantes justo por delante de ***T1: EQU $C8***:

```
FLAGS:  EQU $5dad
MUSIC:  EQU $5dae
```

Estás etiquetas hacen referencia a las posiciones de memoria en las que tenemos definidos los indicadores en Main.asm.

Después de los cuatro ***PUSH*** de la etiqueta ***Isr***, nos encontramos la línea ***LD HL, $5DAD*** que vamos a modificar dejándola como sigue:

```
ld      hl, FLAGS
```

Al final del archivo vamos a añadir las variables donde guardar el ritmo de la canción, y el contador para saber si hay que activar el bit que indicará que hay que hacer sonar una nota.

```
countTempo: db $00
tempo:      db $00
```

Localizamos la etiqueta ***Isr_T1***, y en la quinta línea encontramos ***JR NZ, Isr_end***. Vamos a modificar esta línea dejándola como sigue:

```
jr       nz, Isr_sound
```

Localizamos la etiqueta ***Isr_end*** y justo por encima de ella vamos a implementar el control de la música.

```
Isr_sound:
ld      a, (MUSIC)
and     $0f
ld      hl, tempo
cp      (hl)
jr      z, Isr_soundCont
ld      (hl), a
jr      Isr_soundEnd
```

Cargamos en A los indicadores para la música, ***LD A, (MUSIC)***, nos quedamos con el ritmo, ***AND $0F***, apuntamos HL al ritmo actual, ***LD HL, tempo***, y lo comparamos con el ritmo que hay en los indicadores para la música, ***CP (HL)***. Si los dos valores son iguales, no hay cambio de ritmo y saltamos, ***JR Z, Isr_soundCount***. Si los valores son distintos, actualizamos el ritmo actual, ***LD (HL), A***, y saltamos, ***JR Isr_soundEnd***.

```
Isr_soundCont:
ld      a, (countTempo)
inc     a
ld      (countTempo), a
cp      (hl)
jr      nz, Isr_end
```

Cargamos en A el contador del ritmo, ***LD A, (countTempo)***, lo incrementamos, ***INC A***, lo actualizamos en memoria, ***LD (countTempo), A***, los comparamos, ***CP (HL)***, y si no son iguales, no hay que hacer sonar la nota y saltamos, ***JR NZ, Isr_end***.

```
Isr_soundEnd:
xor     a
ld      (countTempo), a
ld      hl, MUSIC
set     $07, (hl)
```

Si no hemos saltado hay que hacer sonar la nota. Ponemos A a cero, ***XOR A***, actualizamos el contador del ritmo, ***LD (countTempo), A***, apuntamos HL a los indicadores para la música, ***LD HL, MUSIC***, y activamos el bit siete para indicar que debe sonar una nota, ***SET $07, (HL)***.

El aspecto final de Int.asm, una vez comentado, es el siguiente:

```
org     $7e5c


FLAGS:  EQU $5dad   ; Indicadores generales
MUSIC:  EQU $5dae   ; Indicadores para la música
T1:     EQU $c8     ; Interrupciones para activar el cambio de dirección de enemigos
```

```
Isr:
push    hl
push    de
push    bc
push    af                  ; Preserva el valor de los registros

ld      hl, FLAGS           ; Apunta HL a los indicadores
set     $00, (hl)           ; Activa el bit 0, mover nave

ld      a, (countEnemy)     ; Carga en A el contador para mover los enemigos
inc     a                   ; Lo incrementa
ld      (countEnemy), a     ; Lo actualiza
sub     $03                 ; Le resta 3
jr      nz, Isr_T1          ; Si el valor no es cero, salta
ld      (countEnemy), a     ; Pone el contador a cero
set     $02, (hl)           ; Activa el bit 2 de los indicadores, mover enemigos
set     $04, (hl)           ; Activa el bit 4, mover disparo enemigo

; Cambio de dirección de los enemigos
Isr_T1:
ld      a, (countT1)        ; Carga en A el contador para cambiar la dirección
inc     a                   ; Lo incrementa
ld      (countT1), a        ; Lo actualiza en memoria
sub     T1                  ; Le resta las interrupciones que tienen que pasar
jr       nz, Isr_sound      ; Si el valor no es cero, salta
ld      (countT1), a        ; Pone el contador a cero
set     $03, (hl)           ; Activa el bit 3 de los indicadores, cambiar dirección
enemigos

; Sonido
Isr_sound:
ld      a, (MUSIC)          ; Carga en A el valor de los indicadores para la música
and     $0f                 ; Se queda con el ritmo
ld      hl, tempo           ; Apunta HL al ritmo actual
cp      (hl)                ; Lo compara con los indicadores para la música
jr      z, Isr_soundCont    ; Si son iguales, salta
ld      (hl), a             ; Si son distintos, actualiza el ritmo actual
jr      Isr_soundEnd        ; Salta para hacer sonar la nota
```

```
Isr_soundCont:
ld      a, (countTempo)     ; Carga en A el valor del contador del ritmo
inc     a                   ; Lo incrementa
ld      (countTempo), a     ; Lo actualiza en memoria
cp      (hl)                ; Lo compara con el ritmo actual
jr      nz, Isr_end         ; Si son distintos, salta

Isr_soundEnd:
xor     a                   ; Pone A = 0
ld      (countTempo), a     ; Pone el contador de ritmo a 0
ld      hl, MUSIC           ; Apunta HL a los indicadores para la música
set     $07, (hl)           ; Activa el bit 7, sonar

Isr_end:
pop     af
pop     bc
pop     de
pop     hl                  ; Recupera los valores de los registros
ei                          ; Activa las interrupciones
reti                        ; Sale

; Contador para cambio de dirección de los enemigos
countT1:    db $00
; Contador para movimiento de los enemigos
countEnemy: db $00
; Contador para hacer sonar notas
countTempo: db $00
; Ritmo actual
tempo:      db $00
```

Si compilamos y cargamos en el emulador, debemos tener música durante la partida, y cada nivel, ya sea par o impar, debe empezar sonando una u otra canción.

Si la dificultad es demasiada, en el bucle principal comentad la línea ***CALL CheckCrashShip***, para evitar que nos maten.

Efectos de sonido

Además de la música, vamos a implementar efectos de sonido. En concreto vamos a implementar tres efectos distintos:

- Con el movimiento los enemigos.
- Con la explosión la nave.
- Con el disparo.

Estos efectos de sonido los vamos a implementar como rutinas independientes en Game.asm. Como ya hemos visto como hacemos sonar cada nota, vamos a ver el código final de las rutinas sin entrar en detalle, llegados a este punto ya dominamos esta parte.

Localizamos la rutina ***RefreshEnemiesFire*** y justo por delante de ella añadimos las líneas siguientes:

```
; ------------------------------------------------------------------------------
; Emite el sonido del movimiento de los enemigos
;
; Altera el valor de los registros HL y DE
; ------------------------------------------------------------------------------
PlayEnemiesMove:
ld      hl, $0a     ; Carga la nota en HL
ld      de, $00     ; Carga la frecuencia en DE
call    Play_beep   ; Emite la nota

ld      hl, $14     ; Carga la nota en HL
ld      de, $20     ; Carga la frecuencia en DE
call    Play_beep   ; Emite la nota

ld      hl, $0a     ; Carga la nota en HL
ld      de, $10     ; Carga la frecuencia en DE
call    Play_beep   ; Emite la nota

ld      hl, $30     ; Carga la nota en HL
```

```
ld      de, $1e     ; Carga la frecuencia en DE
jr      Play_beep   ; Emite la nota y sale


; ------------------------------------------------------------------------
; Emite el sonido de la explosión de la nave
;
; Altera el valor de los registros HL y DE
; ------------------------------------------------------------------------
PlayExplosion:
ld      hl, $27a0       ; Carga la nota en HL
ld      de, $2b / $20   ; Carga la frecuencia en DE
call    Play_beep       ; Emite el sonido

ld      hl, $13f4       ; Carga la nota en HL
ld      de, $37 / $20   ; Carga la frecuencia en DE
call    Play_beep       ; Emite el sonido

ld      hl, $14b9       ; Carga la nota en HL
ld      de, $52 / $20   ; Carga la frecuencia en DE
call    Play_beep       ; Emite el sonido

ld      hl, $1a2c       ; Carga la nota en HL
ld      de, $41 / $20   ; Carga la frecuencia en DE
jr      Play_beep       ; Emite el sonido y sale

; ------------------------------------------------------------------------
; Emite el sonido del disparo de la nave
;
; Altera el valor de los registros HL y DE
; ------------------------------------------------------------------------
PlayFire:
ld      hl, $64     ; Carga la nota en HL
ld      de, $01     ; Carga la frecuencia en DE
jr      Play_beep   ; Emite el sonido y sale
```

Como podemos ver, en las tres rutinas se van cargando las notas en HL, las frecuencias en DE, y se llama con ***CALL*** a la rutina ***Play_beep***, excepto la última nota de cada efecto, en la que usamos ***JR*** para salir con el ***RET*** de ***Play_beep***.

Ya solo queda llamar a cada rutina. Localizamos la rutina ***MoveEnemiesFire***, y vemos que la última línea es ***JP RefreshEnemiesFire***. Justo por encima de esta línea vamos a añadir la llamada al sonido que vamos a emitir cuando se mueven los enemigos, más en concreto sus disparos.

```
call    PlayEnemiesMove ; Emite el sonido de movimiento de enemigos
```

La siguiente llamada que vamos a incluir es al sonido que se produce al disparar. Localizamos la rutina ***MoveFire*** y tras la sexta línea, ***SET $01, (HL)***, añadimos las líneas siguientes:

```
push    hl                  ; Preserva el valor de HL
call    PlayFire            ; Emite el sonido del disparo
pop     hl                  ; Recupera el valor de HL
```

En el caso del sonido de la explosión, no vamos a llamarlo desde Game.asm, aunque parezca incoherente.

Si localizamos la etiqueta ***checkCrashShip_endLoop***, que está dentro de la rutina ***CheckCreashShip***, y nos fijamos en la línea anterior ***JP PrintExplosion***, podemos deducir que cuando la nave es alcanzada saltamos a pintar la explosión, y si vamos a ***PrintExplosion***, observamos que pinta la explosión y salta a pintar la nave, ***JP PrintShip*** y sale por allí.

Visto esto, y para no tener que modificar el comportamiento actual, y aunque no sea coherente, la llamada a la emisión del sonido la vamos a hacer desde ***PrintExplosion***. Vamos al archivo Print.asm, localizamos la rutina ***PrintExplosion*** y vemos que la última línea es ***JP PrintShip***. Justo por encima de esta línea añadimos la siguiente:

```
call    PlayExplosion   ; Emite el sonido de la explosión
```

Ya tenemos la música y todos los efectos de sonido de nuestro juego. Ahora podemos compilar y ver los resultados.

Conclusión

En este capítulo hemos añadido efectos de sonido e integrado la música del capítulo anterior para que suene durante la partida.

En el próximo capítulo implementaremos la selección de distintos niveles de dificultad, la posibilidad de silenciar la música y añadiremos la pantalla de carga.

0x0E Dificultad, mute y pantalla de carga

En este capítulo vamos a dar la posibilidad de seleccionar entre cinco niveles de dificultad, silenciar la música durante la partida e incluir la pantalla de carga.

Creamos la carpeta Paso14 y copiamos desde la carpeta Paso13 los archivos Cargador.tap, Const.asm, Ctrl.asm, Game.asm, Graph.asm, Int.asm, Main.asm, make o make.bat, Print.asm y Var.asm.

Dificultad

Dependiendo del nivel de dificultad seleccionada, el comportamiento de los enemigos va a variar, así como el número de vidas.

En los niveles uno y dos, las naves enemigas no llegan hasta la posición de nuestra nave, y los disparos enemigos simultáneos es uno en el nivel uno y cinco en el nivel dos.

En el resto de niveles, las naves enemigas llegan hasta la posición de nuestra nave (ese es el comportamiento que tienen ahora), y en el caso del nivel tres los disparos enemigos simultáneos es uno, mientras que en los niveles cuatro y cinco son cinco. La diferencia entre el nivel cuatro y cinco es que en el cuatro, cada vez que se supera un nivel volvemos a tener cinco vidas, al más puro estilo Plaga Galáctica.

Ya que vamos a dar la opción de seleccionar entre cinco niveles de dificultad, debemos modificar la pantalla de inicio.

Vamos al archivo Var.asm, y modificamos las etiquetas ***title*** y ***firstScreen***, dejándolas de la siguiente manera:

```
title:
db  $10, $02, $16, $00, $08, "BATALLA ESPACIAL", $0d, $0d, $ff


firstScreen:
db  $10, $06, "Las naves alienigenas atacan la", $0d
db  "Tierra, el futuro depende de ti.", $0d, $d
db  "Destruye todos los enemigos que", $0d
db  "puedas, y protege el planeta.", $0d, $0d
db  $10, $03, "Z - Izquierda", $16, $08, $15,"X - Derecha"
db  $0d, $0d, "V - Disparo", $16, $0a, $15, "M - Sonido", $0d, $0d
db  $10, $04, "1 - Teclado       3 - Sinclair 1", $0d, $0d
db  "2 - Kempston      4 - Sinclair 2", $0d, $0d
db  $10, $07, $16, $10, $07, "5 - Dificultad ", $0d, $0d
db  $10, $05, "Apunta, dispara, esquiva a las", $0d
db "naves enemigas, vence y libera", $0d
db "al planeta de la amenza."
```

```
db  $ff
```

Como añadimos nuevas opciones, quitamos un retorno de carro en el título y varios en el resto de la pantalla para que quepa todo.

Seguimos en Var.asm, localizamos la etiqueta ***enemiesColor*** y debajo del valor (***DB $06***) añadimos la etiqueta que vamos a usar para guardar la dificultad seleccionada.

```
hardness:
db  $03
```

Ahora que tenemos la pantalla de inicio modificada, es necesario mostrar en la misma el nivel de dificultad seleccionado. Vamos a Print.asm y tras la rutina ***PrintFrame*** implementamos la rutina que pinta la dificultad seleccionada:

```
; ------------------------------------------------------------------------
; Pinta la dificultad seleccionada en el menú
;
; Altera el valor de los registros AF y BC.
; ------------------------------------------------------------------------
PrintHardness:
ld      a, $02          ; Carga en A la tinta
call    Ink             ; Asigna la tinta
ld      b, $08          ; Carga en B la coordenada Y (invertida)
ld      c, $0a          ; Carga en X la coordenada X (invertida)
call    At              ; Posiciona el cursor
ld      a, (hardness)   ; Carga en A la dificultad
add     a, '0'          ; Le suma el carácter 0
rst     $10             ; Pinta la dificultad


ret
```

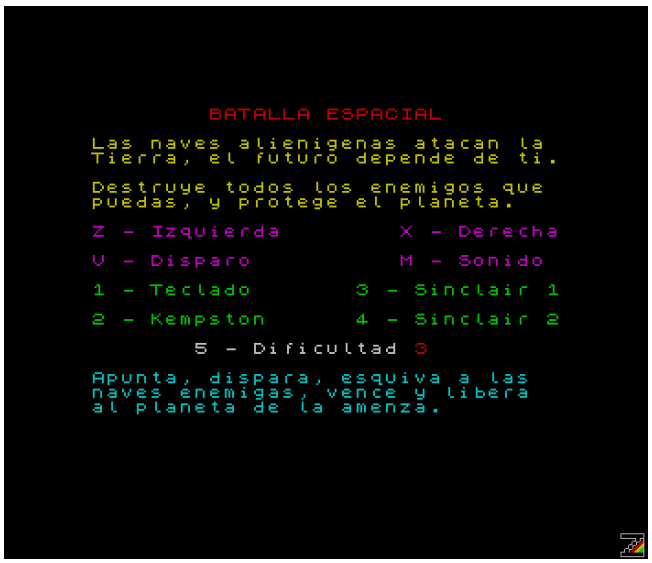
En estos momentos ya debemos tener cierto nivel de conocimientos, así que vamos a explicar la rutina solo por encima.

Ponemos la tinta en rojo (2), posicionamos el cursor, cargamos la dificultad, le sumamos el carácter ‘0’ para calcular el código de carácter de la dificultad y lo pintamos.

Seguimos en Print.asm, localizamos la rutina ***PrintFirstScreen*** y tras quinta línea, ***CALL PrintString***, añadimos la llamada para pintar la dificultad seleccionada.

```
call    PrintHardness
```

Compilamos, cargamos en el emulador y vemos los resultados.



Como podemos observar, hemos quitado retornos de carro, añadido una nueva tecla de control para activar/desactivar la música y añadido la opción de seleccionar la dificultad.

Ahora vamos a añadir la implementación de la selección de dificultad. Seguimos en Print.asm, localizamos la etiqueta ***printFirstSreen_end***, y justo por encima de ella tenemos la línea ***JR C, printFirstScreen_op***. Justo por encima de esta línea, añadimos las siguientes:

```
jr      nc, printFirstScreen_end
rra
```

Si se ha pulsado la tecla 4 saltamos al fin de la rutina, ***JR NC, printFirstScreen_end***. Si no se ha pulsado, rotamos A a la derecha para saber si se ha pulsado el 5.

La siguiente línea ya estaba, ***JR C, printFirstScreen_op***, y la dejamos como está, si no se ha pulsado el 5 salta para seguir en el bucle.

Por debajo de esta línea, añadimos las siguientes, que se ejecutaran en el caso de haber pulsado el 5:

```
ld      a, (hardness)
inc     a
cp      $06
jr      nz, printFirstScreen_opCont
ld      a, $01
```

Cargamos la dificultad en A, ***LD A, (hardness)***, la incrementamos, ***INC A***, evaluamos si ha llegado a seis, ***CP $06***, saltamos si no ha llegado, ***JR NZ, printFirstScreen_opCont***, y si sí hemos llegado la ponemos a uno, ***LD A, $01***.

```
printFirstScreen_opCont:
ld      (hardness), a
call    PrintHardness
jr      printFirstScreen_op
```

Actualizamos la dificultad en memoria, ***LD (hardness), A***, la pintamos, ***CALL PrintHardess***, y seguimos en el bucle hasta que se pulse una tecla del 1 al 4.

El aspecto de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Pantalla de presentación y selección de controles.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
PrintFirstScreen:
call    CLS                 ; Limpia la pantalla
ld      hl, title           ; Carga en HL la definición del título
call    PrintString         ; Pinta el título
ld      hl, firstScreen     ; Carga en HL la definición de la pantalla
call    PrintString         ; Pinta la pantalla
call    PrintHardness       ; Pinta la dificultad

printFirstScreen_op:
ld      b, $01              ; Carga 1 en B, opción teclas
ld      a, $f7              ; Carga en A la semifila 1-5
in      a, ($fe)            ; Lee el teclado
rra                         ; Rota A a la derecha para saber si ha pulsado 1
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Kempston
rra                         ; Rota A a la derecha para saber si ha pulsado 2
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Sinclar 1
rra                         ; Rota A a la derecha para saber si ha pulsado 3
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
inc     b                   ; Incrementa B, opción Sinclar 2
rra                         ; Rota A a la derecha para saber si ha pulsado 4
jr      nc, printFirstScreen_end    ; Si no hay acarreo, se ha pulsado y salta
rra                         ; Rota A a la derecha para saber si ha pulsado 5
jr      c, printFirstScreen_op      ; Si hay acarreo, no se ha pulsado, bucle
ld      a, (hardness)       ; Carga la dificultad en A
inc     a                   ; La incrementa
cp      $06                 ; Comprueba si hemos pasado de 5
jr      nz, printFirstScreen_opCont ; Si no hemos pasado, salta
ld      a, $01              ; Pone A a 1
```

```
printFirstScreen_opCont:
ld      (hardness), a       ; Actualiza la dificultad en memoria
call    PrintHardness       ; La pinta
jr      printFirstScreen_op ; Bucle hasta que pulse tecla del 1 al 4

printFirstScreen_end:
ld      a, b                ; Carga en A la opción seleccionada
ld      (controls), a       ; Lo carga en memoria
call    FadeScreen          ; Fundido de pantalla

ret
```

Compilamos, cargamos en el emulador y probamos la selección de dificultad.

¡No funciona! O al menos, no como nos gustaría. Cambia tan rápido de dificultad que es extremadamente difícil seleccionar la dificultad deseada.

Vamos a cambiar la parte de la rutina que evalúa las teclas pulsadas, apoyándonos en las rutinas de la ROM, que controlan que haya una pausa entre cada detección de tecla para evitar su repetición, y vamos a cambiar al modo de interrupción 1 para que las variables de sistema se actualicen automáticamente.

Vamos al archivo Const.am y vamos a añadir dos contantes que apuntan a dos variables de sistema, y que nos hacen falta para saber cual es la última tecla pulsada usando las rutinas de la ROM.

```
;-------------------------------------------------------------------------------
; Dirección de memoria donde están los flags de estado del teclado cuando
; están activas las interrupciones en modo 1.
;
; Bit 3 = 1 entrada en modo L, 0 entrada en modo K.
; Bit 5 = 1 se ha pulsado una tecla, 0 no se ha pulsado.
; Bit 6 = 1 carácter numérico, 0 alfanumérico.
;-------------------------------------------------------------------------------
FLAGS_KEY:  equ $5c3b


;-------------------------------------------------------------------------------
; Dirección de memoria dónde está la última tecla pulsada
; cuando están activas las interrupciones en modo 1.
;-------------------------------------------------------------------------------
LAST_KEY:   equ $5c08
```

Si leemos los comentarios podremos saber que es cada cosa.

Volvemos a Print.asm, localizamos la etiqueta ***printFirstScreen_op*** y borramos desde ***LD B, $01*** hasta ***JR C, printFirstScreen_op***, que está justo antes de ***LD A , (hardness)***.

Justo por encima de ***printFirstScreen_op*** añadimos las líneas siguientes:

```
di
im      1
ei


ld        hl, FLAGS_KEY
set       $03, (hl)
```

Desactivamos las interrupciones, ***DI***, cambiamos a modo uno, ***IM 1***, reactivamos las interrupciones, ***EI***, cargamos en HL la dirección de los indicadores del teclado, ***LD HL, FLAGS_KEY***, y ponemos la entrada en modo L, ***SET $03, (HL)***.

Justo debajo de ***printFirstScreen_op*** implementamos la lectura del teclado usando la ROM:

```
bit     $05, (hl)
jr      z, printFirstScreen_op
res     $05, (hl)
```

Comprobamos si se ha pulsado una tecla, ***BIT $05, (HL)***, si no se ha pulsado vuelve al bucle, ***JR Z, printFirstScreen_op***, y si se ha pulsado ponemos el bit 5 a cero para futuras inspecciones, ***RES $05, (HL)***.

```
ld      b, $01
ld      c, '0' + $01
ld      a, (LAST_KEY)
cp      c
jr      z, printFirstScreen_end
```

Cargamos uno en B (opción teclado), ***LD B, $01***, cargamos en C el código ASCII del uno, ***LD C, '0' + $01***, cargamos en A la última tecla pulsada, ***LD A, (LAST_KEY)***, comprobamos si es el uno, ***CP C***, y saltamos de ser así, ***JR Z, printFirstScreen_end***.

```
inc     b
inc     c
cp      c
jr      z, printFirstScreen_end
```

Incrementamos B (opción Kempston), ***INC B***, incrementamos C (tecla dos), ***INC C***, comprobamos si se ha pulsado, ***CP C***, y saltamos de ser así, ***JR Z, printFirstScreen_end***.

Hacemos lo mismo para comprobar si se ha pulsado el tres o el cuatro (Sinclair 1 y 2):

```
inc     b
```

```
inc     c
cp      c
jr      z, printFirstScreen_end
inc     b
inc     c
cp      c
jr      z, printFirstScreen_end
```

Ya solo nos queda comprobar si se ha pulsado el cinco (dificultad):

```
inc     c
cp      c
jr      nz, printFirstScreen_op
```

Incrementamos C (tecla 5), ***INC C***, comprobamos si se ha pulsado, ***CP C***, y volvemos al inicio del bucle si no ha sido así, ***JR NZ, printFirstScreen_op***.

Ya solo nos queda un aspecto muy importante. Localizamos la etiqueta ***printFirstScreen_end***, y justo antes de ***RET*** añadimos las líneas siguientes:

```
di
im      2
ei
```

Desactivamos las interrupciones, ***DI***, pasamos a modo dos, ***IM 2***, y activamos las interrupciones, ***EI***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Pantalla de presentación, selección de controles y dificultad.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
PrintFirstScreen:
call    CLS                 ; Limpia la pantalla
ld      hl, title           ; Carga en HL la definición del título
call    PrintString         ; Pinta el título
ld      hl, firstScreen     ; Carga en HL la definición de la pantalla
call    PrintString         ; Pinta la pantalla
call    PrintHardness       ; Pinta la dificultad


di                          ; Desactiva las interrupciones
im      1                   ; Cambia a modo 1
```

```
ei                          ; Reactiva las interrupciones


ld      hl, FLAGS_KEY           ; Carga en HL la dirección de los indicadores del
teclado
set      $03, (hl)           ; Pone entrada en modo L
printFirstScreen_op:
bit     $05, (hl)           ; Comprueba si se ha pulsado una tecla
jr      z, printFirstScreen_op  ; Si no se ha pulsado, vuelve al bucle
res     $05, (hl)           ; Es necesario poner el bit a 0 para futuras inspecciones
ld      b, $01              ; Carga 1 en B, opción teclas
ld      c, '0' + $01        ; Carga el código ASCII del 1 en C
ld      a, (LAST_KEY)       ; Carga en A la última tecla pulsada
cp      c                   ; Comprueba si ha pulsado el 1
jr      z, printFirstScreen_end ; Si se ha pulsado el 1, sale
inc     b                   ; Incrementa B, opción Kempston
inc     c                   ; Incrementa C, tecla 2
cp      c                   ; Comprueba si ha pulsado el 2
jr      z, printFirstScreen_end ; Si se ha pulsado el 2, sale
inc     b                   ; Incrementa B, opción Sinclair 1
inc     c                   ; Incrementa C, tecla 3
cp      c                   ; Comprueba si ha pulsado el 3
jr      z, printFirstScreen_end ; Si se ha pulsado el 3, sale
inc     b                   ; Incrementa B, opción Sinclair 2
inc     c                   ; Incrementa C, tecla 4
cp      c                   ; Comprueba si ha pulsado el 4
jr      z, printFirstScreen_end ; Si se ha pulsado el 4, sale
inc     c                   ; Incrementa C, tecla 5
cp      c                   ; Comprueba si ha pulsado el 5
jr      nz, printFirstScreen_op ; Si no se ha pulsado sigue en el bucle
ld      a, (hardness)       ; Carga la dificultad en A
inc     a                   ; La incrementa
cp      $06                 ; Comprueba si ha pasado de 5
jr      nz, printFirstScreen_opCont ; Si no ha pasado, salta
ld      a, $01              ; Pone A a 1
printFirstScreen_opCont:
ld      (hardness), a       ; Actualiza la dificultad en memoria
call    PrintHardness       ; La pinta
jr      printFirstScreen_op ; Bucle hasta que pulse tecla del 1 al 4
```

```
printFirstScreen_end:
ld      a, b                ; Carga en A la opción seleccionada
ld      (controls), a       ; Lo carga en memoria
call    FadeScreen          ; Fundido de pantalla


di                          ; Desactiva las interrupciones
im      2                   ; Cambia a modo 2
ei                          ; Activa las interrupciones


ret
```

Compilamos, cargamos en el emulador y comprobamos que podemos seleccionar la dificultad deseada.

Ahora que ya seleccionamos la dificultad, vamos a cambiar el comportamiento de nuestro juego en función de la dificultad seleccionada. Según la dificultad, las naves llegan o no hasta la línea donde está nuestra nave, y puede haber uno o cinco disparos enemigos a un mismo tiempo; estos dos aspectos los controlamos con las constantes ***ENEMY_TOP_B*** y ***FIRES***, necesitamos que esos valores sean variables, vamos a Var.asm, localizamos la etiqueta ***hardness***, y justo por encima de ella añadimos estas líneas:

```
enemiesTopB:
db  ENEMY_TOP_B
firesTop:
db  FIRES
```

Ya tenemos nuestras variables, ahora hay que usarlas. Vamos a Game.asm, localizamos la etiqueta ***moveEnemies_Y_down*** y la línea ***SUB ENEMY_TOP_B***. Vamos a sustituir esta línea por las siguientes, leyendo los comentarios sabemos que hace:

```
push    hl                  ; Preserva el valor de HL
ld      hl, enemiesTopB     ; Apunta HL al tope por abajo
sub     (hl)                ; Lo resta
pop     hl                  ; Recupera el valor de HL
```

Seguimos en Game.asm, localizamos la etiqueta ***enableEnemiesFire_loop*** y la línea ***CP FIRES***. Vamos a sustituir esta línea por las siguientes:

```
push    hl                      ; Preserva HL
ld      hl, firesTop            ; Apunta HL al máximo de disparos
ld      c, (hl)                 ; Lo carga en C
pop     hl                      ; Recupera el valor de HL
cp      c                       ; Compara el máximo de disparos con los activos
```

Con esto ya controlamos hasta donde llegan las naves enemigas por abajo, y el número de disparos simultáneos que puede haber, pero necesitamos una rutina que cambie los valores de ***enemiesTop*** y ***firesTop*** en función de la dificultad seleccionada.

Seguimos en Game.asm, localizamos la rutina ***Sleep*** e implementamos justo por encima de ella:

```
SetHardness:
ld      hl, enemiesTopB
ld      (hl), ENEMY_TOP_B
ld      a, (hardness)
cp      $03
jr      nc, setHardness_Fire
inc     (hl)
```

Apuntamos HL al tope de la posición de los enemigos por abajo, ***LD HL, enemiesTopB***, actualizamos con el tope por defecto, ***LD (HL), ENEMY_TOP_B***, cargamos la dificultad en A, ***LD A, (hardness)***, comprobamos si es tres, ***CP $03***, si no hay acarreo es mayor o igual que tres y saltamos, ***JR NC, setHardness_Fire***. Si hay acarreo, la dificultad es menor que tres, incrementamos en una línea el tope de la posición de los enemigos por abajo, ***INC (HL)***. *Recordad que trabajamos con las coordenadas invertidas.*

```
setHardness_Fire:
ld      hl, firesTop
ld      (hl), $01
cp      $01
ret     z
cp      $03
ret     z
ld      (hl), FIRES


ret
```

Apuntamos HL al número máximo de disparos simultáneos, ***LD HL firesTop***, lo ponemos a uno, ***LD (HL), $01***, comprobamos si estamos en dificultad uno, ***CP $01***, salimos si es así, ***RET Z***, comprobamos si la dificultad en tres, ***CP $03***, salimos si es así. Si no hemos salido, cargamos el número máximo de disparos por defecto, ***LD (HL), FIRES***, y salimos, ***RET***.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------
; Asigna la dificultad
;
; Altera el valor de los registros AF y HL
; ------------------------------------------------------------------------
```

```
SetHardness:
ld      hl, enemiesTopB         ; Apunta HL a tope por abajo de los enemigos
ld      (hl), ENEMY_TOP_B       ; Lo actualiza con el tope por defecto
ld      a, (hardness)           ; Carga la dificultad en A
cp      $03                     ; La compara con 3
jr      nc, setHardness_Fire    ; Si no hay acarreo A => 3, salta
inc     (hl)                    ; Sube una línea el tope por abajo de los enemigos
setHardness_Fire:
ld      hl, firesTop            ; Apunta HL al máximo de disparos
ld      (hl), $01               ; Lo pone a 1
cp      $01                     ; Comprueba si la dificultad es 1
ret     z                       ; Sale si es 1
cp      $03                     ; Comprueba si la dificultad es 3
ret     z                       ; Sale si es 3
ld      (hl), FIRES             ; Carga los disparos máximos por defecto


ret
```

Ha llegado el momento de probar si la selección de dificultad se comporta como pretendemos. Vamos a Main.asm, localizamos la etiqueta ***Main_start*** y la línea ***CALL PrintFirstScreen***. Justo después de está línea incluimos la llamada a la asignación de la dificultad:

```
call    SetHardness                ; Asigna la dificultad
```

Ya solo queda uno de los aspectos que señalamos al principio; si el nivel de dificultad seleccionado es el cuatro, cada nivel lo empezamos con cinco vidas.

Seguimos en Main.asm, localizamos la etiqueta ***Main_restart*** y la línea ***JR Z, Win***. Justo debajo de esta línea vamos a implementar el último aspecto de la dificultad:

```
ld      a, (hardness)
cp      $04
jr      nz, main_restartCont
ld      a, $05
ld      (livesCounter), a
main_restartCont:
```

Cargamos la dificultad en A, ***LD A, (hardmess)***, comprobamos si es cuatro, ***CP $04***, y saltamos si no lo es, ***JR NZ, main_restartCont***.

Si no hemos saltado, cargamos cinco en A, ***LD A, $05***, y actualizamos el número de vidas en memoria para empezar cada nivel con cinco, ***LD (livesCounter), A***.

El aspecto final de la rutina es el siguiente:

```
Main_restart:
ld      a, (levelCounter)           ; Carga el número de nivel en A
cp      $1e                         ; Comprueba si es el 31 (tenemos 30)
jr      z, Win                      ; Si es el 31 salta, ¡VICTORIA!

ld      a, (hardness)               ; Carga la dificultad en A
cp      $04                         ; Comprueba si es 4
jr      nz, main_restartCont        ; Si no es 4, salta
ld      a, $05                      ; Carga 5 en A
ld      (livesCounter), a           ; Pone cinco vidas

main_restartCont:
call    FadeScreen                  ; Hace el fundido de la pantalla
call    ChangeLevel                 ; Cambia de nivel
call    PrintFrame                  ; Pinta el marco
call    PrintInfoGame               ; Pinta los títulos de información
call    PrintShip                   ; Pinta la nave
call    PrintInfoValue              ; Pinta la información
call    PrintEnemies                ; Pinta los enemigos
call    ResetEnemiesFire            ; Reinicia los disparos de los enemigos

; Retardo
call    Sleep                       ; Produce un retardo
jr      Main_loop                   ; Bucle principal
```

Compilamos, cargamos en el emulador y comprobamos que los distintos niveles de dificultad se comportan tal y como lo hemos definido.

Mute

La implementación de activar o desactivar la música es relativamente sencilla. Vamos a Main.asm y añadimos un nuevo comentario a la etiqueta ***flags***:

```
; Bit 5 -> mute                             0 = No, 1 = Sí
```

En el bit cinco de flags vamos a indicar si el mute está o no activo.

Ahora tenemos que implementar la activación o desactivación de este bit. Localizamos la etiqueta ***Main_loop***, y justo debajo de ella añadimos la nueva implementación:

```
rst     $38
ld      hl, FLAGS_KEY
set     $03, (hl)
```

```
bit     $05, (hl)
jr      z, main_loopCheck
```

Actualizamos las variables del sistema, ***RST $38***, apuntamos HL a los indicadores del teclado, ***LD HL, FLAGS_KEY***, ponemos la entrada en modo L, ***SET $03, (HL)***, comprobamos si se ha pulsado alguna tecla, ***BIT $05, (HL)***, y si no se ha pulsado saltamos.

```
res     $05, (hl)
ld      a, (LAST_KEY)
cp      'M'
jr      z, main_loopMute
cp      'm'
jr      nz, main_loopCheck
```

Ponemos el bit cinco a cero para futuras inspecciones, ***RES $05, (HL)***, cargamos en A la última tecla pulsada, ***LD A, (LAST_KEY)***, comprobamos si se ha pulsado la m mayúscula, ***CP 'M'***, saltamos si se ha pulsado, ***JR Z, main_loopMute***, comprobamos si se ha pulsado la m minúscula, ***CP 'm'***, y saltamos si no se ha pulsado, ***JR NZ, main_loopCheck***.

```
main_loopMute:
ld      a, (flags)
xor     $20
ld      (flags), a


main_loopCheck:
```

Si se ha pulsado la m, ya sea mayúscula o minúscula, cargamos los indicadores en A, ***LD A, (flags)***, invertimos el valor del bit cinco, ***XOR $20***, y actualizamos el valor en memoria, ***LD (flags), A***. Finalmente, incluimos la etiqueta a la que saltamos y que no existía, ***main_loopCheck***.

Es buen momento para recordar como actúa ***XOR*** a nivel de bit:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

Si los dos bit son iguales el resultado es cero, si son distintos el resultado es 1. Al hacer ***XOR $20***, si el bit cinco está a uno, el resultado es cero, si está a cero el resultado es uno, el resto de bits se quedan como estaban.

El aspecto final del inicio de la rutina ***Main_loop*** es el siguiente:

```
; Bucle principal
Main_loop:
rst     $38                         ; Actualiza las variables de sistema
```

```
ld      hl, FLAGS_KEY               ; Carga en HL la dirección de los indicadores del
teclado
set       $03, (hl)                ; Pone entrada en modo L
bit     $05, (hl)                  ; Comprueba si se ha pulsado una tecla
jr      z, main_loopCheck          ; Si no se ha pulsado, salta
res     $05, (hl)                  ; Es necesario poner el bit a 0 para futuras
inspecciones
ld      a, (LAST_KEY)              ; Carga en A la última tecla pulsada
cp      'M'                        ; Comprueba si ha pulsado la M
jr      z, main_loopMute           ; Si se ha pulsado la M, salta
cp      'm'                        ; Comprueba si ha pulsado la m
jr      nz, main_loopCheck         ; Si no se ha pulsado la m, salta
main_loopMute:
ld      a, (flags)                 ; Carga en A los indicadores
xor     $20                        ; Invierte el bit 5 (mute)
ld      (flags), a                 ; Actualiza el valor en memoria


main_loopCheck:
call    CheckCtrl                  ; Comprueba la pulsación de los controles
call    MoveFire                   ; Mueve el disparo
```

Por último, localizamos la etiqueta ***GameOver***, vemos que la línea de arriba es ésta:

```
jr      Main_loop                  ; Bucle principal
```

Sustituimos ***JR*** por ***JP*** ya que al haber añadido varias líneas, con ***JR*** nos daría un error de salto fuera de rango.

Ahora que ya activamos o desactivamos el bit del mute al pulsar la tecla M, vamos a tenerlo en cuenta para hacer sonar, o no, la música. Vamos a Int.asm, localizamos la etiqueta ***Isr_sound*** y, justo debajo de ella, añadimos la líneas siguientes:

```
bit     $05, (hl)           ; Evalúa si el bit 5 (mute) está activo
jr      nz, Isr_end         ; Si lo está, salta
```

Cuando llegamos a ***Isr_soung*** HL apunta a ***flags*** de Main.asm. Evaluamos si el bit cinco (mute) está activo, ***BIT $05, (HL)***, y saltamos si es así, ***JR NZ, Isr_end***.

Es hora de comprobar si hemos implementado correctamente el mute. Compilamos, cargamos en el emulador, comenzamos la partida y verificamos que al pulsar la M (sin pulsar otra tecla a la vez) la música se silencia, si volvemos a pulsar, vuelve a sonar. Los efectos de sonido siguen sonando.

Pantalla de carga

Llegamos al punto final del desarrollo de Batalla Espacial, vamos a añadir la pantalla de carga.

Desde el inicio del tutorial llevamos arrastrando un aspecto que a la hora de incluir la pantalla de carga nos va a producir un error, ya lo vimos en PoromponPong, lo corregimos, pero otra vez he vuelto a caer en él; tenemos que cambiar la dirección de inicio de nuestro juego.

Vamos a Main.asm y cambiamos la dirección de inicio que ahora es ***ORG $5DAD***, por ***ORG $5DFD***.

Vamos al archivo Int.asm y cambiamos ***FLAGS: EQU $5DAD***, y la dejamos como ***FLAGS: EQU $5DFD***. También cambiamos ***MUSIC: EQU $5DAE***, la dejamos como ***MUSIC: EQU $5DFE***.

Si ahora compilamos y cargamos en el emulador nos dará un error, no hemos cambiado las direcciones en el cargador. En el cargador, a parte de cambiar las direcciones, vamos a meter un ***POKE*** que ya usamos en PorompomPong, y la carga a la pantalla de carga.

El aspecto final del cargador tiene que ser este:

```
  10 CLEAR 24059
  20 POKE 23739, 111: LOAD ""SCR
EEN$
  30 LOAD ""CODE : LOAD ""CODE 3
2348
  40 RANDOMIZE USR 24060
```

La pantalla de carga, que debéis descargar desde aquí y dejar en la carpeta Paso14, debe presentar este aspecto:

Ya tenemos el cargador modificado y la pantalla de carga en el directorio. Solo nos queda incluir la pantalla de carga en BatallaEspacial.tap.

Si estamos trabajando en Linux, editamos el archivo make y lo dejamos así:

```
pasmo --name Marciano --tap Main.asm Marciano.tap --public

pasmo --name Int --tap Int.asm Int.tap

cat Cargador.tap MarcianoScr.tap Marciano.tap Int.tap > BatallaEspacial.tap
```

Si trabajamos con Windows, editamos el archivo make.bat y lo dejamos así:

```
pasmo --name Marciano --tap Main.asm Marciano.tap --public

pasmo --name Int --tap Int.asm Int.tap
```

```
copy Cargador.tap + MarcianoScr.tap + Marciano.tap + Int.tap BatallaEspacial.tap
```

Como podéis observar, hemos añadido MarcianoScr.tap entre Cargador.tap y Marciano.tap.

Ejecutamos make o make.bat, cargamos en el emulador y hemos terminado, a no ser que vosotros queráis cambiar algo…

Conclusión

En este capítulo hemos implementado la selección del nivel de dificultad, la posibilidad de silenciar la música y hemos añadido una pantalla de carga, finalizando así nuestro juego, pero no el tutorial. Todavía nos queda un último capítulo, en el que veremos como configurar algunos aspectos de ZEsarUX y como depurar.

0x0F Depuración

Durante todo el tutorial, y anteriormente en el tutorial de PorompomPong, he venido utilizando ZEsarUX, un emulador desarrollado por César Hernández y que podéis obtener aquí; César está añadiendo nuevas funcionalidades constantemente.

También cabe señalar que, aunque hemos utilizado ZEsarUX como emulador de ZX Spectrum, es capaz de emular muchas máquinas y hacer muchas otras cosas.

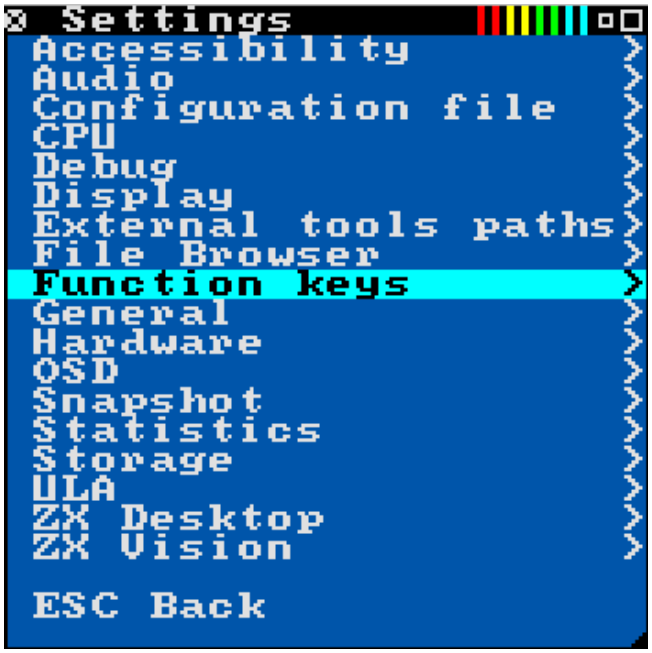
En este capítulo, no vamos a crear carpetas, ni a copiar archivos, tampoco vamos a generar código, en este capítulo vamos a profundizar en el uso de ZXesarUX.

El primer paso es personalizar ZEsarUX a nuestro gusto. Este emulador dispone de muchas opciones que puedes configurar, pero nos vamos a centrar en las que en principio me han parecido más interesantes de cara al uso que le damos en el tutorial; os animo a que investiguéis vosotros por vuestra cuenta.

Personalización

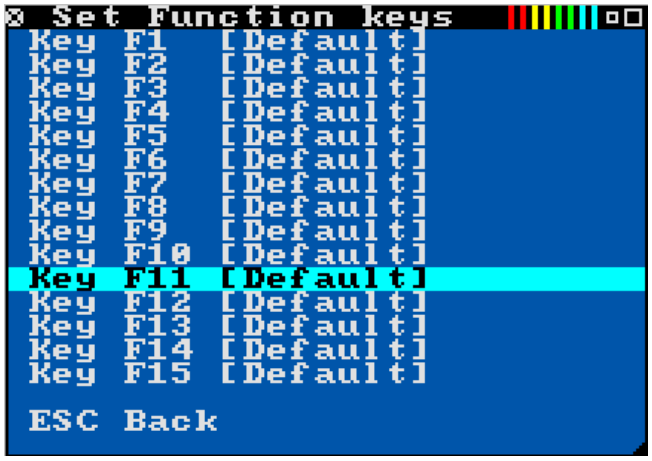
Para personalizar ZEsarUX, accedemos al menú pulsado F5 y luego hacemos clic en Settings.

Lo primero que vamos a hacer es configurar dos teclas de función, una para reiniciar la máquina y otra para entrar en depuración. Dentro de Settings accedemos a la opción Function Keys.



Vamos a configurar la tecla F11 para que abra la ventana de Debug, y la tecla F12 para reiniciar la máquina.

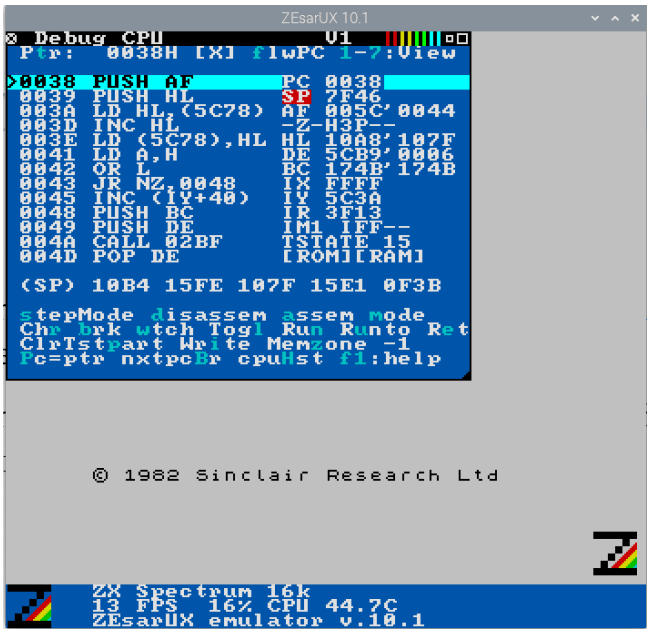
Una vez dentro de Function Keys, seleccionamos la Tecla F11, bien haciendo clic o pulsando Enter.



Una vez seleccionada la tecla, se nos muestran las opciones que podemos elegir; seleccionamos DebugCPU. Repetimos la operación para la tecla F12, pero en este caso seleccionamos Reset.

Ahora tenemos que probar las teclas, podemos salir del menú pulsando la tecla Esc hasta que se cierra completamente, o pulsamos F5 para ir al menú principal y luego Esc.

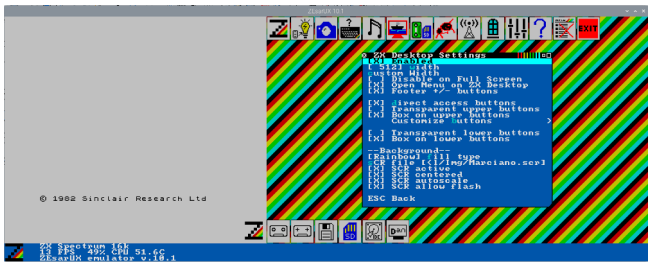
Una vez cerrado el menú, comprobamos si pulsando F12 se reinicia la máquina, y si al pulsar F11 se abre la ventana de Debug.



En mi caso, cuando depuro, suelo tener visible la ventana de Debug y la ventana en la que se muestra la memoria, por lo que se puede deducir que nos falta espacio.

Seguimos en Settings, seleccionamos ZX Desktop y seleccionamos la primera opción, ***Enabled***. Una vez seleccionada se muestran más opciones de las cuales nos interesa la primera, ***width***. Hacemos click sobre ella hasta que el valor sea 512 (u otro que más nos convenga); también podemos poner este valor directamente si seleccionamos ***custom Width***.

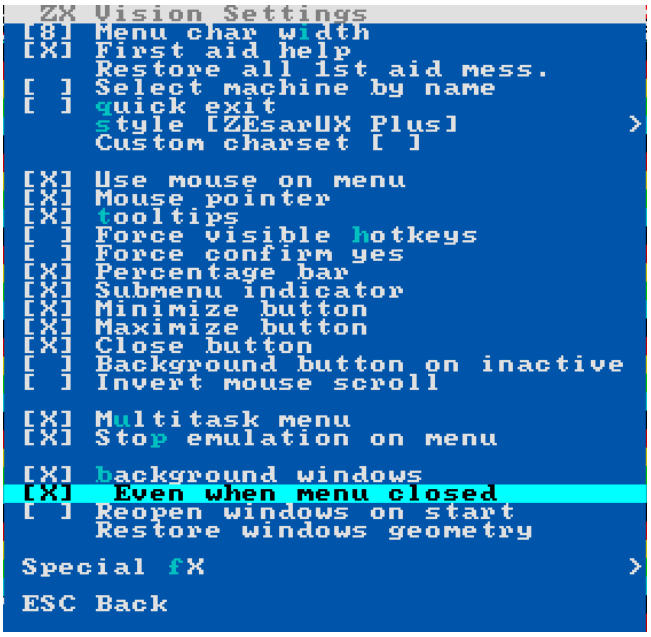
Para que el menú se abra en el escritorio debéis seleccionar la opción ***Open Menu on ZX Desktop***. Por otro lado, si queréis que el menú no se abra al hacer clic, a mi me resulta más cómodo, en Settings/General debéis deseleccionar la opción ***Clicking mouse open menu***.



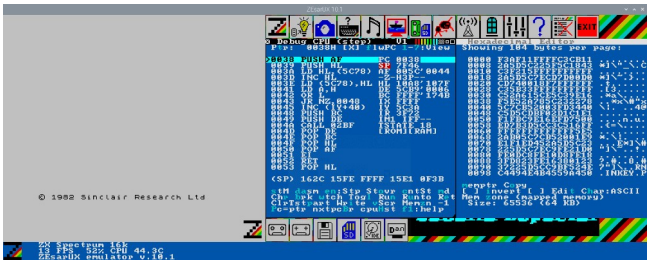
Ahora ya tenemos espacio para abrir otras ventanas y que no compartan espacio con la pantalla de nuestro querido ZX Spectrum.

Como comenté anteriormente, mientras depuro me gusta tener la ventana de depuración y la de memoria visibles. Vamos a modificar unas opciones para que podamos ver como se actualiza la memoria a medida que nuestro programa se vaya ejecutando, y para que la ejecución pare cuando el menú esté abierto.

Vamos a Settings/ZX Vision y casi al final hay que seleccionar las opciones ***Stop emulation on menu*** y ***Background Windows***. Una vez activada la segunda opción, se muestra otra opción que también tenemos que activar, ***Even when menu closed***.



Con esto ya tenemos personalizado, muy por encima, nuestro entorno. Ahora ya solo queda abrir las ventanas de depuración y el editor hexadecimal desde el menú Debug y colocarlas a nuestro gusto. Recordad que la ventana de depuración también la podemos abrir pulsado F11.



Dentro de Settings/General, en la última opción podemos cambiar el idioma, aunque en la actualidad no todos los términos están traducidos.

Por otro lado, para interactuar al 100% con las ventanas que vayamos abriendo, en el caso de que no responda abrid el menú (F5).

Depuración

Supongamos que tenemos un programa que cargamos en la dirección $8000; aseguraos de que no tenéis seleccionada la máquina de 16K o no funcionará.

```
org     $8000


Inicio:
ld      hl, $4000
ld      de, $8100
ld      a, $ff
```

```
Bucle:
ld      (hl), a
ld      (de), a
dec     a
jr      nz, Bucle

Fin:
jr  Inicio

end     $8000
```

En este programa apuntamos HL al inicio de la VideoRAM, DE a la posición $8100 y cargamos 255 ($FF) en A. Tras esto, hacemos un bucle en el que cargamos el valor de A en las posiciones a las que apuntan HL y DE. Una vez finalizado el bucle volvemos a la primera instrucción y seguimos en un bucle infinito.

El programa es muy sencillo, pero es suficiente para mostrar lo que os quiero enseñar.

Lo primero que vamos a hacer es entrar en **Debug** y poner un punto de interrupción en la dirección de memoria $8000, para que la ejecución se pare al inicio del programa.

El acceso a las distintas opciones se obtiene pulsando la tecla correspondiente a la letra que está resaltada.

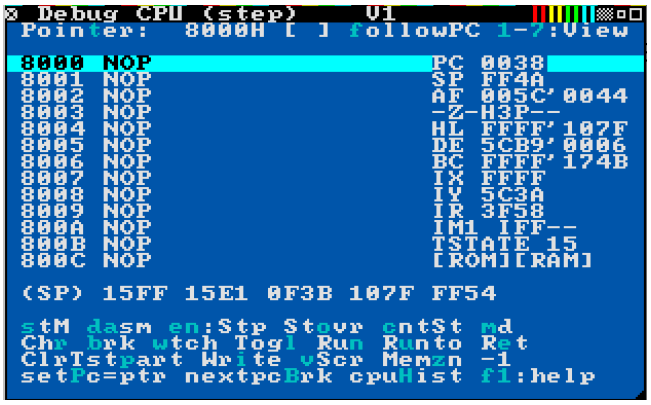
En la parte central de la pantalla, a la izquierda, vemos el código desensamblado y la dirección de memoria dónde se ensambla cada instrucción. A la derecha vemos, principalmente, el valor de los distintos registros y el estado de los flags. Justo debajo vemos los valores de la pila.

En la parte superior e inferior vemos las distintas opciones; solo vamos a ver las mínimas necesarias para poder empezar a depurar nuestros programas.

En la parte superior derecha vemos **1-7:** View. Pulsando del uno al siete se muestran las distintas vistas que tiene esta venta; por defecto se muestra la vista uno.

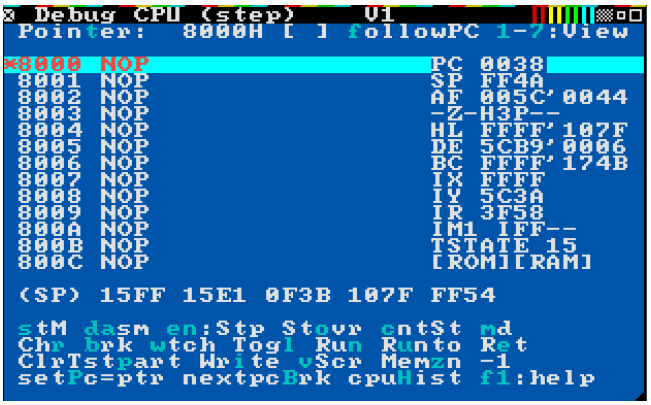
En la parte superior izquierda vemos Poin**t**er. Pulsando la tecla T se abre una ventana emergente en la que podemos especificar la dirección de memoria en la que nos queremos situar. Si ponemos valores hexadecimales, hay que usar el sufijo H.

Una vez que especificamos la dirección, pulsamos Enter y se nos muestra el desensamblado desde esta dirección.



Debido a que no hemos cargado ningún programa, todo el desensamblado que vemos es NOP ($00).

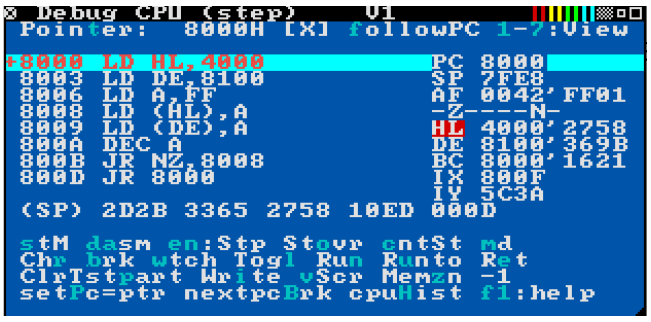
Vamos a establecer un punto de interrupción en la dirección $8000. Si nos fijamos en la cuarta opción de la segunda línea de opciones, en la parte inferior de la ventana, vemos Tog**l**; pulsando la tecla L establecemos o quitamos un punto de interrupción en la dirección de la línea seleccionada.



Ya tenemos establecido un punto de interrupción en la dirección $8000. Al cargar nuestro programa, se parará en esta dirección de memoria y entraremos a depurar. Pulsamos la tecla N (Ru**n**) para continuar y volver al Basic.

ZEsarUX muestra una ventana en la que se nos avisa de que hay un punto de interrupción, solo tendremos que pulsar una tecla para entrar en la ventana de **Debug**.

Cargamos el programa y depuramos.



Para seguir el programa, podemos ejecutar de dos formas distintas: **en**: Stp y St**o**vr. En otros entornos estos dos modos se conocen como Paso a paso por instrucciones y Paso a paso por procedimientos respectivamente.

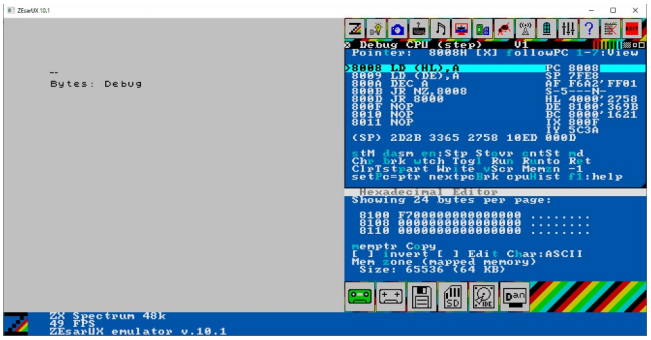
En Paso a paso por instrucciones (tecla Enter) la ejecución se hace instrucción por instrucción y cuando nos encontramos ante un ***CALL*** a una rutina, ya sea nuestra o de la ROM, entramos en la misma y tenemos la posibilidad de ejecutarla paso a paso y ver el código de la misma.

En Paso a paso por procedimientos (tecla O) la ejecución se hace instrucción por instrucción, pero cuando nos encontramos ante un ***CALL*** a una rutina, ya sea nuestra o de la ROM, se ejecuta la misma y el PC (program counter) pasa a la instrucción siguiente al ***CALL***; esto hay que tenerlo muy en cuenta en los bucles.

En nuestro programa, cuando PC está en la instrucción ***JR NZ, 8008*** si pulsamos la tecla Enter y A no vale cero, se salta a la dirección $8008. Por el contrario, si se pulsa la tecla O se ejecuta todo el bucle (por así decirlo) y PC pasa a la siguiente instrucción, ***JR 8000***. Haced pruebas y podréis ver como se comporta.

Nuestro programa, en cada iteración del bucle, carga el valor de A en las direcciones de memoria apuntadas por HL y DE. Hasta ahora solo podemos ver los distintos valores que se cargan en la dirección a la que apunta HL, ya que apunta a la primera dirección de la VideoRAM.

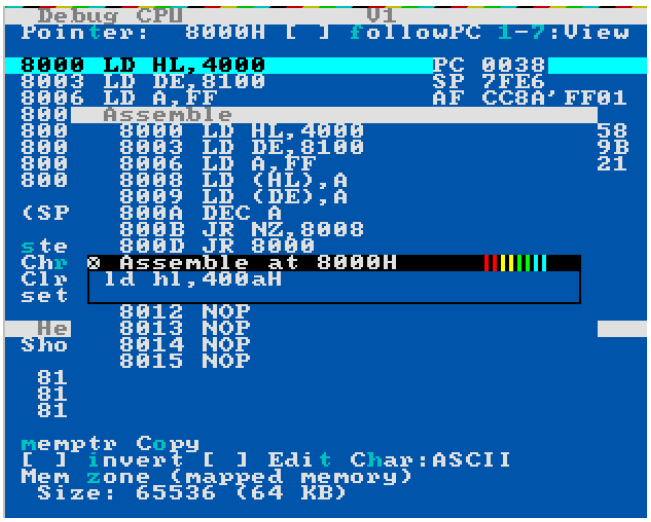
Para poder ver como va cambiando también el valor de la dirección $8100, abrimos el menú (F5), seleccionamos la ventana Hexadecimal Editor, pulsamos la tecla M (**m**emptr), especificamos la dirección $8100 y pulsamos Enter.



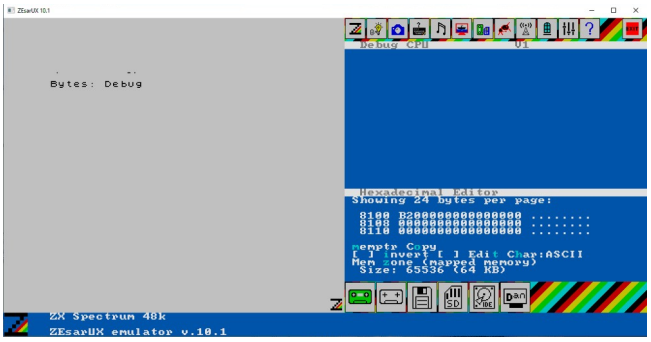
Si eliminamos el punto de interrupción y pulsamos la tecla N (Ru**n**) podremos ver como se actualizan tanto la posición de memoria $4000 como la $8100.

Con esto ya deberíais ser capaces de depurar vuestros programas, aunque ZEsarUX nos proporciona mucha más potencia, pero queda de vuestra cuenta descubrirla, aunque vamos a comentar una última opción: añadir o modificar código directamente en el depurador.

En la primera línea de opciones de la parte inferior, si pulsamos sobre la tecla S (**s**tM) se nos muestran otras opciones. Si pulsamos sobre la tecla A (**a**ssemble) se nos permite modificar la instrucción situada en la dirección de memoria seleccionada. Para salir de la ventana Assemble pulsamos la tecla Esc.



En nuestro caso, vamos a cambiar la dirección de la VideoRAM en la que estamos cargando el valor de A, lo vamos a poner en la columna diez. Si bien ZEsarUX no es sensible a mayúsculas y minúsculas, es muy importante que el sufijo de número hexadecimal sea la H mayúscula.



Tal y como podemos ver en la imagen, el valor de A se está pintando ahora en la columna diez.

Conclusión

En este capítulo hemos personalizado ZEsarUX y adquirido los conocimientos necesarios para depurar nuestro programas, aunque ZEsarUX es muy potente y nos queda mucho por descubrir.

Espero que este tutorial os haya servido para avanzar en vuestro aprendizaje de Ensamblador para ZX Spectrum.

Bibliografía

Curso de Ensamblador Z80 de Compiler Software de Santiago Romero
https://wiki.speccy.org/cursos/ensamblador/indice
Santiago Romero

Dilwyn Jones Sinclair QL Pages
http://www.dilwyn.me.uk/
http://www.dilwyn.me.uk/docs/articles/beeps.pdf

Programación avanzada del ZX Spectrum, Rutinas de la ROM y Sistema Operativo
© Steve Kramer, 1984
© Ediciones Anaya Multimedia, S.A., 1985

Ensamblador para ZX Spectrum
Pong
Juan Antonio Rubio García