

# 8 bits de PODER

En tu AMSTRAD CPC



“Las limitaciones no son un problema,  
sino una fuente de inspiración”

V42\_00

Jose Javier García Aranda



## INDICE

<b>1</b>	<b>¿POR QUÉ PROGRAMAR HOY UNA MAQUINA DE 1984? .....</b>	<b>9</b>
<b>2</b>	<b>FUNCIONES DE 8BP Y USO DE LA MEMORIA .....</b>	<b>11</b>
2.1	<b>¿QUÉ ES UNA LIBRERÍA RSX? .....</b>	<b>12</b>
2.2	<b>FUNCIONES DE 8BP .....</b>	<b>13</b>
2.3	<b>ARQUITECTURA DEL AMSTRAD CPC .....</b>	<b>14</b>
2.3.1	<i>Pila de GOSUB /RETURN .....</i>	<i>19</i>
2.3.2	<i>Un experimento para ver la ROM con Winape .....</i>	<i>20</i>
2.4	<b>MAPA DE MEMORIA DE 8BP Y OPCIONES DE ENSAMBLAJE.....</b>	<b>21</b>
<b>3</b>	<b>HERRAMIENTAS NECESARIAS .....</b>	<b>25</b>
<b>4</b>	<b>PRIMEROS PASOS CON 8BP .....</b>	<b>27</b>
4.1	<b>INSTALAR WINAPE .....</b>	<b>27</b>
4.2	<b>FAMILIARIZÁNDOTE CON WINAPE: “HOLA MUNDO” .....</b>	<b>27</b>
4.3	<b>DESCÁRGATE LA LIBRERÍA 8BP.....</b>	<b>27</b>
4.4	<b>EJECUTA LAS DEMOS .....</b>	<b>28</b>
4.5	<b>CREANDO TU PRIMER PROGRAMA CON 8BP .....</b>	<b>29</b>
4.6	<b>CREA TU .DSK CON TU JUEGO 8BP .....</b>	<b>32</b>
<b>5</b>	<b>PASOS QUE DEBES DAR PARA HACER UN JUEGO.....</b>	<b>33</b>
5.1	<b>ESTRUCTURA EN DIRECTORIOS DE TU PROYECTO.....</b>	<b>33</b>
5.2	<b>TU JUEGO EN SÓLO 3 FICHEROS.....</b>	<b>34</b>
5.3	<b>CREAR UN DISCO O UNA CINTA CON TU JUEGO .....</b>	<b>36</b>
5.3.1	<i>Hacer un disco.....</i>	<i>36</i>
5.3.2	<i>Hacer una cinta con Winape .....</i>	<i>36</i>
5.3.3	<i>Hacer una cinta fácilmente con CPCDiskXP, 2cdt y tape2wav .....</i>	<i>38</i>
5.3.4	<i>Solución de problemas con LOAD y MEMORY.....</i>	<i>39</i>
<b>6</b>	<b>ENSAMBLADO DE LA LIBRERÍA, MÚSICA Y GRÁFICOS .....</b>	<b>42</b>
6.1	<b>MAKE_ALL.ASM .....</b>	<b>43</b>
6.2	<b>ESTRUCTURA DEL FICHERO DE IMÁGENES .....</b>	<b>45</b>
6.3	<b>ESTRUCTURA DEL FICHERO DE SECUENCIAS DE ANIMACIÓN .....</b>	<b>45</b>
6.4	<b>ESTRUCTURA DEL FICHERO DE RUTAS .....</b>	<b>46</b>
6.5	<b>ESTRUCTURA DEL FICHERO DE MAPA DEL MUNDO .....</b>	<b>46</b>
<b>7</b>	<b>CICLO DE JUEGO.....</b>	<b>47</b>
7.1	<b>COMO MEDIR LOS FPS DE TU CICLO DE JUEGO .....</b>	<b>47</b>
<b>8</b>	<b>SPRITES .....</b>	<b>49</b>
8.1	<b>EDITAR SPRITES CON SPEDIT Y ENSAMBLARLOS.....</b>	<b>49</b>
8.2	<b>IMPRIMIR UN SPRITE .....</b>	<b>54</b>
8.3	<b>SPRITE FLIPPING .....</b>	<b>55</b>
8.4	<b>SPRITES CON SOBREESCRITURA .....</b>	<b>57</b>
8.4.1	<i>Uso de sobreescritura para mejorar los solapes de sprites .....</i>	<i>61</i>
8.4.2	<i>Sobreescritura con 4 colores de fondo .....</i>	<i>62</i>
8.4.3	<i>Sobreescritura en MODE 1 .....</i>	<i>63</i>
8.4.4	<i>Cómo pintar sprites “por detrás” del fondo .....</i>	<i>64</i>
8.4.5	<i>Cómo usar más colores con sobreescritura .....</i>	<i>65</i>
8.5	<b>SPRITES CON IMÁGENES DE FONDO .....</b>	<b>67</b>

8.6	TABLA DE ATRIBUTOS DE SPRITES .....	69
8.7	IMPRESIÓN DE TODOS LOS SPRITES Y ORDENADOS .....	74
8.8	COLISIONES ENTRE SPRITES .....	76
8.9	AJUSTE DE LA SENSIBILIDAD DE LA COLISIÓN DE SPRITES .....	78
8.10	QUIÉN COLISIONA Y CON QUIÉN: COLSPALL.....	79
8.10.1	<i>Cómo programar un disparo múltiple usando COLSPALL.....</i>	80
8.10.2	<i>Quien colisiona cuando hay varios solapes .....</i>	81
8.10.3	<i>Uso avanzado del byte de status en colisiones.....</i>	83
8.11	TABLA DE SECUENCIAS DE ANIMACIÓN .....	84
8.12	SECUENCIAS DE ANIMACIÓN ESPECIALES.....	85
8.12.1	<i>Secuencias de muerte.....</i>	86
8.12.2	<i>Secuencias de fin .....</i>	87
8.12.3	<i>Secuencias encadenadas.....</i>	87
8.12.4	<i>Macrosecuencias de animación.....</i>	87
<b>9</b>	<b>TU PRIMER JUEGO SENCILLO .....</b>	<b>91</b>
9.1	AHORA, A SALTAR! BOING, BOING!! .....	91
<b>10</b>	<b>JUEGOS DE PANTALLAS: LAYOUT O “TILE MAP” .....</b>	<b>95</b>
10.1	DEFINICIÓN Y USO DEL LAYOUT .....	95
10.2	EJEMPLO DE JUEGO CON LAYOUT.....	98
10.3	CÓMO ABRIR UNA COMPUERTA EN EL LAYOUT .....	100
10.4	UN COMECOCOS: LAYOUT CON FONDO .....	101
10.5	CÓMO AHORRAR MEMORIA EN TUS LAYOUTS .....	103
<b>11</b>	<b>PROGRAMACIÓN AVANZADA Y “LÓGICAS MASIVAS” .....</b>	<b>105</b>
11.1	MEDICIÓN DE LA VELOCIDAD DE LOS COMANDOS.....	105
11.2	HAZ UNA SOLA LÓGICA PARA GOBERNAR TODAS TUS PANTALLAS .....	112
11.3	TÉCNICA DE “LÓGICAS MASIVAS” .....	113
11.3.1	<i>Mueve 32 sprites con lógicas masivas.....</i>	114
11.3.2	<i>Ejecución alternada y periódica en cascada.....</i>	115
11.3.3	<i>Ejemplo sencillo de lógica masiva .....</i>	117
11.3.4	<i>Movimiento “en bloque” de escuadrones .....</i>	118
11.3.5	<i>Técnica de lógicas masivas en juegos tipo “pacman”.....</i>	119
11.3.6	<i>Reducción de número de instrucciones en ciclo de juego.....</i>	121
11.3.7	<i>Rutas que aceleran el juego manipulando el estado .....</i>	125
11.3.8	<i>Enrutando sprites con “lógicas masivas” .....</i>	125
<b>12</b>	<b>TRAYECTORIAS COMPLEJAS: COMANDO ROUTEALL.....</b>	<b>129</b>
12.1	COLOCA A UN SPRITE EN MITAD DE UNA RUTA : ROUTESP.....	131
12.2	CREACIÓN DE RUTAS AVANZADAS .....	133
12.2.1	<i>Cambios de estado forzados desde rutas.....</i>	133
12.2.2	<i>Cambios de secuencia forzados desde rutas .....</i>	135
12.2.3	<i>Cambios de imagen forzados desde rutas .....</i>	135
12.2.4	<i>Cambios de ruta forzados desde rutas .....</i>	139
12.2.5	<i>Cambios de ruta forzados desde BASIC.....</i>	139
12.2.6	<i>Animación forzada desde rutas .....</i>	141
12.2.7	<i>Como construir rutas “dinámicas” (no predefinidas) .....</i>	141
12.2.8	<i>Programación de rutas que incluyen patrones.....</i>	142
12.2.9	<i>Tipología de rutas.....</i>	143

<b>13</b>	<b>MOVIMIENTO SUAVE DE MEDIO BYTE .....</b>	<b>145</b>
<b>14</b>	<b>JUEGOS CON SCROLL.....</b>	<b>147</b>
14.1	STARS: SCROLL DE ESTRELLAS O TIERRA MOTEADA .....	147
14.2	SCROLL USANDO MOVERALL Y/O AUTOALL .....	150
14.3	TÉCNICA DEL “MANCHADO” .....	152
14.4	MAP2SP: SCROLL BASADO EN UN MAPA DEL MUNDO.....	155
14.4.1	<i>Mapa del mundo (Map Table) .....</i>	157
14.4.2	<i>Uso de la función MAP2SP .....</i>	158
14.4.3	<i>Ejemplo de fichero de fases.....</i>	161
14.4.4	<i>Colisión de enemigos con mapa .....</i>	163
14.4.5	<i>Imágenes de fondo en tu scroll .....</i>	164
14.5	SCROLL PARALLAX .....	165
14.6	ACTUALIZACIÓN DINÁMICA DEL MAPA:  UMAP .....	166
14.7	ANIMACIÓN Y SCROLL POR TINTAS: COMANDO RINK .....	168
14.7.1	<i>Carreras de coches 2D .....</i>	169
14.7.2	<i>Scroll de Ladrillos .....</i>	171
<b>15</b>	<b>JUEGOS DE PLATAFORMAS.....</b>	<b>173</b>
<b>16</b>	<b>HORDAS DE ENEMIGOS EN JUEGOS DE SCROLL .....</b>	<b>175</b>
<b>17</b>	<b>MINICARACTERES REDEFINIBLES: PRINTAT .....</b>	<b>177</b>
17.1	CREA TU PROPIO ALFABETO DE MINICARACTERES .....	178
17.2	ALFABETO POR DEFECTO PARA MODE 1 .....	179
<b>18</b>	<b>PSEUDO 3D .....</b>	<b>181</b>
18.1	PROYECCIÓN 3D .....	183
18.1.1	<i>Matemáticas de la proyección pseudo 3D.....</i>	184
18.1.2	<i>Curvas.....</i>	188
18.2	ZOOM IMAGES .....	189
18.3	USO DE SEGMENTOS .....	191
<b>19</b>	<b>MÚSICA .....</b>	<b>193</b>
19.1	EDITAR MÚSICA CON WYZ TRACKER.....	193
19.2	ENSAMBLAR LAS CANCIONES .....	194
19.3	QUÉ HACER SI NO TE CABE LA MÚSICA EN 1400 BYTES .....	195
<b>20</b>	<b>PROGRAMACIÓN EN C CON 8BP .....</b>	<b>197</b>
20.1	PRIMER PASO: PROGRAMA TU JUEGO BASIC.....	198
20.2	SEGUNDO PASO: TRADUCE TU CICLO DE JUEGO DE BASIC A C.....	199
20.2.1	<i>GOSUB y RETURN en C.....</i>	203
20.2.2	<i>Comunicación BASIC a C con variables BASIC.....</i>	204
20.2.3	<i>Cadenas de texto en BASIC y en C.....</i>	207
20.3	TERCER PASO: COMPILE USANDO “COMPILA.BAT” .....	208
20.4	CUARTO PASO: COMPRUEBA LOS LÍMITES DE LA MEMORIA .....	210
20.5	QUINTO PASO: LOCALIZA LA DIRECCIÓN DE LA FUNCIÓN A INVOCAR DESDE BASIC211	
20.6	SEXTO PASO: INCLUIR EN TU JUEGO.DSK EL NUEVO BINARIO.....	212
20.7	REFERENCIA DE FUNCIONES 8BP EN C .....	213
20.8	REFERENCIA DE FUNCIONES BASIC EN C (“MINIBASIC”) .....	215
<b>21</b>	<b>GUÍA DE REFERENCIA DE LA LIBRERÍA 8BP.....</b>	<b>217</b>

<b>21.1</b>	<b>FUNCIONES DE LA LIBRERÍA .....</b>	<b>217</b>
21.1.1	/3D .....	217
21.1.2	/ANIMA.....	218
21.1.3	/ANIMALL.....	219
21.1.4	/AUTO.....	220
21.1.5	/AUTOALL.....	220
21.1.6	/COLAY.....	220
21.1.7	/COLSP.....	221
21.1.8	/COLSPALL.....	223
21.1.9	/LAYOUT.....	224
21.1.10	/LOCATESP.....	226
21.1.11	/MAP2SP .....	226
21.1.12	/MOVER.....	228
21.1.13	/MOVERALL.....	228
21.1.14	/MUSIC.....	229
21.1.15	/PEEK.....	229
21.1.16	/POKE.....	230
21.1.17	/PRINTAT .....	230
21.1.18	/PRINTSP.....	231
21.1.19	/PRINTSPALL.....	232
21.1.20	/RINK.....	234
21.1.21	/ROUTEALL.....	235
21.1.22	/ROUTESP.....	237
21.1.23	/SETLIMITS.....	238
21.1.24	/SETUPSP.....	238
21.1.25	/STARS.....	240
21.1.26	/UMAP .....	242
<b>22</b>	<b>COMO HACER UNA TABLA DE PUNTUACIONES .....</b>	<b>243</b>
<b>23</b>	<b>POSIBLES MEJORAS FUTURAS A LA LIBRERÍA .....</b>	<b>245</b>
23.1	MEMORIA PARA UBICAR NUEVAS FUNCIONES .....	245
23.2	IMPRESIÓN A RESOLUCIÓN DE PÍXEL .....	245
23.3	LAYOUT DE MODE 1.....	245
23.4	CAPACIDAD FILMATION .....	245
23.5	FUNCIONES DE SCROLL POR HARDWARE .....	246
23.6	MIGRAR LA LIBRERÍA 8BP A OTROS MICROORDENADORES.....	247
<b>24</b>	<b>ALGUNOS JUEGOS HECHOS CON 8BP .....</b>	<b>249</b>
24.1	MUTANTE MONToya .....	249
24.2	ANUNNAKI, NUESTRO PASADO ALIEN .....	250
24.3	NIBIRU .....	251
24.4	FRESH FRUITS & VEGETABLES.....	252
24.5	“3D RACING ONE” .....	252
24.6	SPACE PHANTOM .....	253
24.7	FROGGER ETERNO .....	254
24.8	ERIDU: THE SPACE PORT .....	255
24.9	HAPPY MONTY .....	256
24.10	BLASTER PILOT .....	256
24.11	NOMWARS .....	257
24.12	PACO, EL HOMBRE .....	258

24.13	MINI JUEGOS .....	259
24.13.1	<i>Mini-pong</i> .....	259
24.13.2	<i>Mini-Invaders</i> .....	260
<b>25</b>	<b>APENDICE I: ORGANIZACIÓN DE LA MEMORIA DE VIDEO .....</b>	<b>261</b>
25.1	EL OJO HUMANO Y LA RESOLUCIÓN DEL CPC .....	261
25.2	LA MEMORIA DE VIDEO.....	261
25.2.1	<i>Mode 2</i> .....	261
25.2.2	<i>Mode 1</i> .....	261
25.2.3	<i>Mode 0</i> .....	262
25.2.4	<i>Memoria de la pantalla</i> .....	262
25.3	CÁLCULO DE UNA DIRECCIÓN DE PANTALLA .....	264
25.4	BARRIDOS DE PANTALLA .....	264
25.5	CÓMO HACER UNA PANTALLA DE CARGA PARA TU JUEGO .....	265
<b>26</b>	<b>APENDICE II: LA PALETA .....</b>	<b>269</b>
<b>27</b>	<b>APENDICE III: INKEY CODES .....</b>	<b>271</b>
<b>29</b>	<b>APENDICE IV: TABLA ASCII DEL AMSTRAD CPC .....</b>	<b>273</b>
<b>30</b>	<b>APENDICE V: ALGUNOS EFECTOS DE SONIDO .....</b>	<b>275</b>
<b>31</b>	<b>APENDICE VI: RUTINAS INTERESANTES DEL FIRMWARE .....</b>	<b>277</b>
<b>32</b>	<b>APENDICE VII: TABLA DE ATRIBUTOS DE SPRITES.....</b>	<b>279</b>
<b>33</b>	<b>APENDICE VIII: MAPA DE MEMORIA DE 8BP .....</b>	<b>281</b>
<b>34</b>	<b>APENDICE IX: COMANDOS DISPONIBLES 8BP.....</b>	<b>283</b>
<b>35</b>	<b>APÉNDICE X: OPCIONES DE ENSAMBLAJE DE 8BP.....</b>	<b>285</b>
<b>36</b>	<b>APENDICE XI: CORRESPONDENCIAS RSX/CALL .....</b>	<b>287</b>
<b>37</b>	<b>APENDICE XII: FUNCIONES 8BP EN C.....</b>	<b>289</b>
<b>38</b>	<b>APENDICE XIII: MINIBASIC EN C .....</b>	<b>291</b>



# 1 ¿Por qué programar hoy una maquina de 1984?

Porque las limitaciones no son un problema sino una fuente de inspiración.

Las limitaciones, ya sean de una maquina o de un ser humano, o en general de cualquier recurso disponible estimulan nuestra imaginación para poder superarlas. El AMSTRAD, una maquina de 1984 basada en el microprocesador Z80, posee una reducida memoria (64KB) y una reducida capacidad de procesamiento, aunque sólo si lo comparamos con los ordenadores actuales. Esta máquina es en realidad un millón de veces más rápida que la que construyó Alan Turing para descifrar los mensajes de la maquina enigma en 1944. Como todos los ordenadores de los años 80, el AMSTRAD CPC arrancaba en menos de un segundo, con el intérprete BASIC dispuesto a recibir comandos de usuario, siendo el BASIC el lenguaje con el que los programadores aprendían y hacían sus primeros desarrollos. El BASIC del AMSTRAD era particularmente rápido en comparación al de sus competidores. ¡Y estéticamente era un ordenador muy atractivo!



Fig. 1. El mítico AMSTRAD modelo CPC464

En cuanto al microprocesador Z80 ni siquiera es capaz de multiplicar (en BASIC puedes multiplicar, pero eso se basa en un programa interno que implementa la multiplicación mediante sumas o desplazamientos de registros), tan solo puede hacer sumas, restas y operaciones lógicas. A pesar de ello era la mejor CPU de 8 bit y tan sólo constaba de 8500 transistores, a diferencia de otros procesadores como el M68000 cuyo nombre precisamente le viene de tener 68000 transistores.

CPU	Número de transistores	MIPS (millones de instrucciones por segundo)	Ordenadores y consolas que lo incorporan
6502	3.500	0.43 @ 1Mhz	COMMODORE 64, NES, ATARI 800...
Z80	8.500	0.58 @ 4Mhz	AMSTRAD, COLECOVISION, SPECTRUM, MSX...
Motorola 68000	68.000	2.188 @ 12.5 Mhz	AMIGA, SINCLAIR QL, ATARI ST...
Intel 386DX	275.000	2.1 @ 16Mhz	PC
Intel 486DX	1.180.000	11 @ 33 Mhz	PC
Pentium	3.100.000	188 @ 100Mhz	PC
ARM1176		4744 @ 1Ghz (1186 por core)	Raspberry pi 2, Nintendo 3DS, Samsung galaxy, ...
Intel i7 (5 <sup>a</sup> generación)	2.600.000.000	238310 @ 3Ghz (¡casi 500.000 veces más rápido que un Z80 )	PC
AMD Ryzen 9 3950X (16 core)	3.800.000.000	749070 @ 4.8Ghz (1.3 millones de veces mas rápido que Z80)	PC

*Tabla 1Comparativa de MIPS*

Ello hace que programarlo sea extremadamente interesante y estimulante para lograr resultados satisfactorios. Toda nuestra programación debe ir orientada a reducir complejidad computacional espacial (memoria) y temporal (operaciones), obligándonos a inventar trucos, artimañas, algoritmos, etc., y haciendo de la programación una aventura apasionante. Es por ello, que la programación de máquinas de baja capacidad de procesamiento es un concepto atemporal, no sujeto a modas ni condicionado por la evolución de la tecnología.

Todo el código de este libro, incluida la librería para que hagas tus propios juegos o para que hagas contribuciones a la librería, lo puedes encontrar en el proyecto GitHub “8BP”, en esta URL. Basta con que te descargues el zip (en un capítulo posterior te daré un paso a paso)

<https://github.com/jjaranda13/8BP>

También existe un blog con mucha información en:

<http://8bitsdepoder.blogspot.com.es>

Y un canal de youtube:

[https://www.youtube.com/channel/UCThvesOT-jLU\\_s8a5Z0jBFA](https://www.youtube.com/channel/UCThvesOT-jLU_s8a5Z0jBFA)

## 2 Funciones de 8BP y uso de la memoria

La librería 8BP no es un “motor de juegos”. Es algo intermedio entre una simple extensión de comandos BASIC y un motor de juegos.

Los motores de juegos como el game-maker, el AGD (Arcade Game Designer), el Unity, y muchos otros, limitan en cierta medida la imaginación del programador, obligándole a usar unas determinadas estructuras, a programar en lenguaje limitado de script la lógica de un enemigo, a definir y enlazar pantallas de juego, etc.



Fig. 2 Motores de juego versus 8BP

La librería 8BP es diferente. Es una librería capaz de ejecutar deprisa aquello que el BASIC no puede hacer. Cosas como imprimir sprites a toda velocidad, o mover bancos de estrellas por la pantalla, son cosas que el BASIC no puede hacer y 8BP lo consigue. ¡Y ocupa sólo 8 KB!!

BASIC es un lenguaje interpretado. Eso significa que cada vez que el ordenador ejecuta una línea de programa debe primero verificar que se trata de un comando válido, comparando la cadena de caracteres del comando con todas las cadenas de comandos válidos. A continuación, debe validar sintácticamente la expresión, los parámetros del comando e incluso los rangos permitidos para los valores de dichos parámetros. Además, los parámetros los lee en formato texto (ASCII) y debe convertirlos a datos numéricos. Finalizada toda esta labor, procede con la ejecución. Pues bien, toda esa labor que se realiza en cada instrucción es la que diferencia un programa compilado de un programa interpretado como los escritos en BASIC.

Dotando al BASIC de los comandos proporcionados por 8BP, es posible hacer juegos de calidad profesional, ya que la lógica del juego que programas puede ejecutarse en BASIC mientras que las operaciones intensivas en el uso de CPU como imprimir en pantalla o detectar colisiones entre sprites, etc. son llevadas a cabo en código máquina por la librería. Sin embargo, no todo es facilidad y ausencia de problemas. Aunque la librería 8BP te va a proporcionar funciones muy útiles en videojuegos, deberás usarla con cautela pues cada comando que invoques atravesará la capa de análisis sintáctico del BASIC, antes de llegar al inframundo del código máquina donde se encuentra la función, por lo que el rendimiento nunca será el óptimo. Deberás ser astuto y ahorrar instrucciones, medir los tiempos de ejecución de instrucciones y trozos de tu programa y pensar estrategias para ahorrar tiempo de ejecución. Toda una aventura de ingenio y diversión. Aquí aprenderás como hacerlo e incluso te presentaré una técnica a la que he llamado “lógicas masivas” que te permitirá acelerar tus juegos a límites que quizás considerabas imposibles.

Además de la librería, tienes a tu disposición un sencillo pero completo editor de sprites y gráficos y una serie de herramientas magníficas que te permitirán disfrutar en el siglo XXI de la aventura de programar un microordenador.

El siguiente “simpático” dibujo esquematizaba las capacidades que 8BP proporcionaba y fue utilizado en una feria “retro”. Actualmente dispone de mas capacidades.

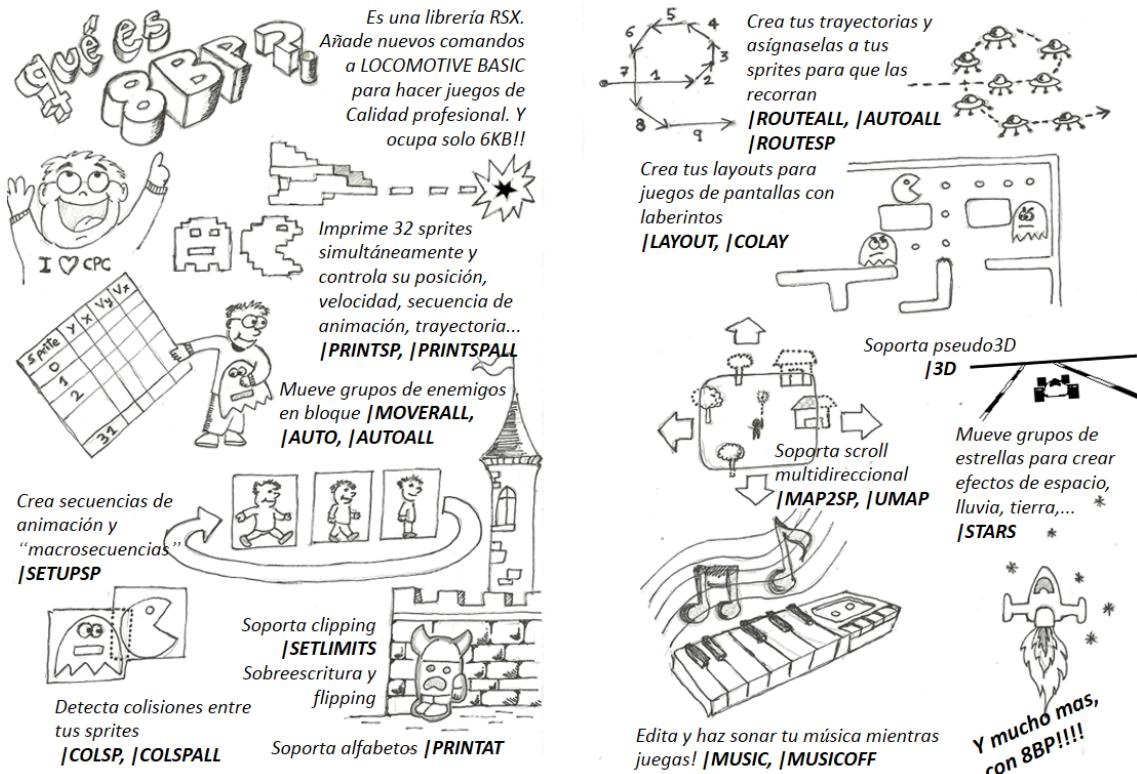


Fig. 3 Resumen de 8BP (actualmente tiene mas comandos)

## 2.1 ¿Qué es una librería RSX?

RSX es el acrónimo de Resident System eXtensions. Las librerías como 8BP que proporcionan comandos para extender el BASIC se les llama librerías RSX.

En el CPC6128, algunos de los comandos que se utilizan para manejar la unidad de disco son comandos RSX que vienen preinstalados, tales como |TAPE, |DISC, |A, |B, |CPM y otros. Si esta funcionalidad no existiese, cada rutina de 8BP habría que invocarla con un CALL <dirección>, por lo que la existencia de RSX hace más entendibles los programas.

No todo es paz y armonía. Usar RSX es más lento que usar CALL directamente y además si declaramos 10 comandos nuevos en una librería, el décimo comando puede tardar 1ms más en empezar a ejecutarse que el primero. La librería 8BP tiene 27 comandos y el último comienza a ejecutarse 2ms mas tarde por encontrarse en el último lugar de la lista. Es uno de los problemas de estar bajo el intérprete BASIC.

8BP compensa este problema creando los comandos de uso más frecuente al principio de la lista, y dejando para el final los menos frecuentes. Como pronto deducirás, el comando más frecuentemente usado en 8BP es |PRINTSPALL, el cual imprime todos los sprites en pantalla. Dicho comando es, por consiguiente, el primero de la lista.

## 2.2 Funciones de 8BP

Tras cargar la librería con el comando: LOAD “8BP.BIN” e invocar desde BASIC la función \_INSTALL\_RSX (definida en código máquina) mediante el comando BASIC:  
**CALL &6b78**

Dispondrás de los siguientes comandos, que aprenderás a usar con este libro

3D, <flag>, #, offsety  3D, 0	Activa el modo de proyección pseudo 3D
ANIMA, #	Cambia el fotograma de un sprite según su secuencia
ANIMALL	Cambia el fotograma de los sprites con flag animación activado (no hace falta invocarla, basta con un flag en la instrucción PRINTSPALL para que sea invocada)
AUTO, #	Movimiento automático de un sprite según su Vy,Vx
AUTOALL, <flag enrutado>	Movimiento de todos los sprites con flag de movimiento automático activo
COLAY, umbral_ascii, @colision, #  COLAY, @colision, #  COLAY, #  COLAY	Detecta la colisión con el layout y retorna 1 si hay colisión. Acepta un número variable de parámetros (siempre en el mismo orden) desde 4 hasta ninguno.
COLSP, #, @collided%  COLSP, 32, ini, fin  COLSP, 33, @collided%  COLSP, 34, dy, dx  COLSP, #	Retorna primer sprite con el que colisiona #. Se puede configurar el comando con los códigos 32,33 y 34
COLSPALL,@quien%, @conquien%  COLSPALL, colisionador  COLSPALL	Retorna quien ha colisionado (collider) y con quién ha colisionado (collided)
LAYOUT, y, x, @string\$	Imprime tira de imágenes de 8x8 y rellena map layout
LOCATESP, #, y, x	Cambia las coordenadas de un sprite (sin imprimirlo)
MAP2SP, y, x  MAP2SP, status	Crea sprites para pintar el mundo en juegos con scroll. Los sprites se crean con estado = status
MOVER, #, dy, dx	movimiento relativo de un solo sprite
MOVERALL, dy,dx  MOVERALL	Movimiento relativo de todos los sprites con flag de movimiento relativo activo
MUSIC, C, flag, canción, speed  MUSIC, flag, canción, speed  MUSIC	Comienza a sonar una melodía. Se puede desactivar el canal C para usarlo con efectos FX si se desea. Sin parámetros deja de sonar
PEEK, dir, @variable%	Lee un dato 16bit (puede ser negativo)
POKE, dir, valor	introduce un dato 16bit (que puede ser negativo)
PRINTAT, flag, y, x, @string	Imprime una cadena de “minicaracteres” redefinibles
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	imprime un solo sprite (# es su número) sin tener en cuenta byte de status. Si se especifica 32 entonces establecemos bits fondo
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode  PRINTSPALL	Imprime todos los sprites con flag de impresión activo. Si se invoca con un solo parámetro, se establece el modo de ordenamiento
RINK,tini,color1,color2,...,colorN  RINK, salto	Rota un conjunto de tintas de acuerdo a un patrón definible compuesto por cualquier número de tintas
ROUTESP, #, pasos	Hace recorrer de golpe N pasos de la ruta del sprite
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta (no hace falta invocarla, basta con flag en AUTOALL)
SETLIMITS, xmin, xmax, ymin, ymax	Define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor  SETUPSP, #, 5, Vy, Vx	Modifica un parámetro de un sprite. Si se indica el parámetro 5, se puede suministrar opcionalmente Vx
STARS, initstar, num, color, dy, dx	Scroll de un conjunto de estrellas
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Actualiza los ítems del mapa del mundo con un subconjunto de elementos de un mapa mayor

Tabla 2 Comandos disponibles en la librería 8BP

Fíjate que aparece una barra vertical al principio de cada uno por ser “extensiones” del BASIC. Algunas variables aparecen con el símbolo “%” para denotar que son números enteros (no decimales) pero si usas DEFINT para forzar que todas las variables sean enteras no necesitas el “%”.

Adicionalmente dispones de un comando experimental:

#### **|RETROTIME, fecha**

Este comando permite transformar tu CPC en una máquina del tiempo, con solo introducir la fecha de destino deseada. La única limitación del comando es que debes introducir una fecha igual o posterior a la del nacimiento del AMSTRAD CPC, Abril de 1984,

#### **|RETROTIME, “01/04/1984”**

Por favor, utiliza esta funcionalidad con precaución. Podrías crear una paradoja temporal y destruir el mundo.

Aunque de momento puedes tener cierto escepticismo respecto lo que puedes llegar a hacer con la librería 8BP, pronto descubrirás que el uso de esta librería junto con técnicas de programación avanzadas que aprenderás en este libro te permitirá hacer juegos profesionales en BASIC, algo que quizás creías imposible.

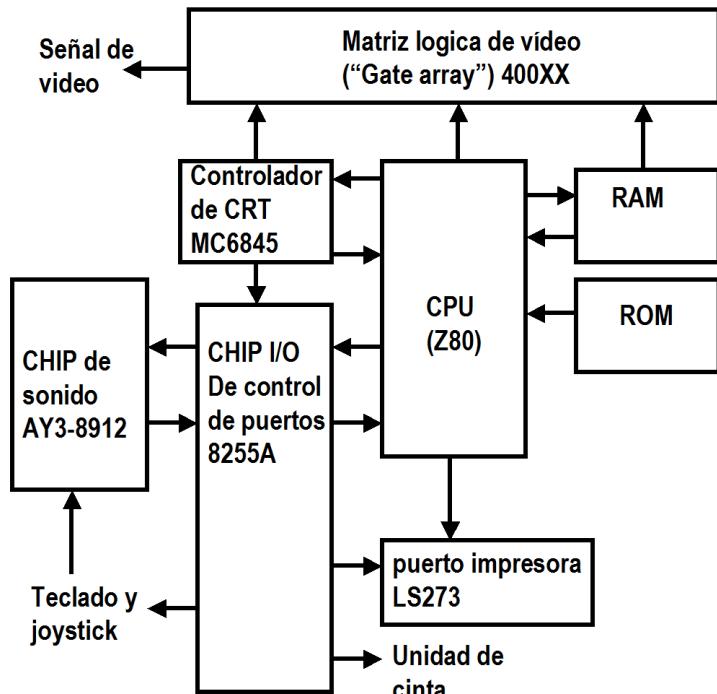
#### **Nota importante para el programador:**

La librería 8BP está optimizada para ser muy rápida. Es por ello que no chequea que hayas colocado correctamente los parámetros de cada comando, ni que tengan un valor adecuado. Si algún parámetro está mal puesto, es muy posible que el ordenador se cuelgue al ejecutar el comando. Chequear estas cosas lleva tiempo de ejecución y el tiempo es un recurso que no se puede desperdiciar, ni un milisegundo.

## **2.3 Arquitectura del AMSTRAD CPC**

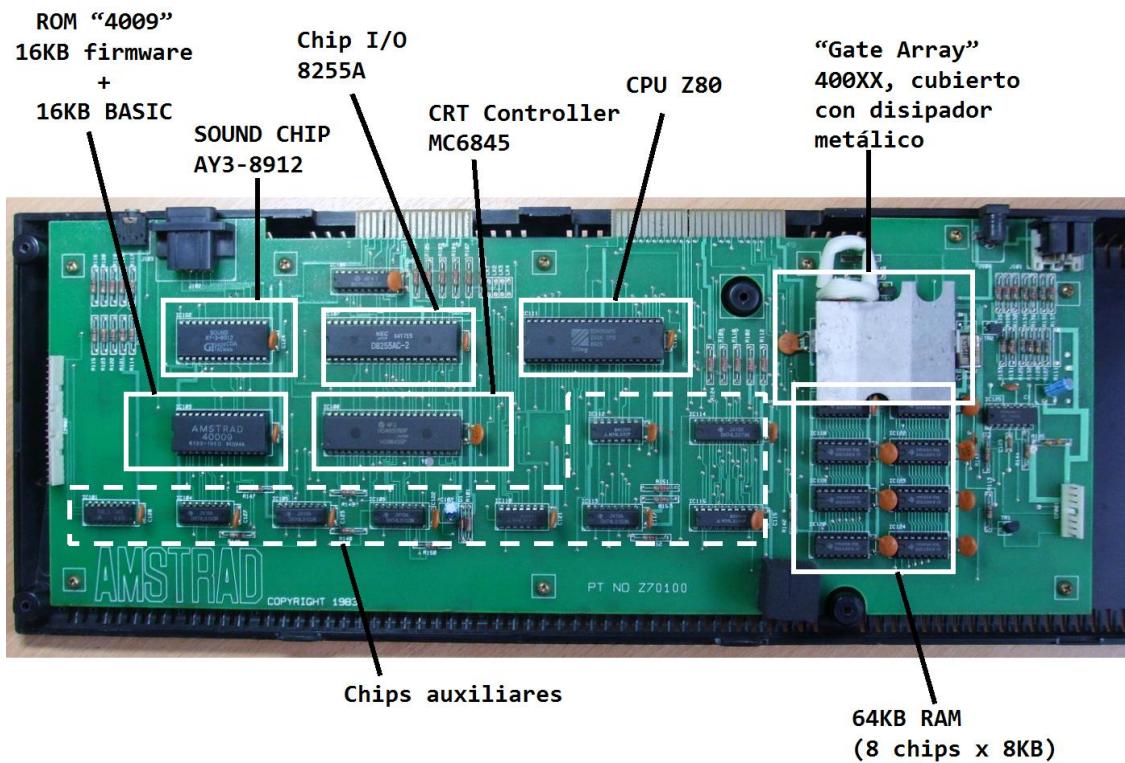
Este apartado es útil para comprender posteriormente como usa la librería 8BP la memoria.

El AMSTRAD es una computadora basada en el microprocesador Z80, funcionando a 4MHz. Como se aprecia en su diagrama de arquitectura, tanto la CPU como la matriz lógica de video (llamada “gate array”) acceden a la memoria RAM, por lo que para poder “turnarse”, los accesos a la memoria desde la CPU son retrasados, dando como resultado una velocidad efectiva de 3.3Mhz. Esto sigue siendo bastante potencia.



*Fig. 4 Arquitectura del AMSTRAD*

La memoria RAM de video es accedida por el chip “gate array” 50 veces por segundo para poder enviar una imagen a la pantalla. En ordenadores más antiguos (como el Sinclair ZX81) esta labor era encomendada al procesador, restándole aun más potencia.



*Fig. 5 Identificación de componentes en la placa*

El gate array es un chip que contiene muchas puertas lógicas, diseñado específicamente para AMSTRAD. En ZX Spectrum hay un chip similar llamado ULA (Uncommitted Logic Array – no confundir con ALU). Tanto el del amstrad como el de ZX son chips diseñados en exclusiva para estos ordenadores. En el ZX además de usarse para generar

La memoria de video, lo que vemos en la pantalla, es parte de las 64KB de memoria RAM, en concreto son las 16KB situadas en la zona superior de la memoria. La memoria se numera desde 0 hasta 65535 bytes. Pues bien, los 16KB comprendidos entre la dirección 49152 y 65535 es la memoria de video. En hexadecimal se representa como &C000 hasta &FFFF.

la imagen de video también se usaba para leer el teclado y la entrada de cassette, sin embargo, en el AMSTRAD estas funciones se realizan mediante otros chips como el 8255A.

El Z80 posee un bus de direcciones de 16bit, por lo que no es capaz de direccionar más de 64KB. Sin embargo, el Amstrad posee 64kB RAM y 32kB ROM. Para poder direccionarlas, el AMSTRAD es capaz de “conmutar” entre unos bancos y otros, de modo que, por ejemplo, si se invoca a un comando BASIC, se conmuta al banco de ROM donde se almacena el intérprete BASIC, que está solapado con las 16KB de pantalla. Este mecanismo es sencillo y efectivo.

Además de la ROM que contiene el intérprete BASIC de 16KB situado en la zona de memoria alta, hay otras 16KB de ROM situadas en la memoria baja, donde se encuentran las rutinas del firmware (lo que podría considerarse el sistema operativo de esta máquina). En total (intérprete BASIC y firmware) suman 32KB

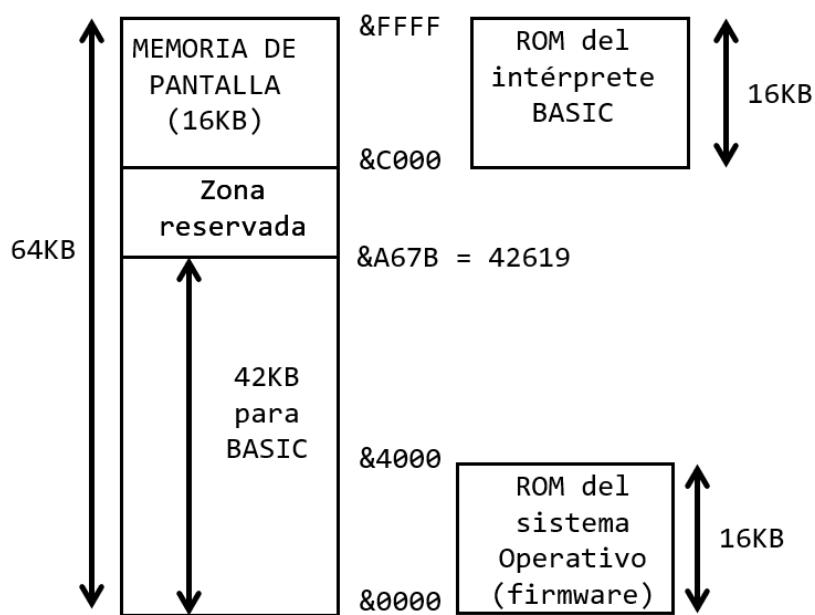


Fig. 6 Memoria del AMSTRAD

Como se aprecia en el mapa de memoria, de los 64KB de RAM, 16KB (desde &C000 hasta &FFFF) son la memoria de vídeo. Los programas en BASIC pueden ocupar desde la posición &40 (dirección 64) hasta la 42619, pues más allá hay variables del sistema. Es decir, que se dispone de unas 42KB para BASIC, tal y como podemos comprobar al imprimir la variable del sistema HIMEM (abreviatura de “High Memory”).

```
print HIMEM
42619
Ready
```

Fig. 7 Variable de sistema HIMEM

El funcionamiento del BASIC tiene en cuenta el almacenamiento del programa en direcciones crecientes desde la posición &40. Una vez en ejecución, las variables que se declaran deben ocupar espacio para almacenar los valores que toman y puesto que no pueden ocupar la misma zona donde se almacena el programa, sencillamente se empiezan a almacenar por encima de la última dirección ocupada por el listado BASIC.

En el AMSTRAD cada variable ocupa información con su nombre y su valor. Una variable numérica de tipo entero ocupará 2 bytes de memoria para el valor, pero ocupa otros tantos bytes para almacenar el nombre. Las variables reales (con decimales) ocupan 5 bytes. Cada vez que usemos una variable consumiremos memoria comenzando en la primera dirección libre por encima del listado BASIC. **Existe el peligro de que si creamos muchas variables y nuestro listado BASIC es muy largo, la pila de variables pueda llegar a devorar toda la memoria libre.** En ese caso el programa BASIC se corrompería y dejaría de funcionar. En cuanto el programa empiece a ejecutarse, las variables empezarán a comer espacio y cuando llenen toda la RAM disponible dará un error de memory full.

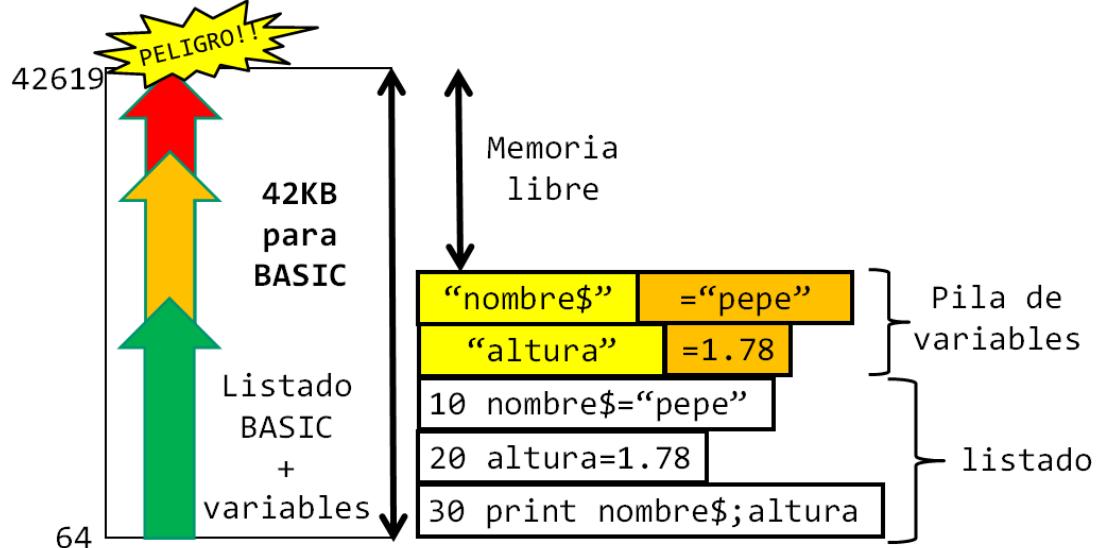


Fig. 8 Crecimiento del consumo de memoria de listado BASIC y variables

Puedes monitorizar en todo momento cuanta memoria te queda disponible con el comando **FRE(0)**, cuyo valor puedes imprimir o cargar en una variable. Puedes hacer este experimento, verás como logras consumir toda la memoria del CPC con un array de enteros:

```
print fre(0)
42209
Ready
list
10 CLEAR:DEFINT a-z
20 DIM a(21099)
30 PRINT FRE(0)
Ready
run
0
Ready
```

Con este sencillo programa se ocupa toda la memoria entre la suma del listado BASIC y el array

Fíjate como FRE(0) nos dice que no queda ni un byte libre.

Cada vez que se asigna un valor a una variable de tipo literal, como nombre\$, se reubicará la variable, dejando cada vez menos espacio disponible, aunque el espacio que anteriormente ocupaba quedará marcado como “disponible”. Cuando un programa se queda sin memoria, compactará todas las variables liberando todos los espacios “disponibles”. Este efecto también se logra al ejecutar **FRE("")**. Pero ojo, el BASIC del Amstrad no te permite ejecutar simplemente **FRE("")**, pues te dará “syntax error”. Debes al menos asignar una variable, por ejemplo **p=FRE("")**. Este comando realiza lo que hoy en día se conoce en algunos lenguajes como “garbage collection”.

Si durante la ejecución de un programa tienes un problema de memory full, puede que lo resuelvas simplemente ejecutando periódicamente una línea como esta:

```
100 c=c+1
110 if c and 63 =0 then 120 else p=fre("") :FRE cada 64 ciclos
120 el programa continua
```

Esta solución funciona solo si el problema es debido a alguna instrucción que consume un poco mas de lo que queda libre justo antes de ejecutar el FRE automáticamente. Si es ese el caso, esa solución funcionará. No obstante, es algo muy “raro” lograr solucionarlo así, porque el Amstrad si no tiene memoria hace FRE automáticamente y en teoría no hace falta hacerlo. Otra solución mas sencilla (y efectiva) es borrar o acortar líneas REM, para darle un poco mas de memoria libre al Amstrad. Si hay memoria suficiente y aun así tienes un memory full, se trata de un problema de exceso de GOSUB sin RETURN.

#### Cada variable numérica consume la siguiente memoria:

- el nombre de la variable: N bytes
- el valor, con el ultimo byte a cero indicando fin de literal: 2 bytes

#### Cada variable literal consume la siguiente memoria:

- el nombre de la variable: N bytes
- dirección de memoria (o “puntero”) donde comienza su valor: 2 bytes
- el valor, con el último byte a cero indicando fin de literal: N bytes

Cada vez que se reubica una variable literal por asignarle un nuevo valor, se carga ese nuevo valor en la zona de memoria disponible y se reasigna el puntero para que apunte a la nueva dirección, quedando el valor anterior sin apuntar por ninguna variable. Ese espacio anterior o “hueco” está disponible, pero hace falta hacer una limpieza para compactar todas las variables y liberar todos los huecos disponibles.

Vamos a ver un sencillo ejemplo que te muestra el espacio disponible al tiempo que va reubicando una variable de tipo literal (o “string”), gastando aparentemente mas y mas memoria, aunque la memoria que no se usa queda disponible (son “huecos” en la pila de variables). Si tratas de hacer lo mismo con una variable numérica el consumo de memoria será constante porque las variables numéricas ocupan siempre lo mismo y no se reubican al cambiar de valor mientras que las variables literales ( o “strings”) al cambiar de valor cambian de longitud.

```

10 numero=rnd*100
20 c$=str$(numero)
30 print fre(0)
40 goto 10

```

Como puedes apreciar en cada iteración queda menos memoria disponible, pero si lo dejas ejecutar indefinidamente, cuando se quede sin memoria el AMTRAD ejecutará un procedimiento de limpieza de memoria no usada y volverá a tener memoria. Ese procedimiento es el mismo que cuanto ejecutas FRE(" ").

No confundir con el comando CLEAR que libera el espacio eliminando todas las variables de la pila de variables.

Es recomendable ejecutar FRE(" ") periódicamente en tu programa, ya que la operación de compactación de todas las variables cuando se ha consumido toda la memoria es más trabajo (y por tanto mas lento) que si periódicamente haces un FRE(" ") que tenga poco trabajo que hacer.

Si tu programa tras un tiempo ejecutándose produce un error de memory full, prueba a ejecutar un FRE("") periódicamente, borra líneas "REM" para darle mas memoria libre o bien comprueba que no se trate de un exceso de anidamiento de GOSUB, que es el error mas frecuente

**Ready**  
**run**  
42150  
42137  
42124  
42111  
42098  
42086  
42073  
42060  
42047  
42034  
42021  
42008  
41995  
41982  
41969  
41956  
41943  
41931  
41918

<tras muchas iteraciones>

141  
128  
115  
102  
89  
76  
64  
51  
38  
25  
12  
42137  
42124  
42111  
42098  
42085  
42072  
42060

Por último una curiosidad: el CPC 464 nada mas encender te da un FRE(0) de 43.553 bytes libres mientras que el 6128 te da menos memoria, concretamente 42.249 bytes. Esto esta directamente relacionado con el hecho de que si haces un PRINT HIMEM en CPC464 te sale 43903, mientras que en CPC6128 te sale 42619.

### 2.3.1 Pila de GOSUB /RETURN

Cada vez que en el BASIC del Amstrad ejecutas GOSUB, el sistema debe apuntar a donde tiene que volver cuando hagas RETURN. Pues bien, el Amstrad CPC dispone de una pila de 83 posiciones para almacenar los saltos. Es frecuente cometer algún error en programación y en algún IF de la subrutina retornar con un GOTO (es decir, no retornar) por lo que se van acumulando si no eres cuidadoso y puedes obtener un error "memory full" durante la ejecución de tu programa.

10 GOSUB 100	63
20 REM aqui nunca llega	74
100 i=i+1	75
110 PRINT i	76
120 GOTO 10	77
	78
	79
	80
	81
	82
	83
<b>Memory full in 100</b>	
<b>Ready</b>	

En este ejemplo aunque imprimas la memoria que queda reemplazando la línea 110 por :

```
110 print i: print fre(0)
```

Verás que la memoria disponible no baja y sin embargo, aun teniendo casi toda la memoria libre, el Amstrad da memory full. Este caso es debido al anidamiento de **GOSUB**, no a que no haya memoria libre. **Solo se permiten 83 GOSUB** sin hacer **RETURN**.

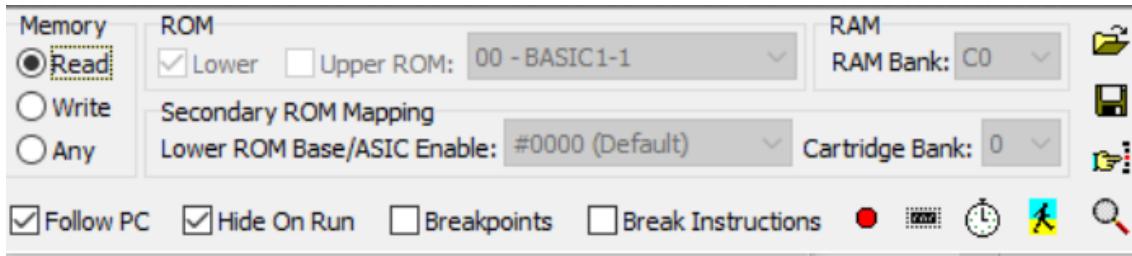
No te puedo asegurar cual es la zona de memoria donde el intérprete BASIC del Amstrad almacena las 83 direcciones para poder hacer RETURN, pero seguramente se encuentre en el área de memoria de sistema (6 KB desde la 42619 hasta la 49152). Es una zona de 6KB justo antes de la memoria de pantalla (la 49152 es la &C0000) donde el BASIC guarda los caracteres redefinibles, y también la “pila” o “stack”. Es una zona utilizada para almacenar la dirección de memoria donde se encuentra un programa antes de saltar a una rutina, de modo que pueda volver por donde estaba cuando dicha rutina termine. Se utiliza en bajo nivel (lenguaje ensamblador). La pila crece hacia direcciones más bajas a medida que almacena direcciones (es decir comienza ocupando la 49152 y va cogiendo direcciones cada vez menores) y al retornar de una rutina vuelve a decrecer. Podríamos llegar a pensar que, si una rutina llama a otra y a su vez esta llama a otra y así sucesivamente, la pila crecería tanto que se saldría de la zona “system” e invadiría otras zonas, pero no te preocupes, 8BP mantiene la pila bajo control y el BASIC del Amstrad también.

### 2.3.2 Un experimento para ver la ROM con Winape

Puedes hacer uso de Winape para ver que contiene la ROM del interprete BASIC que se solapa con las direcciones de pantalla, y lo mismo con la ROM del sistema operativo. Desde la ventana de emulación escribe

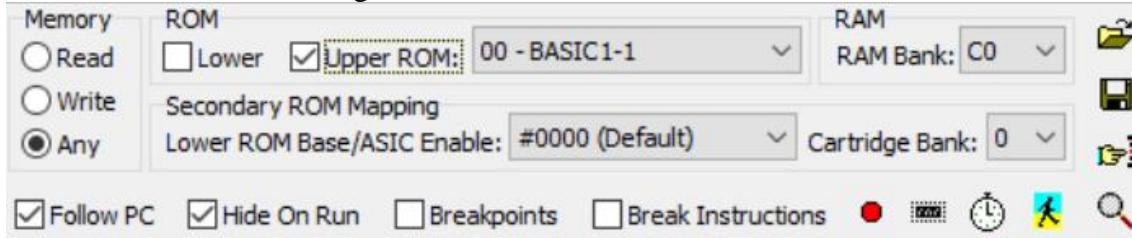
```
POKE &C000, &FF
```

A continuación, pulsas la pausa en el emulador y verás una pantalla con direcciones de memoria y valores. Busca la dirección &C000 y su contenido. Debe ser &FF. En la parte inferior verás algo como:



*Fig. 9 Winape mostrando RAM*

Pues bien, si ahora lo configuras así:



*Fig. 10 Winape mostrando ROM*

Verás el contenido de la dirección &C000 en el banco de ROM que contiene el intérprete BASIC. Es muy interesante poder hacer esto con Winape

## 2.4 Mapa de memoria de 8BP y Opciones de Ensamblaje

Puesto que el BASIC del Amstrad empieza consumiendo las direcciones más bajas, para que puedan “convivir”, la librería 8BP se carga en la zona de memoria alta disponible. Es importante entender cómo funciona el BASIC para poder usar la librería ya que tendrás que usar un comando “**MEMORY**” de BASIC.

<pre> 10 a=5 20 k=@a 30 PRINT k Ready run 411 Ready </pre>	<p>El texto de un programa escrito en BASIC se almacena a partir de la dirección &amp;40 (en decimal es 64, una dirección muy “baja”) y las variables que tu programa crea se almacenan ocupando posiciones justo después de la memoria ocupada por el listado. El siguiente ejemplo muestra donde se almacena la variable “A”, que resulta ser la dirección 441</p>
--	--

El listado BASIC y las variables pueden crecer mucho si tu programa es muy grande, y podrían invadir la zona de memoria donde se almacenan las rutinas de 8BP. Para evitar eso, debes ejecutar una instrucción **MEMORY** que protege la zona de memoria donde se almacena 8BP, tus gráficos y tu música. El comando **MEMORY** es como una “**barrera**” que impide que el BASIC y sus variables puedan ocupar memoria más allá del límite que le digamos. Si lo hace, dará un error “**MEMORY FULL**” y dejará de funcionar.

## Funciones de 8BP,tus gráficos y tu música



**Programa BASIC y variables**

**O bien, programa en C y sus variables**

**La librería 8BP te deja entre 24 y 25 KB libres para tu listado BASIC o tu programa compilado en C (en 8BP también puedes programar en C).** Desde la versión V42 de 8BP es posible elegir varias **opciones de ensamblaje** en función del tipo de juego que vas a desarrollar. La idea es sencilla: si vas a hacer un juego con scroll necesitas los comandos que se encargan del scroll, pero no necesitas los comandos que se encargan de dibujar laberintos (LAYOUT en 8BP) o detectar colisiones con los muros del laberinto. Es decir, a partir de V42 algunas funciones de 8BP **no van a gastar memoria si no se necesitan**. Así 8BP puede dejarte más memoria libre para tu listado BASIC o tu programa C. Deberás elegir una **opción de ensamblaje** dependiendo del tipo de juego que quieras programar. Esta tabla resume las opciones disponibles. Pronto comprenderás el comando SAVE que aparece en cada descripción. Esta tabla también la tienes en un apéndice

Opción	Descripción de la opción	Ejemplo de juego tipo
<b>0</b>	Puedes hacer cualquier juego Todos los comandos disponibles Necesitas usar <b>MEMORY 23499</b> Para salvar librería + gráficos+ música: <b>SAVE “8BP0.bin”,b,23500,19119</b>	cualquiera
<b>1</b>	juegos de laberintos o de pasar pantallas Necesitas usar <b>MEMORY 23999</b> <b>No disponibles en este modo:</b>  MAP2SP,  UMAP,  3D Para salvar librería + gráficos+ música: <b>SAVE “8BP1.bin”,b,25000,17619</b>	
<b>2</b>	Para juegos con scroll Necesitas usar <b>MEMORY 24799</b> No Disponibles en este modo:  LAYOUT,  COLAY,  3D Para salvar librería + gráficos+ música: <b>SAVE “8BP2.bin”, b,24800,17819</b>	
<b>3</b>	Para juegos con pseudo 3D Necesitas usar <b>MEMORY 23999</b> No Disponibles en este modo:  LAYOUT,  COLAY Para salvar librería + gráficos+ música: <b>SAVE “8BP3.bin”, b,24000,18619</b>	

Como puedes observar la opción que te deja más memoria libre para tu programa es la 1, ya que te deja libres 25KB para tu juego. La opción 3 te deja 24 KB y la opción cero te deja 23.5 KB. No te recomiendo que uses la opción cero, a menos que realmente la necesites. Es mejor que escojas la opción adecuada para tu juego y así tendrás más memoria disponible.

Estas 24-25 KB libres son para tu listado, ya que la música y los gráficos de tu juego se almacenan en otra zona más “alta”. Por ejemplo, con la opción 1 tienes:

- 25 KB libres para tu listado (BASIC o C) y variables
- 1.5 KB libres para tu música
- 8.5 KB libres para tus gráficos

**TOTAL: 35 KB libres para tu juego**

La opción de ensamblaje que escojas se define en un fichero de 8BP llamado “make\_all\_mygame.asm”. Este fichero tiene una línea que **debes editar para elegir la opción que prefieras**.

```
; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos

; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
; -----
; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 25000
;                               disponibles los comandos |LAYOUT, |COLAY
;
; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 24800
;                               disponibles los comandos |MAP2SP, |UMA
;
; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 24000
;                               disponible comando |3D
;
; ASSEMBLING_OPTION = 4 --> uso futuro

let ASSEMBLING_OPTION = 1
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

;-----GRAFICOS-----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"
```

Antes de cargar la librería, deberás ejecutar el comando **MEMORY** con el límite asociado al modo de ensamblaje que hayas escogido para tu tipo de juego. Desde la versión V42 quedan libres 24KB, 24.8 KB o 25KB según la opción de ensamblaje que escojas.

Con independencia de la opción de ensamblaje que escojas, las direcciones de memoria donde se encuentran todos los comandos son exactamente las mismas (apéndice XI). Por

ejemplo, el comando LAYOUT podrás invocarlo si escoges la opción de ensamblaje 1 o 2, pero si usas la opción 2 , invocar a dicho comando no hará absolutamente nada, al igual que ocurrirá con el comando MAP2SP si usas la opción de ensamblaje 1.

```

        AMSTRAD CPC464 MAPA DE MEMORIA de 8BP

;
; &FFFF +-----+
; | pantalla + 8 segmentos ocultos de 48bytes cada uno
; &C000 +-----+
; | system (simbolos redefinibles, stack pointer, etc.)
; 42619 +-----+
; | banco de 40 estrellas (desde 42540 hasta 42619 = 80bytes)
; 42540 +-----+
; | map layout de caracteres (25x20 =500 bytes)
; | y mapa del mundo (hasta 82 elementos caben en 500 bytes)
; | ambas cosas se almacenan en la misma zona de memoria
; | porque o usas una o usas otra
; 42040 +-----+
; | sprites (casi 8.5KB para dibujos).
; | disponibles de 8440 bytes si no hay secuencias ni rutas)
; | aquí tambien se almacenan las imágenes del alfabeto
; +-----+
; | definiciones de rutas (de longitud variable cada una)
; +-----+
; | secuencias de animacion de 8 frames (16 bytes cada una)
; | y grupos de secuencias de animacion (macrosecuencias)
; 33600 +-----+
; | canciones
; | (1500 Bytes para musica editada con WYZtracker 2.0.1.0)
; 32100 +-----+
; | rutinas 8BP (8100 bytes o 7100 bytes)
; | aqui estan todas las rutinas y la tabla de sprites
; | incluye el player de musica "wyz" 2.0.1.0
; 25000 +-----+
;
; | TU LISTADO BASIC o C
; | 24KB, 24.8 KB o hasta 25KB libres para BASIC o C según
; | la opción de ensamblaje que uses para 8BP
;
; |
; 0 +-----+

```

*Fig. 11 Memoria usando 8BP*

Si el espacio destinado a gráficos o a música se te queda corto y necesitas más, 8BP te permite que ubiques imágenes y música en otras zonas (por debajo de la dirección 24000) y funcionará perfectamente.

### 3 Herramientas necesarias

**Winape:** emulador para S.O. windows con editor para editar y probar tu programa BASIC. Y también para ensamblar los gráficos y las músicas

**SPEDIT:** (“Simple Sprite Editor”) herramienta BASIC para editar tus gráficos. El resultado de spedit es código en ensamblador que se envía a la impresora del Amstrad CPC. Ejecutando la herramienta dentro de Winape, la impresora se redirige a un fichero de texto de modo que tus gráficos se almacenarán en un fichero txt. Esta herramienta ha sido creada para complementar a la librería 8BP y los graficos de todos los juegos que he creado los he hecho con SPEDIT

**Wyztracker:** para componer música, bajo windows. El programa capaz de tocar las melodías compuestas por Wyztracker es el Wyzplayer, el cual está integrado dentro de 8BP. Tras ensamblar la música podrás hacerla sonar con un sencillo comando |MUSIC

**Librería 8BP:** instala nuevos comandos accesibles desde BASIC para tu programa. Como comprobarás, esto va a ser el “corazón” que mueva la maquinaria que construyas.

**CPCDiskXP :** te permite grabar un disquete de 3.5” que luego podrás insertar en tu CPC6128 si dispones de un cable para conectar una disquetera. Si quieres hacer una cinta de audio para CPC464 esta herramienta no la necesitas

**2CDT:** imprescindible herramienta para crear ficheros .cdt. Yo normalmente extraigo los ficheros de un .dsk al disco de windows usando CPCDiskXP y después uso 2cdt para crear el archivo cdt

**Tape2wav:** herramienta que te permite crear ficheros .wav a partir de .cdt

#### OPCIONALMENTE:

**ConvImgCPC:** editor de imágenes de carga para tus juegos. También convierte desde BMP. Programado por Ludovic Deplanque (“DEMONIAK”)

**RGAS:** (Retro Game Asset Studio) potente editor de sprites, evolucionado del a herramienta AMSprite, creado por Lachlan Keown. Este editor de sprites es compatible con 8BP y corre bajo Windows. Cuando Spedit se te quede pequeño, esta puede ser la mejor opción.

#### NO RECOMENDADAS:

**Fabacom:** compilador ejecutable dentro del AMSTRAD CPC 6128 o desde el emulador Winape para compilar tu programa BASIC y hacerlo ejecutar más rápido. Es compatible con las llamadas a los comandos de la librería 8BP. Sin embargo, no es recomendable por varios motivos:

- Tu programa ocupará mucho mas pues fabacom necesita 10KB adicionales para sus librerías, y además, una vez que compila tu programa sigue ocupando lo

mismo, de modo que un programa de 10KB de BASIC se transforma en uno de 20KB.

- Hay documentados algunos problemas de incompatibilidad de este compilador con algunas instrucciones de BASIC.
- Además, como verás a lo largo de este libro, puedes lograr una velocidad muy alta sin necesidad de compilar.

**CPC BASIC compiler:** compilador ejecutable para windows. Es compatible con las llamadas a los comandos de la librería 8BP. A diferencia de fabacom, el programa compilado solo ocupa unos 5KB extra, sin embargo, reserva 16KB de trabajo para funcionar de modo que apenas te deja espacio para tu programa, ya que en total "roba" 20 KB. Y además no es 100% compatible locomotive BASIC.

La ganancia en velocidad tanto con fabacom como con CPC Basic compiler puede llegar a un 50%, dependiendo del juego. Es decir, que un juego que funcione a 20 FPS pasaría a funcionar a 30 FPS. Esto no está mal, pero piensa que hemos pasado del BASIC interpretado al código máquina y normalmente se dice que la velocidad se debe multiplicar al menos por 100 (hablaríamos de un incremento del 10000%). Sin embargo, sólo hemos ganado un 50%. El motivo de tan "pobre" ganancia es que las instrucciones de 8BP ya hacen todo el trabajo duro y en realidad el compilador sólo traduce a código máquina la parte menos pesada, la lógica del juego.

Por último, has de saber que, si quieres que tus programas funcionen mucho más rápidos, desde la versión v40 de 8BP hay soporte para lenguaje C, de modo que puedes o bien programar todo tu juego directamente en C o bien programar en BASIC y traducir tan solo a C el "ciclo de juego". Hay un capítulo de este manual dedicado exclusivamente a este tema.

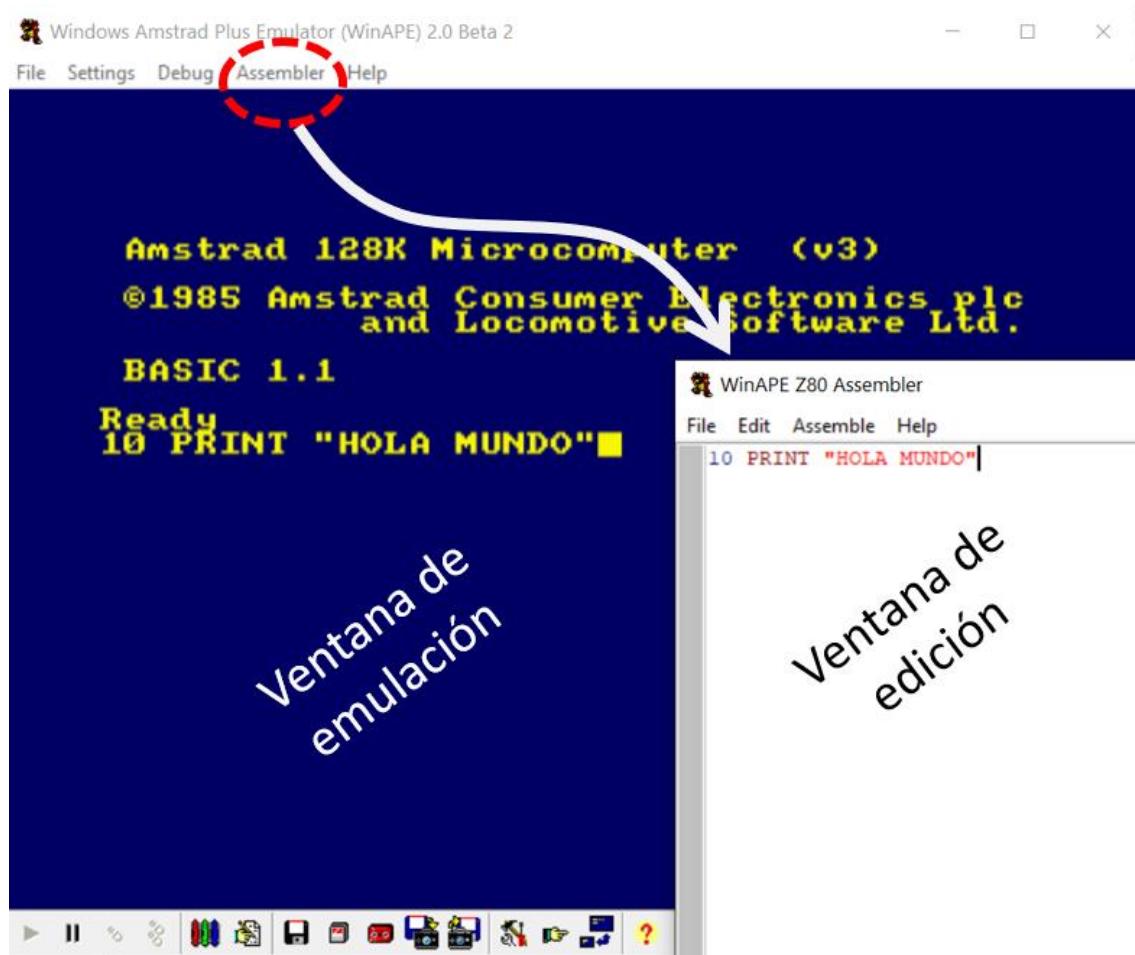
## 4 Primeros pasos con 8BP

### 4.1 Instalar winape

Lo primero que debes hacer es instalarte la última versión de **Winape**, que es un emulador y a la vez editor y ensamblador de Amstrad. Lo puedes descargar desde [www.winape.net](http://www.winape.net)

### 4.2 Familiarizándose con winape: “hola mundo”

Una vez instalado el Winape, familiarízate un poco con él, probando algún juego de Amstrad y probando a cambiar la configuración. Trata de abrir el assembler que lleva incorporado en el menú y edita un “hola mundo” en la ventana de ensamblador. A continuación, copia y pega el texto en la ventana de emulación. Verás como se pega carácter a carácter

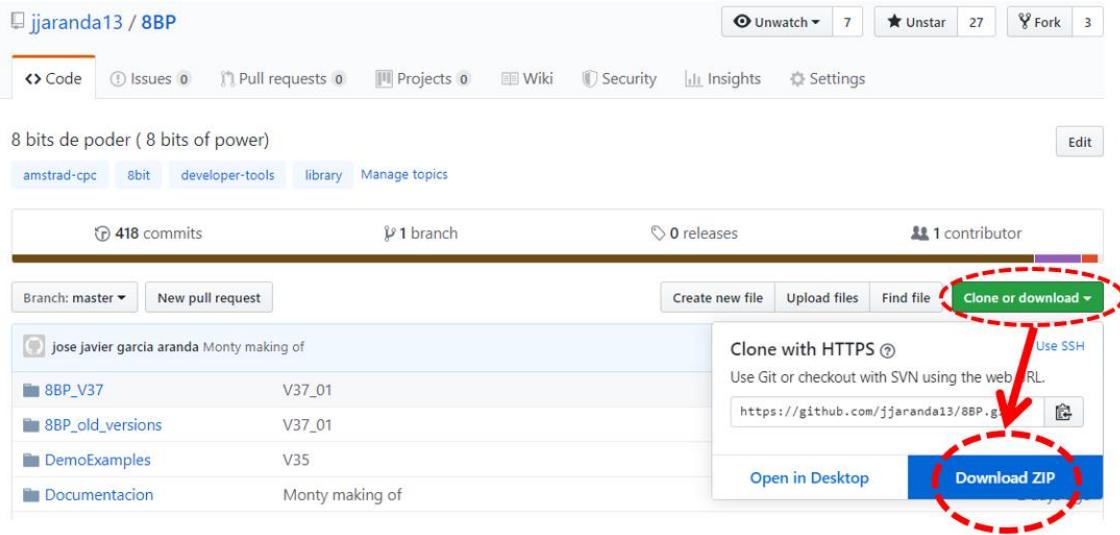


Si haces un programa más largo, para copiarlo en la ventana de emulación es muy interesante la opción settings->high speed. Verás como se copia rápidísimo.

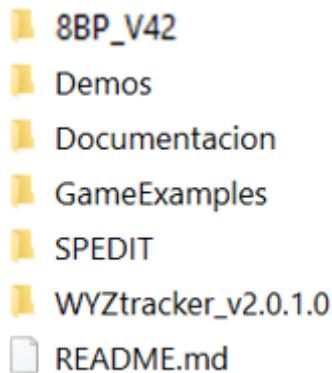
### 4.3 Descárgate la librería 8BP

Estamos llegando a lo más interesante. Ve a la url <https://github.com/jjaranda13/8BP> y descárgate el zip (<https://github.com/jjaranda13/8BP/archive/master.zip>) . Para ello

puedes simplemente pulsar el botón verde “clone or download” y a continuación seleccionas el zip



Una vez que te lo has descargado, descomprímelo en el directorio que quieras. Tendrás el siguiente resultado:



#### 4.4 Ejecuta las demos

Ahora vamos a hacer una primera toma de contacto con 8BP viendo algunas demos. Entra en el directorio Demos. En el encontrarás una serie de subcarpetas : **ASM, BASIC, C, DSK, MUSIC**

Entra en DSK . Allí verás un fichero .dsk con las demos. Desde winape ve al menú File->drive A-> insert Disc image y selecciona el archivo de las demos

Una vez seleccionado, desde la ventana de emulación del Amstrad escribe **CAT** para ver los archivos

**cat**

**Drive A: user 0**

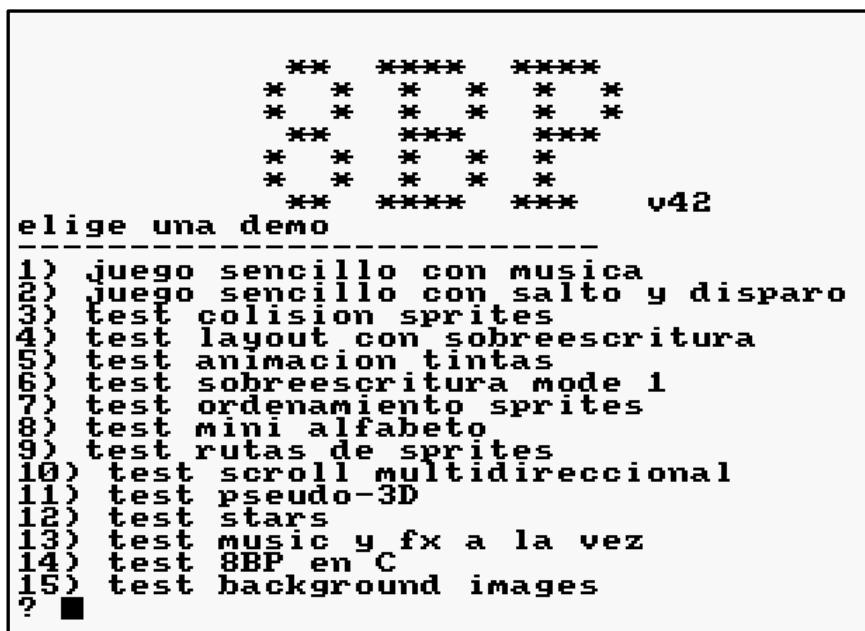
<b>8BP0</b>	<b>.BIN</b>	<b>18K</b>	<b>DEMO15</b>	<b>.BAS</b>	<b>2K</b>
<b>8BP1</b>	<b>.BIN</b>	<b>18K</b>	<b>DEMO2</b>	<b>.BAS</b>	<b>2K</b>
<b>8BP2</b>	<b>.BIN</b>	<b>19K</b>	<b>DEMO3</b>	<b>.BAS</b>	<b>1K</b>
<b>CICLO</b>	<b>.BIN</b>	<b>4K</b>	<b>DEMO4</b>	<b>.BAS</b>	<b>2K</b>
<b>DEMO1</b>	<b>.BAS</b>	<b>2K</b>	<b>DEMO5</b>	<b>.BAS</b>	<b>2K</b>
<b>DEMO10</b>	<b>.BAS</b>	<b>2K</b>	<b>DEMO6</b>	<b>.BAS</b>	<b>1K</b>
<b>DEMO11</b>	<b>.BAS</b>	<b>1K</b>	<b>DEMO7</b>	<b>.BAS</b>	<b>2K</b>
<b>DEMO11</b>	<b>.BIN</b>	<b>1K</b>	<b>DEMO8</b>	<b>.BAS</b>	<b>1K</b>
<b>DEMO12</b>	<b>.BAS</b>	<b>3K</b>	<b>DEMO9</b>	<b>.BAS</b>	<b>1K</b>
<b>DEMO13</b>	<b>.BAS</b>	<b>2K</b>	<b>LOADER</b>	<b>.BAS</b>	<b>2K</b>
<b>DEMO14</b>	<b>.BAS</b>	<b>3K</b>			

**89K free**

**Ready**

Cada fichero .BAS es una demo donde podrás ver alguna de las características de 8BP (no todas sus posibilidades se pueden ver con las demos, pero hay algunas representativas).

Ejecuta el comando **RUN “LOADER.BAS”**. Obtendrás el siguiente menú:

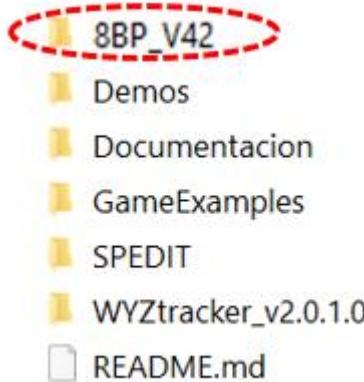


Ya puedes elegir una demo y probarla. A disfrutar. En el siguiente y último paso empezaremos la creación de un juego

#### **4.5 Creando tu primer programa con 8BP**

Hemos probado un fichero .dsk que contiene muchas demos, con gráficos y música. En el directorio **Demos/ASM** y en el **Demos/MUSIC** se encuentran los gráficos y la música de las demos que has probado. Sin embargo, si quieres hacer tu propio juego o demo, es mejor que empieces con unos ficheros “limpios” sin todos los gráficos que requieren las demos que has probado.

Nos vamos al directorio raíz. Allí encontrarás una carpeta llamada “**8BP\_V42**”. Yo te recomiendo que hagas una copia de esta carpeta y la renombres como “**mi\_juego**”. De ese modo preservarás la carpeta “**8BP\_V42**” original, aunque empieces a cambiar cosas



Dentro de la carpeta “**8BP\_V42**” encontrarás las carpetas ASM, BASIC, DSK, MUSIC, TAPE, y output\_spedit

Desde la ventana de ensamblador de winape abre el archivo “**ASM/make\_all\_mygame.asm**” y ejecútalo (simplemente en el menú de winape z80 assembler seleccionas “Assemble” o pulsa Ctrl+F9). Esto comenzará a ensamblar (copiar en memoria) la librería y los gráficos en la memoria del Amstrad. En este caso vamos a ensamblar muy pocos gráficos, tan solo los indispensables para un pequeño juego. Los gráficos se encuentran en el fichero “**images\_mygame.asm**”

Tras ensamblar todo, obtendrás un mensaje como el que se muestra a continuación:

The screenshot shows the WinAPE Z80 Assembler interface. The main window displays the assembly script, which includes comments about makefiles, assembly options (0-4), and includes for 'make\_codigo\_mygame.asm', 'make\_musica\_mygame.asm', and 'make\_graficos\_mygame.asm'. A pop-up window titled 'Assembling' shows the progress: 'Assembling E:\...\\PROYECTO\_V42\\8BP\_V42\\ASM\\make\_all\_mygame.asm' with 'Pass : 2', 'Current Line : 36', 'Total Lines : 11396', and 'Errors : 0'. The assembly results show memory addresses (000059 to 000070) and corresponding hex values (A43D). The results window has an 'OK' button at the bottom right.

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos

; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
; -----
; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 1
;                      disponibles los comandos |LAYOUT|
;

; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 2
;                      disponibles los comandos |MAP2SP|
;

; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 2
;                      disponible comando |3D|
;

; ASSEMBLING_OPTION = 4 --> uso futuro

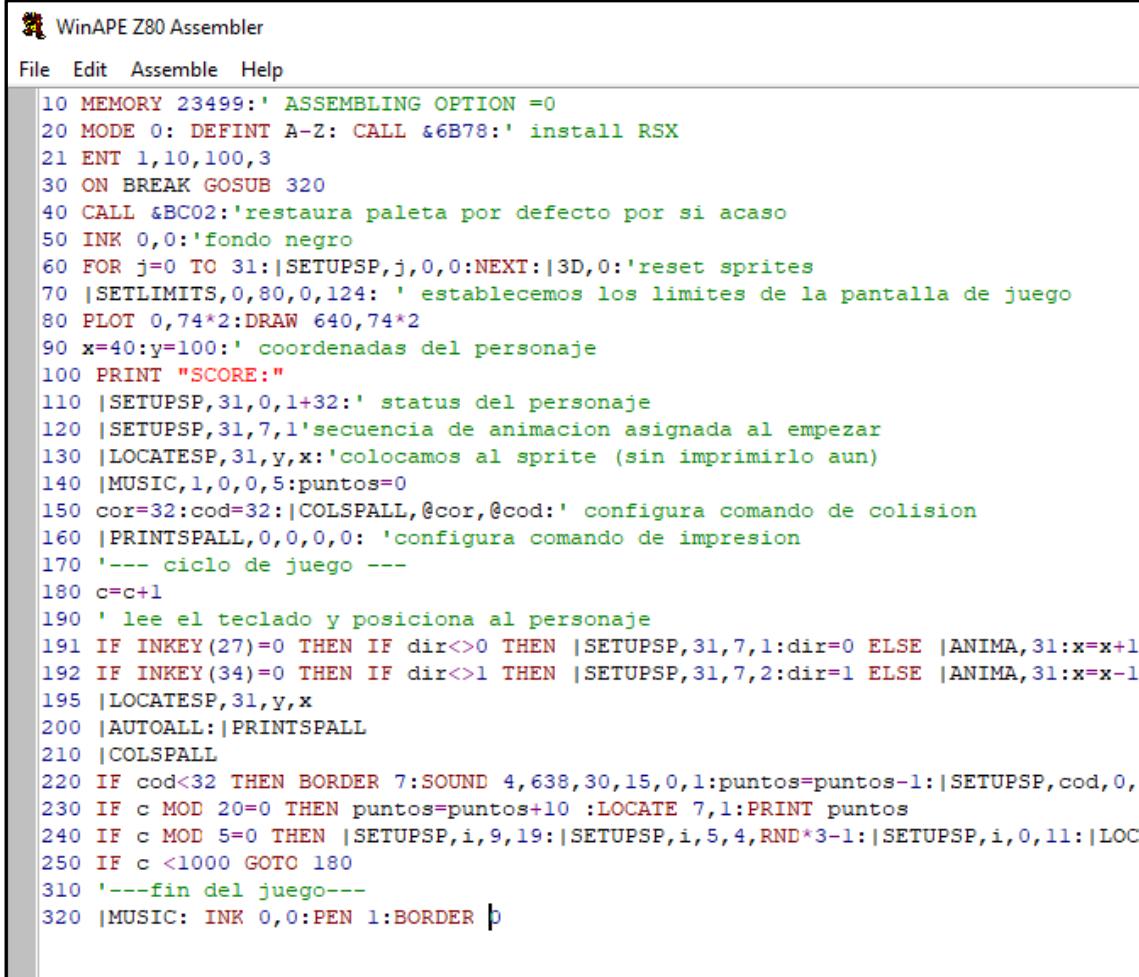
let ASSEMBLING_OPTION = 0
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

; ----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"

```

Pulsas “ok” y desde la ventana de ensamblador vamos a abrir otro archivo. En este caso vamos a abrir un fichero BASIC, concretamente “**tu\_primer\_juego.bas**” que se encuentra en la carpeta BASIC. Tras abrirlo verás el siguiente listado en pantalla, que contiene 32 líneas:



```

WinAPE Z80 Assembler
File Edit Assemble Help

10 MEMORY 23499:' ASSEMBLING OPTION =0
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
21 ENT 1,10,100,3
30 ON BREAK GOSUB 320
40 CALL &BC02:'restaura paleta por defecto por si acaso
50 INK 0,0:'fondo negro
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124: ' establecemos los limites de la pantalla de juego
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordenadas del personaje
100 PRINT "SCORE:"
110 |SETUPSP,31,0,1+32:' status del personaje
120 |SETUPSP,31,7,1'secuencia de animacion asignada al empezar
130 |LOCATESP,31,y,x:'colocamos al sprite (sin imprimirla aun)
140 |MUSIC,1,0,0,5:puntos=0
150 cor=32:cod=32:|COLSPALL,@cor,@cod:' configura comando de colision
160 |PRINTSPALL,0,0,0,0: 'configura comando de impresion
170 '--- ciclo de juego ---
180 c=c+1
190 ' lee el teclado y posiciona al personaje
191 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,31,7,1:dir=0 ELSE |ANIMA,31:x=x+1
192 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,31,7,2:dir=1 ELSE |ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:SOUND 4,638,30,15,0,1:puntos=puntos-1:|SETUPSP,cod,0,
230 IF c MOD 20=0 THEN puntos=puntos+10 :LOCATE 7,1:PRINT puntos
240 IF c MOD 5=0 THEN |SETUPSP,i,9,19:|SETUPSP,i,5,4,RND*3-1:|SETUPSP,i,0,11:|LOC
250 IF c <1000 GOTO 180
310 '---fin del juego---
320 |MUSIC: INK 0,0: PEN 1:BORDER 0

```

Selecciona todo y cópialo. Luego ve a la ventana de emulación del CPC y pégalo usando el menú **FILE->paste**

Como el listado es algo más largo que el “hola mundo”, usa el menú de winape y selecciona **settings->high speed** para copiarlo y luego vuelve al “normal speed”

Como ya está ensamblada la librería y los gráficos, puedes hacer RUN y el juego se ejecutará. Debes esquivar unas bolas que caen del cielo para no morir, moviéndote a derecha e izquierda



Puedes probar a modificar el programa y ver sus efectos. Poco a poco iras aprendiendo 8BP y podrás hacer modificaciones interesantes tales como cambiar la frecuencia con la que salen bolas enemigas o su velocidad, o reemplazar al soldado por una nave espacial y a las bolas por naves enemigas con diversas trayectorias.

Si quieras probar a que velocidad funciona en lenguaje C, basta con que cargues el .dsk y ejecutes “loader.bas”, se cargará una versión del juego que te permite elegir entre la versión BASIC y la versión C y asi podrás comparar. Hay un capitulo de este libro dedicado a la programación en C usando 8BP y un mini BASIC para que programes en C igual que lo harías en BASIC. Si no tienes mucha experiencia en programación C, te recomiendo ir poco a poco y programar en BASIC, los resultados serán rápidos y profesionales.

#### **4.6 Crea tu .dsk con tu juego 8BP**

Por ultimo, vamos a crear un disco con tu juego. Para ello, tras tener el juego funcionando debes dar estos pasos

- Crea un disco nuevo mediante winape: FILE-> drive A-> new Blanc Disc
- Formátalo: FILE->drive A->Format Disc Image
- Tras haber creado tu fichero dsk , desde la ventana de emulación ejecuta los siguientes comandos:

**SAVE “8BP0.bin”, b, 23499,19119  
SAVE “juego.bas”**

El primer comando salva en disco la librería 8BP, los gráficos y la música. En este caso usamos la opción de ensamblaje 0 (ver opciones en capitulo anterior) pero podrías usar cualquier opción para este ejemplo perfectamente. He llamado “**8BP0**” al archivo para reflejar de algún modo que se ha usado la opción 0 de ensamblaje pero el nombre de este binario puede ser cualquiera.

Ya casi hemos terminado. Ahora debes seleccionar otro disco desde el menú de winape o salir de winape para que el .dsk que has creado se haga realidad en el sistema de ficheros de windows.

Sal de winape y vuelve a abrirlo  
Selecciona el disco que has creado y ejecuta:

**MEMORY 23499  
LOAD “8BP0.BIN”  
RUN “juego.bas”**

**Aleluya !!!**

## 5 Pasos que debes dar para hacer un juego

### 5.1 Estructura en directorios de tu proyecto

Lo más recomendable a la hora de programar tu juego es que estructures los diferentes ficheros en 7 carpetas, en función del tipo de fichero del que se trata.

Es perfectamente posible meterlo todo en el mismo directorio y trabajar sin carpetas, pero es más “limpio” hacerlo como te voy a presentar a continuación

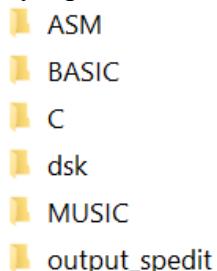


Fig. 12 estructura de directorios

- **ASM:** aquí meterás archivos de texto escritos en ensamblador (.asm), como es la propia librería 8BP, los sprites generados con el editor de sprites SPEDIT, y algunos ficheros auxiliares.
- **BASIC:** aquí meterás tu juego y utilidades como el SPEDIT y el cargador (Loader).
- **C:** esta carpeta es para usuarios “avanzados” que quieren programar en C todo el juego o al menos el ciclo de juego. Sólo es necesaria si vas a programar alguna parte de tu juego en lenguaje C
- **Dsk:** aquí meterás el archivo .dsk listo para ejecutar en un Amstrad CPC. En su interior deberás ubicar 5 ficheros de los que hablaremos en el siguiente apartado
- **Music:** con el secuenciador de música WYZtracker, podrás crear tus canciones y almacenarlas en formato .wyz en este directorio. Una vez que las “exportes”, se generarán un archivo .asm que tendrás que guardar en la carpeta ASM y un archivo binario que también almacenarás en la carpeta ASM (también los puedes dejar en esta carpeta, con tal de que los referencies adecuadamente en el fichero make\_musica.asm del directorio ASM).
- **Output\_spedit:** en esta carpeta puedes almacenar el fichero de texto que genera spedit. SPEDIT lo que hace es mandar a la impresora los sprites en formato ensamblador y el emulador winape puede recoger la salida de la impresora del Amstrad en un fichero. Aquí lo ubicaremos
- **Tape:** aquí puedes almacenar el fichero .wav si deseas hacer una cinta para cargar en el Amstrad CPC464, o el fichero .cdt

## 5.2 Tu juego en sólo 3 ficheros

Para generar tu juego vas a necesitar un fichero binario y dos ficheros BASIC. Se pueden generar varios ficheros binarios independientes (uno con la librería 8BP, otro con los gráficos, otro con la música...) pero como estas zonas de memoria son contiguas es mejor directamente generar un único binario.

El fichero binario contiene la librería, la música, los gráficos, el área de memoria de mapa del mundo o el layout y opcionalmente el banco de estrellas. Se trata de almacenar todos los componentes binarios juntos en un solo fichero, así (dependiendo de la opción de ensamblaje usarás uno u otro de estos comandos). El nombre del fichero lo he terminado con un número que indica la opción de ensamblaje, pero puede llamarse como quieras.

```
SAVE "tujuego0.bin",b,23500, 19119  
SAVE "tujuego1.bin",b,25000, 17619  
SAVE "tujuego2.bin",b,24800, 17819  
SAVE "tujuego3.bin",b,24000, 18619
```

Siendo la longitud la dirección final menos la inicial. La dirección final incluyendo música y gráficos, el mapa y el banco de estrellas, es 42619, de modo que la longitud se puede calcular restando la dirección inicial a 42620. Por ejemplo, en la opción de ensamblaje 1, tenemos que  $42619 - 25000 = 17619$ , justo la longitud que aparece en ese comando.

El segundo fichero es tu juego basic  
**SAVE "tujuego.bas"**

Y el tercer fichero es el cargador ("**loader.bas**"), que simplemente será:

```
10 MEMORY 23499  
15 LOAD "!pant.scr",&c000: 'solo si tu juego tiene pantalla de carga  
20 LOAD "tujuego0.bin"  
50 RUN "!tujuego.bas"
```

He puesto una carga de pantalla inicial en la dirección inicial de memoria de pantalla (&C000). Al final de este manual encontrarás una de las muchas formas de hacer una pantalla de carga. Es algo opcional. Puedes hacer un juego con pantalla de carga o sin ella.

Este método de 3 ficheros es útil sobre todo si ocupas casi toda la memoria disponible para gráficos. En juegos de cinta de cassette cargar 18KB puede llevar un tiempo y si no usas los 8.5KB de gráficos, quizás sea mejor que cargues por separado los distintos binarios y así ahorres tiempo de carga. Por ejemplo, si usas solo 2KB de gráficos, con un solo binario de longitud 18619 cargarías 8.5KB de gráficos, es decir 6.5KB extra vacíos. Esto en tiempo de carga en cinta pueden suponer casi dos minutos extra de tiempo. En disco (CPC 6128) da lo mismo porque no tarda nada en cargarlos. En ese caso es mejor hacer uno o dos binarios que almacenen solo la memoria que uses.

Para hacer estos 3 ficheros debes dar estos pasos:

### PASO 1

Editar gráficos con SPEDIT y el resultado (SPEDIT lo manda a un .txt) copiarlo en **images\_mygame.asm**

## PASO 2

Editar la música con WYZtracker

Modificar music\_mygame.asm para incluir las musicas creadas

Las melodías se ensamblarán una detrás de otra, de modo que cada una comenzará en una dirección de memoria diferente que dependerá del tamaño que ocupen.

## PASO 3

Re-ensamblar la librería 8BP, de modo que la parte de la librería que selecciona las melodías (el player wyz) pueda conocer en qué direcciones de memoria se han ensamblado (hay más dependencias, pero esa es una de ellas). Una vez re-ensamblada, tendrás que salvar todo con uno de estos comandos según la opción de ensamblaje que uses (el nombre puede ser el que quieras, yo he puesto un numero al final para indicar de algún modo la opción de ensamblaje):

```
SAVE "tujuego0.bin",b,23500, 19119  
SAVE "tujuego1.bin",b,25000, 17619  
SAVE "tujuego2.bin",b,24800, 17819  
SAVE "tujuego3.bin",b,24000, 18619
```

Esta será una versión de la librería específica para tu juego. Por ejemplo, el comando **|MUSIC,0,0,3,6** hará sonar la melodía número 3 que tú mismo has compuesto. La melodía número 3 puede ser completamente diferente en otro juego.

## PASO 4

Programar tu juego, el cual debe primeramente ejecutar la llamada para instalar los comandos RSX, es decir **CALL &6b78**. Y algo muy importante: no te olvides de incluir el comando **MEMORY** al principio, para evitar que el BASIC en ejecución guarde variables por encima de la dirección donde comienza 8BP.

```
MEMORY 23499 :rem usa este MEMORY para la opcion 0  
MEMORY 24999 :rem usa este MEMORY para la opcion 1  
MEMORY 24799 :rem usa este MEMORY para la opcion 2  
MEMORY 23999 :rem usa este MEMORY para la opcion 3
```

Tu juego lo puedes programar usando el editor de winape, mucho más versátil que el editor del AMSTRAD y sirve tanto para editar ensamblador (.asm) como para editar BASIC (.bas). El editor de winape es sensible a las palabras clave y las cambia de color automáticamente, facilitando la labor de programar. Tras escribir un programa BASIC hay que copiar/pegar en la ventana de CPC del winape. Para hacerlo más deprisa puedes activar la opción “High Speed” de winape durante el pegado, de ese modo ese proceso será inmediato.

## PASO 5

Cargar todo con un **loader.bas**, que deberás hacer en BASIC

## PASO 6

Crear una cinta o un disco con tu juego

## 5.3 Crear un disco o una cinta con tu juego

### 5.3.1 Hacer un disco

Para crear un nuevo disco desde winape hacemos

```
File->drive A-> new blank disk
```

Con ello te aparecerá una ventana de administración de archivos para que le des nombre al nuevo fichero .dsk

Una vez creado ya puedes guardar ficheros con el comando SAVE. Para borrar un archivo se utiliza el comando “|ERA” (abreviatura de ERASE), que solo existe en CPC 6128 como parte del sistema operativo “AMSDOS” (esto en CPC464 no existe pues funcionaba con cinta de cassette).

```
|ERA, "juego.*"
```

Y se borrarán

Para cargar el juego necesitas un cargador que cargue uno por uno los ficheros necesarios. Algo como (el comando MEMORY depende de la opción de ensamblaje):

```
10 MEMORY 23499: 'el comando memory depende de la opcion de ensamblaje
15 LOAD "!pant.scr",&c000: 'solo si tu juego tiene pantalla de carga
20 LOAD "tujuego0.bin"
50 RUN "!tujuego.bas"
```

Para salvar cada uno de los ficheros debes usar el comando SAVE con los parámetros necesarios, por ejemplo:

```
SAVE "LOADER.BAS"
SAVE "tujuego0.bin",b,23500, 19119
SAVE "tujuego.BAS"
```

Si quieras grabar el .dsk en un disquete de 3.5” y conectarlo a una disquetera externa de tu AMSTRAD CPC 6128, necesitarás el programa CPCDiskXP, muy sencillo de usar. A partir de un .dsk puede grabar un disquete de 3.5” en doble densidad (no olvides tapar el agujero del disquete para “engaños” al PC).

### 5.3.2 Hacer una cinta con Winape

Lo más importante al crear una cinta es guardar en ella los ficheros en el orden en el que van a ser cargados por el ordenador. Una cinta no es como un disco en el que puedes cargar cualquier fichero almacenado, sino que los ficheros se encuentran uno detrás de otro, por lo que debes poner especial cuidado en este punto.

Si tu cargador de juego es así:

```
10 MEMORY 23499
15 LOAD "screen.scr",&c000 : rem si tu juego tiene pantalla de carga
20 LOAD "!tujuego0.bin"
50 RUN "!tujuego.bas"
```

Es muy importante la exclamación “!” para que el Amstrad no saque el mensaje “press play then any key” al ejecutar cada LOAD

Primero debes guardar el cargador (supongamos que se llama “**loader.bas**”), después el fichero “**tujuego.bin**” y por último “**tujuego.BAS**”.

Para crear un “.wav” o desde winape

```
file->tape->press record
```

En ese momento te saldrá un menú de administración de archivos para que podamos decidir qué nombre le damos al fichero “.wav”

Si estás en modo CPC 6128, entonces a continuación debes ejecutar desde BASIC

| **TAPE**

Y luego

**SPEED WRITE 1**

Con este comando lo que habremos hecho es decirle al AMSTRAD que grabe a 2000 baudios. Así la carga durará menos. Si no ejecutas ese comando, la grabación se realizará a 1000 baudios, más segura pero mucho más lenta

**SAVE "LOADER.BAS"**

Saldrá un mensaje indicando que presiones rec&play, y luego pulsas “ENTER”.  
Después grabar todos y cada uno de los ficheros:

```
SAVE "tujuego0.bin",b,23500, 19119  
SAVE "tujuego.BAS"
```

Por último, debemos hacer una última operación para que winape cierre el fichero.

```
file->remove tape
```

Tras hacer el “remove tape”, el fichero adquirirá su tamaño (si no lo haces puedes ver que en el disco de tu PC el fichero no crece y es debido a que no se ha volcado al disco)

Para cargar el juego, si estás en un CPC6128

| **TAPE**  
RUN “”

Para volver a usar el disco

| **DISC**

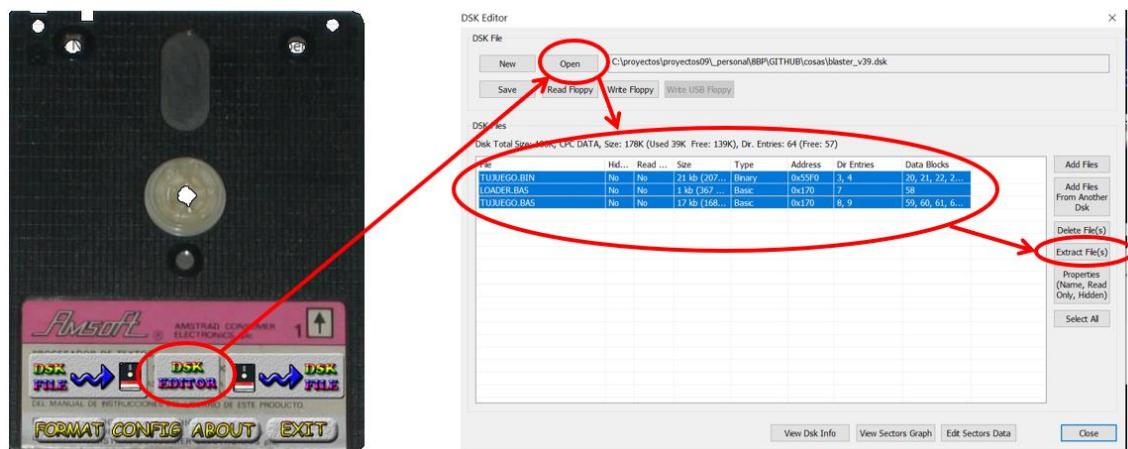
Si quieras salvar una pantalla de carga consulta el apéndice I sobre organización de la memoria de vídeo, ahí te explico cómo se hace.

### 5.3.3 Hacer una cinta fácilmente con CPCDiskXP, 2cdt y tape2wav

Winape es una herramienta fabulosa para programar y para emular. Sin embargo tiene una pequeña limitación: no te permite hacer una cinta en formato cdt, tan solo en wav. Existe una forma muy rápida y fiable que te permitirá hacer ficheros .cdt y wav para la que necesitas las herramientas CPCDiskXP, 2cdt y tape2wav.

Vamos a partir de la base de que tienes creado un .dsk con tus archivos. Teniendo eso, debes dar los siguientes pasos:

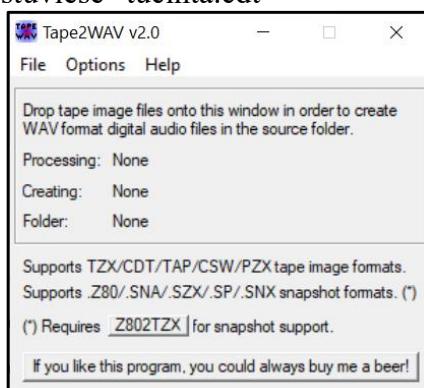
- 1) En primer lugar con la herramienta CPCDiskXP puedes abrir el .dsk y extraer los ficheros necesarios para tu juego (el “loader.bas”, el “tujuego.bin” y “tujuego.bas”). Para ello simplemente pulsas “disk edition”, luego “open”, abres tu .dsk, seleccionas los ficheros y finalmente pulsas “extract files”. Una vez hecho eso, tendrás los ficheros en el file system de windows



- 2) A continuación usas la herramienta 2cdt para grabar los ficheros, uno por uno en un fichero .cdt. Los comandos serían:

```
2cdt.exe -n -s 1 -r "LOADER.bas" "loader.BAS" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego0.bin" "tujuego0.bin" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego.bas" "tujuego.bas" tucinta.cdt
```

- 3) Ahora ya tienes el fichero tucinta.cdt creado. Si además quieres tener un .wav para poder cargarlo en un CPC 464 real, puedes hacer uso de la herramienta tape2wav. Simplemente la arrancas y arrastras con el ratón el fichero .cdt hasta la herramienta. El tape2wav creará inmediatamente un fichero “tucinta.wav” justo en el directorio donde estuviese “tucinta.cdt”



### 5.3.4 Solución de problemas con LOAD y MEMORY

Antes de cargar un fichero BASIC, el Amstrad se asegura que va a tener espacio disponible para ejecutarlo. Puede que el programa BASIC solo use 1KB de variables, pero eso el Amstrad no lo sabe, de modo que es más conservador y exige que tengas 5KB extra adicionales vacíos en la memoria. Estos 5KB pueden parecer excesivos, pero el Amstrad no sabe a priori cuantas variables vas a declarar en tu programa y prefiere tener mucho espacio libre para almacenar variables que quedarse corto y que el programa falle.

Esto significa que si previamente has cargado uno o varios ficheros binarios (con gráficos, librería 8BP o lo que sea) y has dejado libres 20 KB, entonces no podrás cargar un juego BASIC de 20 KB sino como mucho un juego de 15 KB. Pero no te preocupes, hay varios modos de solventarlo. Te explicaré el más sencillo

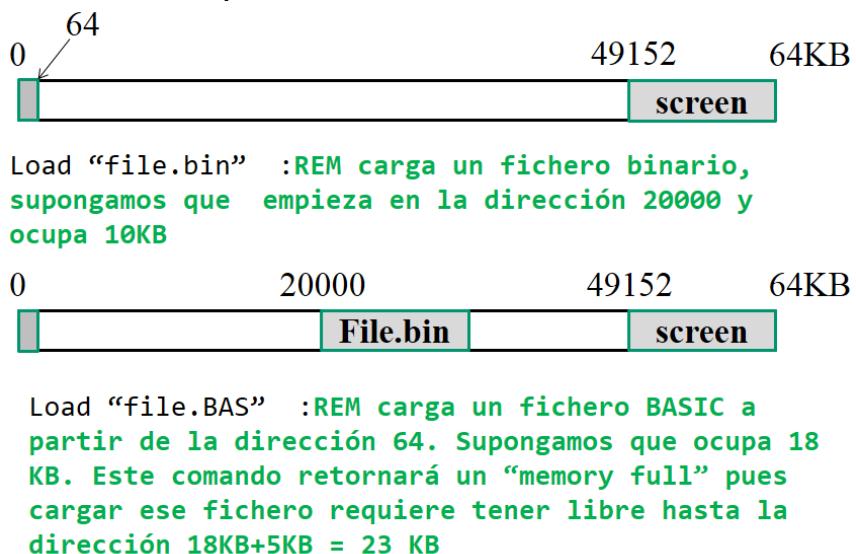


Fig. 13 El problema de LOAD

La solución consiste en hacer un fichero "loader.bas" que cambie el MEMORY. Supongamos que tienes datos binarios a partir de la dirección 20.000 y que tu programa ocupa 18KB, dejando menos de 5KB de margen. Lo único que tienes que hacer es:

```
10 MEMORY 19999
20 LOAD "!juego.bin": rem carga datos a partir de la 20000
30 CLEAR: MEMORY 23000 : rem asi le damos mas margen de RAM libre.
40 RUN "!juego.bas": rem la primera linea de juego.bas debe ser memory
19999
```

Este método es muy simple y 100% fiable porque, aunque dejas desprotegidos partes de los datos binarios durante la carga del BASIC, lo primero que haces en BASIC es ejecutar el MEMORY, con lo que vuelve a quedar protegido antes de haber creado cualquier variable.

Otro de los problemas típicos relacionados con el error MEMORY FULL ocurre cuando cargamos un programa y en mitad de la ejecución lo paramos (pulsando ESC dos veces) Es posible que nos de **MEMORY FULL** al tratar de hacer cosas como acceder al disco con un comando CAT.

```
Break in 420
Ready
CAT
Memory full
Ready
```

Esto es debido a que nuestro programa BASIC puede consumir mucha memoria RAM de variables. Haberlo parado no significa que hayan desaparecido las variables del programa, de hecho, continúan existiendo e incluso puedes imprimirlas para ver su valor. Lo que haremos en este caso es simplemente ejecutar el comando **CLEAR**, el cual libera la memoria de dichas variables y a continuación nuestro comando CAT (o el que queramos).

```
Break in 420
Ready
CAT
Memory full
Ready
CLEAR
Ready
CAT

Drive A: user 0
LOADER :BAK 1K SP :BAS 18K
LOADER :BAS 1K SP :BIN 19K
SP :BAK 18K SP :SCR 17K

104K free
Ready
```

Si has hecho un programa en BASIC tan grande que no te quedan 5KB libres entre el programa BASIC y el binario ensamblado, entonces lógicamente también sufrirás el error **MEMORY FULL** cuando vayas a salvar tu juego al disco

Al intentar salvar el binario te dará un error **MEMORY FULL**, pero es muy fácil de solucionar. Simplemente no cargues el listado BASIC en tu emulador, ni ejecutes ningún comando **MEMORY**. Al ensamblar con winape el fichero “make\_all.asm” tendrás todos tus gráficos, música y librería 8BP ensambladas en la memoria del Amstrad. En ese momento ejecuta tu comando **SAVE** y te funcionará. El comando para salvar todo en un solo binario depende de la opción de ensamblaje que hayas puesto en el fichero **make\_all\_mygame.asm**, y será uno de estos:

SAVE “tujuego0.bin”,b,23500, 19119
SAVE “tujuego1.bin”,b,25000, 17619
SAVE “tujuego2.bin”,b,24800, 17819
SAVE “tujuego3.bin”,b,24000, 18619

Sin embargo, si has almacenado cosas binarias (gráficos extra, mapas, etc.) por debajo de la dirección inicial de 8BP tendrás que tenerlo en cuenta. Por ejemplo, si tu juego comienza a usar memoria en la dirección 20000, el comando será (fíjate que he aumentado la longitud para que ocupe desde la 20000 hasta la 42619)

SAVE “tujuego.bin”,b,20000, 22619
-----------------------------------

En este momento ya puedes copiar y pegar tu programa BASIC en el emulador. Una vez copiado, no ejecutes ningún comando **MEMORY**, y guarda en disco tu fichero .BAS

Como en este momento aun no has ejecutado ningún comando **MEMORY**, el Amstrad “cree” que tiene más de 5KB libres por encima de tu programa BASIC y no nos dará el mensaje **MEMORY FULL**. Una vez que ejecutes tu comando **MEMORY** (por ejemplo, si tus datos binarios comienzan en 20000 será un **MEMORY 19999** en lugar del 23999) el Amstrad comprobará si tienes 5KB libres entre tu programa BASIC y la dirección del **MEMORY** y si no hay suficiente, te dará error al ejecutar el comando **SAVE**, aunque funcione el juego. Si durante la ejecución tu juego intenta consumir más espacio que el que tiene libre hasta la dirección del **MEMORY**, se parará y dará un error **MEMORY FULL**.

## 6 Ensamblado de la librería, música y gráficos

Este capítulo detalla un poco más que es lo que ocurre cuando ejecutas el fichero “make\_all” y te permitirá comprender mejor todo el proceso, **aunque si tienes ganas de empezar a aprender a programar con 8BP, puedes saltarlo y volver aquí más adelante**, cuando quieras entender mejor donde debes meter los gráficos y la música y como se ensambla la librería con ellos.

Para añadir música y gráficos a tu juego deberás re-ensamblar la librería, Esto es debido a que, por ejemplo, el player de música está integrado en la librería y necesita conocer donde comienza (dirección de memoria) cada canción, por lo que es necesario re-ensamblar y guardar la versión de la librería específica para tu juego, así como el fichero de gráficos ensamblado y el fichero de música ensamblado.

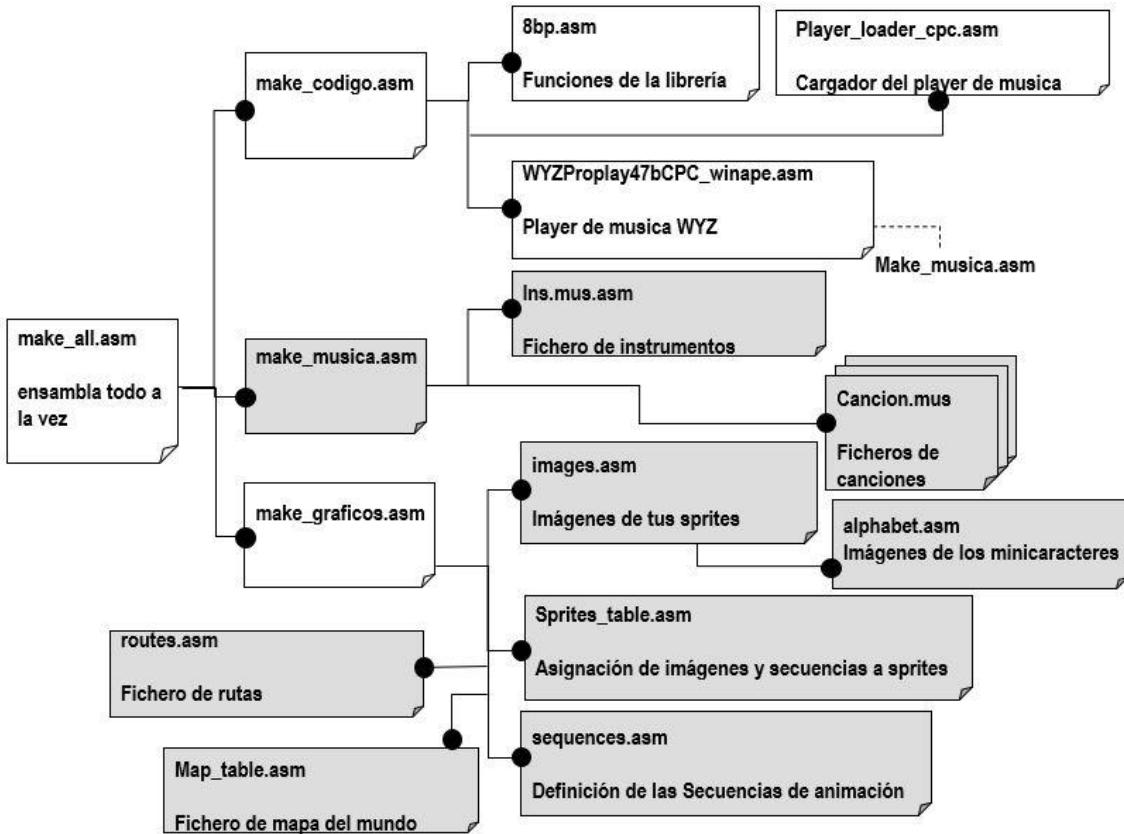
Como expliqué en el apartado de los “pasos” que debes dar, ésta será una versión de la librería específica para tu juego. Por ejemplo, el comando **|MUSIC,0,0,3,6** hará sonar la melodía número 3 que tú mismo has compuesto. La melodía numero 3 puede ser completamente diferente en otro juego. Lo mismo ocurre con los datos del fichero de instrumentos. Hay ciertas dependencias entre el código del player de música y las direcciones donde se ensamblan los datos de instrumentos y las melodías.

Es muy sencillo, pero hay que comprender la estructura de la librería para hacerlo, es decir, la estructura de los ficheros .asm que debes manejar y sus dependencias.

El siguiente diagrama presenta todos los ficheros .asm de un juego que use 8BP así como las dependencias entre ellos. En la figura, **en gris aparecen aquellos ficheros que tienes que editar**, como son:

- las canciones y fichero de instrumentos, que generas con el WYZtracker
- el fichero **make\_musica** donde indicas que ficheros “.mus” hay que ensamblar
- el fichero de imágenes que creas con el SPEDIT
- la tabla de sprites donde asignas imágenes a los sprites (aunque no es estrictamente necesario pues tienes el comando **|SETUPSP** )
- la tabla de secuencias, donde defines que imágenes conforman una secuencia,
- el mapa del mundo: donde defines hasta 64 elementos que conforman el mundo
- el fichero de rutas: donde defines las trayectorias de los sprites que quieras
- el fichero alphabet.asm: si quieres crear un alfabeto diferente al de serie de 8BP

Puedes ensamblarlo todo abriendo el fichero “**make\_all.asm**” y pulsando “assemble” en el menú de Winape. Después puedes usar el comando SAVE para salvar las imágenes, música y librería 8BP en diferentes ficheros binarios, o en uno solo, tal como hemos visto.



*Fig. 14 Ficheros para ensamblar*

Si sólo cambias los gráficos puedes ensamblarlos por separado, seleccionando el fichero “make\_graficos.asm” y pulsando assemble.

Si cambias las músicas debes re-ensamblar el código de la librería pues hay una dependencia entre el código y las canciones, debido a que el código necesita conocer donde comienza cada canción. Por ello si cambias o añades canciones debes ensamblar con `make_all.asm`. Esta dependencia la he reflejado con una línea de puntos entre el player y el fichero `make_musica.asm`

Puede que necesites ensamblar algo más, como un mapa de circuito de carreras que usa tus imágenes. En ese caso, añádelo al fichero “`Make_graficos.asm`” para que se ensamble después de `images.asm`. El orden de ensamblado es importante. Primero se deben ensamblar las imágenes, se le asocian etiquetas y después ya se pueden ensamblar los mapas o circuitos que usan dichas etiquetas.

## 6.1 ***Make\_all.asm***

Este es el fichero que permite ensamblar todo. Internamente invoca a tres ficheros que ensamblan en código de la librería y del player de música, las canciones y los gráficos.

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras una parte solo tienes que
; ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos
; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
;
```

```

; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 25000
;                               disponibles los comandos |LAYOUT, |COLAY
;
; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 24800
;                               disponibles los comandos |MAP2SP, |UMA
;
; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 24000
;                               disponible comando |3D
;
; ASSEMBLING_OPTION = 4 --> uso futuro

let ASSEMBLING_OPTION = 0
-----CODIGO -----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"

```

Cada uno de esos tres ficheros se encarga de ensamblar cosas diferentes y por ejemplo el de los gráficos invoca a otros ficheros como son el de imágenes, el de secuencias, el de rutas y el mapa del mundo

Usa siempre la opción de ensamblaje que te proporcione mas memoria libre. Si tu juego es de laberintos usa la opción 1, y si es con scroll, usa la 2. Si es un juego pseudo3D, usa la 3. En general no te recomiendo usar la opción 0 porque es la que menos memoria libre te deja y seguramente te sirve alguna de las demás opciones.

## 6.2 Estructura del fichero de imágenes

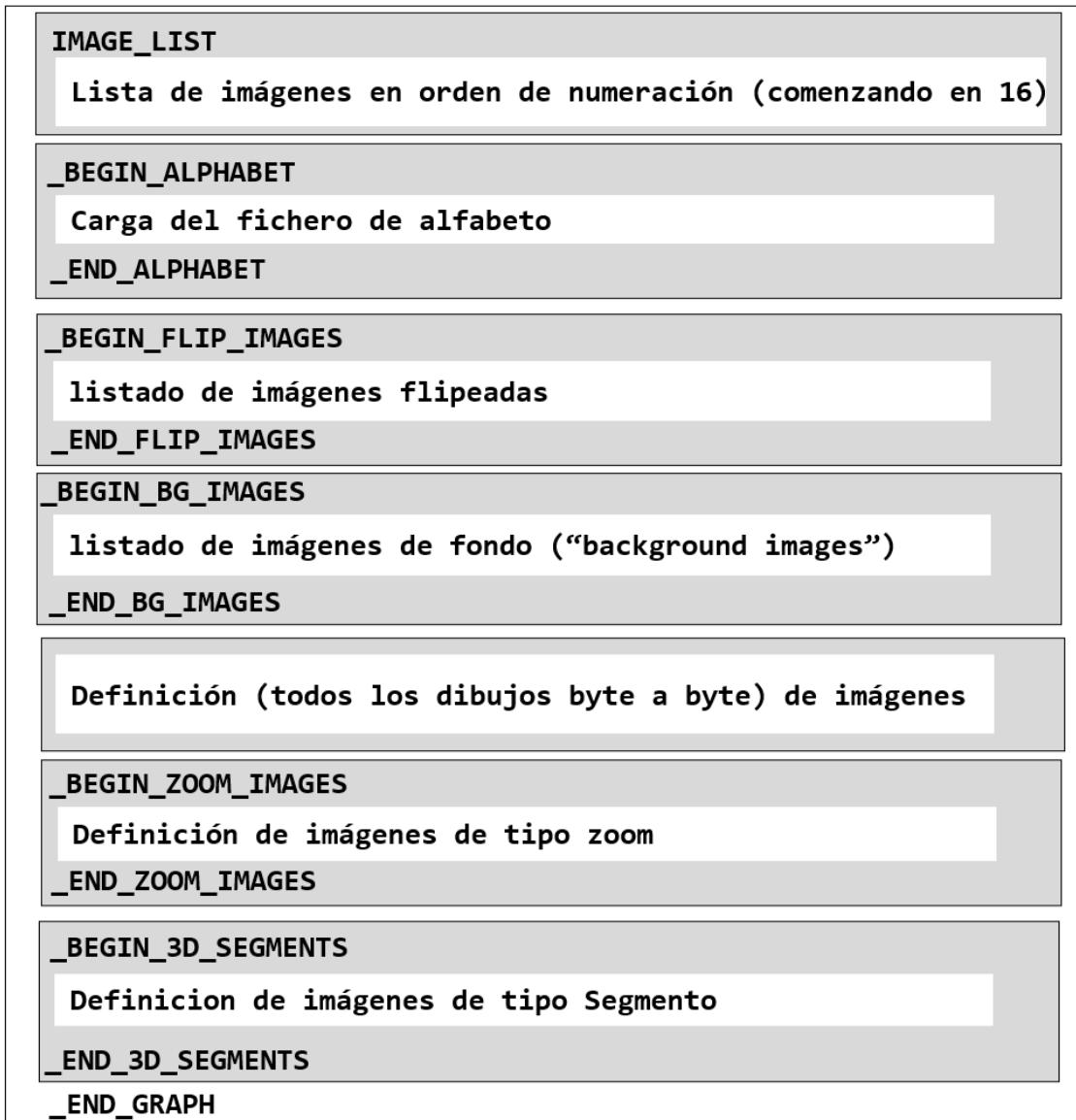


Fig. 15 estructura del fichero de imágenes

## 6.3 Estructura del fichero de secuencias de animación

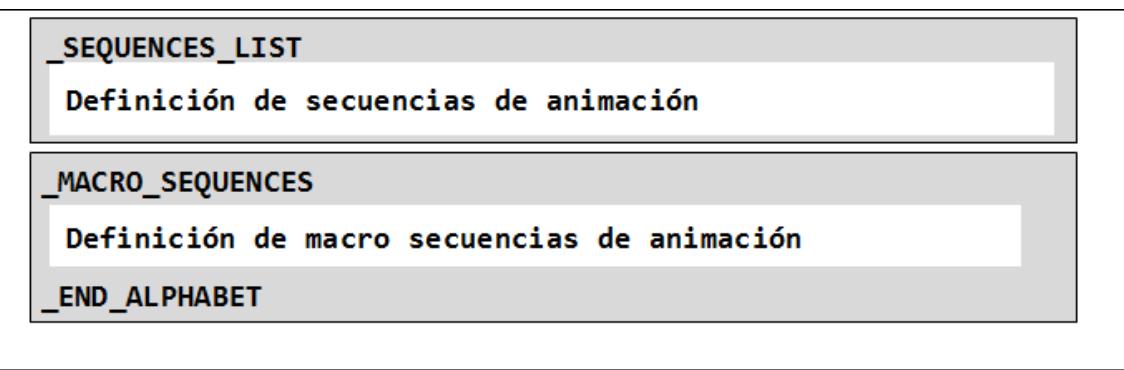


Fig. 16 estructura del fichero de secuencias de animación

## 6.4 Estructura del fichero de rutas

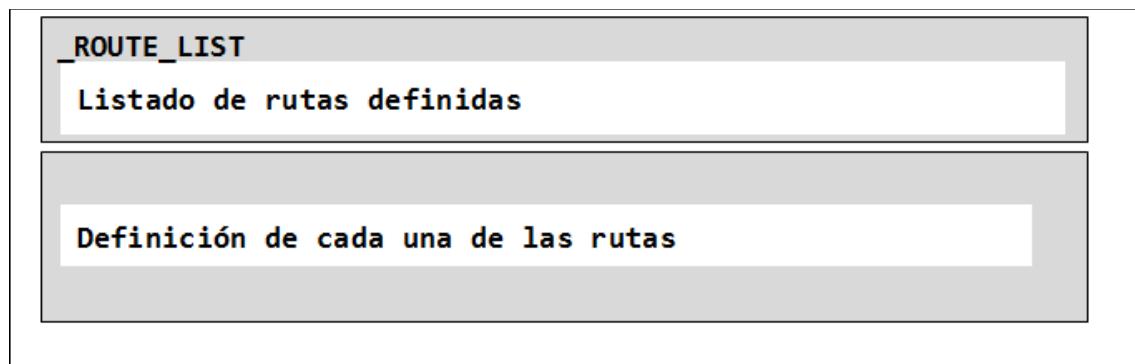


Fig. 17 estructura del fichero de rutas

## 6.5 Estructura del fichero de mapa del mundo

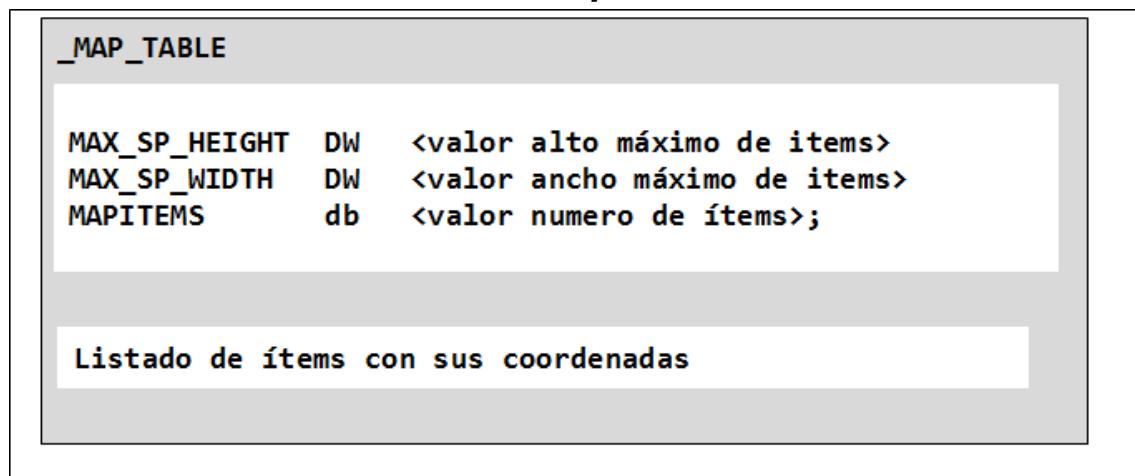


Fig. 18 estructura del fichero de mapa del mundo

## 7 Ciclo de juego

Un videojuego de arcade, plataformas, aventuras, generalmente tiene un tipo de estructura similar, en la que unas ciertas operaciones se van a repetir cíclicamente en lo que denominaremos “ciclo de juego”.

En cada ciclo de juego actualizaremos posiciones de sprites e imprimiremos en pantalla los sprites, de modo que el número de ciclos de juego que se ejecutan por segundo equivale a los “fotogramas por segundo” (FPS) del juego. El siguiente pseudo código esquematiza la estructura básica de un juego

### INICIO

```
Inicialización de variables globales: vidas, etc  
Espera a que el usuario pulse tecla de comienzo de juego  
GOSUB pantalla1  
GOSUB pantalla 2  
...  
GOSUB pantalla N  
GOTO INICIO
```

### Código de PANTALLA N (siendo N cualquier pantalla)

```
Inicialización de coordenadas de enemigos y personaje  
Pintado de la pantalla (layout), si procede
```

### BUCLE PRINCIPAL (ciclo del juego en esta pantalla)

```
Lógica de personaje : Lectura de teclado y actualización de  
coordenadas del personaje y si procede, actualización de su  
secuencia de animación
```

```
Ejecución de lógica de enemigo 1  
Ejecución de lógica de enemigo 2  
...  
Ejecución de lógica de enemigo n
```

```
Impresión de todos los sprites  
Condición de salida de esta pantalla (IF ... THEN RETURN)  
GOTO BUCLE PRINCIPAL
```

*Fig. 19 Estructura básica de un juego*

Si la lógica de los enemigos es muy pesada por haber muchos enemigos o por ser muy compleja, esto consumirá más tiempo en cada ciclo del juego y por lo tanto el número de ciclos por segundo se verá reducido. Intenta no bajar de 10fps para que el juego mantenga un nivel de acción aceptable.

### 7.1 Como medir los FPS de tu ciclo de juego

Para saber si tu juego tiene un nivel de acción aceptable, nada mejor que jugar a él y si te gusta, pues estará bien. Sin embargo, quizás quieras medir exactamente cuántos frames por segundo es capaz de generar tu videojuego, porque así puedes tomar decisiones en la lógica de tu programa y medir cuánto perjudican o benefician esas decisiones de programación.

Lo que haremos para medir es simplemente tomar nota del instante de tiempo antes de empezar el primer ciclo de juego, en el principio del “código de la pantalla N”. Luego tomaremos nota del tiempo tras unos cuantos ciclos de juego y haremos una sencilla división. Vamos a verlo paso a paso:

**A=TIME : rem esta línea almacena en la variable A el tiempo en 1/300 fracciones de segundo**

El número que se va a almacenar en A puede ser un número muy grande, de hecho, puede ser mayor que lo que es capaz de almacenar una variable entera como es “A”. Para que la asignación no produzca error, es conveniente resetear el temporizador del AMSTRAD, que se inicia cada vez que arrancamos la máquina. Para resetearlo, antes de asignar la variable “A”, simplemente ejecuta:

**En un CPC 6128**

**POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0**

**En un CPC 464**

**POKE &b187,0: POKE &b188,0: POKE &b189,0: POKE &b18a,0**

Para diferenciar en que maquina esta tu programa debes desactivar la música y consultar una dirección con PEEK

**|MUSIC: If peek(&39)=57 then Modelo=464 else modelo=6128**

**If modelo=464 then ...**

Con ello habrás puesto a cero las direcciones de memoria donde el AMSTRAD almacena el temporizador. Después, ejecutamos tantos ciclos de juego como queramos, y controlamos en que ciclo estamos con la variable “ciclo”, que incrementaremos en una unidad en cada ciclo. Tras salir de esa fase o pantalla, ejecutamos :

**FPS= ciclo \* 300/ (TIME - A)**

Y ya tenemos los FPS de nuestro juego. Te lo pongo todo en orden a continuación:

**Rem suponemos que estamos en un 6128**

**POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0**

**A=TIME**

**<aquí va el programa que ejecuta tu ciclo de juego, incluyendo ciclo=ciclo+1 >**

**Llegaremos aquí tras la condición de salida de la pantalla**

**FPS= ciclo \* 300/ (TIME - A)**

**PRINT “FPS =”;FPS**

Para que quede claro: El tiempo transcurrido desde que empezó el programa hasta que ha terminado es TIME-A expresado en 1/300 fracciones de segundo. Para pasarlo a segundos hay que dividirlo por 300

**Segundos= (TIME -A) /300**

Si en dichos segundos se han ejecutado n ciclos (por ejemplo), entonces un ciclo ha tardado: **Tc= 300\*(TIME-A)/n**

Y el numero de ciclos que se pueden ejecutar en un segundo (los FPS) es la inversa, es decir, **FPS = 1/Tc = n\*300/(TIME-A)**

## 8 Sprites

### 8.1 Editar sprites con SPEDIT y ensamblarlos

Spedit (Simple Sprite Editor) es una herramienta que te va a permitir crear tus imágenes de personajes y enemigos y usarlos en tus programas BASIC

Spedit está hecha en BASIC, y es muy sencilla, de modo que puedes modificarla para que haga cosas que no están contempladas y te interesen. Se ejecuta en el Amstrad CPC, aunque está pensada para que la utilices desde el emulador winape.

Lo primero que debes hacer es configurar winape para que la salida de la impresora la saque a un fichero. En este ejemplo he puesto la salida de la impresora al fichero printer5.txt

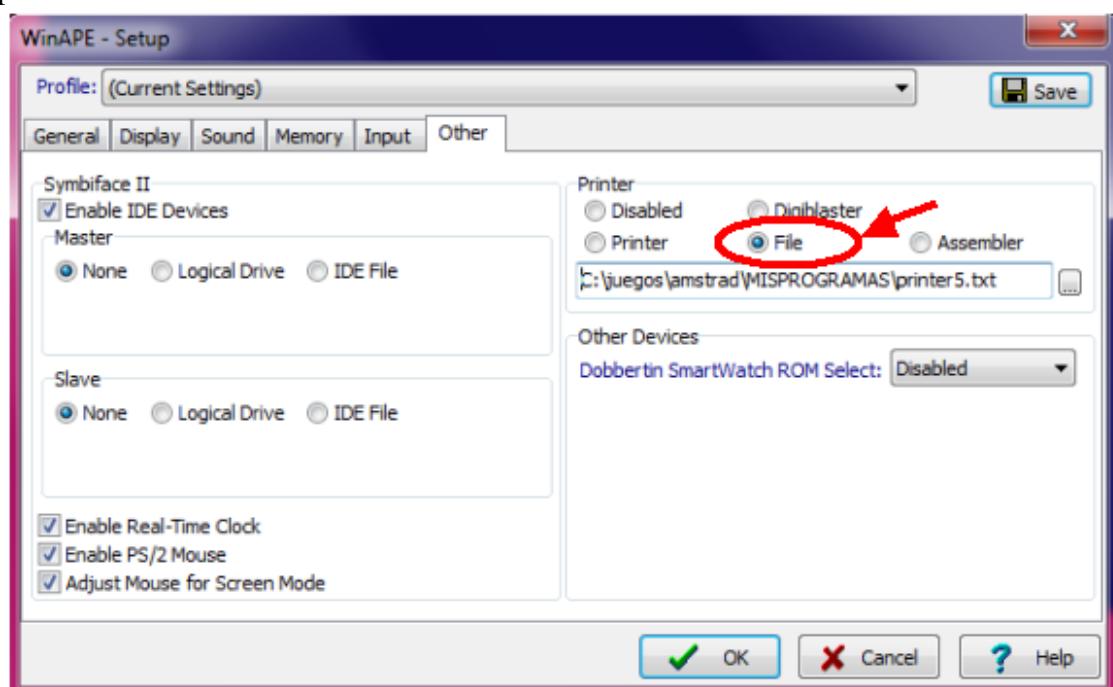


Fig. 20 Redirección de la impresora del CPC a un fichero con Winape

Cuando ejecutes SPEDIT te aparecerá el siguiente menú, donde puedes elegir entre editar un Sprite o capturar un Sprite desde un fichero de imagen (.scr).

La captura de sprites está disponible desde SPEDIT V14. En caso de que desees capturar un Sprite, necesitas tener en el disco un fichero de imagen (.scr) que no es mas que un binario con 16384 bytes. Puede ser una imagen de un juego que hayas capturado donde aparezcan imágenes de personajes que te gusten y no quieras invertir tiempo en editar.

En caso de que escojas editar un Sprite, el programa te pregunta por la paleta a usar. Puedes escoger una paleta por defecto o bien una tuya que quieras definir. También puedes usar una que previamente hayas guardado (se guarda siempre en pal.dat, basta con pulsar "i" mientras editas un sprite). Si decides definir tu propia paleta, deberás reprogramar las líneas de BASIC donde se define la paleta alternativa (o "custom"), que

es una subrutina a la que se invoca con GOSUB cuando pulsas “2” en la respuesta a la pregunta sobre qué paleta deseas usar.

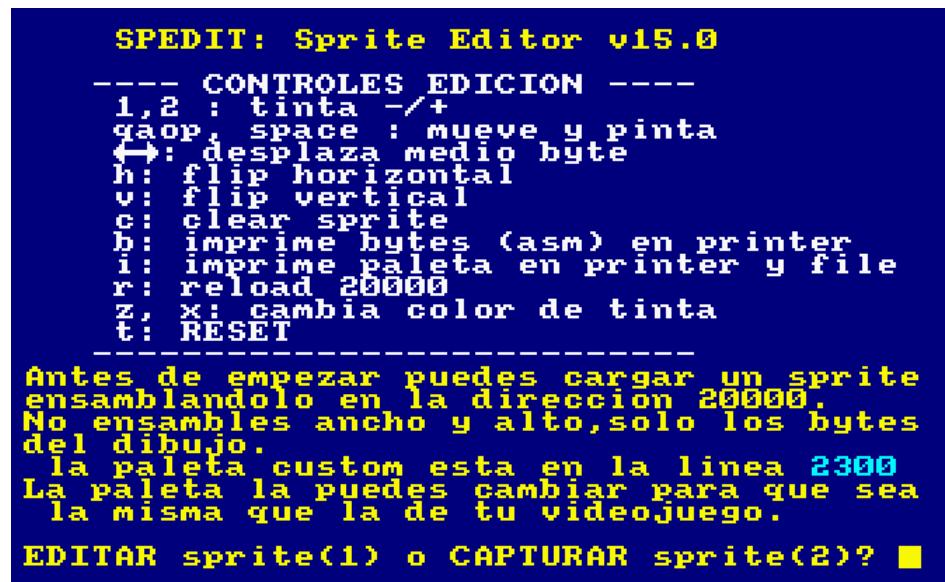


Fig. 21 Pantalla inicial de SPEDIT

La herramienta te permite elegir mode 1 o mode 0. Una vez en edición te permite editar dibujos, con la ayuda en la pantalla. Manejas un píxel que parpadea y en la parte inferior se muestra las coordenadas donde te encuentras y el valor del byte en el que te encuentras.

Cuando te pregunte el ancho y alto del sprite, recuerda que **la altura máxima de un sprite en 8BP es 127 líneas**, al igual que su anchura máxima en bytes. También ten en cuenta que la anchura te la pregunta en pixeles, pero debes saber que el editor internamente trabaja en bytes, de modo que, si vas a hacer una imagen en mode 0, el ancho debe ser un numero par (un byte = 2 pixeles) y si vas a hacer una imagen en mode 1, el ancho debe ser un múltiplo de 4 (un byte= 4 pixeles).

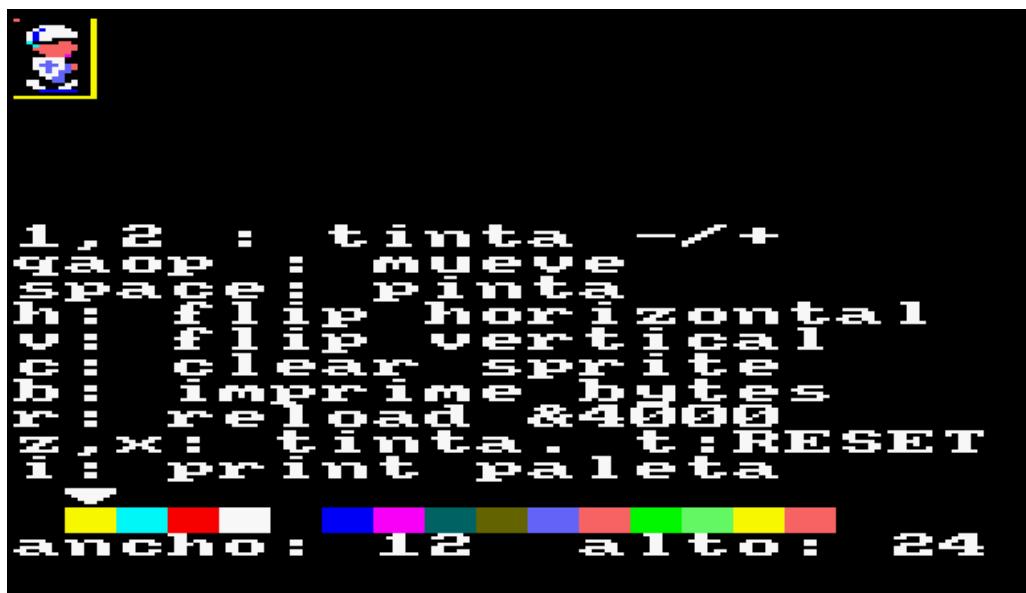
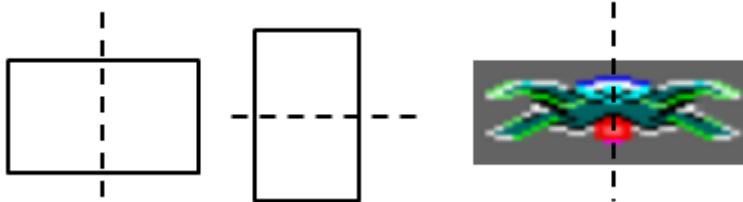


Fig. 22 Pantalla de edición de SPEDIT

SPEDIT te permite “espejar” tu imagen para hacer el mismo muñeco caminando hacia la izquierda sin esfuerzo, basta con pulsar H (flip horizontal) y lo mismo se puede hacer en vertical. También te permite “espejar la imagen” respecto de un eje imaginario situado en el centro del personaje, tanto en vertical como en horizontal. Esto es muy útil para personajes simétricos o casi simétricos, donde una ayuda al dibujarlo siempre viene bien.



*Fig. 23 sprites simétricos con SPEDIT*

Desde la versión 11 de SPEDIT, el modo 1 de AMSTRAD está soportado, de modo que puedes editar sprites en mode 1 sin problema, y usar también el mecanismo de espejado.



*Fig. 24 edición de sprites en MODE 1 con SPEDIT*

Una vez que has definido tu muñeco, para extraer el código ensamblador deberás pulsar la “b”. Esto mandará a la impresora (al fichero que hayamos definido como salida) un texto como el siguiente, al que puedes añadir un nombre, yo le he llamado “SOLDADO\_R1”

```

;----- BEGIN IMAGE -----
SOLDADO_R1
db 6 ; ancho
db 24 ; alto
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;----- END IMAGE -----

```



Fíjate como he dejado siempre un byte a la izquierda a cero. Lo he hecho para que, al mover el soldado hacia la derecha, se “borre a sí mismo”, ya que de lo contrario dejaría un rastro, “manchando” la pantalla mientras avanza.

*Fig. 25 Soldado en formato .asm*

Una vez que has hecho el primer fotograma de tu soldado puedes dejar el trabajo y continuar otro día. Para partir del soldado que has dibujado y continuar retocándolo o bien modificarlo para construir otro fotograma, puedes ensamblar el soldado en la dirección **20000**, quitando el ancho y el alto. Una vez ensamblado desde winape, le dices a SPEDIT que vas a editar un sprite del mismo tamaño y una vez estés en la pantalla de edición pulsa “r” (reload). El sprite se cargará desde la dirección **20000**, que es donde lo has “ensamblado”.

Gran parte del atractivo de un juego son sus sprites. No escatimes tiempo en esto, hazlo despacio y con gusto y tu juego parecerá mucho mejor.

```

.org 20000
;----- BEGIN IMAGE -----
SOLDADO_R1
;db 6 ; ancho ojo! comentamos esta linea
;db 24 ; alto ojo! comentamos esta linea
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;----- END IMAGE -----

```

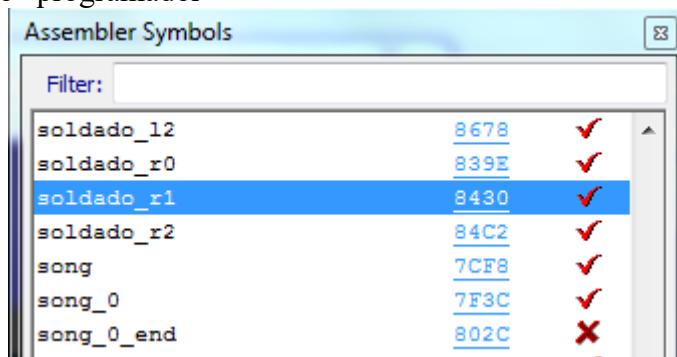
Con esto ya sabes lo que significa “ensamblar” un sprite. Es simplemente meter los bytes de datos que lo constituyen en direcciones de memoria consecutivas, en este caso comenzando por la **20000**

SPEDIT ocupa muy poca memoria y esa dirección está muy lejos del programa de modo que no hay problema de que al ensamblarlo estemos dañando” el programa SPEDIT.

*Fig. 26 Ensamblado de gráficos*

Para saber en qué dirección de memoria se ha ensamblado cada imagen, utiliza desde el menú de winape: Assemble->symbols

Con ello veras una relación de las etiquetas que has definido, como “SOLDADO\_R1” y la dirección de memoria (en hexadecimal) a partir de la cual se ha ensamblado. Para transformar las direcciones de hexadecimal a decimal puedes usar la calculadora de windows en modo “programador”



*Fig. 27 detalle de lo que muestra symbols en Winape*

Una vez que hayas hecho las diferentes fases de animación de tu soldado, puedes agruparlas en una “secuencia” de animación. Las secuencias de animación son listas de imágenes y no se definen con SPEDIT. Con SPEDIT simplemente editas los

“fotogramas”. En un apartado posterior te explicaré como decirle a la librería 8BP qué conjunto de imágenes constituyen una secuencia de animación.

Las imágenes que vayas haciendo para tu juego ve guardándolas todas en un único fichero, que se titule “images\_mijuego.asm”, por ejemplo. Dicho fichero comienza con una lista de imágenes que puedes referenciar en los comandos 8BP en BASIC con un índice, con independencia de la dirección en la que se ensamblen, por ejemplo:

```
IMAGE_LIST  
; la primera imagen siempre se asignará al índice 16, la siguiente al  
17 y así sucesivamente  
-----  
dw SOLDADO_R0; 16  
dw SOLDADO_R1; 17  
dw SOLDADO_R2; 18
```

Una vez que estén todas las imágenes hechas podrás ensamblar la librería junto con la música y los graficos.

**MUY IMPORTANTE:** asegúrate de no exceder los 8440 bytes de gráficos. Para ello comprueba donde se ha ensamblado la etiqueta “\_END\_GRAPH”, la cual debe ser inferior a 42040 (ya que  $42040 - 33600 = 8440$  bytes). Si se ensambla en una dirección superior entonces estás “machacando” direcciones que necesita el interprete BASIC y el ordenador podrá bloquearse. En caso de necesitar más memoria para gráficos deberás ensamblar los gráficos “extra” en una zona de memoria no ocupada, por ejemplo, en la 22000, y usar en tu programa un MEMORY 21999, reduciendo la memoria disponible para el BASIC

## 8.2 Imprimir un sprite

Vamos a ver lo básico para imprimir un sprite. Supongamos que has dibujado un soldado lo has incluido en el fichero images\_mygame.asm. Vamos a suponer que es tu primera imagen y que por lo tanto tiene el identificador 16.

En 8BP dispones de 32 sprites (numerados del 0 al 31). Puedes asignar una imagen a cualquier sprite con el comando |SETUPSP. Este comando te permite modificar muchos atributos de un sprite, no solo su imagen. El atributo que quieras modificar del sprite es el segundo parámetro del comando SETUPSP

```
|SETUPSP, <sprite id>, <parámetro> , <valor>
```

Para asignar una imagen debes usar el parámetro 9. Un programa muy sencillo que imprime un sprite sería el siguiente:

```
10 MEMORY 23499  
20 CALL &6B78: REM instala los comandos RSX  
30 DEFINT A-Z : REM variables numericas enteras (mas rápidas)  
40 |SETUPSP,31,9,16: REM asigna la imagen 16 al sprite 31  
50 x=40:y=100: REM coordenadas donde queremos imprimir  
60 |LOCATESP,31,y,x: REM coloca el sprite 31  
70 |PRINTSP,31: REM imprime el sprite 31
```

Este sería el resultado

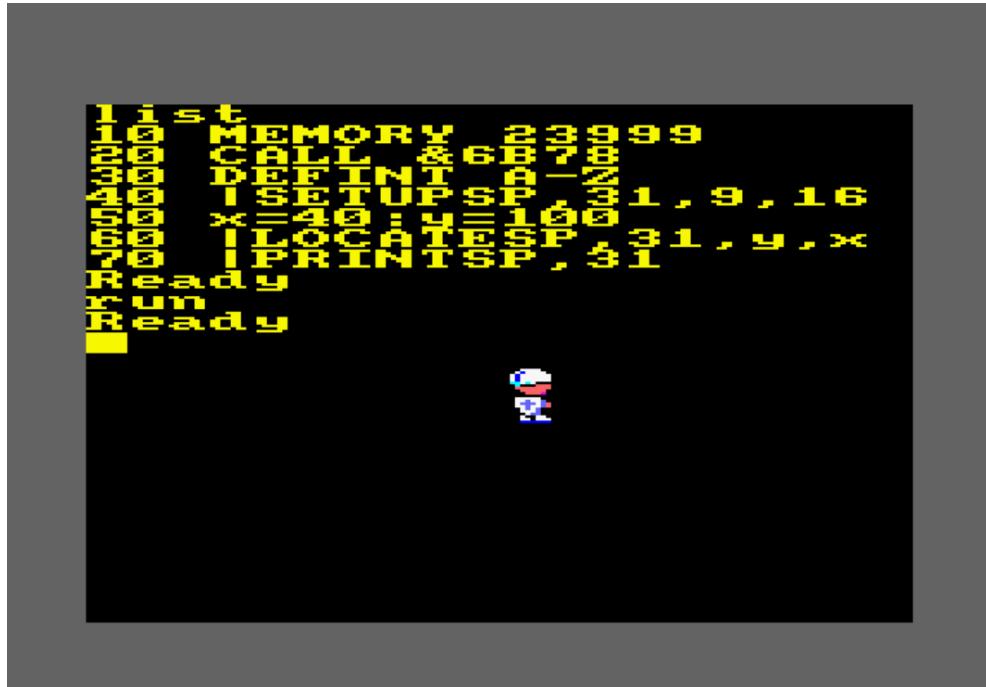


Fig. 28 Impresión de un sprite en pantalla

El comando **|SETUPSP** lo vas a usar mucho y poco a poco lo irás conociendo en profundidad. Los parámetros de SETUPSP no son cualquier número. Hay 7 valores posibles y son los siguientes (0,5,6,7,8,9,15):

- Parámetro 0: cambia el byte de estado (el “status”) del sprite
- Parámetro 5: cambia Vy. También se puede modificar Vx a la vez si lo añadimos al final como parámetro extra ( así : SETUPSP, <id>,5, Vy,Vx )
- Parámetro 6: cambia Vx
- Parámetro 7: cambia secuencia (toma valores 0..31)
- Parámetro 8: cambia frame\_id (toma valores 0..7)
- Parámetro 9: cambia imagen. La imagen especificada puede ser una de la lista inicial de imágenes del fichero images\_mygame.asm,
- Parámetro 15: cambia la ruta (ocupa 1bytes)

A medida que avances en la lectura de este manual irás comprendiendo el significado de los atributos de un sprite, y como usarlos en función de tus necesidades.

### 8.3 Sprite flipping

En muchas ocasiones necesitarás dibujar personajes que caminan en diferentes direcciones, con diferentes imágenes para cada caso. La imagen del sprite en dirección izquierda será la imagen especular de la dirección derecha. Se pueden definir dos imágenes y almacenarlas en memoria, pero desde la V33, hay una forma de evitar el consumo de memoria RAM para estas imágenes. Se trata de las imágenes “flipeadas”.



*Fig. 29 ejemplo de imágenes flipeadas*

Una imagen “flipeada” es la imagen especular de otra imagen que se ha creado e incluido en el fichero de imágenes. Definiendo una imagen de esta manera, se evita tener que almacenarla. Para hacerlo, simplemente debes incluir una lista de imágenes flipeadas dentro del fichero de imágenes (al que normalmente llamo “images\_mygame.asm”). Encontrarás al principio del fichero una sección delimitada por las etiquetas “`BEGIN FLIP_IMAGES`” y “`END FLIP_IMAGES`” destinada a este propósito.

```

;-----
; BEGIN_FLIP_IMAGES
; aqui pon las imagenes que se definen como otras existentes pero
; flipeadas horizontalmente.
JOE_LEFT dw JOE_RIGHT; joe_left sera la version flipeada de joe_right

; los frames del soldado a la izquierda los defino como flipeados
SOLDADO_L0 dw SOLDADO_R0;
SOLDADO_L1 dw SOLDADO_R1;
SOLDADO_L2 dw SOLDADO_R2;
SOLDADO_L1_UP dw SOLDADO_R1_UP
SOLDADO_L1_DOWN dw SOLDADO_R1_DOWN

_END_FLIP_IMAGES
;-----
```

Las imágenes flipeadas las puedes usar igual que si fuesen imágenes normales. Mas adelante encontrarás como crear secuencias de animación, las cuales puedes construir con imágenes flipeadas y no flipeadas. A todos los efectos, es como si una imagen “flipeada” fuese real, aunque se trata de una imagen “virtual”, que no está almacenada y que al imprimirla se calcula como la imagen especular de otra que sí existe. Las imágenes flipeadas se soportan tanto en mode 0 como en mode 1.

El inconveniente de las imágenes flipeadas es que su impresión tiene mayor coste, concretamente consume un 1.8 veces el tiempo que consume una impresión normal, lo cual se podría traducir en una menor velocidad de tu juego. Si tu juego es un arcade (un “shoot’em up”) en el que necesitas la máxima velocidad, mi recomendación es no usar imágenes flipeadas masivamente. Sin embargo, en juegos de aventuras, de pasar pantallas, de laberintos, etc. es una excelente opción. De todos modos, prueba en tu arcade a usarlas pues si no hay muchas flipeadas a la vez, la velocidad resultante puede ser muy aceptable.

He hecho el flipping horizontal y no el vertical porque normalmente un personaje que camina a la izquierda es la imagen especular del mismo caminando hacia la derecha, mientras que si sube muestra la espalda y al bajar muestra el pecho y la cara. Por lo tanto, el flipping vertical no es tan útil como el horizontal, y en aras de reducir el tamaño de 8BP, no lo he incluido entre sus capacidades.

**IMPORTANTE:** el flipping no es aplicable a las imágenes de tipo segmento que se pueden usar en el modo pseudo-3D de 8BP

## 8.4 Sprites con sobreescritura

Desde la versión v22 de 8BP es posible editar sprites transparentes, es decir, que pueden sobrevolar un fondo y lo restablecen al pasar. Para ello los sprites que disfrutan de esta posibilidad deben ser configurados con un “1” en el flag de sobreescritura del byte de estado (bit 6). En el siguiente apartado se detallará debidamente el byte de status. Veamos cómo se edita un sprite con esta capacidad con SPEDIT.

Muchos juegos utilizan una técnica llamada “doble buffer” para poder restablecer el fondo cuando un sprite se mueve por la pantalla. Se basa en tener una copia de la pantalla (o del área de juego) en otra zona de memoria, de modo que, aunque nuestros sprites destruyan el fondo, siempre podemos consultar en dicha área que había debajo y así restablecerlo. En realidad, ese es el principio básico, pero es algo más complejo. Se imprime en el doble buffer (también llamado “backbuffer”) y cuando ya está todo impreso, se vuelca a la pantalla o bien se hace commutar la dirección de comienzo de la memoria de video desde la dirección original de pantalla a la nueva, la del doble buffer. La commutación es instantánea (depende del tipo de máquina). Para construir el siguiente fotograma se usa la dirección de pantalla original donde ahora ya no está apuntando la memoria de video. Allí se construye el nuevo fotograma y se vuelve a commutar, alternativamente, en cada fotograma. Estas técnicas, aunque funcionan muy bien, tienen un par de desventajas para nuestros propósitos: llevan más tiempo de CPU y consumen mucha más memoria (hasta 16KB adicionales), dejándonos muy poca memoria para nuestro programa BASIC. Si un juego se desarrolla enteramente en ensamblador, esto no es tan grave porque 10KB de ensamblador dan para mucho, pero 10KB de BASIC es poco. Algunos videojuegos reducen el área de juego para no gastar tanta memoria, pero eso los hace algo mas pobres.

La solución adoptada en 8BP está inspirada en el programador Paul Shirley (autor de “misión Genocide”, pero es ligeramente diferente. Contaré directamente la de 8BP:



Fig. 30 Sprites con sobreescritura en 8BP

La idea consiste en que **el fondo nunca es destruido por los sprites que pasan por encima**, por lo que no es necesario guardarla. Esta aparente “magia” tiene su lógica: consiste en “esconder” el color de fondo en el color del sprite que se pinta sobre él.

En el AMSTRAD un pixel de mode 0 es representado con 4 bits, por lo que son posibles hasta 16 tintas diferentes de una paleta de 27 colores. Pues bien, si usamos un bit para el color de fondo y 3 para los colores de los sprites, tendremos un total de 2 colores de fondo + 7 colores + 1 color para indicar transparencia = 9 colores en total. Esto nos va a permitir “esconder” el color de fondo en el color del sprite, aunque pagamos el precio de reducir el número de colores de 16 a tan sólo 9. Además, el fondo solo podrá ser de dos colores. Sin embargo, ciertos elementos ornamentales de la pantalla de juego pueden tener más color, pues los sprites no pasarán por encima (como las hojas de los árboles o el tejado del ejemplo siguiente), de modo que podemos conseguir cierta dosis de colorido en nuestro juego.

Para editar este tipo de sprites debemos usar una paleta adecuada, de 9 colores, donde para cada color de sprites se usan dos códigos binarios (los correspondientes al 0 y 1 del bit de fondo). En el SPEDIT si eliges la opción de paleta “2”, tendrás una paleta definida de ese modo, aunque puedes cambiarla a tu gusto. Se ha construido así:

```

2300 REM ----- PALETA sprites transparentes MODE 0 -----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : REM negro
2304 INK 3,0:
2305 INK 4,26: REM blanco
2306 INK 5,26:
2307 INK 6,6: REM rojo
2308 INK 7,6:
2309 INK 8,18: REM verde
2310 INK 9,18:
2311 INK 10,24: REM amarillo
2312 INK 11,24:
2313 INK 12,4: REM magenta
2314 INK 13,4:
2315 INK 14,16 : REM naranja
2316 INK 15,16:
2317 AMARILLO=10
2420 RETURN

```

*Fig. 31 Paleta ejemplo de sobreescritura*

Como ves, tras el color 0 y 1, todos los colores se repiten dos veces. Tú puedes construir tu propia paleta de este modo. Puedes ayudarte consultando el apéndice de este manual dedicado a la paleta de color.

<table border="1"><tr><td>000</td><td>1</td></tr></table>	000	1	=		Color de fondo	Paleta ejemplo
000	1					
<table border="1"><tr><td>110</td><td>0</td></tr></table>	110	0	=		Color de sprite	0000 =  0001 =  1100 =  1101 = 
110	0					
<i>Cuando el sprite se imprime:</i>						
Fondo OR sprite = 1101 =						
<i>Cuando el sprite se marcha:</i>						
Pixel OR 0001 = 0001 =						
<i>El fondo nunca fue destruido, estaba “escondido” en el sprite</i>						

La técnica se podría resumir diciendo que en realidad **el fondo nunca es destruido por los sprites**, sino que se “esconde” en los propios sprites que se imprimen sobre el fondo

*Fig. 32 Mecanismo de sobreescritura en 8BP*

Con el editor SPEDIT puedes modificar la paleta a tu gusto sin necesidad de editar manualmente con comandos INK, y permite exportarla para copiarla en nuestros programas BASIC. La exportación se realiza mandando a la impresora los comandos INK que conforman la paleta (la impresora la redirigimos a un fichero desde winape). Disponemos de las teclas z/x para alterar la paleta y de la opción "i" para exportarla al fichero de salida. Este es un ejemplo de lo que exporta (es una paleta sin sobreescritura):

```
' ----- BEGIN PALETA -----
INK  0 , 1
INK  1 , 24
INK  2 , 20
INK  3 , 6
INK  4 , 26
INK  5 , 0
INK  6 , 2
INK  7 , 8
INK  8 , 10
INK  9 , 12
INK  10 , 14
INK  11 , 16
INK  12 , 18
INK  13 , 22
INK  14 , 0
INK  15 , 11
' ----- END PALETA -----
```

Al pulsar la Tecla “i” , además se guarda el fichero “pal.dat” en el disco (el .dsk) de modo que la puedes cargar posteriormente eligiendo la opción 3 para contestar a la pregunta de elección de paleta.

Los sprites que uses para construir los dibujos del fondo sólo podrán tener los colores 0 y 1 pero los sprites que uses para ornamentar, por donde no vayan a pasar los sprites en movimiento pueden usar los 9 colores.

También puedes aumentar el colorido de los decorados con elementos que sean sprites en lugar de fondos, como el caldero verde del ejemplo anterior. De este modo podrás tener resultados muy coloristas.

La tinta 0001 tiene un uso “especial”. Si editas un sprite que no use el flag de sobreescritura, la tinta 1 será simplemente un color. Pero si editas un sprite con flag de sobreescritura activo en su byte de status, al imprimirse se dejarán sin pintar esos píxeles, respetando lo que hubiese debajo. Eso permite que las colisiones entre sprites no sean “rectangulares”, sino que conserven la forma del sprite.

code	Significado	code	significado
0000	Color 1 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, significa "transparencia", es decir, imprimir restableciendo el fondo	0110	color 3 de sprite
0001	Color 2 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, deja de significar un color para significar "no imprimir". Se respeta lo que haya en ese pixel, por ejemplo, un pixel coloreado por otro sprite anteriormente impreso con el que nos estamos solapando.	0111	
0010	color 1 de sprite	1000	color 4 de sprite
0011		1001	
0100	color 2 de sprite	1010	color 5 de sprite
0101		1011	
		1100	color 6 de sprite
		1101	
		1110	color 7 de sprite
		1111	

9 colores en total:

- 2 de fondo
- 7 para sprites (en realidad 8 pero uno -000- significa transparencia)
- Los elementos ornamentales pueden usar los 9.

A continuación, voy a mostrarte un sprite donde he pintado de tinta 0001 lo que no se va a pintar, es decir, donde ni siquiera se va a restablecer el fondo, ya que con el resto de píxeles a 0000 ya es suficiente para borrar el rastro del sprite mientras se desplaza.

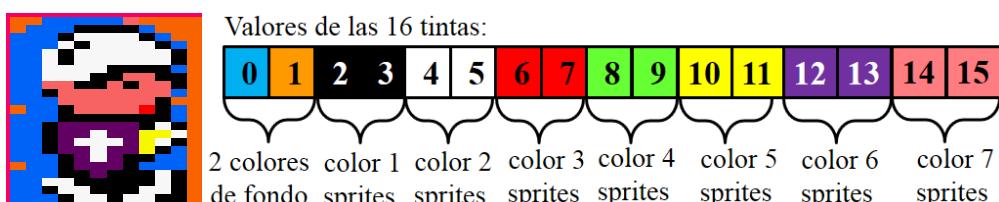


Fig. 33 sprite y paleta diseñados para sobreescritura

#### IMPORTANTE, CUANDO EDITES TUS SPRITES CON SOBREESCRITURA:

No uses los bits de fondo para las tintas de tus sprites, a menos que busques efectos especiales como los que se describen más adelante en este capítulo. Si no sigues esta regla, quizás notes efectos "raros". Es decir:

- Si el fondo tiene 1 bit, entonces colorea tus sprites con tintas que acaben en 0 (es decir, las tintas 2,4,6,8,10,12,14)
- Si el fondo tiene 2 bits, entonces colorea tus sprites con tintas que acaben en 00 (es decir las tintas 0100,1000,1100 que son las tintas 4, 8 y 12 respectivamente)

Como podrás imaginar, en el caso del caldero, al ser un sprite que no se desplaza y por lo tanto no se borra a sí mismo, todo su contorno está pintado con la tinta 0001. Ello permite colisiones perfectas, sin formas rectangulares que evidencian que en realidad los sprites

son rectángulos. El resultado final es el que se muestra a continuación en una colisión múltiple.



Como habrás podido adivinar, la colisión además de ser perfecta, evidencia que los sprites han sido ordenados según su coordenada Y, de modo que el último en imprimirse es el ubicado en la posición más inferior. Esto se hace con un simple parámetro al imprimir los sprites con el comando |PRINTSPALL, que veremos más adelante.

*Fig. 34 Colisión múltiple, efecto de la tinta 0001*

Las operaciones de impresión con este mecanismo son muy rápidas, sin necesidad de definir lo que se conoce como “máscaras de sprites”. Las máscaras son mapas de bits del tamaño de un sprite que sirven para acelerar las operaciones de impresión. En este caso no son necesarias. La siguiente figura representa una típica máscara asociada a un sprite. Primero se suele hacer la operación AND entre el fondo y la máscara y después se hace el OR con el sprite. En 8BP es más rápido, pues el sprite no toca el bit destinado al fondo, de modo que la operación OR entre el fondo y el sprite respeta el fondo a la vez que pinta el sprite. Si no entiendes esto muy bien, no te preocupes, no es importante entenderlo pues no es necesario en 8BP.

sprite	mask	Metodo convencional:
0 2 2 0	1 0 0 1	Se imprime
2 3 3 2	0 0 0 0	Fondo AND mask OR sprite
0 2 2 0	1 0 0 1	
0 2 2 0	1 0 0 1	
0 0 0 0	1 1 1 1	

*Fig. 35 En 8BP no son necesarias máscaras*

La impresión de sprites con el flag de sobreescritura activo es más costosa que la impresión sin sobreescritura. A pesar de no requerir máscara y ser muy rápida, esta impresión consume aproximadamente 1.6 veces el tiempo que consume la impresión de un sprite sin sobreescritura. Por ese motivo, úsala cuando sea necesario, y no la uses si tu juego no va a tener un dibujo de fondo que los sprites deban respetar. La combinación de sobreescritura y flipping es aún más costosa (consume 2.2 veces el tiempo de una impresión normal sin sobreescritura ni flipping) de modo que tenlo en cuenta en tus juegos.

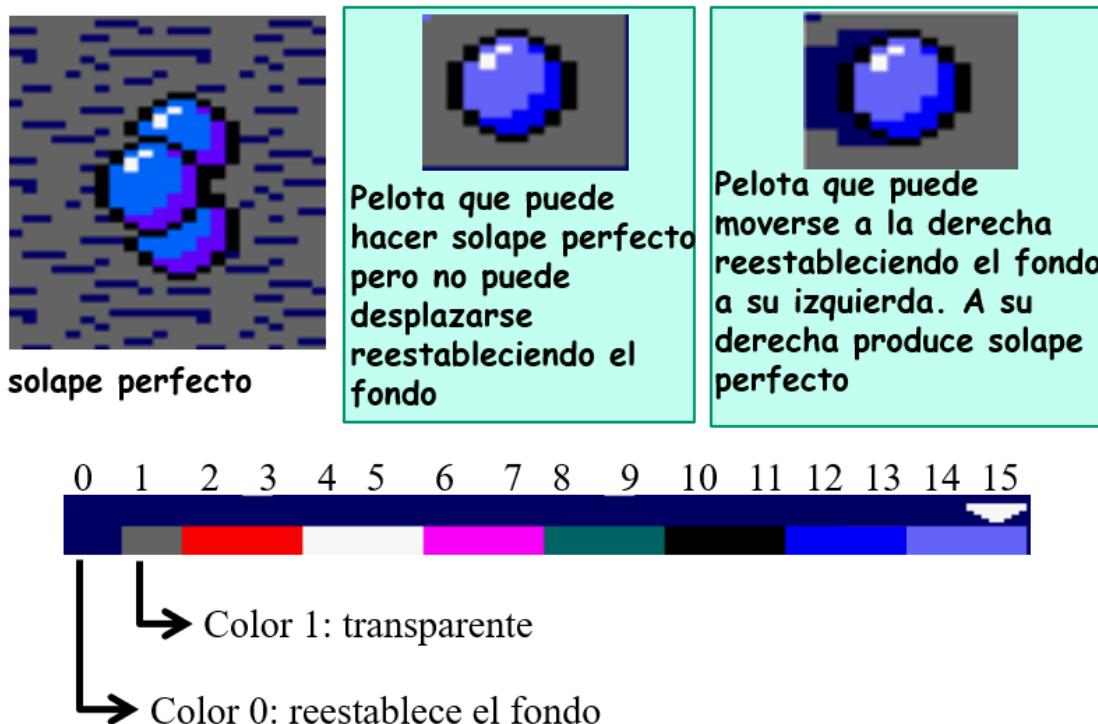
#### 8.4.1 Uso de sobreescritura para mejorar los solapes de sprites

Una característica muy valiosa de la sobreescritura es que permite que los solapes entre sprites sean “perfectos”, ya que si al editar el sprite (al que vas a dotar de sobreescritura) usas la tinta 1 alrededor, cuando lo imprimas todos los pixeles que tengan tinta 1 se

convertirán en “transparentes”, es decir, que si previamente había pintado otro sprite, sus píxeles se respetarán, dando lugar a solapes “perfectos”. Puede que esta característica de los sprites con sobreescritura te interese, aunque tu juego no requiera sobreescritura porque tenga un fondo negro o monocolor.

Solo se borrarán (reestableciendo el fondo) aquellos píxeles que hayas dibujado con tinta cero. En el ejemplo de la bola, los píxeles cero son los traseros para poder borrarla cuando la mueves hacia la derecha.

Con el siguiente ejemplo ilustrativo podrás entenderlo perfectamente. Pierdes color porque solo hay 9 colores, pero los solapes entre sprites son muy buenos. Ojo, porque también pierdes algo de velocidad de impresión.



*Fig. 36 Solapes perfectos*

#### 8.4.2 Sobreescritura con 4 colores de fondo

Desde la versión V33 es posible elegir el número de bits que se usan para el fondo mediante una invocación al comando |PRINTSP especificando el Sprite 32, el cual no existe. Se pueden elegir 1 o 2 bit para el fondo, permitiendo 2 y 4 colores de fondo respectivamente.

|PRINTSP, 32, <num bits fondo>

Ejemplos:

PRINTSP, 32, 1 : ' con 1 bit de fondo tenemos 2 colores para el fondo
PRINTSP, 32, 2 : ' con 2 bit de fondo tenemos 4 colores para el fondo

Una vez que invocamos este comando, la librería 8BP se configura para tener en cuenta el numero de bits que se van a usar como bits de fondo. Si configuramos 2 bit de fondo, nuestra paleta de color tendrá que ser coherente con esta circunstancia. A continuación, se muestra un ejemplo

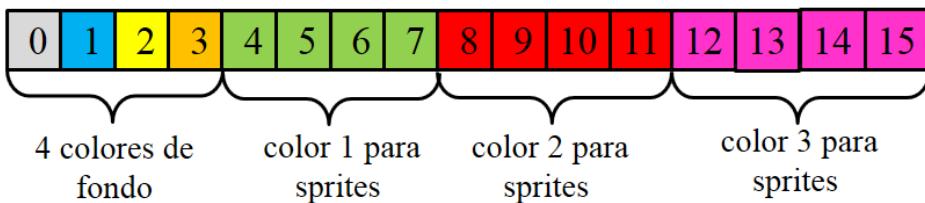


Fig. 37 Ejemplo de paleta con cuatro colores de fondo (2 bit de fondo)

En este ejemplo tienes 4 colores de fondo y tres colores para los sprites. Al igual que cuando se usa 1 bit para el fondo, al definir tus sprites debes tener en cuenta que la tinta 0 significa transparencia y la 1 significa no reestablecer el fondo, permitiendo formas no rectangulares de sprites. En el siguiente ejemplo los 3 colores escogidos para los sprites son el negro, el verde claro y el blanco.



Fig. 38 ejemplo de juego con paleta con hasta cuatro colores de fondo (2 bit)

Aunque con dos bits para el fondo puedes conseguir decorados más bonitos, la desventaja es que solo te quedan 3 colores para los sprites, mientras que si usas 1 solo bit para el fondo, tienes hasta 7 colores para tus sprites.

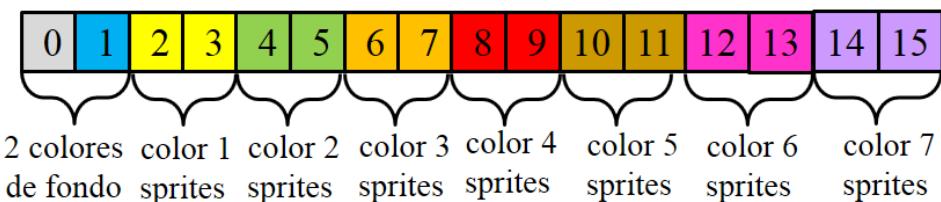


Fig. 39 Ejemplo de paleta con dos colores de fondo (1 bit de fondo)

### 8.4.3 Sobreescritura en MODE 1

Desde la versión V34 de 8BP, es posible usar sprites con sobreescritura en MODE 1. Aquí nos encontramos con una limitación muy fuerte porque, aunque tenemos dos colores para el fondo, tan solo disponemos de un color para los sprites.

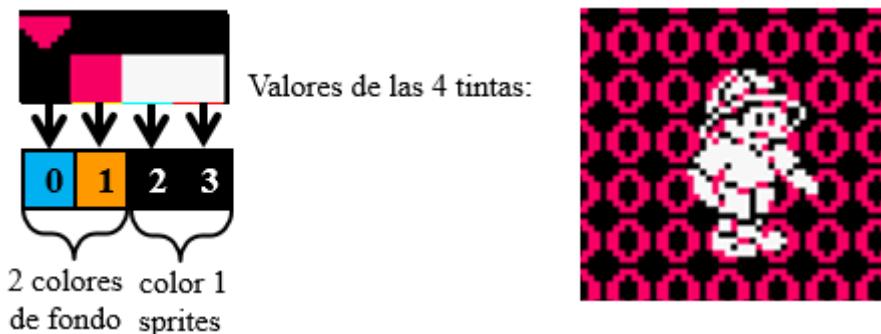
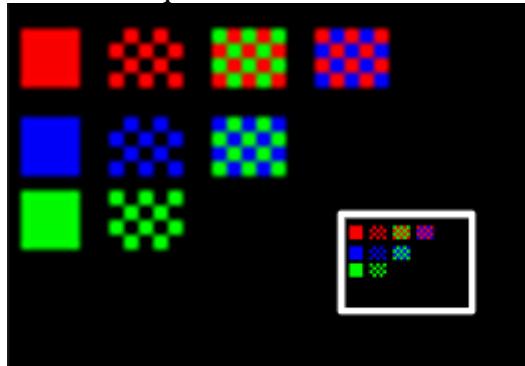


Fig. 40 Ejemplo de paleta de mode 1

Se puede caer fácilmente en el error de pensar que los sprites disponen del color que sea y además, del negro. Eso no es así. El negro es un color más y como tal debe consumir dos tintas con este mecanismo. Solo disponemos de dos tintas para el Sprite y las hemos usado en el blanco (en este ejemplo). Como puedes apreciar, el personaje donde no es blanco es transparente y no negro.

A pesar de esta estricta limitación, si te esmeras puedes hacer unos sprites muy vistosos en MODE 1, y si en cada pantalla cambias los colores de fondo, y usas mezclas (entrampados) de color en los marcadores del juego, puedes conseguir un resultado muy satisfactorio.

Usando mezclas en MODE 1 puedes simular 10 colores con tan solo 4. Aquí puedes ver un ejemplo. Los colores del entramado se fusionan y por ejemplo, verde + rojo se ve amarillo. Puede que en esta imagen no lo veas tan convincente, pero en la pantalla de tu Amstrad si que lo verás bien.



#### 8.4.4 Cómo pintar sprites “por detrás” del fondo

El mismo mecanismo para imprimir sprites por encima del fondo puede servir para pintar por detrás del fondo.

Como ya hemos visto, para imprimir por delante del fondo usamos bits que no se usan en el fondo, de modo que, aunque reducimos el número de colores, no dañamos el bit o los bits de fondo. Si usamos un bit para el fondo, tendremos que usar dos tintas para representar el mismo color de sprite: una con el bit de fondo a cero y otra con el bit de fondo a 1.

Ahora bien, si en lugar de asignar el mismo color a estas dos tintas, asignamos el mismo color que el fondo a la que tiene el bit de fondo a 1, entonces ante un solape del sprite con el fondo, se verá el color del fondo, dando la sensación de que el sprite pasa por detrás. Esto funciona tanto en MODE 0 como en MODE 1.

En el siguiente ejemplo se usa un bit para el fondo, el cual consiste en unas letras amarillas sobre un fondo negro. Los sprites son unas “monedas” que como se puede apreciar se han pintado aparentemente detrás del fondo.

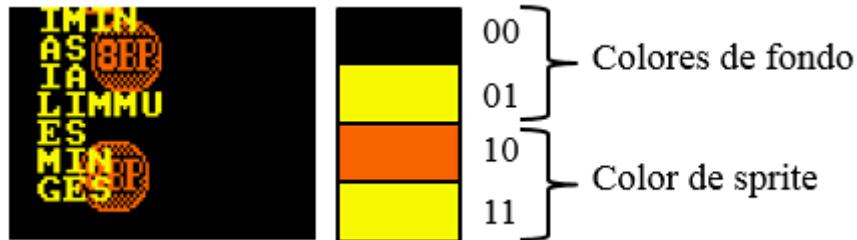


Fig. 41 Sprites impresos “detrás” de las letras

#### 8.4.5 Cómo usar más colores con sobreescritura

Si has hecho tus primeras pruebas con sobreescritura y necesitas más colores en tu videojuego, hay tres formas de lograrlo, pero necesitas entender bien el método de 8BP:

- 1) Usar algunos sprites con sobreescritura y otros sin ella
- 2) Usar sprites cuyo color depende del fondo (color condicional)
- 3) Usar un sprite como fondo

Vamos a ver uno por uno estos tres “trucos”:

##### Usar algunos sprites con sobreescritura y otros sin ella:

Este es el más sencillo y más utilizado de los tres trucos. Suponte que solo uno de tus sprites requiere sobreescritura y consume 3 colores. Eso significa que debes destinar 6 tintas a este Sprite. Sin embargo, si el resto de sprites no requieren sobreescritura, puedes imprimirlos sin sobreescritura y usar más colores en ellos, es decir:

- 2 tintas para el fondo (2 colores)
- 6 tintas para el Sprite con sobreescrituras (3 colores)
- 8 tintas para los demás sprites (8 colores)

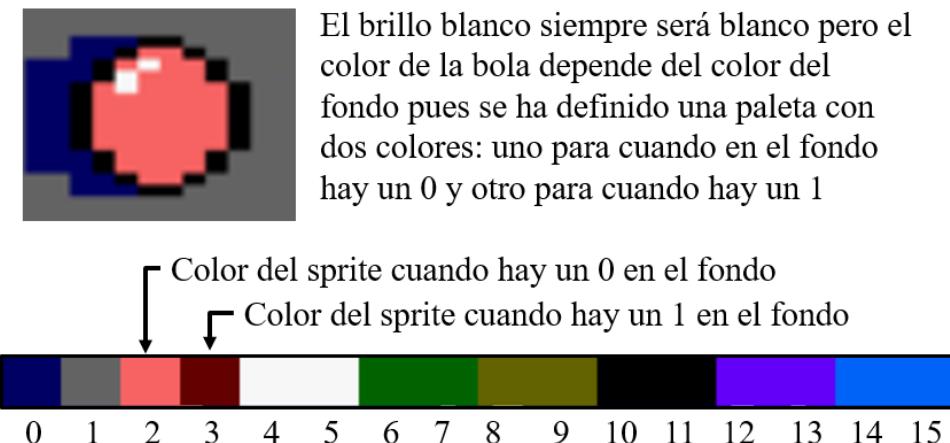
¡En este caso, en total podrás usar  $2+3+8 = 13$  colores!!!! Simplemente tienes que activar el flag de sobreescritura en el Sprite que lo necesita y dejarlo inactivo en los otros sprites. En los sprites que no usen sobreescritura podrás usar los 13 colores, en el Sprite con sobreescritura usarías 3 y en el fondo 2.

Otros ejemplos son posibles. Por ejemplo, si los sprites con sobreescritura necesitan 4 colores, entonces gastarán 8 tintas. Aparte tendremos 2 tintas para el fondo y las 6 tintas restantes pueden identificar 6 colores diferentes, es decir que podremos usar un total de 12 colores.

##### Usar sprites cuyo color depende del fondo (color condicional):

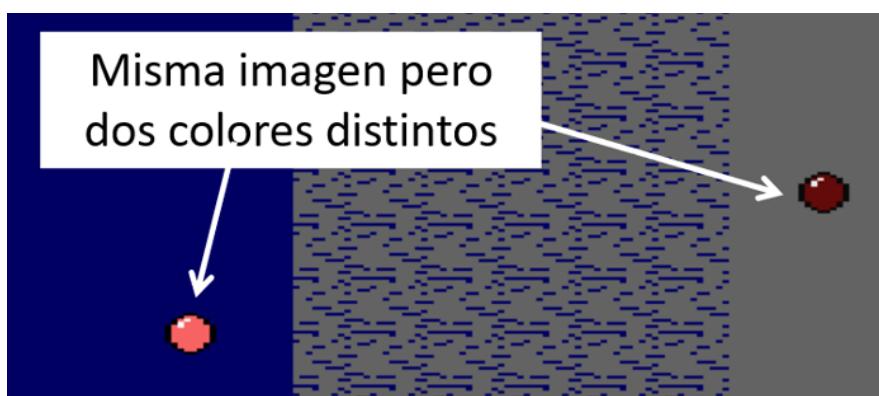
Consiste en usar una paleta donde en lugar de repetir cada color, usamos dos colores diferentes. En ese caso, cuando el fondo sea cero tendremos un sprite de un color y cuando el fondo sea 1 tendremos otro color. Esto puede ser útil para dibujar pájaros blancos que vuelan sobre un cielo azul (color 0) mientras osos rojos caminan por un suelo marrón (color 1). Es decir, que podemos usar mas color siempre que la textura del cielo o el suelo

no tenga demasiados cambios de color, ya que cada vez que el oso pase por encima de un pixel de fondo azul veremos un pixel blanco y si un pájaro pasa por encima de un pixel de fondo marrón veremos un pixel rojo. Vamos a ver un ejemplo:



*Fig. 42 Sprites creado con un color “condicional”*

Una vez creado el sprite con color condicional, en la imagen siguiente vemos el efecto que tiene al imprimirla en dos zonas de la pantalla, una con fondo 0 y otra con fondo 1.



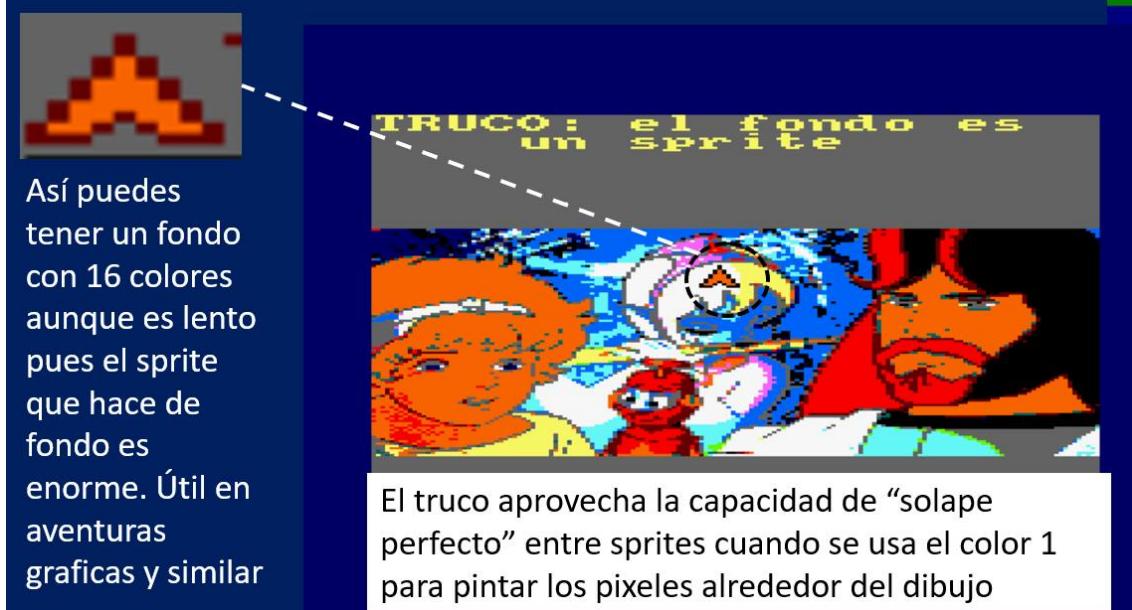
*Fig. 43 efecto del color “condicional”*

### Usar un sprite como fondo:

Este truco te permite usar hasta 16 colores de fondo con sobreescritura. El truco se basa en que, al activar la sobreescritura en un sprite, todos los píxeles definidos con “1” respetarán el valor del pixel que haya en pantalla, ya sea un pixel de un verdadero fondo o de un sprite impreso anteriormente. Imprimiremos en cada ciclo tanto el sprite que hace de fondo como los sprites que van a aparecer sobreimpresos. O al menos en cada ciclo que los movamos.

El sprite que usemos como fondo, lo imprimiremos **sin activar la sobreescritura**, mientras que los sprites que imprimamos encima la tendrán activa. El único inconveniente es que si el sprite de fondo es muy grande, se tardará en imprimir y el número de fps del juego no permitirá hacer un juego de arcade aunque sí puede servir para aventuras gráficas. Además, **recuerda que en 8BP la altura máxima de un sprite es 127 líneas**. La anchura máxima no es un problema pues también es 127 bytes y la pantalla tiene solo 80 bytes de ancho.

Crearemos el sprite que vamos a imprimir encima rodeado de unos, sin ceros ya que no va a re establecer el fondo al moverse. En cada ciclo se imprimirán tanto el sprite de fondo como nuestros pequeños sprites encima. En este ejemplo hemos dibujado una especie de puntero o flecha que controlas con el teclado, sobre un fondo que ocupa media pantalla, es decir 8KB (80bytes de ancho x 100 líneas)



*Fig. 44 Un sprite como fondo*

Como el sprite del puntero tiene sobreescritura, sufrirá los efectos del truco anterior (color condicional) a menos que definas algunos colores “dobles” para no sufrir ese efecto. Es decir, si por ejemplo repites un color para no sufrir ese efecto en un sprite, entonces tendrás una paleta de 15 colores diferentes, no de 16. Es decir, que de algún modo tienes que aplicar el principio del primer truco: un sprite sin sobreescritura que hace de fondo y uno o más sprites con sobreescritura que se imprimen encima. Cuanto más color tenga el fondo, menos colores tendrán tus sprites con sobreescritura. Por ejemplo (otras combinaciones son posibles) puedes usar:

- **Fondo:** 2 colores de fondo + 8 colores + 3 colores repetidos = 13 colores
- **Sprites sobreimpresos:** 6 tintas = 3 colores repetidos

Una forma de acelerar la velocidad del sprite de fondo “gigante” es fragmentarlo en franjas. Por ejemplo, en 8 sprites horizontales de altura 16. Los sprites alargados horizontalmente se imprimen más deprisa que los alargados verticalmente. En ese caso sólo necesitas imprimir la franja o par de franjas que intersecten con tu sprite puntero.

## 8.5 Sprites con imágenes de fondo

Las imágenes de fondo son una característica de 8BP V42. La idea es que en un scroll puedan pasar árboles o casas por debajo de tu avión y no por ello hagan parpadear al avión. Piensa que, aunque el avión tenga impresión transparente y se pinta sobre el fondo, si el fondo se mueve, no va a respetar al sprite y provocará un parpadeo. Con esta nueva capacidad, se pueden hacer juegos con scroll donde tu protagonista o nave pasa por encima de cosas y **no hay parpadeos**. Para entender esto mejor hay que comprender el mecanismo de scroll de 8BP que se cuenta más adelante.

En el conjunto de demostraciones de 8BP se ha incluido una que demuestra el efecto que tiene usar imágenes “de fondo” en tu scroll.



Casas de tipo “background images”

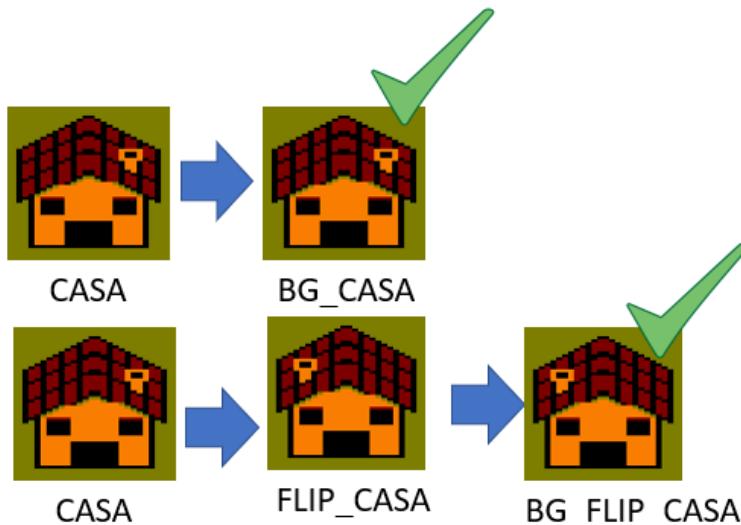
Sin esta capacidad, (antes de la versión V42 de 8BP) los sprites podían tener sobreescritura sin sufrir parpadeos siempre que el fondo no se moviese, pero el movimiento del fondo (el scroll) provocaba que hubiese parpadeos inevitables. Desde V42 es posible crear imágenes de fondo y asignárselas a los sprites.

Lo que hay que hacer es simplemente asignar al Sprite una imagen de fondo, que en el archivo de imágenes deben estar contenidas entre las etiquetas:

```
_BEGIN_BG_IMAGES  
BG_CASA DW CASA  
BG_CASA_FLIP DW CASA_FLIP  
_END_BG_IMAGES
```

La imagen **CASA** es una imagen creada normalmente con una particularidad: solo usa los colores de fondo. De ese modo cuando en un juego haya un Sprite que tenga asignada la imagen **BG\_CASA**, automáticamente tendrá asignada un tipo de transparencia especial, en la que solo se van a modificar los bits que representan los colores del fondo y se respetará cualquier Sprite que se encuentre por encima del dibujo, evitando parpadeos.

El proceso de creación de imágenes de fondo es muy sencillo, pero también muy estricto: A partir de una imagen (que use solo las tintas de fondo) se puede construir una imagen de fondo. Si la quieres usar flipeada primero debes flippear la imagen y luego construir la imagen de fondo con la imagen flipeada



En caso de que quieras flippear una imagen de fondo, primero debes flippearla con la sección **FLIP\_IMAGES** y después ya la puedes usar para construir una imagen de fondo. Ese debe ser el orden y no al revés. Es decir, no puedes crear una imagen, crear con ella una imagen de fondo y después tratar de flippear una imagen de fondo.

Las imágenes de fondo siempre que se asignen a un Sprite se imprimen con esta transparencia especial y da igual si el Sprite tiene asignado o no el flag de transparencia en su byte de estado (ahora veremos qué es esto).

**IMPORTANTE:** las imágenes de fondo son costosas de imprimir, y si las flipeas aún son mas costosas. Si tu juego tiene scroll, trata de usarlas para elementos sobre los que vayan a sobrevolar tu personaje o los enemigos. Por ejemplo, para acelerar el scroll usa imágenes normales en casas o rocas que aparezcan en los laterales de tu scroll, que no van a ser frecuentemente solapados por los sprites.

## 8.6 Tabla de atributos de sprites

Los sprites se almacenan en una tabla que contiene un total de 32 sprites.

Cada entrada de la tabla contiene todos los atributos del sprite y ocupa 16 bytes por razones de rendimiento, ya que 16 es múltiplo de 2 y ello permite acceder a cualquier sprite con una multiplicación muy poco costosa. La tabla se encuentra ubicada en la dirección de memoria 27000, de modo que se puede acceder desde BASIC con PEEK y POKE, aunque también disponemos de comandos RSX para manipular los datos de esta tabla, tales como |SETUPSP o |LOCATESP

Los sprites tienen un conjunto de parámetros, de los que el primero de ellos es el byte de flags de status. En este byte, cada bit representa un flag y cada flag significa una cosa, concretamente representan si el sprite se toma en consideración al ejecutar ciertas funciones. En la siguiente tabla se resume lo que ocurre si están activos (a “1”)

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Fig. 45 flags en el byte de estado

Para entender la potencia de estos flags vamos a ver algunos ejemplos:

- **Bit 0:** flag de impresión: nuestro personaje o las naves enemigas lo tendrán activado y en cada ciclo del juego invocaremos a |PRINTSPALL y se imprimirán todos a la vez
- **Bit 1:** flag de colisión: una fruta o moneda por ejemplo pueden no tener flag de impresión, pero tener el de colisión. Este flag significa que un sprite “colisionador” se puede colisionar con el sprite que tenga este flag activo.
- **Bit 2:** flag de animación automática: se tiene en cuenta en |ANIMALL. En el caso del personaje, recomiendo desactivarlo, ya que si me quedo quieto no hay que cambiar el fotograma.
- **Bit 3:** flag de movimiento automático. Se mueve solo al invocar |AUTOALL teniendo en cuenta su velocidad. útil en meteoritos y guardias que van y vienen.
- **Bit 4:** flag de movimiento relativo. Todos los sprites que tengan este flag se mueven a la vez al invocar “[MOVEALL, dy, dx]” muy útil en naves en formación y llegadas a planetas. También sirve para simular un scroll si dejas tu personaje en el centro y al pulsar los controles se desplazan casas o elementos de alrededor. Parecerá que es tu personaje el que avanza por un territorio.
- **Bit 5:** flag de colisionador. Todos los sprites con este flag activo son considerados por la función |COLSPALL, a la hora de detectar su posible colisión con el resto de sprites.
- **Bit 6:** flag de sobreescritura: si este flag está activo, el sprite se podrá desplazar por encima de un fondo, restableciéndolo al pasar. Esta es una opción avanzada e implica el uso de una paleta de color especialmente preparada, más limitada en número de colores. La sobreescritura tiene este “precio”.
- **Bit 7:** flag de ruta: si este flag está activo, el comando |ROUTEALL te permitirá mover un sprite cíclicamente a través de una trayectoria que tu definas, definida por una serie de segmentos. El comando |ROUTEALL sabe en qué segmento y posición se encuentra cada sprite y si llega a un cambio de segmento, modifica la velocidad del sprite de acuerdo a las condiciones del siguiente segmento. |ROUTEALL no mueve al sprite, solo modifica su velocidad. Para moverlo hay que usarlo conjuntamente con |AUTOALL.

Ejemplos de asignación del valor del byte de status:

Típico enemigo: un sprite que se debe imprimir en cada ciclo, con detección de colisión con otros sprites y animación debe tener:

$$\text{status} = 1(\text{bit 0}) + 2 (\text{bit1}) + 4 (\text{bit 2}) = 7 = \&x0111$$

Una casa que se desplaza al movernos: un sprite que se imprime en cada ciclo, pero sin detección de colisión con otros y movimiento relativo

$$\text{status} = 1(\text{bit 0}) + 0 (\text{bit1}) + 0 (\text{bit 2}) + 0 (\text{bit 3}) + 16 (\text{bit 4}) = 17 = \&x10001$$

Una fruta que nos da bonus: es un sprite que no se imprime en cada ciclo, pero tiene detección de colisión

$$\text{status} = 0(\text{bit } 0) + 2 (\text{bit}1) = 2 = \&x10$$

Una nave que va a seguir una trayectoria predefinida. Va a requerir del flag de ruta, el de movimiento automático, el de animación, el de colisión y el de impresión. Esta vez te lo voy a poner en binario directamente. Como ves el bit 7 lo he puesto a 1, después hay 3 ceros porque no he puesto el de sobreescritura ni el de colisionador ni el de movimiento relativo y por último he puesto 4 flags activos correspondientes respectivamente a los de movimiento automático, animación, colisión e impresión

$$\text{status}=10001111$$

La tabla de atributos de sprites se compone de 32 entradas de 16 bytes cada una, comenzando en la dirección 27000.

El motivo de tener 16 bytes no es otro que el del rendimiento, ya que calcular la dirección del sprite N implica multiplicar por 16, lo cual, al ser un múltiplo de 2, se puede hacer con un desplazamiento. Esto es útil en operaciones que involucran un único sprite. Para operaciones que recorren la tabla de sprites (como |PRINTSPALL o |COLSP), internamente se recorre la tabla con un índice al que se le suma 16 para pasar de un sprite al siguiente. La suma es lo más rápido en ese caso.

Los atributos que tiene cada sprite son:

<b>atributo</b>	<b>Byte</b>	<b>Longitud (bytes)</b>	<b>Significado</b>
status	0	1	Byte que contiene los flags de status para las operaciones PRINTSPALL, COLSP, ANIMALL, AUTOALL , MOVERALL, COLSPALL y ROUTEALL
Y	1	2	Coordenada Y [-32768..32768] los valores correspondientes al interior de la pantalla son [0..199]
X	3	2	Coordenada X en bytes [-32768..32768] los valores correspondientes al interior de la pantalla son [0..79]
Vy	5	1	Paso a dar en el movimiento automático
Vx	6	1	Paso a dar en el movimiento automático
Secuencia	7	1	Identificador de la secuencia de animación [0..31]. Si no posee secuencia se debe asignar un cero
Fotograma	8	1	Numero de frame en la secuencia [0..7]
Imagen	9	2	Dirección de memoria donde está la imagen
Sprite anterior	10	2	Uso interno para el mecanismo de ordenación de sprites
Sprite siguiente	12	2	Uso interno para el mecanismo de ordenación de sprites
Ruta	15	1	Identificador de ruta que debe seguir el sprite

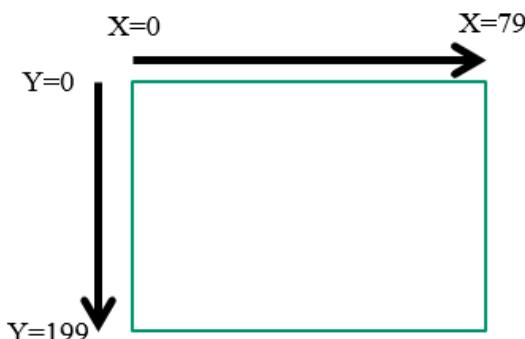
La dirección de memoria donde se almacenan las coordenadas de cada sprite se pueden calcular así:

**Dirección coordenada Y =** $27000 + 16*N + 1$

**Dirección coordenada X =** $27000 + 16*N + 3$

Accediendo con **POKE** a esas direcciones podemos modificar su valor, aunque también dispones de **|LOCATESP**.

La librería 8BP no usa “píxeles” en la coordenada X, sino bytes, de modo que la coordenada X que cae dentro de la pantalla se encuentra en el rango [0..79]. La coordenada Y se representa en líneas de modo que el rango representable en pantalla es [0..200]. Si ubicas un sprite fuera de esos rangos, pero parte del sprite se encuentra en la pantalla, la librería hará el “clipping” y pintará el trozo que se tenga que ver.



*Fig. 46 coordenadas de pantalla*

Las direcciones de los atributos de los 32 sprites se pueden manejar con PEEK y POKE, aunque la asignación de secuencia de animación y de ruta implican más operaciones y si quieras cambiarlas no basta con un POKE, sino que hay que usar **|SETUPSP**. Aquí tienes la lista de direcciones de todos los atributos de los 32 sprites:

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tabla 3 Direcciones de atributos de los 32 sprites

El espacio ocupado por cada sprite en la tabla es de 16 bytes. Como ves las coordenadas X e Y son números de 2bytes. Los sprites aceptan coordenadas negativas por lo que puedes imprimir parcialmente un sprite en la pantalla, dando la sensación de que va entrando poco a poco. No podrás establecer coordenadas negativas con POKE, pero si podrás hacerlo con |LOCATESP y también con |POKE, que es una versión del comando POKE de BASIC pero que acepta números negativos (y números de 16 bit).

Es una buena práctica ubicar al personaje o nave espacial en la posición 31 (hay 32 sprites numerados del 0 al 31). Si tu nave tiene la posición 31 se imprimirá la última, encima del resto de sprites en caso de solape.

## 8.7 Impresión de todos los sprites y ordenados

En la librería 8BP dispone de un comando que imprime a la vez todos los sprites que tengan el flag de impresión activo. Se trata del comando |PRINTSPALL

Este comando tiene 4 parámetros, aunque solo necesitas rellenarlos la primera vez que lo invoques, pues las siguientes veces, se acordará de los parámetros, y solo deberás rellenarlos de nuevo si deseas cambiar alguno de ellos. Esto es útil porque el paso de parámetros consume mucho tiempo (te puedes ahorrar más de 1ms evitando el paso de parámetros).

Los parámetros son:

|PRINTSPALL, <ordenini>, <ordenfin>, <animar>, <sincronismo>

- **El parámetro de sincronismo** puede tomar los valores 0 o 1 e indica que se esperará a una interrupción de barrido de pantalla para imprimir. Si deseas velocidad no te lo recomiendo. Si deseas más suavidad, quizás sí.
- **El parámetro de animación** puede tomar los valores 0 o 1. En caso de estar activo, antes de imprimir cada sprite se comprobará su flag de animación en el byte de status y si lo tiene activo, entonces se cambiará de fotograma. Es muy útil con los enemigos, pero con tu personaje puede que no, pues quizás sólo desearás animarle al moverle y no en cada fotograma. IMPORTANTE: la animación se hace antes de imprimir, no después de imprimir. Eso significa que si acabas de asignar una secuencia de animación, no verás el primer fotograma de dicha secuencia.
- **Los parámetros de orden (“ordenini”, “ordenfin”)** indican los sprites inicial y final que definen el grupo de sprites ordenados por coordenada “Y” que vamos a imprimir. Por ejemplo, si asignamos los valores 0,0 entonces se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si asignamos 0,8 se imprimirán del 0 al 8 ordenados (9 sprites) y del 10 al 31 de modo secuencial. Si ponemos un 0,31 se imprimirán todos los sprites ordenados. Si ponemos un 10,20 se imprimirán secuencialmente los sprites del 0 al 9, luego se imprimirán ordenados del 10 al 20 y finalmente se imprimirán secuencialmente del 21 al 31

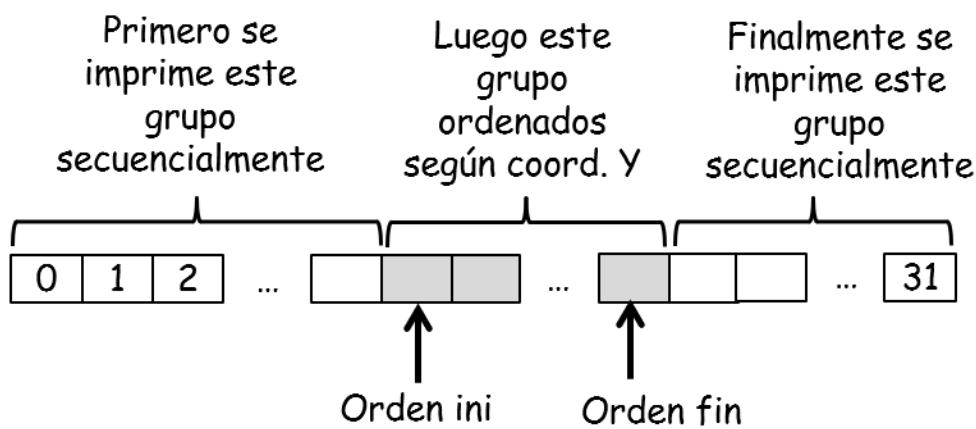


Fig. 47 Grupos de sprites secuenciales y ordenados

El ordenamiento es muy útil para hacer juegos tipo “Renegade” o “Golden AXE”, donde es necesario dar un efecto de profundidad. El ordenamiento se aprecia cuando hay solapamientos entre sprites.



*Fig. 48 Efecto del ordenamiento de sprites*

```
|PRINTSPALL, 0,0,1,0 : imprime de forma secuencial todos los sprites
|PRINTSPALL, 0,31,1,0 : imprime de forma ordenada todos los sprites
|PRINTSPALL, 0,7,1,0: imprime 8 ordenados y el resto secuencial
|PRINTSPALL, 16,24,1,0: 16 secuenciales, 9 ordenados y 7 secuenciales
```

Si el parámetro “ordenini” se omite, se considera el ultimo valor asignado, o bien cero si nunca se ha asignado un valor. Además, si vas a modificar alguno de los dos parámetros de ordenamiento, conviene primero ejecutar PRINTSPALL,0,0,0,0 para que primero se reordenen los sprites secuencialmente antes de ordenarlos con una nueva configuración.

Imprimir de forma ordenada es más costoso computacionalmente que imprimir de forma secuencial. Si solo tienes 5 sprites que deben ser ordenados, pasa por ejemplo un 0,4 como parámetros de ordenamiento, no pases un 0,31. Ordenar todos los sprites lleva unos 2.5 ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizás tengas muchos sprites y no merezca la pena ordenar algunos, como los disparos o sprites que sabes que no se van a solapar.

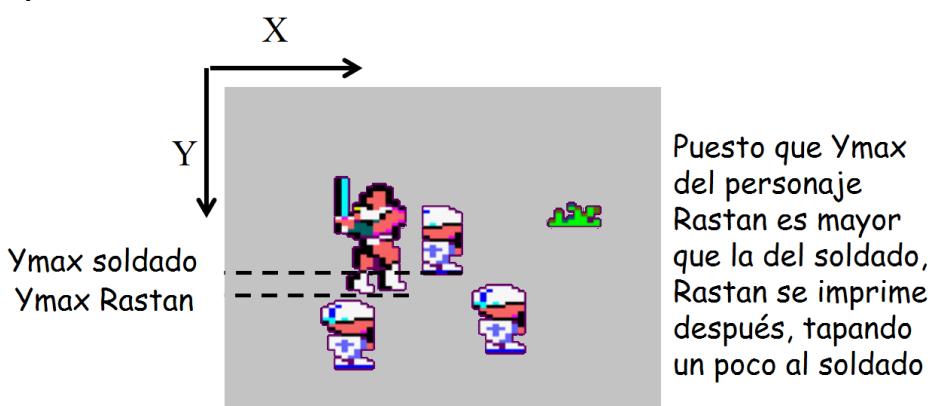
El algoritmo que se usa para ordenar los sprites es una variante del algoritmo conocido como “burbuja”. Aunque encontrarás en la literatura que el algoritmo llamado “burbuja” es muy poco eficiente, eso lo dicen los que hablan pensando en una lista de números aleatorios a ordenar. Nos encontramos ante un caso donde normalmente los sprites están casi ordenados y de un fotograma a otro solo se desordena uno o dos sprites, no más, pues sus coordenadas evolucionan “suavemente”. Por ello, el algoritmo recorre la lista de sprites y cuando encuentra un par de sprites desordenados, les da la vuelta y deja de seguir ordenando. Es tremadamente rápido, y aunque solo sea capaz de ordenar un par de sprites cada vez, es ideal para este caso de uso. Solo en el caso de que haya 2 sprites desordenados y que además se estén solapando, habrá un fotograma en el que veamos uno de ellos imprimiéndose en desorden, pero quedará corregido en el siguiente fotograma. Es imperceptible.

Puede que alguna vez deseas que la ordenación sea completa en cada fotograma. Es decir, que no se ordene un par de sprites en cada invocación a |PRINTSPALL, sino tener la seguridad de que todos están ordenados. La librería 8BP te lo permite mediante sus cuatro modos de ordenamiento, que puedes establecer mediante la invocación del comando |PRINTSPALL con un solo parámetro (basta con ejecutarlo una vez para fijar el modo de ordenación):

<b>PRINTSPALL,0</b> : ordenamiento parcial usando Ymin
<b>PRINTSPALL,1</b> : ordenamiento completo usando Ymin
<b>PRINTSPALL,2</b> : ordenamiento parcial usando Ymax
<b>PRINTSPALL,3</b> : ordenamiento completo usando Ymax

Los ordenamientos que usan Ymax se basan en la coordenada Y mayor de los sprites, es decir, donde se encuentran sus pies en lugar de su cabeza. Si los sprites son del mismo tamaño, un ordenamiento basado en Ymin te puede servir, pero si los sprites tienen diferente altura puede que desees ordenar según donde se encuentren los pies de cada personaje y para ello tendrás que usar el modo de ordenar 2 o el 3.

Los modos de ordenar con Ymax son más lentos, aproximadamente 0.128 ms por sprite, de modo que úsalos cuando los necesites realmente.



*Fig. 49 Ordenamiento según Ymax*

La ordenación completa consume muy poco mas que la parcial (aproximadamente 0.3ms). Esto es debido a que los sprites apenas se desordenan de un fotograma al siguiente, pero incluso esos 0.3ms merece la pena ahorrarlos si es posible.

Recuerda que el comando PRINTSPALL tiene “memoria”, de modo que basta con invocar la primera vez con parámetros y a partir de ese momento podemos invocar a PRINTSPALL sin parámetros pues el comando “conserva” los valores de los parámetros con los que fue invocado y no hace falta pasárselos a menos que cambien. Esto permite ahorrar más de 1ms, ya que el analizador sintáctico trabaja menos.

## 8.8 Colisiones entre sprites

Para comprobar si tu personaje o tu disparo han colisionado con otros sprites dispones del comando

**|COLSP, <sprite\_number>, @colision%**

Donde sprite number es el sprite que quieras comprobar (tu personaje o tu disparo) y la variable “colision” es una variable entera que previamente ha tenido que ser definida, asignando un valor inicial, por ejemplo:

**colision%=0**

|COLSP, 1, @colision%

La variable “colision” se llenará con el primer identificador de sprite que se detecte que ha colisionado con tu sprite, aunque podría ocurrir una colisión múltiple, pero el comando solo te entrega un resultado.

Internamente la librería 8BP recorre los sprites colisionables desde el 31 hasta el 0 (los recorre en orden inverso), y si tienen el flag de colisión activo (bit 1 del byte de status) entonces se comprueba si colisiona con tu sprite. Si no hay colisión con ninguno, la variable colision% queda con valor 32. En caso de haberla retornará el número de sprite que esté colisionando con tu sprite. Si por ejemplo colisionan el 4 y el 12, la función retornará un 12 pues comprueba antes el 12 que el 4.

**¡Ni tu personaje ni tu disparo deben tener el flag de “colisionable” (bit 1) y “colisionador” (bit 5) activos a la vez, ya que de lo contrario siempre colisionarán...consigo mismos!. Es decir, un sprite no puede tener los bits 1 y 5 activos a la vez.**

La colisión entre sprites es una tarea costosa. Internamente la librería necesita calcular la intersección entre los rectángulos que contiene cada sprite para determinar si hay solape entre ellos. Para ahorrar cálculos, lo mejor es ubicar a los enemigos en posiciones consecutivas de sprites. Si por ejemplo los enemigos con los que podemos chocar son los sprites del 15 hasta el 25, podemos configurar la colisión para que sólo compruebe esos sprites. Para ello invocaremos la colisión sobre el sprite 32 que no existe. Eso le indicará a la librería 8BP que se trata de información de configuración para el comando, indicando el rango de sprites colisionables que se va a explorar por cada colisionador:

|COLSP, 32, <sprite inicial>, <sprite final>

Ejemplo:

|COLSP, 32, 15, 25

Esta optimización si bien no es muy significativa, lo empieza a ser cuando se invoca varias veces a COLSP o se usa el comando |COLSPALL que internamente invoca varias veces a COLSP.

Otra interesante optimización, capaz de ahorrar 1 milisegundos en cada invocación, es decirle al comando que siempre use la misma variable BASIC para dejar el resultado de la colisión. Para ello se lo indicaremos usando como sprite el 33, que tampoco existe

col%=0

|COLSP, 33, @col%

Una vez ejecutadas estas dos líneas, las siguientes invocaciones a COLSP, dejarán el resultado en la variable col, sin necesidad de indicarlo, por ejemplo:

|COLSP, 23 : REM esta invocación es equivalente a |COLSP, 23, @col%

**IMPORTANTE:** la variable de colisión del comando |COLSP no es la que se usa en el comando |COLSPALL. Son variables diferentes (a menos que le pases a ambos comandos la misma variable para que actúen sobre ella).

## 8.9 Ajuste de la sensibilidad de la colisión de sprites

Es posible ajustar la sensibilidad del comando COLSP, decidiendo si el solape entre sprites debe ser de varios pixeles o de uno solo, para considerar que ha habido colisión.

Para ello se puede configurar el número de pixeles (pixeles en dirección Y, bytes en dirección X) de solape necesario tanto en la dirección Y como en la dirección X, usando el comando COLSP y especificando el sprite 34 (que no existe)

|COLSP, 34, <dy>, <dx>

La librería 8BP no usa “pixeles” en la coordenada X, sino bytes, de modo que debes tener en cuenta que una colisión de 1 byte, en realidad son 2 pixeles y esa es la mínima colisión posible cuando ajustas dx=0.

En la coordenada Y, la librería trabaja con líneas de modo que dy=0 significa una colisión de un solo pixel.

Una colisión estricta, útil para disparos sería aquella que no tolera ningún margen, considerando colisión en cuanto hay un mínimo solape entre sprites (1 pixel en dirección Y o un byte en dirección X)

|COLSP, 34, 0, 0: rem colision en cuanto hay un mínimo solape

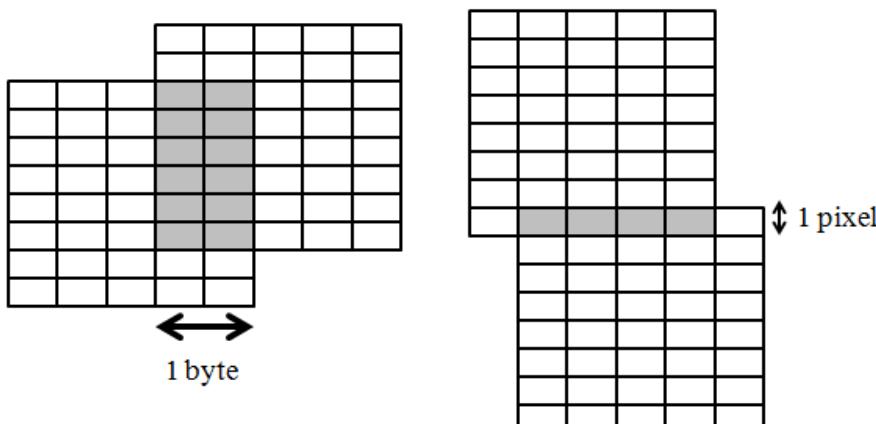


Fig. 50 Colisión estricta con COLSP, 34, 0, 0

Sin embargo, si estamos haciendo un juego en MODE 0, donde los pixeles son más anchos que altos, es quizás más adecuado dar algo de margen en eje Y, pero nada en eje X. Por ejemplo:

|COLSP, 34, 2, 0 : rem colision con 3 pix en Y y 1 byte en X

Mi recomendación es que, si hay disparos estrechos o pequeños, ajustes la colisión con (dy=1, dx=0) mientras que si solo hay personajes grandes puedes dejarla con más margen (dy=2, dx=1). También debes considerar que, si tus sprites tienen un “margen” de borrado alrededor para desplazarse borrándose a sí mismos, dicho margen no debería formar parte de la consideración de colisión por lo que tiene sentido que tanto dy como dx no sean cero. En cualquier caso, es algo que decidirás en función del tipo de juego que hagas.

## 8.10 Quién colisiona y con quién: COLSPALL

Con la función **|COLSP** que hemos visto hasta ahora, es posible la detección de colisión de un sprite con todos los demás. Sin embargo, si tenemos un disparo múltiple, donde por ejemplo nuestra nave puede disparar hasta 3 disparos simultáneamente, tendríamos que detectar la colisión de cada uno de ellos y adicionalmente la de nuestra nave, resultando en 4 invocaciones a **|COLSP**.

Debemos tener presente que cada invocación atraviesa la capa de análisis sintáctico, por lo que cuatro invocaciones resultan costosas. Para ello disponemos de un comando muy potente: **|COLSPALL**.

Esta función funciona en dos pasos: primero debemos especificar que variables van a almacenar el sprite colisionador y el colisionado. La siguiente instrucción la ejecutaremos una sola vez, y sirve para definir las variables sobre las obtendremos los resultados, las cuales deben existir previamente:

**|COLSPALL, @colisionador%, @colisionado%**

Y posteriormente, en cada ciclo de juego simplemente invocamos la función **|COLSPALL** sin parámetros.

La función va a considerar como sprites “colisionadores” aquellos que tengan el flag de colisionador a “1” en el byte de estado (es el bit 5), y como “colisionados” aquellos sprites que tengan a “1” el flag de colisión (bit 1) del byte de estado. Los sprites colisionadores deberán ser nuestra nave y nuestros disparos y los colisionados todos aquellos con los que nos podamos chocar: naves y disparos enemigos, montañas, etc. Como anteriormente dije, un sprite no debe tener ambos bits activos (=1) a la vez.

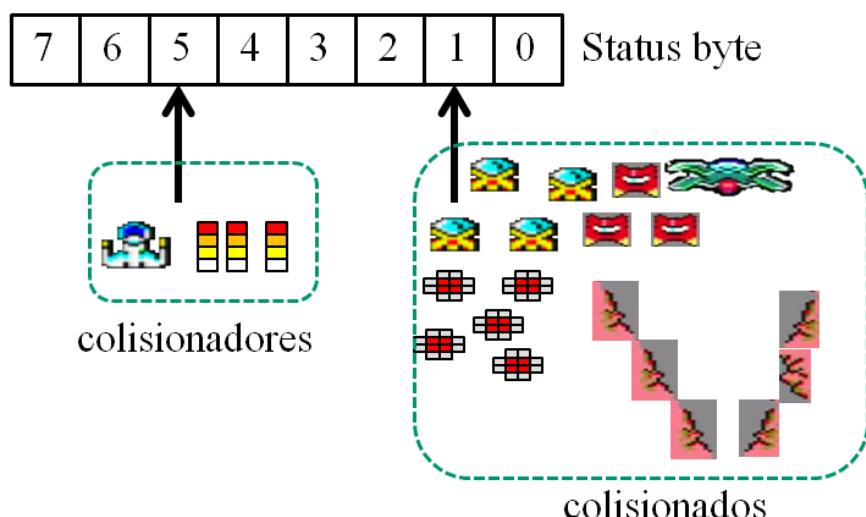


Fig. 51 Colisionadores versus colisionados

La función **|COLSPALL** empieza comprobando el sprite 31 (si es colisionador) y va descendiendo hasta el sprite 0, invocando internamente a **|COLSP** para cada sprite colisionador. Para cada colisionador, los sprites colisionables también se recorren en orden decreciente (desde 31 hasta 0). En cuanto detecta una colisión, interrumpe su ejecución y retorna el valor del colisionador y el colisionado. Por ello es importante que nuestra nave tenga un sprite superior a nuestros disparos. De ese modo, si nos alcanzan

lo detectaremos, aunque hayamos alcanzado a un enemigo con un disparo en el mismo instante.

En cada ciclo de juego sólo se podrá detectar una colisión, pero es suficiente. No es una limitación importante que en cada fotograma solo pueda empezar a “explotar” un enemigo. Si, por ejemplo, tiras una granada y hay un grupo de 5 soldados afectados, cada soldado comenzará a morir en un fotograma distinto, y al cabo de 5 fotogramas estarán todos explotando. Usando |COLSPALL no explotarán todos a la vez, pero tu juego será más rápido y en un arcade es algo muy importante.

En caso de invocar a |COLSPALL con un único parámetro,

**|COLSPALL, <colisionador inicial>**

Se explorarán los colisionadores desde el colisionador indicado -1 hasta el sprite cero, en orden descendente. De este modo si necesitas detectar más de una colisión por ciclo de juego, podrás hacerlo invocando sucesivamente a **COLSPALL, <colisionador>** hasta que la variable colisionador tome el valor 32

**Ejemplo:**

**|COLSPALL, 7 : rem busca colisiones a partir del colisionador 6**

### 8.10.1 Cómo programar un disparo múltiple usando COLSPALL

Lo primero que debes hacer es decidir el número de disparos activos que van a poder existir. Si decides que tu nave puede disparar 3 proyectiles a la vez, entonces debes reservar 3 identificadores de sprite para disparar. A continuación, debes ajustar el retardo entre un disparo y el siguiente para evitar que dos proyectiles salgan casi pegados si se dispara muy rápido. Esto lo puedes hacer definiendo una demora mínima entre disparo y disparo.

En el siguiente ejemplo he establecido una demora entre disparos de 10 ciclos de juego, para ello, al pulsar la barra espaciadora (tecla 47) se comprueba si han transcurrido al menos 10 ciclos desde el ultimo disparo. De no ser así, no disparará.

```
130 'ciclo de juego -----
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' rutina movimiento personaje -----
172 IF INKEY(47)=0 THEN IF demora<ciclo-10 THEN demora=ciclo:disp= 1+
disp MOD 3:|LOCATESP,10+disp,PEEK(27001)+8,PEEK(27003):
|SETUPSP,10+disp,0,137: |SETUPSP,10+disp,15,3+dir
180 IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'ir izquierda
193 ciclo=ciclo+1
310 GOTO 150
```

Para elegir el sprite que se va a usar como disparo se ejecuta la instrucción:

**disp = 1+ disp Mod 3**

Esto va a permitir que tu disparo tome los valores 1,2,3,4 alternativamente. Si necesitas que los sprites sean el 20,21,22,23 puedes usar **disp = 20 + disp Mod 3**. Ojo

porque si pones como sumando un 21 no funciona (pruébalo tú mismo). Es lo que tiene la aritmética modular.

Y Ahora vamos con las colisiones y la ventaja de utilizar COLSPALL, mucho más rápido que invocar múltiples veces a COLSP. Las únicas recomendaciones importantes son:

- Que nuestro <sprite number> sea superior a nuestros disparos, para que |COLSPALL| lo compruebe antes que a los disparos.
- Que tengamos configurado |COLSP| para solo comprobar la lista de sprites que son enemigos y son necesarios de colisionar, mediante el uso de |COLSP 32, <inicio>, <fin>

Antes de comenzar el ciclo de juego definimos nuestras variables:

**collider=32:collided=32:|COLSPALL,@collider,@collided**

En el ciclo de juego pondremos:

```
|COLSPALL:IF collider<32 THEN if collider=31 THEN GOSUB 300:goto 2000:  
ELSE GOSUB 770
```

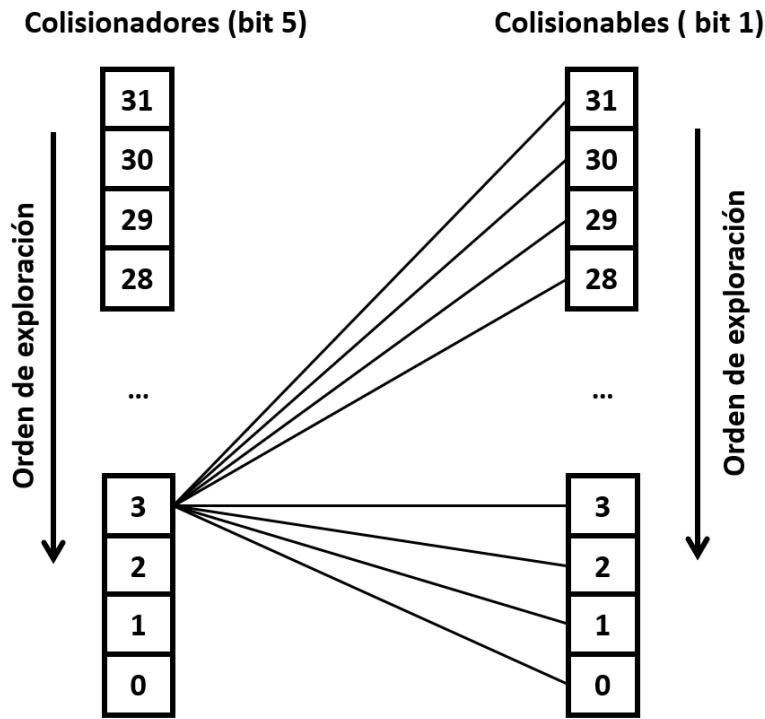
Con esta línea ya sabemos si hay colisión, pues entonces la variable “collider” será <32>. Además, si es igual a 31 entonces es nuestra nave (nos han dado) y si no, entonces seguro que uno de nuestros disparos ha alcanzado a una nave enemiga e iremos a la rutina ubicada en la línea 770, donde se encontrará algo como esto:

```
769' --- rutina colision disparo -----  
770 |SETUPSP, collider, 9, imgborrado:'asociamos imagen borrado al  
disparo  
772 |PRINTSP, collider: 'borramos el disparo  
775 |SETUPSP, collider, 0, 0: 'desactivamos el disparo  
777 if collided>=duros then return:'enemigo indestructible  
778 ' la secuencia 4 es una secuencia de animación de "Muerte", una  
explosion  
780 |SETUPSP, collided, 7, 4:|SETUPSP, collided, 0, &x101: return
```

En resumen, con una sola invocación a COLSPALL ya sabemos quién ha colisionado (“collider”), y con quién ha colisionado (“collided”).

### 8.10.2 Quien colisiona cuando hay varios solapes

Es muy importante tener en cuenta que 8BP recorre los colisionadores desde el 31 hasta el 0 y para cada uno de ellos, recorre los colisionables desde el 31 hasta el 0. Debemos asociar los Sprite ID a nuestros sprites en función de cómo queremos que colisionen.



*Fig. 52 orden de comprobación de colisiones*

Si nuestro sprite (colisionador) colisiona con dos sprites en la misma área, podemos saber a priori con cuál de ellos vamos a colisionar, lo cual es muy útil para ciertos juegos.

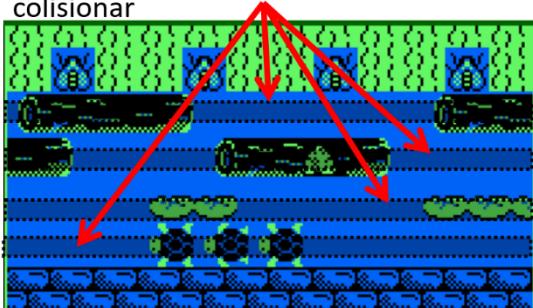
Supongamos el juego “frogger”, en el que una rana debe cruzar un río saltando sobre los troncos. Si la colisión es sobre un tronco, no moriremos, pero si la colisión es sobre un río, moriremos.

Para programarlo podemos poner 4 ríos (4 sprites alargados inmóviles) y sobre ellos mover unos sprites que son troncos. Podríamos plantear el siguiente reparto:

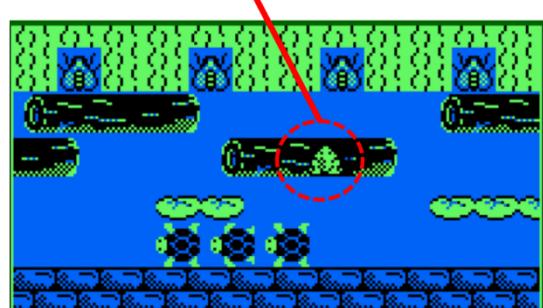
- La rana es el sprite 31 (supongamos que es colisionador)
- Los troncos son los sprites 4, 5, 6, 7 (colisionables)
- Los ríos son los sprites 0, 1, 2, 3 (colisionables)

Los ríos pueden tener el flag de impresión desactivado, de modo que pueden colisionar con la rana (flag de collided) sin necesidad de que se impriman. Es decir, les pondríamos status =2

Ríos (rectángulos largos) que no se imprimen pero que están ahí y pueden colisionar



Se detecta la colisión con el tronco antes que con el río por que el río (collided) tiene un Sprite ID menor que el tronco



*Fig. 53 en caso de solape nos interesa que colisione el tronco*

Pues bien, como los troncos y el rio se solapan, en el momento en que la rana se sube a un tronco colisiona con ambos, pero se comprueba primero el tronco pues tiene un sprite ID mayor. El comando de colisión detectará únicamente la colisión con el tronco. Por el contrario, si la rana salta sobre el agua, entonces el comando de colisión detectará al rio y tras evaluar desde **BASIC** la variable “collided” y ver que es un rio, determinaríamos que nuestra rana debe morir

### 8.10.3 Uso avanzado del byte de status en colisiones

En ocasiones puedes querer que un enemigo no te mate al colisionar con tu personaje debido a que se encuentra en un estado especial, o porque simplemente se encuentra lejos en un juego que pretende simular que los enemigos se están acercando.

Algunas circunstancias especiales pueden requerir el uso de una “marca” en el sprite para indicar que, aunque haya habido colisión, no debe morir, o no debe matarte.

Para ello te puedes valer de los flags que no uses en el sprite y chequearlos en tu rutina de colisión.

Vamos a ver un ejemplo para un enemigo que queremos hacer inofensivo por encontrarse lejos (simulando 3D) o con poca energía pero que colisiona con nosotros.

Supongamos que tu personaje es collided y el enemigo es collider. Puedes forzar que solo se detecte colisión con sprites mayores que 25 y si el enemigo es el numero 20 (por ejemplo) no podrá colisionar consigo mismo.

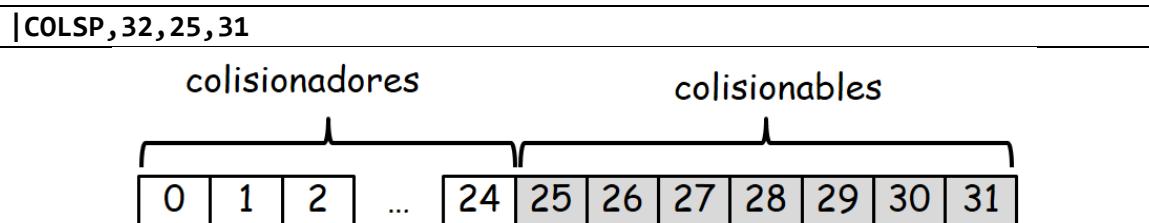


Fig. 54 efecto de COLSP,32,25,31

Como indicador de “inofensivo” vamos a usar el flag de collided, poniéndolo a 1. De modo que pondremos al enemigo (sprite 20) dicho flag a 1 en su byte de status

Ahora supongamos que el comando |COLSPALL detecta una colisión, y deja el resultado en collider y collided

```
|COLSPALL
If collider<32 then GOSUB 100
<instrucciones>

100 rem rutina de colision
110 dir=27000 + collider*16 :rem dirección de byte status de collider
120 if PEEK (dir) and 2 THEN RETURN: rem inofensivo si bit collided=1
130 <ha colisionado un enemigo que no es inofensivo>
```

La instrucción “**if PEEK (dir) and 2**” es la que comprueba el bit collided ya que 2 en binario es 00000010, justo la posición de ese bit en el estado del sprite.

Cuando el enemigo deje de ser inofensivo simplemente ponemos su flag de collided a 0 y ante una nueva colisión, nos matará.

Esta técnica es perfectamente válida usando cualquier otro flag que no se use.

## 8.11 Tabla de secuencias de animación

Las animaciones suelen componerse de un número par de fotogramas, aunque esto no es una regla estricta. Piensa por ejemplo en la animación simple de un personaje con solo dos fotogramas: piernas abiertas y cerradas. Son dos fotogramas. Ahora piensa en una animación mejorada, con una fase de movimiento intermedia. Esto supone crear la secuencia: cerradas-intermedia-abiertas-intermedia- y vuelta a empezar. Como ves es número par, son 4

Las secuencias de animación de 8BP son listas de 8 fotogramas, no pueden tener más, aunque siempre puedes hacer secuencias más cortas.

Los fotogramas de una secuencia de animación son las direcciones de memoria donde están ensambladas las imágenes de las que se componen, pudiendo ser diferentes en tamaño, aunque lo normal es que sean iguales. Si a mitad de la secuencia introduces un cero, el significado es que la secuencia ha terminado.

Aunque antes de la versión V33 existía un comando RSX llamado **|SETUPSQ** para crear secuencias desde BASIC, lo he eliminado en la V33, debido a que su uso es más complejo que definir secuencias desde el fichero sequences.asm y, de hecho, nunca he usado el comando **|SETUPSQ** en ninguno de mis juegos, de modo que decidí sacrificarlo para ahorrar memoria.

Veamos un ejemplo de creación de una secuencia en el fichero sequences.asm

```
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0
```

Antes de explicarte como asignar una secuencia a un sprite, te recuerdo cómo se asignan imágenes a sprites: Desde la versión V26 de 8BP, existe la posibilidad de incluir una lista de imágenes (sus etiquetas) en una lista llamada IMAGE\_LIST de tu fichero images\_tujuuego.asm. Con ello puedes referenciar las imágenes desde BASIC con un índice en lugar de una dirección de memoria. Así no tendrás que consultar las direcciones de memoria cada vez que ensambles. Esto aplica a la instrucción:

**|SETUPSP, #, 9, <dirección>**

El ejemplo muestra una secuencia de animación de 3 fotogramas diferentes, pero para que sea fluida antes de volver a volver a empezar hay que pasar por el fotograma “intermedio” otra vez (fíjate que el segundo y el cuarto son iguales), de modo que al final son 4 fotogramas:



y vuelta a empezar

Fig. 55 secuencia de animación

Si quisieras hacer una secuencia de más de 8 fotogramas podrías simplemente encadenar dos secuencias seguidas y cuando el personaje llegase al último fotograma de la primera secuencia usar el comando |SETUPSP para asignarle la segunda secuencia

Las secuencias de animación se ensamblan a partir de la dirección 33600 y puedes definir hasta 31 secuencias de animación (a partir del número 32 no se consideran secuencias, sino “macrosecuencias”, que es otro concepto). Cada secuencia estará identificada por un número que se encontrará en el rango [1..31]. La secuencia cero no existe, **se usa para indicar que un sprite no tiene secuencia**.

Para asignar una secuencia a un Sprite usa el comando SETUPSP con parámetro 7:

SETUPSP, <sprite_id>, 7, <sequence number>
--

Con este comando se asigna la secuencia de animación al sprite en el campo correspondiente de la tabla de sprites, y se pone un cero en el campo frame ID. Además, se le asigna la imagen correspondiente a la primera imagen de la secuencia. Si estás usando |ANIMALL antes de imprimir o |PRINTSPALL con flag de animación, aunque SETUPSP te coloque la animación en el frame cero, saltarás al frame 1 antes de imprimir. Esto normalmente no va a suponer ningún problema, pero en caso de tratarse de una “secuencia de muerte” (más adelante las veremos en detalle) en la que por ejemplo el primer frame es para borrar al sprite, puede que no te interese pasar directamente al frame 1. En ese caso un sencillo truco puede ser repetir el frame cero en la definición de la secuencia de muerte. Así te aseguras de que dicho frame se vea. Otra opción es quitarle el flag de animación y animarle con |ANIMASP después de imprimir.

Cada secuencia almacena 8 direcciones de memoria correspondientes a los 8 fotogramas, esto son 16 bytes consumidos por cada secuencia.

Tu fichero de secuencias de animación se puede parecer a esto:

```
;=====
; hasta 31 secuencias de animacion
;=====
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se gastan 8 words
; por secuencia
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo
; si a un Sprite se le asigna la secuencia cero es que no tiene secuencia.
; empezamos desde la secuencia 1
;-----secuencias de animacion del personaje montoya-----
SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0 ;1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0 ;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0 ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0 ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0 ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0 ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0 ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0 ;8

;-----secuencias de animacion del soldado -----
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;9
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;10
```

## 8.12 Secuencias de animación especiales

Dispones de varios tipos de secuencias de animación en 8BP:

Tipo de secuencia	descripción
Secuencia Normal	La secuencia de frames de la animación se repite una y otra vez. Terminan en una dirección de imagen o bien en

	un cero si queremos hacer una secuencia de menos de 8 imágenes
<b>Secuencia de muerte</b>	El último frame de la secuencia es un “1”. Esto le indica a 8BP que debe modificar el estado del Sprite a cero. Normalmente el frame anterior al “1” es una imagen de borrado.
<b>Secuencia de fin</b>	Tras recorrer la secuencia el Sprite se queda sin animación El último frame de la secuencia es un “2”. Esto le indica a 8BP que debe quitarle el flag de animación al estado del Sprite.
<b>Secuencia encadenada</b>	Tras recorrer la secuencia, el último frame indica el identificador de la siguiente secuencia que se asignará al Sprite. Mediante este tipo de secuencias puedes construir secuencias de animación más largas que 8 frames.
<b>macrosecuencias</b>	Permiten asociar una secuencia en función de la velocidad del Sprite, de forma automática.

### 8.12.1 Secuencias de muerte

La librería 8BP te permite hacer “secuencias de muerte”, que son secuencias que al terminar de recorrerlas, el sprite pasa a estado inactivo. Esto se indica con un simple “1” como valor de la dirección de memoria del fotograma final. Estas secuencias son muy útiles para definir explosiones de enemigos que están animados con |ANIMA o |ANIMALL. Tras alcanzarles con tu disparo, les puedes asociar una secuencia de animación de muerte y en los siguientes ciclos del juego pasarán por las distintas fases de animación de la explosión, y al llegar a la última pasarán a estado inactivo, no imprimiéndose más. Este paso a inactivo se hace automáticamente, de modo que lo que debes hacer es simplemente chequear la colisión de tu disparo con los enemigos y si colisiona con alguno le cambias el estado con SETUPSP para que no pueda colisionar más y le asignas la secuencia de animación de muerte, también con SETUPSP

Si usas una secuencia de muerte, no te olvides de que el último fotograma antes de encontrar el “1” sea uno completamente vacío, de modo que no quede ningún resto de la explosión.

Ejemplo de secuencia de muerte (fíjate que incluye un “1”):

```
dw EXPLOSION_1,EXPLOSION_2,EXPLOSION_3,1,0,0,0,0
```

Un efecto interesante es hacer pasar por varios fotogramas repetidamente antes de terminar con un fotograma negro que sirva para borrar

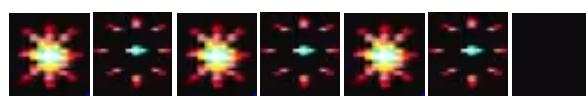


Fig. 56 Secuencia de muerte

Ahora el “1” aparece en octava posición:

```
dw EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2,  
EXPLOSION_3,1
```

Recuerda que si usas el comando |PRINTSPALL con el flag de animación activo, primero se anima y luego se imprime, por lo que la primera imagen de la secuencia de muerte no se verá. Un sencillo truco para que se vea es repetirla dos veces.

### 8.12.2 Secuencias de fin

Las secuencias de fin existen desde la versión V42. Permiten asociar una secuencia de animación a un Sprite que queremos que solo se lleva a cabo una vez. Al terminar la secuencia el Sprite quedará sin flag de animación. Los demás flags de su estado se respetan.

Simplemente necesitamos poner un “2” en el ultimo frame de la secuencia  
Aquí tenemos dos ejemplos. En cuanto la secuencia llega al frame “2”, el Sprite deja de estar animado.

```
SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,2,0,0,0 ;1
dw IMG1, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7,2 ;2
```

### 8.12.3 Secuencias encadenadas

Las secuencias encadenadas existen desde la versión 8BP V42 . Permiten construir secuencias más largas que 8 frames, encadenando unas secuencias con otras.

El mecanismo es sencillo. Simplemente el ultimo frame de la secuencia debe ser el numero la secuencia que quieras asignar a continuación.

Como puedes imaginar, no podrás asignar la secuencia “1” ni la “2”, puesto que esos números significan “morir” o “terminar”. Es decir, podrás encadenar secuencias a partir de la numero 3

En este ejemplo he encadenado las secuencias 4 y 5 de modo que tras una se asigna la otra y viceversa. Es como tener una secuencia de 14 fotogramas. Podríamos encadenar más secuencias y hacer la animación mucho más larga. Cada secuencia que añadimos son 7 fotogramas más (no 8) porque el último lo necesitamos para indicar la siguiente secuencia.

También puedes hacerlas más cortas y llenar con ceros hasta 8 frames, pero el número de secuencia debe aparecer antes que ningún cero pues al encontrar un cero la animación cicla.

```
_SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0 ;1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0 ;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0 ;3
dw IMG11, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7, 5 ;4
dw IMG18, IMG9, IMG10, IMG11, IMG12, IMG13, IMG14, 4 ;5
```

### 8.12.4 Macrosecuencias de animación

Esta es una característica “avanzada” disponible a partir de la versión V25 de la librería 8BP. Una “macrosecuencia” es una secuencia formada por secuencias. Cada una de las

secuencias de animación constituyentes es la animación que hay que efectuar en una dirección concreta. La dirección viene determinada por los atributos de velocidad del sprite, que están en la tabla de sprites. De este modo, cuando animemos a un sprite con |ANIMALL, automáticamente cambiará su secuencia de animación sin que tengamos que hacer nada (en realidad no hace falta invocar a |ANIMALL porque |PRINTSPALL ya lo hace internamente si se lo indicamos en un parámetro).

Las macrosecuencias se numeran comenzando en la 32. Es muy importante colocar las secuencias dentro de la macrosecuencia en el orden correcto, es decir, la primera secuencia debe ser para cuando el personaje está quieto, la siguiente para cuando va a la izquierda ( $Vx < 0$ ,  $Vy = 0$ ), la siguiente para la derecha ( $Vx > 0$ ,  $Vy = 0$ ), etc, siguiendo el siguiente orden (ten cuidado porque es fácil equivocarse):



*Fig. 57 Orden de secuencias en una macrosecuencia*

Si la secuencia asignada a la posición quieto es cero, entonces simplemente se anima con la última secuencia asignada.

Las macrosecuencias hay que especificarlas en el fichero sequences\_tujuego.asm, del que a continuación tienes un ejemplo:

```
;=====
; secuencias de animacion
;=====

SEQUENCES_LIST
dw NAVE,0,0,0,0,0,0,0;1
dw JOE1,JOE2,0,0,0,0,0;2 UP JOE
dw JOE7,JOE8,0,0,0,0,0;3 DW JOE
dw JOE3,JOE4,0,0,0,0,0;4 R JOE
dw JOE5,JOE6,0,0,0,0,0;5 L JOE

MACRO_SEQUENCES
-----MACRO SECUENCIAS -----
; son grupos de secuencias, una para cada dirección. el significado es:
; still, left, right, up, up-left, up-right, down, down-left, down-right
; se numeran desde 32 en adelante
db 0,5,4,2,5,4,3,5,4;la secuencia 32 contiene las secuencias del soldado Joe
```

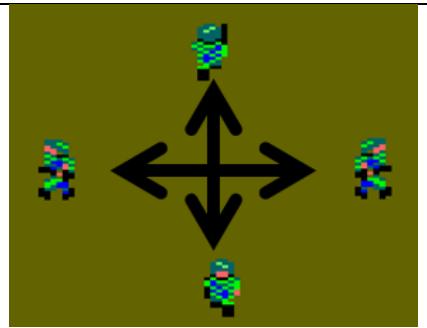
Con esa definición de secuencias podemos hacer un sencillo juego que permita mover a “Joe” por la pantalla sin controlar su secuencia de animación. Le asignamos la secuencia 32 y alterando la velocidad, el comando |ANIMA (invocado desde dentro de |PRINTSPALL) se encarga de cambiarle la secuencia de animación si su velocidad denota un cambio de dirección. Para mover al sprite necesitamos invocar a |AUTOALL, ya que al pulsar los controles no cambiamos sus coordenadas sino su velocidad y |AUTOALL actualizará las coordenadas del sprite de acuerdo a su velocidad.

<pre>10 MEMORY 24999 20 MODE 0:INK 0,12 30 ON BREAK GOSUB 280 40 CALL &amp;6B78 50 DEFINT a-z 111 x=36:y=100 120  SETUPSP,31,0,&amp;X1111 130  SETUPSP,31,7,2: SETUPSP,31,7,32</pre>	
--	--

```

140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,200
161 |PRINTSPALL,0,1,0
190 'comienza ciclo de juego
199 vy=0:vx=0
200 IF INKEY(27)=0 THEN vx=1: GOTO 220
210 IF INKEY(34)=0 THEN vx=-1
220 IF INKEY(69)=0 THEN vy=2: GOTO 240
230 IF INKEY(67)=0 THEN vy=-2
240 |SETUPSP,31,5,vy,vx
250 |AUTOALL:|PRINTSPALL
270 GOTO 199
280 |MUSIC:MODE 1: INK 0,0: PEN 1

```



Fíjate que no he definido la secuencia para cuando el personaje no se mueve. En dicha posición he puesto un cero en la macrosecuencia. Eso significa que, si el personaje comienza quieto, no se sabe qué secuencia asignar pues no hay una “última” secuencia usada. Es por ello que asigno la secuencia 2 antes de asignar la 32, así me aseguro de que el personaje ya tiene una secuencia, aunque se encuentre quieto.

<b>130  SETUPSP, 31, 7, 2: SETUPSP, 31, 7, 32</b>
---



## 9 Tu primer juego sencillo

Ya tienes los conocimientos para intentar un primer paso en la creación de videojuegos. Para ello vamos a ver un sencillo ejemplo de un soldado al que vas a controlar, haciéndole caminar a derecha e izquierda por la pantalla

Supongamos que hemos editado a un soldado, gracias a SPEDIT. Y hemos construido sus secuencias de animación, las cuales han sido definidas en el fichero “**sequences\_mygame.asm**” y han quedado con el identificador 9 y 10 para las direcciones de movimiento derecha e izquierda respectivamente.

Las dos secuencias de animación las hemos creado desde el fichero de secuencias.asm

```
10 MEMORY 24499
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,0:'fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla de juego
50 x=40:y=100: ' coordenadas del personaje
51 |SETUPSP,0,0,1:' status del personaje
52 |SETUPSP,0,7,9:'secuencia de animacion asignada al empezar
53 |LOCATESP,0,y,x:'colocamos al sprite (sin imprimirla aun)

60 'ciclo de juego
70 gosub 100
80 |PRINTSPALL,0,0
90 goto 60

99 ' rutina movimiento personaje -----
100 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,0,7,9:dir=0:return ELSE
|ANIMA,0:x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,0,7,10:dir=1:return ELSE
|ANIMA,0:x=x-1
120 |LOCATESP,0,y,x
130 RETURN
```

Con este listado ya tienes un minijuego que te permite controlar un soldado y hacerlo corretear horizontalmente. Fíjate que, si al caminar hacia la izquierda sobrepasas el valor mínimo del límite establecido con |SETLIMITS, se producirá el “clipping” del personaje, mostrándose tan solo la parte que queda dentro del área de juego permitida

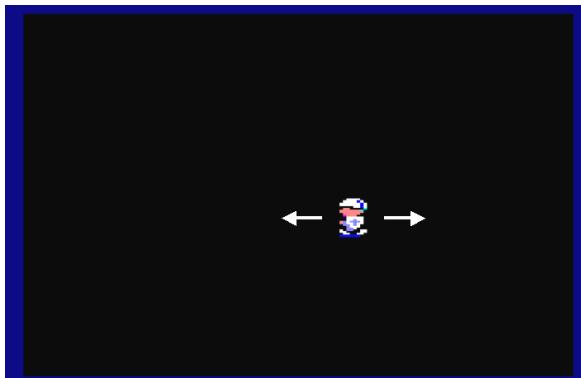
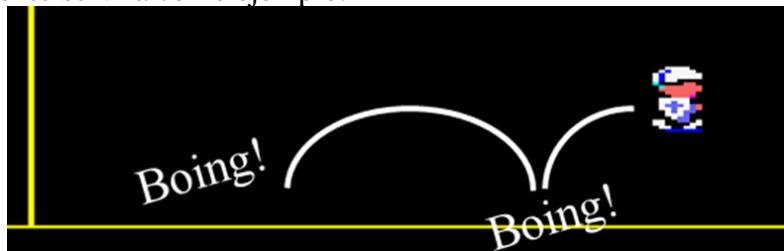


Fig. 58 Un sencillo juego

### 9.1 Ahora, a saltar! Boing, boing!!

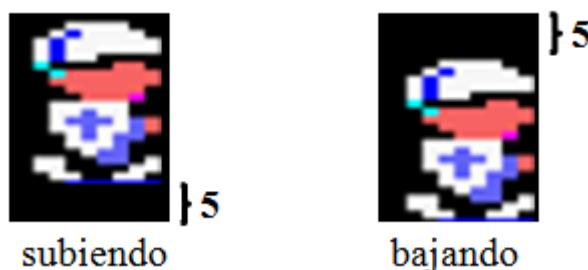
En el ejemplo anterior nuestro personaje sólo se mueve de izquierda a derecha. Si queremos programar un salto, podemos hacerlo almacenando la trayectoria vertical en un

array de BASIC. Más adelante veremos una forma mejor de hacerlo (con rutas de 8BP) pero de momento servirá como ejemplo.



*Fig. 59 nuestro personaje puede saltar*

La trayectoria de salto se define para la coordenada Y. Primero sube 5 líneas de golpe, luego sube 4, luego sube 3, etc. hasta llegar a cero. En ese momento se invierte la dirección de movimiento y comenzamos a bajar 1 línea, luego 2, luego 3, etc. hasta 5. Para que cuando suba y baje el muñeco no deje rastro, tendremos que tener un dibujo del muñeco subiendo con 5 líneas negras por debajo para borrarse a sí mismo al subir y del mismo modo, otra imagen con 5 líneas negras por encima para bajar. En este caso son las imágenes 22 y 23 para saltar a la derecha y 24,25 para la izquierda



*Fig. 60 Imagen de subida y de bajada*

En el punto cenital del salto se debe cambiar la imagen de subida por la de bajada, pero antes debemos subir de golpe 5 líneas pues si comparas ambas imágenes te darás cuenta de que si pasas de una a otra directamente es como bajar 5 líneas.

```

10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78: ' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02: 'restaura paleta por defecto por si acaso
40 INK 0,0: 'fondo negro
50 FOR j=0 TO 31: |SETUPSP,j,0,&X0:NEXT: 'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
90 x=40:y=100: ' coordenadas del personaje
100 |SETUPSP,0,0,1: ' status del personaje
110 |SETUPSP,0,7,9: 'secuencia de animacion asignada al empezar
120 |LOCATESP,0,y,x: 'colocamos al sprite (sin imprimirla aun)
121 DIM salto(24): ' datos del salto
122 for i=-5 to 5: k=k+1:salto(k)=i: k=k+1: salto(k)=i: next:
    salto(11)=-5: salto(23)=5
125 PLOT 1,150:DRAW 640,150: plot 92,150:draw 92,400: 'suelo y pared
126 |MUSIC,0,0,5: 'comienza a sonar la musica
130 'ciclo de juego -----
150 |LOCATESP,0,y,x: |PRINTSPALL,0,0
151 GOSUB 170
160 GOTO 130: ' fin ciclo juego

```

```

170 ' rutina movimiento personaje -----
171 IF jump =0 THEN IF INKEY(67)=0 THEN jump=1:|SETUPSP,0,9,DIR*2+22
180 IF INKEY(27)=0 THEN x=x+1:if jump=0 then IF dir<>0 THEN
|SETUPSP,0,7,9:dir=0:x=x-1:RETURN ELSE |ANIMA,0:GOTO 210
190 IF INKEY(34)=0 THEN x=x-1:if jump=0 then IF dir<>1 THEN
|SETUPSP,0,7,10:dir=1:x=x+1:RETURN ELSE |ANIMA,0
210 if jump=0 then RETURN
260 'rutina de salto -----
270 IF jump=11 THEN |SETUPSP,0,9,DIR*2+23 ELSE IF jump=23 THEN
y=y+salto(jump):jump=0:|SETUPSP,0,7,DIR+9:return
280 y=y+salto(jump)
300 jump=jump+1
310 return
2800 |MUSIC:MODE 1: INK 0,0:PEN 1

```

Como ves en el listado, si pulsas la tecla “Q”, la variable “jump” se iguala a 1 y en ese momento la lógica del muñeco se complica pues requiere ejecutar una sentencia IF para cambiar la imagen al llegar al punto cenital y además es necesario actualizar la coordenada Y del muñeco y la variable “jump”.

Más adelante veremos cómo hacer esto mismo con una técnica más avanzada, usando “rutas” de sprites. **Las rutas programables de 8BP proporcionan un método más eficiente de hacer este tipo de cosas, por lo que verás como tu personaje salta mucho más rápido.** Las rutas te van a permitir ejecutar una trayectoria (un salto, un círculo, etc.) sin necesidad de controlar en cada instante las coordenadas. Y además podrás cambiar el estado de un sprite en mitad de una ruta, o cambiarle su imagen asociada, su secuencia o incluso cambiarle de ruta, concatenando diferentes rutas.



# 10 Juegos de pantallas: layout o “tile map”

## 10.1 Definición y Uso del layout

A menudo querrás que tus juegos consistan en un conjunto de pantallas donde el personaje deba recoger tesoros o esquivar enemigos en un laberinto. En esos casos se hace indispensable el uso de una matriz donde definas los bloques constituyentes de cada “laberinto” o también llamado “layout” de la pantalla. A veces a este concepto también se le llama “mapa de tiles” (un “tile” es la palabra en inglés para decir “azulejo”)

En la librería **8BP** tienes un mecanismo sencillo para hacerlo, que además de proporciona una función de colisión para que compruebes si tu personaje se ha desplazado a una zona ocupada por un “ladrillo”. Este mecanismo se llama “layout”. En 8BP un layout se define con una matriz de 20x25 “bloques” de 8x8 pixeles, los cuales pueden estar ocupados o no. Es decir, hay tantos bloques como tiene la pantalla de caracteres en mode 0.

Para imprimir un layout en la pantalla dispones del comando:

**|LAYOUT, <y>, <x>, @string\$**

Esta rutina imprime una fila de sprites para construir el layout o “laberinto” de cada pantalla. La matriz o “mapa del layout” se almacena en una zona de la memoria que maneja 8BP de modo que cuando imprimes bloques en realidad **no solo estás imprimiendo en la pantalla, sino que también estas rellenando el área de memoria que ocupa el layout** (20x25 bytes) donde cada byte representa un bloque.

Las coordenadas y,x se pasan en formato caracteres, es decir

y toma valores [0,24]

x toma valores [0,19]

Los bloques que imprime la función |LAYOUT se construyen con cadenas de caracteres y cada carácter se corresponde con un sprite que debe existir. De este modo el bloque “Z” se corresponde con la imagen que tenga asignada el sprite 31. El bloque “Y” se corresponde con la imagen que tenga asignada el sprite 30, y así sucesivamente.

Los sprites a imprimir se definen con un string, cuyos caracteres (32 posibles) representan a uno de los sprites siguiendo esta sencilla regla, donde la única excepción es el espacio en blanco que representa la ausencia de sprite.

<b>Caracter</b>	<b>Sprite id</b>	<b>Codigo ASCII</b>
“ “	NINGUNO	32
“;”	0	59
“<”	1	60
“=”	2	61
“>”	3	62
“?”	4	63
“@”	5	64
“A”	6	65
“B”	7	66
“C”	8	67
“D”	9	68
“E”	10	69
“F”	11	70
“G”	12	71
“H”	13	72
“I”	14	73
“J”	15	74
“K”	16	75
“L”	17	76
“M”	18	77
“N”	19	78
“O”	20	79
“P”	21	80
“Q”	22	81
“R”	23	82
“S”	24	83
“T”	25	84
“U”	26	85
“V”	27	86
“W”	28	87
“X”	29	88
“Y”	30	89
“Z”	31	90

*Tabla 4 correspondencia entre caracteres y Sprites para el comando /LAYOUT*

El @string es una variable de tipo cadena. No puedes pasar directamente la cadena. Es decir, sería ilegal algo como:

| LAYOUT, 1, 0, "ZZZ YYY"

Lo correcto es:

Cadena\$ = "ZZZ YYY"  
| LAYOUT, 1, 0, @cadena\$

Ten cuidado de que la cadena no esté vacía, ¡de lo contrario puede bloquearse el ordenador!. Además, debes anteponer el símbolo “@” en la variable de tipo string para

que la librería pueda ir a la dirección de memoria donde se almacena la cadena y así poder recorrerla, imprimiendo uno a uno los sprites correspondientes.

Debes tener en cuenta que los espacios en blanco significan ausencia de sprite, es decir, en las posiciones correspondientes a los espacios no se imprime nada. Si había previamente algo en esa posición, no se borrará. Si deseas borrar necesitas definirte un sprite de borrado de 8x8, donde todo sean ceros.

Aunque usas los sprites para imprimir el layout, justo después de imprimirla puedes redefinir los sprites con |SETUPSP y asignarles imágenes de soldados, monstruos o lo que quieras, es decir, el layout se “apoya” en el mecanismo de sprites para imprimir, pero no te limita el número de sprites, pues dispones de los 32 para que sean lo que tú quieras justo después de imprimir el layout

Para detectar colisiones con el layout dispones de la función COLAY, que se puede usar con un número de parámetros variable.

```
|COLAY,<umbral ASCII>, @colision , <sprite number>
|COLAY, @colision , <sprite number>
|COLAY, <sprite number>
|COLAY
```

Dado un sprite y dependiendo de sus coordenadas y de su tamaño, esta función averiguará si está colisionando con el layout y te avisará a través de la variable colisión, la cual debe estar previamente definida.

El parámetro **<umbral ASCII>** es opcional y sirve para que el comando no considere colisión a aquellos códigos ASCII inferiores a dicho umbral. Por defecto es 32 (que es el correspondiente al espacio en blanco). Para entender esto hay que tener en cuenta la correspondencia entre valores ASCII y Sprites que se ha mostrado en la tabla anterior. Por ejemplo, si ponemos como umbral el 69 (código de la “E”, sprite 10), entonces los sprites 9, 8, 7, 6, 5, 4, 3, 2, 1, y 0 no serán “colisionables”, de modo que, si nuestro personaje pasa por encima, simplemente no será detectada la colisión.

Tan solo hace falta invocar a COLAY con el parámetro de umbral una vez, ya que las sucesivas invocaciones tienen ya en cuenta dicho umbral.

Ejemplo de uso:

```
col%=0
|COLAY, @col%, 20: REM este es un ejemplo con spriteID=20
```

Si invocas al comando COLAY sin parámetros, considerará los últimos que usó. De ese modo te puedes ahorrar el paso de parámetros y acelerar el comando 0.5 ms.

Si no hay colisión, la variable tomará el valor cero. Si hay colisión, tomará el valor 1. Hay colisión si el sprite choca con algún elemento del layout diferente del espacio en blanco (“ ”), cuyo código ASCII es 32. En caso de usar el umbral, habría colisión si el elemento del layout tiene un ASCII superior al umbral que se defina.

Vamos a ver un ejemplo de creación de un layout y de movimiento de un personaje dentro del layout, corrigiendo su posición si ha colisionado.

Dos últimas consideraciones sobre el comando |COLAY:

- Al comando |COLAY, no le afecta el ajuste de la sensibilidad de colisión entre sprites (configurable con |COLSP,34, dy, dx). El ajuste de sensibilidad de colisión solo afecta a los comandos |COLSP y |COLSPALL
- El comando |COLAY no tiene en cuenta el tamaño real de las imágenes usadas como “Tiles” o “bloques” del layout. Es decir que considera que todos los bloques especificados en el comando |LAYOUT miden 8x8 pixeles de mode 0 (4 bytes x 8 líneas), aunque tu pongas imágenes más grandes.

## 10.2 Ejemplo de juego con layout

Vamos a evolucionar un poco el juego presentado en el anterior capítulo, en lo que respecta al control del personaje. Esta vez vamos a usar a Montoya como ejemplo, el cual tiene 8 secuencias de animación, cada una para moverse en una dirección diferente. A las secuencias de animación les hemos asignado un número que va del 1 al 8.

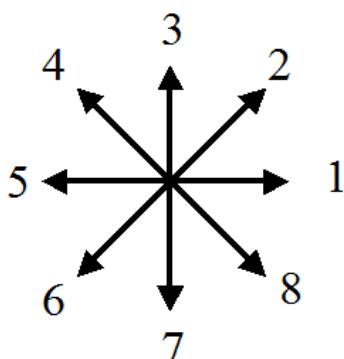


Fig. 61 Uso del layout en un juego

En la rutina de control del personaje hemos incluido colisión con el layout. En función de la dirección en la que avanzamos, modificamos las coordenadas “nuevas” (yn , xn) e invocamos a la función de colisión con layout |COLAY,0 para chequear si el sprite 0 (nuestro personaje) ha colisionado. Si ha colisionado, corregimos las coordenadas (una o las dos) para dejarle en una posición sin colisión antes de imprimirla de nuevo

```

10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 CALL &bc02:'restaura paleta por defecto por si acaso
26 ink 0,0:'fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,0,80,0,200: ' establecemos los limites de la pantalla de juego
50 dim c$(25):for i=0 to 24:c$(i)="":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZ"
110 c$(2)= "Z"           Z"
120 c$(3)= "Z"           Z"
125 c$(4)= "Z"           Z"
130 c$(5)= "Z  ZZZ  ZZZZ  ZZZ  Z"
140 c$(6)= "Z  ZZZ  ZZZZ  ZZZ  Z"
150 c$(7)= "Z  ZZZ  ZZZZ  ZZZ  Z"

```

```

160 c$(8)= "Z" Z"
170 c$(9)= "Z" Z"
190 c$(10)="Z" Z"
195 c$(11)="Z" ZZZZZ ZZZZZZ Z"
200 c$(12)="Z" ZZZZZ ZZZZZZ Z"
210 c$(13)="Z" Z"
220 c$(14)="Z" Z"
230 c$(15)="Z" Z"
240 c$(16)="ZZZZZZZZZZZZZZZ ZZZZ"
250 c$(17)="Z" Z"
260 c$(18)="Z" Z"
270 c$(19)="Z" Z"
271 c$(20)="Z" Z"
272 c$(21)="Z" Z"
273 c$(22)="Z" Z"
274 c$(23)="ZZZZZZZZZZZZZZZZZ"
300 gosub 550: ' imprime el layout
310 xa=40: xn=xa: ya=150: yn=ya: ' coordenadas del personaje
311 |SETUPSP,0,0,&x111: ' detección de colisión con sprites y layout
312 |SETUPSP,0,7,1: ' secuencia = 1
320 |LOCATESP,0,ya,xa: ' colocamos al personaje (sin imprimirla)
325 c1%=0: 'declaramos la variable de colisión, explicitamente entera (%)

330 '----- ciclo de juego -----
340 gosub 1500:'rutina de lectura teclado y movimiento de personaje
350 |PRINTSPALL,0,0
360 goto 340

550 'rutina print layout-----
560 FOR i=0 TO 23:|LAYOUT,i,0,@c$(i):NEXT
570 RETURN

1500 ' rutina movimiento personaje -----
1510 IF INKEY(27)<0 GOTO 1520
1511 IF INKEY(67)=0 THEN IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:GOTO 1533 ELSE
|ANIMA,0: xn=xa+1:yn=ya-2:GOTO 1533
1512 IF INKEY(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:GOTO 1533 ELSE
|ANIMA,0: xn=xa+1:yn=ya+2:GOTO 1533
1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:GOTO 1533 ELSE |ANIMA,0: xn=xa+1:GOTO
1533
1520 IF INKEY(34)<0 GOTO 1530
1521 IF INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:GOTO 1533 ELSE
|ANIMA,0: xn=xa-1:yn=ya-2:GOTO 1533
1522 IF INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:GOTO 1533 ELSE
|ANIMA,0: xn=xa-1:yn=ya+2:GOTO 1533
1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:GOTO 1533 ELSE |ANIMA,0: xn=xa-1:GOTO
1533
1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:GOTO 1533 ELSE
|ANIMA,0: yn=ya-4:GOTO 1533
1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:GOTO 1533 ELSE
|ANIMA,0: yn=ya+4:GOTO 1533
1532 RETURN
1533 |LOCATESP,0,yn,xn:ynn=yn:|COLAY,@c1%,0:IF c1%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,@c1%,0:IF c1%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,@c1%,0:IF c1%=1
THEN yn=ya:|POKE,27001,yn
1536 ya=yn:xa=xn
1537 RETURN

```

### 10.3 Cómo abrir una compuerta en el layout

Si deseas que tu personaje pueda coger una llave y abrir una compuerta o en general eliminar una parte del layout para permitir el acceso, lo que tienes que hacer son dos pasos:

- 1) Tener definido un sprite de borrado de 8x8 donde todo sean ceros. Usando |LAYOUT lo imprimes en las posiciones que deseas
- 2) A continuación, usando nuevamente |LAYOUT, imprimes espacios donde has borrado. Así el map layout quedará con el carácter “ ” en esas posiciones y la función de colisión con el layout resultará cero.

En el juego “Mutante Montoya” se utiliza esta técnica para abrir la puerta del castillo, así como para abrir las compuertas que conducen a la princesa.



Fig. 62 Modificación del layout al coger la llave

En el siguiente ejemplo se ilustra el concepto, abriendo una compuerta situada en las coordenadas (10, 12) de un tamaño de 2 bloques, al coger una llave que está definida con el sprite 16.

Nada más coger la llave se abre la compuerta y la llave queda desactivada para no evaluar más veces la colisión con ella, es decir, el comando |COLSP retornará un 32 a partir del momento que cojas la llave si vuelves a colisionar con ella.

Tras abrir la compuerta, si desplazas el personaje hasta el lugar que ocupaba dicha compuerta, la colisión con layout dará como resultado 0.

```
----- esta parte esta dentro del bucle de logica -----
6410 |PRINTSPALL,1,0
6411 |COLSP,@cs%,0:IF cs%<32 THEN IF cs%>=15 then gosub 6500
(... mas instrucciones . . .)

----- rutina de apertura de compuerta-----
6499 ' comprueba que tu colision sea con la llave, que es el sprite 16
6500 borra$="MM":spaces$=" ":" el sprite de borrado se ha definido
como "M" (M es el sprite 18 en el "idioma" del comando |LAYOUT)
6501 if cs%=16 then|LAYOUT,10,12,@borra$ : |LAYOUT,10,12,@spaces$:
|SETUPSP,16,0,0
6502 return
```

## 10.4 Un comecocos: LAYOUT con fondo

A continuación, vamos a ver un ejemplo que usa layout y sobreescritura, para lo cual establece un umbral ASCII que permite que el comando |COLAY no considere colisión con los elementos de fondo. Concretamente como elemento de fondo se usa la letra “Y”, la cual se corresponde con el sprite id= 30, y el ASCII de la “Y” es el 89.



Fig. 63 Layout con un patrón de fondo y sobreescritura

Como puedes ver en el ejemplo tan solo hace falta invocar a COLAY con el parámetro de umbral una vez, ya que las sucesivas invocaciones tienen ya en cuenta dicho umbral

Otro de los aspectos interesantes es la gestión del teclado de este ejemplo. Es óptima para ejecutar el menor número de operaciones |COLAY y a la vez da una sensación muy agradable al avanzar por un pasillo y conectar con otro teniendo dos teclas pulsadas a la vez

```
10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78: ' install RSX
21 on break gosub 5000
25 call &bc02: 'restaura paleta por defecto por si acaso
26 gosub 2300: ' paleta con sobreescritura
30 FOR j=0 TO 31: |SETUPSP,j,0,&X0:NEXT: 'reset sprites
40 |SETLIMITS,0,80,0,200: ' limites de la pantalla de juego
45 |SETUPSP,30,9,&84d0: ' rejilla de fondo ("Y")
46 |SETUPSP,31,9,&84f2: ' ladrillo ("Z")
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZ"
110 c$(2)= "YYYYYYYYYYYYYYYYYYYYYZ"
120 c$(3)= "YYYYYYYYYYYYYYYYYYYYYZ"
125 c$(4)= "ZYZZYZZZZZZYZZZYZ"
130 c$(5)= "ZYZZYZZZZZZYZZZYZ"
140 c$(6)= "YYYYYYYYYYYYYYYYYYYYYZ"
150 c$(7)= "YYYYYYYYYYYYYYYYYYYYYZ"
160 c$(8)= "ZYZZZZZZYZZZZZZZYZ"
170 c$(9)= "ZY"      "YZ"      "ZY"
190 c$(10)= "ZY"      "YZ"      "ZY"
195 c$(11)= "ZYZZZZZZYZZZZZZZYZ"
200 c$(12)= "YYYYYYYYYYYYYYYYYYYYYZ"
210 c$(13)= "YYYYYYYYYYYYYYYYYYYYYZ"
220 c$(14)= "ZYZZYZZZZZZYZZZYZ"
230 c$(15)= "YYYYYYYYYZ" "YYYYYYYYYZ"
240 c$(16)= "YYYYYYYYYZ" "YYYYYYYYYZ"
250 c$(17)= "ZYZZZZYZZZYZZZZZYZ"
260 c$(18)= "YYYYYYYYYYYYYYYYYYYYYZ"
270 c$(19)= "YYYYYYYYYYYYYYYYYYYYYZ"
```

```

271 c$(20)="ZZZZZZZZZZZZZZZZZZZZZ"
272 c$(21)=""
273 c$(22)=""
274 c$(23)=""
300 'imprimimos el layout
310 FOR i=0 TO 20:|LAYOUT,i,0,@c$(i):NEXT
311 locate 1,1:pen 9:print "DEMO SOBREESCRITURA"
312 locate 3,23:pen 11:print "BASIC usando 8BP"
320 |SETUPSP,0,0,&x01000111:' deteccion colision con sprites y layout
330 |SETUPSP,0,7,1:dir=1: ' secuencia = 1 (coco derecha)
340 xa=20*2:xn=xa:ya=12*8:yn=ya:' coordenadas del personaje
350 |LOCATESP,0,ya,xa: 'colocamos al personaje (sin imprimirlo)
360 |PRINTSPALL,0,1,0:' imprime sprites
361 cl%=0:' variable colision
362 |COLAY,89,cl%,0:' umbral chr$("Y") es 89
400 ' COMIENZA EL JUEGO
401 |MUSIC,0,0,5
402 ' lectura teclado y colisiones. si vamos en direccion H (o p), primero
  chequeamos si hay pulsada tecla direccion V (q a) y viceversa
404 if dirn <3 then gosub 450: gosub 410 else gosub 410:gosub 450
405 |LOCATESP,0,yn,xn:|PRINTSPALL
406 ya=yn:xa=xn
407 goto 404

409 ' teclado direccion horizontal ---
410 if INKEY(27)<0 then 430
420 xn=xa+1poke 27003,xn:|COLAY: IF cl%=0 then if dir<>1 then
  |SETUPSP,0,7,1:DIR=1:xn=xa:return else dirn=1:return
421 xn=xa:poke 27003,xn:return:'hay colision
430 if INKEY(34)<0 then return
440 xn=xa-1:poke 27003,xn:|COLAY: IF cl%=0 then if dir<>2 then
  |SETUPSP,0,7,2:DIR=2:xn=xa:return else dirn=2:return
441 xn=xa:poke 27003,xn:'hay colision
442 return
449 'teclado direccion vertical
450 if INKEY(67)<0 then 480
460 yn=ya-2:poke 27001,yn:|COLAY: IF cl%=0 then if dir<>3 then
  |SETUPSP,0,7,3:DIR=3:yn=ya:return else dirn=3:return
461 yn=ya:poke 27001,yn:'hay colision
480 if INKEY(69)<0 then return
490 yn=ya+2:poke 27001,yn:|COLAY: IF cl%=0 then if dir<>4 then
  |SETUPSP,0,7,4:DIR=4:yn=ya:return else dirn=4:return
491 yn=ya:poke 27001,yn:'hay colision
492 return

2300 REM ----- PALETA sprites transparentes MODE 0-----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : INK 3,0: REM negro
2305 INK 4,26: INK 5,26: REM blanco
2307 INK 6,6: INK 7,6: REM rojo
2309 INK 8,18: INK 9,18: REM verde
2311 INK 10,24: INK 11,24: REM amarillo
2313 INK 12,4: INK 13,4 :REM magenta
2315 INK 14,16 : INK 15, 16:REM naranja
2317 AMARILLO=10
2420 RETURN
5000 |MUSIC
5010 end

```

## 10.5 Cómo ahorrar memoria en tus layouts

Si tu juego tiene muchas pantallas y necesitas ahorrar espacio puedes utilizar muchas técnicas sencillas para ahorrar memoria. Una pantalla consume casi 0.5 KB de modo que es importante usar métodos para reducir su tamaño

La forma más sencilla de hacerlo es editar las pantallas como si fuesen sprites. Las editas con SPEDIT (por ejemplo) y cada pixel representará un elemento del layout. Dependiendo del color, representará rocas, ladrillos, espacio vacío, agua, tierra, etc. Como hay 16 colores en mode 0 disponibles, tendrás 16 tipos de ladrillo. El sprite que generes lo tendrás que almacenar como una imagen más y antes de mostrarlo en pantalla, convertirlo a layout, explorando pixel a pixel y transformándolo en el código ASCII que necesites (esto te lo tendrás que programar en BASIC o como deseas). Si consideramos que usarás 5 líneas de caracteres para los marcadores del juego, una pantalla consumirá 20x20 pixeles = 400 pixeles = 200 Bytes. Por consiguiente, en 1KB te caben 5 pantallas y en 10KB te caben 50 pantallas. Habrás usado 4 bits por elemento de layout.

La edición de pantallas como si fuesen sprites es muy “visual”, aunque puedes decidir usar menos bits por elemento del layout. Si usas solo 2 bits, tendrás 4 tipos de elementos y también podrás editar pantallas como si fuesen imágenes, usando MODE 1. En ese caso te cabrán 100 pantallas en 10KB. Si usas un numero diferente de bits por ladrillo la cosa se complica pues no puedes dibujar las pantallas como si fuesen sprites, pero te puedes programar algo que te convierta una imagen que dibujes en los bits que necesites.

Otra solución sencilla y eficaz consiste en definir cada layout con bloques grandes, por ejemplo, de 8x16 pixels. Y construir las pantallas definiéndolas con caracteres que podemos almacenar en memoria. En el video juego “**Happy Monty**”, se recrea la primera pantalla del “Mutant Monty” con una matriz de 16x10 caracteres =160 bytes, de modo que 25 pantallas ocupan 4 KB. Es cierto que este layout tiene solo 16 bloques de ancho y no los 20 que puede llegar a tener, y además solo se definen 10 bloques verticalmente, en lugar de los 25 que puede llegar a tener, pero así ocupa muy poca memoria y podemos crear muchas pantallas.

```
"IK m      o      JG"
" IGGGGGH IGGGGHq"
" z      c      xo      "
" C      CCGK      d"
" F      OC      IDD      DDDDD "
" F      C      F      F "
" Fv      C      JHz      b      xoF "
" F      C      IGGGGGGGGH "
" IGGH      z      a      xw"
" EEEEEoEEE      "
```



Fig. 64 un layout definido con 160 caracteres



# 11 Programación avanzada y “lógicas masivas”

## 11.1 Medición de la velocidad de los comandos

El intérprete BASIC es muy pesado en ejecución debido a que no solo ejecuta cada comando, sino que analiza el número de línea, realiza un análisis sintáctico del comando introducido, valida su existencia, el número y tipo de parámetros, que sus valores que se encuentren en rangos validos (por ejemplo, PEN 40 es ilegal) y muchas más cosas. Es el análisis sintáctico y semántica de cada comando lo que realmente pesa y no tanto su ejecución. El caso de los comandos RSX no es una excepción. El intérprete BASIC comprueba su sintaxis y eso pesa mucho, a pesar de que sean rutinas escritas en ASM, pues antes de invocarlas, el intérprete BASIC ya ha hecho muchas cosas.

Por consiguiente, hay que ahorrar ejecuciones de comandos, programando con astucia para que la lógica del programa pase por el menor número de instrucciones posibles, aunque ello a veces implique escribir más. Una práctica indispensable es usar instrucciones que muevan o afecten a un grupo de sprites, tales como COLSPALL, |AUTOALL o |MOVEALL, evitando el uso de bucles con instrucciones que afectan a un solo sprite.

Un factor decisivo a la hora de invocar un comando es el paso de parámetros. Cuantos más parámetros tiene, más costoso es su interpretación por parte del BASIC, incluso aunque sea una rutina ASM que se invoque por CALL, pues el comando CALL sigue siendo BASIC y antes de acceder a la rutina en ASM, se analiza el número y tipo de parámetros irremediablemente.

Para evaluar el coste de ejecución de un comando puedes usar el siguiente programa. También te servirá para evaluar el rendimiento de nuevas funciones en ensamblador que incorpores a la librería 8BP si deseas hacerlo.

```
1 call &6b78
10 MEMORY 23499
11 DEFINT a-z
12 c%=0: a=2
30 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT: 'reset
31 iteraciones=1000
40 a!= TIME
50 FOR i=1 TO iteraciones
60 <aquí pones un comando, por ejemplo, PRINT "A">
70 NEXT
80 b!=TIME
90 PRINT (b!-a!): rem lo que tarda en unidades de tiempo cpc. (1/300
segundos)
100 c!=((b!-a!)*1/300)/iteraciones: rem c! = lo que tarda cada iteracion
en segundos
120 d!=(1/50)/c!
130 PRINT "puedes ejecutar ",d!, "comandos por barrido (1/50 seg)"
140 PRINT "el comando tarda ";(c!*1000 -0.47);"milisegundos"
```

Nota: para los expertos en lenguaje ensamblador, debéis tener en cuenta que si pretendéis medir el tiempo de ejecución de una rutina que internamente desactiva las interrupciones (usa las instrucciones DI, EI) el tiempo que transcurre durante la desactivación no es medible con este programa BASIC. Los comandos de 8BP no desactivan las interrupciones y son todos medibles.

Vamos a ver a continuación el resultado del rendimiento de algunos comandos (medidos con el programa anterior). Hay que decir que es más rápido ejecutar una llamada directa a la dirección de memoria (un CALL &XXXX) que invocar el comando RSX correspondiente. En la siguiente tabla obviamente cuanto menor sea el resultado (expresado en milisegundos), más rápido es el comando. La tabla que aquí se presenta debes tenerla en todo momento presente y tomar tus decisiones de programación en base a ella. Es una tabla con medidas de comandos BASIC y comandos 8BP

Comando	ms	Comentario
<b>PRINT "A"</b>	3.63	Lentísimo. Ni se te ocurra usarlo, salvo puntualmente para cambiar el número de vidas, pero no imprimas puntuación en un juego por cada enemigo que mates
<b>LOCATE 1,1: PRINT puntos</b>	24.8 + 7	Colocar el cursor de texto con LOCATE e imprimir el valor de luna variable “puntos” es costosísimo. Si actualizas puntos hazlo sólo de vez en cuando y no en cada ciclo de juego
<b>C\$=str\$(puntos)  PRINTAT,0, y, x, @c\$</b>	10	Imprimir los puntos usando PRINTAT es mucho mas eficiente que usar LOCATE + PRINT (=32 ms) , pero aun asi es costoso. Usa PRINTAT con moderación.
<b>REM hola</b>	0.20	Los comentarios consumen
<b>' hola</b>	0.25	Ahorras 2 bytes de memoria, pero es más lento!!
<b>GOTO 60</b>	0.19	Muy rápido!!! Más rápido incluso que REM. Usa este comando sin piedad, úsallo!!!
<b>A = 3</b>	0.55	Una simple asignación cuesta. Todo cuesta, cada instrucción debe ser pensada.
<b>A = B</b>	0.72	Asignar el valor de una variable a otra es mas costoso que asignar un valor. Y asignar el valor de un array es aun mas costoso, porque acceder al array cuesta. Y si el array es bidimensional aun cuesta mas.
<b>A = miarray(x)</b>	1.33	
<b>A= miarray(x,y)</b>	1.84	
<b> LOCATESP,i,10,20</b>	2.8	Si no usas coordenadas negativas es mejor usar el comando BASIC POKE para establecer coordenadas.
<b> LOCATESP,i,y,x</b>	3.22	Si las coordenadas son variables entonces tarda mas.
<b>CALL &amp;XXXX,i,x,y</b>	1.81	El equivalente CALL es mucho mas rápido.
<b> MOVER,31,1,1</b>	3.23	Es algo lento y por ello debes usarlo con moderación
<b>CALL &amp;XXXX,31,1,1</b>	1.77	El equivalente call es mucho mas rápido
<b>POKE &amp;XXXX, valor</b>	0.71	Muy rápido! Úsalo para actualizar las coordenadas de los sprites (si son positivas) POKE no acepta números negativos, pero puedes usar la formula 255+x+1 si quieres meter un numero negativo. Por ejemplo, para meter un -4 debes meter 255-4+1=252 Otra forma sencilla de meter positivos y negativos es usar POKE dirección, x and 255
<b>POKE dir,dato</b>	0.85	Muy rápido teniendo en cuenta que además debe traducir la variable “dir”
<b> POKE,&amp;xxxx,valor</b>	2.5	Permite números negativos y si sólo actualizas una coordenada (X o Y) es mejor que LOCATESP

<code>X=PEEK(&amp;xxxx)</code>	0.93	Muy rápido! Según el tipo de videojuego puede ser una alternativa a COLSP, mirando el color de una dirección de memoria de pantalla. En el apéndice sobre la memoria de video te explico como hacerlo.
<code>X=INKEY(27)</code>	1.12	Muy rápido. Apto para videojuegos, aunque debes usarlo inteligentemente como se recomienda en este libro.
<code>IF x&gt;50 THEN x=0</code>	1.42	Cada IF pesa, hay que tratar de ahorrarlos porque una lógica de juego va a tener muchos
<code>IF A=valor THEN GOTO 100</code>  <b>Vs</b>  <code>IF A=valor THEN 100</code>	1.24  Vs  1.18	Ambas sentencias son equivalentes pero la segunda tarda menos
<code>IF inkey(27)=0 then x=5</code>	1.75	Aceptable. Es más rápido que hacer <code>b=INKEY(27)</code> y después el IF...THEN
<code>10 If inkey(27) then 30 20 x=5 30 &lt;instrucciones&gt;</code>	1.0	Una forma mucho mas eficiente de hacer lo mismo
<code>IF x&gt;0 then</code>  <b>Vs</b>  <code>IF x then</code>	1.3  Vs  0.8	En BASIC es posible ahorrar 0.5ms teniendo en cuenta que cualquier valor distinto de cero significa TRUE. Si queremos controlar un valor concreto haremos:  <code>10 IF x=20 THEN 30 20 &lt;cosas a hacer si x=20&gt; 30 ...</code> El uso de esta técnica es muy recomendable en la lectura de teclado
<code>A=A+1: IF A&gt;4 then A=0</code>  <b>Vs</b>  <code>A=A MOD 3 +1</code>  <code>A=A MOD 3 + &lt;impar&gt;</code>	2.6  Vs  1.7	Es mucho mejor usar la segunda opción (usando MOD) Por otro lado, el uso de MOD hay que hacerlo con cautela. Si hacemos:  <code>A=(A+1) MOD 3</code> Nos cuesta 2 ms ya que los paréntesis son muy costosos y sin embargo conseguimos lo mismo. Si queremos hacer la cuenta desde un numero distinto de 1, ponemos un impar cualquiera y con ello lo conseguiremos.  Hay una forma mas rápida aun de hacer esto lo, con el operador binario AND que ahora veremos
<code>A=1+A AND 7 (valor inicial 0 Valor final 7)</code>  <code>A=20 +A MOD 7 (valor inicial 0 Valor final 26)</code>  <code>A=21 + (A and 7) (valor inicial 21 Valor final 28)</code>	1.6  1.88  1.95	<b>Esto te permite hacer variar una variable cíclicamente entre N valores</b> de modo que te sirve para elegir un sprite ID para tu nuevo disparo o para un enemigo que entra en pantalla  Es mejor usar AND que MOD, ya que AND es una operación binaria rápida y MOD implica una división, muy costosa para nuestro querido microprocesador Z80. Sin embargo, si necesitamos usar sprites ID que no comiencen en 1, entonces necesitaremos paréntesis porque el operador “+” tiene prioridad frente a “AND” y la ventaja de velocidad del AND se pierde. En ese caso

		es mejor MOD. De todos modos, pruebalo siempre y elige porque dependiendo del numero inicial a sumar obtendrás tiempos distintos. Increíble pero cierto 20+A mod 7 → tarda 1.88 29+A mod 7 → tarda 1.94
<b>If A &lt;0 then A=15</b> <b>A=A AND 15</b>	1.71 1.24	<b>Comprobar que un numero no es negativo</b> Se puede simplificar la comprobación porque un numero negativo en realidad es un numero que tiene un “1” en el bit mas significativo y le quitamos la negatividad con un simple and
:	0.05	No ahorra mucho, pero es mas rápido usar “:” en lugar de un nuevo número de linea, y si aplicas esto muchas veces acabas ahorrando de forma significativa. Dos instrucciones en dos lineas gastan 0.03ms mas que si ambas estan en la misma linea separadas por “:”
PRINTSP,0,10,10	5.3	Un solo sprite de 14 x 24 (7 bytes x 24 lineas) Ojo, si vas a imprimir varios compensa mucho mas imprimir todos los sprites de golpe con PRINTSPALL
CALL &xxxx,0,10,10	3.5	Equivalente a PRINTSP, así es mas rápido, aunque menos legible
PRINTSPALL  (32 sprites 8x16 de mode 0, es decir 4 bytes x 16 lineas)	55.4	<p>Esto son unos 18 fps a plena carga de sprites. Lo que tarda es</p> $T = 3.25 + N \times 1.7$ <p>Es decir, 1.7 ms por sprite y un coste fijo de 3 ms. Este coste fijo es el coste del análisis sintáctico de BASIC sumando al de recorrer la tabla de sprites buscando cuales hay que imprimir. Si se omiten los parámetros (es posible y se tomarian los valores de la ultima invocación), se ahorran 0.6ms en la parte fija, es decir:</p> $T = 2.6 + N \times 1.1$ <p>Si la impresión es con sobreescritura y/o flipeada, es mas costosa. A continuación, se muestran los costes relativos de cada tipo de impresión:</p> <p><b>Impresión normal: 100%</b>  <b>Impresión con sobreescritura: 164%</b>  <b>Impresión flipeada: 179%</b>  <b>Impresión flipeada con sobreescritura: 220%</b></p>
PRINTSPALL,N,0,0 (ningún sprite activo)  N=0 N=10 N=31	2.6 4.3 5.9	<p>Coste de ordenar los sprites:</p> <p>Cuando N=0, no habiendo ningún sprite que imprimir, la función debe recorrer la tabla de sprites de forma secuencial. Pero recorrerla de forma ordenada es más costoso, tal como evidencia el tiempo consumido al aumentar N. La diferencia de tiempos (5.9 -2.6 =2.5ms) es lo que cuesta ordenar todos los sprites</p>
COLAY,@x%,0	3.0 Vs	Usar solo con el personaje, no con los enemigos o el juego ira lento. Si el personaje mide múltiplos de 8 es más rápido. En este ejemplo era de 14x24 y lógicamente 14

COLAY	2.4	no es múltiplo de 8. cuanto mayor es el sprite más tarda. ¡Si invocas el comando sin parámetros es mucho mas rápido! ( ahorras 0.6 ms)
COLAY vs CALL &XXXX	2.4 vs 2.0	Usar CALL como siempre es más rápido, pero menos legible.
GOSUB / RETURN	0.56	Aceptablemente rápido. La medida la he hecho con una rutina que solo hace return.
SETUPSP, id, param, valor	2.7	Aceptable, aunque POKE es mucho mejor para ciertos parámetros. Hay parámetros que se pueden establecer con POKE como el estado, pero otros no (como una ruta). Consulta la guía de referencia
FOR / NEXT	0.6	Lo puedes usar para recorrer varios enemigos y que cada uno se mueva de acuerdo a una misma regla. Debes valorar si puedes usar AUTOALL o MOVEALL para tus propósitos ya que en un solo comando moverías a todos los que quieras, lo cual es mucho mejor que un bucle.
COLSP,31, @c%	5.5	Tarda casi lo mismo con independencia del número de sprites activos. Evita llamar siempre con la variable de colision para acelerarlo a 4.3 ms
COLSP,31	4.3	Si tienes una nave o personaje y varios disparos es mucho mas eficiente que invoques a COLSPALL en lugar de invocar varias veces a COLSP
ANIMALL (este comando solo esta disponible con un parámetro desde PRINTSPALL, no se puede invocar directamente)	3.5	Es costoso pero hay una forma de invocarlo conjuntamente al invocar  PRINTSPALL , mediante un parámetro que hace que se invoque a esta función antes de imprimir los sprites. Ello permite ahorrar la capa del BASIC, es decir lo que consume enviar el comando, que es >1ms. Por ello podemos decir que este comando consumirá normalmente algo menos de 2ms
AUTOALL	2.76	No es costosa y puede mover a la vez los 32 sprites
MOVEALL,1,1	3.4	No es muy costosa y puede mover a la vez los 32 sprites
SOUND	10	El comando sound es “bloqueante” en cuanto se llena el buffer de 5 notas. Esto significa que tu lógica de BASIC no debe encadenar más de 5 comandos SOUND o se parará hasta que alguna nota termine..Si decides usarlo debe ser con sumo cuidado ya que consume mucho tiempo su ejecución (10 ms es muchísimo)
IF a>1 AND a>2 THEN a=2  Versus  IF a>1 THEN IF a>2 THEN a=2	2.52  Vs  2.39	Una sencilla forma de ahorrar 0.13 ms  En cada cosa que programes ten en cuenta estos detalles, cada ahorro es importante
A=RND*10	4.2	La función RND de BASIC es muy costosa. Puedes usarla, pero no en cada ciclo de juego sino solo eventualmente, por ejemplo, cuando aparezca un nuevo enemigo o cosas así. Otra solución sencilla es almacenar

		10 números aleatorios en un array y utilizarlos en lugar de invocar a RND
<b>Border &lt;x&gt;</b>	0.75	Bastante rápida. Útil para usarla en combinación con algún tipo de colisión de sprites, reforzando el efecto explosivo
<b>IF a AND 7 then 30</b>	1.19	He puesto el tiempo de ejecución cuando se cumple la condición. Ambos casos son bastante rápidos.
<b>IF A MOD 8 then 30</b>	1.29	
<b>ON x GOTO L1,L2,L3,L4</b>  <b>Vs</b>  <b>60 if x &gt;2 then 63</b> <b>61 if x=1 then 70</b> <b>62 goto 70</b> <b>63 if x=3 then 70</b> <b>64 goto 70</b>	3.67  Vs  4.8	Un comando ON GOTO es de media 1 ms más rápido que su equivalente con comandos “IF”, aunque también depende de la probabilidad de ocurrencia de cada uno de los 4 valores. Si la probabilidad de los 4 valores es la misma, podemos ahorrar 1ms usando ON GOTO Se puede optimizar aún más así  <b>10 ON X GOTO 30,40,50</b> <b>20 &lt;instrucciones caso x=4&gt;: GOTO 60</b> <b>30 &lt;instrucciones caso x=1&gt;: GOTO 60</b> <b>40 &lt;instrucciones caso x=2&gt;: GOTO 60</b> <b>50 &lt;instrucciones caso x=3&gt;: GOTO 60</b> <b>60 continuacion del programa</b>  Con esta estrategia bajamos a 3.54 ms

*Tabla 5 Relación de tiempos de ejecución de algunas instrucciones*

### Recomendaciones importantes:

- **Usar DEFINT A-Z al principio del programa.** El rendimiento mejorará muchísimo. Esto es casi obligatorio. Este comando borra las variables que existiesen antes y obliga a que todas las nuevas variables sean enteros a menos que se indique lo contrario con modificadores como “\$” o “!” (Consulta la guía de referencia de programador BASIC de Amstrad). Ojo, en cuanto uses DEFINT, si quieras asociar un número mayor que 32768 tendrás que hacerlo en hexadecimal.
- Si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible
- Cuando te falte velocidad y necesites un poquito más de rapidez utiliza **CALL <dirección>** en lugar de RSX. En caso de hacer esto, has de pasar los parámetros que contengan números negativos en formato hexadecimal.
- No sincronices el comando **|PRINTSPALL** con el barrido de pantalla a menos que tu juego funcione muy rápido. Sincronizar puede reducir tus FPS. En general con que consigas 12 FPS tu juego será “jugable”.

- **Eliminar espacios en blanco.** Cada espacio en blanco en tu listado BASIC consume 0.01ms en ejecución.
- **Acorta el nombre de tus variables.** Cuanto más largas son, más cuestan los accesos

operación	Tiempo
A=A+1	Una letra, tarda 1.18ms
HO=HO+1	Dos letras, tarda 1.2ms (un 2% más)
HOLAA=HOLAA+1	5 letras, tarda 1.25 ms (un 6% más)
HOLAAMIGOS=HOLAAMIGOS+1	10 letras 1.34 ms (un 13% más)

- **Reduce el número de variables.** Si hay muchas variables, los accesos de lectura y escritura son más lentos.
- Una vez que hayas invocado con parámetros al comando **|STARS** o al comando **|PRINTSPALL**, o a **|COLAY** o a otros comandos de 8BP, las siguientes veces no lo invoques con parámetros. La librería 8BP tiene “memoria” y usará los últimos parámetros que usaste. Esto ahorra milisegundos al atravesar la capa de análisis sintáctico del intérprete BASIC.
- Ten siempre en cuenta que una expresión diferente de cero es TRUE. Esto te permitirá ahorrar 0.5ms en cada IF y lo puedes usar en la lectura de teclado y en el control de variables

Mala opción	Buena opción (ahorras 0.5ms)
<b>IF x&lt;&gt;0 THEN &lt;instrucciones&gt;</b>	<b>IF x THEN &lt;instrucciones&gt;</b>
<b>IF x=20 THEN...</b>	<b>10 IF x-20 THEN 30 20 &lt;instrucciones&gt; 30</b>
<b>IF INKEY(34)=0 THEN &lt;instrucciones&gt;</b>	<b>10 IF INKEY(34) THEN 30 20 &lt;instrucciones&gt; 30</b>

- En juegos de naves donde no uses sobreescritura, procura que tu nave sea el sprite 31, de este modo pasará por “encima” de los sprites que simulan ser el fondo, pues tu nave se imprimirá después.
- Prueba versiones alternativas de una misma operación
 

```
A=A+1:IF A>4 then A=0 : REM esto consume 2.6ms
A=A MOD 3 +1 : REM esto consume 1.84 ms
A=1 + A AND 3 : REM esto consume 1.6 ms
```
- Evita el uso de coordenadas negativas. Ello te permitirá usar POKE para actualizar la posición de tu personaje. El comando POKE (el de BASIC) es muy veloz pero solo soporta números positivos, al igual que PEEK. En caso de usar coordenadas nativas, usa **|POKE** y **|PEEK** (comandos de 8BP). Reserva el uso de **|LOCATESP** para cuando vayas a modificar ambas coordenadas a la vez y puedan ser positivas y/o negativas. Recuerda también que un POKE de un valor x negativo se puede hacer usando POKE dirección, 255+x+1. En caso de que quieras usar coordenadas negativas para que se vea como poco a poco los enemigos entran a la pantalla por el lateral izquierdo (que se perciba el clipping), puedes evitar las coordenadas

negativas usando un **|SETLIMITS** y de esa manera producir el mismo efecto con coordenadas que comienzan en cero y una pantalla de juego ligeramente mas pequeña

- Si necesitas comprobar algo, no lo hagas en todos los ciclos de juego. A lo mejor basta que compruebes ese "algo" cada 2 o 3 ciclos, sin ser necesario que lo compruebes en cada ciclo. Para poder elegir cuando ejecutar algo, haz uso de la "aritmética modular". En BASIC dispones de la instrucción MOD que es una excelente herramienta. Por ejemplo, para ejecutar una de cada 5 veces puedes hacer: **IF ciclo MOD 5 = 0 THEN ...** aunque es mejor que uses operaciones **AND** que operaciones **MOD**
- Haz uso de las "**secuencias de muerte**". Ello te permitirá ahorrar instrucciones para comprobar si un sprite que está explotando ha llegado a su último fotograma de animación para desactivarlo.
- La sobreescritura es costosa: si puedes hacer tu juego sin sobreescritura ahorrarás milisegundos y ganaras colorido. Úsala cuando la necesites, pero no sin motivo
- Las macrosecuencias de animación te ahorran líneas de BASIC ya que no necesitas chequear la dirección de movimiento del sprite. Úsalas siempre que puedas.

## **11.2 Haz una sola lógica para gobernar todas tus pantallas**

Nuestro Amstrad solo tiene 64KB y si descontamos la memoria de video, la memoria para tus sprites, tu música y la librería 8BP, te quedan 24KB de BASIC que debes usar muy bien. Si cada pantalla tiene su propio "programa" dentro de tu juego, apenas podrás hacer un juego de 10 pantallas.

Hay dos cosas que debes tratar de hacer para reducir el uso de la memoria

- Crear pantallas con pocos bytes
- Crear una sola lógica que gobierne todas las pantallas, es decir que un mismo ciclo de juego es el que se va a ejecutar en todas las pantallas del juego.

En los juegos de pasar pantallas que usan layout, cada pantalla puede ocupar 20x25 bytes, es decir 500 bytes. Si empleas ciertos "trucos" como los explicados en el capítulo 8, puedes reducir esta memoria. En el videojuego "**Happy Monty**" se construyen 25 pantallas con solo 160 bytes cada una y hay una única lógica de ciclo de juego para todas las pantallas.

**Es muy importante que programes una única lógica de ciclo de juego y la apliques a todas las pantallas.** Si programas una lógica de ciclo de juego para cada pantalla, el código fuente de tu programa será enorme y podrás programar pocas pantallas pues se te acabará la memoria pronto.

También puedes superar las limitaciones de memoria mediante algoritmos que generen laberintos, o pantallas sin necesidad de almacenarlos. De este modo podrás hacer muchas más pantallas. Esto requiere creatividad, desde luego, pero es posible. Un algoritmo

siempre ocupa menos que los datos que genera, aunque lógicamente cuesta más tiempo ejecutarlo que simplemente leer datos almacenados.

Puedes reutilizar la lógica de enemigos de una pantalla en otra, ahorrando líneas de código. Aprovecha el mecanismo GOSUB/RETURN para ello. También es muy útil usar rutas en los enemigos. **Con el mecanismo de rutas, el enemigo se mueve sin ejecutar lógica BASIC** y funcionará muy rápido. Basta con asignar una ruta a un enemigo para que este la recorra una y otra vez sin que necesitemos costosas instrucciones IF, asignaciones, etc.

También puedes hacer juegos que carguen por fases, de modo que no tengas todo el juego en memoria a la vez. Esto es un poco molesto para el usuario de cinta (CPC464) aunque no lo es para el de disco (CPC6128)

Usa **sprite flipping** para ahorrar memoria en tus sprites

### **11.3 Técnica de “Lógicas masivas”**

A menudo vas a necesitar mover muchos sprites, sobre todo en juegos de arcade del espacio o de estilo “commando” (el clásico de Capcom de 1985).

Podrías actuar por separado en las coordenadas de todos los sprites y actualizarlas usando POKE, pero resultaría muy lento, inviable si quieras fluidez de movimientos. Lo más recomendable (y sencillo) es hacer uso combinado de las funciones de movimiento automático y de movimiento relativo, que son |AUTOALL y |MOVEALL respectivamente.

La clave de lograr velocidad en muchos sprites es utilizar la técnica que he bautizado como “lógicas masivas”. Esta técnica consiste fundamentalmente en ejecutar menos lógica en cada ciclo de juego (lo que se denomina “reducir la complejidad computacional”) y para ello hay varias opciones:

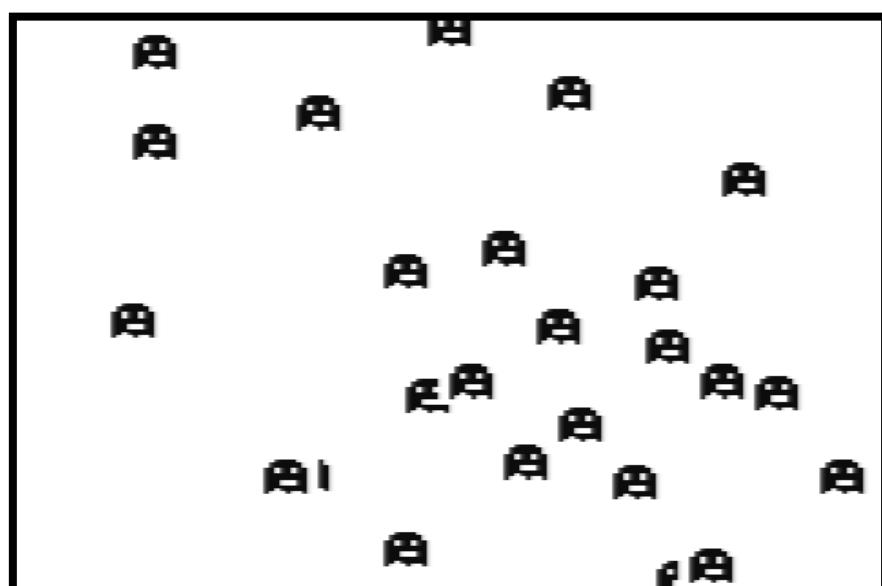
- Usar **una sola lógica** que afecta a muchos sprites a la vez (usando los flag de movimiento automático y/o relativo)
- Ejecutar varias tareas, pero solo una de ellas o unas pocas en cada ciclo del juego, usando **aritmética modular (u operaciones binarias) en cascada**.
- Introducir **limitaciones en el juego que no sean importantes** o no afecten a la jugabilidad, para reducir el número de tareas que se ejecutan en cada ciclo de juego o simplificar las tareas de modo que se ejecuten más rápido.
- Como norma general, reduce el número de instrucciones por las que pasa tu programa en cada ciclo de juego, reemplazando a veces algoritmos por precálculos o poniendo más instrucciones para lograr (paradójicamente) que se ejecuten menos cada ciclo.

Estas ideas tienen un mismo objetivo: **ejecutar menos lógica en cada ciclo**, permitiendo que todos los sprites se muevan a la vez, pero tomando menos decisiones en cada ciclo del juego. A esto se le llama “**reducir la complejidad computacional**”, transformando **un problema de orden N (N sprites) en un problema de orden 1 (una sola lógica a ejecutar en cada fotograma)**.

La clave está en determinar qué lógica o lógicas ejecutar en cada ciclo. En el caso más sencillo, si tenemos N sprites simplemente ejecutaremos una de las N lógicas. Pero en casos más complejos deberemos ser astutos para determinar que lógicas conviene ejecutar.

### 11.3.1 Mueve 32 sprites con lógicas masivas

Ahora vamos a ver un sencillo ejemplo para mover 32 sprites simultáneamente y suavemente (a 14fps). Es perfectamente posible. Solo un fantasma va a tomar decisiones en cada ciclo, aunque se van a mover todos los fantasmas en todos los ciclos. También podemos animar a todos (asociándoles una secuencia de animación y usando |PRINTSPALL,1,0 ) y seguirá quedando suave, pero aun parecerá que hay mayor movimiento pues el aleteo de las alas de una mosca (por ejemplo) genera mucha sensación de movimiento



*Fig. 65 con lógicas masivas puedes mover 32 sprites simultáneamente*

Lo que hemos hecho ha sido reducir la complejidad computacional. Hemos partido de un problema de “orden N”, siendo N el número de sprites. Suponiendo que cada lógica de sprite requiera 3 instrucciones BASIC, en principio habría que ejecutar  $N \times 3$  instrucciones en cada ciclo. Con la técnica de “lógicas masivas”, transformamos el problema de “orden N” en un problema de “orden 1”. Se llama problemas de “orden 1” a los que involucran un número constante de operaciones independientemente del tamaño del problema. En este caso hemos pasado de  $N \times 3 = 32 \times 3 = 96$  operaciones BASIC a sólo 3 operaciones BASIC. Esta reducción de complejidad es la clave del alto rendimiento de la técnica de lógicas masivas.

```

1 MODE 0
10 MEMORY 24999: CALL &6B78
20 DEFINT a-z
25 ' reset enemigos
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num enemigos de 12 x 16 (6bytes de ancho x 16 lineas)
36 num=32: x%=0:y%=0
40 FOR i=0 TO num-1:|SETUPSP,i,9,&8ee2: |SETUPSP,i,0,&X1111:
41 |LOCATESP,i,rnd*200,rnd*80
42 next

```

```

43 i=0
45 gosub 100
46 i=i+1: if i=num then i=0
50 |PRINTSPALL,0,0
60 |AUTOALL
70 goto 45

100 |peek,27001+i*16,@y%
110 |peek,27003+i*16,@x%
120 if y%<=0 then |SETUPSP,i,5,2:|SETUPSP,i,6,0: return
130 if y%>=190 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0: return
140 if x%<=0 then |SETUPSP,i,5,0:|SETUPSP,i,6,1: return
150 if x%>=76 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1: return

160 azar=rnd*3
170 if azar=0 then |SETUPSP,i,5,2:|SETUPSP,i,6,0:return
180 if azar=1 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0:return
190 if azar=2 then |SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if azar=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1:return

```

### 11.3.2 Ejecución alternada y periódica en cascada

No es necesario que ejecutes todas las tareas en cada ciclo de juego. Por ejemplo, si quieras comprobar si el disparo ha salido de la pantalla, puedes comprobarlo cada dos o 3 ciclos, en lugar de comprobarlo en cada ciclo.

También puedes hacer que los enemigos disparen cada cierto número de ciclos y no cada ciclo (¡de lo contrario te dispararían muchísimo!!)

En definitiva, hay cosas que no necesitas hacer en cada ciclo y puedes ahorrar ejecución de instrucciones por ciclo y por lo tanto lograrás mayor velocidad. Esto es en realidad el fundamento básico de la técnica de “lógicas masivas”.

Hay dos técnicas básicas para esto: El uso de aritmética modular y las operaciones binarias **AND**. Las operaciones binarias **AND** son más rápidas que las **MOD**

Técnica	Tiempo consumido
A = A+1: if A =5 then A=0: GOSUB <rutina>	2.6 ms
IF ciclo MOD 5 =0 THEN gosub rutina	1.84ms, suponiendo que ya tienes una variable llamada ciclo que se actualiza

La operación MOD es algo costosa y por ello a veces es mejor una operación binaria.

Suponiendo que tienes la variable ciclo que se actualiza cada vez, podemos hacer una operación binaria para ver cuando un grupo de bits da un determinado valor. Por ejemplo, si observamos los 4 bits menos significativos de la variable ciclo siempre van a ir desde 0000 hasta 1111 y vuelta a empezar. Pues bien, si hacemos un AND 15 con dicha variable podremos hacer lo mismo que con MOD 15. El número 15 en binario es 1111 y por eso un AND nos revela el valor de esos 4 bits.

Técnica	Tiempo consumido
If ciclo AND 15=0 then gosub rutina	1.6 ms (se ejecuta una de cada 16 veces)

If ciclo AND 1=0 then gosub rutina	1.6ms (Se ejecuta una de cada 2 veces)
------------------------------------	--

Si tienes varias cosas periódicas a ejecutar puedes hacerlo así:

```
c=ciclo AND 15 :' rem 15 es en binario 1111
IF c=0 THEN GOSUB <rutina1> (se ejecuta "rutina1" una de cada 16 veces)
IF c=8 THEN GOSUB <rutina2>... ( rutina2 se ejecuta una vez cada 16 veces,
pero alejado en el tiempo de la ejecución de la rutina1)
```

De esta forma estás repartiendo el tiempo en distintas tareas, de forma que en cada ciclo sólo haces una tarea, pero al cabo de varios ciclos has hecho todas las tareas.

Para comprobar la variable ciclo y decidir ejecutar una tarea hay una forma de ejecutar las operaciones binarias mejor y es la siguiente:

Técnica	Tiempo consumido
<pre>10 If ciclo and 7 then 30 20 &lt;instrucciones que se ejecutan cada 8 ciclos&gt; 30 &lt; continuación del programa&gt;</pre>	<b>1.18 ms.</b> Esta es sin lugar a dudas la mejor estrategia para la ejecución periódica de tareas

Aplicando la misma estrategia a MOD logramos también aumentar la velocidad, aunque menos que con AND. Sin embargo, es muy buena por que funciona, aunque el periodo no sea múltiplo de 2 (puedes poner MOD 7, MOD 5, MOD 10, etc.)

Técnica	Tiempo consumido
<pre>10 If ciclo MOD 8 then 30 20 &lt;instrucciones que se ejecutan cada 8 ciclos&gt; 30 &lt; continuación del programa&gt;</pre>	<b>1.29 ms.</b> Casi tan bueno como AND y la mejor estrategia cuando necesitamos un periodo que no es múltiplo de 2

Como ves, por cada tarea que queramos introducir en la lógica, deberemos introducir un “IF”. **Sin embargo, aún se puede mejorar** y hacerlo más eficiente usando intervalos de tiempo de ejecución de tareas que sean múltiplos. Si son múltiplos, **el “IF” de la tarea 2 se puede hacer dentro de la tarea 1, y el de la tarea 3 dentro de la tarea 2, en “cascada”**. Esto reduce enormemente el número de IF que ejecutamos en cada ciclo, siendo en muchos casos de un solo IF.

Vamos a ver un ejemplo completo, que te permite multitarea de 4 tareas diferentes, pero reduciendo el número de tareas que se ejecutan a la vez. La siguiente secuencia representa el orden de ejecución de las tareas y a continuación el código fuente.



```
10 IF ciclo AND 1 THEN 90
20 REM cada dos ciclos entramos aquí
25 IF ciclo AND 3 THEN 80
30 REM cada 4 ciclos entramos aquí
35 IF ciclo AND 7 THEN 70
40 REM cada 8 ciclos entramos aquí
50 <tarea 4> : GOTO 100
70 <tarea 3> : GOTO 100
```

```

80 <tarea 2> : GOTO 100
90 <tarea 1>
100 REM --- fin de tareas ---

```

En este ejemplo hemos elegido los intervalos 2,4 y 8  
AND 1 : esto me da un intervalo de 2 porque es cero cada 2 ciclos  
AND 3 : es cero cada 4 ciclos  
AND 7 : es cero cada 8 ciclos

Gracias a que hemos elegido operaciones en intervalos multiplo, los IF se ejecutan “en cascada”: solo entramos en un IF si hemos entrado en el anterior:

- La mitad de los ciclos ejecutan un único IF (línea 10)
- La mitad de los ciclos ejecutan dos sentencias IF (líneas 10 y 25), de los cuales, la mitad (es decir un 25%) ejecutarán tres IF (líneas 10, 25 y 35)

En media se ejecutan  $1*50\%+2*25\%+3*25\% = 1.75$  sentencias IF por ciclo

Gracias a esta estrategia de **usar aritmética modular con operaciones binarias en intervalos múltiplos para hacerlas en cascada**, podemos reducir el número de operaciones “IF” al mínimo y a la vez reducimos la complejidad computacional de orden N (n tareas) a orden 1 (una sola tarea por ciclo). Esto acelera muchísimo tus juegos.

### 11.3.3 Ejemplo sencillo de lógica masiva

En el videojuego “Mutante Montoya”, los sprites enemigos se turnan para ejecutarse en los distintos ciclos del juego. **Cuando programé este juego aún no había programado el mecanismo |ROUTEALL que permite asignar trayectorias fijas a los sprites, pero pude solventarlo con lógicas masivas.** En caso de quisieses hacer un juego donde los enemigos tuviesen “inteligencia”, no serviría una ruta fija, de modo que aunque tengamos el comando ROUTEALL, tendríamos que turnar la lógica de los sprites tal como se describe a continuación, por eso este ejemplo es interesante.

Supongamos que tenemos 3 soldados enemigos que se mueven de derecha a izquierda y de izquierda a derecha. Para ganar velocidad vamos a ejecutar sólo la lógica de un soldado en cada ciclo de juego.

Para que, a pesar de ello, la coordenada x de cada soldado siga avanzando, usaremos el flag de movimiento automático, en lugar de actualizarla nosotros.

```

10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
26 |SETLIMITS,0,80,0,200
30 'parametrizacion de 3 soldados
40 dim x(3):x%=0
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:direccion(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x1111:
|SETUPSP,1,5,0: |SETUPSP,1,6,1
60 x(2)=20:xmin(1)=15:xmax(2)=40:
y(2)=100:direccion(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x1111:
|SETUPSP,2,5,0: |SETUPSP,2,6,-1

```

```

70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x1111:
|SETUPSP,3,5,0: |SETUPSP,3,6,1
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next: 'colocamos los sprites
81 i=0
89 '----- BUCLE PRINCIPAL DEL JUEGO (CICLO DE JUEGO) -----
90 i=i+1:gosub 100
92 if i=3 then i=0
93 |AUTOALL
94 |PRINTSPALL,1,0: ' anima e imprime los 3 soldados
95 goto 90
96 '----- FIN DEL CICLO DE JUEGO -----
99 '----- rutina de soldado -----
100 |PEEK,27003+i*16,@x%: x(i)=x%
101 IF direccion(i)=0 THEN IF x(i)>=xmax(i) THEN
direccion(i)=1:|SETUPSP,i,7,10: |SETUPSP,i,6,-1 ELSE return
110 IF x(i)<=xmin(i) THEN direccion(i)=0:|SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 return

```

Cada soldado tiene su propia lógica, pero solo ejecutamos una en cada ciclo de juego, aligerando muchísimo el ciclo de juego.

La única limitación es que al ejecutar la lógica de cada soldado una de cada 3 veces, la coordenada podría sobrepasar el límite que hemos establecido durante dos ciclos. Eso hace que debamos ser más cuidadosos al fijar el límite, asegurándonos al ejecutarlo que nunca invade y borra un muro de nuestro laberinto de pantalla, por ejemplo. Voy a tratar de explicar este problema con más precisión:

Supongamos que tenemos 8 sprites y nuestro sprite se mueve en todos los ciclos, pero sólo ejecutamos su lógica una de cada 8 veces. Imagínate un sprite que está en la posición x=20 y queremos que se mueva hasta la posición x=30 y dar la vuelta. Consideremos que el sprite tiene un movimiento automático con Vx=1. En ese caso comprobaremos su posición cuando x=20, x=28, x=36. ¡Al llegar a 36 nos daremos cuenta de que nos hemos pasado!!! y cambiaremos la velocidad del sprite a Vx=-1

Como ves el control de los límites de la trayectoria no es preciso, a menos que tengamos en cuenta esta circunstancia y fijemos el límite en algo que podamos controlar, que será Xfinal = Xinicial + n\*8.

Esta limitación es minúscula si la comparamos con la ventaja de mover muchos sprites a gran velocidad. Con algo de astucia podemos incluso ejecutar la lógica menos veces, de modo que solo uno de cada dos ciclos se ejecute algún tipo de lógica de enemigos.

#### 11.3.4      Movimiento “en bloque” de escuadrones

Si lo que quieras es simplemente mover a la vez un escuadrón en una dirección, cualquiera de las dos funciones siguientes de la librería 8BP te servirán:

- Si usas **|AUTOALL** tienes que ponerle velocidad automática a los sprites en la dirección que quieras (en Vx, en Vy o en ambas) y por supuesto activar el bit 4 del byte de estado. El comando AUTOALL tiene un parámetro opcional para invocar internamente a **|ROUTEALL** antes de mover a los sprites

- Si usas **|MOVEALL** tienes que activar el bit 5 del byte de estado a los sprites que vayas a mover. Este comando requiere como parámetros cuento movimiento relativo en Y y en X deseas

De esta forma con una sola instrucción estás moviendo muchos sprites a la vez. En caso de que cada uno de tus sprites se deba mover de forma independiente y con una lógica independiente como ocurre en juegos tipo “pacman”, habrá que ser más astutos, como te contaré a continuación.

### **11.3.5 Técnica de lógicas masivas en juegos tipo “pacman”**

Si tienes muchos enemigos y deben tomar decisiones en cada bifurcación de un laberinto, no es una buena estrategia simplemente turnar la ejecución de lógicas de enemigos en cada ciclo. Lo que conviene es ejecutar la lógica cuando dicha lógica deba tomar una decisión. En juegos tipo pacman esto ocurre cuando un fantasma llega a una bifurcación en la que puede tomar una nueva dirección de movimiento en función de su inteligencia. Esto se puede llevar a cabo con un sencillo “truco”. Es simplemente colocar a los enemigos en posiciones bien escogidas al empezar el juego.

Supongamos que tienes 4 enemigos y que las bifurcaciones del laberinto ocurren en múltiplos de 4. Si el primer enemigo está en una posición múltiplo de 4 le tocará ejecutar su lógica. Al segundo enemigo le toca ejecutar su lógica de decisión en el ciclo siguiente. Si no se encuentra en una posición de bifurcación del laberinto no podrá cambiar su rumbo

Para que “encaje” su posición con un múltiplo de 4 y así poder decidir qué camino tomar en la bifurcación, simplemente empezamos el juego con este segundo enemigo colocado en un múltiplo de 4 menos uno. Considerando coordenadas que comienzan en cero, los múltiplos de 4 son:

Primer enemigo: posición 0 o 4 o 8 o 12 o 16 o 20 o 24 o XX (en eje x o y, da igual)

Segundo enemigo: posición 1 o 5 o 9 ...

Tercer enemigo: posición 2 o 3 o 10 ...

Y así sucesivamente. Colocas a tus enemigos siguiendo esta regla:

**Posición = múltiplo de 4 - n, siendo n el número de sprite**

Y cada vez que le toque a un enemigo ejecutar su lógica, podrá encontrarse en una bifurcación. Si un fantasma debe tomar una decisión en el instante “i”, posteriormente lo hará en  $t=i+4$  y la siguiente será en  $t=i+4+4$ , etc.

Vamos a ver un ejemplo gráfico en el que he destacado con un rectángulo que enemigo va a tomar una decisión por encontrarse en bifurcación. Como puedes comprobar solo se ejecuta una lógica de bifurcación en cada ciclo de juego

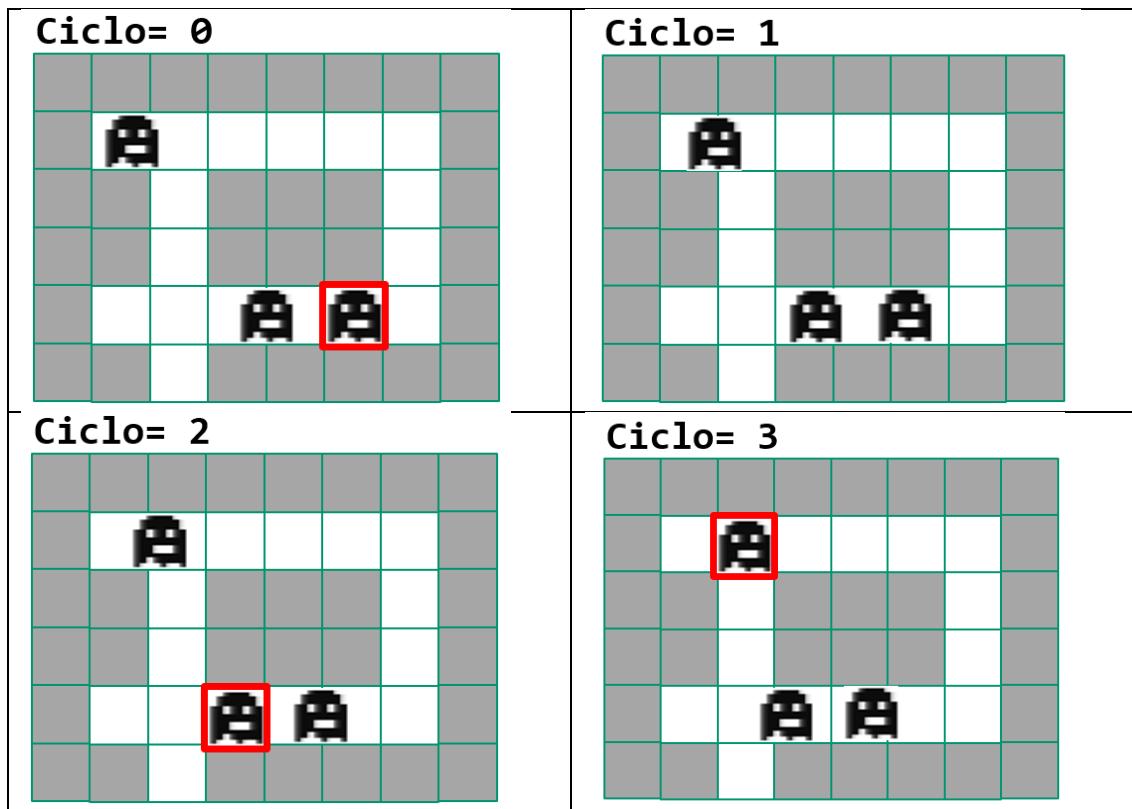


Fig. 66 Lógicas masivas en juegos tipo PACMAN

Cuando le toque ejecutar su lógica deberá comprobar que no se encuentra en una posición en mitad de un pasillo sin bifurcaciones. Debe comprobarlo y para ello puede usar el comando |COLAY.

El videojuego “Paco, el hombre” usa esta técnica. De hecho, ejecuta la lógica de 4 fantasmas, cada uno en un ciclo de juego diferente, además de repartir el resto de tareas entre los distintos ciclos. Es un ejemplo magnífico de lógicas masivas por lo que te recomiendo que leas el “making of” del videojuego, que encontrarás en la documentación de 8BP

	ciclos			
Tarea	0	1	2	3
Lectura teclado	X			
Detección colisión con fantasmas y con sprites invisibles		X		
Detección colisión con layout y recolocación si Paco choca			X	
Detección de puntos (cocos)				X
Lógica fantasma 1	X			
Lógica fantasma 2		X		
Lógica fantasma 3			X	
Lógica fantasma 4				X

El videojuego “Paco el hombre” reparte las tareas en 4 ciclos de juego, permitiendo alcanzar casi 20 FPS en BASIC

### 11.3.6 Reducción de número de instrucciones en ciclo de juego

Reducir el número de instrucciones por las que pasa el ciclo de juego es fundamental para acelerar tu programa. A veces ello implica que el programa tenga más líneas, aunque al final se ejecuten menos líneas en cada ciclo. Otras veces puedes introducir limitaciones “indetectables” que aceleran tu juego sin que el jugador perciba la diferencia. Vamos a ver algunos ejemplos

#### 11.3.6.1 Gestión de teclado con menos instrucciones

Gestionar el teclado (y en general esto es aplicable a cualquier cosa que hagas) ejecutando el menor número de instrucciones. Aquí tienes un ejemplo (primero mal hecho y luego bien hecho), donde como mucho se pasa por 4 operaciones INKEYS con sus correspondientes IF. Ejecútalo mentalmente y comprobarás lo que digo. Es mucho más rápida la segunda

Ejemplo mal hecho (caso peor = 8 ejecuciones “IF INKEY”):

```
1000 rem rutina de teclado ineficiente
1010 IF INKEY(27)=0 and INKEY(67)=0 THEN <instrucciones>:RETURN
1020 IF INKEY(27)=0 and INKEY(69)=0 THEN <instrucciones>:RETURN
1030 IF INKEY(34)=0 and INKEY(67)=0 THEN <instrucciones>:RETURN
1040 IF INKEY(34)=0 and INKEY(69)=0 THEN <instrucciones>:RETURN
1050 IF INKEY(27)=0 THEN <instrucciones>:RETURN
1060 IF INKEY(34)=0 THEN <instrucciones>:RETURN
1070 IF INKEY(67)=0 THEN <instrucciones>:RETURN
1080 IF INKEY(69)=0 THEN <instrucciones>:RETURN
```

A continuación, el ejemplo de lectura de teclado, pero bien hecho (con caso peor = 4 ejecuciones “IF INKEY”). Además, se ha tenido en cuenta que, en un IF, las expresiones distintas de cero son TRUE. Las instrucciones a ejecutar en tu juego pueden ser diferentes pero el esquema de lectura de teclado debería ser el mismo en caso de manejar diagonales (arriba y derecha a la vez, por ejemplo). Puede parecer más largo, pero es mucho más rápido que el ejemplo anterior.

```
REM rutina del teclado eficiente
'saltar a 1550 si no se ha pulsado "P"
1510 if inkey(27) THEN 1550: 'tecla P
1520 if inkey(67) THEN 1530: 'tecla Q

1525 <instrucciones en caso de haber pulsado "P" y "Q" a la
vez>:RETURN
1530 if inkey(69) THEN 1540:'tecla A

1535 <instrucciones en caso de haber pulsado "P" y "A" a la
vez>:RETURN

1540 <instrucciones en caso de haber pulsado "P" solamente>:RETURN

1550 if inkey(34) THEN 1590:'tecla O
```

```

1560 if INKEY(67) THEN 1570: 'tecla Q
1565 <instrucciones en caso de haber pulsado "O" y "Q" a la vez>:RETURN
1570 if INKEY(69) THEN 1580: 'tecla A
1575 <instrucciones en caso de haber pulsado "O" y "A" a la vez>:RETURN
1580 <instrucciones en caso de haber pulsado "O" solamente>:RETURN
1590 IF INKEY(67) THEN 1600: 'tecla Q
1595 <instrucciones si se ha pulsado "Q">: RETURN
1600 IF INKEY(69) THEN return: 'tecla A
1610 <instrucciones si se ha pulsado "A" solamente>: RETURN

```

Otra cosa que debes hacer para acelerar tu juego es usar una tarea periódica para explorar teclas “secundarias” tales como teclas para activar/desactivar la música, teclas para pasar a un menú o visualizar algo especial, etc. Son teclas que puedes explorar periódicamente y no cada ciclo. Eso sí, debes tener en cuenta lo que cuesta explorarlas, que no es mucho (1 ms)

```

10 if inkey(47) then 30: ' esto cuesta 1.0 ms
20 <instrucciones si pulsas la tecla 47>
30 rem llegas aquí si no la has pulsado

```

Analizar una variable con AND para que ocurra una tarea cada (por ejemplo) 4 ciclos cuesta 1.18 ms, por lo tanto nos va a costar

1.18 ms x 4 ciclos (la evaluación del ciclo) + 1.0ms ( el **inkey**) =5.72 ms  
 si en lugar de hacer eso ejecutamos el **inkey** en cada ciclo, habríamos gastado solo 4ms por lo tanto, explorar ciertas teclas basándonos en el ciclo de juego solo tiene sentido si al menos vamos a evitar explorar en algunos ciclos dos teclas

Supongamos el siguiente programa:

```

10 if ciclo and 3 then 50: ' esto cuesta 1.18 ms
20 if inkey(47) ... ' esto cuesta 1 ms
30 if inkey(35) ... ' esto cuesta 1 ms
50 <mas instrucciones>

```

Tal y como está hecho, el programa evalúa las teclas 47 y 35 uno de cada 4 ciclos. Cuando las evalúa gasta  $1.18 + 1 + 1 = 3.8$  ms mientras que cuando no las evalúa gasta 1.18 ms. Por lo tanto, el tiempo gastado en 4 ciclos es

Tiempo  $3 * 1.18 + 3.8 = 6.72$  ms

Mientras que si hubiésemos evaluado las teclas todos los ciclos habríamos gastado  $4 * 2$  ms= 8 ms. Por consiguiente, hay un ahorro de  $8 - 6.7 = 1.3$  ms por cada 4 ciclos, o lo que es lo mismo, aproximadamente 0.3 ms por ciclo de juego.

Una de las opciones que tienes, dependiendo del tipo de juego, es explorar unas teclas en los ciclos pares y las otras en los ciclos impares. Por ejemplo, en un juego que use las teclas QAOP, puedes explorar QA en los ciclos pares y OP en los impares o viceversa.

De esa forma puedes obtener mayor velocidad con una limitación que el jugador seguramente no perciba. Este es el tipo de limitaciones que a veces merece la pena introducir, pero depende de cada juego.

### 11.3.6.2 Evitar pasar por IF innecesarios

Vamos a ver dos formas de hacer esto:

```
10 IF A=1 THEN < instrucciones cuando A=1>
20 IF A=2 THEN < instrucciones cuando A=2>
30 IF A=3 THEN < instrucciones cuando A=3>
40 <mas instrucciones >
```

Si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible. El GOTO es un gran aliado de la técnica de lógicas masivas.

```
10 IF A=1 THEN < instrucciones cuando A=1> : GOTO 40
20 IF A=2 THEN < instrucciones cuando A=2> : GOTO 40
30 < instrucciones cuando A=3> : rem si A no es 1 ni 2 entonces es 3
40 <mas instrucciones>
```

Otra forma de hacer esto es mediante la instrucción ON <variable> GOTO o también la instrucción ON <variable> GOSUB

Tal como he presentado en la tabla de 11.1, puedes ahorrar mas de 1ms usando ON GOTO

```
10 on A GOTO 30,40,50
20 goto 60: rem llegamos aquí si A=4
30 rem llegamos aquí si A=1
35 goto 60
40 rem llegamos aquí si A=2
45 goto 60
50 rem llegamos aquí si A=3
60 rem aquí continua la logica
```

### 11.3.6.3 Reemplaza algoritmos por precálculos

Piensa en una pelota que bota. En lugar de usar las ecuaciones del movimiento acelerado, construye una ruta que mueve un sprite hacia abajo con un incremento de coordenada "Y" mayor en cada paso y al tocar el suelo sube con un incremento cada vez menor. No hay ecuaciones complejas que ejecutar y sin embargo el efecto visual es el mismo. Así fue como hice los saltos del juego "**Fresh fruits & vegetables**". Esto es posible programarlo así porque **dentro del juego el universo es "determinista". Es decir, se puede predecir en cada instante de tiempo que va a ocurrir**, por muy complejas que sean las ecuaciones que gobiernan un salto de un personaje o el movimiento de un escuadrón.

En juegos donde la lógica de los enemigos requiere del uso del cálculo de alguna función (como el coseno), precalcula todo y guárdalo en un array que uses durante la ejecución de la lógica. Calcular durante la lógica del juego tiene un coste prohibitivo.

Una lógica compleja es una lógica lenta. Si quieras hacer algo complicado, una trayectoria compleja, un mecanismo de inteligencia artificial...no lo hagas, trata de "simularlo" con un modelo de comportamiento más sencillo pero que produzca el mismo efecto visual. Por ejemplo, un fantasma que es inteligente y te persigue, en lugar de que tome decisiones inteligentes, haz que trate de tomar la misma dirección que tu personaje, sin ninguna lógica. Si no puedes simplificar de este modo, entonces piensa que incluso la inteligencia artificial se puede transformar en "determinista". Si un enemigo toma una decisión sobre como moverse en función de tu posición y tu velocidad, podríamos almacenar el resultado de ese pesado algoritmo un array y evitar todos los cálculos.

```

10 Rem Vx, Vy, X, Y son la velocidad y posición de mi personaje
20 rem supongamos que tengo precalculadas las decisiones para 3
velocidades, 10 franjas de coordenadas X y 10 franjas en Y. Esto es
menos de 1KB
30 DIM perseguir(3,3,10,100)
40 rem ' cargaría los valores en el array tras calcularlos despacio
50 nuevadireccion=perseguir(Vx,Vy,x,y): rem mecanismo en acción

```

El universo que estas programando es "determinista". Por muy complejo que sea el comportamiento de enemigos y elementos de la pantalla, si su comportamiento no depende de tu interacción, entonces hay una posición para cada cosa determinada en cada instante de tiempo. Una posición que podría precalcularse para evitar cualquier algoritmo complejo de comportamiento y el resultado sería el mismo.

#### 11.3.6.4 No ejecutar comentarios

Eliminar cualquier comentario en la lógica de juego y si dejas alguno que sea REM (mas rápido), no uses la comilla. Si usas la comilla es para ahorrar 2 bytes de memoria, y es adecuado para comentar el resto del programa (inicializaciones y cosas así). Si quieres comentar partes de lógica puedes hacer lo siguiente:

```

If x>23 gosub 500
...
499 rem por esta linea no se pasa y asi comento esta rutina
500 if x > 50 THEN ...
...
550 RETURN

```

Cada comentario que ejecutes consume 0.20 ms y ahorrar su ejecución es muy fácil sin por ello dejar de poner comentarios. Hay ocasiones en las que puedes poner comentarios en líneas siempre que haya saltos (GOTO y GOSUB/RETURN) sin miedo a gastar nada de tiempo, veamos algunos ejemplos:

```

10 goto 50 : rem este comentario no consume tiempo
20 gosub 200: rem este comentario no consume tiempo
200 return: rem este comentario no consume tiempo

```

#### 11.3.6.5 Sólo muere un sprite en cada ciclo

El comando |COLSPALL de 8BP está pensado con "lógica masiva". Esto significa que potencialmente te detecta la colisión de todos los sprites, pero en cuanto detecta una colisión te retorna indicando quien es el sprite colisionador y quien el colisionado. En ese momento puedes anular al sprite colisionado (desactivando su capacidad de ser colisionado en el bit 1 de su byte de estado) y volver a lanzar el comando, hasta que no haya más colisiones (te retorno un 32 en el colisionador y el colisionado). Sin embargo,

lo más eficiente es no volver a lanzar el comando hasta el siguiente ciclo de juego. Esto significa que solo un enemigo podrá morir en cada fotograma, pero es una limitación imperceptible que acelerará mucho tus juegos.

### 11.3.7 Rutas que aceleran el juego manipulando el estado

Puedes hacer rutas que activen y desactiven alternativamente el flag de colisionador o de colisionable en tus sprites. Dependiendo del tipo de enemigo, esto te puede proporcionar velocidad extra en el comando de colisión. Las rutas se explican en un capítulo posterior de modo que antes de leer esto conviene que te familiarices con ellas.

Por ejemplo, si tienes 8 enemigos en pantalla, puedes hacer que en los ciclos pares haya 4 colisionables y en los ciclos impares los otros 4. Para ello puedes hacer una ruta como la siguiente:

```
ROUTE1;
;-----
    db 1,0,1
    db 255,128+8+2+1,0 ; cambio de estado a colisionable
    db 1,0,1
    db 255,128+8+1,0 ; cambio de estado a no colisionable
    db 0
```

Esta ruta cambia el estado de tu sprite, a la vez que lo mueve hacia la derecha. Si un sprite tiene asignada esta ruta, se moverá hacia la derecha y alternativamente será colisionable y no colisionable.

Puedes asignar la ruta a 8 sprites y después ejecutar el siguiente comando sobre 4 de los sprites:

```
|ROUTE$P, <sprite_id>, 1
```

Con ello, el comando de colisión |COLSPALL será más rápido pues solo tendrá que detectar colisiones con 4 enemigos colisionables en cada fotograma. Este truco acelera un poco tu juego y unido a otros trucos puedes alcanzar finalmente los FPS que necesites

El mismo truco te permite activar y desactivar el flag de impresión (bit 0 del estado) y en una ruta donde el enemigo pase algún fotograma sin moverse, puedes desactivar el flag de impresión a través de la ruta y así acelerar el comando PRINTSPALL.

### 11.3.8 Enrutando sprites con “lógicas masivas”

Para mover sprites a través de trayectorias existe un comando llamado |ROUTEALL que lo hace muy eficientemente, pero **como ejercicio para comprender la filosofía de lógicas masivas es muy interesante estudiar este difícil pero emblemático caso de lógicas masivas**. Para programar el videojuego “*Anunnaki*” empleé la técnica que voy a describir a continuación, ya que aún no había programado la capacidad de enrutamiento de sprites en 8BP. Básicamente empleé lógicas masivas en el enrutamiento de las naves enemigas.

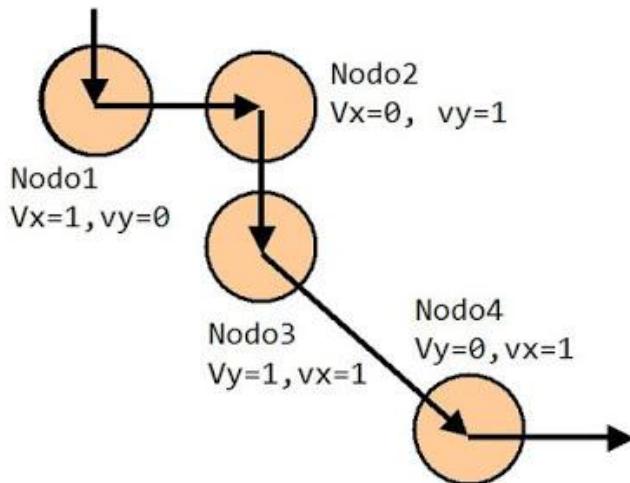
Comencemos imaginando una trayectoria para una hilera de 8 naves enemigas. las naves pasarán una a una por una serie de “nodos de control”, que son lugares en el espacio donde deben cambiar su dirección, definida por sus velocidades en X, e Y, es decir (Vx,Vy)

Una forma de controlar que las 8 naves cambien de dirección en dichos lugares sería comparar sus coordenadas X,Y con la de cada uno de los nodos de control y si coinciden con alguno de ellos, entonces aplicamos las velocidades nuevas asociadas al

cambio en ese nodo. Puesto que hablamos de 2 coordenadas, 8 naves y 4 nodos, estamos ante:

$$2 \times 8 \times 4 = 64 \text{ comprobaciones en cada fotograma}$$

Esto no es viable si queremos velocidad desde BASIC, pues no es una estrategia computacionalmente eficiente. Puesto que estamos ante un escenario "**determinista**", podemos estar seguros en cada instante de tiempo donde van a encontrarse cada una de las naves y por lo tanto en lugar de hacer comprobaciones en el espacio, podemos únicamente **centrarnos en la coordenada temporal** (que es el número de fotograma del juego o el también llamado número de "ciclo de juego"). No consideres al tiempo en segundos, sino en fotogramas.



*Fig. 67 Trayectoria definida con “nodos de control”*

Puesto que conocemos a la velocidad a la que se mueven las naves, podemos saber cuándo la primera de ellas pasará por el primer nodo. A ese instante lo llamaremos  $t(1)$ . También asumiremos que debido a la separación entre las naves, la segunda de las naves pasará por el nodo en el instante  $t(1)+10$ . La tercera en  $t(1)+20$  y la octava en  $t(1)+70$ . En lugar de usar fotogramas como unidades de tiempo, usemos decenas de fotogramas: en ese caso los instantes serán  $t(1), (1)+1, t(1)+2$ , etc.

Sabiendo esto podemos controlar el tiempo con dos variables: una contará las decenas ( $i$ ) y otra las unidades ( $j$ ). Para controlar el cambio de las 8 naves en el primer nodo podemos escribir:

```

j=j+1: IF j=10 THEN j=0: i=i+1: IF i>=t(1) AND i<=t(1) +8 THEN
[actualiza velocidad de nave i-t(1) con los valores de velocidad del
nodo 1]
  
```

Como vemos con una sola línea podemos ir cambiando las velocidades de cada nave a medida que van pasando cada una de ellas por el nodo 1. Cada vez que “ $j$ ” se hace cero, incrementamos la variable “ $i$ ” y actualizamos una de las naves. Durante los primeros 80 instantes de tiempo (8 en decenas de fotogramas) se van actualizando cada una de las 8 naves, justo cuando pasan por el nodo de control, es decir, en el instante  $t(1)$  se actualiza el Sprite 0, en el  $(t1)+1$  se actualiza el Sprite 1, en el  $t(1)+2$  se actualiza el Sprite 2, etc.

El Sprite number que aparece en la línea es  $i-t(1)$ , de ese modo si  $t(1)=1$  queremos que sea el fotograma 40, ( $t=4$ ) entonces cuando “ $i$ ” sea 4 se empezara a actualizar el Sprite 0, y cuando “ $i$ ” valga 11 se actualizará el Sprite 7 (8 naves en total).

Ahora vamos a aplicar lo mismo a los 4 nodos. Podríamos ejecutar 4 comprobaciones en lugar de una, pero sería ineficiente. Además, si tuviésemos muchos nodos esto supondría muchas comprobaciones. Podemos hacerlo solo con una, teniendo en cuenta que la primera nave pasa por un nodo en un instante  $t(n)$  y la octava nave pasa por ese nodo en  $t(n)+7$ .

Cuando la primera nave pasa por el primer nodo, tiene sentido pensar en empezar a comprobar el nodo 2, pero no el nodo 3 ni el 4. Ya tenemos el nodo mayor que vamos a controlar.

En cuanto al nodo menor, podemos asumir que, aunque tengamos 20 nodos, estén lo suficientemente separados como para que no haya naves atravesando más de 3 nodos a la vez (vamos a suponer eso y usaremos ese “3” como parámetro). Por lo tanto, el nodo menor a comprobar es el mayor - 3. Al nodo menor lo vamos a llamar “nmin” y al mayor “nmax”. ( $nmin = nmax - 3$ ). El caso que queramos tener plena libertad para poder definir cualquier trayectoria, nmin debe ser nmax menos el número de naves de la hilera.

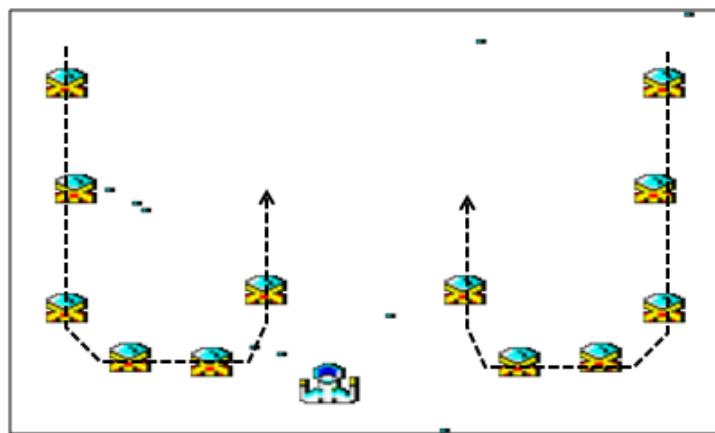
```
10 j=j+1: IF j=10 THEN j=0: i=i+1: n=nmax
20 IF n<nmin THEN 50: ' no hay que actualizar mas naves
30 IF i>=t(n) AND i<=t(n)+8 THEN [actualiza nave i-t(n) con velocidades
de nodo n]:IF i-t(n)=0 THEN nmax=nmax+1: nmin=nmax-3
40 n=n-1
50 ' mas instrucciones del juego
```

Como ves, cuando se incrementa en 1 la decena de tiempo (variable “ $i$ ”), se empieza a comprobar si hay alguna nave en uno de los nodos desde “nmax” hasta “nmin”, actualizando una sola nave en cada fotograma. Si la nave que se actualiza es la cero, entonces el nodo máximo se incrementa, pues esa nave va camino del siguiente nodo.

Para el siguiente fotograma se decrementa el número de nodo (instrucción  $n = n-1$ ) de modo que lo que comprobaremos será si hay una nave en el nodo anterior, así sucesivamente hasta nmin. Siempre, eso sí, comprobando una única nave en cada fotograma.

En resumen, hemos transformado 64 comprobaciones en solo 1, usando “Lógicas masivas”. Y si la trayectoria tuviese 40 en lugar de 4 nodos, ¡habríamos transformado 640 operaciones en una sola!

El videojuego “**Annunaki**” utiliza esta técnica para manejar las trayectorias de dos hileras simétricas de 6 naves cada una. Es complicado, pero como ves desde BASIC puedes tomar el control de 12 naves y hacerlas moverse siguiendo caprichosas trayectorias, usando más inteligencia que potencia, gracias a la técnica de lógicas masivas.



*Fig. 68 Dos hileras con lógicas masivas*

## 12 Trayectorias complejas: Comando ROUTEALL

Este es un comando “avanzado” disponible desde la versión V25 de la librería 8BP. Simplifica muchísimo la programación porque puedes definir una trayectoria y hacer que un sprite la recorra paso a paso mediante el comando ROUTEALL.

Primeramente, necesitas crear una ruta. Para ello necesitas editarla en el fichero routes\_tujuego.asm.

Cada ruta posee un número indeterminado de segmentos (aunque la longitud máxima de una ruta son 255 bytes) y cada segmento tiene tres parámetros:

- Cuantos pasos vamos a dar en ese segmento (entre 1 y 250)
- Qué velocidad Vy se va a mantener durante el segmento ( $-127 \leq Vy \leq 127$ )
- Qué velocidad Vx se va a mantener durante el segmento ( $-127 \leq Vx \leq 127$ )

Como una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, una ruta puede tener como mucho 84 segmentos.

Al final de la especificación de segmentos debemos poner un cero para indicar que la ruta se ha terminado y que el sprite debe comenzar a recorrer la ruta desde el principio.

Veamos un ejemplo:

```
; LISTA DE RUTAS
;=====
;pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4
; DEFINICION DE CADA RUTA
;=====
ROUTE0; un circulo
;-----
    db 5,2,0; cinco pasos con Vy=2
    db 5,2,-1; cinco pasos con Vy=2, Vx=-1
    db 5,0,-1
    db 5,-2,-1
    db 5,-2,0
    db 5,-2,1
    db 5,0,1
    db 5,2,1
    db 0

ROUTE1; izquierda-derecha
;-----
    db 10,0,-1
    db 10,0,1
    db 0

ROUTE2; arriba-abajo
;-----
    db 10,-2,0
    db 10,2,0
```

```

db 0

ROUTE3; un ocho
;-----
db 15,2,0
db 5,2,-1
db 5,0,-1
db 25,-2,-1
db 5,0,-1
db 5,2,-1
db 15,2,0
db 5,2,1
db 5,0,1
db 25,-2,1
db 5,0,1
db 5,2,1
db 0

ROUTE4; un loop y se va hacia la izquierda
;-----
db 120,0,-1
db 10,-2,-1
db 20,-2,0
db 10,-2,1
db 5,0,1
db 10,2,1
db 20,2,0
db 10,2,-1
db 80,0,-1
db 0

```

Ahora para usar las rutas desde BASIC, simplemente asignamos la ruta a un sprite con el comando SETUPSP indicando que queremos modificar el parámetro 15, que es el que indica la ruta. Además, hay que activar el flag de ruta (bit 7) en el byte de status del sprite y le pondremos con el flag de movimiento automático y el de animación y el de impresión.

```

10 MEMORY 24999
11 ON BREAK GOSUB 280
20 MODE 0:INK 0,0
21 LOCATE 1,20:PRINT "comando |ROUTEALL y macrosecuencias de
animacion"
30 CALL &6B78:DEFINT a-z
31 |SETLIMITS,0,80,0,200
40 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT
41 x=10
50 FOR i=1 TO 8
51 x=x+20:IF x>=80 THEN x=10:y=y+24
60 |SETUPSP,i,0,143: rem con esto activo el flag de ruta
70 |SETUPSP,i,7,2:|SETUPSP,i,7,33: rem macrosecuencia de animacion
71 |SETUPSP,i,15,3: rem asigno la ruta numero 3
80 |LOCATESP,i,30,70
82 FOR t=1 TO 10:|ROUTEALL:|AUTOALL,0:|PRINTSPALL,1,0:NEXT
91 NEXT
100 |AUTOALL,1:|PRINTSPALL,1,0: rem aqui AUTOALL ya invoca a ROUTEALL
120 GOTO 100

```

## 280 |MUSIC:MODE 1: INK 0,0: PEN 1

Ya lo tenemos todo. Esta técnica avanzada te va a simplificar la programación muchísimo con resultados espectaculares.



Fig. 69 una ruta en forma de 8

Como has visto, el comando no modifica las coordenadas de los sprites, de modo que deben ser movidos con **|AUTOALL** e impresos (y animados) con **|PRINTSPALL**. Es por ello que dispones de un parámetro opcional en **|AUTOALL**, de modo que **|AUTOALL,1** invoca internamente a **|ROUTEALL** antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

### 12.1 Coloca a un Sprite en mitad de una ruta : ROUTESP

Si asignas a varios sprites la misma ruta y quieras que vayan todos en fila india como el ejemplo anterior, entonces necesitas ingeníártelas para que cada Sprite se encuentre en un punto diferente de la ruta. Para ello existe puedes usar dos alternativas

La primera consiste en ir asignando poco a poco la ruta a los sprites. De ese modo, el sprite que va en cabeza es el primero en ser enrutado y tras unos cuantos ciclos de juego enrutas al siguiente y más tarde al siguiente, etc. En cada ciclo ejecutas **|AUTOALL,1** y eso enruta a los sprites que ya tengan ruta asignada, que se van adelantando en el número de pasos respecto de los sprites que aún no la tienen asignada.

La segunda opción consiste en usar el comando **|ROUTESP**

**|ROUTESP, <spriteid>, <pasos>**

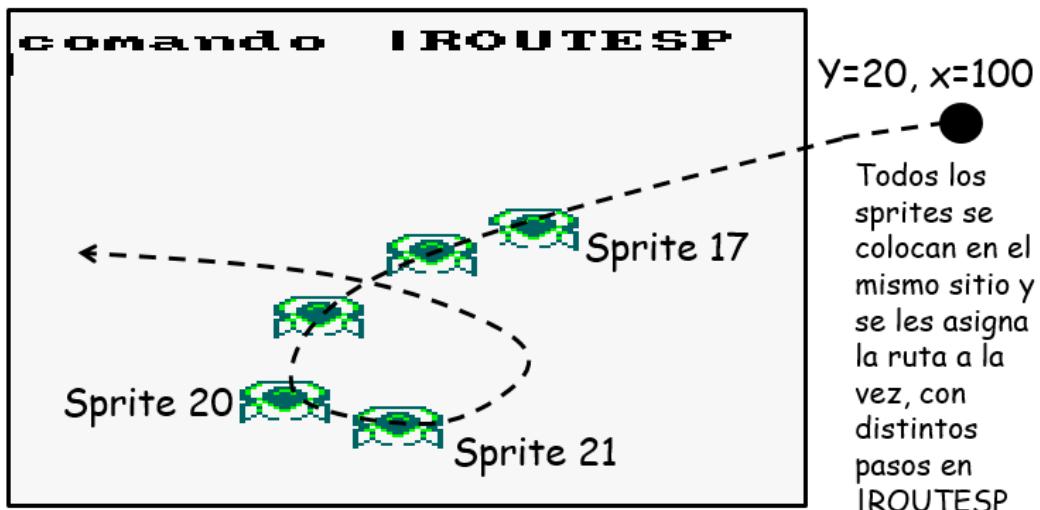
Este comando mueve un sprite el número de pasos que deseas (**hasta 255 pasos**) a lo largo de la ruta que tenga asignada. En el ejemplo he destacado en rojo la asignación de la ruta 8 y los comandos **|ROUTESP** que posicionan cada uno de los sprites en la ruta

```
10 memory 24999
10 ON BREAK GOSUB 12
11 GOTO 20
12 |MUSIC:CALL &BC02:PAPER 0: PEN 1: MODE 1: END
20 CALL &BC02:DEFINT A-Z: MODE 0
50 CALL &6B78
70 ' todas las naves colocada en la misma coordenada inicial
75 ' y con la misma ruta pero con un numero inicial de pasos distinto
```

```

76 locate 2,3: Print "comando |ROUTEESP "
80 for s=16 to 21: |SETUPSP,s,9,33: |SETUPSP,s,0,137:
|SETUPSP,s,15,8: |LOCATESP,s,20,100:next
90 s=21: |ROUTEESP,s,40: 'nave cabeza
100 s=20: |ROUTEESP,s,30
110 s=19: |ROUTEESP,s,20
120 s=18: |ROUTEESP,s,10
130 s=17: |ROUTEESP,s,0
131 |PRINTSPALL,0,0,0
140 '--- ciclo de juego ---
150 |AUTOALL,1: |PRINTSPALL
160 goto 150

```



*Fig. 70 naves en hilera gracias al uso de ROUTESP*

La ruta 8 estaría definida en el fichero routes\_mygame.asm tal como sigue:

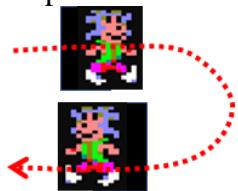
```

ROUTE8;
db 255, 128+64+32+8+1,0; cambio de estado
db 70,1,-1
db 10,3,-1
db 5,3,0
db 5,3,1
db 5,0,1
db 20,-3,1
db 5,0,1
db 5,3,1
db 5,3,0
db 10,3,-1
db 5,0,-1
db 20,-3,-1
db 40,-1,-1
; recolocacion
db 1,0,115
db 1,-30,0
db 120,0,0
db 120,0,0
db 0

```

## 12.2 Creación de Rutas avanzadas

Existen 4 funcionalidades que puedes usar en mitad de cualquier ruta (**ojo, en mitad, no al final**), usando como valor del número de pasos de un segmento un código de escape:

Código de escape	Descripción	Ejemplo
<b>255</b>	Cambio de estado del sprite.	DB 255, 3, 0 Estado pasa a valor 3. El cero del final es de relleno
<b>254</b>	Cambio de secuencia de animación del sprite  Tras cambiar la secuencia, si quieras que también cambie la imagen debes usar el código 251	DB 254, 10, 0 Se asocia la secuencia 10. El cero es de relleno Si la secuencia asignada es la que ya tiene el sprite, entonces es inocuo (no se reinicia el frame id). En caso de querer reiniciar el frame id, el tercer parámetro debe ser un 1 , por ejemplo: DB 254, 10, 1
<b>253</b>	Cambio de imagen 	DB 253 DW new_img Se asocia la imagen “new_img” que debe ser una dirección de memoria
<b>252</b>	Cambio de ruta	DB 252,2,0 Se asocia la ruta 2
<b>251</b>	Pasa al siguiente frame de la animación. 	DB 251,0,0 Se anima el Sprite. Los dos ceros son de relleno

**IMPORTANTE:** ten mucho cuidado de escribir DB y DW donde deben usarse, es decir, por ejemplo, si cambias de imagen debes preceder la imagen con DW y no con DB. Si cometes un error de este tipo, tu ruta no funcionará.

**IMPORTANTE:** los códigos de escape puedes emplearlos en mitad de una ruta, pero el ultimo segmento no puede ser un código de escape, debe ser un movimiento, aunque sea quedarse quieto, algo como “DB 1,0,0”

### 12.2.1 Cambios de estado forzados desde rutas

Esta capacidad es muy interesante para acelerar tus juegos y está disponible desde la V27. Consiste en que podemos forzar un cambio de estado en mitad de una ruta. Para ello indicaremos que deseamos un cambio de estado indicando como valor de número de pasos del segmento = 255.

El cambio de estado es un segmento más y es importante que mantengas el mismo número de parámetros por segmento, es decir, 3 bytes. Un cambio de estado a status=13 podría escribirse como:

**255,13,0**

El tercer parámetro (el cero) no significa nada, es solo un “relleno” para que el segmento mida 3 bytes, pero es imprescindible.

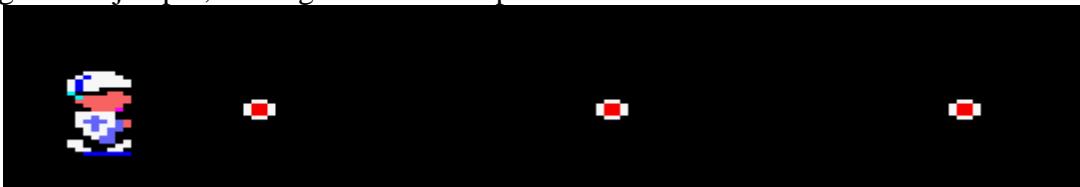
El valor 255 le indicará al comando **|ROUTEALL** que lo que debe hacer esta vez es cambiar el estado del sprite, asignándole el que se indique a continuación. El cambio de estado se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de estado. Si no queremos que el sprite se mueva mas, simplemente definiremos un segmento de un paso sin movimiento en X ni en Y justo después del cambio de estado. Veamos un ejemplo:

```
ROUTE3; disparo_dere
;-----
    db 40,0,2; cuarenta pasos a la derecha con Vx=2
    db 255,0,0; cambio de estado a cero
    db 1,0,0; quieto Vy=0, Vx=0
    db 0

ROUTE4; disparo_izq
;-----
    db 40,0,-2; cuarenta pasos a la izquierda con Vx=-2
    db 255,0,0; cambio de estado a cero
    db 1,0,0; quieto Vy=0, Vx=0
    db 0
```

Estas dos rutas las vamos a usar para disparar con nuestro personaje. La primera de ellas, tras recorrer 40 pasos en los que avanza 2 bytes en X, sufre un cambio de estado y el sprite pasa a estado 0, es decir, desactivado. El siguiente segmento solo tiene un paso y no hay movimiento (vy=0, vx=0).

Con este mecanismo podemos disparar y que los disparos se desactiven solos cuando se salen de la pantalla. Ello ahorra lógica de BASIC y acelera nuestros juegos. En el siguiente ejemplo, la imagen 26 es el disparo.



*Fig. 71 disparos con cambio de estado en ruta*

```
10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 ON BREAK GOSUB 280
30 CALL &BC02:'restaura paleta por defecto por si acaso
40 INK 0,0: 'fondo negro
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT: 'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
90 x=40:y=100: ' coordenadas del personaje
100 |SETUPSP,0,0,1: ' status del personaje
110 |SETUPSP,0,7,1:dir=1: 'secuencia de animacion asignada al empezar
120 |LOCATESP,0,y,x: 'colocamos al sprite (sin imprimirlo aun)
```

```

125 |MUSIC,0,0,6
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'disparos
130 'ciclo de juego---
150 |AUTOALL,1:|PRINTSPALL,0,0
170 ' rutina movimiento personaje -----
180 IF INKEY(27)=0 THEN IF dir=2 THEN dir=1:|SETUPSP,0,7,dir ELSE
|ANIMA,0:x=x+1:GOTO 191
190 IF INKEY(34)=0 THEN IF dir=1 THEN dir=2:|SETUPSP,0,7,dir ELSE
|ANIMA,0:x=x-1
191 IF espera<ciclo-10 then if INKEY(47)=0 THEN espera=ciclo:disp= 1+
disp mod 4 :|LOCATESP,10+disp,y+8,x: |SETUPSP,10+disp,0,137:
|SETUPSP,10+disp,15,2+dir
200 |LOCATESP,0,y,x
201 ciclo=ciclo+1
210 goto 150
280 |MUSIC:MODE 1: INK 0,0:PEN 1

```

Los cambios de estado se pueden forzar en cualquier segmento de la ruta, no necesariamente al final, aunque en el caso de un disparo es muy lógico hacerlo al final de la ruta.

### 12.2.2 Cambios de secuencia forzados desde rutas

Podemos cambiar la secuencia de animación de un sprite usando un segmento especial. Cuando pongamos 254 en el valor del número de pasos, el comando ROUTEALL interpretará que se debe realizar un cambio de secuencia de animación en el sprite. Ejemplo:

**254,10,0**

Este segmento cambia la secuencia de animación del sprite, estableciendo la secuencia número 10. El tercer parámetro (el cero) significa que no se va a reiniciar la secuencia, de modo que si el sprite se encuentra en el frame 5 de otra secuencia, se asignará la nueva secuencia y se mantendrá disco frame, aunque ahora corresponderá a otra imagen. Esto es muy útil porque podemos asignar una secuencia a un sprite dentro de una ruta y si ya la tenía, es inocuo. Para que dicha secuencia asignada se reinicie (se pase a frame 0) hay que poner un 1 en el tercer parámetro:

**254,10,1**

En caso de asignar una secuencia y querer que se reinicie, es recomendable justo después usar el código de animación, el 251 que luego veremos. De ese modo, la imagen asociada al sprite cambiará en la tabla de sprites y no solo el **frame\_id**.

Al igual que con el cambio de estado, el cambio de secuencia se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de secuencia.

### 12.2.3 Cambios de imagen forzados desde rutas

Hemos visto como enrutar sprites con ROUTEALL o aun mejor, con AUTOALL,1

A menudo no queremos enrutar un sprite a través de una trayectoria sino algo más cotidiano: dar un salto con un personaje. En el ejemplo del capítulo 9 vimos cómo hacerlo con un array de BASIC que contiene los movimientos relativos de la coordenada Y. En este caso vamos a hacer lo mismo con una ruta, logrando un movimiento más veloz.

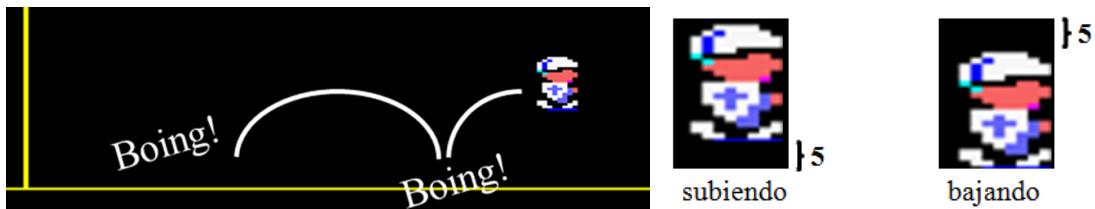


Fig. 72 Saltar usando una ruta

Para no tener que controlar si el personaje ha llegado al punto cenital del salto, podemos usar un segmento especial que indica cambio de imagen. Al igual que cualquier otro segmento, consume 3 bytes, pero en este caso el primero es el indicador de cambio de imagen (un valor 253) y los dos siguientes se corresponden con la dirección de memoria de la imagen. **MUCHA ATENCIÓN**, deberás usar “**dw**” antes del nombre de la imagen que quieras asignar, por lo que este segmento de cambio de imagen lo tendrás que escribir en dos líneas. Un “**db**” para el 253 y un “**dw**” para la dirección de memoria de la imagen.

```
db 253
dw SOLDADO_R1_UP
```

En el ejemplo siguiente tenemos un muñeco que salta. Al subir el muñeco que se borra a sí mismo por abajo mientras que al bajar se borra a sí mismo por arriba. Para que no haya discontinuidad en el movimiento, justo al cambiar una imagen por otra es necesario realinear verticalmente al soldado, subiendo la imagen de bajada exactamente 5 líneas, para hacerla coincidir con el soldado que estaba subiendo.

Este sería el fichero sequences\_mygame.asm

```
;=====
; hasta 31 secuencias de animacion
;=====
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se gastan 8 words
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo

; la secuencia cero es que no hay secuencia.
; empezamos desde la secuencia 1

;-----secuencias de animacion -----
SEQUENCES_LIST
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;1
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;2
dw SOLDADO_R1_UP,0,0,0,0,0,0,0;3
dw SOLDADO_R1_DOWN,0,0,0,0,0,0,0;4
dw SOLDADO_L1_UP,0,0,0,0,0,0,0;5
dw SOLDADO_L1_DOWN,0,0,0,0,0,0,0;6
```

```

_MACRO_SEQUENCES
;-----MACRO SECUENCIAS -----
; son grupos de secuencias, una para cada direccion
; el significado es:
; still, left, right, up, up-left, up-right, down, down-left, down-right
; se numeran desde 32 en adelante
db 0,2,1,3,5,3,4,6,4; 32 --> secuencias del soldado , id=32. la siguiente
seria la 33

```

Usaremos dos rutas, una para saltar a la derecha y otra a la izquierda. Este sería el fichero routes\_mygame.asm

```

; LISTA DE RUTAS
;=====
; pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4

; DEFINICION DE CADA RUTA
;=====
ROUTE0; jump_right
    db 253
    dw SOLDADO_R1_UP
    db 1,-5,1
    db 2,-4,1
    db 2,-3,1
    db 2,-2,1
    db 2,-1,1
    db 253
    dw SOLDADO_R1_DOWN
    db 1,-5,1; subo para que UP y down encajen
    db 2,1,1
    db 2,2,1
    db 2,3,1
    db 2,4,1
    db 1,5,1
    db 253
    dw SOLDADO_R1
    db 1,5,1; baja una mas, ya que R1 no tiene negros arriba
    db 255,13,0; nuevo estado, ya sin flag de ruta y con flag de animacion
    db 254,32,0; macrosecuencia 32
    db 1,0,0; quietooo.!!!!
    db 0

ROUTE1; jump_left
    db 253
    dw SOLDADO_L1_UP
    db 1,-5,-1
    db 2,-4,-1
    db 2,-3,-1
    db 2,-2,-1
    db 2,-1,-1
    db 253
    dw SOLDADO_L1_DOWN
    db 1,-5,-1; subo para que UP y down encajen
    db 2,1,-1

```

```

db 2,2,-1
db 2,3,-1
db 2,4,-1
db 1,5,-1
db 253
dw SOLDADO_L1
db 1,5,-1; baja una mas
db 255,13,0; nuevo estado, ya sin flag de ruta y con flag de animacion
db 254,32,0; macrosecuencia 32
db 1,0,0; quietooo.!!!!
db 0

ROUTE2; bird
db 30,0,-1
db 10,0,0
db 20,2,-1
db 20,-2,-1
db 0

ROUTE3; disparo_dere
;-----
db 40,0,2
db 255,0
db 1,0,0
db 0

ROUTE4; disparo_izq
;-----
db 40,0,-2
db 255,0
db 1,0,0
db 0

```

Y este sería el ejemplo de programa. Compara su ejecución con la del capítulo 7 para ver la diferencia de velocidad. Notarás una gran diferencia

```

10 MEMORY 24999
20 MODE 0: DEFINT A-Z: CALL &6B78: ' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02:ink 0,0:'restaura paleta por defecto
50 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
90 x=40:y=100:jump=0:ciclo=40: ' coordenadas del personaje
100 |SETUPSP,0,0,13:|SETUPSP,0,5,0,0: ' status del personaje
110 |SETUPSP,0,7,1:|SETUPSP,0,7,32: ' secuencia de animacion asignada al
empezar
120 |LOCATESP,0,y,x: 'colocamos al sprite (sin imprimirlo aun)
123 locate 1,1: print "pulsa Q" : print "para saltar": print "ejemplo
con ruta"
124 print "pulsa SPACE para disparar"
125 PLOT 1,150:DRAW 640,150: PLOT 92,150:DRAW 92,400: 'suelo y pared
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'disparos
127 |MUSIC,0,0,5: 'comienza a sonar la musica
130 'ciclo de juego -----
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' rutina movimiento personaje -----

```

```

172 IF INKEY(47)=0 THEN if esperaciclo<10 then espera=ciclo:disp= 1+
disp mod
4: |LOCATESP,10+disp,peek(27001)+8,peek(27003):|SETUPSP,10+disp,0,137:|
SETUPSP,10+disp,15,3+dir
173 if peek(27000)>128 then 193 else |SETUPSP,0,6,0: ' si estado es
>128 es que estoy saltando (tiene ruta)
174 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'saltar
180 IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'ir izquierda
193 ciclo=ciclo+1
310 goto 150
2800 |MUSIC:MODE 1: INK 0,0:PEN 1

```

Para comprobar que el muñeco se encuentra saltando simplemente consulto el estado mediante un peek (27000). De este modo, sabremos si tiene el flag de enrutamiento activo y en ese caso no pasaremos por las líneas que lo mueven a derecha e izquierda.

#### **12.2.4 Cambios de ruta forzados desde rutas**

Podemos cambiar la ruta de un sprite usando un segmento especial. Cuando pongamos 252 en el valor del número de pasos, el comando ROUTEALL interpretará que se debe realizar un cambio de ruta en el sprite. Ejemplo:

**252,2,0**

Este segmento cambia la ruta del sprite, estableciendo la ruta número 2. El tercer parámetro (el cero) no significa nada, es solo un “relleno” para que el segmento mida 3 bytes, pero es imprescindible.

Al igual que con el cambio de estado, el cambio de ruta se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de ruta, que será el primer paso del primer segmento de la nueva ruta.

Como una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, una ruta puede tener como mucho 84 segmentos. Es posible que necesites construir una ruta aun más larga y en ese caso podrás hacerlo concatenando el fin de una ruta con un cambio de ruta hacia otra ruta, y puedes concatenar tantas rutas como deseas

#### **12.2.5 Cambios de ruta forzados desde BASIC**

Supongamos que quieres que tu personaje salte hacia la derecha y en mitad del salto quieras que el personaje pueda cambiar hacia la izquierda, continuando el salto. Lo que estamos planteando se puede representar con este dibujo:

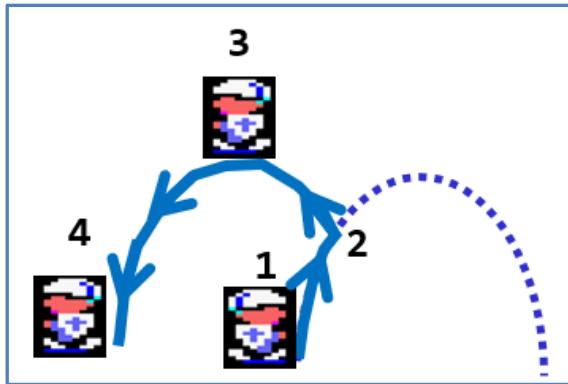


Fig. 73 Cambio de ruta en mitad de un salto

En este ejemplo nuestro personaje salta y en mitad de la ruta de salto hacia la derecha, cambia de dirección en el punto 2, continuando con la ruta de salto a la izquierda, pero sin comenzar un nuevo salto, simplemente continuando el salto y alcanzando la misma altura (punto 3) que alcanzaría con el salto hacia la derecha para finalmente terminar el salto a la izquierda en el punto 4.

Para poder hacer esto necesitamos cambiar la ruta a nuestro personaje desde BASIC en el punto 2, pero no podemos usar el comando **|SETUPSP** porque ello iniciaría la ruta, obteniendo un salto mucho mayor, que comenzaría en el punto 2. Lo que podemos hacer en ese caso es simplemente un **POKE** a la dirección de memoria que almacena la ruta del Sprite, que es la dirección de su estado +15, tal como muestra la tabla de atributos de sprites. Este **POKE** cambia la ruta, pero mantiene intacta la posición (número de segmento y posición en que se encuentra el personaje). OJO: las dos rutas deben tener los mismos segmentos y longitud, porque si saltas a una ruta más corta y te encuentras en un segmento que no existe en esa ruta, puede ocurrir un efecto imprevisible.

Suponiendo que tenemos dos rutas de salto (ruta 0 salto a derecha y ruta 1 salto a izquierda), el siguiente ejemplo ilustra el concepto:

```

130 'ciclo de juego -----
150 ciclo=ciclo+1:AUTOALL,1:|PRINTSPALL,0,1,0
170 ' rutina movimiento personaje -----
173 IF PEEK(27000)<128 THEN 178
174 'estamos en mitad de un salto
175 IF INKEY(27)=0 THEN POKE 27015,0:GOTO 180: 'cambio ruta a derecha
176 IF INKEY(34)=0 THEN POKE 27015,1:GOTO 180: 'cambio ruta a izq
177 GOTO 150
178 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir: 'saltar
180 IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1: 'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1: 'ir izquierda
310 GOTO 150

```

La única limitación del uso de **POKE** para este propósito es que la imagen del personaje no cambia hasta que no encuentra un código de cambio de imagen en mitad de la ruta. Cambiar la imagen usando **|SETUPSP** es posible pero peligroso, ya que no sabes si el personaje está subiendo (borrado con líneas inferiores) o bajando (borrado con líneas superiores). Por ello es mejor simplemente asignar la ruta y que la propia ruta cambie la imagen lo antes posible. Puedes incluso poner en mitad de la ruta cambios de imagen, aunque no sean necesarios por si ocurre esta circunstancia. Lo que si tienes que hacer es que la imagen de salto tenga bytes de borrado en ambos lados pues si no lo tiene y cambia de dirección, dejaría un rastro.

## 12.2.6 Animación forzada desde rutas

Podemos dejar inactivo el flag de animación de un Sprite y animarlo solo en determinados instantes de la ruta usando el código 251. Ejemplo:

**251,0,0**

Esto puede ser muy útil para dar sensación de que un Sprite se acerca en juegos que utilizan técnicas pseudo 3D. Por ejemplo, para un meteorito que se acerca y que queremos que cambie de fotograma para aparecer más grande. El meteorito se moverá unas cuantas veces antes de pasar al siguiente tamaño. Este mecanismo es muy similar al de cambio de imagen, solo que te permite definir el cambio de imagen sin especificar explícitamente la imagen, sino simplemente indicando un cambio de frame en la secuencia de animación que tenga asignado el Sprite.



*Fig. 74 Animación forzada desde ruta*

Con este flag puedes usar la misma ruta para un pájaro espacial que se acerca o para un meteorito. Al no indicar la imagen, en cada caso se aplicará la imagen que corresponda a la secuencia que tenga cada sprite.

## 12.2.7 Como construir rutas “dinámicas” (no predefinidas)

Una ruta dinámica es una ruta cuya trayectoria se decide en tiempo de ejecución de tu programa BASIC. Es útil cuando tu programa genera dinámicamente laberintos o circuitos de carreras que un enemigo debe recorrer y que a priori no son conocidas y por lo tanto no pueden ser definidas en el fichero “routes\_mygame.asm”

Para poder hacer una ruta así y asignársela a un Sprite, lo que tenemos que hacer es crear una ruta “vacía” en el fichero “routes\_mygame.asm”, en este caso es la ruta 2

```
; LISTA DE RUTAS
;=====
;pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4

; DEFINICION DE CADA RUTA
;=====
```

```

ROUTE0; derecha izquierda
db 10,0,1
db 10,0,-1
db 0

ROUTE1; arriba abajo
db 10,-1,0
db 10,1,0
db 0

ROUTE2; ruta dinamica
ds 100

```

Hemos creado la ruta 2 con 100 bytes libres para llenar desde BASIC. Puede que la ruta que construyamos ocupe menos y en ese caso bastará con reservar menos bytes.

Una vez ensamblada la librería y los gráficos debemos buscar la dirección de memoria de la etiqueta “ROUTE2” en la ventana de símbolos de winape. Cuando la tengamos, desde BASIC programaremos la ruta usando POKE a partir de esa dirección de memoria y en las siguientes

POKE mete un byte en una dirección de memoria. Nuestros números deben pertenecer al rango -127..128 y POKE no permite meter números negativos. Para hacerlo debes usar el valor positivo con el que internamente el Amstrad representa a los negativos (es decir, el complemento a dos). Para hacerlo basta una operación AND 255

```

10 A=-10
20 PRINT A: REM esto imprime un 10
30 PRINT A AND 255: REM esto imprime un 246 que es el mismo -10

```

En definitiva, si quieras insertar un -10 lo que tienes que insertar es un 246 y usar la misma estrategia para cualquier número negativo. No te olvides que el ultimo POKE de la ruta debe ser la inserción de un cero, que significa fin de ruta.

### 12.2.8 Programación de rutas que incluyen patrones

Vamos a suponer que quieres hacer una ruta para que un enemigo atraviese la pantalla de derecha a izquierda una y otra vez, suponiendo que partimos de una posición inicial en la que el sprite se encuentra en el extremo derecho de la pantalla

```

ROUTE0; ruta sencilla
DB 80,0,-1 ; 80 pasos. En cada paso se mueve 1 byte
DB 1,0,80 ; recolocamos el sprite a su posición original
DB 0

```

Esta ruta mueve a un sprite dando un paso de 1 byte en cada fotograma. Si quisiésemos que fuese más despacio, moviéndose 1 byte cada dos fotogramas, podríamos hacer lo siguiente:

```

ROUTE0; ruta no tan sencilla
DB 1,0,-1
DB 1,0,0
DB 0

```

Ahora esta ruta permite mover a un sprite más despacio, pero no podemos recolocar al sprite a su posición original. Esto es debido a que como la pantalla tiene 80 bytes de ancho, necesitamos 160 segmentos, ya que el sprite avanza 1 byte cada dos segmentos. La ruta resultante sería larguísima y de hecho tendríamos que concatenar 2 rutas pues una ruta solo puede medir 255 bytes (84 segmentos).

La solución óptima es definir una ruta corta sin recolocación del sprite y recolocarlo desde BASIC, mediante algo como:

```
80 |SETUPSP,31, 0, 128+16+1: REM rutable, mov automatico, imprimible
85 |LOCATESP,31,100,80: ' colocado a la derecha de la pantalla.
90 rem ciclo de juego
100 ciclo=ciclo +1
110 |AUTOALL,1: |PRINTSPALL
120 if ciclo MOD 160=0 THEN |LOCATESP,31,100,80: ' recolocacion
130 goto 100
```

Este tipo de estrategias son útiles siempre que no queramos repetir el mismo movimiento en cada fotograma, sino que queremos definir un patrón de repetición en el que en ciertos fotogramas en sprite se mueve en una dirección y en otros se mueve en otra dirección o incluso no se mueve. Veamos otro ejemplo, una ruta inclinada en la que por cada 3 movimientos en vertical nos movemos uno en horizontal

```
ROUTE0; ruta inclinada
DB 2,1,0
DB 1,1,1
DB 0
```

### 12.2.9 Tipología de rutas

Con todo lo que hemos visto podemos clasificar las rutas en los siguientes tipos:

- **Rutas sin fin cíclicas:** recolocan al sprite o simplemente acaban en las mismas coordenadas donde empezaron
- **Rutas sin fin no cíclicas:** avanzan indefinidamente y no recolocan el sprite por lo que se pueden alejar infinitamente del área de juego, a menos que recoloquemos el sprite desde BASIC
- **Rutas con fin:** en el último paso cambian el estado del sprite desactivando el flag de enrutamiento
- **Rutas encadenadas:** desde una ruta se puede saltar a otra y que esta segunda ruta sea cíclica o no cíclica o tenga fin, o incluso acabe saltando a una tercera ruta.

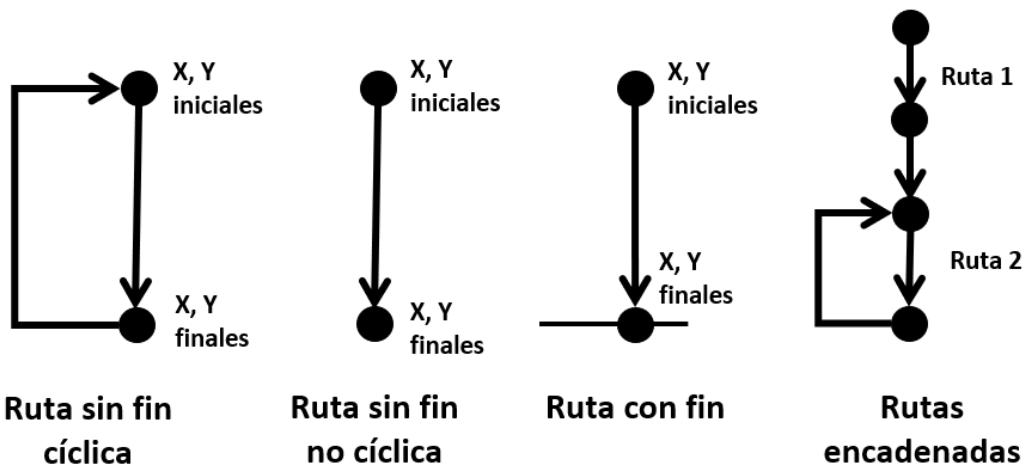
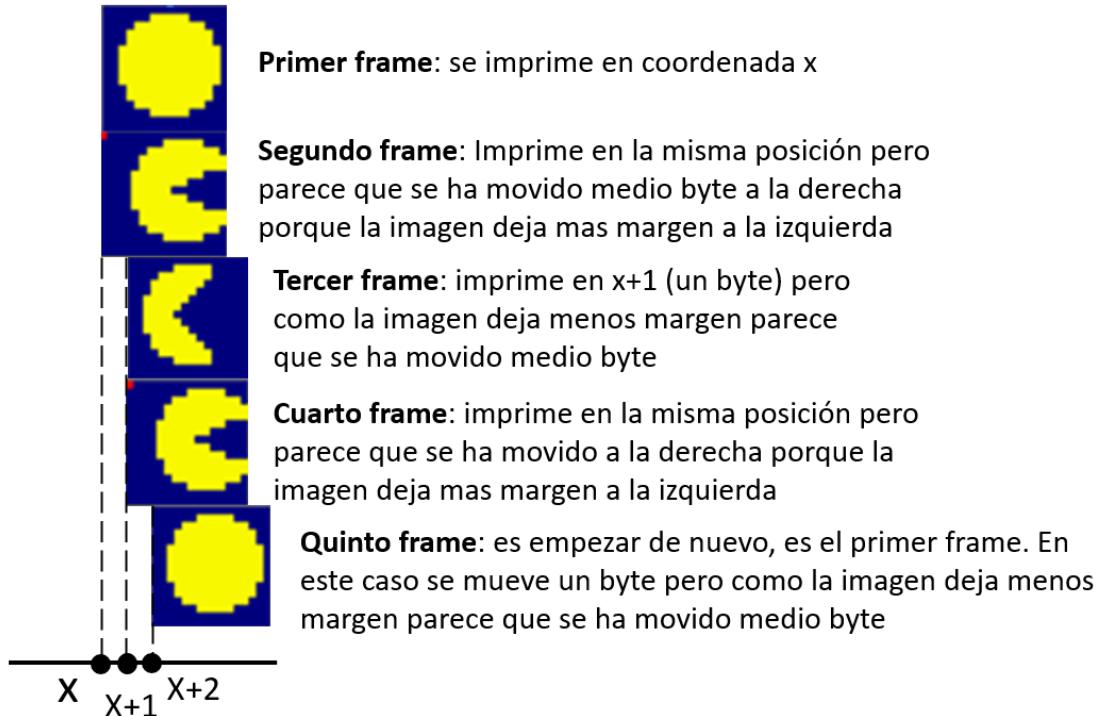


Fig. 75 Tipos de rutas

## 13 Movimiento suave de medio byte

8BP mueve los sprites byte a byte con comandos como MOVER, y su sistema de coordenadas es de bytes, no de pixeles. Por lo tanto, hablamos de 80 posiciones en el eje horizontal y de 200 en el vertical.

Un byte contiene 2 pixeles en mode 0, o bien 4 pixeles en mode 1. Puede que queramos un movimiento más suave (pixel a pixel en mode cero o de dos pixeles en mode 1). Para ello hay un sencillo truco que podemos emplear. Se trata de tener una imagen del personaje desplazada medio byte y simplemente asignársela al Sprite. Aunque se imprima en la misma coordenada, parecerá que se ha movido.



En esta secuencia de animación, hay veces que el comeculos no se mueve, pero como cambiamos la imagen es como si se moviese 2 pixeles (medio byte). En los momentos en los que se mueve 1 byte, la imagen aparece desplazada hacia la izquierda medio byte, de modo que el resultado “neto” también es como si se moviese 2 pixeles (medio byte). Para definir el movimiento de este comeculos podemos usar el mecanismo de las rutas de 8BP. Este sería el ejemplo de la ruta de movimiento hacia la derecha:

```
ROUTE0; derecha
db 253
dw COCO_R1
db 1,0,0
db 253
dw COCO_R2
db 1,0,1
db 253
dw COCO_R1
db 1,0,0
db 253
dw COCO_R0
db 1,0,1
db 0
```

Dibujar las imágenes desplazadas puede resultar algo tedioso. Por ello desde la versión **V15 de SPEDIT** posees un mecanismo de desplazamiento de una imagen que la desplaza medio byte (2 pixeles de mode 1 o un pixel de mode 0) a la derecha o a la izquierda. Como ves en el menú de SPEDIT te aparece una nueva opción: con las flechas de cursor puedes producir un desplazamiento de la imagen que has dibujado



**SPEDIT: Sprite Editor v15.0**  
---- CONTROLES EDICION ----  
1,2 : tinta -/+  
space : mueve u pinta  
**↔: desplaza medio byte**  
n: flip horizontal  
v: flip vertical  
c: clear sprite  
b: imprime bytes (asm) en printer  
i: imprime paleta en printer y file  
r: reload 20000  
z, x: cambia color de tinta  
t: RESET  
  
Antes de empezar puedes cargar un sprite ensamblandolo en la direccion 20000. No ensambles ancho y alto, solo los bytes del dibujo.  
la paleta custom esta en la linea 2300  
La paleta la puedes cambiar para que sea la misma que la de tu videojuego.  
EDITAR sprite(1) o CAPTURAR sprite(2)? █

Esto nos permite fácilmente desplazar una imagen para aplicar la técnica descrita de movimiento suave

## 14 Juegos con scroll

La librería 8BP te permite hacer scroll de diferentes modos que se pueden combinar simultáneamente, aunque el método más importante se basa en el comando **|MAP2SP**. Las técnicas disponibles te las resumo a continuación:

- **Mediante comandos de movimiento en bloque de sprites:** Una forma sencilla de hacer scroll con 8BP es simplemente crear unos sprites decorativos a los que configuramos su estado para ser movidos por los comandos **|MOVEALL** y/o **|AUTOALL**.
- **Mediante |MAP2SP:** La idea que subyace al scroll multidireccional proporcionado por **|MAP2SP** en 8BP es sencilla: todos los elementos que se representan en pantalla son sprites, de modo que los elementos del mundo que vamos a imprimir y mover por pantalla son sprites cuyas imágenes asociadas serán montañas, casas, árboles o lo que necesites para construir tu “mundo”. Para seleccionar una porción del mundo y transformarla en una lista de sprites se usa la función **|MAP2SP**. En cuanto a la función **|UMAP** permite actualizar el mundo con una porción de un mundo mayor.
- **Mediante el comando |STARS,** el cual te permitirá hacer scroll multidireccional de un banco de 40 pixeles que puedes ubicar donde quieras y que puedes mover en diferentes planos y a diferente velocidad.
- **Mediante el comando |RINK,** el cual te permitirá rotar un patrón de tintas, dando sensación de movimiento de avance que puedes utilizar en ciertos tipos de scroll, tales como movimiento de un suelo de ladrillos, agua, etc.

### 14.1 STARS: Scroll de estrellas o tierra moteada

En la librería 8BP dispones de una función muy sencilla de utilizar para crear un efecto de fondo de estrellas que se mueven, dando la sensación de scroll. Se trata de la función **|STARS**. Esta función es capaz de mover hasta 40 estrellas simultáneamente sin alterar tus sprites, de modo que es como si pasasen “por debajo”

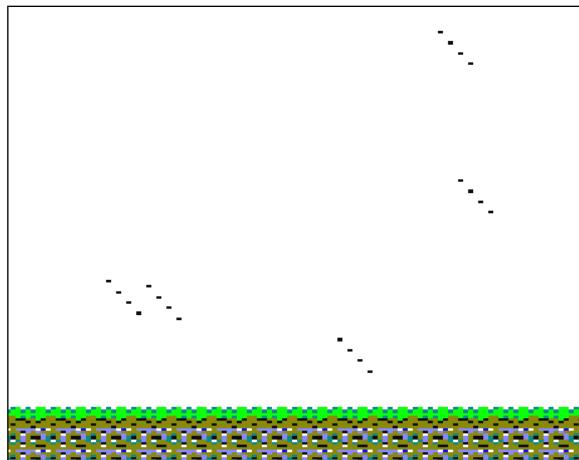
**|STARS,<estrella inicial>,<num estrellas>,<color>,<dy>,<dx>**

Dispones de un banco de estrellas y puedes combinar varios comandos STARS para trabajar con grupos de estrellas a diferente velocidad, dando sensación de planos con distinta profundidad.

El banco de estrellas consiste en 40 pares de bytes representando coordenadas (y,x). Ocupando desde la dirección 42540 hasta 42619 (son 80 bytes en total). Una forma de generar 40 estrellas aleatorias sería (ojo si ya hemos ejecutado DEFINT A-Z el numero 42540 lo debemos poner en hexadecimal porque es mayor de 32768).

```
FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200: POKE dir+1,RND*80:NEXT
```

Para una descripción detallada del comando, consulta el capítulo de “guía de referencia”. En ese capítulo encontrarás distintos ejemplos para simular estrellas, tierra, estrellas con dos planos de profundidad, lluvia o incluso nieve. Probablemente con imaginación sea posible simular más cosas con esta misma función. Por ejemplo, si colocas las estrellas en secuencias de 2 o tres píxeles en diagonal, en lugar de repartirlas aleatoriamente, podrás conseguir un desplazamiento de movimiento de “segmentos”, algo que podría ser ideal para simular lluvia.



*Fig. 76 Efecto de lluvia con STARS*

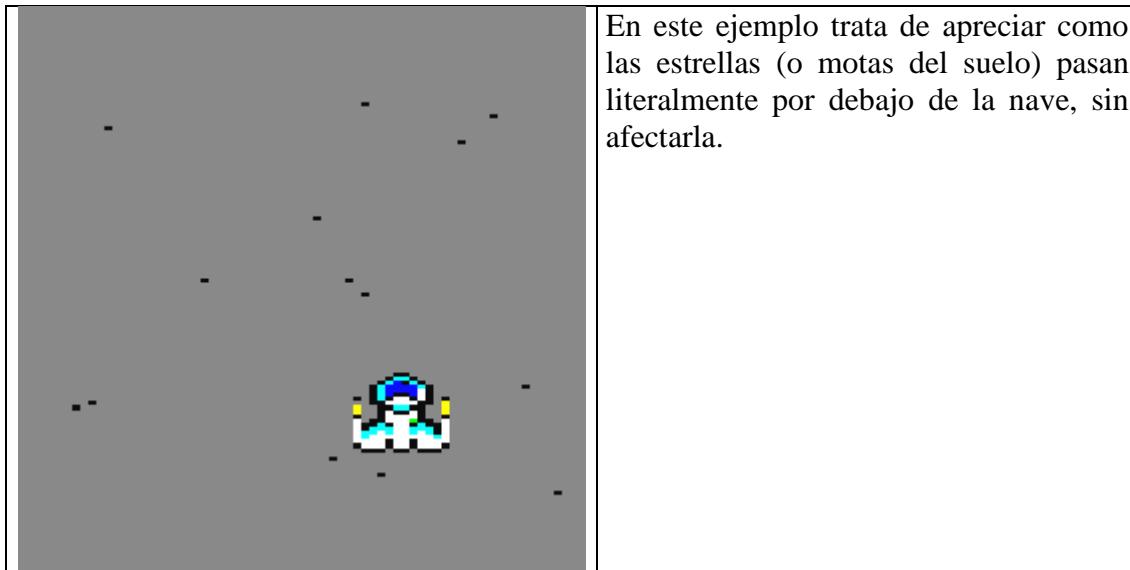
```

10 MEMORY 24999
20 CALL &6B78:' install RSX
40 mode 0:CALL &BC02:'restaura paleta por defecto por si acaso
50 banco=42540
60 FOR dir=banco TO banco+40*2 STEP 8:
70 y=INT(RND*190):x=INT(RND*60)+4
80 POKE dir,y:POKE dir+1,x:
90 POKE dir+2,(y+4):POKE dir+3,x-1
100 POKE dir+4,(y+8):POKE dir+5,x-2
110 POKE dir+6,(y+12):POKE dir+7,x-3
120 NEXT

140 'ESCRITURA DE LLUVIA
141 -----
150 |SETLIMITS,0,80,50,200: ' limites de la pantalla de juego
151 cesped=&84d0:|SETUPSP,30,9,cesped:'letra Y es el sprite 31
152 rocas=&84f2:|SETUPSP,21,9,rocas: 'letra P es el sprite 21
160 cadena$="YYYYYYYYYYYYYYYYYYYY"
170 |LAYOUT,22,0,@cadena$: 'estos pinta el cesped
180 cadena$="PPPPPPPPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@cadena$: 'pinta una fila de rocas
200 |LAYOUT,24,0,@cadena$: 'pinta otra fila de rocas
210 ----- ciclo de juego-----
211 defint a-z
220 LOCATE 1,10:PRINT "DEMO DE LLUVIA"
221 LOCATE 1,11:PRINT "pulsa ENTER"
230 |STARS,0,40,4,2,-1
240 IF INKEY(18)=0 THEN 300
250 GOTO 230

```

Como el ejemplo de doble plano de estrellas lo tienes en el capítulo de referencia de la librería, aquí vamos a ver un ejemplo en el que una nave espacial sobrevuela un planeta de tierra moteada, con sensación de scroll vertical



*Fig. 77 Efecto de tierra moteada con STARS*

Existe un modo de invocar de forma optimizada el comando STARS y consiste simplemente en invocarlo una primera vez con parámetros y las siguientes veces sin parámetros. El comando asumirá que los valores de los parámetros son los mismos que los de la última invocación con parámetros y ello permite ahorrar tiempo que el intérprete BASIC dedica a procesar los parámetros, hasta 1.7ms

```

10 MEMORY 24999
11 'pongo estrellas aleatorias
12 FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: 'limites de la pantalla de juego

41 ' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
44 x=40:y=150: ' coordenadas de nave

49 '----- ciclo de juego-----
50 |STARS,0,20,5,1,0:' estrellas negras sobre suelo gris
55 gosub 100: ' movimiento de la nave
60 |PRINTSPALL,0,0
70 goto 50

99 ' rutina movimiento nave -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN

```

## 14.2 Scroll usando MOVERALL y/o AUTOALL

Ahora haciendo uso combinado del movimiento relativo, del scroll de estrellas y del orden en que se imprimen los sprites vamos a ver un ejemplo de cómo simular que una nave sobrevuela un paisaje lunar.

Primeramente, hemos escogido el sprite 31 para nuestra nave, porque eso hará que se imprima la última. Los sprites se imprimen en orden, comenzando por el cero y acabando en el 31. Si un cráter es un sprite inferior a 31 se imprimirá antes que la nave y la nave quedará “por encima”, dando la sensación de que lo está sobrevolando.

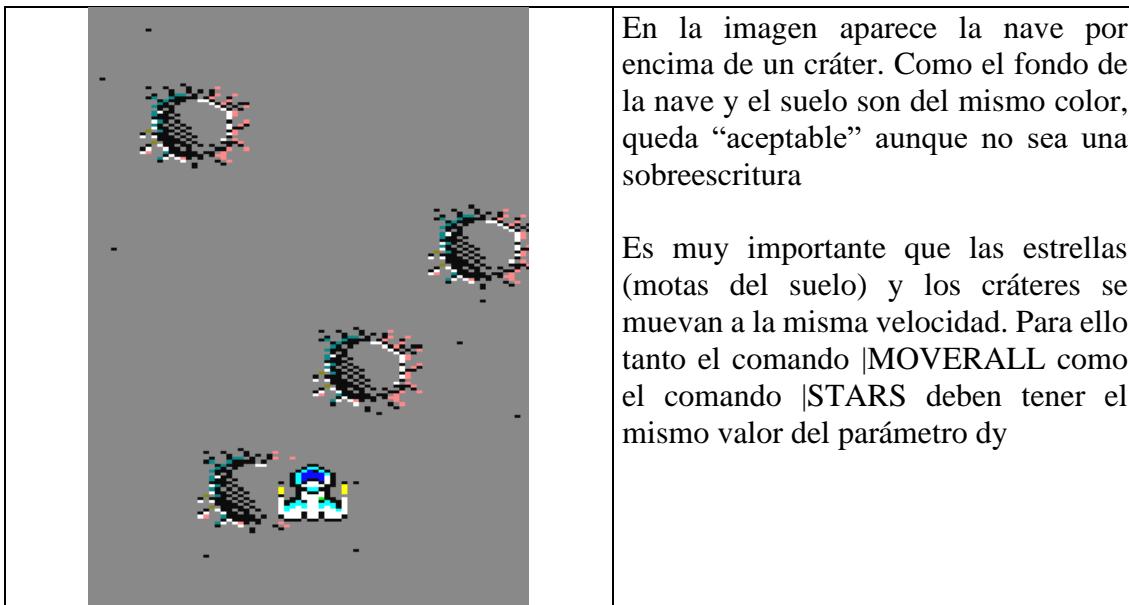


Fig. 78 sobrevolando la luna

En la imagen aparece la nave por encima de un cráter. Como el fondo de la nave y el suelo son del mismo color, queda “aceptable” aunque no sea una sobreescritura

Es muy importante que las estrellas (motas del suelo) y los cráteres se muevan a la misma velocidad. Para ello tanto el comando |MOVERALL como el comando |STARS deben tener el mismo valor del parámetro dy

Este es el código BASIC :

```
10 MEMORY 24999
11 'pongo estrellas aleatorias
12 FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' limites de la pantalla de juego

41 ' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
45 x=40:y=150: ' coordenadas de nave

46 ' ahora los crateres
47 crater=&a39a: cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&x10001: ' impresion y movimiento relativo
50 x(i)=rnd*40+20:y(i)=i*40
60 |locatesp,i,y(i),x(i)
70 next
```

```

71 t=0

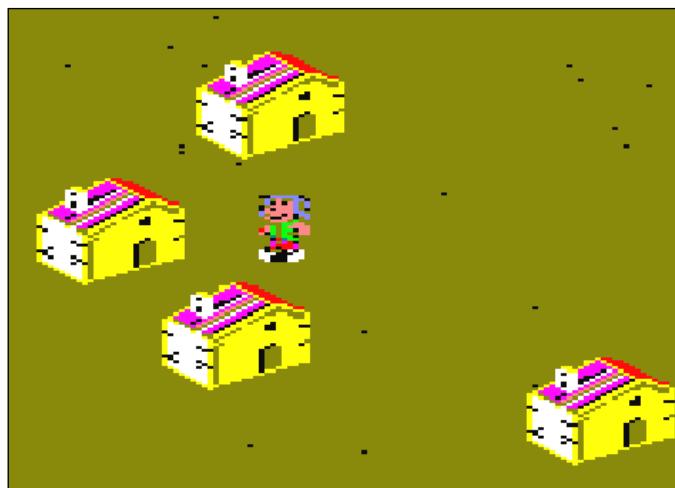
80 ----- ciclo de juego-----
81 |STARS,0,20,5,3,0: ' movimiento estrellas negras
82 gosub 100: ' movimiento de la nave
83 |MOVERALL,3,0: 'movimiento de crateres
84 t=t+1: if t> 10 then t=0:gosub 200: ' control de crateres
90 |PRINTSPALL,0,0: ' impresion de nave y crateres
91 goto 81

99 ' rutina movimiento nave -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN

199 ' control de reentrada de crateres
200 c=c+1: if c=6 then c=0
220 |PEEK,27001+c*16,@cy%
230 if cy%>200 then |POKE,27001+c*16,-20
240 return

```

Vamos a ver un último ejemplo que utiliza movimiento relativo para dar la sensación de scroll, usando sprites con dibujos de casas, un suelo moteado y un personaje situado en el centro que según la dirección que tome, hace que todo se mueva entorno a él. Es un ejemplo muy básico, pero te da una idea del potencial de estas funciones. ¡Aquí lo que se mueve es todo el pueblo!



*Fig. 79 El pueblo entero se mueve*

```

10 MEMORY 24999
20 MODE 0: call &6b78
30 DEFINT a-z
240 INK 0,12
241 border 7
250 FOR i=0 TO 31
260 |SETUPSP,i,0,&X0
270 NEXT
280 FOR i=0 TO 3

```

```

290 |SETUPSP,i,0,&X10001
300 |SETUPSP,i,9,&A01c:rem casas
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 NEXT
320 |SETUPSP,31,7,6: rem personaje
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,&X1111
400 xa=0:ya=0
410 IF INKEY(27)=0 THEN xa=-1:
420 IF INKEY(34)=0 THEN xa+=1:
430 IF INKEY(67)=0 THEN ya+=2
440 IF INKEY(69)=0 THEN ya-=2
450 |MOVERALL,ya,xa
460 |PRINTSPALL,1,0
470 |STARS,1,20,5,ya,xa
480 GOTO 410

```

### 14.3 Técnica del “manchado”

La técnica para pintar montañas en juegos con scroll horizontal y lagos en juegos con scroll vertical es la misma. Lo que haremos es pintar solo el comienzo de la montaña, mediante un sprite que nos sirve para pintar su lado izquierdo. Pondremos tantos como queramos. En este caso yo he puesto tres

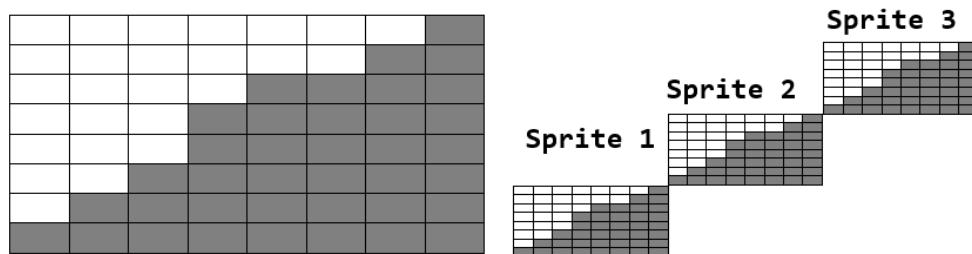


Fig. 80. Definir la ladera de una montaña con varios sprites

Hacemos lo mismo con una imagen espejada que asociaremos a otros 3 sprites, y los situaremos a la derecha, construyendo el lateral derecho de la montaña. Cuidado de que la imagen espejada al menos tenga las dos últimas columnas de pixels a cero. Esto le permitirá borrarse a sí misma al avanzar a la izquierda. Ten en cuenta que en 8BP los sprites se mueven byte a byte (dos pixels a la vez).

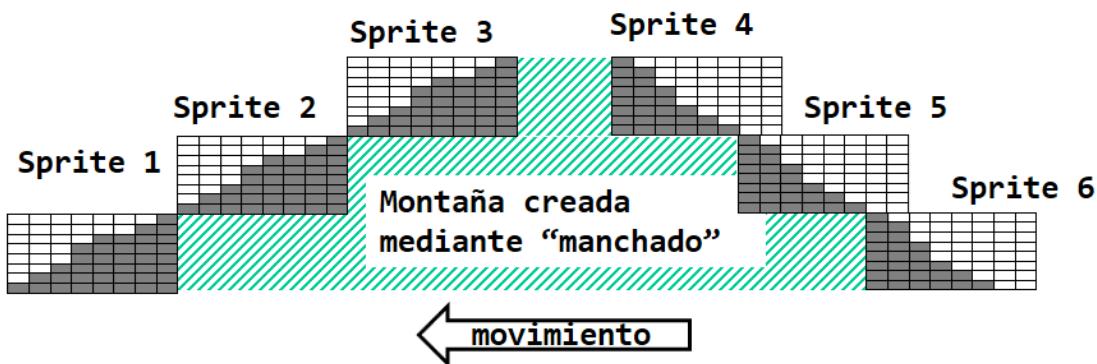
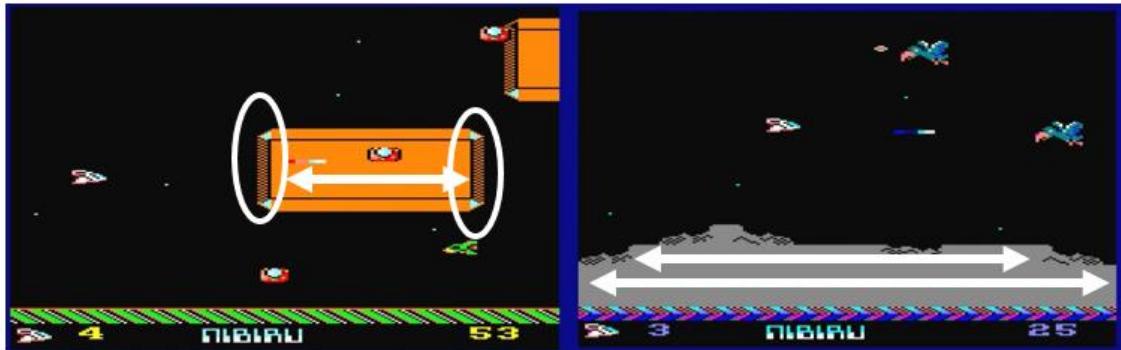


Fig. 81 Montaña creada con 6 sprites y la técnica del “manchado”

Al mover todos los sprites hacia la izquierda mediante movimiento automático o relativo, los sprites de la izquierda empezarán a “manchar” el fondo y por consiguiente

“rellenando” la montaña, al tiempo los sprites de la derecha empezarán a limpiarlo. Si la montaña aparece poco a poco entrando en la pantalla, parecerá un sprite de una montaña enorme, cuando en realidad se trata de 6 pequeños sprites.

El videojuego “Nibiru” hace uso de la técnica del “manchado” para dibujar las montañas y otros elementos grandes, en combinación con el comando MAP2SP que luego veremos.



*Fig. 82. Ejemplos de la técnica del manchado*

También utilicé la técnica del manchado en el videojuego “Eridu”, donde enormes montañas se mueven suavemente por la pantalla.



*Fig. 83. Técnica del manchado en “Eridu”*

En el caso de un juego de scroll vertical, si queremos pintar un lago sobre un terreno marrón, haremos lo mismo, unos sprites que van “manchando” el terreno y otros más lejos que lo van “limpiando”, aparentando que se trata de un lago enorme, de una sola pieza.

Solo debes tener una precaución, y es que las naves no sobrevuelen el lago o ¡tu “truco” quedará al descubierto! En caso de que quieras que sea posible que se sobrevuelen el lago, entonces deberás usar sobreescritura en los sprites, tal y como se hace en la fase 2 del videojuego “Nibiru”, donde tu nave y las naves enemigas pueden pasar por encima de grandes rectángulos de color sin destruirlos.



## 14.4 MAP2SP: Scroll basado en un mapa del mundo

Todas las técnicas anteriores son perfectamente válidas para hacer scroll, e incluso compatibles con lo que vamos a ver ahora, que es la técnica fundamental que te va a permitir diseñar un “mapa del mundo” y hacer que tu personaje o tu nave se desplace por él, con tan sólo una línea de código.

Para usar el comando **|MAP2SP** es necesario seleccionar la “**opción de ensamblaje**” 2, la cual nos proporciona 24.8 KB libres para el listado BASIC

La idea es sencilla: crearemos una lista de elementos que conforman el mapa del mundo (hasta 82 elementos a los que llamaremos “elementos de mapa” o “map ítems”). Cada elemento está descrito por las coordenadas Y,X donde se ubica y la dirección de memoria donde se encuentra la imagen del elemento en cuestión (una casa, un árbol, etc.). La imagen asociada a un elemento de mapa podrá tener el tamaño que quieras. Las coordenadas de cada elemento serán un número entero positivo, desde 0 hasta 32000.

Una vez creado el mapa, invocaremos la función:

**|MAP2SP, Yo, Xo**

Esta función analiza la lista de elementos del mundo y determina cuáles de ellos están siendo visualizados si el mundo se observa colocando la esquina inferior de la pantalla en las coordenadas (Yo, Xo). La función transforma en sprites los “map ítems”, ocupando las posiciones de la tabla de sprites de la cero en adelante. Esto puede consumir muchos o pocos sprites, dependiendo de la densidad de map ítems que tengas. En otra invocación posterior a la misma función, los map ítems que ya no están presentes en la escena no consumirán sprites en la tabla, y otros map ítems tomarán el relevo. Esto significa que **la función MAP2SP consume un número de sprites variable e indeterminado, que depende del número de map ítems visibles en pantalla en cada momento**. En el ejemplo siguiente usaría 3 sprites al invocar a MAP2SP en las coordenadas señaladas.

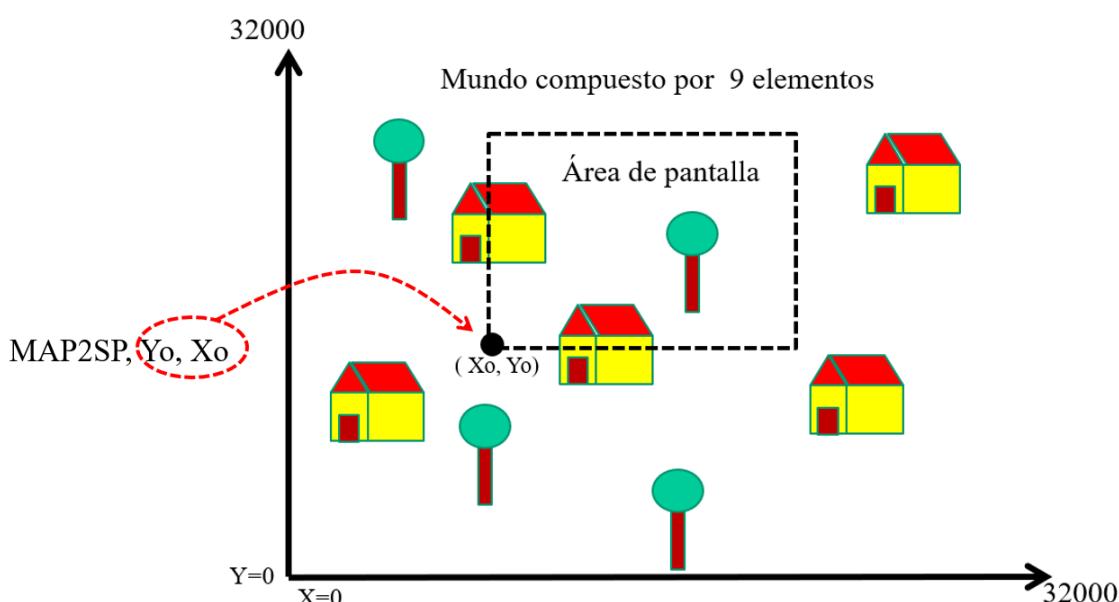


Fig. 84 Mapa del mundo y MAP2SP

Si usas este mecanismo, tu personaje y los enemigos deben usar los sprites desde 31 hacia abajo, de ese modo evitarás posibles choques entre los sprites que usa el mecanismo de scroll y tus personajes. Si por un casual MAP2SP se encuentra con más de 32 items para traducir a sprites, ignorará los que excedan de 32.

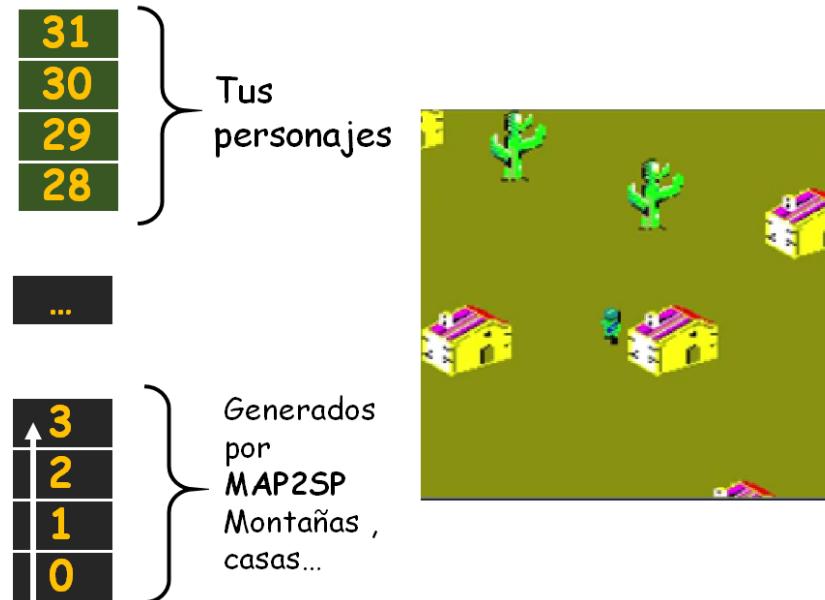


Fig. 85 sprites consumidos por MAP2SP

Debes invocar MAP2SP en cada ciclo de juego o al menos cada vez que modifiques las coordenadas del punto de vista desde donde quieras visualizar el mundo. La ventana “deslizante” con la que cazas las imágenes que se transforman en sprites siempre mide lo que mide la pantalla (80 bytes x 200 líneas) con independencia de como configures el comando SETLIMITS. Es decir, **aunque SETLIMITS establezca un área de pintado muy pequeño, se transforman en sprites todo lo que haya “cazado” la ventana de 80 bytes x 200 líneas**.

Los sprites creados por MAP2SP se crean por defecto con estado 3, es decir, con el flag de impresión activo (PRINTSPALL lo imprime) y con el flag de colisionable activo (COLSP colisionará con el). Si necesitas que los sprites sean creados con otro estado, simplemente debes invocar una vez al comando MAP2SP con un solo parámetro indicando el estado con el que se deben crear los sprites. Con una única invocación de este tipo, queda configurado el comando para las siguientes invocaciones con coordenadas.

|MAP2SP, <status>

Ejemplo:

|MAP2SP, 1 : REM esto configura al comando MAP2SP para que se impriman pero no sean colisionables

Una vez aclarado el concepto vamos a revisar en detalle cómo se especifica el mapa del mundo y un ejemplo de uso de la función MAP2SP.

#### 14.4.1 Mapa del mundo (Map Table)

La tabla donde daremos de alta todos los elementos del mapa se llama MAP\_TABLE y se especifica en un fichero .asm llamado **map\_table\_tujuego.asm**

Esta tabla contiene los ítems que definen las imágenes del mapa del mundo para tus juegos con scroll. La tabla se ensambla en la misma dirección de memoria que el LAYOUT, es decir, en la dirección 42040. Esto significa que no se pueden usar simultáneamente el layout y el mapa del mundo, pero no es un problema ya que un juego con scroll no va a usar el layout y viceversa. Además, la restricción es únicamente que no se pueden usar a la vez, pero un juego podría tener una fase en la que usa el layout y otra en la que hace scroll basado en mapa del mundo.

La tabla de entradas del mapa del mundo comienza con 3 parámetros globales (que ocupan 5 bytes en total) y una lista de "map items", los cuales están descritos por 3 parámetros cada uno (x, y, dirección de imagen)

La lista puede contener hasta 82 items, pero el número de ítems se puede limitar con uno de los parámetros globales. La lista, como máximo, ocupa los 5 bytes iniciales + 82 items x 6 bytes = 5+492=497 bytes. Si metiésemos un ítem mas, superaríamos los 500Bytes reservados para el mapa (que son los mismos reservados para el Layout).

La tabla comienza con 3 parámetros:

- El alto máximo de cualquier map ítem
- Ancho máximo de cualquier map ítem (debe expresarse como un número negativo)
- Número de ítems (como máximo será 82)

Los dos primeros parámetros son importantes para chequear cuando un sprite puede estar parcialmente apareciendo en pantalla, ya que la función MAP2SP no conoce ni averigua el ancho ni el alto de cada imagen. Tan solo conoce donde está situado el map ítem y suponiendo el alto y ancho máximos, averigua si ese ítem puede estar entrando en la pantalla. En caso de que así sea, se crea un sprite a partir del map ítem. Si esos dos parámetros se ponen a cero, será necesario que la esquina superior izquierda del map ítem esté dentro de la pantalla para que dicho ítem sea transformado en un sprite.

Cada ítem es una tupla de 3 parámetros precedidos por el nemónico "DW":

**DW Y, X, <imagen>**

Veamos un ejemplo del fichero llamado **map\_table\_tujuego.asm**

```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
dw -40; maximo ancho de un sprite por si se cuela por la izquierda
(numero negativo)
db 82; numero de elementos del mapa.como mucho debe ser 82

; a partir de aqui comienzan los items
dw 100,10,CASA; 1
dw 50,-10,CACTUS;2
dw 210,0,CASA;3
```

```

dw 200,20,CACTUS;4
dw 100,40,CASA;5
dw 160,60,CASA;6
dw 70,70,CASA;7
dw 175,40,CACTUS;8
dw 10,50,CASA;9
dw 250,50,CASA;10
dw 260,70,CASA;11
dw 290,60,CACTUS;12
dw 180,90,CASA;13
dw 60,100,CASA;14

```

...

Para diseñar tu mundo te recomiendo que cojas un cuaderno a cuadros y vayas dibujando sobre él los elementos que quieras que tenga tu mundo. Cada cuadradito del cuaderno puede representar una cantidad fija como 8 píxeles o 25 píxeles. El caso es que debes tomarte tu tiempo en dibujar el mundo que deseas y el modo en que se va a recorrer. Por ejemplo, hay juegos tipo “Gauntlet” multidireccionales y otros de scroll vertical como el comando. Tú eliges, pero en cualquier caso hazlo con tiempo y paciencia y el resultado valdrá la pena.

Cada fase de tu juego puede ser un mapa. En 8BP puedes cambiar el mapa cuando quieras usando funciones POKE. Yo normalmente utilizo 1KB fuera del espacio de memoria usado por 8BP, por ejemplo, desde la 23000 hasta la 24000, para almacenar todas las fases (mapas) del juego y cada vez que entro en una fase cargo en la dirección 42040 el mapa correspondiente haciendo PEEK y POKE. Es decir, que mi fichero de mapa lo construyo en la dirección 23000 y ocupa 1Kbyte, dejando 23 KB para mi programa BASIC. Para que esta información de mapas no sea machacada por el basic, tendré que hacer un MEMORY 22999 al principio del juego.

#### **14.4.2      Uso de la función MAP2SP**

Ahora vamos a ver un ejemplo de uso de esta función. Básicamente hay que invocarla una vez en cada ciclo de juego con las nuevas coordenadas del origen desde donde se observa el mundo.

La función creará un numero de sprites variable desde el sprite 0 en adelante y al crearlos lo va a hacer con sus coordenadas de pantalla adaptadas. Es decir, aunque un map ítem tenga una coordenada x=100, si el origen móvil lo ubicamos en la posición x=90 entonces ese sprite será creado con la coordenada de pantalla x'=x-90=10. La coordenada en el eje Y tendrá en cuenta que el eje Y en el Amstrad crece hacia abajo, mientras que el mapa del mundo crece hacia arriba. Por ello la coordenada Y es adaptada usando la ecuación  $Y' = 200 - (Y - Y_{orig})$ . Pero no te preocupes, esta adaptación ya la hace la función MAP2SP. Tu solo tienes que ir cambiando el origen móvil desde donde se debe visualizar el mapa del mundo.

En este mini juego se ha realizado un mundo compuesto de casas y cactus y nuestro personaje camina entre los elementos. En este ejemplo, en caso de colisión (detectado con COLSPALL), el personaje no podrá continuar. En un juego de aviones en el que los map ítems sean “sobrevolables”, podríamos parametrizar la colisión para que solo se detecten colisiones con enemigos y disparos y no con elementos de fondo, usando COLSP, 32, <sprite inicial>, <sprite\_final>.

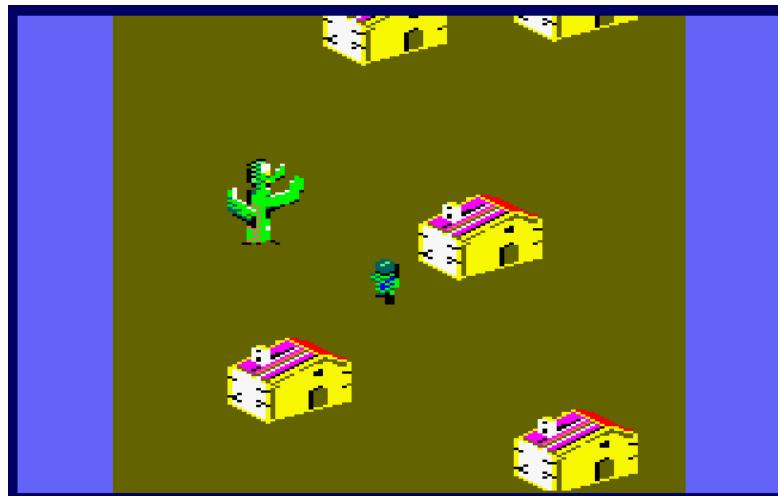


Fig. 86 Mini juego con scroll inspirado en “commando”

Ahora veamos el listado, como ves, es muy pequeño, pero lo tiene todo: scroll multidireccional, lectura de teclado, cambio de secuencias de animación del personaje, detección de colisión, música...

**IMPORTANTE:** fíjate en el comando MEMORY. Hemos usado la opción 2 de ensamblaje, ideal para juegos con scroll que nos deja casi 25 KB libres

```

10 MEMORY 24799
20 MODE 0
30 ON BREAK GOSUB 280
40 CALL &6B78
50 DEFINT a-z
60 INK 0,12
70 FOR y=0 TO 400 STEP 2
80 PLOT 0,y,10:DRAW 78,y
90 PLOT 640-80,y,10:DRAW 640,y
100 NEXT
110 x=0:y=0
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1: ' direccion inicial hacia arriba
140 |locatesp,31,100,36
150 |MUSIC,0,1,5
160 |SETLIMITS,10,70,0,199: |PRINTSPALL,0,1,0
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0: REM colision en cuanto hay un mnimo solape
190 'comienza ciclo de juego
200 IF INKEY(27)=0 THEN x=x+1:IF dir<>3 THEN dir=3:|SETUPSP,31,7,3:
GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0:ELSE IF dir<>4 THEN
dir=4:|SETUPSP,31,7,4
220 IF INKEY(67)=0 THEN y=y+2:IF x=xa AND dir <> 1 THEN
dir=1:|SETUPSP,31,7,1: GOTO 240
230 IF INKEY(69)=0 THEN y=y-2:IF y<0 THEN y=0:ELSE IF x=xa AND dir <>2
THEN dir=2:|SETUPSP,31,7,2:
240 IF xa=x AND ya=y THEN dir=0 ELSE |ANIMA,31
250 |MAP2SP,y,x:|COLSPALL: IF col<32 THEN x=xa:y=ya:|MAP2SP,y,x ELSE
xa=x:ya=y
260 |PRINTSPALL

```

```

270 GOTO 200
280 |MUSIC:MODE 1: INK 0,0: PEN 1

```

Vamos a ver ahora otro ejemplo de scroll horizontal, donde se ha conseguido un efecto interesante de mapa del mundo “infinito”, haciendo que el final del mapa sea igual al principio y provocando un salto brusco cuando Xo llega a un determinado valor. De hecho, este mapa del mundo solo tiene 13 elementos



*Fig. 87 Mapa del mundo “infinito”*

Este es el mapa que se ha utilizado

```

_MAP_TABLE
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
dw -18; ancho maximo de cualquier map item. debe expresarse como
numero negativo
db 13; numero de elementos del mapa. como mucho debe ser 82

; a partir de aqui comienzan los items
dw 36,80,MONTUP; 1
dw 48,100,MONTUP;2
dw 60,120,MONTUP;3
dw 72,130,MONTUP;4
dw 72,140,MONTDW;5
dw 60,160,MONTH;6
dw 60,180,MONTDW;7
dw 48,190,MONTDW;8
; aqui repito elementos para encajar con la posicion 100
dw 48,210,MONTUP;9
dw 60,230,MONTUP;10
dw 72,240,MONTUP;11
dw 72,250,MONTDW;12
dw 60,270,MONTH;13
;-----

```

Y este el programa BASIC, donde he destacado la línea en la que el mundo vuelve atrás sin que el jugador note nada.

```

10 MEMORY 24799
11 FOR dir=42540 TO 42618 STEP 2: POKE dir,20+RND*110:POKE
dir+1,RND*80:NEXT
20 MODE 0

```

```

30 ON BREAK GOSUB 280
40 CALL &6B78
50 DEFINT a-z
51 INK 0,0
52 |MUSIC,0,0,5
110 xo=0:yo=0
111 x=36:y=100
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1: ' direccion inicial hacia arriba
140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,176: |PRINTSPALL,0,1,0
161 LOCATE 1,23 :PEN 1: PRINT "VIDAS:3 MISILES:250"
162 LOCATE 1,1:PRINT " DEMO SCROLL 8BP"
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0: REM colision en cuanto hay un mnimo solape
190 'comienza ciclo de juego
200 IF INKEY(27)=0 THEN x=x+1: GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0
220 IF INKEY(69)=0 THEN y=y+2: GOTO 240
230 IF INKEY(67)=0 THEN y=y-2:IF y<0 THEN y=0
240 IF xa=x AND ya=y THEN dir=0 ELSE |ANIMA,31
250 |MAP2SP,yo,xo:|COLSPALL:IF col<32 THEN END
260 |PRINTSPALL
261 ciclo=ciclo +1: IF ciclo=2 THEN |STARS,0,5,2,0,-1:ciclo=0
262 xo=xo+1:IF xo=210 THEN xo=100
263 |LOCATESP,31,y,x
270 GOTO 200
280 |MUSIC:MODE 1: INK 0,0:PEN 1

```

#### 14.4.3 Ejemplo de fichero de fases

Si quieras tener varias fases en un juego con scroll, como he comentado antes puedes tenerlas precargadas en un área de memoria. Por ejemplo, puedes ensamblar las fases en la dirección 23000 y así dispones de 1000 bytes para almacenar varios mapas del mundo, ya que 8BP comienza en la dirección 24000. En ese caso tu juego tendrá que empezar con un MEMORY 22999

El videojuego “**Nibiru**” lo hace así, aunque fue creado cuando 8BP empezaba en la dirección 26000 (fue creado con la 8BP v26), por lo que almacena el mapa a partir de la 25000. Para cargar una fase simplemente lee la zona donde se ha ensamblado cada fase y la escribe sobre la dirección donde debe encontrarse la tabla del mundo antes de empezar a jugar en esa fase. En estas tres líneas de BASIC se muestra cómo se copia la fase 1 del juego (&61a8 = 25000)

```

310 ' pokes del mapa del mundo
320 dirmap!=42040:FOR i!=&61A8 TO &620D
330 dato=PEEK(i!):POKE dirmap!,dato:dirmap!=dirmap!+1
340 NEXT

```

Existe una forma más rápida de cargar las fases, mediante el comando |UMAP que más adelante explicaré, pero este bucle FOR con POKEs es perfectamente válido.

Por último, a continuación, te muestro el fichero de fases del juego “**Nibiru**”, que ensamblamos en la dirección 25000 (recordemos que la versión de 8bp con la que fue creado “**Nibiru**” empezaba en la 26000, por lo que desde la 25000 hasta la 26000 había 1000 bytes libres. Con la versión actual de 8BP ensamblaríamos las fases sin llegar a la dirección 24800)

```

org 25000
;FASE1
;=====
;START_FASE1
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -18; ancho maximo de cualquier map item. debe expresarse como numero
negativo
db 16; num items
dw 36,82,MONTUP; 1
dw 48,104,MONTUP;2
dw 60,126,MONTUP;3
dw 72,138,MONTUP;4
dw 72,150,MONTDW;5
dw 60,172,MONTH;6
dw 60,194,MONTDW;7
dw 48,206,MONTDW;8
;aqui repito elementos para encajar con la posicion 100
dw 48,228,MONTUP;9
dw 60,250,MONTUP;10
dw 72,262,MONTUP;11
dw 72,274,MONTDW;12
dw 60,296,MONTH;13
dw 60,320,MONTDW;14
dw 48,350,MONTDW;15
dw 36,380,MONTDW;16
-END_FASE1
;=====
;FASE2
;=====
;START_FASE2
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -6; ancho maximo de cualquier map item. debe expresarse como numero
negativo
db 15
dw 128,80,PLACA2_L_OV
dw 128,110,PLACA2_R_OV
dw 192,116,PLACA2_L_OV
dw 192,126,PLACA2_R_OV
dw 92,130,PLACA_L_OV
dw 92,150,PLACA_R_OV
dw 124,151,PLACA_L_OV
dw 124,171,PLACA_R_OV
dw 128,200,PLACA2_L_OV
dw 128,210,PLACA2_R_OV
dw 92,220,PLACA2_L_OV
dw 92,230,PLACA2_R_OV
dw 164,240,PLACA2_L_OV
dw 164,260,PLACA2_R_OV
dw 156,254,CUPULA2_OV

```

```

-END_FASE2
=====
;FASE3
=====
_START_FASE3
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -80; ancho maximo de cualquier map item. debe expresarse como numero
negativo
db 4
dw 40,0,MAR; 1
dw 40,80,MAR; 2
dw 189,0,NUBES; 2
dw 189,80,NUBES; 2
-END_FASE3

```

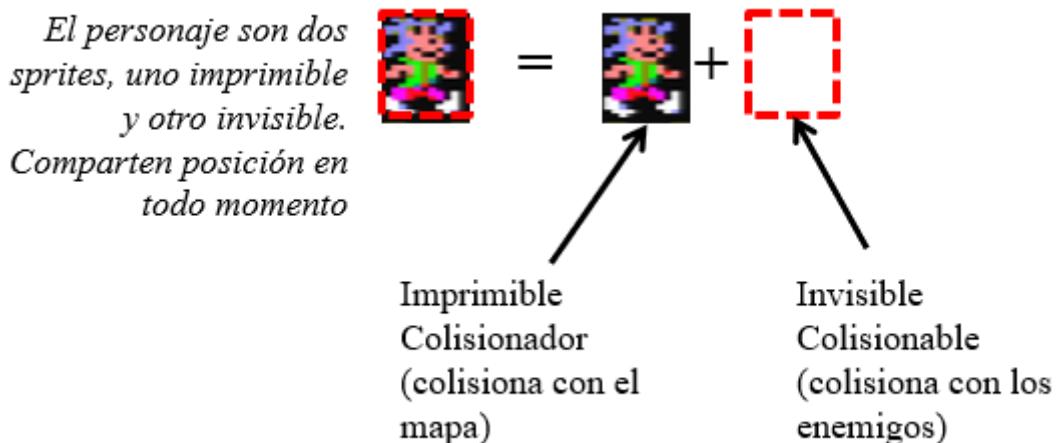
#### 14.4.4 Colisión de enemigos con mapa

Es posible que quieras hacer un juego en el que tu personaje colisione con el mapa (usando COLSPALL) y que por lo tanto sea colisionador. Supongamos que tienes esto:

- Tu personaje: colisionador
- Los elementos del mapa: colisionables
- Tu disparo: colisionador
- Los enemigos: colisionables

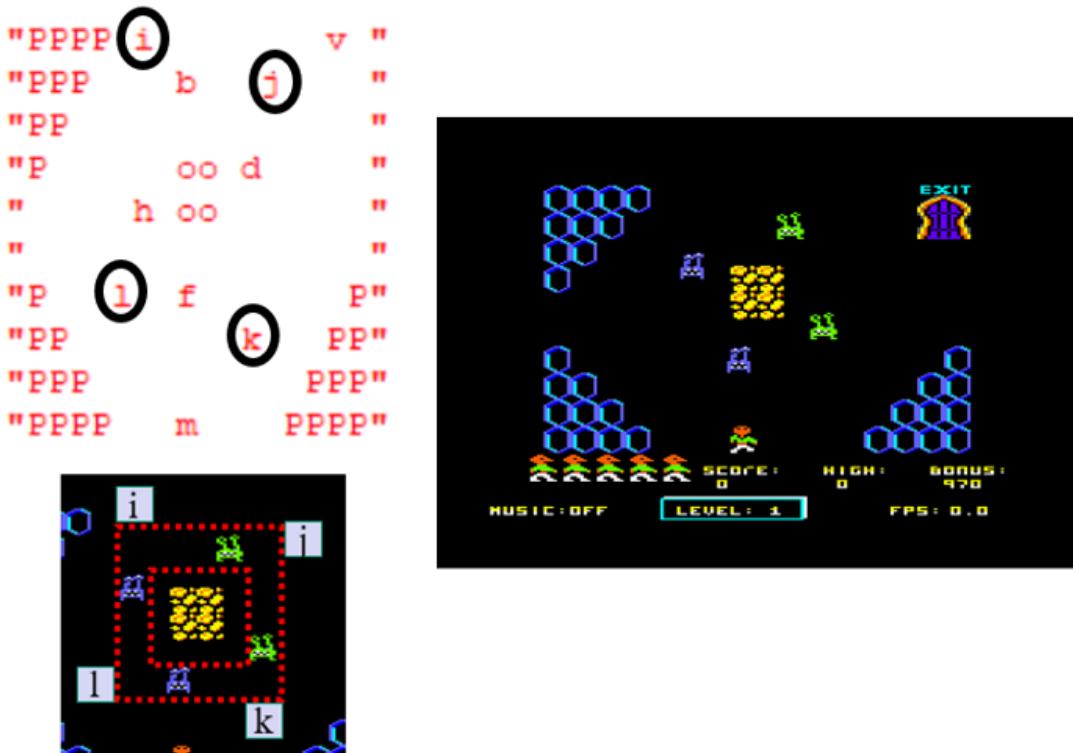
Ahora bien, si tus enemigos son colisionables, entonces no pueden colisionar con el mapa, y quizás te interesa, para hacer un juego tipo gauntlet, por ejemplo. La solución pasa por hacer que los enemigos sean colisionadores. Pero claro, en ese caso ya no te pueden matar. Y también hay un problema con el disparo que es colisionador también.

La solución a este problema se basa en usar un sencillo truco basado en sprites “invisibles” sin capacidad de impresión, que no gastan CPU en imprimirse pero que están ahí. **Tu personaje gastará dos sprites:** uno colisionador imprimible y otro colisionable no imprimible, pero del mismo tamaño. Con los disparos haremos lo mismo, así podrán colisionar con los enemigos y con el mapa.



Un truco similar se utiliza en el juego “**Frogger eterno**”. Mediante el uso de sprites invisibles se logra que la rana muera al caer al río (consulta el apartado donde se explica el comando COLSPALL). Este sencillo truco puede aplicarse también a los disparos del

personaje. Los sprites “invisibles” son muy útiles. En el juego “Happy Monty” se usan para hacer cambiar de dirección a los enemigos, de modo que cuando un enemigo choca con un Sprite invisible, cambia su dirección. Ello permitió no requerir definir muchísimas rutas adaptadas a cada pantalla, sino tan solo colocar en cada nivel una serie de sprites invisibles que permitían alterar las trayectorias de los enemigos. (ver documento “making off” de **Happy Monty**)



#### 14.4.5 Imágenes de fondo en tu scroll

Las imágenes de fondo están pensadas para juegos con scroll y son una característica que proporciona 8BP a partir de la versión V42. Es un tipo de impresión transparente en el que los sprites que constituyen el fondo (el mapa del mundo) pueden imprimirse por debajo de tu personaje y los enemigos sin causar parpadeos.

Las imágenes de fondo siempre que se asignen a un Sprite se imprimen con esta transparencia especial y da igual si el Sprite tiene asignado o no el flag de transparencia en su byte de estado. Consulta el capítulo 8 para aprender a usarlas.

**IMPORTANTE:** en el mapa del mundo puedes combinar imágenes normales con “imágenes de fondo” (apartado 8.5). Las imágenes de fondo siempre tienen transparencia. El flag de transparencia que uses en **|MAP2SP, <status>** sólo aplicará a las imágenes normales

**IMPORTANTE:** las imágenes de fondo son costosas de imprimir, y si las flipeas aún son más costosas. Si tu juego tiene scroll, trata de usarlas para elementos sobre los que vayan a sobrevolar tu personaje o los enemigos. Por ejemplo, para acelerar el scroll usa

imágenes normales en casas o rocas que aparezcan en los laterales de tu scroll, que no van a ser frecuentemente solapados por los sprites

## 14.5 Scroll Parallax

Vamos a ver cómo se puede hacer un scroll parallax, es decir, con varios “planos” a diferente velocidad. Quizás se te ocurra a ti otra forma de hacerlo, esto es solo una posible manera, empleada en el juego “**Nibiru**”.

Lo primero que debes saber es que es mucho más rápido imprimir un sprite horizontal muy largo que muchos sprites pequeños. Esto es debido a las operaciones que hay que realizar tras terminar de pintar cada scanline de un sprite. Por el mismo motivo es mucho más rápido imprimir un sprite horizontal muy grande que un sprite vertical del mismo tamaño.

Colocaremos en el mapa del mundo dos sprites gigantes para hacer el agua. Yo hice uno de 160 x8 de modo que puse dos para que cuando se desplazase empezase a aparecer el siguiente. Al recorrer toda la pantalla, la función MAP2SP se vuelve a colocar en x=0 y todo se repite indefinidamente. En el mapa del mundo también puse dos sprites para las nubes.

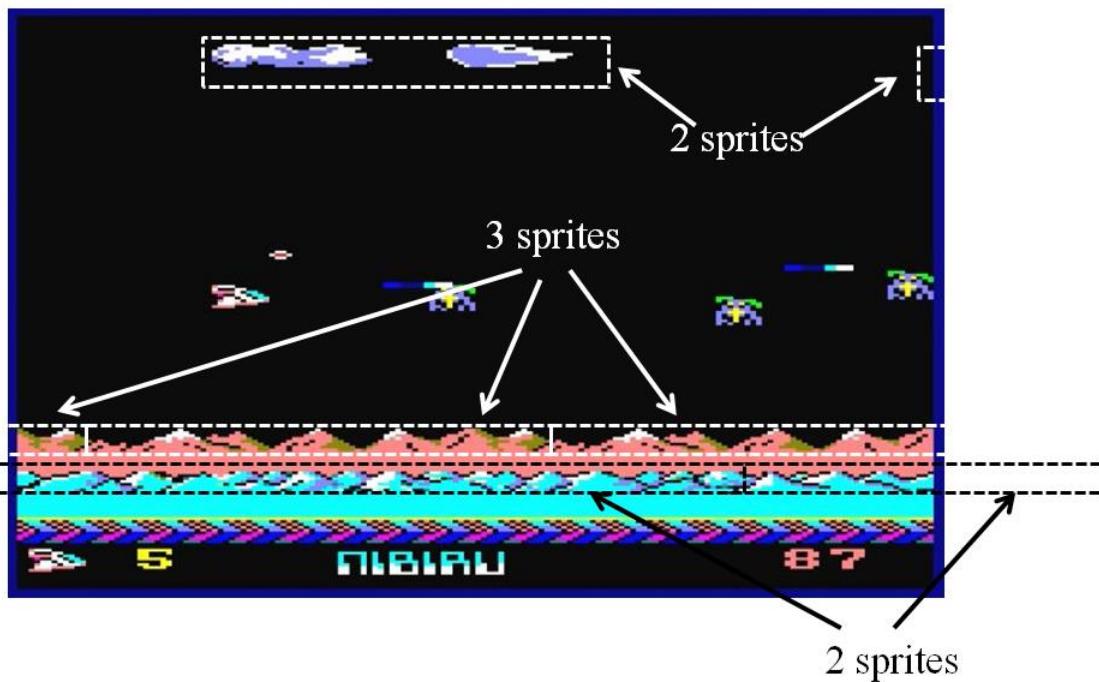


Fig. 88 scroll parallax

Para las montañas usé 3 sprites normales, fuera del mapa del mundo. Les di movimiento automático en su flag de status y les hice moverse hacia la izquierda. Pero en los ciclos impares les desactivo tanto el flag de impresión como el flag de movimiento automático, de modo que solo se mueven e imprimen uno de cada dos ciclos de juego, logrando una velocidad mitad que la que lleva el agua. Los sprites de las montañas son el 16,17 y 18 y con estos pokes se actúa sobre su byte de status.

```
mc=ciclo AND 1: IF mc THEN POKE 27256,0: POKE 27272,0: POKE 27288,0  
ELSE POKE 27256,11: POKE 27272,11: POKE 27288,11
```

En el ejemplo “mc” es la variable que determina si el ciclo es par o impar.

## **14.6 Actualización dinámica del mapa: |UMAP**

Puede que en nuestro juego necesitemos un mapa de mas de 82 elementos. O bien simplemente queremos que el comando |MAP2SP se ejecute mas rápido usando un mapa más pequeño que actualicemos periódicamente. ¡O bien queremos ambas cosas!

Para ello desde la versión 32 de 8BP, existe el comando **|UMAP** (abreviatura de “UPDATE MAP”). Este comando actualiza el mapa con información ubicada en otra zona de memoria donde tengamos un mapa mayor. El comando hace que se reconstruya el mapa por completo, incluyendo solo aquellos ítems que cumplan unos determinados rangos de coordenadas X, Y (todos los parámetros son números de 16 bits)

```
|UMAP, <map_ini>, <map_fin>, <y_ini>, <y_fin>, <x_ini>, <x_fin>
```

UMAP no es un simple copiado de elementos. Es un copiado “selectivo”. Por ejemplo, si tenemos un mapa ubicado en la dirección 22500 que ocupa 1500bytes y queremos que se actualice el mapa con las coordenadas de nuestro personaje, con margen suficiente para avanzar en la coordenada Y hasta 100 líneas y en coordenada x hasta 20 bytes en todas direcciones:

```
|UMAP, 22500,23999, y-100, y+100, x-20, x+20
```

Este comando chequeara las coordenadas de los ítems localizados en el mapa de la dirección 22500 y si se encuentran dentro de los márgenes de X, Y que hemos puesto, se copiaran en la zona de memoria que 8BP usa para el comando |MAP2SP, es decir, los copiará a partir de la dirección 42040. Eso sí, tan solo copiará los que cumplen la condición. Al ser menos items, el comando |MAP2SP se ejecutará más rápido pues tendrá que leer y comprobar si se encuentran dentro de la pantalla menos items.

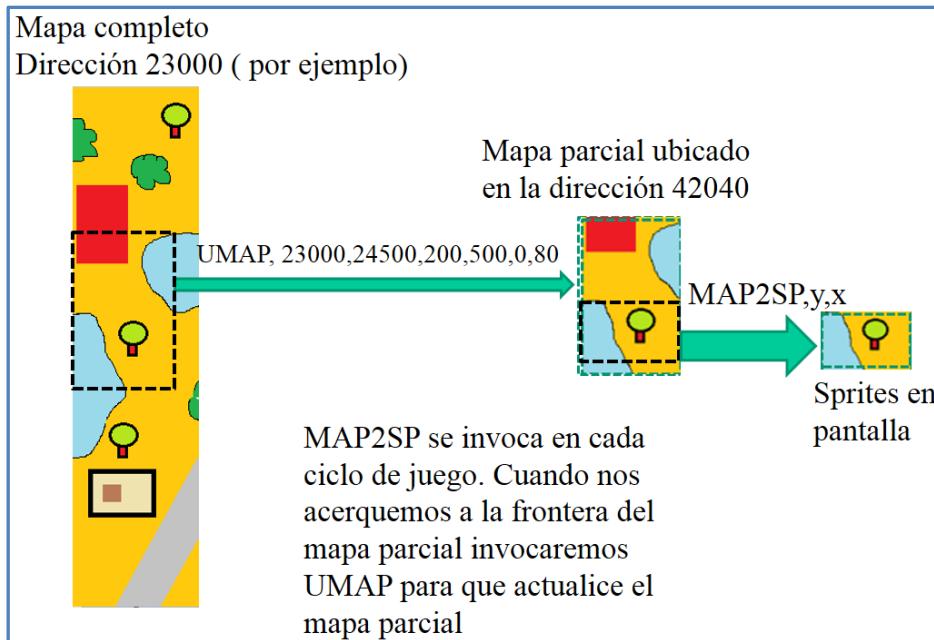


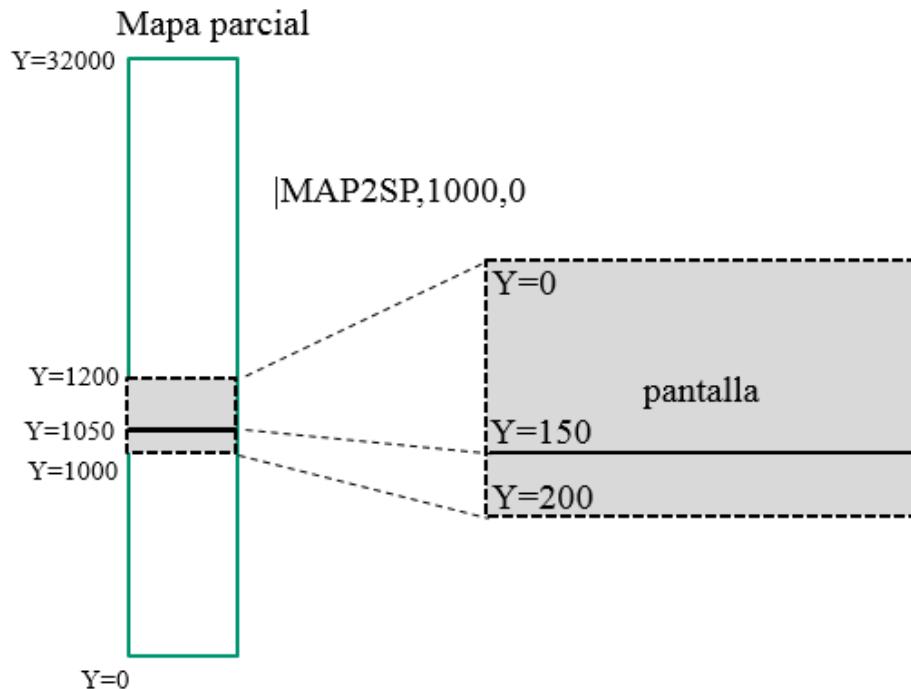
Fig. 89 UMAP

Periódicamente (no en cada ciclo de juego) podemos actualizar el mapa con UMAP y así podemos crear mundos muy grandes con muchos elementos y a la vez conseguiremos más velocidad en MAP2SP. El comando UMAP es muy rápido, pero invocarlo en cada ciclo de juego no tiene sentido ya que MAP2SP puede trabajar con un mapa mucho más grande que lo que cabe en pantalla y podemos invocar a |UMAP únicamente cuando necesitemos zonas del mapa original que no están presentes en el mapa parcial. Yo lo he usado en el juego de carreras de coches “**3D Racing one**”, pues el mapa del circuito era muy grande (excedía de 82 elementos) y lo que hice fue invocar |UMAP periódicamente a medida que el coche avanza.

En la dirección del mapa completo (en el ejemplo la 22500), únicamente tendremos una lista de ítems. **No tendremos los 3 datos iniciales** que contiene el mapa de la 42040 (me refiero al alto máximo de cualquier map ítem, el ancho máximo de cualquier map ítem y el número de ítems). El número de ítems lo actualiza |UMAP (según cuantos ítems cumplan los márgenes impuestos). Los otros dos parámetros los fijarás tu a tu gusto en el fichero “map\_table\_tujuego.asm”.

El comando |UMAP mete los ítems en el mapa parcial en un orden pensado para que el mapa parcial quede ordenado según la coordenada Y de pantalla. Lo normal es que tu edites tu mapa global en un fichero llamado “misupermapa.asm” o algo así. Y lo ensambles en la 22500 (por ejemplo). En dicho fichero escribirás uno por uno los ítems de tu mapa, en orden ascendente de coordenada Y. Pues bien, para conseguir que los sprites salgan ya ordenados por coordenada Y (en orden ascendente de coordenada de pantalla), el comando |UMAP los lee desde el final hasta el principio. De ese modo los sprites que se vayan generando posteriormente con |MAP2SP estarán ordenados por coordenada Y de pantalla. Recuerda que la pantalla usa un sistema de coordenadas inverso respecto el mapa, es decir, la coordenada 150 de la pantalla es la coordenada 50 del mapa (en el caso |MAP2SP, 0, 0). Si no entiendes esto muy bien, no te preocupes. Es algo del funcionamiento interno de |UMAP y |MAP2SP simplemente para hacerlo más eficiente. En la siguiente figura he representado el mapa y la pantalla del CPC. Como ves

el mapa puede ser muy grande pero además sus coordenadas crecen hacia arriba mientras que las coordenadas de pantalla crecen hacia abajo.



*Fig. 90 MAPA y Pantalla*

#### **14.7 Animación y scroll por tintas: comando RINK**

Existen juegos que requieren mover grandes bloques de memoria de pantalla para dar sensación de movimiento, como es el caso de las franjas laterales de los juegos de carreras o bloques grandes de ladrillos o tierra. El comando MAP2SP permite hacer todo eso, pero la velocidad no es trepidante porque gasta mucha CPU moviendo sprites. La animación por tintas es el complemento perfecto en estos casos.

En ordenadores como el AMSTRAD con su potente paleta de 16 colores simultáneos, muchos juegos hacen uso de la animación por tintas. Un claro ejemplo son algunos juegos de coches cuyas franjas laterales de carretera se prestan a este tipo de animación.



*Fig. 91 Franjas animadas por tintas*

La animación por tintas consiste en definir un conjunto de tintas sobre las que se va a hacer rotar un conjunto de colores. Vamos a ver un ejemplo con unas franjas blancas/grises y 8 tintas:

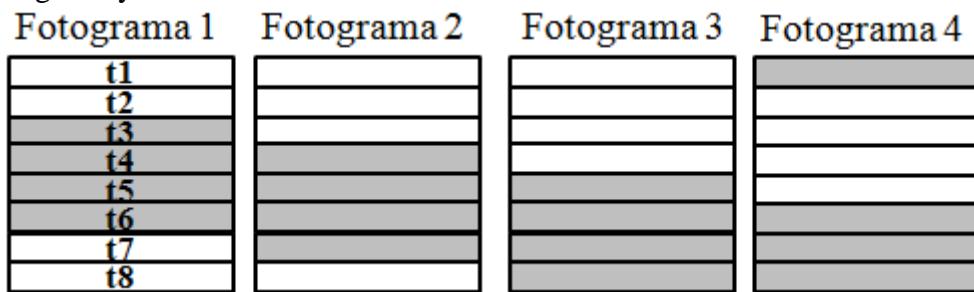


Fig. 92 Animación por tintas

Básicamente para dar sensación de movimiento lo que hay que hacer primeramente es asignar los colores a las tintas que van a rotar. En este caso los colores blanco (26) y gris (13) se asignan a las tintas t1..t8. Vamos a suponer que la tinta t1 es la 8, de modo que la tinta t8 será la 15. El resto de tintas (0 a 7) las usaremos para los sprites. En cada instante de tiempo habrá que reasignar los valores de las 8 tintas para dar la sensación de rotación. Esto es lo que hace el comando |RINK (abreviatura de “Rotate INK”).

RINK te permite definir un patrón de colores a rotar sobre un conjunto de tintas. Una tinta no es un color. Una tinta es un identificador en el rango [0..15] que identifica a un color del rango [0..26]. Para definir el patrón de colores a rotar, usaremos el comando RINK del siguiente modo:

**RINK, <tinta\_inicial>, <color1>,<color2>, <color3>, ... ,<colorN>**

Esto indica que van a rotar N tintas comenzando por la tinta inicial usando el patrón de colores que se indica. Ten en cuenta que, si usas muchas tintas para hacer una animación, te quedarán menos tintas para tus sprites.

Una vez establecido el patrón, podemos rotar las tintas más o menos deprisa con

**RINK, <step>**

El valor del step es el número de desplazamientos que sufrirán las tintas del patrón y por consiguiente un valor superior proporciona un efecto de desplazamiento a mayor velocidad

Recomendación: debido al uso de interrupciones RINK produce “parones” en algunos casos cuando se usa a la vez que el comando |MUSIC a velocidad 6. En caso de querer usar ambos a la vez sin que haya interferencias, usa otra velocidad para la música (puedes usar velocidad 5 o 7, ambas te funcionarán bien).

### 14.7.1 Carreras de coches 2D

El ejemplo del juego de coches lo tienes en el siguiente listado. El sonido del motor intenta causar la sensación de la aceleración. Para los carteles que aparecen a los lados se ha usado un sprite con movimiento relativo, el cual se mueve a la misma velocidad que el step de las franjas. El patrón de tintas se define en la línea 100

|RINK,1,3,3,3,3,24,24,24,24: rem franjas amarillas y rojas

```
1 MEMORY 24999
2 CALL &6B78
3 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:|AUTOALL,0:|PRINTSPALL,0,0,0
4 |SETUPSP,31,9,16: |SETUPSP,31,0,1: vy=0
10 MODE 0
20 DEFINT a-z
31 |LOCATESP,31,160,40: x=40
32 |SETUPSP,30,9,17: |SETUPSP,30,0,17: |LOCATESP,30,-20,10
40 CALL &BC02:'default paleta
50 GOSUB 430
60 INK 0,13
70 INK 14,10
80 linestinta=3
90 rangotintas=8
91 ' establecimiento de patron de tintas
100 |RINK,1,3,3,3,3,24,24,24,24: rem franjas amarillas y rojas
101 |RINK,0
110 y=400
120 ' PAINT ROAD -----
121 tini=1
130 FOR franjas=1 TO 10
140 FOR t=tini TO rangotintas+tini-1
150 FOR j=1 TO linestinta
160 PLOT 0,y,14:DRAW 136,y
170 PLOT 140,y,t:DRAW 160,y
180 PLOT 480,y,t:DRAW 500,y
190 PLOT 504,y,14:DRAW 640,y
200 y=y-2
210 NEXT j
220 NEXT
240 NEXT franjas
250 saltob=-16:xc=65: cosa=0
270 REM ciclo de juego -----
293 IF saltob=-16 THEN 296
294 IF saltob>0 THEN cosa=-salto ELSE cosa=cosa-1
295 IF cosa<0 THEN |RINK,cosa:vy=3*cosa:posv=posv-3*cosa:|MOVERALL,-vy,0:IF
saltob <=0 THEN cosa=2-saltob
296 |PRINTSPALL
351 ciclo=ciclo+1:IF posv>240 THEN posv=-30:|LOCATESP,30,posv,xc:IF xc=10 THEN
xc=65 ELSE xc=10
361 IF INKEY(27)=0 THEN IF x<52 THEN x=x+1:POKE 27499,x:GOTO 370
362 IF INKEY(34)=0 THEN IF x>21 THEN x=x-1:POKE 27499,x
370 IF INKEY(67)=0 THEN IF saltob<16 THEN saltob=saltob+1:salto=saltob/4
380 IF INKEY(69)=0 THEN IF saltob>-16 THEN saltob=saltob-1:salto=saltob/4
390 SOUND 1,6000/(salto+17),1,15
400 GOTO 270
421 REM PALETA
430 INK 0 , 12
440 INK 1 , 5
450 INK 2 , 20
460 INK 3 , 6
470 INK 4 , 26
480 INK 5 , 0
490 INK 6 , 2
500 INK 7 , 8
510 INK 8 , 10
```

```

520 INK  9 , 12
530 INK  10 , 6
540 INK  11 , 15
550 INK  12 , 0
560 INK  13 , 23
570 INK  14 , 0
580 INK  15 , 11
590 RETURN

```

### 14.7.2 Scroll de Ladrillos

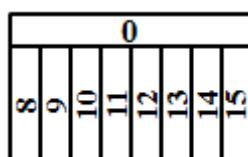
En este ejemplo vamos a combinar el uso de la animación por tintas con el scroll basado en |MAP2SP. Mediante la animación por tintas moveremos un patrón de ladrillos que habremos dibujado con repetición de un sprite mientras que el castillo y el árbol serán movidos por |MAP2SP.

El movimiento de los ladrillos implicaría un trabajo inmenso si se realizase por CPU, de modo que esta técnica permite “lo imposible”. Es una técnica muy potente si la empleas con ingenio.



*Fig. 93 Scroll usando animación por tintas y MAP2SP a la vez*

El ladrillo empleado es en realidad un sprite de 8 tintas, con el diseño que se muestra a continuación:



Y el patrón de tintas es 0,6,3,3,3,3,3,3 . para crearlo simplemente se ejecuta el comando: |RINK,8,0,6,3,3,3,3,3,3

El personaje (sprite 31) permanece en el centro de la pantalla, pudiendo saltar y moverse en ambas direcciones. Para sincronizar el movimiento de tintas y el MAP2SP, se establece un step=2 para el comando |RINK, ya que un byte son dos pixeles y MAP2SP mueve a nivel de byte.

Es interesante observar como el pájaro (sprite 30) debe ser afectado también por el movimiento ya que a su velocidad hay que sumar o restar la del personaje.

En cuanto al color, puesto que se usa un patrón de 8 colores y además se usa sobreescritura, nos quedan 2 colores para el fondo (castillo, ramas) y 3 colores para los sprites (personaje y pájaro). Es sencillo modificar el programa para usar un patrón de rotación de 4 colores y de ese modo disponer de 5 colores para sprites, más razonable.

A continuación, tienes el listado completo.

```

10 MEMORY 24999
11 ON BREAK GOSUB 5000
20 CALL &6B78
30 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:|AUTOALL,1:|PRINTSPALL,0,1,0
40 |SETUPSP,31,7,1:|SETUPSP,31,0,65:|LOCATESP,31,130,36: 'personaje
50 |SETUPSP,30,7,7:|SETUPSP,30,0,157:|LOCATESP,30,50,80:
|SETUPSP,30,15,2: 'bird
60 MODE 0:DEFINT a-z
80 CALL &BC02:'default paleta
90 BORDER 10
100 INK 6,15:INK 7,15
110 INK 4,26:INK 5,26
120 INK 2,0:INK 3,0
130 INK 1,14
140 ' establecimiento de patron de tintas
170 tini=8:|RINK,tini,0,6,3,3,3,3,3,3
200 |RINK,0
210 LOCATE 1,1:pen 4:PRINT "DEMO |RINK y |MAP2SP"
220 ' PAINT wall -----
230 |SETUPSP,29,9,23:'ladrillo
231 y=152
232 FOR fila=1 TO 6
240 FOR ladrillo=xini to 42 step 2:|PRINTSP,29,y,ladrillo*2:next
241 xini=(xini-1) mod 2: y=y+8
242 next
390 dir=1:x=0:xp=80: ciclo=40: stepy=2
400 |MUSIC,0,0,7
409 ' ciclo de juego -----
410 |AUTOALL:|PRINTSPALL
450 IF INKEY(27)=0 THEN |RINK, stepy:x=x+1:|MAP2SP,0,x:|MOVER,30,0,-
1:if jump=0 then IF dir=2 THEN dir=1:|SETUPSP,31,7,dir ELSE |ANIMA,31
460 IF INKEY(34)=0 THEN |RINK,-stepy::x=x-1:|MAP2SP,0,x:if jump=0 then
IF dir=1 THEN dir=2:|SETUPSP,31,7,dir ELSE |ANIMA,31
471 IF INKEY(67)+jump=0 THEN
jump=ciclo:|SETUPSP,31,0,205:|SETUPSP,31,15,dir-1:|SETUPSP,31,7,31+dir
472 IF ciclo-jump=20 THEN jump=0:|SETUPSP,31,7,dir:|MOVER,31,5,0:
490 |PEEK,27483,@xp:IF xp<-20 THEN |LOCATESP,30,50,80:'pajaro vuelta a
empezar
501 ciclo=ciclo+1
502 IF xant=x THEN |MAP2SP,0,32000
503 xant=x:' IF quieto THEN no imprimo el castillo asi no parpadea
510 GOTO 410
5000 |MUSIC:CALL &BC02:pen 1:MODE 2:END

```

## 15 Juegos de plataformas

La dificultad de programar un juego de plataformas reside fundamentalmente en manejar correctamente la física de los saltos y las colisiones de plataformas. Algo que podrás encontrar en el videojuego: “**Fresh fruits & vegetables**” disponible en 8BP



Fig. 94 Fresh fruits & vegetables

### Saltos:

Básicamente para la física de los saltos en lugar de usar la ecuación de Newton he definido una ruta en la que la Vy comienza en -5 y va disminuyendo fotograma a fotograma. Al llegar a la posición cenital, se cambia la imagen para que se borre a si mismo en su parte superior y la velocidad Vy se torna positiva, y poco a poco va aumentando. En el fondo es como aplicar la ecuación de Newton, pero sin cálculos.

### Comprobación del suelo

Mientras el personaje camina, compruebo en cada fotograma si hay suelo. Como las plataformas pertenecen al mapa del mundo, usan identificadores de sprite bajos (<10) de modo que si con COLSPALL detecto un numero <10 es que hay suelo. Si el resultado de la detección es un 32 (los sprites van de 0 a 31) es que no hay nada y el personaje debe empezar a caer. Los enemigos no detectan nada. simplemente caminan a través de rutas predefinidas, en las que se indica cuantos pasos deben dar en cada dirección y vuelta a empezar. prácticamente su lógica es casi nada pues le asignas una ruta a un sprite y le activas el flag de movimiento automático y de ruta. A partir de ahí el sprite recorre solito la ruta paso a paso al invocar a |**AUTOALL** (la cual internamente ya invoca a |**ROUTEALL**).

### Colisiones de plataformas:

cuando el personaje está en ruta de caída, lo muevo (sin imprimirlo) hacia abajo 5 unidades y detecto colisión de sprites. A continuación, lo muevo (sin imprimirlo) 5 unidades hacia arriba. si la colisión es 32 es que no hay colisión y mantengo al personaje cayendo. Si la colisión es menor que 10 es que ha colisionado con una plataforma. En ese caso muevo el personaje para que encaje perfectamente con el comienzo de la plataforma, así: posición de la cabeza del muñeco es "posy" posición de los pies es posy+26 pues el personaje mide 21 y le sumo 5 pues está cayendo (hay 5 de autborrado), de modo que en realidad mide 26. Como las plataformas las hemos ubicado en múltiplos de 8, simplemente me quedo con el resto de la división entera, que lo puedo obtener con un AND 7 en resumen, dos instrucciones muy bien pensadas, pero solo dos:

```
dy =(posy%+26) AND 7  
| MOVER, 31, 5-dy, 0
```

Luego le asigno la secuencia de animación de andar, que no es la de caer y sus fotogramas miden 21 de alto.



## 16 Hordas de enemigos en juegos de scroll

Los juegos con scroll normalmente se plantean como una sucesión de hordas de enemigos de diferente tipo y trayectorias, mientras avanzas vertical u horizontalmente.

Si quisieras que tu mapa tuviese 10 hordas en diferentes instantes de avance del mapa (o de ciclo de juego) podrías usar 10 sentencias IF, pero sería muy lento de ejecutar en cada ciclo (cada comprobación IF cuesta un milisegundo)

La mejor solución es mantener dos arrays, uno de posición de la horda y otro de tipo de horda

Index (Número secuencial de orda)	Nexthorda (Ciclo en el que debe aparecer)	Tipohorda (Tipo de enemigos de la horda)
0	100	1 – aviones
1	200	2 – misiles
2	250	4 – ovnis
3	320	3 – dragones

```
10 dim tipohorda(10)
20 tipohorda(0)=1: tipohorda(1)=2: tipohorda(2)=4: tipohorda(3)=3:
30 dim nexthorda(0)
40 nexthorda(0)=100:nexthorda(1)=200: nexthorda(2)=250 ...
50 index=0

100 rem ciclo de juego
110 ciclo=ciclo +1
120 IF ciclo = nexthorda(index) THEN GOSUB 500
...
500 rem rutina creación horda
510 on tipohorda(index) goto 600,700,800
520 rem rutina creacion horda tipo 4. Al final haremos goto 1000
600 rem rutina creacion horda tipo 1. Al final haremos goto 1000
700 rem rutina creacion horda tipo 2. Al final haremos goto 1000
800 rem rutina creacion horda tipo 3. Al final haremos goto 1000
...
1000 Index=index+1
1010 RETURN
```

En lugar de usar el ciclo en el que debe aparecer también puedes hacer la comparación con la posición de mapa en la que te encuentres, eso ya dependerá de cómo quieras hacerlo, pero con esta idea de los dos arrays ya tienes el planteamiento general para organizar tus hordas de enemigos



## 17 Minicaracteres redefinibles: PRINTAT

El juego de caracteres de Amstrad es bonito y bien construido. Sin embargo, en mode 0 tan solo disponemos de 20 caracteres de ancho por línea y aparecen demasiado “ensanchados”, por lo que en ocasiones no son adecuados para mostrar ciertos textos o marcadores de un juego. Además, el comando PRINT es muy lento, por lo que es poco recomendable actualizar los marcadores frecuentemente, ya que el juego se “detiene” mientras se está imprimiendo, son pocos milisegundos, pero es perceptible.

Por ese motivo, a partir de la versión v31 de 8BP se ha incorporado el comando PRINTAT, el cual puede imprimir una cadena de caracteres usando un nuevo juego de caracteres más pequeño (los llamo “minicaracteres”). Este nuevo comando permite usar el mecanismo de transparencia de los sprites, de modo que podrás imprimir caracteres respetando el fondo. Funciona del siguiente modo:

```
|PRINTAT, <flag transparencia>, y, x, @string
```

Ejemplo:

```
cad$= "Hola"  
|PRINTAT, 0, 100, 10, @cad$
```



Fig. 95 PRINTAT

El comando |PRINTAT imprime cadenas de caracteres y no variables numéricas, por lo que si quieres imprimir un número (por ejemplo, los puntos en el marcador de tu videojuego) debes hacerlo así:

```
puntos=puntos+1  
cad$= str$(puntos)  
|PRINTAT,0,100,10, @cad$
```

El comando |PRINTAT no se ve afectado por los límites para el clipping establecidos con |SETLIMITS. Esto es lo más lógico puesto que normalmente usarás PRINTAT para imprimir puntuaciones en tus marcadores, que se encontrarán fuera del área delimitada por |SETLIMITS.

A diferencia del comando PRINT del BASIC, el comando |PRINTAT es bastante rápido y puede ser utilizado para actualizar los marcadores de tu videojuego con frecuencia. PRINTAT usa un alfabeto redefinible, que puede contener una versión reducida o diferente de los caracteres “oficiales” del Amstrad. Por defecto 8BP proporciona un pequeño alfabeto compuesto por números, letras mayúsculas, el espacio en blanco y algunos símbolos. No podrás usar un carácter que no se encuentre entre este conjunto, como las letras minúsculas. Los caracteres de dicho alfabeto miden todos lo mismo: 4pixels de ancho x 5 pixeles de alto, es decir 2 bytes x 5 líneas.

Esta cadena contiene todos los caracteres que puedes usar con el alfabeto de serie de 8BP. Fíjate que no hay minúsculas y faltan muchos símbolos, aunque puedes crear tu propio alfabeto que los contenga. El último símbolo es el “.”

"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ:!.,"

CARACTERES DISPONIBLES:  
0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ:!.,"  
  
PUEDES CREAR TU PROPIO SET DE CARACTERES

Fig. 96 juego de caracteres disponible por defecto en 8BP para usar con PRINTAT

**IMPORTANTE:** Si tratas de imprimir con PRINTAT un carácter no existente en el alfabeto creado, se imprimirá el último definido en la lista de caracteres, que en el alfabeto por defecto es el punto “.”

He creado los caracteres usando las tintas 2 y 4, para que se pueda usar la sobreescritura, ya que los colores de fondo son 0 y 1 y usando sobreescritura la tinta 2 debe ser igual a la 3 y la 4 debe ser igual a la 5 (consulta el capítulo donde explico la sobreescritura). Para usar la sobreescritura simplemente pon a “1” el flag de transparencia en el comando PRINTAT.

## 17.1 Crea tu propio alfabeto de minicaracteres

Los caracteres que vayas a usar con el comando PRINTAT se definen en un fichero en el directorio ASM y se importan desde el fichero “images.asm”

Veamos un fragmento de “images.asm”

Encontraremos estas tres líneas:

```
; si no vas a usar el comando |PRINTAT, sino simplemente los caracteres del Amstrad  
; entonces puedes comentar las siguientes 3 líneas  
_BEGIN_ALPHABET  
read "alphabet_default.asm"  
_END_ALPHABET
```

El alfabeto son unos pocos datos y las imágenes de cada carácter. Todo ello se ensambla dentro de la zona de memoria destinada a imágenes de 8BP. El alfabeto por defecto ocupa poco más de 400 bytes.

El fichero alphabet\_default.asm contiene el alfabeto que 8BP incluye de serie. Tu mismo puedes crearte tu propio fichero de alfabeto. Este fichero contiene 3 variables que indican el tamaño de los caracteres, los cuales puedes dibujarlos del tamaño que quieras. Por defecto yo he creado un alfabeto de 2 bytes de ancho por 5 líneas de alto, pero tu puedes decidir usar otro tamaño, incluso crear caracteres gigantes. Si cambias el ancho o el alto, también deberás cambiar consecuentemente la variable ALPHA\_TAMANO en tu fichero alphabet.asm.

```
ALPHA_ancho db 2; ancho alfabeto.Todas las letras miden lo mismo  
ALPHA_alto db 5; alto alfabeto.todas las letras miden lo mismo  
ALPHA_tamano db 2*5
```

A continuación, encontramos la cadena que indica los caracteres válidos para usar con el comando PRINTAT. Estos caracteres son válidos porque tienen un dibujo asociado. Tras esta cadena, se encuentran uno por uno los dibujos de todos esos caracteres, en el mismo orden en el que figuran en la cadena de texto ALPHA\_LIST

```
ALPHA_LIST
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ:!.," ; caracteres creados
db 0 ;byte cero indica final de cadena de caracteres de la Alpha_list
```

Los dibujos de cada carácter de la cadena ALPHA\_LIST vienen a continuación. Yo los he creado con la herramienta SPEDIT. El primero de ellos debe ser el primero de la cadena ALPHA\_LIST, es decir, el “0”.

Aquí se muestran los bytes correspondientes a este carácter:

<pre>; caracter 0 db 12 , 8 db 8 , 8 db 8 , 8 db 32 , 32 db 48 , 32</pre>	
---	--

Seguidamente encontraremos una por una el resto de las letras, números y símbolos definidos. Con algo de paciencia puedes crearte tu propio juego de minicaracteres, lo cual le dará más personalidad a tu videojuego. Sólo necesitas crear los que vayas a usar y cuantos menos sean, menos memoria gastarás de la zona de imágenes.

Recuerda que, si tratas de imprimir un carácter que no has creado, se imprimirá el último de los caracteres definidos en la ALPHA\_LIST

## 17.2 Alfabeto por defecto para MODE 1

El alfabeto por defecto de 8BP esta creado en MODE 0 y si lo intentas usar en MODE 1 no te va a funcionar bien, porque son dibujos hechos en mode 0. De serie 8BP trae también un alfabeto que funciona en MODE 1. Simplemente tendrás que alterar una línea de images.asm. Este alfabeto está inspirado en un tipo de letra llamado “5th agent”

```
_BEGIN_ALPHABET
read "alphabet_default_mode1.asm"
-END_ALPHABET
```

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ:!.,"
```



## 18 Pseudo 3D

En los años 80 fueron populares los juegos de coches estilo “Pole position”. Daban sensación 3D, pero no realizaban los cálculos 3D, tan solo para el plano de la carretera, y en ocasiones ni eso, pues eran aproximaciones que daban una buena sensación de velocidad.

En las máquinas recreativas se usaban chips específicos para realizar “sprite scaling” haciendo que los sprites aumentasen de tamaño suavemente, y el cálculo de la carretera mediante un chip específico (como el “sega Road chip”) que se dedicaba exclusivamente a pintar la carretera con sus franjas. El plano de la carretera después se sumaba al plano de sprites y se componía la imagen final. Estos chips se usaban en máquinas arcade tales como “Pole Position” y “Space Harrier”.



Fig. 97 Pole position y Out Run (máquinas arcade)

La característica común a las técnicas software (usadas en ordenadores de 8bit) y hardware (usadas en máquinas recreativas) es que las curvas son una ilusión: se deforma la carretera al tiempo que se mueven unas montañas en el horizonte para dar la sensación de giro, pero no hay ningún giro. Los resultados eran en no pocas ocasiones muy convincentes pero las curvas eran realmente una ilusión.

Las técnicas utilizadas en ordenadores de 8bits eran muy variopintas. Cualquier cosa era aceptable con tal de hacerle creer al jugador que se encontraba en un circuito de carreras. En muchas ocasiones se usaba rotación de tintas para dar sensación de velocidad. Muchos juegos sufrían de un bajo frame rate, por debajo de 5 fps. Entre los mejores juegos para Amstrad se encuentran el “**3D grand Prix**” (que usa sprite scaling por software combinado con animación por tintas), el “**Buggy boy**” basado en una técnica de programación muy avanzada de efecto raster y algunos otros como el “**Crazy cars**” o el “**Chase HQ**”.

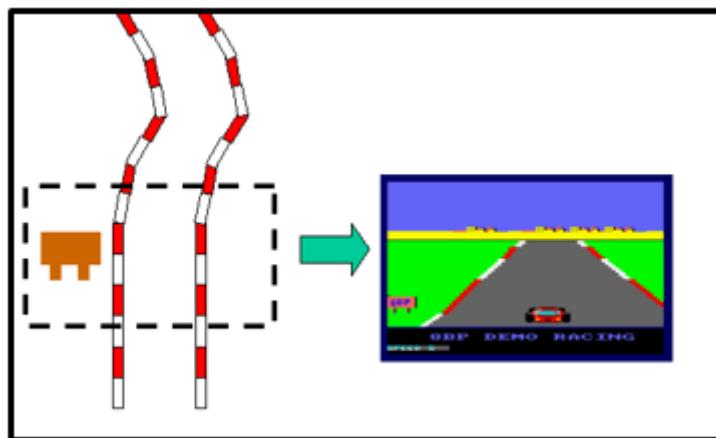
Desde la versión V32 de 8BP tienes a tu disposición capacidad Pseudo-3D, utilizada en el juego de demostración “3D Racing one”. Es muy sencillo usarla y con ella podrás construir tus juegos de carreras, de tanques, de naves, etc

Para usar Pseudo-3D en 8BP debes usar la “**opción de ensamblaje**” 3, que te deja 24 KB libres para tu programa BASIC

Las capacidades que aporta 8BP para poder disponer de pseudo-3D son:

- 1) **Proyección 3D:** consiste en proyectar un mapa del mundo 2D en 3D, de modo que, aunque lo recorramos en 2D, nos dé la sensación de verlo en 3D. Esto lo haremos invocando al comando |3D para configurar los comandos PRINTSPALL

y PRINTSP de modo que proyecten en 3D antes de pintar en pantalla. No habrá posibilidad de girar en el plano, siempre “miraremos” hacia delante, pero el efecto combinado de una curva unido a unas casas o montañas en el horizonte que se desplacen, simulará que estamos tomando una curva.



*Fig. 98 Proyección 3D*

- 2) **Zoom:** consiste en poder usar distintas versiones de una misma imagen de un objeto para dar efecto de zoom a medida que nos acercamos a él. Esto simplemente lo haremos en el fichero de imágenes, definiendo 3 versiones de una misma imagen y agrupándolas en una estructura. En la imagen se muestra el típico ejemplo del cartel que se acerca. Los comandos |PRINTSPALL y |PRINTSP escogerán la versión más adecuada de la imagen según a la distancia a la que se encuentre, sin necesidad de hacer nada más que definir la imagen como una imagen de tipo “Zoom”.



*Fig. 99 Zoom images*

- 3) **Segmentos:** consiste en poder disponer de sprites de tipo “segmento”, definidos por una sola scanline horizontal (por lo que ocupan muy poco), a los que podemos asociar una longitud y una inclinación. Con ellos podremos construir caminos, ríos, etc. y bordes de carreteras. Esto simplemente lo haremos en el fichero de imágenes definiendo un tipo de imagen que además de ancho y alto posee inclinación. Estos segmentos los utilizaremos en la construcción del mapa del mundo.



*Fig. 100 segmentos con distinta inclinacion*

Los segmentos utilizados en el juego “**3D Racing one**” son rojos o blancos y aunque sean largos, solo están definidos por una scanline. Por ejemplo, el segmento blanco izquierdo consiste en unos pocos bytes verdes para el césped, un par de blancos y otros pocos grises para la carretera. En el momento de imprimirse, el segmento así definido se replica hasta medir la longitud deseada, y se imprime en perspectiva, por lo que, aunque sea un segmento recto, se mostrará torcido.

Por último, mencionar que en muchas ocasiones será necesaria la actualización dinámica del mapa (comando |UMAP). Gracias a este comando, el mapa puede ser muy grande (lo cual es necesario si creamos un circuito con muchísimos segmentos). Este comando se describe en el capítulo de scroll.

A continuación, vamos a ver en profundidad estas 3 características y como usarlas en tus programas.

## 18.1 Proyección 3D

Para proyectar disponemos del comando |3D

Uso

```
|3D, 1, <sprite_fin>, <offsety> : REM esto activa la proyección 3D
|3D, 0 : REM desactiva la proyección 3D
```

Este comando activa la proyección 3D en el comando |PRINTSP y en |PRINTSPALL. Esto significa que antes de imprimirse en pantalla se calcularán las coordenadas “proyectadas” y a continuación se imprimirán en pantalla. Las coordenadas de los sprites no se ven afectadas, es decir, las coordenadas seguirán siendo las mismas, pero en el mundo 2D. En pantalla ahora tenemos una vista 3D y las coordenadas en las que se imprimirán serán el resultado de ejecutar la función de proyección sobre sus coordenadas 2D.

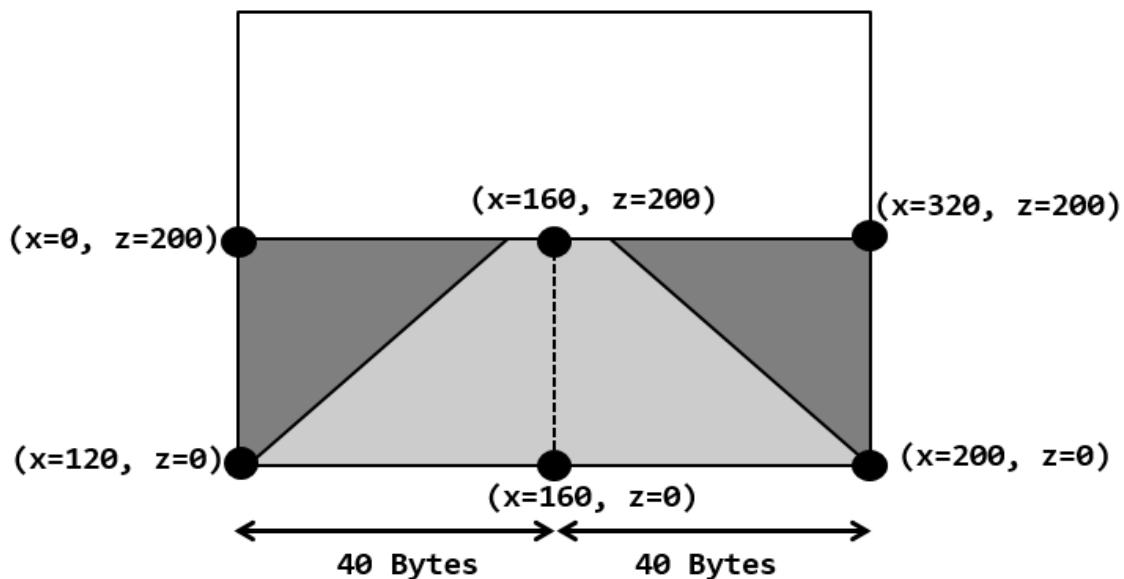
Los sprites afectados son todos desde el Sprite 0 hasta el <**sprite\_fin**>. El resto de sprites no se proyectarán, simplemente se imprimirán en pantalla en sus coordenadas 2D.

Este comando **no afecta a los mecanismos de colisión**, es decir, si usamos COLSPALL y detectamos una colisión entre sprites proyectados, la colisión se estará produciendo en el plano 2D. Puede que en algún caso se solapen parcialmente dos sprites proyectados en

pantalla, pero eso no significa que hayan colisionado, puede que uno esté ligeramente más adelante que el otro y al proyectarse se solapen, pero eso no es una colisión real. Las colisiones se detectan en el plano 2D.

En cuanto al último parámetro `<offsety>` es para proyectar mas arriba o mas abajo, de modo que podamos ubicar los marcadores del juego donde queramos. Al proyectar la pantalla, que mide 200 de alto, se transforma en 100 pixels de alto, de modo que podemos escoger a que altura ubicamos la proyección. Si un Sprite no es afectado por la proyección por ser superior a `<Sprite_fin>`, entonces tampoco le afecta el `<offsety>`. Este es el caso, por ejemplo, de las nubes y las casas en el horizonte en el juego “3D Racing one”

Para entender las coordenadas de pantalla en la que finalmente aparecen tus sprites proyectados, te recomiendo que leas el siguiente apartado de matemáticas, muy sencillo, con el que lo entenderás por completo. La siguiente figura representa cuales son las coordenadas del mapa del mundo que se proyectan en ciertos puntos representativos de la pantalla cuando MAP2SP es invocado con ( $y_0=0$ ,  $x_0=0$ ). La coordenada Z representa la distancia a la que se encuentran los objetos, también llamada “profundidad”.



*Fig. 101 coordenadas del mundo proyectadas*

Si en lugar de ( $x_0=0$ ,  $y_0=0$ ) usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen estarán desplazadas en ( $x$ ,  $z$ ) lo que se indique con ( $x_0$ ,  $y_0$ ).

### 18.1.1 Matemáticas de la proyección pseudo 3D

La proyección pseudo 3D consiste en proyectar sobre la pantalla el plano del suelo, que es donde está nuestro mapa 2D del mundo.

#### Cálculo de la coordenada Y

Nuestro suelo puede ser infinito, pero proyectaremos solo 200 pixeles de largo. A dicho largo se le llama “profundidad” o coordenada “Z”. Dichos 200 pixeles de profundidad, al proyectarse se comprimen en 100 líneas de alto (coordenada Y proyectada). Los pixeles

más lejanos proyectados constituyen la línea de horizonte. Ojo, no vamos a ver objetos lejanísimos en el horizonte, tan solo los que se encuentren como mucho a 200 de profundidad.

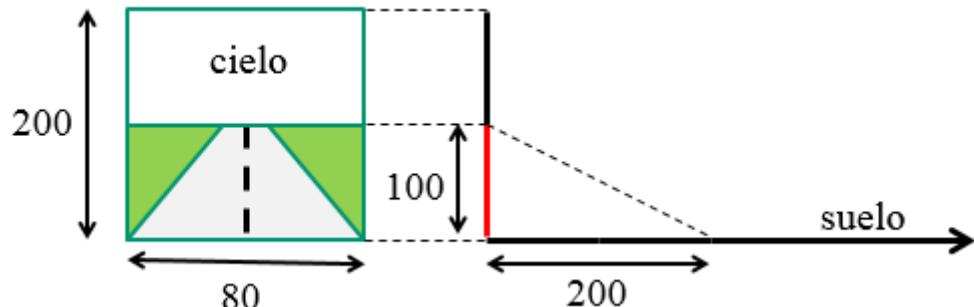


Fig. 102 proyección del suelo en la pantalla

A medida que los objetos se alejan se ven mas y mas pequeños. No es una relación lineal la existente entre la pantalla y el suelo, es decir, para saber a que altura se debe imprimir un pixel que esta a cierta distancia, es incorrecto pensar que como la profundidad abarcada es 200 y se proyecta sobre 100 líneas de alto, basta dividir entre dos. En una función lineal (tal como  $y = f(z) = A \cdot z + B$ ) la derivada ( $A$ ) es constante, pero en la proyección, lo que es constante es la “segunda derivada” de la función  $f(z)$ . Ahora veremos esto en detalle.

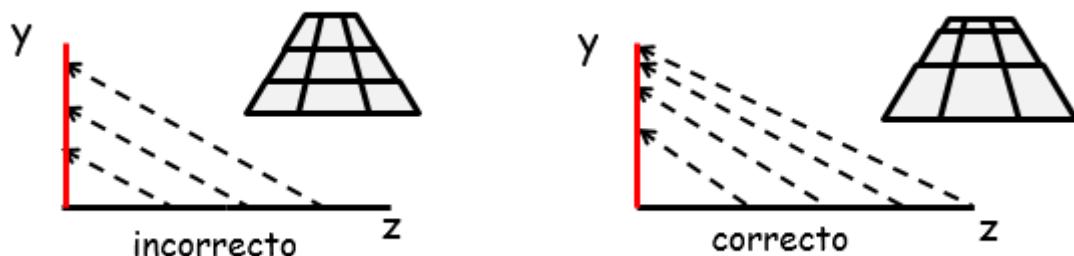


Fig. 103 La proyección correcta no es lineal

Supongamos que empezamos con “ $Z=0$ ” y queremos saber cuánto vale “ $Y$ ”. Es muy fácil, en la línea de suelo ( $z=0$ ) la coordenada “ $y$ ” también es cero.

Si incrementamos  $z$  con una pequeña variación “ $dz$ ”, la “ $y$ ” valdrá la “ $y$ ” anterior (cero) más un incremento  $dy$ , que inicialmente será igual al incremento  $dz$  pues el incremento ha sido pequeño y estamos cerca del horizonte.

$$dy \text{ (inicial)} = dz$$

Ahora bien, si de nuevo nos alejamos  $dz$ , el incremento  $dy$  debe de disminuir, y si volvemos a alejarnos  $dz$ , el  $dy$  que debemos sumar cada vez será más pequeño. Es decir:

**Cada vez que nos alejamos  $dz$ , sumamos un incremento a “ $y$ ” que cada vez es menor**  
 $z=z+dz$   
 $dy = dy + ddy$  , siendo  $ddy$  negativo  
 $y= y + dy$

El incremento que sufre  $dy$  es constante. Ese incremento lo hemos llamado  $ddy$ . Pues bien,  $dy$  es la “derivada” de  $y$ , mientras que “ $ddy$ ” es lo que se llama “segunda derivada”. Aquí la derivada no es constante, pero la segunda derivada sí lo es.

El valor constante que asignemos a “ $ddy$ ” producirá proyecciones mas o menos exageradas. En 8BP el valor  $ddy$  es negativo, de aproximadamente -0.005, haciendo que en la línea de suelo el  $dy$  sea prácticamente 1, mientras que, en la línea de horizonte, la acumulación de 200  $ddy$  hace que el valor de  $dy$  acabe en cero.

A pesar de la simplicidad de estos cálculos, hacerlos en tiempo real es costoso para nuestro amado Amstrad CPC, por lo que en realidad 8BP realiza una aproximación para evitar cálculos y traducir “ $z$ ” a “ $y$ ” de un modo mucho mas simple y con un resultado muy similar.

Si  $0 \leq z < 50$ , entonces  $dy = dz$ , por lo tanto  $y = z$

Si  $50 \leq z < 110$ , entonces  $dy = dz/2$ , por lo tanto  $y = 50 + (z-50)/2$

Si  $110 \leq z \leq 200$ , entonces  $dy = dz/4$ , por lo tanto  $y = 50 + (110-50)/2 + (z-110)/2$

Es decir, hemos dividido la pantalla en tres franjas y cada franja se trata como una zona “lineal” (pues “ $dy$ ” es constante) pero con diferente valor de “ $dy$ ” respecto las otras franjas. Esta aproximación da resultados visualmente muy similares a los matemáticamente correctos.

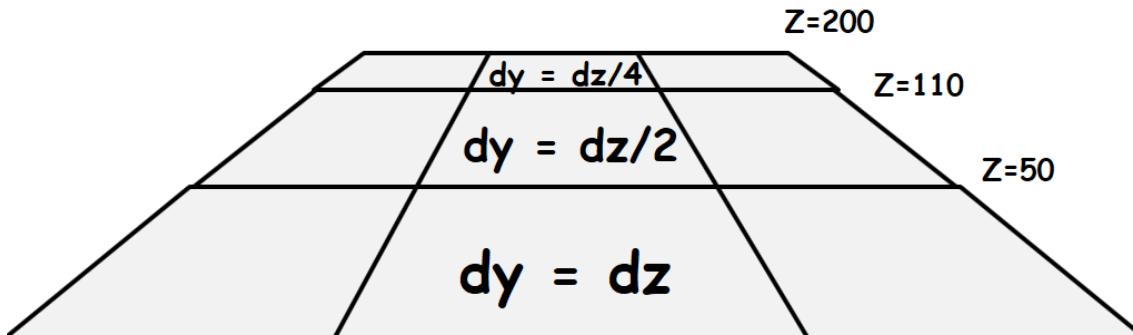


Fig. 104 Aproximación de 8BP

Las ecuaciones usadas en 8BP tienen además en cuenta que las coordenadas de pantalla en realidad están “invertidas”, es decir la coordenada cero es la superior y la 200 es la inferior, pero eso es solo un ajuste final muy sencillo.

#### Ahora pasemos al cálculo de la coordenada “X”:

Hay algo fundamental que diferencia la línea de horizonte de la línea de suelo. La línea de suelo mide 80 Bytes, pero la de horizonte representa más anchura, pues la carretera es recta y lo que mide 80 en el suelo, mide una fracción en el horizonte, concretamente en 8BP mide 4 veces menos. La carretera se estrecha en el horizonte, porque el horizonte representa mucho más. En la proyección que aplica 8BP representa 320 bytes.

Y si el horizonte mide 320 y el suelo 80, es porque el área real total que somos capaces de abarcar en pantalla una vez efectuada la proyección es un área trapezoidal. NO es que el suelo sea un trapecio, sino que el área de suelo que somos capaces de ver en pantalla tiene forma de trapecio

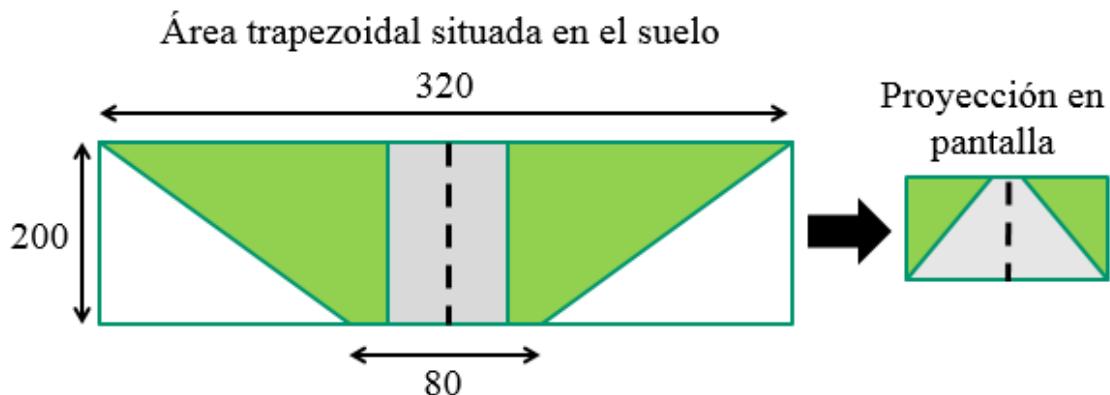


Fig. 105 Área trapezoidal visualizada

Intuitivamente y sin mostrar ninguna ecuación, ya está claro lo que queremos hacer y cómo funciona la proyección. Ahora veamos las ecuaciones.

En esencia las matemáticas de la coordenada "X" son las mismas que las de la coordenada "Y". Sin embargo, no se hacen del mismo modo, porque una vez que tenemos la coordenada "Y" calculada, podemos hacer una ecuación lineal  $x=f(y)$  ya que debido a la no linealidad de "Y" respecto "Z", es como crear una relación no lineal entre "X" y "Z". Anteriormente hemos dicho que el horizonte mide 320 Bytes y el suelo mide 80 Bytes. Esto significa que el suelo mide 4 veces menos que el horizonte. En la lejanía debemos dividir entre 4 (dividir es costoso así que preferiremos multiplicar por un factor 0.25) mientras que en la cercanía deberemos multiplicar por un factor = 1.

Lo que tenemos que hacer es centrar la coordenada X del objeto a proyectar respecto el centro de la pantalla. A continuación, multiplicaremos por un factor que dependerá de la coordenada "Y" proyectada. Esto establecerá una relación no lineal entre "X" y "Z"

Debemos construir una ecuación que retorne un resultado 4 veces menor en el horizonte, por ejemplo:

```

Factor = ((100-y)+ 32 ) / 2
x= (x - centro) * Factor + centro
si z=200 entonces y= 100, y entonces factor =16
si z=0 entonces y= 0, y entonces entonces factor =64 ( 4 veces mas)

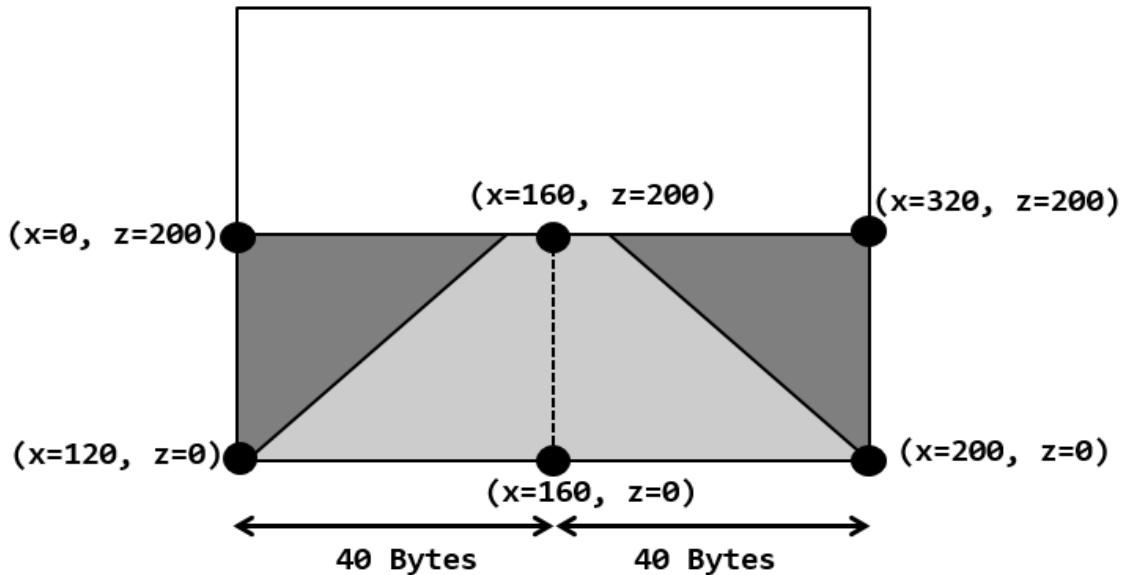
```

La ecuación escogida es interesante porque la división entre 2 es algo que el Amstrad puede hacer rotando en binario un bit. Es decir, puede hacerla en pocos ciclos de reloj.

Como se ha mostrado en la ecuación, para proyectar la coordenada "X", basta con centrarla y a continuación multiplicarla por el factor obtenido. El número obtenido así es muy grande, pues en el caso de la línea de horizonte estaremos multiplicado por 16 y en la línea de suelo se ha multiplicado por 64, es decir hemos multiplicado la coordenada x por un factor cuyo valor se encuentra entre 16 y 64. Entre el horizonte y suelo tendremos todos los 48 valores posibles para el factor, por lo que, aunque el factor sea un número entero, evolucionará suavemente.

A continuación, dividimos entre 64, que no es más que rotar en binario 6 veces, y listo. Esto último será equivalente a haber multiplicado por 0.25 el horizonte, por 1 el suelo y por el valor decimal que corresponda a cualquier altura intermedia entre el suelo y el horizonte, pero lo hemos hecho con números enteros, que el Amstrad puede manejar rápido. A esta técnica se la llama "aritmética de coma fija".

Teniendo todo esto en cuenta, y suponiendo que le pedimos a MAP2SP que nos genere los sprites del mapa del mundo a partir de la coordenada ( $yo=0$ ,  $xo=0$ ) lo que estaremos viendo en la pantalla tendrá las siguientes coordenadas del mundo. Seguro que ahora te resulta sencillo entender la figura.



*Fig. 106 coordenadas del mundo proyectadas*

Como podrás deducir el punto ( $x=0$ ,  $y=0$ ) del mapa del mundo queda proyectado fuera de la pantalla, no se visualiza. Si en lugar de ( $xo=0$ ,  $yo=0$ ) usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen, estarán desplazadas en ( $x$ ,  $z$ ) lo que se indique con ( $xo$ ,  $yo$ ).

### 18.1.2 Curvas

Como he mencionado al principio de este capítulo, la proyección 3D que usa 8BP no permite rotaciones del plano, por lo que para simular una curva tendremos que hacer un pequeño truco. Se trata de torcer la carretera a derecha o izquierda, al tiempo que movemos sprites tales como casas o montañas en el horizonte. Estos sprites no serán proyectados y para evitarlo usaremos identificadores por encima de `<Sprite_fin>`. Recuerda que para activar la proyección 3D haremos:

**|3D, 1, <Sprite\_fin>, <offsety>**

Por consiguiente, un aparente circuito con curvas quedará representado en nuestro mapa 2D como un camino con inclinaciones a derecha e izquierda, simplemente.

Con esta estrategia, podemos crear la sensación de que un piloto de carreras recorre un circuito con verdaderas curvas, y dar la sensación de que su coche gira en las curvas mediante unas casas o montañas en el horizonte que se mueven en dirección contraria al avance en la coordenada X. Si eres un buen artista y tuerces en mayor o menor grado paulatinamente diferentes segmentos, darás la sensación de curvas muy realistas

Esta estrategia es computacionalmente muy eficiente y suficientemente realista. Si quisiersemos rotar el plano del suelo de verdad, tendríamos que aplicar cálculo matricial para rotar, con muchas operaciones muy costosas y, además, las texturas de los sprites ubicados en el suelo deberían también rotar, de modo que el costo computacional sería enorme.

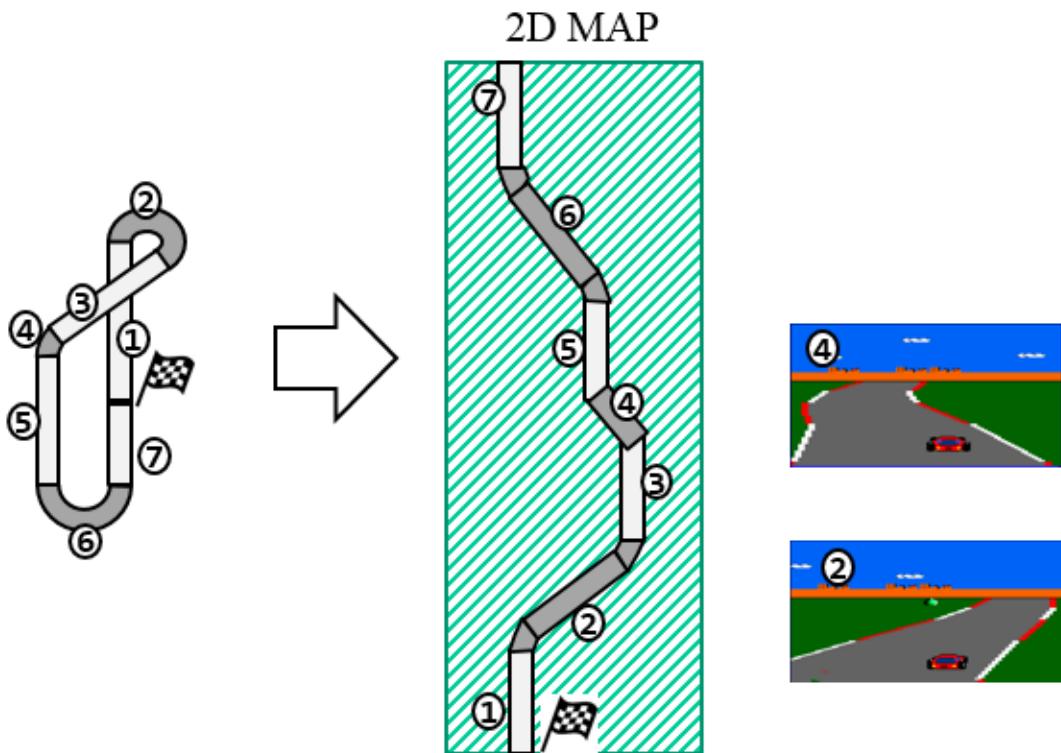


Fig. 107 circuito imaginario y mapa del mundo 2D

Hoy en día los ordenadores son tan potentes que las estrategias aquí presentadas carecen de sentido, pero son estrategias superiores en elegancia e ingenio a la “fuerza bruta” actual. Recuerda que “las limitaciones no son un problema, sino una fuente de inspiración”.

Te deberás servir del comando **|UMAP** para recorrer mapas de circuitos grandes, pero recuerda invocar a **|UMAP** no en cada ciclo sino tan solo de vez en cuando.

## 18.2 Zoom images

Para definir una “zoom image”, simplemente crearemos las distintas versiones de la imagen (3 versiones). Después, en el fichero de imágenes “images\_mygame.asm” buscaremos una etiqueta llamada “\_3D\_ZOOM\_IMAGES”.

Encontraremos esto:

```
_BEGIN_3D_ZOOM_IMAGES
;=====
; limites aplicables a todas las imagenes con zoom
; para estos limites se considera el horizonte como el 0 y hacia abajo
; va creciendo hasta 200
_ZOOM_LIMIT_A
db 120; entre 200 (suelo) y limitA se pone imagen 3
_ZOOM_LIMIT_B
db 50
; entre este limite y el limite A, se pone imagen 2
; mas cerca del horizonte que limit B se pone imagen 1
=====
```

```

CARTEL_ZOOM
db 1; ancho simbolico
db 1; alto simbolico
dw CARTEL1, CARTEL2, CARTEL3

```

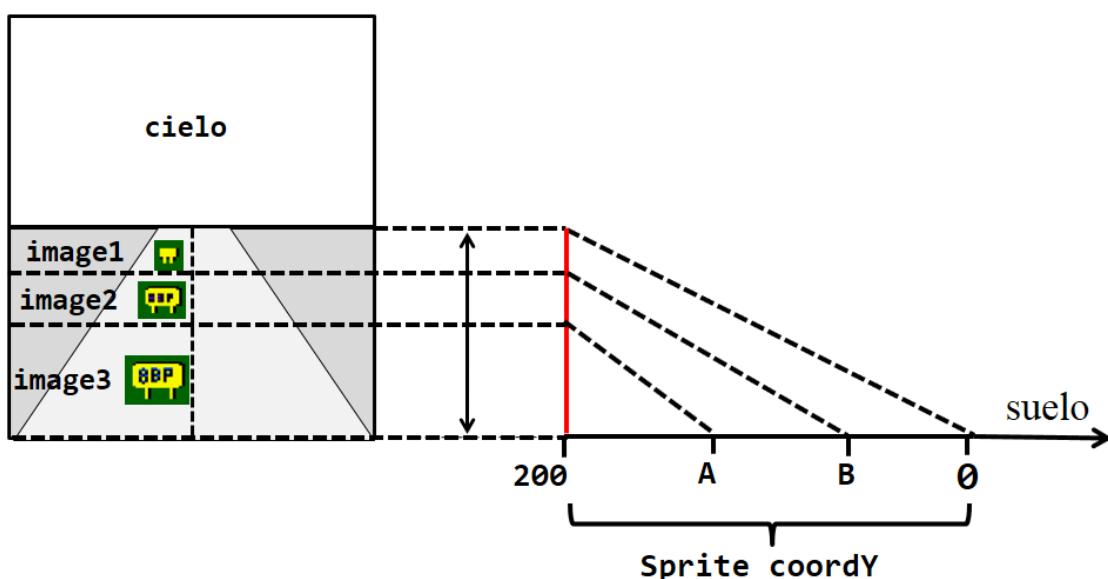
```
_END_3D_ZOOM_IMAGES
```

Todas las imágenes de tipo ZOOM se deben definir después de la etiqueta “**\_BEGIN\_3D\_ZOOM\_IMAGES**”. Son imágenes que tienen un ancho y un alto simbólico, pues en realidad un Sprite al que se asigne una de estas imágenes, usará el ancho y alto de la versión de la imagen que se escoja automáticamente en función de su coordenada Y. Es decir, “CARTEL1” es una imagen normal, definida previamente, con su ancho, su alto y sus bytes que contienen el dibujo. Y lo mismo se puede decir de “CARTEL2” y “CARTEL3”. Si asociamos la imagen “CARTEL\_ZOOM” a un Sprite (esto lo podemos hacer asociando un identificador a dicha imagen al principio del fichero de imágenes) lo que ocurrirá es que en función de la posición del Sprite en pantalla se imprimirá una u otra versión de la imagen.

Para la elección automática de la imagen se establecen unos umbrales de coordenada “y”. Dichos umbrales los puedes cambiar, aunque los que hay por defecto funcionan bien y son:

- entre el horizonte =0 y 50, se escoge la primera imagen (en el ejemplo, “CARTEL1”)
- entre el 50 y 120, se escoge la segunda imagen (en el ejemplo, “CARTEL2”)
- entre el 120 y 200, se escoge la tercera imagen (en el ejemplo, “CARTEL3”)

La elección de estos límites para delimitar las franjas de la pantalla es configurable y puedes poner los que quieras. Ten en cuenta que la elección se realiza en base a la coordenada Y del Sprite sin proyectar. Una vez proyectado, su coordenada Y varía mucho, pero no es la “Y” proyectada la que se utiliza para delimitar las 3 franjas, sino la “Y” del Sprite en 2D. Cuando el Sprite pase de una franja a otra, su imagen cambiará automáticamente. Si te gusta más que aparezca antes alguna imagen puedes modificar los umbrales.



*Fig. 108 franjas de uso de las 3 versiones de la ZOOM image*

Por ejemplo:

```
REM supongamos que CARTEL_ZOOM tiene id=16 en el fichero de imágenes  
|SETUPSP,20,9,16 : REM asocia CARTEL_ZOOM al Sprite 20  
|LOCATESP, 20,100, 160: REM Sprite en el centro del "trapecio"  
(coordy=100)  
|3D,1,31,0: REM se proyectaran todos los sprites  
|PRINTSP,20: REM se imprime la versión 2 del CARTEL
```

### 18.3 Uso de segmentos

Ahora que sabes cómo construir un mapa 2D que “simule” ser un circuito con curvas, te presento una forma potente de construir los segmentos con diferente grado de inclinación que necesitas para construir circuitos de carreras o caminos

Un segmento es una imagen de una sola línea de alto, que por repetición puede alcanzar la longitud que queramos. Además de la longitud, tiene otro parámetro, que es la inclinación total del segmento, en bytes. Dicha inclinación puede ser negativa (inclinación hacia la derecha) o positiva (inclinación hacia la izquierda).

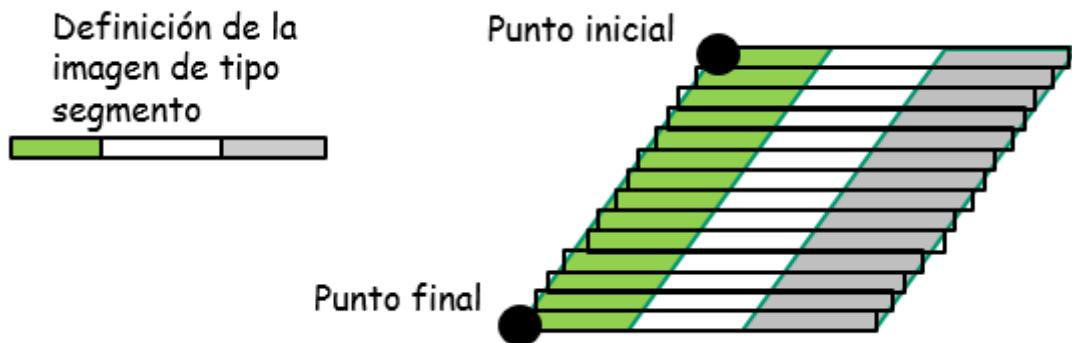
Para definir este tipo de imágenes debes crearlas en el fichero de imágenes de tu juego “images\_mygame.asm”, después de la etiqueta “\_BEGIN\_3D\_SEGMENTS”

```
;===== _BEGIN_3D_SEGMENTS =====  
;podria haber una definicion diferente de segmentos  
; el ancho es el de la scanline  
; el alto es el alto 2D del segmento  
; luego va el dx, que puede ser positivo ( inclinado a izquierda) o  
negativo (inclinado a derecha)  
db 0; esto es para que la primera imagen de tipo segmento sea >  
_3D_SEGMENTS  
  
;----- SEGMENT_EDGE_LWI10 -----  
SEGMENT_EDGE_LWI10  
db 22; ancho  
db 50; alto  
db 10; dx  
db 192,192,192,192,192,192,192,192,192, 240, 240  
,0,0,0,0,0,0,0,0,0,0  
;
```

En el ejemplo hemos definido un segmento que tiene césped a la izquierda (bytes verdes), luego dos bytes blancos y después bytes grises (carretera) a la derecha. Lo de verdes o grises depende de los colores que hayamos asociado a las tintas.

Es un segmento con una inclinación a la izquierda de 10 bytes. Si en la inclinación (dx) hubiésemos puesto un cero, sería un segmento recto, no torcido. Mide 50 líneas de alto, pero cuando se proyecta mide menos, salvo que se encuentre muy cerca.

Los puntos del segmento que se proyectan son sólo dos. Una vez proyectados, se pintan las scanlines una por una con un cierto desplazamiento horizontal para que la última línea acabe empezando en el punto final. Observa que, aunque el segmento se pinte en perspectiva, su ancho es constante. No se pinta más estrecha la parte superior (más lejana) y más ancha la inferior (más cercana), sino que se va a pintar cada scanline exactamente tal como ha sido definida en el fichero de imágenes.



*Fig. 109 en un segmento se proyectan 2 puntos*

Como ves he usado muchos bytes de césped y de carretera. Es para que el segmento se “borre a sí mismo” al avanzar, aunque si el coche va muy deprisa podrán quedar rastros. Una vez que pongas tu juego a funcionar comprobarás si la velocidad es excesiva y quedan rastros.



*Fig. 110 Relación del grado de inclinación de un segmento y la velocidad máxima*

Como se aprecia en la figura, la velocidad de avance máxima para un segmento que se borra a sí mismo puede ser mayor si el grado de inclinación es moderado. Si está muy torcido la velocidad no puede ser tan grande, o quedarán rastros. Una vez que el segmento se proyecta, quedará más corto por efecto de la perspectiva y en función de donde se encuentre puede quedar aún más torcido o menos. Conviene hacerlos anchos para que se borren a sí mismos con más seguridad.

En el juego “**3D Racing one**” hay una fase llamada “superfast” en la cual, usando segmentos menos torcidos, aumenté la velocidad del coche sin que por ello los segmentos dejases rastros. Un sencillo “truco” muy efectivo. Si el juego es lento (por ejemplo, un juego de tanques, que se supone que van despacio) entonces los segmentos pueden estar muy torcidos pues no dejarán rastro por efecto de la velocidad.

En resumen, para aumentar la velocidad tienes dos opciones:

- Usar segmentos menos torcidos
- Usar segmentos más anchos (con más margen de borrado de sí mismos)

## 19 Música

La herramienta de las que voy a hablar en este apartado no la he programado yo, pero están integradas en 8BP y son realmente buenas.

### 19.1 Editar música con WyZ tracker

Esta herramienta es un secuenciador de música para el chip de sonido AY3-8912. Las músicas que genera se pueden exportar y dan como resultado dos archivos

- Un archivo de instrumentos “.mus.asm”
- Un archivo de notas musicales “.mus”

Puedes componer canciones con esta herramienta y la única limitación que tendrás es que todas las canciones que integres en tu juego deberán compartir el mismo fichero de instrumentos.

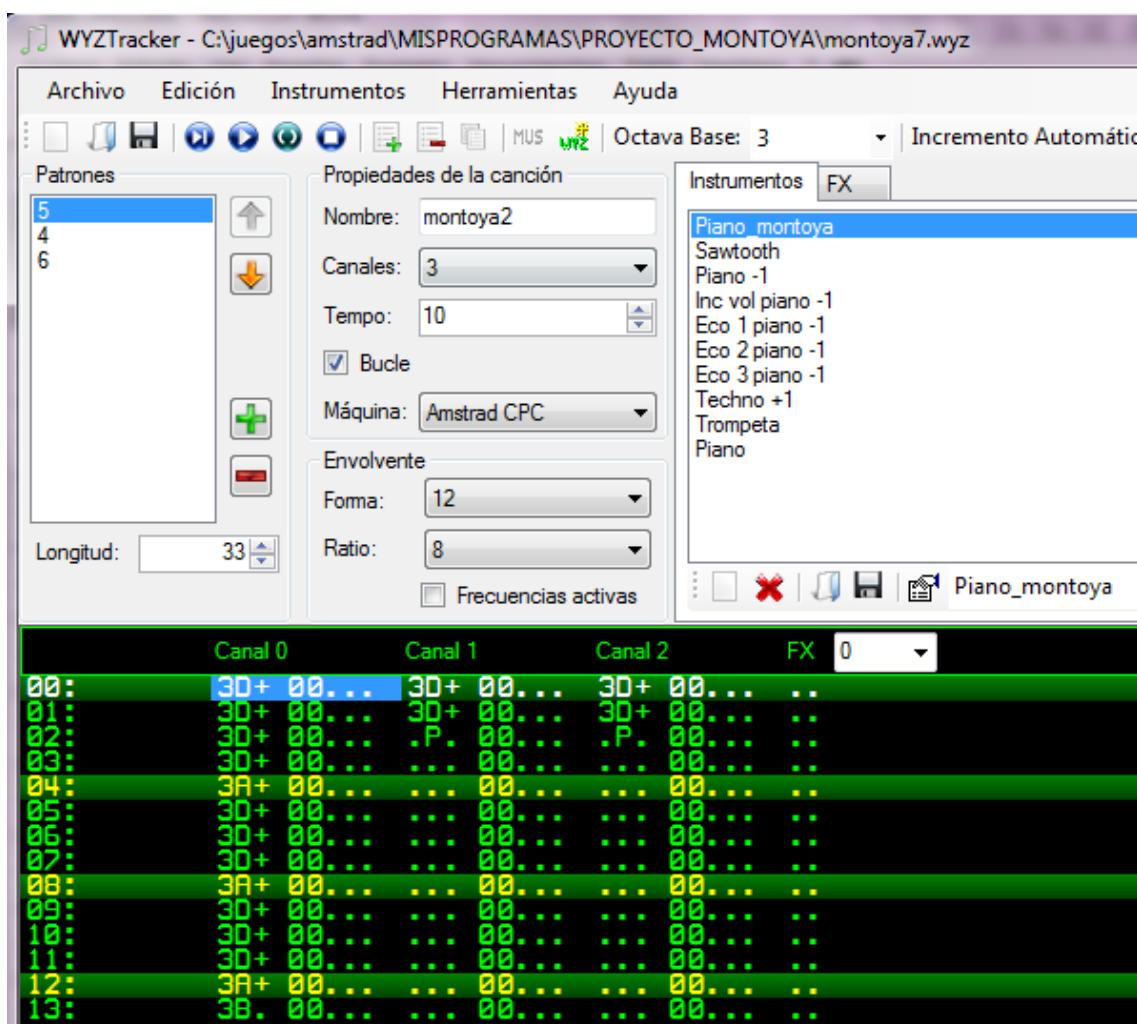


Fig. 111 WYZTracker

Desde la versión V41 puedes usar el canal “FX” para incorporar efectos de sonido. Hasta esta versión ese pseudo canal era ignorado en le player de 8BP. Ojo al editar la música pues, aunque el canal FX está siempre asociado a un canal, dentro del WYZ tracker suena como si fuese independiente y el resultado puede ser diferente a cuando lo oyes en el

amstrad. Si el FX está asociado al canal 0 y pones un FX en un instante en el que el canal 0 no suena, no se oirá nada en el amstrad aunque en el WYZtracker se oiga tu efecto.

Este secuenciador de música se complementa con el WYZplayer que está integrado en la librería 8BP.

Desde la versión V26 de 8BP, las músicas pueden ser compuestas con la versión 2.0.1.0 de WYZtracker y funciona realmente bien. Hasta la versión V25 de 8BP la compatibilidad era con WYZtracker 0.5.07 y había algunos pequeños problemas, pero todo eso ha desaparecido con WYZtracker 2.0.1.0

**IMPORTANTE:** no uses la octava 7 aunque el tracker te permita usarla. Esta octava no es capaz de hacerla sonar y se pueden producir fallos de sincronización durante la reproducción de la música en el Amstrad

Una recomendación importante a la hora de crear tus canciones con el WyZtracker es eliminar los instrumentos que no vayas a usar. De este modo el fichero de instrumentos (que acaba en “.mus.asm”) ocupará menos y puesto que 8BP sólo reserva **1.500 bytes** para la música cada byte es importante

**IMPORTANTE:** si editas la música con una versión de WYZtracker posterior a la 2.0.1.0 puede que no te funcione bien la música y se produzcan efectos raros tales como que un canal deje de sonar o que se desincronicen las notas. Si te ocurre algo así, la solución pasa por crear un fichero desde cero con el WYZtracker 2.0.1.0 y copiar manualmente nota a nota de la música que no te funciona.

## 19.2 Ensamblar las canciones

Una vez que has compuesto tu canción y ya tienes los dos archivos, simplemente editas el fichero make\_music.asm e incluyes tus ficheros de música así:

```
; tras ensamblarlo, salvalo con save "musica.bin",b,32200,1400

org 32200
-----MUSICA-----
; tiene la limitacion de tan solo poder incluir un solo fichero de
; instrumentos para todas las canciones
; la limitacion se solventa simplemente metiendo todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO
read "instrumentos.mus.asm"

; archivos de musica

SONG_0:
INCBIN      "micancion.mus" ;
SONG_0_END:

SONG_1:
```

```

INCBIN      "otra_cancion.mus" ;
SONG_1_END:

SONG_2:
INCBIN      "tercera_cancion.mus" ;
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:

```

Por último re-ensamblas la librería 8BP para que el player de música (que está integrado en la librería) conozca los parámetros de instrumentos y el lugar donde han quedado ensambladas las canciones.

Para ello simplemente ensamblas el fichero make\_all.asm, que tiene este aspecto

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make
correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica.asm"

; ----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos.asm"

```

Y con esto ya tienes todo ensamblado. Ahora debes generar tu librería 8BP así:

**SAVE “8BP.LIB”, b, 24000, 8200**

Y las músicas:

**SAVE “music.bin”, b, 32200, 1400**

### **19.3Qué hacer si no te cabe la música en 1400 bytes**

Es posible que 1.400 bytes no sean suficientes para tus canciones. En caso de que una canción no te quepa (y lo sabrás chequeando donde se ensambla la etiqueta “\_END\_MUSIC”) puedes especificar una dirección diferente de ensamblado para esa canción y las siguientes. En el caso del videojuego “Nibiru”, se hace esto con la tercera canción, ensamblándola en una dirección inferior a la de la librería 8BP (por ejemplo, la 23000 podría servir). En caso de hacer esto, tu juego BASIC deberá comenzar con un

comando MEMORY que especifique una dirección inferior a este nuevo límite, por ejemplo, MEMORY 22999 serviría.

A partir de la versión 34 de 8BP, la librería se ensambla a partir de la dirección 24000 por lo que, si quieras usar espacio extra para la música, deberá ocupar direcciones inferiores a la 24000. Por ejemplo, desde la 23500 hasta la 23999.

Este es el

```
; tras ensamblarlo, salvalo con save "musica.bin",b,32200,1400
org 32200
-----MUSICA-----
; tiene la limitacion de tan solo poder incluir un solo fichero de ;
instrumentos para
; todas las canciones. la limitacion se solventa simplemente metiendo
todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO
read "../MUSIC/nibiru5.mus.asm" ;

; archivos de musica
SONG_0:
INCBIN      "../MUSIC/attack5.mus" ;
SONG_0_END:

SONG_1:
INCBIN      "../MUSIC/nibiru5.mus" ;
SONG_1_END:

org 23500 ; esta linea la uso porque no me cabe la tercera cancion !!

SONG_2:
INCBIN      "../MUSIC/gorgo3.mus" ;

SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
_END_MUSIC
```

Este mismo tipo de solución es aplicable para el caso en el que no te quepan todos tus gráficos en la zona reservada por 8BP, aunque como dispones de 8540 bytes para gráficos es menos probable que tengas ese problema.

## 20 Programación en C con 8BP

Al principio de este manual te recomendaba no usar compiladores de BASIC como Fabacom o CPC BASIC 3 por las limitaciones de memoria tan fuertes que imponen (pierdes unas 20KB). Si lo que quieras es aumentar la velocidad de tus juegos, a partir de 8BP v40 tienes a tu disposición un “wrapper” (interzaz en C) de la librería 8BP para ser usada desde C, y un pequeño set de comandos BASIC invocables desde C (al cual llamo “minibasic”) para que puedas traducir tu programa BASIC de forma casi inmediata y obtener ese resultado tan veloz que estas buscando.

Con esta nueva funcionalidad tienes 3 opciones:

- 1) **Hacer tu programa 100% en BASIC** ( es decir, no usar la funcionalidad). Esta opción simplifica mucho la tarea de programar pero tienes menos velocidad.
- 2) **Hacer tu programa 100% en C**. Esta opción es compleja pues programar, compilar, buscar y corregir errores, se hace una tarea mucho mas lenta que programando en BASIC
- 3) **Hacer tu programa en BASIC y al final traducir tan solo el ciclo de juego a C**. Esta opción tan fácil como la primera salvo por la ultima fase de traducción a C.

Te voy a explicar la tercera opción, que es muy interesante porque te permite programar en BASIC con la comodidad que tiene (fácil de programar, de detectar y corregir errores etc.). Si quieres programar tu juego enteramente en C, esta explicación te sirve exactamente igual, de modo que no temas y continúa leyendo. Un ejemplo comercial de la opción 3 es el famoso y mítico juego “plaga galáctica”, una producción de Indescomp creada por el excelente programador Paco Suárez en 1984. Le debemos muchas horas de entretenimiento al gran Paco Suárez.

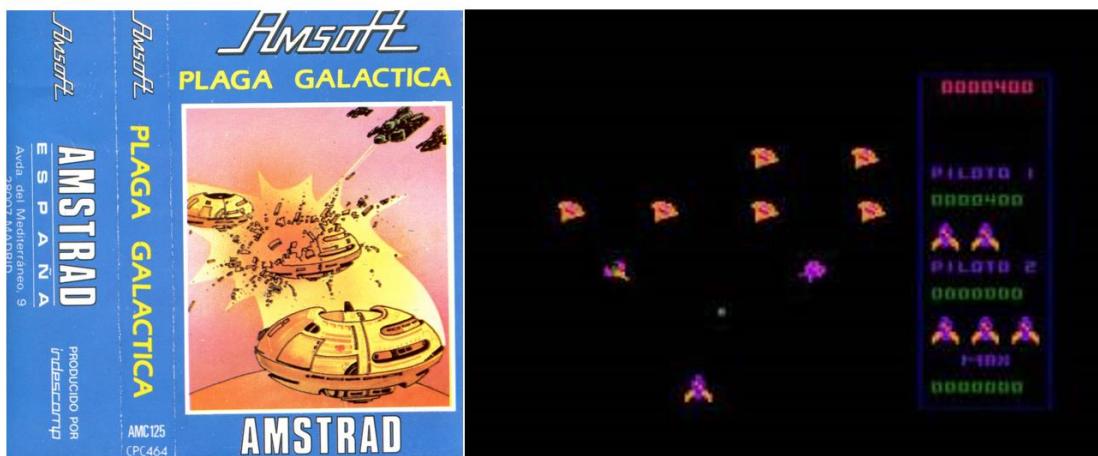


Fig. 112 La “plaga galáctica”

Para poder programar en C necesitamos unas herramientas, pero no te preocupes que no tienes que aprender a usarlas porque 8BP lo hace por ti, gracias a un .bat que se encarga de todo como comprobarás en breve. Las herramientas son:

- **compila.bat**: es el fichero .bat que llama a las diferentes herramientas en cascada y partiendo de un fichero .c obtienes un fichero .dsk con un binario dentro. Esta herramienta la encontrarás en el subdirectorio "C" de 8BP
- **SDCC** ( Small Device C compiler): debes descargarla e instalarla. La encontrarás en <http://sdcc.sourceforge.net/> Este es posiblemente el mejor compilador de C para Z80 (aunque SDCC también soporta otros microprocesadores). Existe otra

famosa opción, el Z88dk pero preferí escoger el SDCC porque algunos análisis revelan que el programa compilado resultante con SDCC es mas rápido que el equivalente a z88dk.

- **Hex2bin.exe:** la usaremos ( desde el .bat) para pasar a binario el fichero que generamos en hexadecimal con SDCC. No hace falta que te la descargues, es pequeña y la encontrarás en el subdirectorio “C” de 8BP
- **ManageDsk.exe:** la usaremos (desde el .bat) para meter nuestro binario compilado en un fichero .dsk . No hace falta que te la descargues. Es pequeña y la encontrarás en el subdirectorio “C” de 8BP

Vamos a dar los pasos necesarios con un ejemplo y después te detallaré como se invoca a cada una de las funciones de 8BP desde C, así como los nuevos comandos de “minibasic” que 8BPV40 te proporciona para programar en C como si estuvieses en BASIC

## **20.1 Primer paso: programa tu juego BASIC**

Vamos a utilizar el programa de ejemplo que viene con la librería 8BP. Es un juego muy sencillo en el que manejas un soldado que debe esquivar unas bolas que caen del cielo en distintas direcciones. Cada vez que te golpea una bola pierdes puntos, los cuales crecen con el tiempo.



*Fig. 113 El juego de ejemplo*

El listado del juego es el que viene a continuación , en el cual he destacado en rojo la parte que se corresponde con el “ciclo de juego”

```

10 MEMORY 19999
11 LOAD "ciclo.bin",20000: REM cargamos el ciclo de juego compilado
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
30 ON BREAK GOSUB 320
40 CALL &BC02:'restaura paleta por defecto por si acaso
50 INK 0,0:'fondo negro
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124: ' establecemos los límites de la pantalla de
juego
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordenadas del personaje
100 PRINT "SCORE:      FPS:"
110 |SETUPSP,31,0,1+32:' status del personaje
120 |SETUPSP,31,7,1'secuencia de animacion asignada al empezar
130 |LOCATESP,31,y,x:'colocamos al sprite (sin imprimirlo aun)
140 |MUSIC,0,0,0,5: puntos=0

```

```

150 cor=32:cod=32:|COLSPALL,@cor,@cod:' configura comando de colision
151 LOCATE 1,20:INPUT "basic(1) o C(2)", a: IF a=1 THEN 160 ELSE CALL &56b0
152 GOTO 320
160 |PRINTSPALL,0,0,0,0: 'configura comando de impresion
161 POKE &B8B4,0: POKE &B8B5,0: POKE &B8B6,0: POKE &B8B7,0:'reset timer
cpc6128. hace falta pues TIME puede retornar un numero muy grande y puede
dar overflow con DEDFINT
162 t1=TIME
170 '--- ciclo de juego. Esta es la parte que se ha traducido a C ---
180 c=c+1
190 ' lee el teclado y posiciona al personaje
191 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,31,7,1:dir=0 ELSE
|ANIMA,31:x=x+1:GOTO 195
192 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,31,7,2:dir=1 ELSE
|ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:puntos=puntos-1:LOCATE 7,1:PRINT puntos:GOTO
250
221 REM para calcular los FPS tenemos en cuenta que TIME me da en unidades
1/300 segundos y voy a medir cada 20 ciclos. Por tanto fps= 20 ciclos x 300
/ dt, siendo dt= t2-t1
230 IF c MOD 20=0 THEN puntos=puntos+10 :LOCATE 7,1:PRINT puntos:
t2=t1:time:fps=6000/(t1-t2):LOCATE 17,1:PRINT fps
240 IF c MOD 5=0 THEN |SETUPSP,i,9,19:|SETUPSP,i,5,4,RND*3-
1:|SETUPSP,i,0,11:|LOCATESP,i,10,RND*80: i=i+1:IF i=30 THEN i=0
250 IF c<500 THEN GOTO 180
251 '--- end ciclo de juego ---
252 |POKE,42038,puntos
310 '---fin del juego---
320 |MUSIC:INK 0,0: PEN 1:BORDER 0:|PEEK,42038,@puntos
330 LOCATE 3,10:PRINT "FINAL SCORE:";puntos

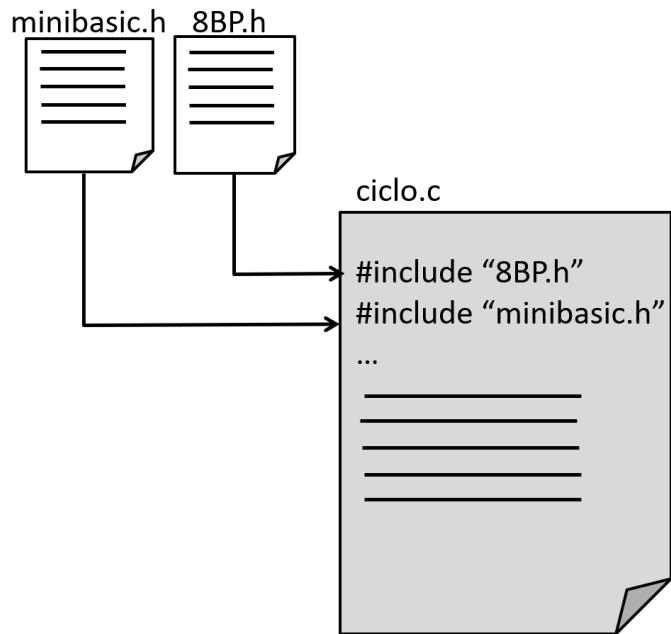
```

## **20.2 Segundo paso: traduce tu ciclo de juego de BASIC a C**

Para traducir el ciclo de juego a C, debemos escribir un programa en C, al que llamaremos ciclo.c

Entra en la subcarpeta “C” de 8BP. Allí encontrarás todo lo que necesitas y este mismo ejemplo, con el fichero ciclo.c.

Lo primero que debes saber a la hora de programar el ciclo de juego en C, es que dispones de dos pequeñas librerías que debes incluir: el wrapper de 8BP ( 8BP.h) y el minibasic (minibasic.h). La 8BP.h está en el subdirectorio “8BP\_wrapper” y la minibasic.h está en el subdirectorio “mini\_basic”



*Fig. 114 ficheros a compilar*

Este es el listado en C, que como ves tiene una correspondencia directa con el trozo de listado BASIC **correspondiente al ciclo de juego**, prácticamente es una traducción literal. Verás unas etiquetas como “label\_195” que hacen las veces de números de línea para poder saltar con GOTO. Es prácticamente el listado BASIC traducido instrucción por instrucción, sin necesidad de volver a pensar como programarlo. En este caso tan sencillo solo hay una función (la función main) y retorna cuando se te acaba el tiempo.

Para dar este paso dispones del “minibasic” para ayudarte en la traducción de BASIC a C, pero si eres experto en C y conoces el firmware del AMSTRAD o dispones de alguna otra librería de ayuda puedes hacer el ciclo de juego directamente en C, sin haberlo programado y validado previamente en BASIC. La forma que te propongo es fácil y potente, tu programa correrá como un galgo, pero aquí eres libre de hacerlo como quieras.

**IMPORTANTE:** recuerda al programar en C que la notación para los números decimales, hexadecimales y binarios es:

```

mivariable = 165 ; //notación decimal
mivariable = 0xA5 ; //notación hexadecimal
mivariable = 0b10100101; //notación binaria

```

```

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "8BP.h"
#include "minibasic.h"

//declaramos las variables todas globales para poder
//acceder a ellas desde cualquier funcion, como en BASIC
// aunque aqui no se inicializan
//-----
int c;

```

```

char dir;
int x;
int y;
int cod;
int cor;
int i;
int puntos;
int t1;
int t2;
int fps;
/******************
 * MAIN
 *****/
int main()
{
    //inicializamos las variables
    c=0;
    dir=0;
    x=40;
    y=100;
    cod=32;
    cor=32;
    i=0;
    puntos=0;
    fps=0;
    t1=_basic_time();

    //configuramos comandos
    _8BP_printspall_4(0,0,0,0);
    _8BP_colspall_2(&cor,&cod);

    //ciclo de juego
    //-----
    label_CICLO:
    c=c+1;

    if (_basic_inkey(27)==0) {
        if (dir !=0) {
            _8BP_setupsp_3(31,7,1);
            dir=0;
        }
        else {
            _8BP_anima_1(31);
            x=x+1;
            goto label_195;
        }
    }
    if (_basic_inkey(34)==0) {
        if (dir !=1) {

```

```

    _8BP_setupsp_3(31,7,2);
    dir=1;
}
else {
    _8BP_anima_1(31);
    x=x-1;
}
}

label_195:
//-----
_8BP_locatesp_3(31,y,x);
_8BP_autoall();
_8BP_printspall();
_8BP_colspall();

if (cod<32) {
    _basic_border(7);
    _basic_sound(1,100,14,0,0,1,0);
    puntos=puntos-1;
    _8BP_setupsp_3(cod,0,9);
    _basic_locate(7,1);
    _basic_print(_basic_str(puntos));
    goto _label_250;

}
else _basic_border(0);
if (c %20 ==0) {
    puntos=puntos+10;
    _basic_locate(7,1);
    _basic_print(_basic_str(puntos));
    t2=t1;t1= _basic_time();
    fps=6000/(t1-t2);
    _basic_locate(17,1); _basic_print(_basic_str(fps));
}

if (c %5 ==0){
    _8BP_setupsp_3(i,9,19);
    _8BP_setupsp_4(i,5,4,_basic_rnd(3)-1);

    _8BP_setupsp_3(i,0,11);_8BP_locatesp_3(i,10,_basic_rnd(80));
    i=i+1;if (i==30) i=0;
}

_label_250:
if (c<500) goto label_CICLO;

_8BP_poke_2(42038,puntos);
return 0;

```

```
}
```

Como ves las funciones de 8BP se invocan como:

```
_8BP_<funcion>_<N>( parámetros )
```

Siendo N el numero de parámetros de la función ya que hay versiones de cada función con diferente número de parámetros (como ocurre en las versiones RSX de los comandos)

Y las funciones de basic que no están disponibles en C pero que hemos creado en el minibasic se invocan así:

```
_basic_<funcion>(parámetros)
```

Como puedes ver, conociendo la traducción de cada comando es muy fácil traducir un listado BASIC de tu ciclo de juego en un listado ciclo.c

Una cosa necesaria que debes hacer es comunicar el LOCOMOTIVE BASIC con el C. Por ejemplo, hay datos que te puede interesar pasar al programa C como el número de vidas que te quedan o los puntos que llevas acumulados. Para eso puedes usar las funciones:

- Desde LOCOMOTIVE BASIC tienes **PEEK**, **POKE**, **|PEEK** y **|POKE**
- Desde C tienes **\_basic.Peek()**, **\_basic.Poke()**, **\_8BP.Peek\_2()**, **\_8BP.Poke\_2()**

Con estas funciones puedes reservar una dirección de memoria para almacenar las vidas, los puntos, la fase, etc y así invocar al ciclo de juego cada vez que te maten con todo el contexto del juego en unas pocas variables que pueden leer y modificar tanto BASIC como C. En el ejemplo puedes ver como se hace esto con la variable “puntos”.

**IMPORTANTE:** aunque todo tu programa sea en C, debes inicializar 8BP con un CALL &6b78, ya que aparte de instalar los comandos RSX, mediante ese call también se inicializan tablas de la librería de uso interno

### 20.2.1 GOSUB y RETURN en C

Los GOSUB deberías traducirlos a funciones de C porque a una rutina GOSUB puedes llegar desde cualquier punto del programa y debes poder volver. En BASIC se vuelve con RETURN al lugar desde donde invocaste GOSUB. En C lo natural es traducirlo a una función, para poder volver al punto del programa desde donde llamaste

Veamos un ejemplo:

BASIC	C
<b>10</b> a=5 <b>20</b> GOSUB 100 <b>30</b> PRINT "PEPE" <b>40</b> end <b>100</b> REM RUTINA <b>110</b> PRINT STR\$(a) <b>120</b> RETURN	#include <stdlib.h> #include <string.h> #include <stdio.h>  <b>#include "8BP.h"</b> <b>#include "minibasic.h"</b>

```

void mifucion(int id);
int a;
int main()
{
a=5;
mifucion(a);
_basic_print(_"PEPE \r\n");

return 0;
}

void mifucion(int a)
{
_basic_print({_basic_str(a)});
_basic_print("\r\n");
}

```

### 20.2.2 Comunicación BASIC a C con variables BASIC

Si estas empezando a usar C con 8BP puedes pasar al siguiente paso. Este es un apartado “avanzado” para enseñarte a comunicar entre el BASIC y el C con variables BASIC en lugar de con PEEK/POKE, pero no es imprescindible.

Para comunicar el BASIC con el C, en lugar de una dirección de memoria y PEEK/POKE, puedes usar también una variable que exista en BASIC. Es algo más complejo, pero se puede hacer. Vamos a ver como se hace con una variable sencilla, y después lo veremos con variables de tipo array.

Lo primero que debes saber es como averiguar donde BASIC almacena una variable. Para ello disponemos del operador “@”. Vamos a ver un sencillo ejemplo:

```

10 DEFINT A-Z: 'importante para que el tipo de datos sea int
20 a=5
30 print @a: 'imprime la dirección memoria donde se almacena a
40 poke @a,7: 'esto es lo mismo que hacer a=7
50 PRINT a : ' esto imprime un 7

```

Hay un pequeño “error” en el programa. Se trata de la instrucción POKE @a,7 pues sólo almacena un byte y la variable “a” tiene 2 bytes porque es un entero. Este caso funciona porque el byte más significativo de “a” es cero, pero si “a” tuviese un valor superior a 255 entonces su byte más significativo no sería cero y el POKE solo estaría alterando el byte menos significativo. Un POKE de 8BP funcionaría siempre porque es de 16 bit:

|POKE,@a,7: 'mete un 7 en la variable “a”

Sabiendo esto, sólo debemos pasarle a C la dirección donde se almacena nuestra variable BASIC. Para ello disponemos de la instrucción |POKE de 8BP que funciona con 16 bits (ten en cuenta que una dirección de memoria ocupa 16 bit). Vamos a pasar la dirección de la variable “a” en la dirección 40000, aunque podríamos usar cualquier otra que esté libre.

```
|POKE, 40000, @a: 'dejamos @a en la dirección 40000
```

Un método alternativo en BASIC es usar dos POKEs:

```
dir=@a:
```

```
POKE 40001, INT (dir/256)
```

```
POKE 40000, INT (dir MOD 256)
```

Lo que hemos escrito en la dirección 40000 es la dirección de memoria donde la variable a esta almacenada.

**IMPORTANTE:** el BASIC puede reubicar de sitio una variable cuando nuevas variables son creadas. Esto significa que si le pasas a una rutina de C una dirección de memoria mediante un |POKE y posteriormente creas nuevas variables BASIC, puede que la dirección de memoria de la variable que has pasado haya cambiado. **Únicamente se garantiza que no cambia si no se crean nuevas variables.** El siguiente ejemplo lo ilustra muy bien, se producen 2 cambios de ubicación

<pre>10 DIM a(100): i=0 20 PRINT @a(0) 30 b=2: 'nueva variable reubica a() 40 PRINT @a(0) 50 c=2: 'nueva variable reubica a() 60 PRINT @a(0) 70 goto 20</pre>	
---	--

La solución a este problema es declarar todas las variables al principio del programa

<pre>10 dim a(100) 20 b=2 30 c=3 40 print @a(0) 70 goto 40</pre>	
--	--

Ahora desde C podemos acceder a la variable “a” de dos modos, aunque el segundo modo (mediante una variable de C “mapeada”) es el mas interesante.

<pre>// variables globales int dir_a; int dato;  int main() {     //vamos a guardar en dir_a la dirección de la variable a     _8BP_peek_2(40000, &amp;dir_a);     _8BP_peek_2(dir_a, &amp;dato); //esto mete en la variable "dato" de C, el     //valor de la variable "a" de BASIC. Es una forma de leer el valor de "a"     _8BP_poke_2(dir_a,7); //esto mete en la variable "a" de BASIC un 7     return 0; }</pre>
---

Vamos a ver el mismo ejemplo de otro modo, con una variable en C que está “mapeada” a la variable basic. Para ello debemos usar la notación con “\*”

```
// variables globales
int dir;
int *a;

int main()
{
//vamos a guardar en dir_a la direccion de la variable a
_8BP_peek_2(40000, &dir);
a=dir; //para evitar warning al compilar usa a=(int*)dir
*a=5; //esto mete un 5 en la variable “a” de BASIC
return 0;
}
```

Hemos visto como funciona con variables sencillas. Ahora vamos a ver como funcionan los arrays de BASIC para acceder a ellos desde C. Para ello lo primero que vamos a hacer es entender como almacena el BASIC los datos de un array en memoria

<pre> 1 DEFINT a-z 2 MODE 2 10 a=5 20 PRINT "la dir de a es @a=";@a 25 PRINT "-----" 30 DIM b(5) 40 FOR i=0 TO 5 50 PRINT "la dir de b(";i;") es ";@b(i) 60 NEXT 70 PRINT "-----" 80 DIM c(3,4) 90 FOR j=0 TO 4:FOR i=0 TO 3 100 PRINT "la dir de c(";i;",";j;") es ";@c(i,j) 110 NEXT:NEXT  120  POKE,40000,@c(0,0)  Con la linea 120 hemos guardado en la dirección 40000, la dirección de memoria donde se almacena el primer dato del array bidimensional  Podríamos guardar la del array unidimensional con  POKE,40000, @b(0)</pre>	<pre> la dir de a es @a= 681 ----- la dir de b( 0 ) es 698 la dir de b( 1 ) es 700 la dir de b( 2 ) es 702 la dir de b( 3 ) es 704 la dir de b( 4 ) es 706 la dir de b( 5 ) es 708 ----- la dir de c( 0 , 0 ) es 727 la dir de c( 1 , 0 ) es 729 la dir de c( 2 , 0 ) es 731 la dir de c( 3 , 0 ) es 733 la dir de c( 0 , 1 ) es 735 la dir de c( 1 , 1 ) es 737 la dir de c( 2 , 1 ) es 739 la dir de c( 3 , 1 ) es 741 la dir de c( 0 , 2 ) es 743 la dir de c( 1 , 2 ) es 745 la dir de c( 2 , 2 ) es 747 la dir de c( 3 , 2 ) es 749 la dir de c( 0 , 3 ) es 751 la dir de c( 1 , 3 ) es 753 la dir de c( 2 , 3 ) es 755 la dir de c( 3 , 3 ) es 757 la dir de c( 0 , 4 ) es 759 la dir de c( 1 , 4 ) es 761 la dir de c( 2 , 4 ) es 763 la dir de c( 3 , 4 ) es 765 Ready</pre>
---	--

El programa anterior nos enseña como almacena BASIC las variables

- Los datos de los arrays unidimensionales los almacena todos seguidos. Cada dato ocupa 2 bytes porque estamos trabajando con enteros

Por ejemplo, en BASIC escribes:

```
DIM b(20)
|POKE,40000,@b
```

```
CALL <rutina C>:'direccion donde se haya ensamblado main()
```

```

PRINT b(8)

y desde C escribes:
int dir;
int *b; //variable con la que vamos a acceder al array BASIC
int main(){
    _8BP_peek_2(40000, &dir);
    b= dir; //b es un puntero y *b[] son variables
    *b[8]=5; //mete un 5 en la variable b(8) de BASIC
    return 0;
}

```

- **Los datos de los arrays bidimensionales** los almacena consecutivos siendo las variaciones de la primera dimensión las que producen datos consecutivos. En el ejemplo con **DIM(3,4)** el dato (2,3) esta a continuación del (1,3) pero el dato (2,3) esta  $4 \times 2 = 8$  bytes mas lejos que el dato (2,2), ya que la primera dimensión del array es 4. **Un DIM (3,4) es un array de dimensiones (4, 5) pues el cero también cuenta.** Puedes comprobarlo imprimiendo @c(3,2) y @c(2,2), verás como hay una diferencia de 8.

Para acceder a los datos desde C se puede hacer con un puntero simple teniendo en cuenta la primera dimensión al acceder. En el siguiente ejemplo he creado un array c(12,16) y para acceder desde C a la posición (2,7) uso  $2+7*13$  , ya que la primera dimensión es  $12+1 = 13$

```

En BASIC escribes:
DIM c(12,16)
|POKE, 40000, @c
CALL <rutina C:>; direccion ensamblado rutina
PRINT c(2,7)

y desde c escribes:
int dir;
int *b; //variable con la que vamos a acceder al array BASIC
int main(){
    _8BP_peek_2(40000, &dir); //lee la dir 40000 y escribe en dir
    c= dir; //c es un puntero y *c[] son variables
    c[2+7*13]=5; //mete un 5 en la variable c(2,7) de BASIC
    return 0;
}

```

Si eres programador de C, sabrás que en C existen punteros dobles que se expresan con un doble asterisco (ejemplo `**c`). A priori parecen adecuados porque podrías referenciar datos con `c[x][y]`. Sin embargo, sólo sirven para guardar memoria reservada dinámicamente con instrucciones “malloc” y “calloc” y requieren memoria tanto para los datos como para los punteros de cada fila. Ello significa que podrías usarlos, pero tendrías que reservar memoria para los punteros. No te los recomiendo.

### 20.2.3 Cadenas de texto en BASIC y en C

Debes saber que una cadena de caracteres en C es un sencillo puntero a char, mientras que una variable de tipo cadena en BASIC (por ejemplo `mivar$="hola"`) se compone de un descriptor de 3 bytes que contiene la dirección de memoria donde se ubica la cadena y la longitud de la misma. Es decir, **una cadena de C y una cadena de BASIC son cosas diferentes.**

Por lo tanto, si quieres imprimir una variable que contiene cadena de texto, no la puedes pasar desde BASIC a C porque no son una misma cosa. Pero puedes imprimir cadenas de texto sin problemas si en lugar de pasárselas desde BASIC, las defines en C, por ejemplo:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "8BP.h"
#include "minibasic.h"

//----variables globales
char* cad; //podríamos inicializarla aqui

//----funciones----
int main(){
cad="has fracasado en tu mision"; //la inicializamos con una frase
_basic_print(cad); //imprime estilo BASIC
_8BP_printat4(0,0,60,cad); //imprime estilo 8BP printat
return 0;
}
```

### 20.3 Tercer paso: Compilar usando “compila.bat”

Estamos a punto de dar el gran paso: compilar para generar un binario de AMSTRAD. Para dar este paso dispones de un .bat de ayuda, llamado “compila.bat”. **Antes de dar este paso debes tener instalado el compilador SDCC en tu ordenador, de lo contrario fallará.** Puedes descargarlo desde <http://sdcc.sourceforge.net/>.

**Nota:** En windows10, la ejecución de SDCC me ha fallado por tenerlo instalado dentro de “Program files” y los espacios en blanco parece que no le sientan bien. Si es tu caso, simplemente instala en un directorio cuyo nombre no incluya espacios en blanco

Una vez instalado SDCC, abre una ventana de comandos (el script ses.bat hace eso) y teclea compila.bat. La pantalla cambiará de color: verde si todo ha ido bien y rojo si hay problemas. El script “compila.bat” ejecuta los siguientes pasos:

1. Borra los ficheros de salida del propio script (ciclo.dsk y ciclo.bin entre otros)
2. Invoca a SDCC para compilar tu programa
3. Traduce la salida de SDCC a binario mediante la herramienta hex2bin
4. Inserta el binario generado en un disco al que llama ciclo.dsk, usando la herramienta manageDsk

Tras ejecutar “compila.bat”, obtendrás una pantalla roja en los siguientes casos:

- Errores de sintaxis de tu programa C
- El fichero ciclo.dsk que va a generar compila.bat está abierto por el winape. En este caso debes conectar otro disco a winape para que compila.bat pueda recrearlo.
- Errores de compilación como una librería que has tratado de usar y no has incluido, etc.

En caso de error de sintaxis de C, u otro tipo de errores de C o si el fichero ciclo.dsk que va a generar compila.bat esta abierto por el winape, obtendrás una pantalla roja de error como esta:

```
8888  BBBB  PPPPPP
88  88  BB  BB  PP  PP
88  88  BB  BB  PP  PP
8888  BBBB  PPPPP
88  88  BB  BB  PP
88  88  BB  BB  PP
8888  BBBB  PPPP
8 bits de poder . Un tributo al AMSTRAD CPC
Jose Javier Garcia Aranda 2016-2020

*****
*      compilacion con SDCC          *
*****
borramos los ficheros de compilacion anterior
*****



main.c : compilamos y linkamos, generando un main.ihx
*****
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --
BP_wrapper -Imini_BASIC ciclo.c
sdcc: Calling preprocessor...
sdcc: sdcpp -nostdinc -Wall -std=c11 -I"8BP_wrapper" -I"mini_BASIC"
SDCC_CHAR_UNSIGNED -D__SDCC_INT_LONG_REENT -D__SDCC_FLOAT_REENT -D__
__SDCC_VERSION_MINOR=0 -D__SDCC_VERSION_PATCH=0 -D__SDCC_REVISION=1
=1 -D__STDC_NO_THREADS_=1 -D__STDC_NO_ATOMICS_=1 -D__STDC_NO_VLA_
DC_UTF_16_=1 -D__STDC_UTF_32_=1 -isystem "C:\proyectos\proyectos0
" -isystem "C:\proyectos\proyectos09\_personal\8BP\SDCC\bin..\incl
sdcc: Generating code...
ciclo.c:36: syntax error: token -> 'puntos' ; column 8
ciclo.c:37: error 1: Syntax error, declaration ignored at 'fps'
ciclo.c:38: error 1: Syntax error, declaration ignored at 't1'
ciclo.c:41: syntax error: token -> '0' ; column 21
.
"+-----+
"| HAY ERRORES DE COMPIILACION! | "
"+-----+"

C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

*Fig. 115 Compila.bat se queja de que tienes errores de C*

En caso de que todo haya ido bien esta sería la salida

```

transformamos el .ihx en un .bin
*****
hex2bin -output\ciclo.ihx
hex2bin v1.0.1, Copyright (C) 1999 Jacques Pelletier
Lowest address = 00004E20
Highest address = 00005A41

metemos el .bin en un disco de amstrad cpc
*****
managedsk -C -S"output\ciclo.dsk"
managedsk -L"output\ciclo.dsk" -I"output\ciclo.bin"/CICLO.BIN/BIN/20

*****
**          FIN DEL PROCESO          **
**  ASEGUrate DE QUE NO EXCEDES LA DIRECCION 24000      **
** es la (highest address) de la transformacion ihx en bin **
**          se ha generado ciclo.dsk y dentro esta ciclo.bin   **
**          Pasos para cargarlo en el amstrad               **
**  1) carga o ensambla 8BP, con tus graficos, musica etc    **
**  2) carga tu juego BASIC                                **
**  3) ejecuta LOAD "ciclo.bin", 20000                      **
** para invocar a tu programa o rutina simplemente:        **
** call <direccion de main en fichero ciclo.map>           **
**          Para mover ciclo.bin de ciclo.dsk a otro disco debes   **
** conocer su longitud:                                     **
**     longitud=Highest address - Lowest address           **
**     lo cargas desde ciclo.dsk                            **
**     LOAD "ciclo.bin", 20000                             **
**     Y salvas en el disco donde esta tu juego           **
**     SAVE "ciclo.bin",b,20000,longitud                  **
*****
C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>

```

*Fig. 116 Compila.bat produce una pantalla verde: todo ha ido bien*

En caso de obtener la pantalla verde, significa que todos los pasos que ejecuta compila.bat han ido bien ( borSDCC, tendrás informaciones valiosas en pantalla y además, en el subdirectorio output encontrarás ficheros que vas a necesitar:

- Ciclo.dsk
- Ciclo.map

## 20.4 Cuarto paso: comprueba los límites de la memoria

El script compila.bat te muestra en su pantalla verde dos informaciones muy valiosas: “**lowest address**” y “**highest address**”. Estas son las direcciones de memoria donde comienza y termina el binario generado. El script “compila.bat” utiliza la dirección 20000 como dirección de compilación en la invocación de SDCC y si el fichero binario resultante es muy grande, podría exceder de la dirección 24000, con lo que dañaría la librería 8BP. Debes asegurarte de que la “**highest address**” es inferior a 24000. En caso de no ser inferior, **debes modificar en el script “compila.bat” la invocacion a SDCC**

para que compile asignando una dirección inferior a 20000. De este modo tu nuevo binario y la librería 8BP no se solaparán. Debes modificar dos líneas del script compila.bat. La primera es la que invoca a SDCC. Es una línea muy larga. Debes cambiar el parámetro 20000 por la nueva dirección

```
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --fomit-frame-pointer --opt-code-size -I8BP_wrapper -lmini_BASIC -o output/ciclo.c
```

La segunda línea que debes modificar es la que invoca a managedsk para que quede coherente con la nueva dirección de memoria. Esta es la línea y como puedes ver, también aparece la dirección 20000.

```
managedsk -L"output\ciclo.dsk" -I"output\ciclo.bin"/CICLO.BIN/BIN/20000 -S"output\ciclo.dsk"
```

Obviamente si tu binario comienza en 20000, tu programa BASIC deberá incorporar un MEMORY 19999. Esto te restaría 4KB a tu espacio libre pero también te vas a ahorrar las líneas de BASIC correspondientes al ciclo de juego, de modo que una cosa por la otra y es como si no hubieses perdido nada.

Si la dirección “highest address” es inferior a 24000, **conviene que la ajustes lo más posible, es decir, que sea lo más próxima a 24000, para no desperdiciar memoria.** Ello puede implicar (por ejemplo) usar la dirección 21000 al invocar a SDCC. Hazlo para disponer de la mayor cantidad de memoria para BASIC. Deberás poner un MEMORY coherente con esa dirección en tu programa BASIC. Por ejemplo, si el binario comienza en 21000, tendrás que poner un MEMORY 20999

**En definitiva, puede que tengas que modificar dos líneas en el script “compila.bat” para ajustar la dirección de comienzo de la compilación, que inicialmente he puesto en 20000.**

## 20.5 Quinto paso: localiza la dirección de la función a invocar desde BASIC

Tras compilar con “compila.bat”, en el subdirectorio “output” habrás obtenido un fichero llamado ciclo.map. En este fichero debes buscar la dirección de memoria de la función o funciones que pretendes invocar desde BASIC. En este ejemplo solo vamos a invocar la función main(), que se encuentra en &56b0 como se aprecia en este fragmento del fichero ciclo.map

0000566B	__basic_paper
00005682	__basic_plot
00005699	__basic_move
000056B0	main
00005920	_abs
0000592C	_strlen
0000593B	__modschar
00005948	__modsint
00005954	__moduchar

*Fig. 117 la dirección de cada función está en ciclo.map*

Esto significa que para invocar la función main() desde BASIC simplemente debemos hacer:

**CALL &56B0**

Ten cuidado pues si haces algún cambio en la fase de compilación (modificación del ciclo.c o cambios en las direcciones de memoria del script compila.bat) la dirección de cada función puede variar al volver a compilar.

## **20.6 Sexto paso: Incluir en tu juego.dsk el nuevo binario**

Ya has generado un fichero ciclo.dsk que contiene el archivo ciclo.bin que debes cargar para poder invocar la función main (y/o las funciones que quieras). Para que tanto este fichero como tu juego estén en el mismo disco, debes seleccionar ciclo.dsk desde winape y simplemente cargar el ciclo.bin

**LOAD "CICLO.BIN", 20000**

Luego desde el menú de winape seleccionas tu disco (donde tengas tu juego) y salvas este binario

**SAVE "CICLO.BIN", b, 20000, <longitud>**

Siendo longitud =highest address – lowest address +1

Ahora en tu fichero loader.bas deberás cargar este nuevo binario adicional e invocarlo desde tu listado BASIC con CALL <dirección>

```
10 MEMORY 24999
15 LOAD "!pant.scr",&c000: 'solo si tu juego tiene pantalla de carga
20 LOAD "tujuego.bin"
25 LOAD "ciclo.bin", 20000
50 RUN "!tujuego.bas"
```

Esto es todo. Ya puedes programar en C con 8BP!

## 20.7 Referencia de funciones 8BP en C

RSX	C prototype
3D, 0	void _8BP_3D_1(int flag);
3D, <flag>, #, offsety	void _8BP_3D_3(int flag, int sp_fin, int offsety);
ANIMA, #	void _8BP_anima_1(int sp);
ANIMALL	void _8BP_animall();
AUTO, #	void _8BP_auto_1(int sp);
AUTOALL, <flag enrutado>	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, umbral_ascii, @colision, #	void _8BP_colay_3(int umbral, int* colision, int sp);
COLAY, @colision, #	void _8BP_colay_2(int* colision, int sp);
COLAY, #	void _8BP_colay_1(int sp);
COLAY	void _8BP_colay();
COLSP, #, @collided%	/* operation 32, ini,fin or operation 34,dy,dx*/
COLSP, 32, ini, fin	void _8BP_colsp_3(int operation, int a, int b);
COLSP, 33, @collided%	/*operation 33 or sp*/
COLSP, #	void _8BP_colsp_2(int sp, int* colision);
COLSP, 34, dy, dx	void _8BP_colsp_1(int sp);
COLSPALL,@quien%,@conquien%	void _8BP_colspall_2(int* collider, int* collided);
COLSPALL, colisionador	void _8BP_colspall_1(int collider_ini);
COLSPALL	void _8BP_colspall();
LAYOUT, y, x, @string\$	void _8BP_layout_3(int y, int x, char* cad);
LOCATESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x);
MAP2SP, y, x	void _8BP_map2sp_2(int y, int x);
MAP2SP, status	void _8BP_map2sp_1(unsigned char status);
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy,int dx); void _8BP_mover_1(int sp);
MOVERALL, dy,dx	void _8BP_moverall_2(int dy, int dx); void _8BP_moverall();
MUSIC, C, flag, canción, speed	void _8BP_music_4(int flag_c, int flag_repetition,int song, int speed);
MUSIC, flag, canción, speed	void _8BP_music();
MUSIC	
PEEK, dir, @variable%	void _8BP.Peek_2(int address, int* dato);
POKE, dir, valor	void _8BP.poke_2(int address, int dato);
PRINTAT, flag, y, x, @string	void _8BP.printat_4(int flag,int y,int x,char* cad);
PRINTSP, #, y, x	void _8BP.printsp_1(int sp) ;
PRINTSP, #	void _8BP.printsp_2(int sp, int bits_background) ;
PRINTSP,32, bits	void _8BP.printsp_3(int sp,int y,int x) ;
PRINTSPALL, ini, fin, anima, sync	void _8BP.printspall_4(int ini, int fin, int flag_anima, int flag_sync);
PRINTSPALL, ordermode	void _8BP.printspall_1(int order_type);
PRINTSPALL	void _8BP.printspall();
RINK,tini,color1,color2,...,colorN	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
RINK, salto	
ROUTESP, #, pasos	void _8BP_routesp_2(int sp, int pasos); void _8BP_routesp_1(int sp);
ROUTEALL	void _8BP_routeall();
SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits (int xmin, int xmax, int ymin, int ymax)
SETUPSP, #, param_number, valor	void _8BP_setupsp_3(int sp, int param, int value);
SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_4(int sp, int param, int value1,int value2);
STARS, initstar, num, color, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int color, int dy, int dx); void _8BP_stars();
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin);

A continuación, tienes algunos ejemplos de uso, que complementan el ejemplo utilizado en la traducción del BASIC a C.

RSX	<b> 3D, &lt;flag&gt;, #, offset</b>
	10  3D,1,10,200
C	<b>void _8BP_3D_3(int flag, int sp_fin, int offset);</b>  <b>_8BP_3D_3(1, 10, 200);</b>

RSX	<b> COLSPALL,@quien%,@conquien%</b>
	10 collider%=0: collided%=0 20  COLSPALL, @collider%, @colliderd%
C	<b>void _8BP_colspall_2(int* collider, int* collided);</b>  Int cor=0; Inr cod=0; <b>_8BP_colspall_2 (&amp;cor, &amp;cod);</b>

RSX	<b> RINK,tini,color1,color2,...,colorN</b> <b> RINK, salto</b>
	10  RINK,1,2,2,3,3 20  RINK,1
C	<b>void _8BP_rink_N(int num_params,int* ink_list);</b> <b>void _8BP_rink_1(int step);</b>  Int tintas[5]={1,2,2,3,3}; <b>_8BP_rink_N(5,tintas);</b> <b>_8BP_rink_1(1);</b>

RSX	<b> LAYOUT, y, x, @string\$</b>
	10 cad\$="AA AAAA AAAA AAAA" 20  LAYOUT,10,1,@cad\$
C	<b>void _8BP_layout_3(int y, int x, char* cad);</b>  <b>_8BP_layout_3(10,1," AA AAAA AAAA AAAA");</b>  O bien:  <b>char* cad=" AA AAAA AAAA AAAA";</b> <b>_8BP_layout_3(10,1,cad)</b>

RSX	<b> PRINTAT, flag, y, x, @string</b>
	10 cad\$=str\$(125) 20  PRINTAT,0,100,40,@cad\$
C	<b>void _8BP_printat_4(int flg,int y,int x,char* cad)</b>  <b>char* cad=_basic_str(125);</b> <b>_8BP_printat_4(0,100,40, cad);</b>

## 20.8 Referencia de funciones BASIC en C (“minibasic”)

Este es el pequeño set de instrucciones similares a BASIC que 8BP te proporciona a través de la librería minibasic.h, para que traduzcas fácilmente tu listado BASIC a C. Si tratas de traducir únicamente el ciclo de juego, este set de instrucciones será suficiente.

<b>BASIC</b>	<b>C prototype</b>
BORDER	<code>void _basic_border(char color); //example _basic_border(7)</code>
CALL	<code>void _basic_call(unsigned int address); // example _basic_call(0xbd19)</code>
DRAW	<code>void _basic_draw(int x, int y);</code>
INK	<code>void _basic_ink(char ink1,char ink2);</code>
INKEY	<code>char _basic_inkey(char key); //takes around 0.3 ms. slow but simple</code>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y); // example: _basic_locate(2,25);_basic_print("TEST");</code>
MOVE	<code>void _basic_move(int x, int y);</code>
PAPER	<code>void _basic_paper(char ink);</code>
PEEK	<code>char _basic_peek(unsigned int address);</code>
GRAPHICS PEN	<code>void _basic_pen_graph(char ink);</code>
PEN	<code>void _basic_pen_txt(char ink);</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char data);</code>
PLOT	<code>void _basic_plot(int x, int y);</code>
PRINT	<code>void _basic_print(char *cad); //example: _basic_print("Hola \r\nAdios")</code>
RND	<code>unsigned int _basic_rnd(int max); //example: num=_basic_rnd(50)</code>
SOUND	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod);</code>
STR\$	<code>char* _basic_str(int num); //similar to STR\$ //example: _basic_print(_basic_str(num))</code>
TIME	<code>unsigned int _basic_time(); //return an unsigned int,(0..65535). As integer, when // reach 32768 go to -32768</code>



## 21 Guía de referencia de la librería 8BP

### 21.1 Funciones de la librería

#### 21.1.1 |3D

Este comando activa la proyección 3D en los comandos PRINTSP y PRINTSPALL.

Para proyectar disponemos del comando |3D

Uso

Para activar la proyección 3D:

|3D, 1, <Sprite\_fin>, <offsety>

Para desactivarla:

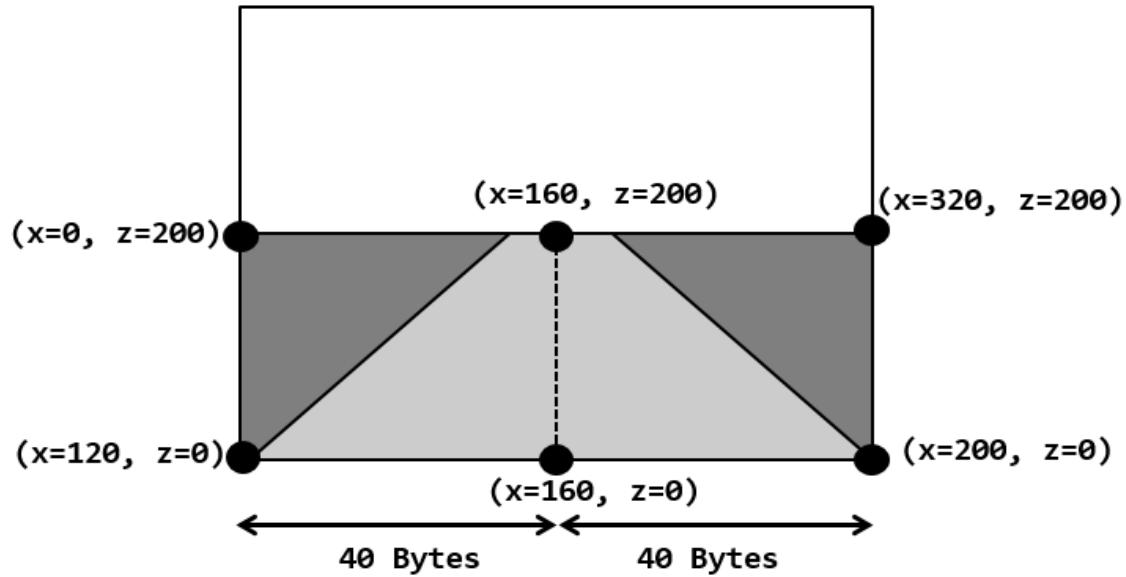
|3D, 0

Los sprites afectados son desde el Sprite 0 hasta el <Sprite\_fin>. Este comando activa la proyección 3D en el comando |PRINTSP y en |PRINTSPALL. Esto significa que antes de imprimirse en pantalla se calcularán las coordenadas “proyectadas” y a continuación se imprimirán en pantalla. Las coordenadas de los sprites no se ven afectadas, es decir, las coordenadas 2D en la tabla de sprites seguirán siendo las mismas.

Este comando **no afecta a los mecanismos de colisión**, es decir, si usamos COLSPALL y detectamos una colisión entre sprites proyectados, la colisión se está produciendo en el plano 2D.

En cuanto al último parámetro <offsety> es para proyectar más arriba o más abajo, de modo que podamos ubicar los marcadores del juego donde queramos. AL proyectar la pantalla, que mide 200 de alto, se transforma en 100 pixels de alto, de modo que podemos escoger a que altura ubicamos la proyección. Si un Sprite no es afectado por la proyección por ser superior a <Sprite\_fin>, entonces tampoco le afecta el <offsety>.

La siguiente figura representa cuales son las coordenadas del mapa del mundo que se proyectan en ciertos puntos representativos de la pantalla cuando |MAP2SP es invocado con (yo=0, xo=0).



*Fig. 118 coordenadas del mundo proyectadas*

Si en lugar de  $(x_0=0, y_0=0)$  usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen, estarán desplazadas en  $(x, z)$  lo que se indique con  $x_0$  e  $y_0$ .

### 21.1.2 |ANIMA

Este comando cambia el fotograma de animación de un sprite, teniendo en cuenta su secuencia de animación asignada

Uso:

**|ANIMA, <sprite number>**

Ejemplo:

**|ANIMA, 3**

El comando lo que hace es consultar la secuencia de animación del sprite, y si es distinta de cero entonces se va a la tabla de secuencias de animación (la primera secuencia valida es la 1 y la última es la 31). Escoge la imagen cuya posición es la siguiente al fotograma actual y actualiza el campo frame de la tabla de atributos de sprites.

Si en la secuencia el siguiente fotograma es cero entonces se cicla, es decir, se escoge el primer fotograma de la secuencia.

Además de cambiar el campo frame, se cambia el campo image y se le asigna la dirección de memoria del lugar donde se almacena el nuevo fotograma.

|ANIMA no imprime el sprite, pero lo deja preparado para cuando se imprima, de modo que se imprima el siguiente fotograma de su secuencia

|ANIMA no verifica que el flag de animación esté activo en el byte de estado del sprite. De hecho, nuestro personaje normalmente solo lo vamos a querer animar cuando se mueva y no siempre que se imprima.

Si la secuencia de animación es una “secuencia de muerte” (incluye un “1” en su último fotograma), entonces al llegar al frame cuya dirección de memoria de imagen sea 1, el sprite pasará a inactivo.

La librería 8BP te permite hacer “secuencias de muerte”, que son secuencias que, al terminar de recorrerlas, el sprite pasa a estado inactivo. Esto se indica con un simple “1” como valor de la dirección de memoria del fotograma final. Estas secuencias son muy útiles para definir explosiones de enemigos que están animados con |ANIMA o |ANIMALL. Tras alcanzarles con tu disparo, les puedes asociar una secuencia de animación de muerte y en los siguientes ciclos del juego pasarán por las distintas fases de animación de la explosión, y al llegar a la última pasarán a estado inactivo, no imprimiéndose más. Este paso a inactivo se hace automáticamente, de modo que lo que debes hacer es simplemente chequear la colisión de tu disparo con los enemigos y si colisiona con alguno le cambias el estado con |SETUPSP para que no pueda colisionar más y le asignas la secuencia de animación de muerte, también con |SETUPSP.

Si usas una secuencia de muerte, no te olvides de que el último fotograma antes de encontrar el “1” sea uno completamente vacío, de modo que no quede ningún resto de la explosión.

Ejemplo de secuencia de muerte

```
dw EXPLOSION_1,EXPLOSION_2,ExPLOSION_3,1,0,0,0,0
```

### 21.1.3 |ANIMALL

Este comando anima todos los sprites que tengan el flag de animación activado en el byte de estado. Este comando no tiene parámetros

Uso

|ANIMALL

**IMPORTANTE:** desde la versión v37 de la librería, este comando solo es accesible via CALL (ver tabla de correspondencias en el apéndice), y no mediante comando RSX. Sacarlo de la lista permitió ahorrar algunos bytes de memoria y se puede seguir usando tanto desde un parámetro en PRINTSPALL como desde una llamada CALL.

Es recomendable su uso si vas a animar muchos sprites ya que es mucho mas rápido que invocar varias veces al comando |ANIMA

Como normalmente se va a desear invocar a |ANIMALL en cada ciclo de juego, antes de imprimir los sprites, hay una forma de invocar más eficiente y consiste en poner a “1” el parámetro correspondiente del comando |PRINTSPALL, es decir

|PRINTSPALL,1,0

Esta función invoca internamente a |ANIMALL antes de imprimir los sprites, ahorrando 1.17ms respecto de lo que se tardaría en invocar separadamente |ANIMALL y |PRINTSPALL

## 21.1.4 |AUTO

Este comando mueve un sprite (cambia sus coordenadas) de acuerdo a sus atributos de velocidad Vy,Vx. Estos atributos son los que tenga el sprite en la tabla de sprites.

Uso:

**|AUTO, <sprite number>**

Ejemplo:

**|AUTO, 5**

Lo que hace este comando es actualizar las coordenadas en la tabla de sprites, sumando la velocidad a la coordenada actual

Las coordenadas nuevas son

X nueva = coordenada X actual + Vx

Y nueva = coordenada Y actual + Vy

No es necesario que el sprite tenga el flag de movimiento automático activo en el campo status

## 21.1.5 |AUTOALL

Este comando mueve todos los sprites que tengan el flag de movimiento automático activo, de acuerdo a sus atributos de velocidad Vy, Vx.

Uso:

**|AUTOALL, <flag de enrutado>**

Ejemplo

**|AUTOALL,1** invoca a **|ROUTEALL** antes de mover los sprites

**|AUTOALL,0** no invoca a **|ROUTEALL**

**|AUTOALL** se utiliza como parámetro el último valor usado (tiene memoria)

El flag de enrutado es opcional. Puesto que el comando **|ROUTEALL** no modifica las coordenadas de los sprites, deben ser movidos con **|AUTOALL** e impresos (y animados) con **|PRINTSPALL**. Es por ello que dispones de un parámetro opcional en **|AUTOALL**, de modo que **|AUTOALL,1** invoca internamente a **|ROUTEALL** antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

## 21.1.6 |COLAY

Detecta la colisión de un sprite con el mapa de pantalla (el layout). Tiene en cuenta el tamaño de dicho sprite para saber si colisiona, y considera que los elementos del layout miden todos 8x8 pixeles de mode 0 (es decir, 4 bytes x 8 líneas). Se puede invocar con 3,2,1 o ningún parámetro. Si se invoca sin parámetros se usarán los valores de la última llamada con parámetros y es mucho mas rápido.

Uso:

```
|COLAY, <umbral ASCII>, @colision% ,<sprite number>
|COLAY, @colision% ,<sprite number>
|COLAY, <num_sprite>
|COLAY
```

El parámetro opcional <umbral ASCII> basta con usarlo en una primera invocación para establecer el umbral de colisión en el comando |COLAY. Este umbral representa el mayor código ASCII del elemento de layout que es considerado como “no colisión”. Por defecto es 32 (el del espacio en blanco). Para configurar el umbral que deseas, consulta la tabla ASCII del comando |LAYOUT.

Ejemplo:

```
|COLAY, 65, @col%,31 : rem sprite es 31, umbral es 65
```

La variable que uses para colisión puede llamarse como quieras. Yo he puesto “col”

Esta rutina modifica la variable colision (la cual debe ser entera y por eso el “%”) poniéndola a 1 si hay colisión del sprite indicado con el layout. Si no hay colisión el resultado es 0.

```
10 xanterior=x
20 x=x+1
30 |LOCATESP,0,y,x: ' posicionamos el sprite en nueva posicion
40 |COLAY,@colision%,0: 'chequeo de la colision
```

Ahora comprobamos la colisión y si hay colisión lo dejamos en su ubicación anterior

```
50 if colision%=1 then x=xanterior: LOCATESP,0,y,x
```

También puedes utilizar el comando |MOVER para posicionar el sprite y hacer el chequeo

```
10 |COLAY,65,@col,31: 'configuracion. Solo lo hacemos una vez
20 |MOVER,31,1,1: ' lo movemos a la derecha y abajo
30 |COLAY:' lo invocamos sin parámetros (mas rápido)
30 if col THEN MOVER,31,-1,-1 : rem ha colisionado y por eso lo
dejo donde estaba
```

## 21.1.7 |COLSP

Este comando permite detectar la colisión de un sprite con el resto de sprites que tengan el flag de colisión activo

Uso :

Para configurar:

```
|COLSP, 32, <sprite inicial>, <sprite final>
|COLSP, 33, @colision%
|COLSP, 34, dy, dx
```

Para detector colisiones:

**|COLSP,<sprite number>, @colsp%**

Ejemplo:

**col%=0**

**|COLSP,0,@col%**

La función retorna en la variable que le pasemos como parámetro, el número del sprite con el que colisiona, o si no hay colisión retorna un 32 pues el sprite 32 no existe (solo existen del 0 al 31).

**IMPORTANTE:** la variable de colisión del comando COLSP no es la que se usa en el comando COLSPALL. Son variables diferentes (a menos que le pases a ambos comandos la misma variable para que actúen sobre ella)

Al igual que la impresión de sprites con PRINTSPALL, la función COLSP chequea los sprites comenzando en el 31 y acabando en el cero. Si tienen flag de colisión de sprites activo (bit 2 del byte de status) entonces se comprueba la colisión. Si dos sprites colisionan a la vez con nuestro sprite, se retorna el número de sprite mayor pues es el que se comprueba antes.

#### **Invocaciones para configurar el comando:**

Existe una forma de configurar COLSP para que realice menos trabajo chequeando la colisión de menos sprites y así ahorrar tiempo de ejecución. La configuración se la indicaremos con el uso del sprite 32 (el cual no existe).

**|COLSP, 32, <sprite inicial a chequear>, <sprite final a chequear>**

Si por ejemplo los enemigos de nuestro personaje son los sprites 25 al 30, y los configuramos como colisionables (no colisionadores) podemos invocar (una sola vez) al comando así:

**|COLSP, 32, 25, 30**

Con eso estaremos indicando que cualquier invocación posterior al comando |COLSP tan solo debe chequear la colisión de los sprites del 25 al 30 (siempre que tengan el flag “collided” activo).

Esta estrategia permite reducir bastante tiempo, por ejemplo, si solo debemos chequear 6 enemigos, pre-configuremos el comando para que solo chequee desde el 25 en adelante, podemos ahorrar hasta 2.5ms en cada ejecución. Esto se hace especialmente importante en juegos donde el personaje puede disparar, ya que en cada ciclo de juego al menos habrá que chequear la colisión del personaje y de los disparos.

Otra interesante optimización, capaz de ahorrar 1.1 milisegundos en cada invocación, es decirle al comando que siempre use la misma variable BASIC para dejar el resultado de la colisión. Para ello se lo indicaremos usando como sprite el 33, que tampoco existe

**col%=0**

**|COLSP, 33, @col%**

Una vez ejecutadas estas dos líneas, las siguientes invocaciones a COLSP, dejarán el resultado en la variable col, sin necesidad de indicarlo, por ejemplo:

### |COLSP, 23

Por último, es posible ajustar la sensibilidad del comando COLSP, decidiendo si el solape entre sprites debe ser de varios pixeles o de uno solo, para considerar que ha habido colisión.

Para ello se puede configurar el número de pixeles de solape necesario tanto en la dirección Y como en la dirección X, usando el comando COLSP y especificando el sprite 34 (que no existe)

### |COLSP, 34, dy, dx

Los valores por defecto para dy y dx son 2 y 1 respectivamente. Ten en cuenta que en la dirección Y se consideran pixeles, pero en la dirección X se consideran bytes (un byte son dos pixeles en mode 0).

Para una detección con un solape mínimo (de un pixel en vertical y/o un byte en horizontal) debes hacer:

### |COLSP, 34, 0, 0

## 21.1.8 |COLSPALL

Uso:

Para configurar:

|COLSPALL,@colisionador%, @colisionado%

Para comprobar las colisiones

|COLSPALL  
|COLSPALL, <colisionador inicial>

Esta función comprueba quien ha colisionado (entre el grupo de sprites que tengan a “1” el flag de colisionador del byte de status) y con quien ha colisionado (entre el grupo de sprites que tengan a “1” el flag de colisión del byte de status).

Es una función muy recomendable cuando tienes que manejar colisiones de tu personaje y de varios disparos, ya que ahorra invocaciones a |COLSP y, por consiguiente, acelera tu videojuego.

**Importante:** los colisionadores (bit 5 de estado) se comprueban desde el 31 hasta el 0. Para cada colisionador, los colisionables (bit 1 de estado) también se comprueban desde el 31 al 0. Ello nos permite determinar quién va a colisionar en caso de solape múltiple

En caso de invocar a COLSPALL con un único parámetro,

|COLSPALL, <colisionador inicial>

Se explorarán los colisionadores desde el colisionador indicado -1 hasta el sprite cero, en orden descendente. De este modo si necesitas detectar más de una colisión por ciclo de

juego, podrás hacerlo invocando sucesivamente a **COLSPALL**, <colisionador> hasta que la variable colisionador tome el valor 32

Ejemplo:

|COLSPALL, 7 : rem busca colisiones a partir del colisionador 6

## 21.1.9 |LAYOUT

Uso:

|LAYOUT, <y>, <x>, <@cadena\$>

Ejemplo:

cadena\$ = "XYZZZZ ZZ"

|LAYOUT, 0,1, @cadena\$

ojo, usar |LAYOUT, 0,1, "XYZZZZ ZZ" sería incorrecto en un CPC464 aunque funciona en un CPC6128. Además, en CPC6128 puedes obviar el uso de la "@" pero en CPC464 es obligatorio.

Esta rutina imprime una fila de sprites para construir el layout o "laberinto" de cada pantalla. Además de dibujar el laberinto, o cualquier gráfico en pantalla construido con pequeños sprites de 8x8, también podrás detectar las colisiones de un sprite con el layout, usando el comando |COLAY

Los sprites a imprimir se definen con un string, cuyos caracteres (32 posibles) representan a uno de los sprites siguiendo esta sencilla regla, donde la única excepción es el espacio en blanco que representa la ausencia de sprite.

Caracter	Sprite id	Codigo ASCII
" "	NINGUNO	32
";"	0	59
"<"	1	60
"=="	2	61
">"	3	62
"?"	4	63
"@"	5	64
"A"	6	65
"B"	7	66
"C"	8	67
"D"	9	68
"E"	10	69
"F"	11	70
"G"	12	71
"H"	13	72
"I"	14	73
"J"	15	74
"K"	16	75
"L"	17	76
"M"	18	77

“N”	19	78
“O”	20	79
“P”	21	80
“Q”	22	81
“R”	23	82
“S”	24	83
“T”	25	84
“U”	26	85
“V”	27	86
“W”	28	87
“X”	29	88
“Y”	30	89
“Z”	31	90

*Tabla 6 correspondencia entre caracteres y Sprites para el comando /LAYOUT*

**IMPORTANTE:** Tras imprimir el layout puedes cambiar los sprites para que sean personajes, por lo que seguirás disponiendo de los 32 sprites

Las coordenadas y,x se pasan en formato caracteres. La librería mantiene internamente un mapa de 20x25 caracteres, por lo que las coordenadas toman los siguientes valores:

y toma valores [0,24]

x toma valores [0,19]

Los sprites a imprimir deben ser de 8x8 píxeles. son "ladrillos" ("bricks" en inglés). A este tipo de concepto también se le suele llamar "tiles" (azulejos)

Si usas otros tamaños de sprite, esta función no funcionará bien. Realmente imprimirá los sprites, pero si un sprite es grande tendrás que colocar espacios en blanco para dejarle espacio.

La librería mantiene un mapa interno del layout y esta función actualiza los datos del mapa interno del layout de modo que será posible detectar colisiones. Dicho mapa es un array de 20x25 caracteres, donde cada carácter se corresponde con un sprite

El @string es una variable de tipo cadena. no puedes pasar directamente la cadena, aunque en el CPC6128 el paso de parámetros lo permite, pero sería incompatible con CPC464

#### Precauciones:

La función no valida la cadena que le pasas. Si contiene minúsculas u otro carácter diferente de los permitidos puede provocar efectos indeseados, tales como el reinicio o cuelgue del ordenador. ¡Tampoco puede ser una cadena vacía!

Los límites establecidos con SETLIMITS deben permitir que se imprima donde deseas. Si posteriormente quieres hacer clipping en una zona más reducida puedes invocar de nuevo a SETLIMITS cuando todo el layout este impreso

Ejemplo:

2070  SETLIMITS,0,80,0,200	En este ejemplo se usan varios ladrillos que
2090 c\$(1)= "PPPPPPPPPPPPPPPPP P"	previamente se han creado con la herramienta
2100 c\$(2)= "PU P"	"SPEDIT"
2110 c\$(3)= "P P"	; sprite 20 --> O arbusto
2120 c\$(4)= "P P"	

```

2130 c$(5)= "P    TPPPPPU   TPPPPPPPPP"
2140 c$(6)= "P                      TP"
2150 c$(7)= "P                      P"
2160 c$(8)= "P                      P"
2170 c$(9)= "P      YYYYYYYYYYY  P"
2190 c$(10)="P      TPPPPPPPPPU  P"
2195 c$(11)="P                      P"
2200 c$(12)="P                      P"
2210 c$(13)="P                      P"
2220 c$(14)="YYYYYYYYYYY  PYYYYYYY"
2230 c$(15)="RRRRRRRRRRR  RRRRRRRR"
2240 c$(16)="PPPPPPPPPPP  PPPPPPPP"
2250 c$(17)="PU        TP  PU  TP"
2260 c$(18)="P          T  U   P"
2270 c$(19)="P                      P"
2271 c$(20)="P                      P"
2272 c$(21)="P                      W  P"
2273 c$(22)="PP          W  PP"
2274 c$(23)="PPPPPPPPPPPPPPPPPPPPPPPP"

2280 for i=0 to 23
2281 |LAYOUT,i,0,@c$(i)
2282 next

; sprite 21 --> P roca
; sprite 22 --> Q nube
; sprite 23 --> R agua
; sprite 24 --> S ventana
; sprite 25 --> T arco de puente derecho
; sprite 26 --> U arco de puente izq
; sprite 27 --> V bandera
; sprite 28 --> W planta
; sprite 29 --> X pico de torre
; sprite 30 --> Y césped
; sprite 31 --> Z ladrillo


```



## 21.1.10 |LOCATESP

Este comando cambia las coordenadas de un sprite en la tabla de atributos de sprites

Uso

| LOCATESP, <sprite number>, <y>, <x>

## Ejemplo

| LOCATE\$P,0,10,20

Una alternativa a este comando, si solo deseamos cambiar una coordenada es usar el comando POKE de BASIC, insertando en la dirección de memoria ocupada por la coordenada X o Y, el valor que queramos. Si deseamos introducir una coordenada negativa es necesaria el comando |POKE, ya que con el POKE de BASIC sería ilegal

El comando **LOCATE** no imprime el sprite, sólo lo posiciona para cuando sea impreso.

## 21.1.11 |MAP2SP

Esta función recorre el mapa del mundo que se describe en el fichero map\_table.asm y transforma en sprites los map ítems que puedan estar entrando en pantalla parcial o totalmente.

Uso

| MAP2SP, <y>, <x>

| MAP2SP, <status>

### Ejemplo

| MAP2SP, 1500, 2500

Los sprites creados por MAP2SP se crean por defecto con estado 3, es decir, con el flag de impresión activo (**|PRINTSPALL** lo imprime) y con el flag de colisionable activo (**|COLSP** colisionará con él). Si necesitas que los sprites sean creados con otro estado, simplemente debes invocar una vez al comando **|MAP2SP** con un solo parámetro indicando el estado con el que se deben crear los sprites.

Si por un casual **|MAP2SP** se encuentra con más de 32 items para traducir a sprites, ignorará los que excedan de 32.

#### **|MAP2SP, <status>**

**|MAP2SP , 1 : REM esto configura al comando MAP2SP para que se impriman pero no sean colisionables**

**IMPORTANTE:** en el mapa del mundo puedes combinar imágenes normales con “imágenes de fondo” (apartado 8.5). Las imágenes de fondo siempre tienen transparencia. El flag de transparencia que uses en **|MAP2SP, <status>** sólo aplicará a las imágenes normales

Los parámetros **<y>,<x>** de la función son el origen móvil desde el cual se muestra el mundo en la pantalla. Hay otros tres parámetros que se encuentran en la **MAP\_TABLE**, la tabla con la que se define el mundo. Estos parámetros son el alto máximo, el ancho máximo (en negativo) y el número de ítems del mundo (máximo 82)

Cada ítem es una tupla de 3 parámetros precedidos por el nemónico “DW”:

**DW Y, X, <imagen>**

```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50 ; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
dw -40 ; maximo ancho de un sprite por si se cuela por la izquierda
(numero negativo)
db 64 ; numero de items del mapa a considerar. como mucho debe ser 82

; a partir de aqui comienzan los items
dw 100,10,CASA; 1
dw 50,-10,CACTUS;2
dw 210,0,CASA;3
dw 200,20,CACTUS;4
dw 100,40,CASA;5
dw 160,60,CASA;6
dw 70,70,CASA;7
dw 175,40,CACTUS;8
dw 10,50,CASA;9
dw 250,50,CASA;10
dw 260,70,CASA;11
dw 290,60,CACTUS;12
dw 180,90,CASA;13
dw 60,100,CASA;14
...
```

### **21.1.12 |MOVER**

Este comando mueve un sprite de forma relativa, es decir, sumando a sus coordenadas unas cantidades relativas

Uso:

**|MOVER, <sprite number>, <dy>, <dx>**

Ejemplo:

**|MOVER, 0, 1, -1**

El ejemplo mueve el sprite 0 hacia abajo y hacia la derecha a la vez. No es necesario que el sprite tenga el flag de movimiento relativo activado

Hay una forma de usar **|MOVER** sin especificar ni “dy” ni “dx”. Para ello indicaremos el sprite 32, que no existe, y pondremos como parámetros las direcciones de memoria de las variables que queremos usar para almacenar tanto “dy” como a “dx”.

La dirección de memoria de una variable se obtiene simplemente anteponiendo el símbolo “@”

Ejemplo:

**dy% = 5**

**dx% = 2**

**|MOVER, 32, @dy, @dx**

A partir de este momento podremos usar:

**|MOVER, <id>**

Y con ello el sprite “id” se moverá según indiquen las variables dy, dx. Este mecanismo también funciona con **|MOVERALL**

### **21.1.13 |MOVERALL**

Este comando mueve de forma relativa todos los sprites que tengan el flag de movimiento relativo activado

Uso

**|MOVERALL, <dy>, <dx>**

Ejemplo

**|MOVERALL, 2, 1**

El ejemplo mueve todos los sprites con flag de movimiento relativo hacia abajo (2 líneas) y 1 byte hacia la derecha.

Si no se especifican parámetros, se usarán las variables especificadas en la invocación de MOVER con sprite 32, es decir

**|MOVER, 32, @dy, @dx**

**|MOVERALL**

Es equivalente a **|MOVERALL, dy, dx**

Este uso “avanzado” del comando evita el paso de parámetros en cada invocación y por lo tanto es más rápido, lo cual es fundamental en nuestros programas BASIC

### 21.1.14 |MUSIC

Este comando permite que una melodía comience a sonar

Uso:

|MUSIC,<flag\_canal\_C>,<flag\_repetición>, <numero\_melodía>, <velocidad>  
|MUSIC, <flag\_repetición>, <numero\_melodía>, <velocidad>

|MUSIC :' sin parámetros termina de sonar la musica

El flag canal-C con valor 1 permite dejar libre el tercer canal de sonido de modo que se puede usar para hacer efectos sonoros (disparos etc) con el comando

SOUND 4,<nota>,<duración>, ...

Fíjate que debes usar el canal 4, ya que en BASIC se tipifican los canales como A=1, B=2, C=4

Si el flag de canal se omite o se pone a cero, entonces la música usará los 3 canales y no podrás usar el comando SOUND a la vez (podrás pero con efectos raros)

El flag de repetición debe ser cero para reproducir en bucle. Si solo quieres que suene una vez la música, debes usar el valor 1

El número de la melodía estará comprendido entre 0 y 7.

La velocidad “normal” es 6. Si usamos un número superior se reproducirá más lentamente y si el número es inferior se reproducirá más deprisa.

El comando |MUSIC invocado sin parámetros desactiva la interrupción de música y deja de sonar cualquier melodía.

Ejemplos:

|MUSIC,0,0,0,6  
|MUSIC,1,0,0,6  
|MUSIC,0,0,6  
|MUSIC

Internamente el comando lo que hace es instalar una interrupción que se dispara 300 veces por segundo. Si ponemos velocidad 6, una de cada 6 veces que se dispara, se ejecuta la función de reproducción musical

Al basarse en una interrupción, es necesario que haya un programa en ejecución para que pueda sonar la música, pues mientras el intérprete BASIC se encuentra esperando a recibir comandos, dichas interrupciones no están habilitadas. Si ejecutas simplemente el comando |MUSIC, no oirás nada, pero si lo ejecutas dentro de un programa como el que se muestra a continuación, la música sonará

```
10 |MUSIC,0,0,5
20 goto 20: ' bucle infinito. Al estar en ejecución, la música suena
```

### 21.1.15 |PEEK

Este comando lee el valor de un dato de 16 bit de una dirección de memoria dada. Está pensado para consultar las coordenadas de sprites que se mueven con movimiento automático o relativo

Uso  
**|PEEK, <dirección>, @dato%**

Ejemplo  
**dato%=0**  
**|PEEK, 27001, @dato%**

Si las coordenadas son solo positivas y menores de 255 puedes usar el comando PEEK de BASIC, ya que es algo más rápido.

### 21.1.16 |POKE

Este comando introduce un dato de 16 bit (positivo o negativo) en una dirección de memoria. Está pensada para modificar coordenadas de sprites, ya que el comando POKE no puede manejar coordenadas negativas o mayores de 255 ya que POKE funciona con bytes mientras que |POKE es un comando que funciona con 16bit

Uso:  
**|POKE, <dirección>, <valor>**

Ejemplo:  
**|POKE, 27003, 23**

Este ejemplo pone el valor 23 en la coordenada x del sprite 0.  
Es una función muy rápida, aunque si vas a manejar solo coordenadas positivas es mejor usar POKE pues es más rápida aun

### 21.1.17 |PRINTAT

|PRINTAT puede imprimir una cadena de caracteres usando un nuevo juego de caracteres más pequeño (los llamo “minicaracteres”). Este nuevo comando permite usar el mecanismo de transparencia de los sprites, de modo que podrás imprimir caracteres respetando el fondo. Funciona del siguiente modo:

Uso:  
**|PRINTAT,<flag transparencia>, y,x,@string**

Ejemplo:  
**cad\$= “Hola”**  
**|PRINTAT,0,100,10, @cad\$**

El comando |PRINTAT imprime cadenas de caracteres y no variables numéricas, por lo que si quieres imprimir un número (por ejemplo, los puntos en el marcador de tu videojuego) debes hacerlo así:

```
puntos=puntos+1
cad$= str$(puntos)
|PRINTAT,0,100,10, @cad$
```

El comando |PRINTAT no se ve afectado por los límites para el clipping establecidos con |SETLIMITS. Esto es lo más lógico puesto que normalmente usarás PRINTAT para imprimir puntuaciones en tus marcadores, que se encontrarán fuera del área delimitada por |SETLIMITS

A diferencia del comando PRINT del BASIC, el comando |PRINTAT es bastante rápido y puede ser utilizado para actualizar los marcadores de tu videojuego con frecuencia. PRINTAT usa un alfabeto redefinible, que puede contener una versión reducida o diferente de los caracteres “oficiales” del Amstrad. Por defecto 8BP proporciona un pequeño alfabeto compuesto por números, letras mayúsculas y algunos símbolos. Es el siguiente:

**"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ: ! , . "**

No podrás usar un carácter que no se encuentre entre este conjunto, como las letras minúsculas. Si tratas de hacerlo, se imprimirá el último carácter definido en la cadena (en este caso el espacio).

Los caracteres de dicho alfabeto miden todos lo mismo: 4pixels de ancho x 5 pixeles de alto, es decir 2 bytes x 5 líneas.

El alfabeto por defecto no contiene minúsculas y faltan muchos símbolos, aunque puedes crear tu propio alfabeto que los contenga.

### **21.1.18 |PRINTSP**

Uso:

**|PRINTSP, <sprite id >, <y >,<x>**  
**|PRINTSP, <sprite id >**  
**|PRINTSP, 32, <numero de pixels de fondo>**

Ejemplo:

Imprime el sprite 23 en las coordenadas y=100, x=40 (actualizando sus coordenadas)

**|PRINTSP, 23,100,40**

Ejemplo:

Imprime el sprite 23 en las coordenadas que ya tenga asignadas en la tabla de sprites:

**|PRINTSP, 23**

Si se especifica el sprite 32 (el cual no existe), entonces el siguiente parámetro se utiliza para especificar el número de bits de fondo en la impresión transparente. Si se usa 1 bit, entonces se podrán usar 2 colores de fondo. Si se especifica un 2, entonces se podrán usar 4 colores de fondo.

Si se especifica un sprite\_id <32 entonces el comando imprime un sprite en la pantalla, y si se especifican coordenadas, además las actualiza.

Las coordenadas consideradas son:

- Número de líneas en vertical [-32768..32768]. Las correspondientes al interior de la pantalla son [0..199]
- Número de bytes en horizontal [-32768..32768]. Las correspondientes al interior de la pantalla son [0..79]

Normalmente en la lógica de un videojuego harás uso de **|PRINTSPALL**, ya que es más rápido imprimirllos todos de golpe. Sin embargo, en otros momentos del juego puede interesarte imprimir sprites por separado. En este ejemplo se muestra la bajada de un “telón”, usando un solo sprite que se repite horizontalmente y al ir bajando va “tiñendo” de rojo la pantalla, dando la sensación de un telón que baja

```

7089 telon=&8ec0
7090 |setupsp,1,9,telon
7100 for y=8 to 168 step 4
7110 for x=12 to 64 step 4
7111 |PRINTSP,1,y,x
7112 next
7113 next

```



*Fig. 119 Un ejemplo de uso de PRINTSP*

### 21.1.19 |PRINTSPALL

Esta rutina imprime de una sola vez todos los sprites que tengan el bit0 de estado activo.

Uso:

**|PRINTSPALL, <ordenini>, <ordenfin>, <flag anima>, <flag sync>**  
**|PRINTSPALL, < tipo de orden>,**

Ejemplo:

Con los siguientes valores, el comando imprime todos los sprites animándolos primero y sin sincronizar con barrido, y sin ordenar:

**|PRINTSPALL, 0, 0, 1, 0**  
**|PRINTSPALL, 0, 1, 0: rem si se omite el ordenini, toma el ultimo valor asignado, o cero si nunca se asignó**

**flag de animación**, puede valer 1 o 0. Si se asigna un 1, entonces antes de imprimir los sprites se cambia el fotograma en su secuencia de animación, siempre que los sprites tengan el bit 3 de estado activo. **IMPORTANTE**: la animación se hace antes de imprimir, no después de imprimir. Eso significa que, si acabas de asignar una secuencia de animación, no verás el primer fotograma de dicha secuencia.

**El <flag sync>** es un flag de sincronización con el barrido de pantalla. Puede ser 1 o 0 . La sincronización solo tiene sentido si compilas el programa con un compilador como “Fabacom”. La lógica en BASIC se ejecuta lentamente y sincronizar con el barrido produce pequeñas esperas adicionales en cada ciclo del juego de modo que no es conveniente.

Como regla general, solo es conveniente si tu juego es capaz de generar 50fps por segundo, o lo que es lo mismo, un ciclo completo de juego cada 20 milisegundos. Si compilas el juego con un compilador como “fabacom”, entonces es recomendable que sincronices con el barrido de pantalla porque casi seguro que vas a alcanzar esos 50fps y

si los superas, tu juego producirá más fotogramas de los que puede mostrar la pantalla y entonces algunos no se podrán mostrar y el movimiento no será suave.

Cuantos más sprites tengas en pantalla imprimiéndose, más tardará el comando, aunque es muy rápido. Hay muchos sprites que pueden aparecer en pantalla, pero no es necesario imprimir (pueden tener el bit 0 de estado desactivado) como pueden ser frutas, monedas, elementos de bonus en general y/o personajes que no se mueven y no tienen animación. Aunque no se impriman pueden tener el bit de colisión activo y así afectar en la rutina |COLSP y |COLSPALL

**Los parámetros de orden (“ordenini”, “ordenfin”)** indican los sprites inicial y final que definen el grupo de sprites ordenados por coordenada “Y” que vamos a imprimir. Por ejemplo, si asignamos los valores 0,0 entonces se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si asignamos 0,8 se imprimirán del 0 al 8 ordenados (9 sprites) y del 10 al 31 de modo secuencial. Si ponemos un 0,31 se imprimirán todos los sprites ordenados. Si ponemos un 10,20 se imprimirán secuencialmente los sprites del 0 al 9, luego se imprimirán ordenados del 10 al 20 y finalmente se imprimirán secuencialmente del 21 al 31

El ordenamiento es muy útil para hacer juegos tipo “Renegade” o “Golden AXE”, donde es necesario dar un efecto de profundidad.

Si el parámetro “ordenini” se omite, se considera el ultimo valor asignado, o bien cero si nunca se ha asignado un valor. Además, si vas a modificar alguno de los dos parámetros de ordenamiento, conviene primero ejecutar PRINTSPALL,0,0,0,0 para que primero se reordenen los sprites secuencialmente antes de ordenarlos con una nueva configuración.

Imprimir de forma ordenada es más costoso computacionalmente que imprimir de forma secuencial. Si solo tienes 5 sprites que deben ser ordenados, pasa un 4 como parámetro de ordenamiento, no pases un 31. Ordenar todos los sprites lleva unos 2.5 ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizás tengas muchos sprites y no merezca la pena ordenar algunos, como los disparos o sprites que sabes que no se van a solapar.

El ordenamiento de |PRINTSPALL es parcial, es decir, solo se ordena una pareja de sprites desordenados en cada invocación. Puede que alguna vez deseas que la ordenación sea completa en cada fotograma. Es decir, que no se ordene un par de sprites en cada invocación a |PRINTSPALL, sino tener la seguridad de que todos están ordenados. La librería 8BP te lo permite mediante sus cuatro modos de ordenamiento, que puedes establecer mediante la invocación del comando |PRINTSPALL con un solo parámetro (basta con ejecutarlo una vez para fijar el modo de ordenación):

<b>PRINTSPALL,0 : ordenamiento parcial usando Ymin</b>
<b>PRINTSPALL,1 : ordenamiento completo usando Ymin</b>
<b>PRINTSPALL,2 : ordenamiento parcial usando Ymax</b>
<b>PRINTSPALL,3 : ordenamiento completo usando Ymax</b>

Los ordenamientos que usan Ymax se basan en la coordenada Y mayor de los sprites, es decir, donde se encuentran sus pies en lugar de su cabeza. Si los sprites son del mismo tamaño, un ordenamiento basado en Ymin te puede servir, pero si los sprites tienen diferente altura puede que deseas ordenar según donde se encuentren los pies de cada personaje y para ello tendrás que usar el modo de ordenar 2 o el 3.

Los modos de ordenar con Ymax son mas lentos, aproximadamente 0.128 ms por sprite, de modo que úsalos cuando los necesites realmente.

La ordenación completa consume muy poco mas que la parcial (aproximadamente 0.3ms). Esto es debido a que los sprites apenas se desordenan de un fotograma al siguiente, pero incluso esos 0.3ms merece la pena ahorrarlos si es posible.

Hay un comportamiento de esta función muy interesante para ahorrar 1ms en su ejecución. Consiste en invocarla con parámetros una vez y las siguientes veces invocarla sin parámetros. En ese caso se asumirán que, aunque no se pasen parámetros, sus valores son iguales a los últimos que se pasaron. De esta manera el analizador sintáctico trabaja menos y reduce el tiempo de ejecución.

### 21.1.20 |RINK

Este comando permite realizar una animación por tintas.

Uso:

|RINK, <tinta\_inicial>, <color1>, <color2>, . . . , <colorN>  
|RINK, <step>

RINK rota un conjunto de tintas comenzando en la tinta inicial N tintas (cualquier numero de tintas), según el tamaño del patrón de color que se defina.

La velocidad de rotación se puede controlar con el parámetro step, que indica el número de saltos de color que realiza cada tinta en una invocación.

Recomendación: debido al uso de interrupciones |RINK produce “parones” en algunos casos cuando se usa a la vez que el comando |MUSIC a velocidad 6. En caso de querer usar ambos a la vez sin que haya interferencias, usa otra velocidad para la música (puedes usar velocidad 5 o 7, ambas te funcionarán bien).

Ejemplos:

Este comando define un patrón de 4 rojos (color =3) y 4 amarillos (color =24) a rotar desde la tinta 8 hasta la tinta 15

|RINK, 8, 3, 3, 3, 3, 24, 24, 24, 24

Este comando define un patrón de 2 blancos (color =26) y 2 grises (color=13) a rotar desde la tinta 3 hasta la tinta 6

|RINK, 3, 26, 26, 13, 13

Rota un color del patrón todas las tintas

|RINK, 1

En un patrón de 8 colores, el siguiente comando deja todo como estaba

|RINK, 8

Este comando no haría nada, salvo forzar que las tintas adopten el color del patrón

|RINK, 0

### 21.1.21 |ROUTEALL

Este comando te permite enrutar a todos los sprites que tengan activo el flag de ruta en su byte de estado a través de la ruta que tengan asignada (parámetro 15 de |SETUPSP)

Uso:

|ROUTEALL

No tiene parámetros por lo que es muy sencillo de invocar. Este comando lo que hace internamente es llevar una cuenta de pasos por el segmento que está cursando cada sprite, de modo que, si se acaba el segmento, altera la velocidad del sprite.

El comando no modifica las coordenadas de los sprites, de modo que deben ser movidos con |AUTOALL e impresos (y animados) con |PRINTSPALL. Es por ello que dispones de un parámetro opcional en |AUTOALL, de modo que |AUTOALL,1 invoca internamente a |ROUTEALL antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

Las rutas se definen en el fichero de rutas routes\_tujuego.asm

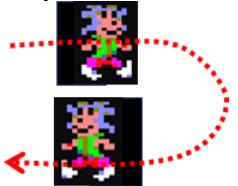
```
; DEFINICION DE CADA RUTA
;=====
ROUTE0; un circulo
;-----
db 5,2,0
db 5,2,-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0
```

El último segmento es cero, indicando que la ruta se ha terminado y que el sprite debe comenzar desde el principio. Asegúrate de que el número de pasos de cada segmento no excede de 250 y que tanto Vy como Vx están entre -127 y 127.

Para asignar una ruta a un sprite debes usar el comando SETUPSP especificando el parámetro 15. El siguiente ejemplo asocia la ruta 3 al sprite 31

|SETUPSP, 31, 15, 3

Existen 4 funcionalidades que puedes usar en mitad (**ojo en mitad, no al final**) de cualquier ruta:

Código de escape (campo “número de pasos”)	Descripción	Ejemplo
<b>255</b>	Cambio de estado del sprite.	DB 255, 3, 0 Estado pasa a valor 3. El cero del final es de relleno
<b>254</b>	Cambio de secuencia de animación del sprite  Tras cambiar la secuencia, si quieres que también cambie la imagen debes usar el código 251	DB 254, 10, 0 Se asocia la secuencia 10. El cero es de relleno. Si la secuencia asignada es la que ya tiene el sprite, entonces es inocuo (no se reinicia el frame id) En caso de querer reiniciar el frame id, el tercer parámetro debe ser un 1 , por ejemplo: DB 254, 10, 1
<b>253</b>	Cambio de imagen 	DB 253 DW new_img Se asocia la imagen “new_img” que debe ser una dirección de memoria
<b>252</b>	Cambio de ruta	DB 252, 2, 0 Se asocia la ruta 2
<b>251</b>	Pasa al siguiente frame de la animación. 	DB 251, 0, 0 Se anima el Sprite. Los dos ceros son de relleno

**IMPORTANTE:** ten mucho cuidado de escribir DB y DW donde deben usarse, es decir, por ejemplo, si cambias de imagen debes preceder la imagen con DW y no con DB. Si cometes un error de este tipo, tu ruta no funcionará.

**IMPORTANTE:** Una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, por lo tanto, una ruta puede tener como mucho 84 segmentos. Es posible que necesites construir una ruta aún más larga y en ese caso podrás hacerlo concatenando el fin de una ruta con un cambio de ruta hacia otra ruta (código 252), y puedes concatenar tantas rutas como deseas.

**IMPORTANTE:** los códigos de escape puedes emplearlos en mitad de una ruta, pero el ultimo segmento no puede ser un código de escape, debe ser un movimiento, aunque sea quedarse quieto, algo como “DB 1,0,0”

En estos casos, |ROUTEALL interpretará que se debe forzar un cambio de estado, secuencia, imagen, ruta del sprite o animar y además, ejecutar el paso siguiente. Los cambios se pueden forzar en cualquier segmento de la ruta, no hace falta que sea al final, y se pueden forzar tantos cambios como se desee.

```

ROUTE0; un disparo a la izquierda
;-----
db 100,0,-1; cien pasos a la izquierda a velocidad Vx=-1
db 255,0,0; deactivacion del sprite con status=0
db 1,0,0 ; no mover nada en este paso
db 0
ROUTE1; un salto a la derecha
db 253
dw SOLDADO_R1_UP
db 1,-5,1
db 2,-4,1
db 2,-3,1
db 2,-2,1
db 2,-1,1
db 253
dw SOLDADO_R1_DOWN
db 1,-5,1; subo para que UP y down encajen
db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDADO_R1
db 1,5,1; baja una mas
db 255,13,0; nuevo estado, sin flag ruta y con flag animacion
db 254,32,0; macrosecuencia 32
db 1,0,0; quietooo.!!!!
db 0

```

### 21.1.22 |ROUTESP

Este comando te permite enrutar un solo Sprite que tenga activo el flag de ruta en su byte de estado a través de la ruta que tengan asignada (parámetro 15 de SETUPSP)

Uso:

|ROUTESP, <spriteid>, <pasos>  
|ROUTESP, <spriteid> : rem en este caso “pasos” se considera=1

Este comando mueve un sprite el número de pasos que deseas (**hasta 255**) a lo largo de la ruta que tenga asignada. El comando hace recorrer al sprite todos los pasos que se indiquen, dejándolo finalmente ubicado en la misma posición que tendría si hubiésemos ejecutado |AUTOALL,1 un número de veces igual al número de pasos.

**IMPORTANTE:** pasos no puede tomar un valor superior a 255

### **21.1.23 |SETLIMITS**

Este comando establece los límites del área donde se van a poder imprimir sprites o estrellas.

Uso:

**|SETLIMITS, <xmin>, <xmax>, <ymin>, <ymax>**

Ejemplo que establece toda la pantalla como área permitida

**|SETLIMITS,0,80,0,200**

Fuera de estos límites se realiza clipping de los sprites, de modo que, si un sprite se encuentra parcialmente fuera del área permitida, las funciones |PRINTSP y |PRINTSPALL imprimirán solo la parte que se encuentra dentro del área permitida.

### **21.1.24 |SETUPSP**

Este comando carga datos de un sprite en la SPRITES\_TABLE

Uso:

**|SETUPSP, <id\_sprite>, <param\_number>, <valor>**

Ejemplo:

**|SETUPSP, 3, 7, 2**

Permite por ejemplo asignar una nueva secuencia de animación cuando el sprite cambia de dirección, o simplemente cambiar su registro de flags de status

Con esta función podemos cambiar cualquier parámetro de un sprite, menos X, Y (que se hace con LOCATE\_SPRITE)

Solo podremos cambiar un parámetro a la vez. El parámetro que vamos a cambiar se especifica con *param\_number*. El *param\_number* es en realidad la posición relativa del parámetro en la SPRITES\_TABLE

<b>Param number</b>	<b>Acción</b>	<b>Possible uso de POKE o  POKE como alternativa</b>
0	cambia el status (ocupa 1 byte)	SI
5	cambia Vy (ocupa 1byte, valor en líneas verticales). También se puede modificar Vx a la vez si lo añadimos al final como parámetro	SI
6	cambia Vx (ocupa 1byte, valor en bytes horizontales)	SI
7	cambia secuencia (ocupa 1byte, toma valores 0..31)	NO, porque  SETUPSP además reinicia el frame_id y también le asigna la dirección de la primera imagen.
8	cambia frame_id (ocupa 1byte, toma valores 0..7)	SI

9	cambia dir imagen (ocupa 2bytes). La imagen especificada puede ser una de la lista inicial de imágenes del fichero images_mygame.asm,	No es igual si se usa una imagen <255. Si se usa una dirección de memoria podría usarse  POKE como alternativa
15	cambia la ruta (ocupa 1bytes)	<b>NO</b> , porque SETUPSP hace más cosas en tablas internas para que funcione la ruta

Ejemplo:

En este ejemplo le hemos dado al sprite 31 la imagen de una nave que está ensamblada en la dirección &a2f8

```
nave = &a2f8
|SETUPSP, 31, 9, nave
```

Hay una forma más sencilla de especificar la imagen para el sprite haciendo uso de la lista IMAGE\_LIST que aparece en el fichero images\_tujujeo.asm. Si tenemos la NAVE en la IMAGE\_LIST, podremos asociar un identificador entre 16 y 255

|SETUPSP,31, 9, 16 : rem el 16 es el identificador de la NAVE en la IMAGE\_LIST

<pre>IMAGE_LIST ----- ; pondremos aqui una lista de las imagenes que queremos usar sin especificar la direccion de memoria desde basic ; de este modo el comando  SETUPSP,&lt;id&gt;,9,&lt;address&gt; se transforma en  SETUPSP,&lt;id&gt;,9,&lt;numero&gt; ; la ventaja de no usar direcciones de memoria en BASIC es que si ampliamos los graficos o se reensamblan en ; direcciones diferentes, el numero que asignemos no cambiara ; NO tienen que tener todas un numero, solo aquellas que vamos a usar con  setupsp, id, 9,&lt;num&gt; ; se empiezan a numerar en 16 ; podemos usar hasta 255 imagenes especificadas de este modo ; no hace falta que la lista tenga 255 elementos. es de longitud variable, incluso puede estar vacia ----- DW NAVE ; 16 DW OTRA_NAVE ; 17  ----- BEGIN IMAGE ----- NAVE db 7 ; ancho db 12 ; alto db 0 , 0 , 0 , 0 , 0 , 0 , 0 db 0 , 0 , 0 , 0 , 0 , 0 , 0 db 0 , 154 , 48 , 0 , 0 , 0 , 0 db 0 , 112 , 240 , 48 , 0 , 0 , 0 db 0 , 207 , 207 , 112 , 12 , 0 , 0 db 0 , 84 , 240 , 48 , 164 , 8 , 0 db 0 , 0 , 48 , 176 , 112 , 12 , 0 db 0 , 69 , 48 , 112 , 48 , 101 , 0 db 0 , 16 , 48 , 207 , 207 , 0 , 0 db 0 , 207 , 207 , 80 , 0 , 0 , 0 db 0 , 0 , 0 , 0 , 0 , 0 , 0 db 0 , 0 , 0 , 0 , 0 , 0 , 0 ----- END IMAGE -----</pre>
---

En el caso de param\_number=5, podemos incluir Vx como parámetro al final:  
|SETUPSP, 31, 5, Vy, Vx

De este modo actualizaremos las dos velocidades con un solo comando, que cuesta 3.73ms frente a los 6.8 que costaría invocar a dos comandos separadamente.

En el caso de usar param\_number=7, además de cambiar la secuencia de animación, automáticamente el comando actualiza el fotograma (frame id), colocándolo en el inicial (el cero) y se actualiza la dirección de la imagen, de modo que ya no necesitas invocar con param\_number=9 para que cambie la imagen del sprite a la primera imagen de la nueva secuencia asignada. Si estás usando |ANIMALL antes de imprimir o |PRINTSPALL con flag de animación, aunque SETUPSP te coloque la animación en el frame cero, saltarás al frame 1 antes de imprimir. Esto normalmente no va a suponer ningún problema, pero en caso de tratarse de una secuencia de muerte en la que por ejemplo el primer frame es para borrar al sprite, puede que no te interese pasar directamente al frame 1. En ese caso un sencillo truco puede ser repetir el frame cero en la definición de la secuencia de muerte. Así te aseguras de que dicho frame se vea. Otra opción es quitarle el flag de animación y animarle con ANIMASP después de imprimir.

En el caso de param\_number=15, además de asignar la ruta al sprite en la tabla de atributos, el comando realiza un reseteo de datos internos para que el sprite comience a recorrer dicha ruta a partir del primer segmento de la ruta en cuestión.

### 21.1.25 |STARS

Mueve un banco de hasta 40 estrellas en la pantalla (dentro de los límites establecidos por |SETLIMITS), sin pintar sobre otros sprites que ya existiesen impresos.

|STARS,<estrella inicial>,<num estrellas>,<color>,<dy>,<dx>

Ejemplo:

|STARS, 0, 15, 3, 1, 0

El ejemplo desplaza 15 estrellas de color 3 (rojo) un píxel verticalmente (ya que dy=1 y dx=0). Invocado repetidas veces da sensación de fondo de estrellas que se desplaza. Cuando una estrella se sale del límite de la pantalla o el establecido por |SETLIMITS, reaparece por el lateral opuesto, de modo que hay sensación de continuidad en el fluir de las estrellas.

El banco de estrellas está situado en la dirección 42540 (=A62C) y tiene capacidad para 40 estrellas, llegando hasta la dirección 42619. Cada estrella consume 2 bytes, uno para la coordenada Y, y el otro para la coordenada X.

Se pueden mover grupos de estrellas por separado, comenzando en la estrella que quieras. Las coordenadas iniciales de las estrellas deben ser inicializadas por el programador.

Ejemplo de inicialización y uso en un scroll de cuatro planos de estrellas para dar sensación de profundidad. Cada plano va a moverse a una velocidad diferente

```
1 MEMORY 24999  
10 CALL &6b78: rem instala los commandos RSX
```

```

20 banco=42540
30 FOR star=0 TO 39: ' bucle para crear 40 estrellas
40 POKE banco+star*2,RND*200
50 POKE banco+star*2+1,RND*80
60 NEXT
70 MODE 0
80 REM vamos a pintar y mover 4 planos de estrellas de 10 estrellas
cada uno
90 |STARS,0,10,3,0,-1: ' el 3 es rojo. Las mas lejanas se mueven mas
despacio
91 |STARS,10,10,2,0,-2: ' el 2 es azul
92 |STARS,20,10,1,0,-3: ' el 1 es amarillo
93 |STARS,30,10,4,0,-4: ' el 4 es blanco. Las mas cercanas van mas
deprisa
95 goto 90

```

Los usos de este comando pueden ser muy diversos.

- Usando varios bancos de estrellas a la vez con diferente velocidad y color puedes dar sensación de profundidad
- Si la dirección de las estrellas es diagonal puedes hacer un “efecto de lluvia”
- Si el color es negro y el fondo es marrón o naranja puedes dar sensación de avance sobre un territorio arenoso
- Si el movimiento es de balanceo y el color de las estrellas es blanco puedes dar sensación de nieve. El movimiento de balanceo lo puedes lograr con un zigzag en X manteniendo la velocidad en Y, o incluso usando funciones trigonométricas como el coseno. Obviamente si usas el coseno en la lógica de un juego va a ser muy lento, pero puedes almacenar el valor del coseno precalculado en un array.

Ejemplo de efecto nieve:

```

1 MEMORY 23999: MODE 0
30 ' inicializacion banco de 40 estrellas
40 FOR dir=42540 TO 42619 STEP 2
45 POKE dir,RND*200:POKE dir+1,RND*80
48 NEXT
50 |STARS,0,20,4,2,dx1
60 |STARS,20,20,4,1,dx2
61 dx1=1*COS(i):dx2=SIN(i)
69 i=i+1: IF i=359 THEN i=0
70 GOTO 50

```

Existe un modo de conseguir una ejecución más rápida, y es evitando pasar parámetros. A lo largo de este libro hemos visto como el paso de parámetros es costoso incluso aunque el comando invocado no haga nada. Pues bien, estamos ante un comando que requiere 5 parámetros por lo que es especialmente costoso. Si queremos reducir el tiempo que requiere el BASIC para interpretar los parámetros, simplemente podemos invocar una vez el comando con parámetros y las siguientes veces no pasar parámetros.

|STARS,0,10,1,5,0

|STARS :' esta invocación sin parámetros asume los mismos valores de la última invocación

Esta posibilidad es especialmente útil en juegos donde queremos invocar el comando en cada ciclo de juego para mover estrellas, pues ahorraremos unos 1.7 ms

**IMPORTANTE:** al comando STARS le afectan los límites de **|SETLIMITS** pero solo si Vx o Vy son distintos de cero. Si ambos son cero entonces **|SETLIMITS** no le afecta y se pueden pintar estrellas en toda la pantalla.

### **21.1.26 |UMAP**

Este comando actualiza el mapa con información ubicada en otra zona de memoria donde tengamos un mapa mayor. EL comando hace que se reconstruya el mapa por completo, incluyendo solo aquellos ítems que cumplan unos determinados rangos de coordenadas X, Y (todos los parámetros son números de 16 bits)

Uso:

**|UMAP, <map\_ini>, <map\_fin>, <y\_ini>, <y\_fin>, <x\_ini>, <x\_fin>**

Por ejemplo, si tenemos un mapa ubicado en la dirección 22.000 que ocupa 1500bytes y queremos que se actualice el mapa con las coordenadas de nuestro personaje, con margen suficiente para avanzar en la coordenada Y hasta 100 lineas y en coordenada x hasta 20 bytes en todas direcciones:

**|UMAP, 22000, 23500, y-100, y+100, x-20, x+20**

Este comando chequeara las coordenadas de los ítems localizados en el mapa de la dirección 23000 y si se encuentran dentro de los márgenes de X, Y que hemos puesto, se copiaran en la zona de memoria que 8BP usa para el comando |MAP2SP, es decir, los copiará a partir de la dirección 42040. Eso si, tan solo copiará los que cumplan la condición. Al ser menos items, el comando |MAP2SP se ejecutará mas rápido pues tendrá que leer y comprobar si se encuentran dentro de la pantalla menos items.

**IMPORTANTE:** el mapa a copiar NO debe incluir los 3 parámetros de todo mapa:

<Max\_alto> ( que es un DW , es decir 2 bytes)

<max\_ancho> ( que es un DW , es decir 2 bytes)

<Num\_items> ( que es un DB , es decir 1 byte)

Es decir, que son 5 bytes que no se deben incluir en el mapa a copiar.

## 22 Como hacer una tabla de puntuaciones

En muchos juegos es interesante disponer de un mecanismo de tabla de puntuación (también llamada “Hall of fame”) que almacene las mejores puntuaciones de forma ordenada de distintas partidas. Se puede programar en BASIC mediante un array que almacene la puntuación de cada partida, pero cada vez que paramos el juego y hacemos RUN, el intérprete BASIC ejecuta internamente un CLEAR y se borran todos los valores que tuviese dicha tabla. Una forma de evitar esto es impedir que el usuario interrumpa el programa pulsando la tecla ESC dos veces mediante la rutina de firmware CALL &bb48. Otra opción es almacenar los puntos en una tabla en memoria y leerla y modificarla desde BASIC. Se puede programar de muchas formas, y aquí te propongo un ejemplo. Los pasos que hay que dar son:

Incluir en make\_all.asm una lectura del fichero “score\_table.asm”

```
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"
;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"
; ----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
read "make_graficos_mygame.asm"
read "score_table.asm"
```

A continuación, debemos crear dicho fichero score\_table.asm con datos de ejemplo. Yo he creado uno con nombres de diosas sumerias y puntuaciones del 10 al 1. Cada nombre ocupa 8 caracteres. Algo muy importante es el **org \_end\_graph**. Con este comando estamos indicando que la tabla se va a ensamblar después de los gráficos, en las direcciones de memoria que van a continuación.

```
org _end_graph
_SCORE_TABLE
db "ISHTAR    "
dw 10
db "ANTU      "
dw 9
db "INANNA    "
dw 8
db "NIMUG     "
dw 7
db "NIMBARA   "
dw 6
db "ASTA      "
dw 5
db "DAMKINA   "
dw 4
db "NEBAT     "
dw 3
db "NISABA    "
dw 2
db "NINSUB    "
dw 1
_END_SCORE_TABLE
```

Ahora viene la parte BASIC: Ensamblaremos con winape y después miraremos (con el winape, opción assemble->symbols) en que dirección de memoria se ha ensamblado la etiqueta end\_graph. En mi caso ha sido la &9685. En nuestro programa BASIC lo tendremos en cuenta (lo almacené en la variable "dir")

```

190 ' --- hall of fame
200 DIM pts(11): DIM name$(11):'scores
210 GOSUB 2040:'read score table
220 INK 3,7: PEN 3: LOCATE 15,12: PRINT " Hall of fame ": LOCATE 15,13: PRINT
"-----"
230 p=1:FOR i=0 TO 9:LOCATE 1,i+14: PEN p :PRINT , name$(i), pts(i) :
p=p+(p MOD 3):NEXT
240 b$=INKEY$:IF b$="" THEN 240 ELSE 250

```

Veamos la rutina que lee la tabla de score en la línea 2040

```

2040 '--- READ SCORE TABLE
2050 dir=&9685: FOR i=0 TO 9: name$(i)=""
2070 FOR j=dir TO dir +7:'lee character a character las 8 letras
2080 letra=PEEK (j): name$(i)=name$(i)+CHR$(letra)
2090 NEXT j: dir=dir+8:'tras las 8 letras estan los puntos (un entero)
2100 pts(i)=0:|PEEK,dir,@pts(i):dir=dir+2:'un entero son 2 bytes
2110 NEXT i
2120 RETURN

```

Por último, cada vez que se acaba una partida, chequeamos si la puntuación (variable "score" en el ejemplo) es superior a algún registro de la tabla de scores (array "pts") y de ser así, modificamos dicha tabla. Al modificar la tabla en las direcciones de memoria, no perderemos los valores, aunque hagamos RUN.

```

1800 '--- FIN JUEGO & CHECK HIGH SCORE ---
1810 INK 0,0:BORDER 5: INK 2,15:INK 1,20:|MUSIC
1820 j=10:FOR i=9 TO 0 STEP -1:IF score>pts(i) THEN j=i:NEXT
1830 IF j=10 THEN RUN:'end game & start
1831 'desplaza una posicion todas las puntuaciones inferiores
1840 FOR i=8 TO j STEP -1: pts(i+1)=pts(i): name$(i+1)=name$(i): NEXT
1850 b$=INKEY$:IF b$<>"" THEN 1850:'limpia buffer teclado
1860 MODE 1: BORDER 5: INK 3,8: LOCATE 6,8: PEN 3: PRINT
"CONGRATULATIONS! NEW HIGH SCORE"
1880 LOCATE 14,10: PEN 2: PRINT "ENTER YOUR NAME"
1900 LOCATE 15,12: PEN 1: INPUT name$(j): name$(j)=MID$(name$(j),1,8)
1910 pts(j)=score
1920 '--- WRITE SCORE TABLE ON MEMORY ---
1930 dir=&9685: FOR i=0 TO 9: k=1
1950 FOR j=dir TO dir +7:'escribimos carácter a carácter, las 8 letras
1960 dato$=MID$(name$(i),k,1): IF dato$="" THEN dato$=" "
1970 dato=ASC(dato$)
1980 POKE j,dato:k=k+1:NEXT j
1990 dir=dir+8: 'escribimos la puntuacion tras el nombre (8 letras)
2000 |POKE,dir,pts(i)
2010 dir=dir+2:'la puntuación es un numero entero = 2 bytes
2020 NEXT i
2030 RUN

```

## **23 Posibles mejoras futuras a la librería**

La librería 8BP es mejorable, añadiendo nuevas funciones que podrían abrir nuevas posibilidades para el programador. Aquí se muestran algunas sugerencias para hacerlo

### **23.1 Memoria para ubicar nuevas funciones**

Actualmente, mediante el mecanismo de las “**opciones de ensamblaje**”, la librería te deja una cantidad de memoria libre para tu listado BASIC que depende de cada opción

- Opción 0: 23.5 KB libres (solo debería usarse para probar cosas)
- Opción 1: 25 KB libres (juegos de laberintos)
- Opción 2: 24.8 KB libres (juegos con scroll)
- Opción 3: 24 KB libres (juegos con pseudo 3D)

La librería aún podría crecer, mediante una opción 4 que amplíe las capacidades de la opción 1, por ejemplo, mediante capacidades “filmation”. Esta opción 4 podría aportar dichas capacidades usando 1 KB y dejando al usuario 24 KB libres

### **23.2 Impresión a resolución de píxel**

Actualmente 8BP usa resolución de bytes y coordenadas de byte, que son 2 pixels de mode 0. En realidad, una forma de solventar esta limitación es mediante la definición de 2 imágenes para el mismo sprite que se encuentren desplazadas un solo pixel. Al mover dicho sprite por la pantalla puedes alternar entre simplemente cambiar la imagen por la desplazada y mover el sprite un byte. De ese modo conseguirás un movimiento pixel a pixel. Esta técnica la tienes detallada en el capítulo 13

### **23.3 Layout de mode 1**

El layout actual funciona como un buffer de caracteres de  $20 \times 25 = 500$  Bytes

Se puede usar en juegos en mode 1 sin problemas, pero habrá cosas que no podemos hacer, como definir una pieza que ocupe 3 caracteres de ancho de mode 1, ya que los caracteres de mode 0 ocupan el doble de ancho de los de mode 1. No es un problema, pero sí es una limitación.

Un layout de mode 1 ocuparía 1KB pues  $40 \times 25 = 1000$ . Puesto que el layout de mode 0 y el de mode 1 no se usarían simultáneamente, podrían solaparse en memoria y teniendo en cuenta que el de mode 0 está en 42000 hasta 42500, simplemente el de mode 1 lo situaríamos entre 41500 y 42500, “robando” 500bytes a la memoria de sprites de 8KB, situada entre 34000 y 42000

Los cambios para soportar esta mejora son mínimos, afectando solo a dos funciones **|LAYOUT** y **|COLAY** que deberían ser conscientes del modo de pantalla, mediante una variable que actuase a modo de flag (layout0/layout1). Esta modificación podría estar bien pero aun sin tenerla, podemos usar layouts en juegos de mode 1 sin problemas.

### **23.4 Capacidad filmation**

Podría ser interesante crear un modo “filmation” para hacer juegos de tipo “knight lore”. Haciendo uso de funciones ya existentes en la librería, con un poquito de código extra se podría implementar esta interesante capacidad. Es posible que pronto incorpore esta capacidad.

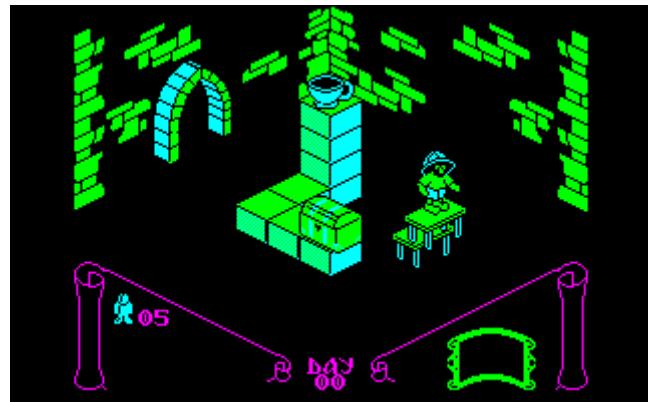


Fig. 120 el mítico “Knight Lore”

### 23.5 Funciones de scroll por hardware

Existen pocos juegos de Amstrad CPC con un scroll suave de calidad, programado usando las capacidades del chip controlador de video M6845. Actualmente la librería dispone de un mecanismo de scroll basado en CPU (no es por hardware pero es eficiente y versátil para juegos con scroll en cualquier dirección de movimiento).

El hecho de que no existan muchos juegos así responde al hecho de que en los años 80 los programadores de videojuegos no tenían mucha información y además en muchos casos eran aficionados

Entre los pocos juegos que tienen scroll suave destacan 2 de la casa Firebird:

- “Misión genocide” (de Firebird, 1987, por Paul Shirley, un excelente programador que además inventó una técnica de sobrescritura ultrarrápida sin uso de máscaras)
- “Warhawk” ( de Firebird, 1987)



Fig. 121 Juegos de Firebird con scroll rápido y suave

La técnica de scroll de estos dos juegos es la misma, conocida como “ruptura vertical”. La técnica de ruptura consiste en controlar exactamente el instante en el que se produce el barrido de pantalla. En ese momento engañamos al CTRC 6845 diciéndole que la pantalla termina antes de lo normal. Eso sí, antes de terminar esa sección de la pantalla, le decimos que incorpore menos scanlines de las que corresponden a una sección de ese tamaño. A continuación, y en un instante muy preciso que debemos controlar al microsegundo, le decimos al chip que comience una nueva pantalla, sin haberse producido la señal de sincronismo vertical. Eso nos permite dibujar una segunda zona de

pantalla (los marcadores, por ejemplo) y compensamos el número de scanlines de la primera sección. Si hacemos bien el mecanismo de compensación de número de scanlines, podemos hacer que una de las dos secciones de pantalla se mueva con extraordinaria suavidad. El problema de llevar esta técnica a un comando para ser usado desde BASIC es que el control de las interrupciones es impreciso debido a la ejecución del intérprete, y aquí necesitamos un control muy muy preciso.

El problema del scroll por hardware (que afecta también a un scroll por software que mueva toda la pantalla) es que “arrastra” los sprites presentes con él, de modo que al recolocarlos notaremos una vibración indeseada en los enemigos y/o en nuestro personaje. Para solventarlo se puede usar doble buffer y comutar entre dos bloques de 16KB cada vez que un fotograma esté listo. Eso evitará que se pueda ver “cómo se hace” cada fotograma. En 8BP el doble buffer lo descarté para poder dejar un buen espacio de RAM al programador y por eso estas técnicas no han sido implementadas.

Por todos los motivos expuestos, el scroll en 8BP se basa en un mapa del mundo que al moverse no arrastra a los sprites y por lo tanto es más eficiente pues mueve menos memoria y a la vez permite movimiento multidireccional.

### **23.6 Migrar la librería 8BP a otros microordenadores**

Esta librería sería fácilmente portable a otros microordenadores basados en el Z80, como el Sinclair ZX Spectrum. En el caso del ZX spectrum habría que rescribir las rutinas que pintan en pantalla pues la memoria de video se maneja de modo diferente. La migración a ZX ya es un proyecto en firme, tras haber recibido numerosas peticiones de usuarios de ZX.

La migración de la librería a un Commodore 64 también sería factible, aunque no se podría reutilizar el código ensamblador, ya que está basado en otro microprocesador. Además, en el caso del commodore 64, la migración de la librería 8BP debería aprovechar las características propias de la máquina como sus 8 sprites hardware, de modo que lo que debería incorporar internamente la librería 8BP sería un sprite multiplexer, ofreciendo 32 sprites, pero internamente usando los 8 sprites hardware.



*Fig. 122 Sinclair ZX y Commodore 64, dos clásicos*



## 24 Algunos juegos hechos con 8BP

En este capítulo voy a describir como están hechos algunos juegos que puedes encontrar en la web <https://github.com/jjaranda13/8BP> realizados con 8BP (de mas reciente a mas antiguo):

- **Paco el hombre:** un juego estilo comeculos, que hace uso de la técnica de movimiento suave (medio byte) y de lógicas masivas avanzadas.
- **NOMWARS:** un juego al mas puro estilo “commando”
- **Blaster pilot:** un juego de scroll multidireccional y esta inspirado en el estilo de juegos como “Time Pilot” o “Asteroids”.
- **Happy Monty:** velocísimo juego del estilo mutant monty
- **Eridu:** un juego del estilo del clásico “scramble” con scroll horizontal
- **Space phantom:** inspirado en space harrier
- **Frogger eterno:** un remake del clásico frogger, presentado en la famosa feria “amstrad eterno”
- **3D Racing one:** primer juego de carreras que usa la capacidad pseudo 3D
- **Fresh Fruits & vegetables:** un juego de plataformas que usa scroll horizontal y gestión avanzada de rutas de sprites
- **Nibiru:** un juego de naves de scroll horizontal, que usa características avanzadas de 8BP
- **Anunnaki:** un juego de naves, género arcade
- **Mutante Montoya:** un juego de pasar pantallas. Podría encuadrarse en género plataformas. Fue el primer juego que hice con 8BP.
- **Minijuegos:** son juegos didácticos, sencillos y cortos, para iniciarse en la programación con 8BP. Hay disponible una versión del clásico “pong”, llamado “Mini-pong” y una versión del clásico “Space Invaders”, llamado “mini-invaders”

### 24.1 Mutante Montoya

Un primer tributo al Amstrad CPC, con un título inspirado en el clásico "mutant monty"

Es un juego sencillo de 5 niveles. Se basa en el uso del layout de 8BP para construir cada pantalla.





## 24.2 Anunnaki, nuestro pasado alien

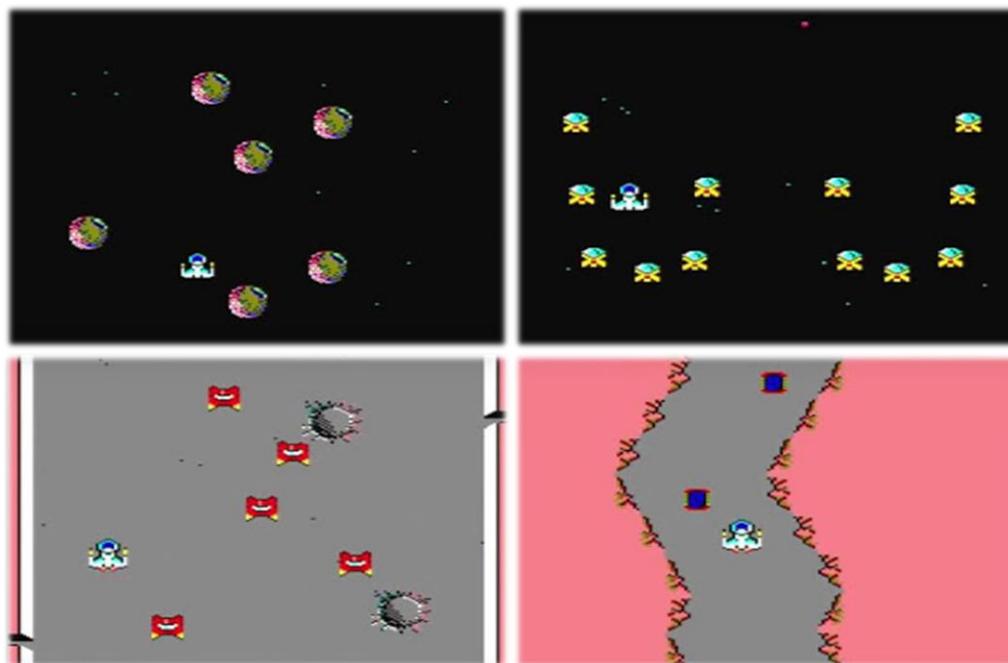
Este es un videojuego de arcade muy interesante para analizar y adentrarse en la técnica de programación de "lógicas masivas". Cuando fue programado, la librería 8BP aun no disponía de comando de scroll ni de enrutamientos de sprites, de ahí que la programación de este juego sea tan interesante, pues mediante lógicas masivas lo logra todo.

A diferencia del "Mutante Montoya", el videojuego "Anunnaki" no hace uso del layout, ya que se trata de un juego donde se trata de avanzar y destruir naves enemigas, no es un juego de laberintos ni de pasar pantallas. Este juego, además hace uso de técnicas de scroll "simulado", muy interesantes.

Eres Enki, un comandante anunnaki que se enfrenta a razas alienígenas para conquistar el planeta tierra y así someter a los humanos a su voluntad.

El juego consta de 2 niveles, aunque si pierdes una vida, continuas en el punto del nivel en que te encuentres, no vuelves al principio del nivel.

El primer nivel es una fase en el espacio interestelar, donde debes esquivar meteoritos y matar hordas de naves y pajarracos espaciales. Al final del nivel debes destruir un "jefe". El segundo nivel se desarrolla en la Luna, donde debes destruir hordas de naves, tras lo cual debes atravesar un túnel plagado de minas hasta encontrarte con tres "jefes" a los que debes destruir.



## 24.3 Nibiru

Este es un juego que pone a prueba muchas de las características de 8BP y de la técnica de programación de "lógicas masivas", y tiene detalles como un gráfico de carga lujoso y tres melodías durante el juego, así como una tabla de scores que no se pierde, aunque reinicies el juego y otros aspectos técnicos avanzados como scroll parallax, rutas, macrosecuencias, etc. El listado BASIC ocupa poco más de 16KB.

Eres el piloto de una nave destructora y debes vencer al planeta Nibiru y a su líder, "Gorgo", un reptil milenario casi invencible. Debes destruir a los pájaros galácticos que viven en sus lunas y una vez que llegues al planeta te debes enfrentar a sus peligros antes de poder luchar con Gorgo.

El Juego consiste en tres fases y utiliza el mecanismo de scroll de 8BP basado en el comando MAP2SP y también usa enrutamiento de sprites y macrosecuencias de animación. ¡Todo desde BASIC! gracias a 8BP y a la técnica de "lógicas masivas"



El juego mantiene una tabla de scores que no se borra, aunque pares el juego. Esto se consigue almacenándola en RAM con pokes, en lugar de almacenarla en variables BASIC.

## 24.4 Fresh Fruits & vegetables

Este es un juego de plataformas, donde tu misión consiste en recoger todas las frutas para dejar a la población sin nada que comer, de modo que tengan que sacrificar a un pobre cerdo para alimentarse.

Su mayor novedad es el uso avanzado de rutas, concatenando unas a otras para pasar de “caer” a “andar” y el uso de RINK combinado con MAP2SP como técnica de scroll



Los ladrillos azules de la presentación del juego usan impresión de sprites sincronizada (PRINTSPALL,0,0,1) mientras que durante el juego la impresión de sprites no es sincronizada (PRINTSPALL,0,0,0). El efecto de la sincronización en la presentación es que todo aparece sincronizado, incluida la animación por tintas RINK (usada en los ladrillos azules). Esto proporciona un efecto de scroll suave. En mi opinión, no se debe sincronizar un juego porque, aunque consigues mayor suavidad, también la velocidad del juego disminuye. Dependiendo del tipo de juego te puede interesar o no.

## 24.5 “3D Racing one”

Este es el primer juego realizado usando la capacidad pseudo3D de 8BP. Posee un lujoso gráfico de carga y 4 circuitos: uno de entrenamiento con charcos en la carretera, otro en el que competimos con otros 4 coches, otro donde la velocidad es el doble, usando segmentos de menor inclinación, y una ultima fase nocturna. El juego usa también el comando PRINTAT y un juego de caracteres propio. Además, posee tabla de scores, una música principal, dos secundarias y efectos de sonido.

Para la presentación se usa un mapa 2D lleno de bolas y se configura el comando MAP2SP con sobreescritura. Las bolas son “zoom images”.

El primer circuito se ha construido con un fichero en el que hemos ubicado uno por uno todos los elementos (segmentos, carteles, arboles, charcos...) pero el resto de los circuitos se crean dinámicamente desde BASIC, pokeando la dirección del mapa del mundo.

Para detectar las colisiones y la salida de carretera, se usa el comando PEEK en lugar de COLSPALL, de este modo en cuanto detecta un byte sobre el coche de otro color que no sea la carretera, se considera que estás colisionando y tu motor se daña.

Otra novedad interesante es el uso de rutas dinámicas. Tanto el circuito de competición como las rutas de los coches para recorrerlo se crean desde BASIC, siguiendo el procedimiento explicado en este manual.



## 24.6 Space Phantom

Este es un juego realizado con la versión v35 de la librería. Usa las capacidades pseudo-3D para la presentación de títulos al estilo “Star Wars”. Usa también impresión transparente con sprites (monedas) que pasan detrás del fondo (la tabla de scores).

El juego está inspirado en el clásico “Space Harrier” y manejas a un héroe equipado con un jet-pack que vuela por el espacio, matando meteoritos, ovnis, pájaros espaciales y un dragón como enemigo final. Se compone de tres fases y un final épico.

El juego de caracteres es propio, aunque sólo se han definido los números, al estilo “reloj casio” para los marcadores.

En la primera fase se usan rutas para las estrellas, que son sprites al igual que los meteoritos y los pájaros.

En la segunda se usa sobreescritura de sprites con fondo de 2 bit (4 colores) y animación por tintas usando RINK. Las naves en hilera se construyen ubicándolas mediante el uso del comando mejorado ROUTESP, disponible en la V35.

Aunque el juego simula 3 dimensiones, no se usa la proyección pseudo-3D, sino rutas de sprites en los que se va cambiando la versión del enemigo por una de mayor tamaño para dar sensación de que se acerca. Para que una colisión con un enemigo lejano no nos afecte,

se utiliza un flag no usado registro de estado para marcar los enemigos lejanos como inofensivos.

En la tercera fase se ha usado movimiento relativo horizontal de las piedras del suelo, en combinación con una ruta de movimiento vertical acelerado.



## 24.7 Frogger Eterno

El juego “Frogger Eterno” ha sido realizado con la versión V36 de 8BP, y su título evoca tanto el clásico “frogger” de konami lanzado en 1981 como la feria “Amstrad Eterno” celebrada en 2019, evento en el que este juego hizo su aparición.

Es un juego programado en mode 1, que usa LAYOUT, impresión transparente en la rana y rutas de diversa naturaleza para los sprites. Algunos de los sprites son invisibles, pero colisionables, tales como 4 ríos “invisibles” que se encuentran bajo los troncos, nenúfares y tortugas sobre los que la rana debe saltar, como “muros invisibles” a los laterales del río, para que la rana no pueda escapar.



## 24.8 Eridu: The space port

Este juego es un clone del clásico "Scramble" de Konami creado en 1981. Realmente tiene muchas diferencias con el original, pero en esencia esta inspirado en el juego clásico, ya que incorpora la necesidad de recargar combustible, forzando al jugador a arriesgarse para destruir los tanques de combustible y así no perder una vida.

Es bastante rápido a pesar de funcionar con un scroll potente, mostrando 32 sprites en pantalla en muchos momentos. Llega a alcanzar los 18 fps

El juego tiene 5 fases, y diferentes músicas on-game, además de un gráfico de presentación muy lujoso.

Eridu es un videojuego que conecta con una antigua historia “prohibida” de la humanidad. Eridú fue la primera ciudad del mundo, creada por los “Anunnaki” hace 400.000 años, una raza extraterrestre según las tablillas sumerias encontradas en el desierto de Irak. Allí establecieron un puerto espacial llamado “Tierra 1”.

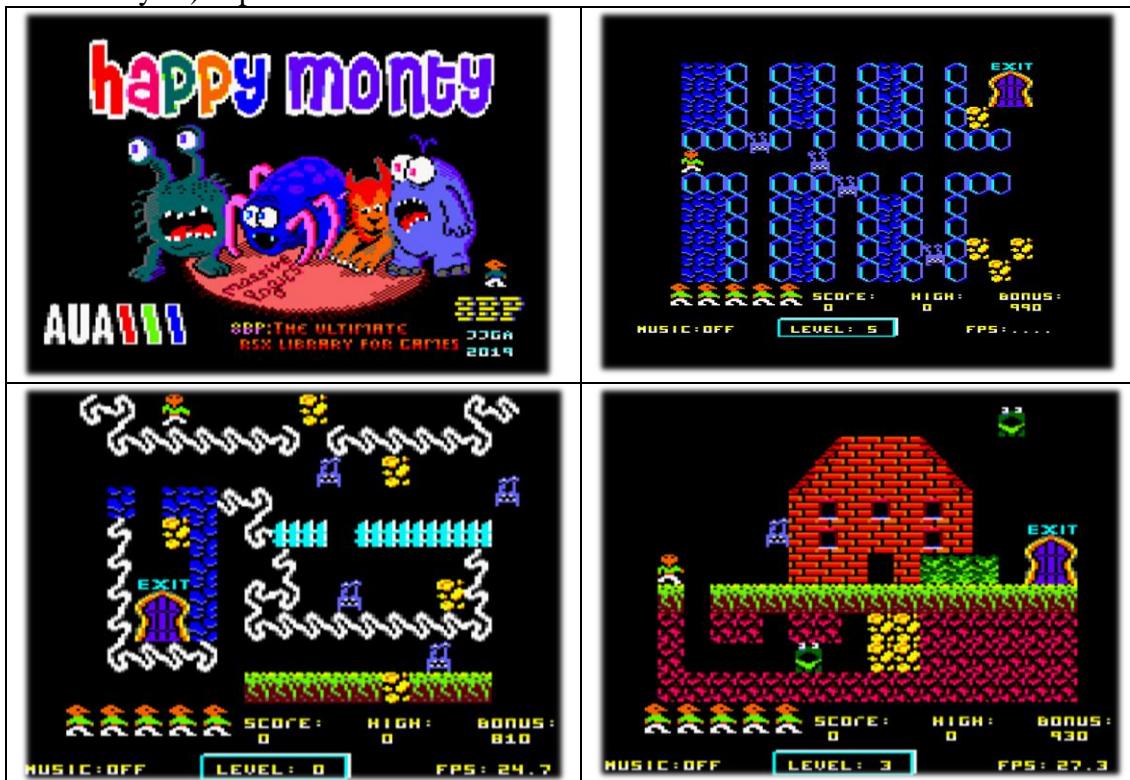


Los mapas de las diferentes fases se cargan en diferentes bancos de memoria ocupando cada uno 500 bytes de datos del mundo y 200 bytes para la descripción de la ubicación de los enemigos. El scroll lógicamente está hecho con el comando |MAP2SP

## 24.9 Happy Monty

Es un velocísimo juego de pasar pantallas, que imita casi a la perfección el clásico mutant Monty. Incluso “clona” su pantalla inicial. Este juego esta hecho con la versión 37 de la librería y alcanza los 25 FPS.

Hace un uso intensivo del layout y por supuesto, de lógicas masivas. Consigue almacenar 25 niveles mediante una sencilla técnica para compactar los layouts (cada layout gasta solo 160 bytes) explicada en este libro.



Una original técnica usada en este juego permite a los enemigos cambiar de ruta. Se trata de sprites invisibles “Inversores”. Cuando un enemigo colisiona con un sprite inversor, cambia de ruta y se le asigna la ruta contraria o incluso una ruta perpendicular, pudiendo de este modo construir trayectorias de cualquier longitud sin definir más que una ruta vertical y una horizontal. Incluso se pueden hacer loops.

Esto también simplifica la creación del mapa (layout + enemigos) de cada nivel, pues al ubicar un enemigo basta usar un código (un carácter) que indique la velocidad y dirección del enemigo (se usan 6 letras para todos los tipos de enemigo y periódicamente se cambia de “paleta” de enemigos, los muñecos asociados a esas 6 letras)

## 24.10 Blaster Pilot

Es un juego creado con la versión v39 de 8BP, en la que es posible tener efectos de sonido (creados con SOUND) a la vez que suena música mediante el comando MUSIC. En este caso el comando SOUND se usa para los disparos y explosiones y utiliza el tercer canal mientras que la música usa los dos primeros canales.



El juego posee scroll multidireccional y está inspirado en el estilo de juegos como “Time Pilot” o “Asteroids”. Junto al panel de puntuación y vidas se presenta un radar con el que puedes orientarte por el espacio para localizar a 3 actornautas perdidos a los que debes rescatar. El juego es interesante por cómo se aborda la programación del movimiento de la nave en 12 direcciones posibles del modo más eficiente posible. Además de 6 niveles, posee una fase de bonus al finalizar cada nivel, que permite acumular puntos y recuperar una vida.

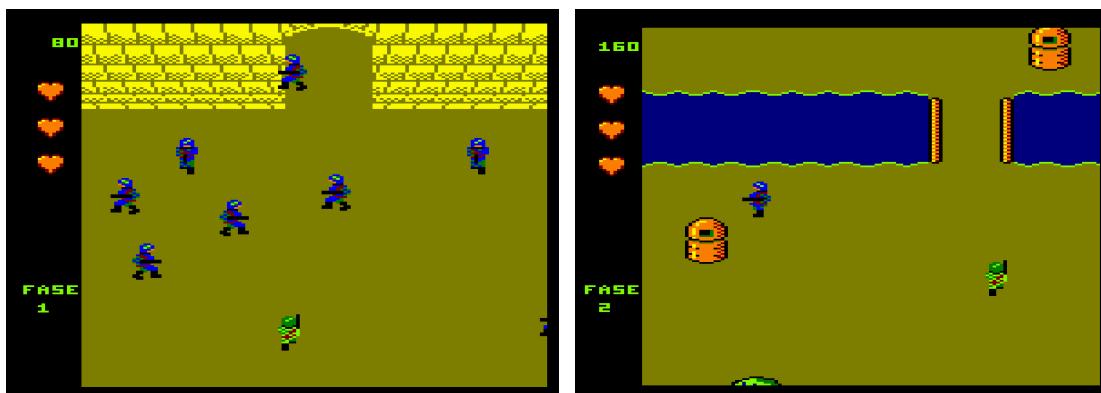
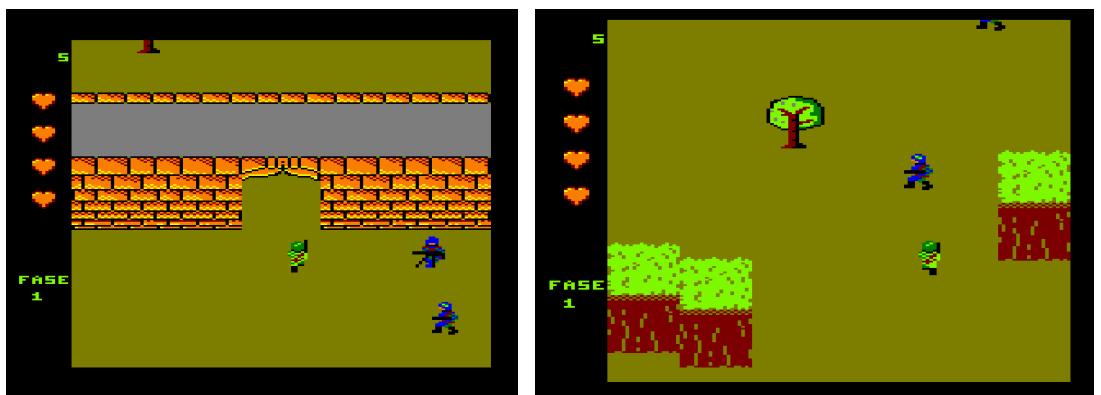
## 24.11 NOMWARS

Es un juego al mas puro estilo del clásico “comando” creado en 1985 por Capcom. Un clásico “shoot ‘em up” de scroll vertical.

El juego consta de 4 fases y de una intro “lujosa”, con una historia sobre la guerra del nuevo orden mundial. Ha sido editado en formato DES.

En esta versión se explota la capacidad de scroll de 8BP (comando MAP2SP) y se incluye el famoso puente por el que Joe pasa por debajo usando una técnica basada en el comando SETLIMITS de 8BP.

EL juego se ofrece en dos versiones: la versión de BASIC pura y la versión de ciclo compilado (ciclo traducido a lenguaje C usando el wrapper de 8BP y el minibasic de 8BP). Ambas versiones son idénticas, pues la traducción a C es una réplica total, casi un espejo de la versión BASIC. De hecho, el juego ha sido programado en BASIC y el ultimo día ha sido pasado a C con la facilidad que tiene para ello 8BP.

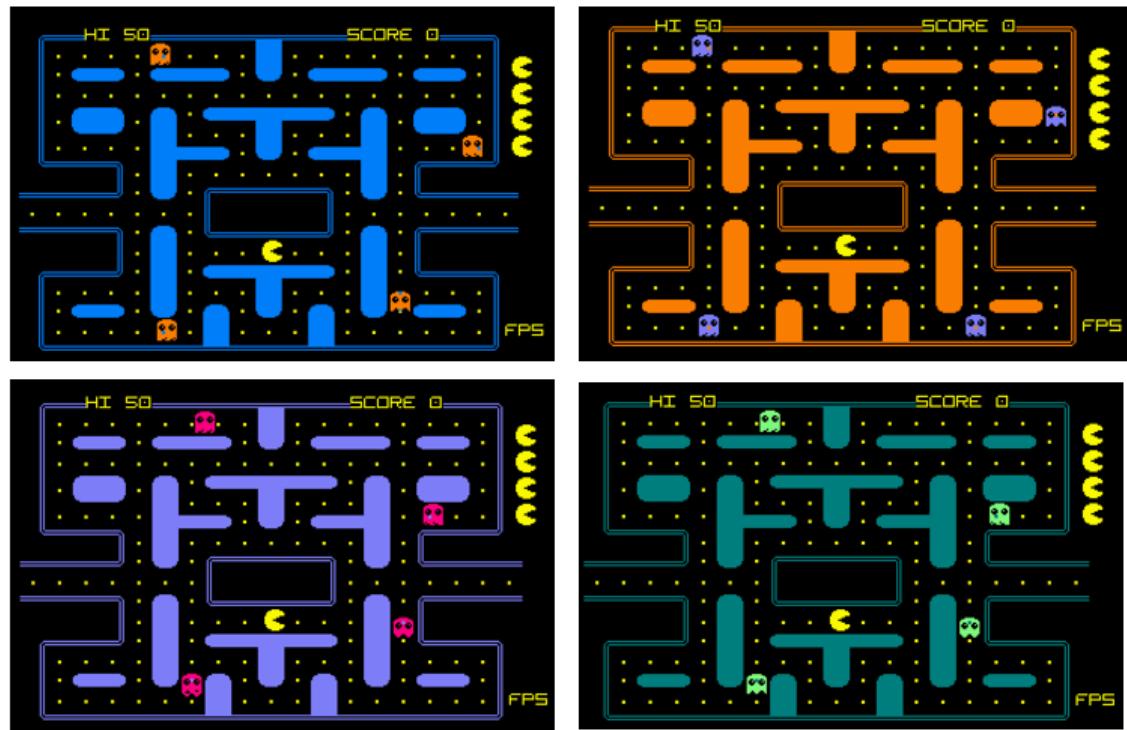


## 24.12 Paco, el hombre

Un juego al mas puro estilo comeculos. El juego contiene dos fases en BASIC y dos con ciclo compilado. En BASIC alcanza 19 FPS con laberinto, colisiones y lógicas de 4 fantasmas y en C alcanza 33 fps



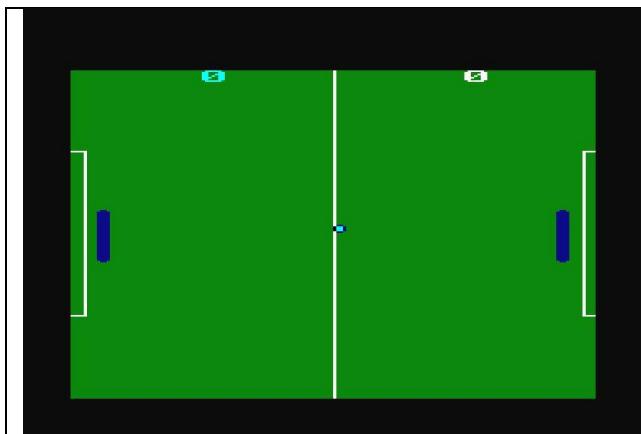
Cada nivel de Paco se identifica por sus colores. Los dos primeros corren en BASIC y para alcanzar los 19 FPS se usan decisiones de fantasmas “precalculadas”. La tercera y cuarta fase usan C y alcanzan 33 fps



## 24.13 Mini Juegos

Son juegos realizados con fines didácticos. Sencillos de entender y cortos, para ayudar en la elaboración de juegos propios al programador novel.

### 24.13.1 Mini-pong



*Fig. 123 Videojuego “mini-pong”*

Es un videojuego muy sencillo y didáctico. Basado en el clásico “Pong” de Atari (1972). La barra del oponente (el ordenador) comienza a tomar decisiones cuando la pelota sobrepasa la mitad del campo, por lo que es posible ganarle. Si le ponemos a tomar decisiones antes llega un momento que es imposible ganar.

Algunas pinceladas sobre el juego:

- Uso de **|COLSPALL**: para detectar colisiones entre el "colisionador" (la pelota) y los "colisionados" (las barras). En el byte de estado de los sprites se activa el flag de colisionador para la pelota (sprite 29) y se activa el flag de colisionado en el 31 (nuestra barra) y el 30 (barra del oponente)
- Usa sobreescritura en la pelota para respetar la raya blanca del campo y no borrarla al pasar. Para ello se usa una paleta con sobreescritura y se activa el flag de sobreescritura en el byte de estado del sprite de la pelota (el 29)
- sólo hay dos imágenes (la pelota=17 y la barra=16) que se asignan a los 2 sprites (a los sprites 30 y 31 se les asigna la imagen 16 y al sprite 29 se le asigna la 17)
- Los sprites usan movimiento automático. Para ello tienen activado el flag de movimiento automático y el comando **|AUTOALL** los mueve (les cambia sus coordenadas) de acuerdo con su velocidad.
- Todos los sprites se imprimen con **|PRINTSPALL** en cada ciclo de juego

#### 24.13.2 Mini-Invaders

Al igual que el “Mini-pong”, este es un juego realizado con fines didácticos, inspirado en el clásico “Space Invaders” de Taito (1978)



Fig. 124 Videojuego “mini-invaders”

Algunas pinceladas sobre cómo está realizado:

- El juego usa 32 sprites
- La nave es el sprite 31
- Los disparos que puedes lanzar con la nave son el 29 y el 30
- Los invaders disparan usando el sprite 28
- Los invaders usan los sprites del 0 al 27 (28 invaders en total)
- Los sprites 31,30 y 29 tienen flag de colisionador activo
- El resto de sprites son "colisionados" y tienen flag de colisionado activo
- Los invasores tienen flag de movimiento automático activo y se les asocia la ruta "0" que los mueve de derecha a izquierda y hacia abajo, lo típico de los invasores
- Los disparos de la nave y de los invaders usan una característica de la V27, recorren la pantalla y al salir se desactivan automáticamente con un cambio de estado definido al final de su ruta, simplificando de este modo la lógica de BASIC y por consiguiente acelerando el juego.

## 25 APENDICE I: Organización de la memoria de video

### 25.1 El ojo humano y la resolución del CPC

La memoria de video del Amstrad CPC tiene 3 modos de funcionamiento. El modo más utilizado para los juegos es el mode 0 (160x200) por disponer de más color, pero también se ha usado bastante el mode 1 (320x200) para programar juegos. El mode 2 (640 x 200) apenas o nunca se usó para juegos debido a sus escasos 2 colores.

Puesto que la cantidad de memoria de video es la misma, se sacrifica resolución para ganar en cantidad de colores, pero curiosamente en horizontal, que es el lado de la pantalla más largo, hay menos resolución que en vertical (160 en horizontal y 200 en vertical). Te preguntarás por qué. Además, no es el único microordenador que hacía eso, muchos otros ordenadores también usaban la misma estrategia con el lado horizontal

El motivo tiene que ver con el funcionamiento del sistema visual humano. El ojo percibe más detalles en vertical que en horizontal, de modo que “dañar” la resolución en el eje horizontal no es tan grave como dañarla en vertical. Subjetivamente es más aceptable el resultado. El sistema visual humano es como el mode 0, pixeles muy anchos. Por eso el campo de visión horizontal es mayor que el vertical

### 25.2 La memoria de video

La información más completa y clara se encuentra en el manual del firmware del Amstrad. Esta información te será útil si quieras construir un editor de sprites mejorado o quieres adentrarte en el ensamblador y programar rutinas de impresión con sobreescritura o cualquier otra cosa.

#### 25.2.1 Mode 2

En mode 2, cada píxel está representado por un bit. De modo que un byte representa a 8 pixeles. Si tomamos cualquier byte de la memoria de video, su correspondencia con los pixeles es de 1 bit por cada píxel, en esta tabla se representan los bits y a qué pixeles ( $p$ ) corresponden

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

En un byte se numera el bit 7 como el más situado a la izquierda. El píxel 0, es precisamente también el pixel situado más a la izquierda, es decir, aquí no hay nada “al revés”. Está todo correcto

#### 25.2.2 Mode 1

En mode 1 tenemos 4 colores (representados por 2 bits). Un byte representa por lo tanto a 4 pixeles. La correspondencia entre pixeles y bits es algo más compleja. El pixel 0 por ejemplo se codifica con los bits 7 y 3

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p2(0)	p3(0)	p0(1)	p1(1)	p2(1)	p3(1)

### 25.2.3 Mode 0

Aquí tenemos un pequeño follón. Cada byte representa solo dos píxeles, de los cuales la correspondencia con los bits del byte es la siguiente: El pixel 0 se codifica con los bits 7,5,3 y 1

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p0(2)	p1(2)	p0(1)	p1(1)	p0(3)	p1(3)

Con la siguiente imagen seguro que te queda más claro:

Byte (lo que se imprime en las direcciones de pantalla)

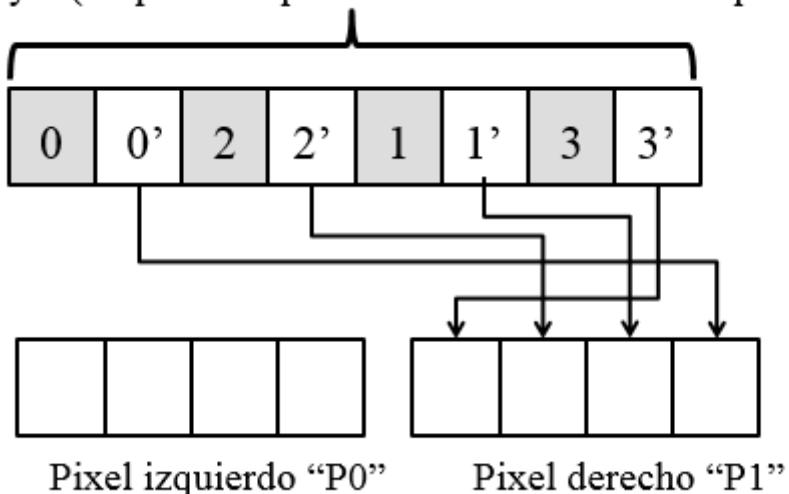


Fig. 125 pixeles y bits en mode 0

No te puedo decir cuál es el oscuro motivo para haber organizado así la memoria, pero me imagino que la causa se encuentra en el GATE ARRAY, el chip que traduce estos bits a señal de video. Imagino que el diseñador redujo circuitería con este retorcido diseño.

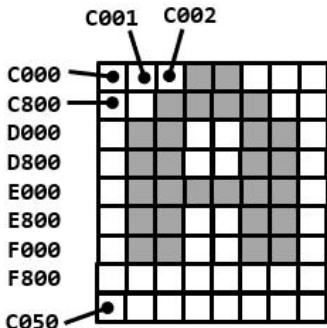
### 25.2.4 Memoria de la pantalla

Los píxeles de la pantalla pertenecientes a una misma línea se encuentran codificados en los bytes que también son contiguos. Sin embargo, de una línea a otra hay saltos.

Si avanzamos en direcciones de memoria, al llegar al final de una línea saltamos a una línea que se encuentra 8 líneas más abajo. Y si queremos continuar en la línea siguiente lo que tenemos que saltar en direcciones de memoria son 2048 posiciones.

En la siguiente tabla está representada la memoria de video. En la izquierda aparece la línea de caracteres (de 1 a 25) y para cada línea, la dirección de comienzo de cada una de las 8 scanlines que la componen (denominadas como ROW0 ...ROW7)

CHARACTER LINE	R0W0	R0W1	R0W2	R0W3	R0W4	R0W5	R0W6	R0W7
1	C000	C800	D000	D800	E000	E800	F000	F800
2	C050	C850	D050	D850	E050	E850	F050	F850
3	C0A0	C8A0	D0A0	D8A0	E0A0	E8A0	F0A0	F8A0
4	C0F0	C8F0	D0F0	D8F0	E0F0	E8F0	F0F0	F8F0
5	C140	C940	D140	D940	E140	E940	F140	F940
6	C190	C990	D190	D990	E190	E990	F190	F990
7	C1E0	C9E0	D1E0	D9E0	E1E0	E9E0	F1E0	F9E0
8	C230	CA30	D230	DA30	E230	EA30	F230	FA30
9	C280	CA80	D280	DA80	E280	EA80	F280	FA80
10	C2D0	CAD0	D2D0	DAD0	E2D0	EAD0	F2D0	FAD0
11	C320	CB20	D320	DB20	E320	EB20	F320	FB20
12	C370	CB70	D370	DB70	E370	EB70	F370	FB70
13	C3C0	CBC0	D3C0	DBC0	E3C0	EBC0	F3C0	FBC0
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CFD0	D5F0	DDF0	E5F0	ED50	F550	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	DFFF	E7FF	EFFF	F7FF	FFFF



*Direcciones de La  
esquina superior  
izquierda de La  
pantalla*

C000 = comienzo de pantalla  
= 49152 , es decir 48KB

FFFF= fin de pantalla  
= 65535

La pantalla mide:  
65535 – 49152 = 16384 =16KB

**Fig. 126 Mapa de memoria de pantalla**

La pantalla del Amstrad tiene 200 líneas x 80 bytes de ancho cada una, por lo que la memoria que se muestra en pantalla es  $200 \times 80 = 16.000$  bytes. Sin embargo, la memoria de video son 16384 bytes. Hay 384 bytes “escondidos” en 8 segmentos de 48 bytes cada uno, los cuales no se muestran en pantalla, aunque formen parte de la memoria de video. Estos 8 segmentos son lo que en la tabla anterior se llaman “spare”. Cada segmento mide 48 bytes porque como he dicho antes para saltar de una línea a la línea siguiente hay que sumar 2048 pero en realidad las 25 líneas de memoria contigua que las separan tan sólo ocupan  $25 \times 80$  bytes =2000 bytes.

Desde la &C7D0 hasta la C7FF ambos inclusive
Desde la &CFD0 hasta la CFFF ambos inclusive
Desde la &D7D0 hasta la D7FF ambos inclusive
Desde la &DFD0 hasta la DFFF ambos inclusive
Desde la &E7D0 hasta la E7FF ambos inclusive
Desde la &EF00 hasta la EFFF ambos inclusive
Desde la &F7D0 hasta la F7FF ambos inclusive
Desde la &FFD0 hasta la FFFF ambos inclusive

Puedes comprobarlo haciendo POKE sobre esas direcciones de memoria, y verás que no alterarás el contenido de la pantalla.

Resulta tentador pensar en usar estas zonas “escondidas” de la memoria para almacenar pequeñas rutinas en ensamblador o variables. Sin embargo, es peligroso porque un

comando MODE ejecutado desde BASIC borra completamente esos segmentos de memoria, por lo que si lo usas debes ser consciente de ello. En la librería 8BP estos segmentos se usan para almacenar variables locales de algunas funciones, cuyo valor puede ser borrado sin riesgos.

### 25.3 Cálculo de una dirección de pantalla

Si quieres saber la dirección de memoria a la que corresponden unas coordenadas concretas de 8BP, deberás realizar la siguiente operación

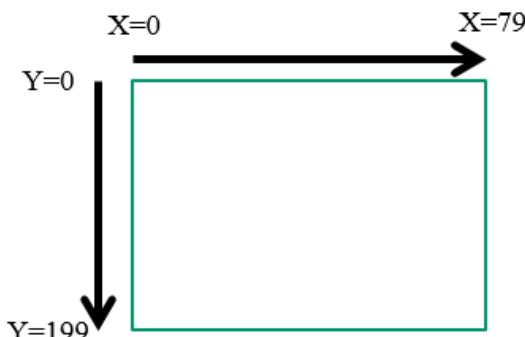
$$\text{Dir} = \&C000 + \text{INT}(y/8)*80 + (y \bmod 8)*2048 + x$$


Fig. 127 coordenadas de pantalla en 8BP

Esto es muy útil si por ejemplo quieras usar PEEK para averiguar si hay un determinado elemento o color en un byte concreto de la pantalla y utilizarlo como mecanismo de detección de colisión. Esta técnica se usa en el videojuego “3D-Racing One”.

En caso de que quieras saber la dirección de unas coordenadas gráficas (las que usa el comando PLOT de BASIC) deberás hacer primero  $y2=(200-y)/2$ ,  $x2=x/8$

$$\text{Dir} = \&C000 + \text{INT}(y2/8)*80 + (y2 \bmod 8)*2048 + x2$$

### 25.4 Barridos de pantalla

El Amstrad genera 50 imágenes por segundo. Eso significa que cada 20ms aproximadamente se debe de producir un nuevo barrido de pantalla.

Podríamos pensar que quizás el barrido, lo que es pintar la pantalla, consume una fracción de esos 20 ms pero no es así. Pintar la pantalla le lleva al Amstrad los 20ms completos, de modo que, aunque sincronices tu impresión de sprites con el barrido de pantalla es muy probable que te pille, dando lugar a dos conocidos efectos:

- **Flickering:** (parpadeo), ocurre cuando borras el sprite antes de imprimirlo en su nueva posición. Para evitarlo hay una solución muy simple: no lo borres. Simplemente haz que el sprite vaya borrando su propio rastro, dejando un borde en el sprite para que cumplan esa función. El sprite es más grande pero no parpadeará, aunque le pille el barrido a la mitad, pues no desaparece
- **Tearing:** (quiebro): ocurre cuando nos pilla el barrido a la mitad del sprite. La mitad se imprime con la nueva posición (la cabeza y el tronco) y la otra mitad no

da tiempo (las piernas). Entonces el sprite queda impreso “mal”, aunque es corregido en el siguiente fotograma, pero por un instante es como si se deformase o quebrase. El tearing es un efecto malo, pero mucho más aceptable que el flickering. La solución perfecta implica controlar en cada milisegundo donde se encuentra el barrido para conseguir imprimir cada sprite sin que nos alcance.

Una típica recomendación es imprimir los sprites de abajo a arriba, para minimizar estos efectos. De ese modo solo es posible que el barrido te pille una vez en uno de los sprites, mientras que imprimiendo de arriba abajo, te puede pillar en varios sprites el barrido porque ambos (CPU y rayo catódico) trabajan en la misma dirección. Lamentablemente lo más interesante es ordenar de arriba abajo para dar efecto de profundidad en los sprites (útil en determinados juegos tipo “Golden axe”, “Double dragon”, “Renegade”, etc.)

Los tiempos que consume la pantalla son los siguientes. Fíjate que desde que se produce la interrupción de barrido, dispones de 3,5ms para pintar sin que sea posible que te pille. Pero en ese tiempo podrás imprimir como mucho 2 sprites pequeños.

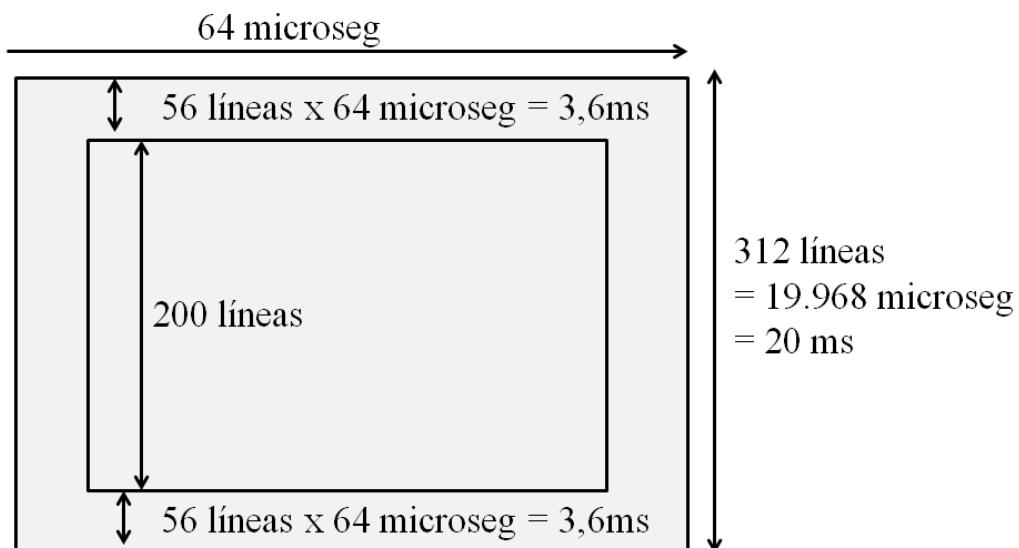


Fig. 128 tiempos en un barrido de pantalla

## 25.5 Cómo hacer una pantalla de carga para tu juego

Hay muchas formas de hacerla. Una muy sencilla es construir un gráfico con el programa SPEDIT modificado para que te permita pintar en toda la pantalla sin mostrarte menús y al final pulsar alguna tecla que te lance un comando SAVE como este

```
SAVE "mipantalla.bin", b, &C000, 16384
```

Como ves el comando salva 16KB desde la dirección de comienzo de la pantalla, que es la &C000

La forma de cargarlo sería

```
LOAD "mipantalla.bin", &C000
```

Si no usas la paleta por defecto, entonces previamente debes de ejecutar los comandos INK correspondientes a la paleta que hayas usado, antes de cargar la pantalla. Al cargar la pantalla verás poco a poco como se va dibujando en pantalla mientras carga, puesto que es ahí precisamente donde lo estas cargando, en la memoria de video.

Otra forma es generar un layout bien trabajado y salvarlo mediante el comando SAVE anterior. El caso es hacer un dibujo y como ves hay muchas formas.

Por último, puedes usar una herramienta como ConvImgCPC (también hay otras), un conversor/editor de imágenes que funciona bajo windows. Esta herramienta te permite transformar una imagen cualquiera (que puede ser un escaneo de un dibujo tuyo) en un fichero binario (con extensión .scr) apto para el CPC. Esta herramienta además te permite editar pixel a pixel y retocarla hasta que te quede perfecta. Incluso puedes editarla desde cero, sin escanear nada. En mi opinión esta es la mejor opción.

Para meter este fichero en un disco (en un fichero .dsk) debes usar el CPCDiskXP que es otra herramienta que te permite meter ficheros dentro de ficheros .dsk

Una vez dentro del .dsk puedes cargarlo con LOAD “mipantalla.bin”,&C000 Sin embargo, los colores no se verán bien porque ConvImgCPC ajusta la paleta para aproximarse lo más posible a los colores originales. Para verlos bien debes ejecutar la rutina donde ConvImg coloca la rutina de cambio de paleta, que es CALL &C7D0. Esta rutina esta “escondida” en el primero de los 8 segmentos ocultos de la memoria de video, por lo que la imagen .scr no ocupa más por contener dicha rutina. Lo malo es que hasta que no cargues la pantalla no puedes ejecutarla y por lo tanto verás como carga la imagen con colores erróneos y al final podrás invocar ese call, cambiando los colores. Lo que puedes hacer es preparar un archivo especial de paleta. Haciendo esto:

```
Load "imagen.scr", &c000
Save "paleta.bin", b, &c7d0, 48, &c7d0
```

Ahora tienes un archivo de 48 bytes que contiene la paleta. En el cargador de tu juego harías esto:

```
Load "!paleta.bin"
Call &c7d0
Load "!imagen.scr", &c000
```

Yo te recomiendo que simplemente coloques unos cuantos comandos INK antes del comando LOAD que carga la imagen

```
Load "imagen.scr", &c000
```

Y justo después, antes de usar 8BP para imprimir sprites, etc. conviene borrar el segmento oculto donde Convimg deja la rutina, ya que es un espacio que 8BP usa para variables y si no están inicialmente a cero, pueden interferir

```
for i = &c000+2000 to &c000+2000+48: poke i,0:NEXT
```

en resumen:

```
10 <varios commandos INK>
20 Load "!imagen.scr", &c000
30 for i = &c000+2000 to &c000+2000+48: poke i,0:NEXT
```

Para dejar la paleta en sus valores por defecto, usa CALL &BC02, que es una rutina del firmware.

**Y para salvar la pantalla en una cinta?**

Hemos visto como hacerlo en un disco pero CPCDiskXP no nos va a dejar el fichero .scr dentro de la cinta, de modo que debemos hacer algo como:

```
|DISC  
MEMORY 15999  
LOAD "imagen.scr", 16000  
|TAPE  
SPEED WRITE 1  
SAVE "imagen.scr",b,16000,16384
```

Y al cargarla, lo haremos igual que en disco:

```
Load "!imagen.scr", &c000
```



## 26 APENDICE II: La Paleta

En la siguiente tabla aparece la paleta del AMSTRAD. Dentro de cada color y entre paréntesis figura el número de tinta que tiene asignado ese color en la paleta por defecto. Los 27 colores son:

0 - Negro (5)	1 - Azul (0,14)	2 - Azul claro (6)	3 - Rojo	4 - Magenta	5 - Violeta	6 - Rojo claro (3)	7 - Púrpura	8 - Magenta claro (7)
9 - Verde	10 - Cyan (8)	11 - Azul cielo (15)	12 - Amarillo (9)	13 - Gris	14 - Azul pálido (10)	15 - Anaranjado	16 - Rosa (11)	17 - Magenta pálido
18 - Verde claro (12)	19 - Verde mar	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

Los valores de la paleta por defecto en cada modo son:

Modo 2:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
--------------------	---------------------------------

Modo 1:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)

Modo 0:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)	2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)
4: Blanco (paleta 26)	5: Negro (paleta 0)	6: Azul claro (paleta 2)	7: Magenta claro (paleta 8)
8: Cyan (paleta 10)	9: Amarillo (paleta 12)	10: Azul pálido (paleta 14)	11: Rosa (paleta 16)
12: Verde claro (paleta 18)	13: Verde pálido (paleta 22)	14: Parpadeo Azul/Amarillo	15: Parpadeo azul cielo/Rosa

Los valores de la paleta en cada modo se gestionan con el comando INK, consulta el manual de referencia BASIC del Amstrad para más información. Por ejemplo, para configurar la tinta cero como rojo, consultamos la paleta de los 27, vemos que el rojo es el sexto color y escribimos

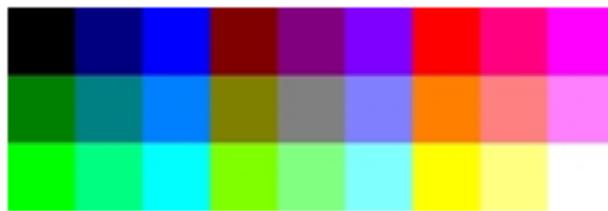
INK 0,6

Y ya tenemos configurada la tinta cero para que sea rojo. Como ves, una tinta no es un color concreto, sino que puede ser configurada para que sea el color que quieras.

El motivo por el que la paleta del Amstrad es tan buena, es porque los 27 colores ofrecen muchas posibilidades, aunque solo se puedan usar 16 a la vez. Los colores están ordenados por brillo.

Si representamos en escala RGB de 24bit (8 bit por componente) los colores del Amstrad, encontramos que dispone de 3 valores de rojo, 3 de verde y 3 de azul, (dichos valores son 0,127 y 255) siendo el número de combinaciones  $3 \times 3 \times 3 = 27$ . El color gris está justo en el centro de la paleta, donde los valores de R,G,B son iguales

(R=127,G=127,B=127). Tambien son las 3 componentes iguales en el blanco (R=255,G=255,B=255) y el negro (R=0,G=0,B=0).



*Fig. 129 Paleta de Amstrad*

El hecho de tener una paleta de 27 permite que, aunque sólo podamos escoger 16, haya siempre colores para elegir, que nos permitan crear difuminados y mezclas. Sin embargo, otros ordenadores de la época como el C64 (gran máquina) tenía 16 colores de una paleta de 16. EL C64 poseía 3 tonos de gris dentro de tan reducida paleta, lo cual, aunque ha sido criticado, opino que no es malo pues al ser colores poco saturados combinan bien con el gris, y también combinan bien entre ellos, es decir, que son 16 colores “cercaños” entre sí.

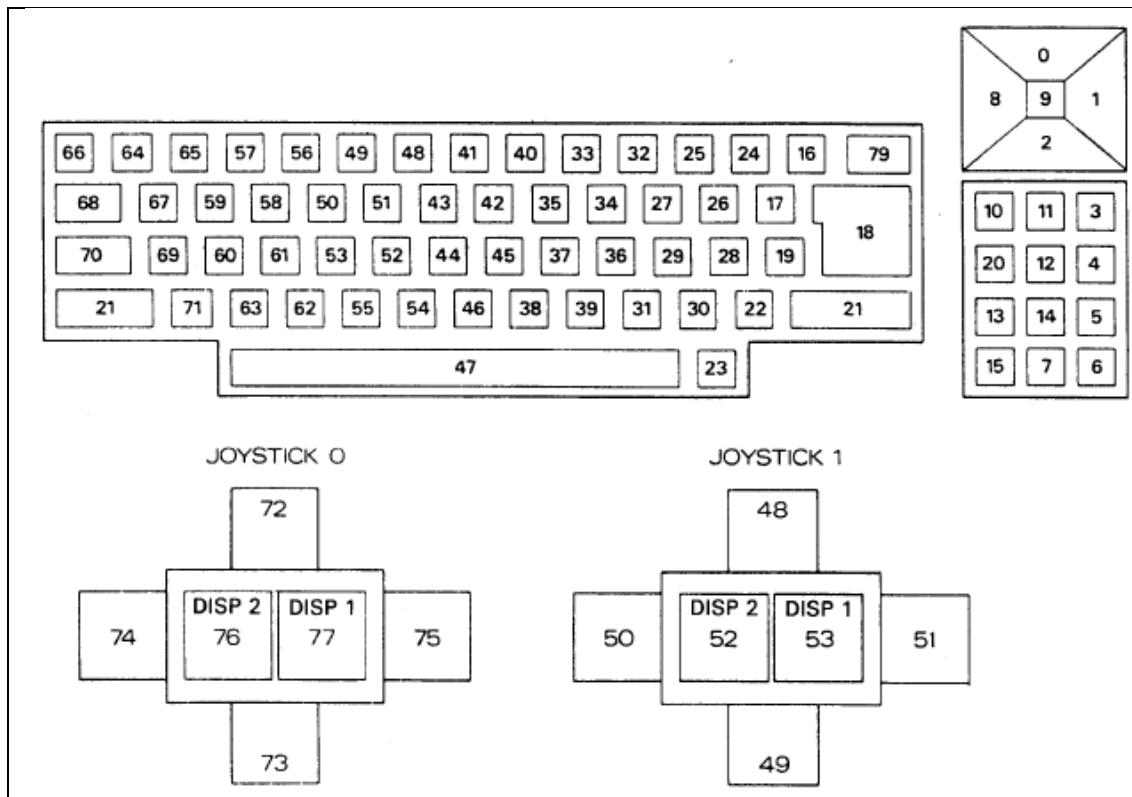


*Fig. 130 Paleta de C64*

En resumen, en Amstrad puedes elegir entre más colores y así siempre encuentras el color adecuado para difuminar, sombrear o simplemente encontrar el tono de color que necesitas. El gran acierto de Amstrad fue crear una paleta de 27 colores, aunque solo se puedan usar 16 a la vez. También puedes elegir 16 colores que no combinan bien entre sí y conseguir gráficos muy horteras (lo cual es imposible en C64, al no poder elegir).

La paleta de 27 permitió que en muchos gráficos de carga de Amstrad haya auténticas obras de arte.

## 27 APENDICE III: INKEY codes



Siempre que quieras leer el teclado procura primero vaciar el buffer de lectura de las ultimas teclas pulsadas. Es muy habitual que en una pantalla en el que le pides el nombre al usuario por haber obtenido una alta puntuación, se “cuelen” las últimas pulsaciones del juego (movimientos y disparos), mostrando cosas como “OPPPOQQAAA” en el comando INPUT. Para evitarlo, ejecuta algo como:

```
10 B$=INKEY$: if B$<>"" THEN 10
20 INPUT "nombre:";$name
```

Además de esta recomendación, recuerda que la forma más rápida de procesar el teclado es la siguiente:

```
10 IF INKEY(keycode) THEN 30: REM salta a 30 si no has pulsado
20 <instrucciones en caso de pulsar keycode>
30
```



## 29 APENDICE IV: Tabla ASCII del AMSTRAD CPC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	P	^	p	.	^	^	^	^	^	^	^	^	^
1	Γ	Φ	!	1	A	Q	a	q	■	■	■	■	■	■	■	■
2	└	Θ	"	2	B	R	b	r	■	■	■	■	■	■	■	■
3	└	Θ	#	3	C	S	c	s	■	■	■	■	■	■	■	■
4	↙	Θ	*	4	D	T	d	t	■	■	■	■	■	■	■	■
5	█	Θ	X	5	E	U	e	u	■	■	■	■	■	■	■	■
6	↙	Π	&	6	F	V	f	v	■	■	■	■	■	■	■	■
7	Ω	—	'	7	G	W	g	w	■	■	■	■	■	■	■	■
8	←	Σ	(	8	H	X	h	x	■	■	■	■	■	■	■	■
9	→	†	)	9	I	Y	i	y	■	■	■	■	■	■	■	■
A	↓	Θ	*	:	J	Z	j	z	■	■	■	■	■	■	■	■
B	↑	Θ	+	;	K	[	k	]	■	■	■	■	■	■	■	■
C	Ψ	Θ	,	<	L	\	l	l	■	■	■	■	■	■	■	■
D	←	Θ	-	=	M	]	m	]	■	■	■	■	■	■	■	■
E	Θ	Θ	.	>	N	†	n	~	■	■	■	■	■	■	■	■
F	Θ	Θ	/	?	O	_	o	█	■	■	■	■	■	■	■	■

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255



## 30 APENDICE V: algunos efectos de sonido

En primer lugar, debes saber que el Amstrad tiene 3 canales y sus identificadores son 1, 2 y 4. Para hacer que suenen dos canales o los 3 a la vez simplemente tienes que sumarlos

### Uso del comando SOUND:

SOUND canal, tono, duración, volumen, envolvente tono, envolvente volumen, ruido

**IMPORTANTE:** El volumen va de 0 a 7 en el CPC464 y de 0 a 15 en el 6128. En el CPC464 los valores 8..15 se pueden usar pero son una repetición de los valores 0..7 (es decir que 8 significa volumen 0). Es una diferencia del BASIC de ambos modelos. Como consecuencia un volumen 10 en CPC6128 es alto mientras que en 464 es muy bajo.

El parámetro de ruido va de 0 a 31

### Ejemplos:

SOUND 1,2000,10,7 : suena el canal 1

SOUND 1+2,2000,10,7 : suenan los canales 1 y 2

Ahora te propongo algunos ejemplos para usar directamente en tus programas o para inspirarte y crear otros sonidos

Recoger un diamante o una moneda ENT 1,10,-100,3: sound 1,638,30,15,0,1	te ha alcanzado una piedra o un proyectil ENT 1,10, 100,3: sound 1,638,30,15,0,1
Boing ENV 1,1,15,1,15,-1,1: SOUND 1,638,0,0,1	Boing 2 ENT 2,20,-125,1: SOUND 1,1500,10,12,,2
Muerte ENT 3,100,5,3: SOUND 1,142,100,15,0,3	
Explosión SOUND 7,1000,20,15,,15	Explosión 2 ENV 1,11,-1,25:ENT 1,9,49,5,9,-10,15 SOUND 3,145,300,12,1,1,12
Disparo ENT -5,7,10,1,7,-10,1: SOUND 1,25,20,12,,5	



## 31 APENDICE VI: Rutinas interesantes del firmware

En este apartado voy a incluir algunas rutinas del firmware que se pueden invocar desde BASIC y que pueden ser interesantes en tus programas.

**CALL 0** : produce un reset del ordenador

**CALL &bc02** : inicializa la paleta a su valor por defecto. Es una buena práctica invocarla al principio de tu programa por si se encontrase alterada

**CALL &bd19** : sincroniza con el barrido de pantalla. Si manejas muy pocos sprites puedes conseguir un movimiento más suave, pero ten en cuenta que esta instrucción va a ralentizar mucho tu programa.

**CALL &bb48** : desactiva el mecanismo de BREAK, impidiendo parar el programa si se encuentra en ejecución.

**CALL &bd21 , &bd22, &bd23, &bd24, &bd25** : produce un efecto de flash en la pantalla

Para resetar el TIMER del AMSTRAD:

En un 6128

POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0

En un 464

POKE &b187,0: POKE &b188,0: POKE &b189,0: POKE &b18a,0

Para diferenciar en que maquina esta tu programa debes desactivar la música y consultar una dirección con PEEK

```
|MUSIC: If peek(&39)=57 then modelo=464 else modelo=6128  
If modelo=464 then ...
```

**CALL &bca7** : deja de sonar cualquier sonido que estuviese sonando

El comando GRAPHICS PAPER existe en CPC6128 pero no en CPC464. Sin embargo, hay una forma de tenerlo, mediante el **CALL &BBE4** y tantos parámetros con valor 1 como color de tinta queramos, por ejemplo:

**CALL &BBE4,1,1:** igual que "GRAPHICS PAPER 2" pero vale en cpc464

**CALL &BB18** : espera a que pulses una tecla



## 32 APENDICE VII: Tabla de atributos de Sprites

La siguiente tabla contiene las direcciones de memoria en la que se encuentran almacenados los atributos de cada sprite, y la longitud en bytes de cada uno.

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tabla 7 direcciones de atributos de sprites

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Tabla 8 flags en el byte de estado



## 33 APENDICE VIII: Mapa de memoria de 8BP

```
AMSTRAD CPC464 MAPA DE MEMORIA de 8BP
;
; &FFFF +-----+
; | pantalla + 8 segmentos ocultos de 48bytes cada uno
; &C000 +-----+
; | system (simbolos redefinibles, stack pointer, etc.)
; 42619 +-----+
; | banco de 40 estrellas (desde 42540 hasta 42619 = 80bytes)
; 42540 +-----+
; | map layout de caracteres (25x20 =500 bytes)
; | y mapa del mundo (hasta 82 elementos caben en 500 bytes)
; | ambas cosas se almacenan en la misma zona de memoria
; | porque o usas una o usas otra
; 42040 +-----+
; | sprites (casi 8.5KB para dibujos).
; | dispones de 8440 bytes si no hay secuencias ni rutas)
; | aquí tambien se almacenan las imágenes del alfabeto
; +-----+
; | definiciones de rutas (de longitud variable cada una)
; +-----+
; | secuencias de animacion de 8 frames (16 bytes cada una)
; | y grupos de secuencias de animacion (macrosecuencias)
; 33600 +-----+
; | canciones
; | (1500 Bytes para musica editada con WYZtracker 2.0.1.0)
; 32100 +-----+
; | rutinas 8BP (8100 bytes o 7100 bytes)
; | aqui estan todas las rutinas y la tabla de sprites
; | incluye el player de musica "wyz" 2.0.1.0
; 25000 +-----+
;
; | TU LISTADO BASIC o C
; | 24KB, 24.8 KB o hasta 25KB libres para BASIC o C según
; | la opción de ensamblaje que uses para 8BP
;
; |
; 0 +-----+
```



## 34 APENDICE IX: comandos disponibles 8BP

Lista de comandos disponibles en orden alfabético:

3D, <flag>, #, offsety  3D, 0	Activa el modo de proyección pseudo 3D
ANIMA, #	Cambia el fotograma de un sprite según su secuencia
ANIMALL	Cambia el fotograma de los sprites con flag animación activado (no hace falta invocarla, basta con un flag en la instrucción PRINTSPALL para que sea invocada)
AUTO, #	Movimiento automático de un sprite según su Vy,Vx
AUTOALL, <flag enrutado>	Movimiento de todos los sprites con flag de movimiento automático activo
COLAY, umbral_ascii, @colision, #  COLAY, @colision, #  COLAY, #  COLAY	Detecta la colisión con el layout y retorna 1 si hay colisión. Acepta un número variable de parámetros (siempre en el mismo orden) desde 4 hasta ninguno.
COLSP, #, @collided%  COLSP, 32, ini, fin  COLSP, 33, @collided%  COLSP, 34, dy, dx  COLSP, #	Retorna primer sprite con el que colisiona #. Se puede configurar el comando con los códigos 32,33 y 34
COLSPALL, @quien%, @conquien%  COLSPALL, colisionador  COLSPALL	Retorna quien ha colisionado (collider) y con quién ha colisionado (collided)
LAYOUT, y, x, @string\$	Imprime tira de imágenes de 8x8 y rellena map layout
LOCATESP, #, y, x	Cambia las coordenadas de un sprite (sin imprimirlo)
MAP2SP, y, x  MAP2SP, status	Crea sprites para pintar el mundo en juegos con scroll. Los sprites se crean con estado = status
MOVER, #, dy, dx	movimiento relativo de un solo sprite
MOVERALL, dy,dx  MOVERALL	Movimiento relativo de todos los sprites con flag de movimiento relativo activo
MUSIC, C, flag, cancion, speed  MUSIC, flag, cancion, speed  MUSIC	Comienza a sonar una melodía. Se puede desactivar el canal C para usarlo con efectos FX si se desea Sin parámetros deja de sonar
PEEK, dir, @variable%	Lee un dato 16bit (puede ser negativo)
POKE, dir, valor	introduce un dato 16bit (que puede ser negativo)
PRINTAT, flag, y, x, @string	Imprime una cadena de "minicaracteres" redefinibles
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	imprime un solo sprite (# es su numero) sin tener en cuenta byte de status. Si se especifica 32 entonces establecemos bits fondo
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode  PRINTSPALL	Imprime todos los sprites con flag de impresion activo. Si se invoca con un solo parámetro, se establece el modo de ordenamiento
RINK,tini,color1,color2,...,colorN  RINK, salto	Rota un conjunto de tintas de acuerdo a un patrón definible compuesto por cualquier número de tintas
ROUTESP, #, pasos	Hace recorrer de golpe N pasos de la ruta del sprite
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta (no hace falta invocarla, basta con flag en AUTOALL)
SETLIMITS, xmin, xmax, ymin, ymax	Define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor  SETUPSP, #, 5, Vy, Vx	Modifica un parámetro de un sprite. Si se indica el parámetro 5, se puede suministrar opcionalmente Vx
STARS, initstar, num, color, dy, dx	Scroll de un conjunto de estrellas
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Actualiza los ítems del mapa del mundo con un subconjunto de elementos de un mapa mayor



## 35 APÉNDICE X: Opciones de ensamblaje de 8BP

Desde la versión V42, la librería 8BP dispone de varias opciones de ensamblaje que te permiten elegir las capacidades que deseas para tu juego y así dispones de más memoria disponible para el listado de tu juego

La opción de ensamblaje la debes especificar en el fichero **Make\_all\_mygame.asm**, el cual tiene una línea específica para asignar el valor del parámetro **"ASSEMBLING\_OPTION"**

Opción	Descripción de la opción	Ejemplo de juego tipo
0	<p>Puedes hacer cualquier juego Todos los comandos disponibles Necesitas usar <b>MEMORY 23499</b></p> <p>Para salvar librería + gráficos+ música: <b>SAVE "8BP0.bin",b,23500,19119</b></p>	cualquiera
1	<p>juegos de laberintos o de pasar pantallas Necesitas usar <b>MEMORY 23999</b></p> <p><b>No disponibles en este modo:</b> <b> MAP2SP,  UMAP,  3D</b></p> <p>Para salvar librería + gráficos+ música: <b>SAVE "8BP1.bin",b,25000,17619</b></p>	
2	<p>Para juegos con scroll Necesitas usar <b>MEMORY 24799</b></p> <p>No Disponibles en este modo: <b> LAYOUT,  COLAY,  3D</b></p> <p>Para salvar librería + gráficos+ música: <b>SAVE "8BP2.bin", b,24800,17819</b></p>	
3	<p>Para juegos con pseudo 3D Necesitas usar <b>MEMORY 23999</b></p> <p>No Disponibles en este modo: <b> LAYOUT,  COLAY</b></p> <p>Para salvar librería + gráficos+ música: <b>SAVE "8BP3.bin", b,24000,18619</b></p>	



## 36 APENDICE XI: correspondencias RSX/CALL

Lista de comandos disponibles en orden alfabético y su dirección asociada para usar la invocación CALL &XXXX cuando sea necesario para aumentar la velocidad. He puesto solo algún ejemplo ilustrativo pues el uso es idéntico al comando RSX, salvo que debes reemplazar el comando por CALL <dirección>.

**IMPORTANTE:** de una versión a otra de 8BP, esta lista de direcciones puede variar. Asegúrate de que estas usando la última versión de 8BP si vas a usar las direcciones que hay en esta tabla

COMANDO	DIRECCION	EJEMPLO
3D	&6BDE	
ANIMA	&6BB7	
ANIMALL	&7479	
AUTO	&6BC9	
AUTOALL	&6B9C	CALL &6B9C,1
COLAY	&6BA8	
COLSP	&6BBA	
COLSPALL	&6B99	
LAYOUT	&6BD5	
LOCATESP	&6BAE	
MAP2SP	&6BA2	
MOVER	&6BC0	
MOVERALL	&6B9F	
MUSIC	&6BD8	CALL &6BD8,0,0,0,6
PEEK	&6BB1	CALL &6BB1,dir,@var
POKE	&6BB4	
PRINTAT	&6BC6	CALL &6BC6,0,y,x,@c\$
PRINTSP	&6BC3	CALL &6BC3,31
PRINTSPALL	&6B96	CALL &62A6,0,0,0,0
RINK	&6BBD	
ROUTESP	&6BCC	
ROUTEALL	&6BD2	
SETLIMITS	&6BDB	
SETUPSP	&6BAB	
STARS	&6BA5	
UMAP	&6BCF	



## 37 APENDICE XII: Funciones 8BP en C

RSX	C prototype
3D, 0  3D, <flag>, #, offsety	void _8BP_3D_1(int flag); void _8BP_3D_3(int flag, int sp_fin, int offsety);
ANIMA, #	void _8BP_anima_1(int sp);
ANIMALL	void _8BP_animall();
AUTO, #	void _8BP_auto_1(int sp);
AUTOALL, <flag enrutado>	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, umbral_ascii, @colision, #  COLAY, @colision, #  COLAY, #  COLAY	void _8BP_colay_3(int umbral, int* colision, int sp); void _8BP_colay_2(int* colision, int sp); void _8BP_colay_1(int sp); void _8BP_colay();
COLSP, #, @collided%  COLSP, 32, ini, fin  COLSP, 33, @collided%  COLSP, #  COLSP, 34, dy, dx	/* operation 32, ini,fin or operation 34,dy,dx*/ void _8BP_colsp_3(int operation, int a, int b); /*operation 33 or sp*/ void _8BP_colsp_2(int sp, int* colision); void _8BP_colsp_1(int sp);
COLSPALL,@quien%,@conquien%  COLSPALL, colisionador  COLSPALL	void _8BP_colspall_2(int* collider, int* collided); void _8BP_colspall_1(int collider_ini); void _8BP_colspall();
LAYOUT, y, x, @string\$	void _8BP_layout_3(int y, int x, char* cad);
LOCATESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x);
MAP2SP, y, x  MAP2SP, status	void _8BP_map2sp_2(int y, int x); void _8BP_map2sp_1(unsigned char status);
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy,int dx); void _8BP_mover_1(int sp);
MOVERALL, dy,dx	void _8BP_moverall_2(int dy, int dx); void _8BP_moverall();
MUSIC, C, flag, cancion, speed  MUSIC, flag, cancion, speed  MUSIC	void _8BP_music_4(int flag_c, int flag_repetition,int song, int speed); void _8BP_music();
PEEK, dir, @variable%	void _8BP_peek_2(int address, int* dato);
POKE, dir, valor	void _8BP_poke_2(int address, int dato);
PRINTAT, flag, y, x, @string	void _8BP_printat_4(int flag,int y,int x,char* cad);
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	void _8BP_printsp_1(int sp) ; void _8BP_printsp_2(int sp, int bits_background) ; void _8BP_printsp_3(int sp,int y,int x) ;
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode  PRINTSPALL	void _8BP_printspall_4(int ini, int fin, int flag_anima, int flag_sync); void _8BP_printspall_1(int order_type); void _8BP_printspall();
RINK,tini,color1,color2,...,colorN  RINK, salto	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
ROUTESP, #, pasos	void _8BP_routesp_2(int sp, int pasos); void _8BP_routesp_1(int sp);
ROUTEALL	void _8BP_routeall();
SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits_4(int xmin, int xmax, int ymin, int ymax)
SETUPSP, #, param_number, valor  SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_3(int sp, int param, int value); void _8BP_setupsp_4(int sp, int param, int value1,int value2);
STARS, initstar, num, color, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int color, int dy, int dx); void _8BP_stars();
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin);



## 38 APENDICE XIII: MiniBASIC en C

BASIC	C prototype
BORDER	<code>void _basic_border(char color); //example _basic_border(7)</code>
CALL	<code>void _basic_call(unsigned int address); // example _basic_call(0xbd19)</code>
DRAW	<code>void _basic_draw(int x, int y);</code>
INK	<code>void _basic_ink(char ink1,char ink2);</code>
INKEY	<code>char _basic_inkey(char key); //takes around 0.3 ms. slow but simple</code>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y); // example: _basic_locate(2,25);_basic_print("TEST");</code>
MOVE	<code>void _basic_move(int x, int y);</code>
PAPER	<code>void _basic_paper(char ink);</code>
PEEK	<code>char _basic_peek(unsigned int address);</code>
GRAPHICS	<code>void _basic_pen_graph(char ink);</code>
PEN	
PEN	<code>void _basic_pen_txt(char ink);</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char data);</code>
PLOT	<code>void _basic_plot(int x, int y);</code>
PRINT	<code>void _basic_print(char *cad); //example: _basic_print("Hola \r\nAdios")</code>
RND	<code>unsigned int _basic_rnd(int max); //example: num=_basic_rnd(50)</code>
SOUND	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod);</code>
STR\$	<code>char* _basic_str(int num); //similar to STR\$ //example: _basic_print(_basic_str(num))</code>
TIME	<code>unsigned int _basic_time(); //return an unsigned int,(0..65535). As integer, when // reach 32768 go to -32768</code>